МАРТИН ГРУБЕР

Понимание SQL

Перевод Лебедева В.Н.

Под редакцией Булычева В.Н.

МОСКВА, 1993

MARTIN GRUBER

Understanding SQL

MOSKOW, 1993

Команда SELECT

```
SELECT * | { [ DISTINCT | ALL] <value expression>.,..}
   FROM { <table name> [ <alias> ] }.,..
   [ WHERE <predicate>]
   [ GROUP BY { <column name> | <integer> }.,..]
   [ HAVING <predicate>]
   [ ORDERBY { <column name> | <integer> }.,..]

   [ { UNION [ALL]

SELECT * | { [DISTINCT | ALL] < value expression >.,..}
   FROM { <table name> [<alias>]} .,..
   [ WHERE <predicate>
   [ GROUP BY { <columnname> | <integer> }.,..]
   [ HAVING <predicate>]
   [ ORDER BY { <columnname> | <integer> }.,..] } ] ...;
```

Элементы, используемые в команде SELECT

ЭЛЕМЕНТ	ОПРЕДЕЛЕНИЕ
<value expression>	Выражение, которое производит значение. Оно может включать в себя или содержать <column name>.
<table name>	Имя или синоним таблицы или представления
<alias>	Временный синоним для <table name>, определеный в этой таблице и используемый только в этой команде
<predicate>	Условие, которое может быть верным или неверным для каждой строки или комбинации строк таблицы в предложении FROM.
<column name>	Имя столбца в таблице.
<integer>	Число с десятичной точкой. В этом случае, оно показывает <value expression> в предложении SELECT с помощью идентификации его местоположения в этом предложении.

Команды UPDATE, INSERT, DELETE

UPDATE

```
UPDATE <tablename>
  SET { | }.,. .< column name> = <value expresslon> [ WHERE <predlcate>
  | WHERE CURRENT OF <cursor name>  (*только для вложения*) ];
```

INSERT

```
INSERT INTO < table name> [(<column name> .,. ]
  { VALUES ( <value expression> .,.. ) } | <query>;
```

DELETE

```
DELETE FROM <table name>
  [ WHERE <predicate>
  | WHERE CURRENT OF <cursor name>  (*только для вложения*) ];
```

Элементы, используемые в командах МОДИФИКАЦИИ

ЭЛЕМЕНТ	ОПРЕДЕЛЕНИЕ
<cursor name>	Имя курсора используемого в этой программе.
<query>	Допустимая команда SELECT.

Для других элементов смотри команду SELECT.

Символы Используемые в Синтаксисе Предлжения

СИМВОЛ	ОБЪЯСНЕНИЕ
|	Любой предшествующий знаку (|) символ может быть произвольно заменен на любой следующий за (|). Это — символический способ высказывания "или" ("or").
{ }	Все, что включено в фигурные скобки обрабатывается как модуль с целью оценки |, .,.. или других символов.
[ ]	Все,включенное в квадратные скобки является необязательным
...	Любое, предшествующее этому, может повторяться любое число раз.
.,..	Любое, предшествующее этому, и в каждом случае отделенное запятыми, может повторяться любое число раз.

Команда CREATE TABLE

```
CREATE TABLE <table name>
  ( { <column name> <data type> | <size>]
  [<colcnstrnt> ...]} .,.. );
  [<tabconstrnt>] .,.. );
```

Элементы, используемые в команде CREATE TABLE

ЭЛЕМЕНТ	ОПРЕДЕЛЕНИЕ
<table name>	Имя таблицы создаваемой этой командой.
<column name>	Имя столбца таблицы.
<data type>	Тип данных, который может содержаться в столбце. Может быть любым из следующих: INTEGER (ЦЕЛОЕ ЧИСЛО), CHARACTER (СИМВОЛЬНОЕ), DECIMAL (ДЕСЯТИЧНОЕ), NUMERIC (ЧИСЛОВОЕ), SMALLINT (НАИМЕНЬШЕЕ) FLOAT (С ПЛАВАЮЩЕЙ ТОЧКОЙ) REAL (РЕАЛЬНОЕ), DOUBLE PRECISION (УДВОЕННАЯ ТОЧНОСТЬ С ПЛАВАЮЩЕЙ ТОЧКОЙ), LONG * (ДЛИННОЕ *), VARCHAR * (ПЕРЕМЕННОЕ СИМВОЛЬНОЕ *), DATE * (ДАТА *), TIME * (ВРЕМЯ *) (* — указывает на нестандартный для SQL тип данных)
<size>	Размер. Его значение зависит от <data type>.
<colconstrnt>	Может быть любым из следующих: NOT NULL (НЕ НУЛЕВОЙ), UNIQUE (УНИКАЛЬНЫЙ), PRIMARY KEY (ПЕРВИЧНЫЙ КЛЮЧ), CHECK(<predicate>) (ПРОВЕРКА предиката), DEFAULT = <value expression> (ПО УМОЛЧАНИЮ = значимому выражению) REFERENCES <table name> [(<column name> .,..)] (ССЫЛКА НА имя таблицы [(имя столбца)])
<tabconstrnt>	Может быть любым из следующих: UNIQUE (УНИКАЛЬНЫЙ), PRIMARY KEY (ПЕРВИЧНЫЙ КЛЮЧ), CHECK (ПРОВЕРКА предиката) FOREIGN KEY(<column name>) (ВНЕШНИЙ КЛЮЧ) REFERENCES <table name> [(<column name> .,..)] (ССЫЛКА НА имя таблицы [(имя столбца)].

ПРЕДИСЛОВИЕ

"ПОНИМАНИЕ SQL" — это полный учебник по программированию на Структурированном Языке Запросов, написанный специально для тех, кто будет использовать SQL в процессе работы. Даже если это ваш первый опыт с компьютерами или управлением базами данных, книга *"ПОНИМАНИЕ SQL"* очень быстро научит вас свободно работать с реальной SQL, использованию простых запросов, а также снабдит вас ясными понятиями об автоматизированном управлении базой данных. Книга даст вам краткое, удобное в чтении введение в реляционные базы данных. Предоставит вам обучающие программы, чтобы, овладевая командами SQL шаг за шагом, помочь вам узнать, как извлекать и обрабатывать информацию, содержащуюся в таблицах данных, т.е.:

* выбирать информацию, с которой вы хотите работать
* добавлять, удалять, и модифицировать информацию в таблице данных
* использовать и-или, верно/неверно и другие условия для обнуления определенной информации
* использовать специальные функции SQL для суммирования ваших данных.

Книга покажет Вам, как эффективно работать с многочисленными таблицами данных, используя улучшеную технику для запроса более чем одной таблицы одновременно, строить комплекс запросов и подзапросов, и использовать представления, чтобы создавать базы данных и работать с базами данных раздельно с многими таблицами.

Научит создавать новые таблицы данных для пользовательских деловых прикладных программ. Вы исследуете принципы эффективного проектирования базы данных, а также техники для обеспечения целостности данных и их защиты.

Вы узнаете, как использовать SQL с другими языками в специальной главе SQL для программистов.

"ПОНИМАНИЕ SQL" — необходима и пригодна для любой реализации Структурированного Языка Запроса. Книга включает и краткий справочный стандарт SQL и руководство к общим нестандартным особенностям SQL.

Об Авторе

Мартин Грубер — свободный писатель, учитель и консультант из Сан Франциско. В дополнении к написанию и редактированию книг, руководств пользователей и документации, он работает в широком спектре интересов, связаных с компьютерами и компьютерными базами данных.

1

ВВЕДЕНИЕ В РЕЛЯЦИОННУЮ БАЗУ ДАННЫХ

ВВЕДЕНИЕ

SQL (ОБЫЧНО ПРОИЗНОСИМАЯ КАК "СЭКВЭЛ") символизирует собой *Структурированный Язык Запросов.* Это — язык который дает вам возможность создавать и работать в реляционных базах данных, которые являются наборами связанной информации, сохраняемой в таблицах.

Мир баз данных становится все более и более единым, что привело к необходимости создания стандартного языка, который мог бы использоваться чтобы функционировать в большом количестве различных видов компьютерных сред. Стандартный язык позволит пользователям знающим один набор команд, использовать их чтобы создавать, отыскивать, изменять, и передавать информацию независимо от того работают ли они на персональном компьютере, сетевой рабочей станции, или на универсальной ЭВМ. В нашем все более и более взаимосвязанном компьютерном мире, пользователь снабженый таким языком, имеет огромное преимущество в использовании и обобщении информации из ряда источников с помощью большого количества способов.

Элегантность и независимость от специфики компьютерных технологий, а также его поддержка лидерами промышленности в области технологии реляционных баз данных, сделало SQL, и вероятно в течение обозримого будущего оставит его, основным стандартным языком. По этой причине, любой кто хочет работать с базами данных 90-х годов должен знать SQL.

Стандарт SQL определяется **ANSI** (*Американским Национальным Институтом Стандартов*) и в данное время также принимается **ISO** (*МЕЖДУНАРОДНОЙ ОРГАНИЗАЦИЕЙ ПО СТАНДАРТИЗАЦИИ*). Однако, большинство коммерческих программ баз данных расширяют SQL без уведомления ANSI, добавляя разные другие особенности в этот язык, которые, как они считают, будут весьма полезны. Иногда они несколько нарушают стандарт языка, хотя хорошие идеи имеют тенденцию развиваться и вскоре становиться стандартами "*рынка*" сами по себе в силу полезности своих качеств. В этой книге, мы будем, в основном, следовать стандарту ANSI, но одновременно иногда будет показывать и некоторые наиболее общие отклонения от его стандарта. Вы должны проконсультироваться с документацией вашего пакета программ который вы будете использовать, чтобы знать где в нем этот стандарт видоизменен.

ПРЕЖДЕ, ЧЕМ ВЫ СМОЖЕТЕ ИСПОЛЬЗОВАТЬ SQL, ВЫ должны понять что такое реляционные базы данных. В этой главе мы это объясним и покажем, насколько реляционные базы данных полезны. Мы не будем обсуждать SQL именно здесь, и если вы уже знаете эти понятия довольно хорошо, вы можете просто пропустить эту главу. В любом случае, вы должны рассмотреть три таблицы которые предоставляются и объясняются в конце главы; они станут основой наших примеров в этой книге. Вторая копия этих таблиц находится Приложении E, и мы рекомендуем скопировать их для удобства ссылки к ним.

ЧТО ТАКОЕ — РЕЛЯЦИОННАЯ БАЗА ДАННЫХ?

Реляционная база данных — это тело связанной информации, сохраняемой в двумерных таблицах. Напоминает адресную или телефонную книгу. В книге имеется большое количество входов, каждый из которых соответствует определеной особенности. Для каждой такой особенности, может быть несколько независимых фрагментов данных, например имя, телефонный номер, и адрес. Предположим, что вы должны сформатировать эту адресную книгу в виде таблицы со строками и столбцами. Каждая строка (называемая также записью) будет соответствовать определенной особенности; каждый столбец будет содержать значение для каждого типа данных —

имени, телефонного номера, и адреса представляемого в каждой строке. Адресная книга могла бы выглядеть следующим образом:

Имя	Телефон	Адрес
Gerry Farish	*(415)365-8775*	*127 Primrose Ave.,SF*
Celia Brock	*(707)874-3553*	*246 #3rd St., Sonoma*
Yves Grillet	*(762)976-3665*	*778 Modernas, Barcelona*

То что вы получили является основой реляционной базы данных как и было определено в начале этого обсуждения — а именно, двумерной (строка и столбец) таблицей информации. Однако, реляционные базы данных редко состоят из одной таблицы. Такая таблица меньше чем файловая система. Создав несколько таблиц взаимосвязанной информации, вы сможете выполнить более сложные и мощные операции с вашими данными. Мощность базы данных зависит от связи которую вы можете создать между фрагментами информации, а не от самого фрагмента информации.

СВЯЗЫВАНИЕ ОДНОЙ ТАБЛИЦЫ С ДРУГОЙ

Позвольте нам использовать пример нашей адресной книги, чтобы начать обсуждение базы данных, которая может реально использоваться в деловой ситуации. Предположим, что персонажи в нашей первой таблице (адресной книги) — это пациенты больницы. В другой таблице, мы могли бы запомнить дополнительную информацию об этих пациентах. Столбцы второй таблицы могли бы быть помечены как Пациент, Доктор, Страховка, и Баланс.

Пациент	Доктор	Страховка	Баланс
Farish	*Drume*	*B.C./B.S.*	*$272.99*
Grillet	*Halben*	*None*	*$44.76*
Brock	*Halben*	*Health,Inc.*	*$9077.47*

Много мощных функций можно выполнить извлекая информацию из этих таблиц согласно указанным параметрам, особенно когда эти параметры включают в себя фрагменты информации свзанные в различных таблицах друг с другом. Например, возьмем — докторов. Предположим доктор Halben захотел получить номера телефонов всех своих пациентов. Чтобы извлечь эту информацию, он мог бы связать таблицу с номерами телефонов пациентов (по адресной книге) с таблицей которая бы указывала, какой из пациентов — его. Хотя, в этом простом примере, он мог бы держать это в голове и сразу получать номера телефонов пациентов Grillet и Brock, эти таблицы могут быть слишком большими и слишком сложными. Программы реляционной базы данных разрабатывались для того чтобы обрабатывать большие и сложные совокупности данных такого типа, что очевидно является более универсальным методом в деловом мире. Даже если бы база данных больницы содержала сотни или тысячи имен — как это вероятно и бывает на практике — одна команда SQL могла бы выдать доктору Halben информацию в которой он нуждался почти немедленно.

ПОРЯДОК СТРОК ПРОИЗВОЛЕН

Чтобы поддерживать максимальную гибкость, строки таблицы, по определению, не должны находиться ни в каком определенном порядке. С этой точки зрения, в этом структура базы данных отличается от нашей адресной книги. Вход в адресную книгу обычно упорядочивается в алфавитном порядке. В системах с реляционной базой

данных, имеется одна мощная возможность для пользователей — это способность упорядочивать информацию так чтобы они могли восстанавливать ее.

Рассмотрим вторую таблицу. Иногда Вам необходимо видеть эту информацию упорядоченной в алфавитном порядке по именам, иногда в возрастающем или убывающем порядке, а иногда сгруппированной по отношению к какому-нибудь доктору. Наложение порядка набора в строках будет сталкиваться со способностью заказчика изменять его, поэтому строки всегда рассматриваются как неупорядоченные. По этой причине, вы не можете просто сказать: "*Мы хотим посмотреть пятую строку таблицы.*" Пренебрегая порядком в котором данные вводились или любым другим критерием, мы определим, не ту строку, хотя она и будет пятой. Строки таблицы которые рассматриваются, не будут в какой-либо определенной последовательности.

ИДЕНТИФИКАЦИЯ СТРОК (ПЕРВИЧНЫЕ КЛЮЧИ)

По этим и другим причинам, вы должны иметь столбец в вашей таблице который бы уникально идентифицировал каждую строку. Обычно, этот столбец содержит номер — например, номер пациента назначаемый каждому пациенту. Конечно, вы могли бы использовать имя пациентов, но возможно что имеется несколько Mary Smiths; и в этом случае, вы не будете иметь другого способа чтобы отличить этих пациентов друг от друга. Вот почему номера так необходимы. Такой уникальный столбец (или уникальная группа столбцов), используемый чтобы идентифицировать каждую строку и хранить все строки отдельно, называются — *первичными ключами таблицы.* Первичные ключи таблицы важный элемент в структуре базы данных. Они — основа вашей системы записи в файл; и когда вы хотите найти определенную строку в таблице, вы ссылаетесь к этому первичному ключу. Кроме того, первичные ключи гарантируют, что ваши данные имеют определенную целостность. Если первичный ключ правильно используется и поддерживается, вы будете знать что нет пустых строк таблицы и что каждая строка отличается от любой другой строки. Мы будем обсуждать ключи и далее когда поговорим относительно справочной целостности в Главе 19.

СТОЛБЦЫ ИМЕНУЮТСЯ И НУМЕРУЮТСЯ

В отличие от строк, столбцы таблицы (также называемые *полями*) упорядочиваются и именуются. Таким образом, в нашей таблице адресной книги, возможно указать на "адрес столбца" или на "столбец 3". Конечно, это означает что каждый столбец данной таблицы должен иметь уникальное имя чтобы избежать неоднозначности. Лучше всего если эти имена указывают на содержание поля. В типовых таблицах этой книги, мы будем использовать такие сокращения для имени столбца, как ***cname*** для имени заказчика, и ***odate*** для даты порядка. Мы также дадим каждой таблице личный числовой номер столбца в качестве первичного ключа. Следующий раздел будет объяснять эти таблицы и их ключи более подробно.

ТИПОВАЯ БАЗА ДАННЫХ

Таблицы 1.1, 1.2, и 1.3 составляют реляционную базу данных которая является минимально достаточной чтобы легко ее отслеживать, и достаточно полной, чтобы иллюстрировать главные понятия и практику использования SQL. Эти таблицы напечатаны в этой главе а также в Приложении E. Так как они будут использоваться для иллюстрирования различных особенностей SQL по всей этой книге, мы рекомендуем чтобы вы скопировали их, для удобства ссылки к ним.

Вы могли уже обратить внимание что первый столбец каждой таблицы содержит номера чьи значения различны для каждой строки. Как вы наверное и предположили, это — первичные ключи таблиц. Некоторые из этих номеров также показаны в столбцах других таблиц. В этом нет ничего неверного. Они поазывают связь между строками которые используют значение принимаемое из первичного ключа, и строками где это значение используется в самом первичном ключе.

Таблица 1.1: **Продавцы**

SNUM	SNAME	CITY	COMM
1001	*Peel*	*London*	*.12*
1002	*Serres*	*San Jose*	*.13*
1004	*Motika*	*London*	*.11*
1007	*Rifkin*	*Barcelona*	*.15*
1003	*Axelrod*	*New York*	*.10*

Таблица 1.2: **Заказчики**

CNUM	CNAME	CITY	RATING	SNUM
2001	*Hoffman*	*London*	*100*	*1001*
2002	*Giovanni*	*Rome*	*200*	*1003*
2003	*Liu*	*SanJose*	*200*	*1002*
2004	*Grass*	*Berlin*	*300*	*1002*
2006	*Clemens*	*London*	*100*	*1001*
2008	*Cisneros*	*SanJose*	*300*	*1007*
2007	*Pereira*	*Rome*	*100*	*1004*

Таблица 1.3: **Порядки**

ONUM	AMT	ODATE	CNUM	SNUM
3001	*18.69*	*10/03/1990*	*2008*	*1007*
3003	*767.19*	*10/03/1990*	*2001*	*1001*
3002	*1900.10*	*10/03/1990*	*2007*	*1004*
3005	*5160.45*	*10/03/1990*	*2003*	*1002*
3006	*1098.16*	*10/03/1990*	*2008*	*1007*
3009	*1713.23*	*10/04/1990*	*2002*	*1003*
3007	*75.75*	*10/04/1990*	*2004*	*1002*
3008	*4723.00*	*10/05/1990*	*2006*	*1001*
3010	*1309.95*	*10/06/1990*	*2004*	*1002*
3011	*9891.88*	*10/06/1990*	*2006*	*1001*

Например, поле *snum* в таблице Заказчиков указывает, какому продавцу назначен данный заказчик. Номер поля *snum* связан с таблицей Продавцов, которая дает информацию об этих продавцах. Очевидно, что продавец которому назначены заказчики должен уже существовать — то есть, значение snum из таблицы Заказчиков должно также быть представлено в таблице Продавцов. Если это так, то говорят, что ***"система находится в состоянии справочной целостности"***. Этот вывод будет более полно и формально объяснен в Главе 19.

ПРИМЕЧАНИЕ: Эти три представленых таблицы в тексте имеют русские имена — Продавцов, Заказчиков и Порядков, и далее будут упоминаться именно под этими именами. Имена

любых других применяемых в книге таблиц будут написаны по английски чтобы отличать их от наших базовых таблиц этой базы данных. Кроме того в целях однозначности, имена заказчиков, продавцов, Системных Каталогов а также полей в тексте, также будут даны на латыни.

Таблицы приведены как пример к похожей ситуации в реальной жизни, когда вы будете использовать SQL чтобы следить за продавцами, их заказчиками, и порядками заказчиков. Давайте рассмотрим эти три таблицы и значения их полей.

Здесь показаны столбцы Таблицы 1.1

ПОЛЕ	СОДЕРЖАНИЕ
snum	уникальный номер, назначенный каждому продавцу ("номер служащего")
sname	имя продавца
city	расположение продавца (город)
comm	комиссионные продавцов в десятичной форме

Таблица 1.2 содержит следующие столбцы:

ПОЛЕ	СОДЕРЖАНИЕ
cnum	Уникальный номер назначенный каждому заказчику
cname	Имя заказчика
city	Расположение заказчика (город)
rating	Код, указывающий уровень предпочтения данного заказчика перед другими. Более высокий номер указывают на большее предпочтение (рейтинг).
snum	Номер продавца, назначенного этому заказчику (из таблицы Продавцов)

И имеются столбцы в Таблице 1.3:

ПОЛЕ	СОДЕРЖАНИЕ
onum	уникальный номер данный каждому приобретению
amt	значение суммы приобретений
odate	дата приобретения
cnum	номер заказчика делающего приобретение (из таблицы Заказчиков)
snum	номер продавца продающего приобретение (из таблицы Продавцов)

РЕЗЮМЕ

Теперь вы знаете что такое реляционная база данных, понятие, которое звучит сложнее чем есть на самом деле. Вы также изучили некоторые фундаментальные принципы относительно того, как сделаны таблицы — как работают строки и столбцы, как первичные ключи отличают строки друга друга, и как столбцы могут ссылаться к значениям в других столбцах. Вы поняли что запись это синоним строки, и что поле это синоним столбца. Оба термина встречаются в обсуждении SQL, и мы будем использовать их в равной степени в этой книге.

Вы теперь знакомы с таблицами примеров. Краткие и простые, они способны показать большинство особенностей языка, как вы это увидите В некоторых случаях, мы

будем использовать другие таблицы или постулаты некоторых различных данных в одной из этих таблиц чтобы показать вам некоторые другие возможности.

Теперь вы готовы к углублению в SQL самостоятельно. Следующая глава даст вам быстрый просмотр языка, и даст вам информацию, которая поможет Вам ссылаться к уже пройденным местам.

РАБОТА С SQL

1. Какое поле таблицы Заказчиков является первичным ключом?
2. Что является столбцом 4 из таблицы Заказчиков?
3. Как по другому называется строка? Столбец?
4. Почему вы не можете запрашивать для просмотра первые пять строк таблицы?

(См. Приложение А для ответов.)

2

SQL: ОБЗОР

ЭТА ГЛАВА ПОЗАКОМИТ ВАС СО СТРУКТУРОЙ SQL языка, а также с определенными общими выводами, такими как тип данных, которые эти поля могут содержать, и некоторые области неоднозначностей, которые существуют в SQL. Она предназначена обеспечить связь с более конкретной информацией в последующих главах. Вы не должны запоминать каждую подробность, упомянутую в этой главе. Краткий обзор представлен здесь в одной удобно размещеной области, многие подробности которой вы можете иметь чтобы в последствии ссылаться к ним по мере овладения языком. Мы поместили все это в начало книги чтобы ориентировать вас на мир SQL без упрощенного подхода к его проблемам и в то же время дать вам привычные в будущем места для ссылки к ним когда у Вас появятся вопросы. Этот материал может стать более понятным когда мы перейдем к описанию конкретных команд SQL, начинающихся с Главы 3.

КАК РАБОТАЕТ SQL?

SQL — это язык, ориентированный специально на реляционные базы данных. Он устраняет много работы, которую вы должны были бы сделать если бы вы использовали универсальный язык программирования, напрмер **C**. Чтобы сформировать реляционную базу данных на **C**, вам необходимо было бы начать с самого начала. Вы должны были бы определить объект, называемый ***таблицей***, которая могла бы расти, чтобы иметь любое число строк, а затем создавать постепенно процедуры для помещения значений в нее и извлечения из них. Если бы вы захотели найти некоторые определенные строки, вам необходимо было бы выполнить по шагам процедуру, подобную следующей :

1. Рассмотрите строку таблицы.
2. Выполните проверку — является ли эта строка одной из строк, которые вам нужны.
3. Если это так, сохраните ее где-нибудь, пока вся таблица не будет проверена.
4. Проверьте, имеются ли другие строки в таблице.
5. Если имеются, возвратитесь на шаг 1.
6. Если строк больше нет, вывести все значения, сохраненные в шаге 3.

(Конечно, это не фактический набор **C** команд, а только логика шагов, которые должны были бы быть включены в реальную программу.)

SQL сэкономит вам все это. Команды в SQL могут работать со всеми группами таблиц как с единым объектом и могут обрабатывать любое количество информации, извлеченной или полученной из их, в виде единого модуля.

ЧТО ДЕЛАЕТ ANSI ?

Как мы уже рассказывали в Введении, стандарт SQL определяется с помощью кода ANSI (*Американский Национальный Институт Стандартов*). SQL не изобретался ANSI. Это по существу изобретение IBM. Но другие компании подхватили SQL сразу же, по крайней мере одна компания (Oracle) отбила у IBM право на рыночную продажу SQL продуктов.

После того, как появился ряд конкурирующих программ SQL на рынке, ANSI определил стандарт, к которому они должны быть приведены (определение таких стандартов и является функцией ANSI). Однако после этого появились некоторые проблемы. Возникли они в результате стандартизации ANSI ввиде некоторых ограничений. Так как не всегда ANSI определяет то, что является наиболее полезным, то программы пытаются соответствовать стандарту ANSI, не позволяя ему ограничивать их слишком сильно. Это, в свою очередь, ведет к случайным несогласованностям.

Программы Баз Данных обычно дают ANSI SQL дополнительные особенности и часто ослабляют многие ограничения из большинства из них. Следовательно, общие разновидности ANSI будут также рассмотрены. Хотя мы очевидно не сможем объять каждое исключение или разновидность, удачные идеи имеют тенденцию к внедрению и использованию в различных программах даже когда они не определены стандартом ANSI. ANSI — это вид минимального стандарта и вы можете делать больше чем он позволяет, хотя и должны выполнять его указания при выполнении задач которые он определяет.

ИНТЕРАКТИВНЫЙ И ВЛОЖЕННЫЙ SQL

Имеются два SQL: **Интерактивный** и **Вложенный**. Большей частью, обе формы работают одинаково, но используются различно.

Интерактивный SQL используется для функционирования непосредственно в базе данных, чтобы производить вывод для использования его заказчиком. В этой форме SQL, когда вы введете команду, она сейчас же выполнится и вы сможете увидеть вывод (если он вообще получится) — немедленно.

Вложенный SQL состоит из команд SQL, помещенных внутри программ, которые обычно написаны на некотором другом языке (типа КОБОЛА или Паскаля). Это делает эти программы более мощными и эффективным. Однако, допуская эти языки, приходится иметь дело с структурой SQL и стилем управления данных который требует некоторых расширений к интерактивному SQL. Передача SQL команд во вложенный SQL является выдаваемой ("*passed off*") для переменных или параметров используемых программой в которую они были вложены.

В этой книге, мы будем представлять SQL в интерактивной форме. Это даст нам возможность обсуждать команды и их эффекты, не заботясь о том, как они связаны с помощью интерфейса с другими языками. Интерактивный SQL — это форма наиболее полезная непрограммистам. Все что вы узнаете относительно интерактивного SQL, в основном применимо и к вложенной форме. Изменения, необходимые для использования вложенной формы, будут использованы в последней главе этой книги.

СУБПОДРАЗДЕЛЕНИЯ SQL

И в интерактивной, и во вложенной формах SQL, имеются многочисленные части, или субподразделения. Так как вы вероятно столкнетесь с этой терминологией при чтении SQL, мы дадим некоторые пояснения. К сожалению, эти термины не используются повсеместно во всех реализациях. Они подчеркиваются ANSI и полезны на концептуальном уровне, но большинство SQL программ практически не обрабатывают их отдельно, так что они по существу становятся функциональными категориями команд SQL.

DDL *(Язык Определения Данных)* — так называемый Язык Описания Схемы в ANSI, состоит из команд, которые создают объекты (таблицы, индексы, просмотры, и так далее) в базе данных.

DML (*Язык Манипулирования Данными*) — это набор команд, которые определяют, какие значения представлены в таблицах в любой момент времени.

DCD (*Язык Управления Данными*) состоит из средств, которые определяют, разрешить ли пользователю выполнять определенные действия или нет.

Они являются составными частями DDL в ANSI. Не забывайте эти имена. Это не различные языки, а разделы команд SQL сгруппированных по их функциям.

РАЗЛИЧНЫЕ ТИПЫ ДАННЫХ

Не все типы значений, которые могут занимать поля таблицы — логически одинаковые. Наиболее очевидное различие — между числами и текстом. Вы не можете помещать числа в алфавитном порядке или вычитать одно имя из другого. Так как системы с реляционной базой данных базируются на связях между фрагментами информации, различные типы данных должны понятно отличаться друга от друга, так чтобы соответствующие процессы и сравнения могли быть в них выполнены.

В SQL это делается с помощью назначения каждому полю типа данных, который указывает на тип, значения которое это поле может содержать. Все значения в данном поле должны иметь одинаковый тип. В таблице Заказчиков, например, *cname* и *city* — содержат строки текста для оценки, *snum*, и *cnum* — это уже номера. По этой причине, вы не можете ввести значение Highest (Наивысший) или значение None (Никакой) в поле *rating,* которое имеет числовой тип данных. Это ограничение удачно, так как оно налагает некоторую структурность на ваши данные. Вы часто будете сравнивать некоторые или все значения в данном поле, поэтому вы можете выполнять действие только на определенных строках а не на всех. Вы не могли бы сделать этого если бы значения полей имели смешанный тип данных.

К сожалению, определение этих типов данных является основной областью, в которой большинство коммерческих программ баз данных и официальный стандарт SQL не всегда совпадают. ANSI SQL стандарт распознает только текст и тип номера, в то время как большинство коммерческих программ используют другие специальные типы. Такие как, **DATA** (ДАТА) и **TIME** (ВРЕМЯ) — фактически почти стандартные типы (хотя точный формат их меняется). Некоторые пакеты также поддерживают такие типы, как например **MONEY** (ДЕНЬГИ) и **BINARY** (ДВОИЧНЫЕ). (MONEY — это специальная система исчисления, используемая компьютерами. Вся информация в компьютере передается двоичными числами и затем преобразовываются в другие системы, что бы мы могли легко использовать их и понимать.)

ANSI определяет несколько различных типов значений чисел, различия между которыми довольно тонки и иногда их путают. Разрешенные ANSI типы данных перечислены в Приложении B.

Сложность числовых типов ANSI можно, по крайней мере частично, объяснить усилием сделать вложенный SQL, совместимым с рядом других языков.

Два типа чисел *ANSI***, INTEGER** (ЦЕЛОЕ ЧИСЛО) и **DECIMAL** (ДЕСЯТИЧНОЕ ЧИСЛО) (которые можно сокращать как **INT** и **DEC**, соответственно), будут адекватны для наших целей, также как и для целей большинства практических деловых прикладных программ. Естественно, что тип ЦЕЛОЕ можно представить как ДЕСЯТИЧНОЕ ЧИСЛО, которое не содержит никаких цифр справа от десятичной точки.

Тип для текста — **CHAR** (или СИМВОЛ), который относится к строке текста. Поле типа CHAR имеет определенную длину, которая определяется максимальным числом символов которые могут быть введены в это поле. Больше всего реализаций также имеют нестандартный тип называемый **VARCHAR** (ПЕРЕМЕННОЕ ЧИСЛО СИМВОЛОВ), который является текстовой строкой которая может иметь любую длину до определенного реализацией максимума (обычно 254 символа). CHARACTER и VARCHAR значения включаются в одиночные кавычки как "текст". Различие между CHAR и VARCHAR в том, что CHAR должен резервировать достаточное количество памяти для максимальной длины строки, а VARCHAR распределяет память так, как это необходимо.

Символьные типы состоят из всех печатных символов, включая числа. Однако, номер 1 не то же что символ "1". Символ "1" — только другой печатный фрагмент текста, не определяемый системой как наличие числового значения 1. Например 1 + 1 = 2, но "1" + "1" не равняется "2". Символьные значения сохраняются в компьютере как двоичные значения, но показываются пользователю как печатный текст. Преобразо-

вание следует за форматом определяемым системой которую вы используете. Этот формат преобразования будет одним из двух стандартных типов (возможно с расширениями), используемых в компьютерных системах: в **ASCII** коде (используемом во всех персональных и малых компьютерах) и **EBCDIC** коде (*Расширенном Двоично-Десятичном Коде Обмена Информации)* (используемом в больших компьютерах). Определенные операции, такие как упорядочивание в алфавитном порядке значений поля, будет изменяться вместе с форматом. Применение этих двух форматов будет обсуждаться в Главе 4.

Мы должны следить за рынком, а не ANSI, в использовании типа называемого **DATE** (ДАТОЙ). (В системе, которая не распознает тип ДАТА, вы конечно можете объявить дату как символьное или числовое поле, но это сделает большинство операций более трудоемкими.) Вы должны смотреть свою документацию по пакету программ, которые вы будете использовать, чтобы выяснить точно, какие типы данных она поддерживает.

SQL НЕСОГЛАСОВАННОСТИ

Вы можете понять из предшествующего обсуждения, что имеются самостоятельные несогласованности внутри продуктов мира SQL. SQL появился из коммерческого мира баз данных как инструмент, и был позже превращен в стандарт ANSI. К сожалению, ANSI не всегда определяет наибольшую пользу, поэтому программы пытаются соответствовать стандарту ANSI, не позволяя ему ограничивать их слишком сильно. ANSI — вид минимального стандарта — вы можете делать больше, чем он это позволяет, но вы должны быть способны получить те же самые результаты, что и при выполнении той же самой задачи.

ЧТО ТАКОЕ — ПОЛЬЗОВАТЕЛЬ?

SQL обычно находится в компьютерных системах, которые имеют больше, чем одного пользователя, и следовательно должны делать различие между ними (ваше семейство PC может иметь любое число пользователей, но оно обычно не имеет способов, чтобы отличать одного от другого). Обычно, в такой системе каждый пользователь имеет некий вид кода проверки прав, который идентифицирует его или ее (терминология изменяется). В начале сеанса с компьютером, пользователь входит в систему (регистрируется), сообщая компьютеру кто этот пользователь, идентифицированный с помощью определенного **ID** (*Идентификатора*). Любое количество людей, использующих тот же самый ID доступа, являются отдельными пользователями; и аналогично, один человек может представлять большое количество пользователей (в разное время), используя различные доступные Идентификаторы.

SQL следует этому примеру. Действия в большинстве сред SQL приведены к специальному доступному Идентификатору который точно соответствует определенному пользователю. Таблица или другой объект принадлежит пользователю, который имеет над ним полную власть. Пользователь может или не может иметь привилегии чтобы выполнять действие над объектом. Для наших целей, мы договоримся, что любой пользователь имеет привилегии необходимые чтобы выполнять любое действие, пока мы не возвратимся специально к обсуждению привилегий в Главе 22.

Специальное значение — **USER** (ПОЛЬЗОВАТЕЛЬ) может использоваться как аргумент в команде. Оно указывает на доступный Идентификатор пользователя, выдавшего команду.

УСЛОВИЯ И ТЕРМИНОЛОГИЯ

Ключевые слова — это слова, которые имеют специальное значение в SQL. Они могут быть командами, но не текстом и не именами объектов. Мы будем выделять ключевые слова печатая их *ЗАГЛАВНЫМИ БУКВАМИ*. Вы должны соблюдать осторожность чтобы не путать ключевые слова с терминами.

SQL имеет определенные специальные термины, которые используются, чтобы описывать его. Среди них — такие слова как *запрос*, *предложение* и *предикат*, которые являются важнейшими в описании и понимании языка, но не означают что-нибудь самостоятельное для SQL.

Команды, или предложения, являются инструкциями, которыми Вы обращаетесь к SQL базе данных. Команды состоят из одной или более отдельных логических частей, называемых *предложениями*. Предложения начинаются ключевым словом, для которого они являются проименованными, и состоят из ключевых слов и аргументов. Например предложения с которыми вы можете сталкиваться — это "FROM Salespeope" и “WHERE city = “London””. *Аргументы* завершают или изменяют значение предложения. В примерах выше, **Salespeople** — аргумент, а **FROM** — ключевое слово предложения FROM. Аналогично, “**city = “London”**” — агрумент предложения WHERE.

Объекты — структуры в базе данных, которым даны имена и сохраняются в памяти. Они включают в себя базовые таблицы, представления (два типа таблиц), и индексы. Чтобы показать Вам, как формируются команды, мы будем делать это на примерах. Имеется, однако, более формальный метод описания команд, использующих стандартизированные условные обозначения. Мы будем использовать его в более поздних главах, для удобства чтобы понимать эти условные обозначения в случае, если вы столкнетесь с ним в других SQL документах.

Квадратные скобки (**[ ]**) будут указывать части, которые могут неиспользоваться, а многоточия (**...)** указывать, что все предшествующее им может повторяться любое число раз. Слова, обозначенные в угловых скобках (**<>**) — специальные термины, которые объясняют что они собой представляют. Мы упростили стандартную терминологию SQL значительно, но без ухудшения его понимания.

РЕЗЮМЕ

Мы быстро прошли основы в этой главе. Но нашим намерением и было — просто пролететь над основами SQL, так чтобы вы могли понять идею относительно всего объема.

Когда мы возвратимся к основе в следующей главе, некоторые вещи станут более конкретными. Теперь вы знаете кое-что относительно SQL — какова его структура, как он используется, как он представляет данные, и как они определяются (и некоторые несогласованности появляющиеся при этом), и некоторые условные обозначения и термины используемые чтобы описывать их.

Все это — много информации для одной главы; мы не ожидаем что бы вы запомнили все эти подробности, но вы сможете вернуться позже к ним если понадобится.

По Главе 3, мы будем идти, показывая конкретно, как формируются команды и что они делают. Мы представим вам команду SQL используемую чтобы извлекать информацию из таблиц, и которая является наиболее широко используемой командой в SQL. К концу этой главы, вы будете способны извлекать конкретную информацию из вашей базы данных с высокой степенью точности.

РАБОТА С SQL

1. Какое наибольшее основное различие между типами данных в SQL?
2. Распознает ANSI тип данных DATA ?
3. Какой подраздел SQL используется чтобы помещать значения в таблицы ?
4. Что такое — ключевое слово?

(См. Приложение А для ответов.)

3

ИСПОЛЬЗОВАНИЕ SQL ДЛЯ ИЗВЛЕЧЕНИЯ ИНФОРМАЦИИ ИЗ ТАБЛИЦ

В ЭТОЙ ГЛАВЕ МЫ ПОКАЖЕМ ВАМ, КАК ИЗВЛЕКАТЬ информацию из таблиц. Вы узнаете как опускать или переупорядочивать столбцы и как автоматически устранять избыточность данных из вашего вывода. В заключение, вы узнаете как устанавливать условие (проверку), которую вы можете использовать, чтобы определить какие строки таблицы используются в выводе. Эта последняя особенность, будет далее описана в более поздних главах и является одной из наиболее изящных и мощных в SQL.

СОЗДАНИЕ ЗАПРОСА

Как мы подчеркивали ранее, SQL символизирует собой Структурированный Язык Запросов. Запросы — вероятно, наиболее часто используемый аспект SQL. Фактически, для категории SQL пользователей, маловероятно чтобы кто-либо использовал этот язык для чего-то друго. По этой причине, мы будем начинать наше обсуждение SQL с обсуждения запроса и как он выполняется на этом языке.

ЧТО ТАКОЕ ЗАПРОС ?

Запрос — команда, которую вы даете вашей программе базы данных, и которая сообщает ей, чтобы она вывела определенную информацию из таблиц в память. Эта информация обычно посылается непосредственно на экран компьютера или терминала, которым вы пользуетесь, хотя, в большинстве случаев, ее можно также послать принтеру, сохранить в файле (как объект в памяти компьютера), или представить как вводную информацию для другой команды или процесса.

ГДЕ ПРИМЕНЯЮТСЯ ЗАПРОСЫ ?

Запросы обычно рассматриваются как часть языка DML. Однако, так как запрос не меняет информацию в таблицах, а просто показывает ее пользователю, мы будем рассматривать запросы как самостоятельную категорию среди команд DML которые производят действие, а не просто показывают содержание базы данных.

Все запросы в SQL состоят из одиночной команды. Структура этой команды обманчиво проста, потому что вы должны расширять ее так, чтобы выполнить высоко сложные оценки и обработки данных. Эта команда называется — **SELECT** (ВЫБОР).

КОМАНДА SELECT

В самой простой форме, команда SELECT просто инструктирует базу данных, чтобы извлечь информацию из таблицы. Например, вы могли бы вывести таблицу Продавцов напечатав следующее:

```
SELECT snum, sname, sity, comm
FROM Salespeople;
```

Вывод для этого запроса показывается в Рисунке 3.1.

```
 ===============  SQL Execution Log ============
| SELECT snum, sname, sity, comm                 |
| FROM Salespeople;                              |
| ===============================================|
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1001      Peel         London        0.12    |
|   1002      Serres       San Jose      0.13    |
|   1004      Motika       London        0.11    |
|   1007      Rifkin       Barcelona     0.15    |
|   1003      Axelrod      New York      0.10    |
 ================================================
```

Рисунок 3.1: команда SELECT

Другими словами, эта команда просто выводит все данные из таблицы. Большинство программ будут также давать заголовки столбца как выше, а некоторые позволяют детальное форматирование вывода, но это уже вне стандартной спецификации.

Имеется объяснение каждой части этой команды:

SELECT — Ключевое слово, которое сообщает базе данных, что эта команда — запрос. Все запросы начинаются этим словом, сопровождаемым пробелом.

snum, sname — Это — список столбцов из таблицы которые выбираются запросом. Любые столбцы не перечисленные здесь не будут включены в вывод команды. Это, конечно, не значит, что они будут удалены или их информация будет стерта из таблиц, потому что запрос не воздействует на информацию в таблицах, он только показывает данные.

FROM Salespeople — Ключевое слово, подобно SELECT, которое должно быть представлено в каждом запросе. Оно сопровождается пробелом и затем именем таблицы используемой в качестве источника информации. В данном случае — это таблица Продавцов (Salespeople).

; — Точка с запятой используется во всех интерактивных командах SQL, чтобы сообщать базе данных что команда заполнена и готова выполниться. В некоторых системах наклонная черта влево (\) в строке, является индикатором конца команды.

Естественно, запрос такого характера не обязательно будет упорядочивать вывод любым указаным способом. Та же самая команда выполненная с теми же самыми данными но в разное время не сможет вывести тот же самый порядок. Обычно, строки обнаруживаются в том порядке в котором они найдены в таблице, поскольку как мы установили в предыдущей главе — этот порядок произволен. Это не обязательно будет тот порядок, в котором данные вводились или сохранялись. Вы можете упорядочивать вывод командами SQL непосредственно, с помощью специального предложения. Позже, мы покажем как это делается. А сейчас, просто усвойте, что в отсутствии явного упорядочения, нет никакого определенного порядка в вашем выводе.

Наше использование возврата (Клавиша ENTER) является произвольным. Мы должны точно установить, как удобнее составить запрос, в несколько строк или в одну строку, следующим образом:

```
SELECT snum, sname, city, comm FROM Salespeople;
```

С тех пор как SQL использует точку с запятой чтобы указывать конец команды, большинство программ SQL обрабатывают возврат (через нажим Возврат или клави-

шу ENTER) как пробел. Это — хорошая идея чтобы использовать возвраты и выравнивание что мы делали это ранее, чтобы сделать ваши команды более легкими для чтения и более правильными.

ВЫБИРАЙТЕ ВСЕГДА САМЫЙ ПРОСТОЙ СПОСОБ

Если вы хотите видеть каждый столбец таблицы, имеется необязательное сокращение которое вы можете использовать. Звездочка (*) может применяться для вывода полного списка столбцов следующим образом:

```
SELECT *
FROM Salespeople;
```

Это приведет к тому же результату, что и наша предыдущая команда.

ОПИСАНИЕ SELECT

В общем случае, команда SELECT начинается с ключевого слова SELECT, сопровождаемого пробелом. После этого должен следовать список имен столбцов, которые вы хотите видеть, отделяемые запятыми. Если вы хотите видеть все столбцы таблицы, вы можете заменить этот список звездочкой (*). Ключевое слово FROM следующее далее, сопровождается пробелом и именем таблицы, запрос к которой делается. В заключение, точка с запятой (;) должна использоваться, чтобы закончить запрос и указать что команда готова к выполнению.

ПРОСМОТР ТОЛЬКО ОПРЕДЕЛЕННОГО СТОЛБЦА ТАБЛИЦЫ

Команда SELECT способна извлечь строго определенную информацию из таблицы. Сначала, мы можем предоставить возможность увидеть только определенные столбцы таблицы. Это выполняется легко, простым исключением столбцов которые вы не хотите видеть, из части команды SELECT. Например, запрос

```
SELECT sname, comm
FROM Salespeople;
```

будет производить вывод показанный на Рисунке 3.2.

```
 ===============  SQL Execution Log ============
| SELECT snum, comm                              |
| FROM Salespeople;                              |
| ===============================================|
|        sname             comm                  |
|   -------------      ---------                 |
|        Peel              0.12                  |
|        Serres            0.13                  |
|        Motika            0.11                  |
|        Rifkin            0.15                  |
|        Axelrod           0.10                  |
 ================================================
```

Рисунок 3.2: Выбор определенных столбцов

Могут иметься таблицы, которые имеют большое количество столбцов, содержащих данные, не все из которых являются относящимися к поставленой задаче. Следовательно, вы можете найти способ подбора и выбора только полезных для Вас столбцов.

ПЕРЕУПОРЯДОЧЕНИЕ СТОЛБЦА

Даже если столбцы таблицы, по определению, упорядоченны, это не означает что вы будете восстанавливать их в том же порядке. Конечно, звездочка (*) покажет все столбцы в их естественном порядке, но если вы укажете столбцы отдельно, вы можете получить их в том порядке, в котором хотите. Давайте рассмотрим таблицу Порядков, содержащую дату приобретения (odate), номер продавца (snum), номер порядка (onum), и суммы приобретения (amt):

```
SELECT odate, snum, onum, amt
FROM Orders;
```

Вывод этого запроса показан на Рисунке 3.3.

```
 ==============  SQL Execution Log  ==============
|  SELECT odate, snum, onum, amt                  |
|  FROM Orders;                                   |
| ------------------------------------------------|
|     odate        snum        onum          amt  |
| -----------   -------      ------     --------- |
| 10/03/1990       1007        3001         18.69 |
| 10/03/1990       1001        3003        767.19 |
| 10/03/1990       1004        3002       1900.10 |
| 10/03/1990       1002        3005       5160.45 |
| 10/03/1990       1007        3006       1098.16 |
| 10/04/1990       1003        3009       1713.23 |
| 10/04/1990       1002        3007         75.75 |
| 10/05/1990       1001        3008       4723.00 |
| 10/06/1990       1002        3010       1309.95 |
| 10/06/1990       1001        3011       9891.88 |
 ================================================
```

Рисунок 3.3: Реконструкция столбцов

Как вы можете видеть, структура информации в таблицах — это просто основа для активной перестройки структуры в SQL.

УДАЛЕНИЕ ИЗБЫТОЧНЫХ ДАННЫХ

DISTINCT (ОТЛИЧИЕ) — аргумент, который обеспечивает Вас способом устранять двойные значения из вашего предложения SELECT. Предположим что вы хотите знать, какие продавцы в настоящее время имеют свои порядки в таблице Порядков. Под порядком (здесь и далее) будет пониматься запись в таблицу Порядков, регистрирующую приобретения, сделанные в определенный день определенным заказчиком у определенного продавца на определенную сумму). Вам не нужно знать, сколько порядков имеет каждый; вам нужен только список номеров продавцов (snum). Поэтому Вы можете ввести:

```
SELECT snum
FROM Orders;
```

для получения вывода, показанного в Рисунке 3.4.

```
 ===============  SQL Execution Log ============
| SELECT snum                                    |
| FROM Orders;                                   |
| ============================================== |
|   snum                                         |
| -------                                        |
|   1007                                         |
|   1001                                         |
|   1004                                         |
|   1002                                         |
|   1007                                         |
|   1003                                         |
|   1002                                         |
|   1001                                         |
|   1002                                         |
|   1001                                         |
 ================================================
```

Рисунок 3.4: SELECT с дублированием номеров продавцов.

Для получения списка без дубликатов, для удобочитаемости, вы можете ввести следующее:

```
SELECT DISTINCT snum
FROM Orders;
```

Вывод для этого запроса показан в Рисунке 3.5.

Другими словами, DISTINCT следит за тем, какие значения были ранее, так что бы они не были продублированы в списке. Это — полезный способ избежать избыточности данных, но важно что бы при этом вы понимали что вы делаете. Если вы не хотите потерять некоторые данные, вы не должны безоглядно использовать DISTINCT, потому что это может скрыть какую-то проблему или какие-то важные данные. Например, вы могли бы предположить что имена всех ваших заказчиков различны. Если кто-то помещает второго Clemens в таблицу Заказчиков, а вы используете **SELECT DISTINCT cname**, вы не будете даже знать о существовании двойника. Вы можете получить не того Clemens и даже не знать об этом. Так как вы не ожидаете избыточности, в этом случае вы не должны использовать DISTINCT.

ПАРАМЕТРЫ DISTINCT

DISTINCT может указываться только один раз в данном предложении SELECT. Если предложение выбирает многочисленные поля, DISTINCT опускает строки, где все выбранные поля идентичны. Строки, в которых некоторые значения одинаковы а некоторые различны — будут сохранены. DISTINCT, фактически, приводит к показу всей строки вывода, не указывая полей (за исключением, когда он используется внутри агрегатных функций, как описано в Главе 6), так что нет никакого смысла чтобы его повторять.

```
 ===============  SQL Execution Log ============
| SELECT DISTINCT snum                           |
| FROM Orders;                                   |
| ============================================== |
|   snum                                         |
| -------                                        |
|   1001                                         |
|   1002                                         |
|   1003                                         |
|   1004                                         |
|   1007                                         |
 ================================================
```

Рисунок 3.5: SELECT без дублирования

DISTINCT ВМЕСТО ALL

Вместо DISTINCT, вы можете указать — ALL. Это будет иметь противоположный эффект, дублирование строк вывода сохранится. Так как это — тот же самый случай когда вы не указываете ни DISTINCT ни ALL, то ALL — по существу скорее пояснительный, а не действующий аргумент.

КВАЛИФИЦИРОВАННЫЙ ВЫБОР ПРИ ИСПОЛЬЗОВАНИИ ПРЕДЛОЖЕНИЙ

Таблицы имеют тенденцию становиться очень большими, поскольку с течением времени, все большее и большее количество строк в нее добавляется. Поскольку обычно из них только определенные строки интересуют вас в данное время, SQL дает возможность вам устанавливать критерии, чтобы определить, какие строки будут выбраны для вывода.

WHERE — предложение команды SELECT, которое позволяет вам устанавливать предикаты, условие которых может быть или верным или неверным для любой строки таблицы. Команда извлекает только те строки из таблицы, для которых такое утверждение верно. Например, предположим вы хотите видеть имена и комиссионные всех продавцов в Лондоне. Вы можете ввести такую команду:

```
SELECT sname, city
FROM Salespeople
WHERE city = "LONDON";
```

Когда предложение WHERE представлено, программа базы данных просматривает всю таблицу по одной строке и исследует каждую строку, чтобы определить верно ли утверждение. Следовательно, для записи Peel, программа рассмотрит текущее значение столбца city, определит что оно равно "London", и включит эту строку в вы-

вод. Запись для Serres не будет включена, и так далее. Вывод для вышеупомянутого запроса показан в Рисунке 3.6.

```
 ===============  SQL Execution Log ============
| SELECT sname, city                             |
| FROM Salespeople                               |
| WHERE city = 'London'                          |
| ============================================== |
|   sname           city                         |
|  -------       ----------                      |
|   Peel            London                       |
|   Motika          London                       |
 ================================================
```

Рисунок 3.6: SELECT с предложением WHERE

Давайте попробуем пример с числовым полем в предложении WHERE. Поле rating таблицы Заказчиков предназначено, чтобы разделять заказчиков на группы, основанные на некоторых критериях, которые могут быть получены в итоге через этот номер. Возможно это — форма оценки кредита или оценки основанные на опыте предыдущих приобретений. Такие числовые коды могут быть полезны в реляционных базах данных как способ подведения итогов сложной информации. Мы можем выбрать всех заказчиков с рейтингом 100, следующим образом:

```
SELECT *
FROM Customers
WHERE rating = 100;
```

Одиночные кавычки не используются здесь потому, что оценка — это числовое поле. Результаты запроса показаны в Рисунке 3.7.

Предложение WHERE совместимо с предыдущим материалом в этой главе. Другими словами, вы можете использовать номера столбцов, устранять дубликаты, или переупорядочивать столбцы в команде SELECT которая использует WHERE. Однако, вы можете изменять порядок столбцов для имен только в предложении SELECT, но не в предложении WHERE.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Customers                                 |
| WHERE rating = 100;                            |
| ============================================== |
|   cnum     cname    city     rating    snum    |
|  ------   --------  ------    ----    ------   |
|   2001     Hoffman  London     100     1001   |
|   2006     Clemens  London     100     1001   |
|   2007     Pereira  Rome       100     1001   |
 ================================================
```

Рисунок 3.7: SELECT с числовым полем в предикате

РЕЗЮМЕ

Теперь вы знаете несколько способов заставить таблицу давать вам ту информацию, какую вы хотите, а не просто выбрасывать наружу все ее содержание. Вы можете переупорядочивать столбцы таблицы или устранять любую из них. Вы можете решать, хотите вы видеть дублированные значения или нет.

Наиболее важно то, что вы можете устанавливать условие называемое *предикатом*, которое определяет или не определяет указанную строку таблицы из тысяч таких же строк, будет ли она выбрана для вывода.

Предикаты могут становиться очень сложными, предоставляя вам высокую точность в решении, какие строки вам выбирать с помощью запроса. Именно эта способность решать точно, что вы хотите видеть, делает запросы SQL такими мощными.

Следующие несколько глав будут посвещены, в большей мере, особенностям, которые расширяют мощность предикатов. В Главе 4 вам будут представлены операторы иные чем те, которые используются в условиях предиката, а также способы объединения многочисленых условий в единый предикат.

РАБОТА С SQL

1. Напишите команду SELECT которая бы вывела номер порядка, сумму, и дату для всех строк из таблицы Порядков.
2. Напишите запрос который вывел бы все строки из таблицы Заказчиков, для которых номер продавца = 1001.
3. Напишите запрос который вывел бы таблицу со столбцами в следующем порядке: city, sname, snum, comm.
4. Напишите команду SELECT которая вывела бы оценку (rating), сопровождаемую именем каждого заказчика в San Jose.
5. Напишите запрос, который вывел бы значения snum всех продавцов в текущем порядке из таблицы Порядков без каких бы то ни было повторений.

(См. Приложение А для ответов.)

4

ИСПОЛЬЗОВАНИЕ РЕЛЯЦИОННЫХ И БУЛЕВЫХ ОПЕРАТОРОВ ДЛЯ СОЗДАНИЯ БОЛЕЕ ИЗОЩРЕННЫХ ПРЕДИКАТОВ

В ГЛАВЕ 3, ВЫ УЗНАЛИ ЧТО ПРЕДИКАТЫ МОГУТ оценивать равенство оператора как верного или неверного. Они могут также оценивать другие виды связей кроме равенств. Эта глава будет исследовать другие реляционные операторы используемые в SQL. Вы также узнаете как использовать операторы Буля, чтобы изменять и объединять значения предиката. С помощью операторов Буля (или проще говоря логических операторов), одиночный предикат может содержать любое число условий. Это позволяет вам создавать очень сложные предикаты. Использование круглых скобок в структуре этих сложных предикатов будет также объясняться.

РЕЛЯЦИОННЫЕ ОПЕРАТОРЫ

Реляционный оператор — математический символ, который указывает на определенный тип сравнения между двумя значениями. Вы уже видели как используются равенства, такие как 2 + 3 = 5 или city = "London". Но также имеются другие реляционные операторы. Предположим что вы хотите видеть всех Продавцов с их комиссионными выше определенного значения. Вы можете использовать тип сравнения "больше чем" — (**>**).

Реляционные операторы которыми располагает SQL :

= *Равный*
> *Больше чем*
< *Меньше чем*
>= *Больше чем или равно*
<= *Меньше чем или равно*
<> *Не равно*

Эти операторы имеют стандартные значения для числовых значений. Для значения символа, их определение зависит от формата преобразования, **ASCII** или **EBCDIC**, который вы используете.

SQL сравнивает символьные значения в терминах основных номеров как определено в формате преобразования. Даже значение символа, такого как "1", который представляет номер, не обязательно равняется номеру, который он представляет. Вы можете использовать реляционные операторы, чтобы установить алфавитный порядок — например, "a" < "n" где средство ***a*** первое в алфавитном порядке — но все это ограничивается с помощью параметра преобразования формата.

И в ASCII и в EBCDIC, символы — по значению: меньше чем все другие символы которым они предшествуют в алфавитном порядке и имеют один вариант (верхний или нижний). В ASCII, все символы верхнего регистра — меньше чем все символы нижнего регистра, поэтому "Z" < "a", а все номера — меньше чем все символы, поэтому "1" < "Z". То же относится и к EBCDIC. Чтобы сохранить обсуждение более простым, мы допустим что вы будете использовать текстовый формат ASCII. Проконсультируйтесь с вашей документацией системы если вы неуверены какой формат вы используете или как он работает.

Значения сравниваемые здесь называются — *скалярными значениями*. Скалярные значения производяться скалярными выражениями; 1 + 2 — это скалярное выражение которое производит скалярное значение 3. Скалярное значение может быть символом или числом, хотя очевидно что только номера используются с арифметическими операторами, такими как + (плюс) или * (звезда).

Предикаты обычно сравнивают значения скалярных величин, используя или реляционные операторы или специальные операторы SQL чтобы увидеть верно ли это сравнение. Некоторые операторы SQL описаны в Главе 5.

Предположим что вы хотите увидеть всех заказчиков с оценкой (rating) выше 200. Так как 200 — это скалярное значение, как и значение в столбце оценки, для их сравнения вы можете использовать реляционный оператор.

```
SELECT *
FROM Customers
WHERE rating > 200;
```

Вывод для этого запроса показывается в Рисунке 4.1.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Customers                                 |
| WHERE rating > 200;                            |
| ============================================== |
|   snum     cname    city     rating    snum    |
|  -----   --------  --------  -----   ------    |
|   2004     Crass    Berlin     300     1002    |
|   2008     Cirneros San Jose   300     1007    |
 ================================================
```

Рисунок 4.1: Использование больше чем (>)

Конечно, если бы мы захотели увидеть еще и заказчиков с оценкой, равной 200, мы стали бы использовать предикат

```
rating >= 200
```

БУЛЕВЫ ОПЕРАТОРЫ

Основные Булевы операторы также распознаются в SQL. Выражения Буля — являются или верными или неверными, подобно предикатам. Булевы операторы связывают одно или более верных/неверных значений и производят едиственное верное/или/неверное значение. Стандартными операторами Буля, распознаваемыми в SQL, являются: **AND**, **OR**, и **NOT**.

Существуют другие, более сложные, операторы Буля (типа "исключенный или"), но они могут быть сформированы из этих трех простых операторов — AND, OR, NOT.

Как вы можете понять, Булева верная / неверная логика — основана на цифровой компьютерной операции; и фактически, весь SQL (или любой другой язык) может быть сведен до уровня Булевой логики.

Операторы Буля и как они работают:

AND берет два Буля (в форме A AND B) как аргументы и оценивает их по отношению к истине, верны ли они оба.

OR берет два Буля (в форме A OR B) как аргументы и оценивает на правильность, верен ли один из них.

NOT берет одиночный Булев (в форме NOT A) как аргументы и заменяет его значение с неверного на верное или верное на неверное.

Связывая предикаты с операторами Буля, вы можете значительно увеличить их возможности. Предположим вы хотите видеть всех заказчиков в San Jose которые имеют оценку (рейтинг) выше 200:

```
SELECT *
FROM Customers
WHERE city = “San Jose”
AND rating > 200;
```

Вывод для этого запроса показан на Рисунке 4.2. Имеется только один заказчик, который удовлетворяет этому условию.

```
 ===============  SQL Execution Log ============
|                                               |
| SELECT *                                      |
| FROM Customers                                |
| WHERE city = 'San Jose'                       |
| AND rating > 200;                             |
| ============================================= |
|   cnum     cname    city     rating    snum   |
|  ------   --------  --------  ----    -----   |
|   2008     Cirneros San Jose   300     1007   |
 ===============================================
```

Рисунок 4.2: SELECT использующий AND

Если вы же используете OR, вы получите всех заказчиков, которые находились в San Jose **или** (OR) которые имели оценку выше 200.

```
SELECT *
FROM Customers
WHERE city = “San Jose” OR rating > 200;
```

Вывод для этого запроса показывается в Рисунке 4.3.

```
 ===============  SQL Execution Log ============
| SELECT *                                      |
| FROM Customers                                |
| WHERE city = 'San Jose'                       |
| OR rating > 200;                              |
| ============================================= |
|   cnum     cname    city     rating    snum   |
|  -----    -------  --------  -----   ------   |
|   2003     Liu      San Jose   200     1002   |
|   2004     Grass    Berlin     300     1002   |
|   2008     Cirneros San Jose   300     1007   |
 ===============================================
```

Рисунок 4.3: SELECT использующий OR

NOT может использоваться для инвертирования значений Буля. Имеется пример запроса с NOT:

```
SELECT *
FROM Customers
WHERE city = "San Jose" OR NOT rating > 200;
```

Вывод этого запроса показывается в Рисунке 4.4.

```
 ===============  SQL Execution Log ============
| SELECT *                                      |
| FROM Customers                                |
| WHERE city = 'San Jose'                       |
| OR NOT rating > 200;                          |
| ============================================= |
|   cnum     cname    city     rating    snum   |
|  ------   --------  ------   -----    -----   |
|   2001     Hoffman  London     100     1001   |
|   2002     Giovanni Rome       200     1003   |
|   2003     Liu      San Jose   200     1002   |
|   2006     Clemens  London     100     1001   |
|   2008     Cirneros San Jose   300     1007   |
|   2007     Pereira  Rome       100     1004   |
 ==============================================
```

Рисунок 4.4: SELECT использующий NOT

Все записи за исключением Grass были выбраны. Grass не был в San Jose, и его оценка была больше чем 200, так что он потерпел неудачу при обеих проверках. В каждой из других строк встретился один или другой или оба критериев. Обратите внимание что оператор NOT должен предшествовать Булеву оператору, чье значение должно измениться, и не должен помещаться перед реляционным оператором. Например неправильным вводом оценки предиката будет:

```
rating NOT > 200
```

Он выдаст другую отметку. А как SQL оценит следующее?

```
SELECT *
FROM Customers
WHERE NOT city = “San Jose” OR rating > 200;
```

NOT применяется здесь только к выражению **city = 'SanJose'**, или к выражению **rating > 200** тоже? Как и написано, правильный ответ будет прежним. SQL может применять NOT с выражением Буля только сразу после него. Вы можете получить другой результат при команде:

```
SELECT *
FROM Customers
WHERE NOT( city = “San Jose” OR rating > 200 );
```

Здесь SQL понимает круглые скобки как означающие, что все внутри них будет оцениваться первым и обрабатываться как единое выражение с помощью всего что снаружи них (это является стандартной интерпретацией в математике). Другими словами, SQL берет каждую строку и определяет, соответствует ли истине равенство **city = "San Jose'** или равенство **rating > 200**. Если любое условие верно, выражение Буля внутри круглых скобок верно. Однако, если выражение Буля внутри круглых скобок верно, предикат как единое целое неверен, потому что NOT преобразует верно в неверно и наоборот.

Вывод для этого запроса показывается в Рисунке 4.5.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Customers                                 |
| WHERE NOT  (city = 'San Jose'                  |
| OR rating > 200);                              |
| ============================================== |
|   cnum     cname    city     rating    snum    |
|  -----   --------  -------   -----   ------    |
|   2001     Hoffman  London     100     1001    |
|   2002     Giovanni Rome       200     1003    |
|   2006     Clemens  London     100     1001    |
|   2007     Pereira  Rome       100     1004    |
 ================================================
```

Рисунок 4.5: SELECT использующий NOT и вводное предложение

Имеется намеренно сложный пример. Посмотрим сможете ли вы проследить его логику (вывод показан в Рисунке 4.6):

```
SELECT *
FROM Orders
WHERE NOT ((odate = 10/03/1990 AND snum >1002) OR amt > 2000.00);
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                         |
| FROM Orders                                      |
| WHERE NOT ((odate = 10/03/1990 AND snum > 1002)  |
| OR amt > 2000.00);                               |
| ================================================ |
|   onum       amt      odate      cnum     snum   |
|  ------   --------  ----------  -----    -----   |
|   3003      767.19  10/03/1990   2001     1001   |
|   3009     1713.23  10/04/1990   2002     1003   |
|   3007       75.75  10/04/1990   2004     1002   |
|   3010     1309.95  10/06/1990   2004     1002   |
 ================================================
```

Рисунок 4.6: Полный (комплексный) запрос

Несмотря на то, что Булевы операторы индивидуально просты, они не так просты, когда комбинируются в комплексное выражение.

Способ оценки комплекса Булева состоит в том, чтобы оценивать Булевы выражения, наиболее глубоко вложенные в круглых скобках, объединять их в единичное Булево значение, и затем объединять его с верхними значениями.

Имеется подробное объяснение того как пример выше был вычислен. Наиболее глубоко вложенные выражения Буля в предикате — это odate = 10/03/1990 и snum > 1002 являются объединеными с помощью AND, формируя одно выражение Буля, которое будет оценено как верное для всех строк, в которых встретились оба эти условия. Это составное Булево выражение (которое мы будем называть Булево номер 1, или B1 для краткости) объединяется с выражением (amt) > 2000.00 (B2) с помощью OR, формируя третье выражение (B3), которое является верным для данной строки, если или B1 или B2 — верны для этой строки. B3 полностью содержится в круглых скобках которым предшествует NOT, формируя последнее выражение Буля (B4), которое является условием предиката. Таким образом B4, предикат запроса, — будет верен всякий раз, когда B3 неправилен. B3 — неправилен всегда, когда B1 и B2 — оба неверны. B1 неправилен для строки если дата порядка строки не 10/03/1990, или если значение snum не большее чем 1002. B2 неправилен для всех строк, значения суммы приобретений которых не превышает 2000.00. Любая строка со значением выше 2000.00 сделает B2 — верным; в результате B3 будет верен, а B4 нет. Следовательно, все эти

строки будут удалены из вывода. Из оставшихся, строки, которые на 3 Октября имеют snum > 1002 (такие, как строки для onum 3001 на 3 Октября со snum = 1007), делают B1 верным, с помощью верного B3 и неверного предиката запроса. Они будут также удалены из вывода. Вывод показан для строк которые оставлены.

РЕЗЮМЕ

В этой главе вы значительно расширили ваше знакомство с предикатами. Теперь вы можете находить значения, которые связаны с данным значением любым способом — определяемым различными реляционными операторами. Вы можете также использовать операторы Буля **AND** и **OR** чтобы много условий, каждое из которых автономно в предикатах, объединять в единый предикат. Оператор Буля **NOT**, как вы уже видели, может изменять значение условия или группы условий на противоположное.

Булевы и реляционные операторы могут эффективно управляться с помощью круглых скобок, которые определяют порядок, в котором операции будут выполнены. Эти операции применимы к любому уровню сложности и вы поняли как сложные условия могут создаваться из этих простых частей.

Теперь, когда мы показали как используются стандартные математические операторы, мы можем перейти к операторам, которые являются исключительными в SQL. Это мы сделаем в Главе 5.

РАБОТА С SQL

1. Напишите запрос который может дать вам все порядки со значениями суммы выше чем $1,000.
2. Напишите запрос который может выдать вам поля sname и city для всех продавцов в Лондоне с комиссионными выше .10.
3. Напишите запрос к таблице Заказчиков чей вывод может включить всех заказчиков с оценкой =< 100, если они не находятся в Риме.
4. Что может быть выведено в результате следующего запроса ?

```
SELECT *
FROM Orders
WHERE (amt < 1000 OR NOT (odate = 10/03/1990 AND cnum > 2003 ));
```

5. Что может быть выведено в результате следующего запроса ?

```
SELECT *
FROM Orders
WHERE NOT ((odate = 10/03/1990 OR snum > 1006) AND amt > = 1500 );
```

6. Как можно проще переписать такой запрос ?

```
SELECT snum, sname, city, comm
FROM Salespeople
WHERE ( comm > + .12 OR comm < .14 );
```

(См. Приложение A для ответов.)

5

ИСПОЛЬЗОВАНИЕ СПЕЦИАЛЬНЫХ ОПЕРАТОРОВ В УСЛОВИЯХ

В ДОПОЛНЕНИИ К РЕЛЯЦИОННЫМ И БУЛЕВСКИМ операторам, обсуждаемым в Главе 4, SQL использует специальные операторы **IN, BETWEEN, LIKE,** и **IS NULL.** В этой главе, вы узнаете как их использовать и как реляционные операторы позволяют создавать более сложные и мощные предикаты. Обсуждение оператора IS NULL будет включать отсутствие данных и значение NULL, которое указывает на то: что данные отсутствуют. Вы также узнаете о разновидностях использования оператора NOT применяющегося с этими операторами.

ОПЕРАТОР IN

Оператор IN определяет набор значений в которое данное значение может или не может быть включено. В соответствии с нашей учебной базой данных на которой вы обучаетесь по настоящее временя, если вы хотите найти всех продавцов, которые размещены в Barcelona или в London, вы должны использовать следующий запрос (вывод показывается в Рисунке 5.1):

```
SELECT *
FROM Salespeople
WHERE city = 'Barcelona' OR city = 'London';
```

```
 ================ SQL Execution Log ============
| SELECT *                                       |
| FROM Salespeople                               |
| WHERE city = 'Barcelona'                       |
| OR city = 'London';                            |
| ===============================================|
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1001      Peel         London        0.12    |
|   1004      Motika       London        0.11    |
|   1007      Rifkin       Barcelona     0.15    |
 ================================================
```

Рисунок 5.1: Нахождение продавцов в Барселоне и Лондоне

Имеется и более простой способ получить ту же информацию:

```
SELECT *
FROM Salespeople
WHERE city IN ( 'Barcelona', 'London' );
```

Вывод для этого запроса показывается в Рисунке 5.2.

```
 ================ SQL Execution Log ============
| SELECT *                                       |
| FROM Salespeople                               |
| WHERE city IN ('Barcelona', 'London';          |
| ===============================================|
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1001      Peel         London        0.12    |
|   1004      Motika       London        0.11    |
|   1007      Rifkin       Barcelona     0.15    |
 ================================================
```

Рисунок 5.2: SELECT использует IN

Как вы можете видеть, IN определяет набор значений с помощью имен членов набора заключеных в круглые скобки и отделенных запятыми. Он затем проверяет различные значения указанного поля пытаясь найти совпадение со значениями из набора. Если это случается, то предикат верен. Когда набор содержит значения номеров, а не символов, одиночные кавычки опускаются. Давайте найдем всех заказчиков относящихся к продавцам имеющих значения snum = 1001, 1007, и 1004. Вывод для следующего запроса показан на Рисунке 5.3:

```
SELECT *
FROM Customers
WHERE cnum IN ( 1001, 1007, 1004 );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Customers                                 |
| WHERE snum IN (1001, 1007, 1004);              |
| ============================================== |
|   snum     cname    city     rating    snum    |
|  ------   --------  ------    ----    ------   |
|   2001     Hoffman  London     100     1001    |
|   2006     Clemens  London     100     1001    |
|   2008     Cisneros San Jose   300     1007    |
|   2007     Pereira  Rome       100     1004    |
 ================================================
```

Рисунок 5.3: SELECT использует IN с номерами

ОПЕРАТОР BETWEEN

Оператор **BETWEEN** похож на оператор IN. В отличии от определения по номерам из набора, как это делает IN, BETWEEN определяет диапазон, значения которого должны уменьшаться что делает предикат верным. Вы должны ввести ключевое слово BETWEEN с начальным значением, ключевое **AND** и конечное значение. В отличие от IN, BETWEEN чувствителен к порядку, и первое значение в предложении должно быть первым по алфавитному или числовому порядку. (Обратите Внимание что, в отличие от Английского языка, SQL не говорит что "*значение находится (между) BETWEEN значением и значением*", а просто "*значение BETWEEN значение AND значение*". Это применимо и к оператору **LIKE**). Следующий пример будет извлекать из таблицы Продавцов всех продавцов с комиссионными между .10 и .12 (вывод показывается в Рисунке 5.4):

```
SELECT *
FROM Salespeople
WHERE comm BETWEEN .10 AND .12;
```

Для включенного оператора **BETWEEN,** значение совпадающее с любым из двух значений границы (в этом случае, .10 и .12) заставляет предикат быть верным.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Salespeople                               |
| WHERE comm BETWEEN .10 AND .12;                |
| ===============================================|
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1001      Peel         London        0.12    |
|   1004      Motika       London        0.11    |
|   1003      Axelrod      New York      0.10    |
 ================================================
```

Рисунок 5.4: SELECT использует BETWEEN

SQL не делает непосредственной поддержки невключения BETWEEN. Вы должны или определить ваши граничные значения так, чтобы включающая интерпретация была приемлема, или сделать что-нибудь типа этого:

```
SELECT *
FROM Salespeople
WHERE (comm BETWEEN .10, AND .12) AND NOT comm IN (.10, .12);
```

Вывод для этого запроса показывается в Рисунке 5.5.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Salespeople                               |
| WHERE ( comm BETWEEN .10 AND .12               |
| AND NOT comm IN (.10 .12);                     |
| ===============================================|
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1004      Motika       London        0.11    |
 ================================================
```

Рисунок 5.5: Сделать BETWEEN — невключенным

По общему признанию, это немного неуклюже, но зато показывает как эти новые операторы могут комбинироваться с операторами Буля чтобы производить более сложные предикаты. В основном, вы используете IN и BETWEEN также как вы использовали реляционные операторы чтобы сравнивать значения, которые берутся либо из набора (для IN) либо из диапазона (для BETWEEN).

Также, подобно реляционным операторам, BETWEEN может работать с символьными полями в терминах эквивалентов ASCII. Это означает что вы можете использовать BETWEEN чтобы выбирать ряд значений из упорядоченных по алфавиту значений.

Этот запрос выбирает всех заказчиков чьи имена попали в определенный алфавитный диапазон:

```
SELECT *
FROM Customers
WHERE cname BETWEEN 'A' AND 'G';
```

Вывод для этого запроса показывается в Рисунке 5.6.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Customers                                 |
| WHERE cname BETWEEN 'A' AND 'G';               |
| ============================================== |
|   cnum     cname    city     rating    snum    |
|  ------   --------  ------    ----    ------   |
|   2006     Clemens  London     100     1001   |
|   2008     Cisneros San Jose   300     1007   |
 ================================================
```

Рисунок 5.6: Использование BETWEEN в алфавитных порядках

Обратите внимание, что Grass и Giovanni отсутствуют, даже при включенном BETWEEN. Это происходит из-за того что BETWEEN сравнивает строки неравной длины. Строка 'G' более короткая чем строка Giovanni, поэтому BETWEEN выводит 'G' с пробелами. Пробелы предшествуют символам в алфавитном порядке (в большинстве реализаций), поэтому Giovanni не выбирается. То же самое происходит с Grass. Важно помнить это когда вы используете BETWEEN для извлечения значений из алфавитных диапазонов. Обычно вы указываете диапазон с помощью символа начала диапазона и символа конца (вместо которого можно просто поставить ‘z’).

ОПЕРАТОР LIKE

LIKE применим только к полям типа CHAR или VARCHAR, с которыми он используется чтобы находить подстроки. Т.е. он ищет поле символа чтобы видеть, совпадает ли с условием часть его строки. В качестве условия он использует групповые символы (wildkards) — специальные символы которые могут соответствовать чему-нибудь. Имеются два типа групповых символов используемых с LIKE:

- символ подчеркивания (_) замещает любой одиночный символ. Например, **'b_t'** будет соответствовать словам **'bat'** или **'bit'**, но не будет соответствовать 'brat'.
- знак процента (**%**) замещает последовательность любого числа символов (включая символы нуля). Например **'%p%t'** будет соответствовать словам **'put', 'posit',** или **'opt',** но не **'spite**'.

Давайте найдем всех заказчиков чьи имена начинаются с G (вывод показывается в Рисунке 5.7):

```
SELECT
FROM Customers
WHERE cname LIKE 'G%';
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Customers                                 |
| WHERE cname LIKE 'G%';                         |
| ============================================== |
|   cnum     cname    city     rating    snum    |
|  ------   --------  ------    ----    ------   |
|   2002     Giovanni Rome       200     1003   |
|   2004     Grass    Berlin     300     1002   |
 ================================================
```

Рисунок 5.7: SELECT использует LIKE с %

LIKE может быть удобен если вы ищете имя или другое значение, и если вы не помните как они точно пишутся. Предположим что вы неуверены как записано по буквам имя одного из ваших продавцов **Peal** или **Peel.** Вы можете просто использовать ту часть которую вы знаете и групповые символы чтобы находить все возможные пары (вывод этого запроса показывается в Рисунке 5.8):

```
SELECT *
FROM Salespeople
WHERE sname LIKE 'P _ _ l %';
```

Групповые символы подчеркивания, каждый из которых представляет один символ, добавят только два символа к уже существующим 'P' и 'l', поэтому имя наподобии Prettel не может быть показано. Групповой символ '%' — в конце строки необходим в большинстве реализаций если длина поля *sname* больше чем число символов в имени Peel (потому что некоторые другие значения *sname* — длиннее чем четыре символа). В таком случае, значение поля sname, фактически сохраняемое как имя Peel, сопровождается рядом пробелов. Следовательно, символ 'l' не будет рассматриваться концом строки. Групповой символ '%' — просто соответствует этим пробелам. Это необязательно, если поля *sname* имеет тип — VARCHAR.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Salespeople                               |
| WHERE sname LIKE ' P _ _ l% ';                 |
| ============================================== |
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1001      Peel         London        0.12    |
 ================================================
```

Рисунок 5.8: SELECT использует LIKE с подчеркиванием (_)

А что же Вы будете делать если вам нужно искать знак процента или знак подчеркивания в строке? В LIKE предикате, вы можете определить любой одиночный символ как символ ESC. Символ ESC используется сразу перед процентом или подчеркиванием в предикате, и означает что процент или подчеркивание будет интерпретироваться как символ, а не как групповой символ. Например, мы могли бы найти наш *sname* столбец где присутствует подчеркивание, следующим образом:

```
SELECT *
FROM Salespeople
WHERE sname LIKE '%/_%'ESCAPE'/';
```

С этими данными не будет никакого вывода, потому что мы не включили никакого подчеркивания в имя нашего продавца. Предложение ESCAPE определяет '/' как символ ESC. Символ ESC используемый в LIKE строке, сопровождается знаком про-

цента, знаком подчеркивания, или знаком ESCAPE, который будет искаться в столбце, а не обрабатываться как групповой символ. Символ ESC должен быть одиночным символом и применяться только к одиночному символу сразу после него.

В примере выше, символ процента начала и символ процента окончания обрабатываются как групповые символы; только подчеркивание предоставлено само себе.

Как упомянуто выше, символ ESC может также использоваться самостоятельно. Другими словами, если вы будете искать столбец с вашим символом ESC, вы просто вводите его дважды. Во-первых это будет означать что символ ESC "*берет следующий символ буквально как символ*", и во-вторых что символ ESC самостоятелен.

Имеется предыдущий пример который пересмотрен чтобы искать местонахождение строки '_/' в sname столбце:

```
SELECT *
FROM Salespeople
WHERE sname LIKE ' % /_ / / %'ESCAPE'/';
```

Снова не будет никакого вывода с такими данными.

Строка сравнивается с содержанием любой последовательности символов (%), сопровождаемых символом подчеркивания (/_), символом ESC (//), и любой последовательностью символов в конце строки (%).

РАБОТА С НУЛЕВЫМИ (NULL) ЗНАЧЕНИЯМИ

Часто, будут иметься записи в таблице которые не имеют никаких значений для каждого поля, например потому что информация не завершена, или потому что это поле просто не заполнялось. SQL учитывает такой вариант, позволяя вам вводить значение **NULL** (ПУСТОЙ) в поле, вместо значения. Когда значение поля равно NULL, это означает, что программа базы данных специально промаркировала это поле как не имеющее никакого значения для этой строки (или записи). Это отличается от просто назначения полю значения нуля или пробела, которые база данных будет обрабатывать также как и любое другое значение. Точно также, как NULL не является техническим значением, оно не имеет и типа данных. Оно может помещаться в любой тип поля. Тем ни менее, NULL в SQL часто упоминается как *нуль*.

Предположим, что вы получили нового заказчика который еще не был назначен продавцу. Чем ждать продавца к которому его нужно назначить, вы можете ввести заказчика в базу данных теперь же, так что он не потеряется при перестановке. Вы можете ввести строку для заказчика со значением NULL в поле snum и заполнить это поле значением позже, когда продавец будет назначен.

NULL ОПЕРАТОР

Так как NULL указывает на отсутствие значения, вы не можете знать, каков будет результат любого сравнения с использованием NULL. Когда NULL сравнивается с любым значением, даже с другим таким же NULL, результат будет ни верным ни неверным, он — неизвестен. Неизвестный Булев, вообще ведет себя также как неверная строка, которая произведя неизвестное значение в предикате не будет выбрана запросом — имейте ввиду, что в то время как NOT (неверное) — равняется верно, NOT (неизвестное) — равняется неизвестно. Следовательно, выражение типа 'city = NULL' или 'city IN (NULL)' будет неизвестно, независимо от значения city. Часто вы должны делать различия между неверно и неизвестно — между строками содержащими значения столбцов, которые не соответствуют условию предиката и которые содержат NULL в столбцах. По этой причине, SQL предоставляет специальный оператор **IS**, ко-

торый используется с ключевым словом NULL, для размещения значения NULL. Найдем все записи в нашей таблице Заказчиков с NULL значениями в city столбце:

```
SELECT *
FROM Customers
WHERE city IS NULL;
```

Здесь не будет никакого вывода, потому что мы не имеем никаких значений NULL в наших типовых таблицах. Значения NULL — очень важны, и мы вернемся к ним позже.

ИСПОЛЬЗОВАНИЕ NOT СО СПЕЦИАЛЬНЫМИ ОПЕРАТОРАМИ

Специальные операторы которые мы изучали в этой главе могут немедленно предшествовать Булеву NOT.

Он противоположен реляционным операторам, которые должны иметь оператор NOT вводимым выражением. Например, если мы хотим устранить NULL из нашего вывода, мы будем использовать NOT, чтобы изменить на противоположное значение предиката:

```
SELECT *
FROM Customers
WHERE city NOT NULL;
```

При отсутствии значений NULL (как в нашем случае), будет выведена вся таблица Заказчиков. Аналогично можно ввести следующее:

```
SELECT *
FROM Customers
WHERE NOT city IS NULL;
```

что также приемлемо.

Мы можем также использовать NOT с **IN**:

```
SELECT *
FROM Salespeople
WHERE city NOT IN ( 'London', 'San Jose' );
```

А это — другой способ подобного же выражения:

```
SELECT *
FROM Salespeople
WHERE NOT city IN ( 'London', ' San Jose' );
```

Вывод для этого запроса показывается в Рисунке 5.9.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Salespeople                               |
| WHERE sity NOT IN ('London', 'San Jose';       |
| ===============================================|
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1003      Rifkin       Barcelona     0.15    |
|   1007      Axelrod      New York      0.10    |
 ================================================
```

Рисунок 5.9: Использование NOT с IN

Таким же способом Вы можете использовать **NOT BETWEEN** и **NOT LIKE**.

РЕЗЮМЕ

Теперь вы можете создавать предикаты в терминах связей специально определенных SQL. Вы можете искать значения в определенном диапазоне (**BETWEEN**) или в числовом наборе (**IN**), или вы можете искать символьные значения которые соответствуют тексту внутри параметров (**LIKE**).

Вы также изучили некоторые вещи относительно того, как SQL поступает при отсутствии данных — что реальность мировой базы данных — используя NULL вместо конкретных значений. Вы можете извлекать или исключать значения NULL из вашего вывода используя оператор **IS NULL**.

Теперь, когда вы имеете в вашем распоряжении весь набор стандартных математических и специальных операторов, вы можете переходить к специальным функциям SQL, которые работают на всех группах значений, а не просто на одиночном значении, что важно. Это уже тема Главы 6.

РАБОТА С SQL

1. Напишите два запроса которые могли бы вывести все порядки на 3 или 4 Октября 1990
2. Напишите запрос который выберет всех заказчиков обслуживаемых продавцами Peel или Motika. (Подсказка: из наших типовых таблиц, поле snum связывает вторую таблицу с первой)
3. Напишите запрос, который может вывести всех заказчиков, чьи имена начинаются с буквы попадающей в диапазон от A до G.
4. Напишите запрос который выберет всех пользователей чьи имена начинаются с буквы C.
5. Напишите запрос который выберет все порядки имеющие нулевые значения или NULL в поле amt (сумма).

(См. Приложение A для ответов.)

6

ОБОБЩЕНИЕ ДАННЫХ С ПОМОЩЬЮ АГРЕГАТНЫХ ФУНКЦИЙ

В ЭТОЙ ГЛАВЕ, ВЫ ПЕРЕЙДЕТЕ ОТ ПРОСТОГО использования запросов к извлечению значений из базы данных и определению, как вы можете использовать эти значения чтобы получить из них информацию. Это делается с помощью *агрегатных* или общих функций которые берут группы значений из поля и сводят их до одиночного значения. Вы узнаете как использовать эти функции, как определить группы значений к которым они будут применяться, и как определить какие группы выбираются для вывода. Вы будете также видеть при каких условиях вы сможете объединить значения поля с этой полученной информацией в одиночном запросе.

ЧТО ТАКОЕ АГРЕГАТНЫЕ ФУНКЦИИ ?

Запросы могут производить обобщенное групповое значение полей точно также как и значение одного поля. Это делает с помощью агрегатых функций. Агрегатные функции производят одиночное значение для всей группы таблицы. Имеется список этих функций:

- **COUNT** производит номера строк или не-NULL значения полей которые выбрал запрос.
- **SUM** производит арифметическую сумму всех выбранных значений данного поля.
- **AVG** производит усреднение всех выбранных значений данного поля.
- **MAX** производит наибольшее из всех выбранных значений данного поля.
- **MIN** производит наименьшее из всех выбранных значений данного поля.

КАК ИСПОЛЬЗОВАТЬ АГРЕГАТНЫЕ ФУНКЦИИ ?

Агрегатные функции используются подобно именам полей в предложении SELECT запроса, но с одним исключением, они берут имена поля как аргументы. Только числовые поля могут использоваться с SUM и AVG.

С COUNT, MAX, и MIN, могут использоваться и числовые или символьные поля. Когда они используются с символьными полями, MAX и MIN будут транслировать их в эквивалент ASCII, который должен сообщать, что MIN будет означать первое, а MAX последнее значение в алфавитном порядке (выдача алфавитного упорядочения обсуждается более подробно в Главе 4).

Чтобы найти SUM всех наших покупок в таблицы Порядков, мы можем ввести следующий запрос, с его выводом в Рисунке 6.1:

```
SELECT SUM ((amt))
FROM Orders;
```

```
 ===============  SQL Execution Log ============
| SELECT SUM (amt)                              |
| FROM Orders;                                  |
| ==============================================|
|                                               |
| -------                                       |
| 26658.4                                       |
 ===============================================
```

Рисунок 6.1: Выбор суммы

Это конечно, отличается от выбора поля, при котором возвращается одиночное значение, независимо от того, сколько строк находится в таблице. Из-за этого, агрегатные функции и поля не могут выбираться одновременно, пока предложение

GROUP BY (описанное далее) не будет использовано. Нахождение усредненой суммы — это похожая операция (вывод следующего запроса показывается в Рисунке 6.2):

```
SELECT AVG (amt)
FROM Orders;
```

```
 ===============  SQL Execution Log ============
| SELECT AVG (amt)                              |
| FROM Orders;                                  |
| ==============================================|
|                                               |
| -------                                       |
| 2665.84                                       |
 ===============================================
```

Рисунок 6.2: Выбор среднего

СПЕЦИАЛЬНЫЕ АТРИБУТЫ COUNT

Функция **COUNT** несколько отличается от всех. Она считает число значений в данном столбце, или число строк в таблице. Когда она считает значения столбца, она используется с DISTINCT, чтобы производить счет чисел различных значений в данном поле. Мы могли бы использовать ее, например, чтобы сосчитать номера продавцов в настоящее время описаных в таблице Порядков (вывод показывается в Рисунке 6.3):

```
SELECT COUNT ( DISTINCT snum )
FROM Orders;
```

ИСПОЛЬЗОВАНИЕ DISTINCT

Обратите внимание в вышеупомянутом примере, что DISTINCT, сопровождаемый именем поля с которым он применяется, помещен в круглые скобки, но не сразу после SELECT, как раньше. Этого использования DISTINCT с COUNT применяемого к индивидуальным столбцам, требует стандарт ANSI, но большое количество программ не предъявляют к ним такого требования.

```
 ===============  SQL Execution Log ============
| SELECT COUNT (DISTINCT snum)                  |
| FROM Orders;                                  |
| ==============================================|
|                                               |
| -------                                       |
|       5                                       |
 ===============================================
```

Рисунок 6.3: Подсчет значений поля

Вы можете выбирать многочисленые счета (COUNT) из полей с помощью DISTINCT в одиночном запросе который, как мы видели в Главе 3, не выполнялся когда вы выбирали строки с помощью DISTINCT. DISTINCT может использоваться таким образом, с любой функцией агрегата, но наиболее часто он используется с **COUNT**. С **MAX** и **MIN**, это просто не будет иметь никакого эффекта, а **SUM** и **AVG**, вы обычно

применяете для включения повторяемых значений, так как они законно эффективнее общих и средних значений всех столбцов.

ИСПОЛЬЗОВАНИЕ COUNT СО СТРОКАМИ, А НЕ ЗНАЧЕНИЯМИ

Чтобы подсчитать общее число строк в таблице, используйте функцию COUNT со звездочкой вместо имени поля, как например в следующем примере, вывод из которого показан на Рисунке 6.4:

```
SELECT COUNT (*)
FROM Customers
```

```
 ===============  SQL Execution Log ============
| SELECT COUNT (*)                              |
| FROM Customers;                               |
| ==============================================|
|                                               |
| -------                                       |
|       7                                       |
 ===============================================
```

Рисунок 6.4: Подсчет строк вместо значений

COUNT со звездочкой включает и NULL и дубликаты, по этой причине DISTINCT не может быть использован. DISTINCT может производить более высокие номера, чем COUNT особого поля, который удаляет все строки, имеющие избыточные или NULL данные в этом поле. DISTINCT не применим с COUNT (*), потому, что он не имеет никакого действия в хорошо разработаной и поддерживаемой базе данных. В такой базе данных, не должно быть ни таких строк, которые бы являлись полностью пустыми, ни дубликатов (первые не содержат никаких данных, а последние полностью избыточны). Если, с другой стороны, все таки имеются полностью пустые или избыточные строки, вы вероятно не захотите чтобы COUNT скрыл от вас эту информацию.

ВКЛЮЧЕНИЕ ДУБЛИКАТОВ В АГРЕГАТНЫЕ ФУНКЦИИ

Агрегатные функции могут также (в большинстве реализаций) использовать аргумент **ALL**, который помещается перед именем поля, подобно DISTINCT, но означает противоположное: — включать дубликаты. ANSI технически не позволяет этого для COUNT, но многие реализации ослабляют это ограничение.

Различия между ALL и * когда они используются с COUNT:

- ALL использует имя поля как аргумент.
- ALL не может подсчитать значения NULL.

Пока ***** является единственым аргументом который включает NULL значения, и он используется только с COUNT; функции отличные от COUNT игнорируют значения NULL в любом случае. Следующая команда подсчитает (**COUNT**) число не-NULL значений в поле rating в таблице Заказчиков (включая повторения):

```
SELECT COUNT ( ALL rating )
FROM Customers;
```

АГРЕГАТЫ ПОСТРОЕННЫЕ НА СКАЛЯРНОМ ВЫРАЖЕНИИ

До этого, вы использовали агрегатные функции с одиночными полями как аргументами. Вы можете также использовать агрегатные функции с аргументами, которые состоят из скалярных выражений, включающих одно или более полей. (Если вы это делаете, DISTINCT не разрешается.) Предположим, что таблица Порядков имеет еще один столбец который хранит предыдущий неуплаченый баланс (поле blnc) для каждого заказчика. Вы должны найти этот текущий баланс, добавлением суммы приобретений к предыдущему балансу. Вы можете найти наибольший неуплаченый баланс следующим образом:

```
SELECT MAX ( blnc + (amt) )
FROM Orders;
```

Для каждой строки таблицы, этот запрос будет складывать blnc и amt для этого заказчика и выбирать самое большое значение которое он найдет. Конечно, пока заказчики могут иметь многочисленые порядки, их неуплаченый баланс оценивается отдельно для каждого порядка. Возможно, порядок с более поздней датой будет иметь самый большой неуплаченый баланс. Иначе, старый баланс должен быть выбран как в запросе выше.

Фактически, имеются большое количество ситуаций в SQL где вы можете использовать скалярные выражения с полями или вместо полей, как вы увидете это в Главе 7.

ПРЕДЛОЖЕНИЕ GROUP BY

Предложение **GROUP BY** позволяет вам определять подмножество значений в особом поле в терминах другого поля, и применять функцию агрегата к подмножеству. Это дает вам возможность объединять поля и агрегатные функции в едином предложении SELECT. Например, предположим что вы хотите найти наибольшую сумму приобретений полученную каждым продавцом. Вы можете сделать раздельный запрос для каждого из них, выбрав MAX (amt) из таблицы Порядков для каждого значения поля snum. GROUP BY, однако, позволит Вам поместить их все в одну команду:

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum;
```

Вывод для этого запроса показывается в Рисунке 6.5.

```
 ===============  SQL Execution Log ==============
| SELECT snum, MAX (amt)                          |
| FROM Orders                                     |
| GROUP BY snum;                                  |
| =============================================== |
|  snum                                           |
|  ------   --------                              |
|   1001      767.19                              |
|   1002     1713.23                              |
|   1003       75.75                              |
|   1014     1309.95                              |
|   1007     1098.16                              |
 =================================================
```

Рисунок 6.5: Нахождение максимальной суммы продажи у каждого продавца

GROUP BY применяет агрегатные функции независимо от серий групп, которые определяются с помощью значения поля в целом. В этом случае, каждая группа состоит из всех строк с тем же самым значением поля snum, и MAX функция применяется отдельно для каждой такой группы. Это значение поля, к которому применяется GROUP BY, имеет, по определению, только одно значение на группу вывода, также как это делает агрегатная функция. Результатом является совместимость, которая позволяет агрегатам и полям объединяться таким образом. Вы можете также использовать GROUP BY с многочисленными полями. Совершенствуя вышеупомянутый пример далее, предположим что вы хотите увидеть наибольшую сумму приобретений получаемую каждым продавцом каждый день. Чтобы сделать это, вы должны сгруппировать таблицу Порядков по датам продавцов, и применить функцию MAX к каждой такой группе, подобно этому:

```
SELECT snum, odate, MAX ((amt))
FROM Orders
GROUP BY snum, odate;
```

Вывод для этого запроса показывается в Рисунке 6.6.

```
 ===============  SQL Execution Log ==============
| SELECT snum, odate, MAX (amt)                   |
| FROM Orders                                     |
| GROUP BY snum, odate;                           |
| =============================================== |
|   snum        odate                             |
|  ------     ----------     --------             |
|   1001      10/03/1990       767.19             |
|   1001      10/05/1990      4723.00             |
|   1001      10/06/1990      9891.88             |
|   1002      10/03/1990      5160.45             |
|   1002      10/04/1990        75.75             |
|   1002      10/06/1990      1309.95             |
|   1003      10/04/1990      1713.23             |
|   1014      10/03/1990      1900.10             |
|   1007      10/03/1990      1098.16             |
 =================================================
```

Рисунок 6.6: Нахождение наибольшей суммы приобретений на каждый день

Конечно же, пустые группы, в дни когда текущий продавец не имел порядков, не будут показаны в выводе.

ПРЕДЛОЖЕНИЕ HAVING

Предположим, что в предыдущем примере, вы хотели бы увидеть только максимальную сумму приобретений, значение которой выше $3000.00. Вы не сможете использовать агрегатную функцию в предложении WHERE (если вы не используете подзапрос, описанный позже), потому что предикаты оцениваются в терминах одиночной строки, а агрегатные функции оцениваются в терминах групп строк. Это означает что вы не сможете сделать что-нибудь подобно следующему:

```
SELECT snum, odate, MAX (amt)
FROM Oreders
WHERE MAX ((amt)) > 3000.00
GROUP BY snum, odate;
```

Это будет отклонением от строгой интерпретации ANSI. Чтобы увидеть максимальную стоимость приобретений свыше $3000.00, вы можете использовать предложение HAVING. Предложение HAVING определяет критерии, используемые чтобы удалять определенные группы из вывода, точно также как предложение WHERE делает это для индивидуальных строк. Правильной командой будет следующая:

```
SELECT snum, odate, MAX ((amt))
FROM Orders
GROUP BY snum, odate
HAVING MAX ((amt)) > 3000.00;
```

Вывод для этого запроса показывается в Рисунке 6.7.

```
 ==============  SQL Execution Log ==============
| SELECT snum, odate, MAX (amt)                   |
| FROM Orders                                     |
| GROUP BY snum, odate                            |
| HAVING MAX (amt) > 3000.00;                     |
| ================================================ |
|   snum        odate                             |
|  ------     ----------     --------             |
|   1001      10/05/1990      4723.00             |
|   1001      10/06/1990      9891.88             |
|   1002      10/03/1990      5160.45             |
 ================================================
```

Рисунок 6.7: Удаление групп агрегатных значений

Аргументы в предложении HAVING следуют тем же самым правилам, что и в предложении SELECT, состоящей из команд использующих GROUP BY. Они должны иметь одно значение на группу вывода. Следующая команда будет запрещена:

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum
HAVING odate = 10/03/1988;
```

Поле odate не может быть вызвано предложением HAVING, потому что оно может иметь (и действительно имеет) больше чем одно значение на группу вывода. Чтобы избегать такой ситуации, предложение HAVING должно ссылаться только на агрегаты и поля выбранные GROUP BY. Имеется правильный способ сделать вышеупомянутый запрос (вывод показывается в Рисунке 6.8):

```
SELECT snum, MAX (amt)
FROM Orders
WHERE odate = 10/03/1990
GROUP BY snum;
```

```
 ===============  SQL Execution Log ==============
| SELECT snum, odate, MAX (amt)                    |
| FROM Orders                                      |
| GROUP BY snum, odate;                            |
| ================================================ |
|   snum                                           |
|  ------     --------                             |
|   1001        767.19                             |
|   1002       5160.45                             |
|   1014       1900.10                             |
|   1007       1098.16                             |
 ================================================
```

Рисунок 6.8: Максимальное значение суммы приобретений у каждого продавца на 3 Октября

Поскольку поля odate нет, не может быть и выбраных полей, значение этих данных меньше чем в некоторых других примерах. Вывод должен вероятно включать что-нибудь такое, что говорит: "это — самые большие порядки на 3 Октября." В Главе 7, мы покажем как вставлять текст в ваш вывод.

Как и говорилось ранее, HAVING может использовать только аргументы которые имеют одно значение на группу вывода. Практически, ссылки на агрегатные функции — наиболее общие, но и поля выбранные с помощью GROUP BY также допустимы. Например, мы хотим увидеть наибольшие порядки для Serres и Rifkin:

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum
HAVING snum B (1002,1007);
```

Вывод для этого запроса показывается в Рисунке 6.9.

```
 ===============  SQL Execution Log ==============
| SELECT snum, MAX (amt)                           |
| FROM Orders                                      |
| GROUP BY snum                                    |
| HAVING snum IN (1002, 1007);                     |
| ================================================ |
|   snum                                           |
|  ------     --------                             |
|   1002       5160.45                             |
|   1007       1098.16                             |
 ================================================
```

Рисунок 6.9: Использование HAVING с GROUP BY полями

НЕ ДЕЛАЙТЕ ВЛОЖЕННЫХ АГРЕГАТОВ

В строгой интерпретации ANSI SQL, вы не можете использовать агрегат агрегата. Предположим что вы хотите выяснять, в какой день имелась наибольшая сумма приобретений. Если вы попробуете сделать это,

```
SELECT odate, MAX ( SUM (amt) )
FROM Orders
GROUP BY odate;
```

то ваша команда будет вероятно отклонена. (Некоторые реализации не предписывают этого ограничения, которое является выгодным, потому что вложенные агрегаты могут быть очень полезны, даже если они и несколько проблематичны.) В вышеупомянутой команде, например, SUM должен применяться к каждой группе поля odate, а MAX ко всем группам, производящим одиночное значение для всех групп. Однако предложение GROUP BY подразумевает что должна иметься одна строка вывода для каждой группы поля odate.

РЕЗЮМЕ

Теперь вы используете запросы несколько по-другому. Способность получать, а не просто размещать значения, очень мощна. Это означает что вы не обязательно должны следить за определенной информацией, если вы можете сформулировать запрос так, чтобы ее получить. Запрос будет давать вам поминутные результаты, в то время как таблица общего или среднего значений будет хороша только некоторое время после ее модификации. Это не должно наводить на мысль, что агрегатные функции могут полностью вытеснить потребность в отслеживании информации такой например как эта.

Вы можете применять эти агрегаты для групп значений определенных предложением GROUP BY. Эти группы имеют значение поля в целом, и могут постоянно находиться внутри других групп которые имеют значение поля в целом. В то же время, предикаты еще используются чтобы определять какие строки агрегатной функции применяются.

Объединенные вместе, эти особенности делают возможным, производить агрегаты основанные на сильно определенных подмножествах значений в поле. Затем вы можете определять другое условие для исключения определенных результатов групп с предложением HAVING.

Теперь, когда вы стали знатоком большого количества того как запрос производит значения, мы покажем вам, в Главе 7, некоторые вещи которые вы можете делать со значениями которые он производит.

РАБОТА С SQL

1. Напишите запрос, который сосчитал бы все суммы приобретений на 3 Октября.
2. Напишите запрос, который сосчитал бы число различных не-NULL значений поля city в таблице Заказчиков.
3. Напишите запрос, который выбрал бы наименьшую сумму для каждого заказчика.
4. Напишите запрос, который бы выбирал заказчиков в алфавитном порядке, чьи имена начинаются с буквы G.
5. Напишите запрос, который выбрал бы высшую оценку в каждом городе.
6. Напишите запрос, который сосчитал бы число заказчиков, регистрирующих каждый день свои порядки. (Если продавец имел более одного порядка в данный день, он должен учитываться только один раз.)

(См. Приложение A для ответов.)

7

ФОРМИРОВАНИЕ ВЫВОДОВ ЗАПРОСОВ

ЭТА ГЛАВА РАСШИРИТ ВАШИ ВОЗМОЖНОСТИ в работе с выводом который производит запрос. Вы узнаете как вставлять текст и константы между выбранных полей, как использовать выбранные поля в математических выражениях, чьи результаты затем становятся выводом, и как сделать чтобы ваши значения выводились в определенном порядке. Эта последняя особенность включена, чтобы упорядочивать ваш вывод по любым столбцам, любым полученным значениям этого столбца, или по обеим.

СТРОКИ И ВЫРАЖЕНИЯ

Большинство основанных на SQL баз данных предоставляют специальные средства позволяющие Вам совершенствовать вывод ваших запросов. Конечно, они претерпевают значительные изменения от программы к программе, и их обсуждение здесь не входит в наши задачи, однако, имеются пять особенностей созданых в стандарте SQL которые позволяют вам делать больше чем просто вывод значений полей и агрегатных данных.

СКАЛЯРНОЕ ВЫРАЖЕНИЕ С ПОМОЩЬЮ ВЫБРАННЫХ ПОЛЕЙ

Предположим что вы хотите выполнять простые числовые вычисления данных чтобы затем помещать их в форму больше соответствующую вашим потребностям. SQL позволяет вам помещать скалярные выражения и константы среди выбраных полей. Эти выражения могут дополнять или замещать поля в предложениях SELECT, и могут включать в себя одно или более выбранных полей. Например, вы можете пожелать, представить комиссионные вашего продавца в процентном отношении а не как десятичные числа. Просто достаточно:

```
SELECT snum, sname, city, comm * 100
FROM Salespeople;
```

Вывод из этого запроса показывается в Рисунке 7.1.

```
 ===============  SQL Execution Log ============
| SELECT snum, sname, city, comm * 100           |
| FROM Salespeople;                              |
| ===============================================|
|   snum      sname       city                   |
| ------    ---------  -----------   ---------   |
|   1001      Peel       London      12.000000   |
|   1002      Serres     San Jose    13.000000   |
|   1004      Motika     London      11.000000   |
|   1007      Rifkin     Barcelona   15.000000   |
|   1003      Axelrod    New York    10.000000   |
 ================================================
```

Рисунок 7.1: Помещение выражения в вашем запросе

СТОЛБЦЫ ВЫВОДА

Последний столбец предшествующего примера непомечен (т.е. без наименования), потому что это — *столбец вывода.* Столбцы вывода — это столбцы данных созданные запросом способом, иным чем просто извлечение их из таблицы. Вы создаете их всякий раз, когда вы используете агрегатные функции, константы, или выражения в предложении SELECT запроса. Так как имя столбца — один из атрибутов таблицы, столбцы которые приходят не из таблиц не имеют никаких имен. Другими словами не-

помеченные, столбцы вывода могут обрабатываться также как и столбцы извлеченные из таблиц, почти во всех ситуациях.

ПОМЕЩЕНИЕ ТЕКСТА В ВАШЕМ ВЫВОДЕ ЗАПРОСА

Символ 'A', когда ничего не значит сам по себе, — является константой, такой например как число 1. Вы можете вставлять константы в предложение SELECT запроса, включая и текст. Однако символьные константы, в отличие от числовых констант, не могут использоваться в выражениях. Вы можете иметь выражение 1 + 2 в вашем предложении SELECT, но вы не можете использовать выражение типа 'A' + 'B'; это приемлемо только если мы имеем в виду что 'A' и 'B' это просто буквы, а не переменные и не символы.

Тем ни менее, возможность вставлять текст в вывод ваших запросов очень удобная штука.

Вы можете усовершенствовать предыдущий пример представив комиссионные как проценты со знаком процента (%). Это даст вам возможность помещать в вывод такие единицы как символы и комментарии, как например в следующем примере (вывод показывается в Рисунке 7.2)

```
SELECT snum, sname, city, ' % ', comm * 100
FROM Salespeople;
```

```
 ===============  SQL Execution Log ============
| SELECT snum, sname, city, '%', comm * 100     |
| FROM Salespeople;                             |
| ==============================================|
|   snum   sname      city                      |
| ------  -------- -----------  ----  --------- |
|   1001   Peel      London       %   12.000000 |
|   1002   Serres    San Jose     %   13.000000 |
|   1004   Motika    London       %   11.000000 |
|   1007   Rifkin    Barcelona    %   15.000000 |
|   1003   Axelrod   New York     %   10.000000 |
 ==============================================
```

Рисунок 7.2: Вставка символов в ваш вывод

Обратите внимание что пробел перед процентом вставляется как часть строки. Эта же самая особенность может использоваться чтобы маркировать вывод вместе с вставляемыми комментариями. Вы должны помнить, что этот же самый комментарий будет напечатан в каждой строке вывода, а не просто один раз для всей таблицы. Предположим что вы генерируете вывод для отчета который бы указывал число порядков получаемых в течение каждого дня. Вы можете промаркировать ваш вывод (см. Рисунок 7.3) сформировав запрос следующим образом:

```
SELECT ' For ', odate, ', there are ',
COUNT ( DISTINCT onum ), 'orders.'
FROM Orders
GROUP BY odate;
```

```
 ===============  SQL Execution Log ==============
| SELECT 'For', odate, ', ' there are ' ,          |
| COUNT (DISTINCT onum), ' orders '                |
| FROM Orders                                      |
| GROUP BY odate;                                  |
| ================================================ |
|           odate                                  |
| ------  ----------   ---------  ------  -------  |
|   For   10/03/1990 , there are       5  orders.  |
|   For   10/04/1990 , there are       2  orders.  |
|   For   10/05/1990 , there are       1  orders.  |
|   For   10/06/1990 , there are       2  orders.  |
 ================================================
```

Рисунок 7.3: Комбинация текста, значений поля, и агрегатов

Грамматической некорректности вывода, на 5 Октября, невозможно избежать не создав запроса, еще более сложного чем этот. (Вы будете должны использовать два запроса с UNION, который мы будем описывать в Главе 14.) Как вы можете видеть, одиночный неизменный комментарий для каждой строки таблицы может быть очень полезен, но имеет ограничения. Иногда изящнее и полезнее, произвести один комментарий для всего вывода в целом, или производить свой собственный комментарии для каждой строки.

Различные программы использующие SQL часто обеспечивают специальные средства типа генератора отчетов (например Report Writer), которые разработаны чтобы форматировать и совершенствовать вывод. Вложенный SQL может также эксплуатировать возможности того языка в который он вложен. SQL сам по себе интересен прежде всего при операциях с данными. Вывод, по существу, это информация, и программа использующая SQL может часто использовать эту информацию и помещать ее в более привлекательную форму. Это, однако, вне сферы самой SQL.

УПОРЯДОЧЕНИЕ ВЫВОДА ПОЛЕЙ

Как мы подчеркивали, таблицы — это неупорядоченные наборы данных, и данные которе выходят из их, не обязательно появляются в какой-то определенной последовательности. SQL использует команду ORDER BY чтобы позволять вам упорядочивать ваш вывод. Эта команда упорядочивает вывод запроса согласно значениям в том или ином количестве выбранных столбцов. Многочисленые столбцы упорядочиваются один внутри другого, также как с GROUP BY, и вы можете определять возрастание (ASC) или убывание (DESC) для каждого столбца. По умолчанию установлено — возрастание. Давайте рассмотрим нашу таблицу порядка приводимую в порядок с помощью номера заказчика (обратите внимание на значения в cnum столбце):

```
SELECT *
FROM Orders
ORDER BY cnum DESC;
```

Вывод показывается в Рисунке 7.4.

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM Orders                                     |
| ORDER BY cnum DESC;                             |
| ================================================ |
|   onum       amt      odate      cnum     snum  |
|  ------   --------  ----------  -----    -----  |
|   3001       18.69  10/03/1990   2008     1007  |
|   3006     1098.16  10/03/1990   2008     1007  |
|   3002     1900.10  10/03/1990   2007     1004  |
|   3008     4723.00  10/05/1990   2006     1001  |
|   3011     9891.88  10/06/1990   2006     1001  |
|   3007       75.75  10/04/1990   2004     1002  |
|   3010     1309.95  10/06/1990   2004     1002  |
|   3005     5160.45  10/03/1990   2003     1002  |
|   3009     1713.23  10/04/1990   2002     1003  |
|   3003      767.19  10/03/1990   2001     1001  |
 ================================================
```

Рисунок 7.4: Упорядочение вывода с помощью убывания поля

УПОРЯДОЧЕНИЕ С ПОМОЩЬЮ МНОГОЧИСЛЕНЫХ СТОЛБЦОВ

Мы можем также упорядочивать таблицу с помощью другого столбца, например с помощью поля amt, внутри упорядочения поля cnum. (вывод показан в Рисунке 7.5):

```
SELECT *
FROM Orders
ORDER BY cnum DESC, amt DESC;
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM Orders                                     |
| ORDER BY cnum DESC, amt DESC;                   |
| ================================================ |
|   onum       amt      odate      cnum     snum  |
|  ------   --------  ----------  -----    -----  |
|   3006     1098.16  10/03/1990   2008     1007  |
|   3001       18.69  10/03/1990   2008     1007  |
|   3002     1900.10  10/03/1990   2007     1004  |
|   3011     9891.88  10/06/1990   2006     1001  |
|   3008     4723.00  10/05/1990   2006     1001  |
|   3010     1309.95  10/06/1990   2004     1002  |
|   3007       75.75  10/04/1990   2004     1002  |
|   3005     5160.45  10/03/1990   2003     1002  |
|   3009     1713.23  10/04/1990   2002     1003  |
|   3003      767.19  10/03/1990   2001     1001  |
 ================================================
```

Рисунок 7.5: Упорядочение вывода с помощью многочисленых полей

Вы можете использовать ORDER BY таким же способом сразу с любым числом столбцов. Обратите внимание что, во всех случаях, столбцы которые упорядочиваются должны быть указаны в выборе SELECT. Это — требование ANSI которые в большинстве, но не всегда, предписано системе. Следующая команда, например, будет запрещена:

```
SELECT cname, city
FROM Customers
GROUP BY cnum;
```

Так как поле cnum не было выбранным полем, GROUP BY не сможет найти его чтобы использовать для упорядочения вывода. Даже если ваша система позволяет это, смысл упорядочения не будет понятен из вывода, так что включение (в предложение SELECT) всех столбцов, используемых в предложении ORDER BY, в принципе желательно.

УПОРЯДОЧЕНИЕ АГРЕГАТНЫХ ГРУПП

ORDER BY может кроме того, использоваться с GROUP BY для упорядочения групп. Если это так, то ORDER BY всегда приходит последним. Вот — пример из последней главы с добавлением предложения ORDER BY. Перед сгруппированием вывода, порядок групп был произвольным, и мы, теперь, заставим группы размещаться в последовательности:

```
SELECT snum, odate, MAX (amt)
FROM Orders
GROUP BY snum, odate
GROUP BY snum;
```

Вывод показывается в Рисунке 7.6.

```
 ===============  SQL Execution Log ==============
| SELECT snum, odate, MAX (amt)                    |
| FROM Orders                                      |
| GROUP BY snum, odate                             |
| ORDER BY snum ;                                  |
| ================================================ |
|    snum       odate        amt                   |
|   -----     ----------  --------                 |
|    1001     10/06/1990    767.19                 |
|    1001     10/05/1990   4723.00                 |
|    1001     10/05/1990   9891.88                 |
|    1002     10/06/1990   5160.45                 |
|    1002     10/04/1990     75.75                 |
|    1002     10/03/1990   1309.95                 |
|    1003     10/04/1990   1713.23                 |
|    1004     10/03/1990   1900.10                 |
|    1007     10/03/1990   1098.16                 |
 ================================================
```

Рисунок 7.6: Упорядочение с помощью группы

Так как мы не указывали на возрастание или убывание порядка, возрастание используется по умолчанию.

УПОРЯДОЧЕНИЕ ВЫВОДА ПО НОМЕРУ СТОЛБЦА

Вместо имен столбца, вы можете использовать их порядковые номера для указания поля используемого в упорядочении вывода. Эти номера могут ссылаться не на порядок столбцов в таблице, а на их порядок в выводе. Другими словами, поле упомянутое в предложении SELECT первым, для ORDER BY — это поле **1**, независимо от того каким по порядку оно стоит в таблице. Например, вы можете использовать следующую команду чтобы увидеть определенные поля таблицы Продавцов, упорядоченными в порядке убывания к наименьшему значению комиссионных (вывод показывается Рисунке 7.7):

```
SELECT sname, comm
FROM Salespeople
GROUP BY 2 DESC;
```

```
 ===============  SQL Execution Log ============
| (SELECT sname, comm                           |
| FROM Salespeople                              |
| ORDER BY 2 DESC;                              |
| ============================================= |
|    sname       comm                           |
|   --------   --------                         |
|   Peel           0.17                         |
|   Serres         0.13                         |
|   Rifkin         0.15                         |
 ===============================================
```

Рисунок 7.7: Упорядочение использующее номера

Одна из основных целей этой возможности ORDER BY — дать вам возможность использовать GROUP BY со столбцами вывода также как и со столбцами таблицы. Столбцы производимые агрегатной функцией, константы, или выражения в предложении SELECT запроса, абсолютнопригодны для использования с GROUP BY, если они ссылаются к ним с помощью номера. Например, давайте сосчитаем порядки каждого из наших продавцов, и выведем результаты в убывающем порядке, как показано в Рисунке 7.8:

```
SELECT snum, COUNT ( DISTINCT onum )
FROM Orders
GROUP BY snum
ORDER BY 2 DESC;
```

```
 ===============  SQL Execution Log ==============
| SELECT snum, odate, MAX (amt)                    |
| FROM Orders                                      |
| GROUP BY snum                                    |
| ORDER BY 2 DESC;                                 |
| ================================================ |
|    snum                                          |
|   -----     ----------                           |
|    1001             3                            |
|    1002             3                            |
|    1007             2                            |
|    1003             1                            |
|    1004             1                            |
 ==================================================
```

Рисунок 7.8: Упорядочение с помощью столбца вывода

В этом случае, вы должны использовать номер столбца, так как столбец вывода не имеет имени; и вы не должны использовать саму агрегатную функцию. Строго говоря по правилам ANSI SQL, следующее не будет работать, хотя некоторые системы и пренебрегают этим требованием:

```
SELECT snum, COUNT ( DISTINCT onum )
FROM Orders
GROUP BY snum
GROUP BY COUNTом ( DISTINCT onum ) DESC;
```

Это будет отклонено большинством систем!

УПОРЯДОЧЕНИЕ С ПОМОЩЬЮ ОПЕРАТОРА NULL

Если имеются пустые значения (NULL) в поле которое вы используете для упорядочивания вашего вывода, они могутут или следовать или предшествовать каждому

другому значению в поле. Это — возможность которую ANSI оставил для индивидуальных программ. Данная программа использует ту или иную форму.

РЕЗЮМЕ

В этой главе, вы изучили как заставить ваши запросы делать больше, чем просто выводить значения полей или объединять функциональные данные таблиц. Вы можете использовать поля в выражениях: например, вы можете умножить числовое поле на 10 или даже умножить его на другое числовое поле. Кроме того, вы можете помещать константы, включая и символы, в ваш вывод, что позволяет вам помещать текст непосредственно в запрос и получать его в выводе вместе с данными таблицы.

Это дает вам возможность помечать или объяснять ваш вывод различными способами.

Вы также изучили как упорядочивать ваш вывод. Даже если таблица сама по себе остается неупорядоченной, предложение ORDER BY дает вам возможность управлять порядком вывода строк данного запроса. Вывод запроса может быть в порядке возрастания или убывания, и столбцы могут быть вложенными один внутрь другого.

Понятие выводимых столбцов объяснялось в этой главе. Вы теперь знаете что выводимые столбцы можно использовать чтобы упорядочивать вывод запроса, но эти столбцы — без имени, и следовательно должны определяться их порядковым номером в предложении ORDER BY.

Теперь, когда вы увидели что можно делать с выводом запроса основанного на одиночной таблице, настало время чтобы перейти к возможностям улучшенного запроса и узнать как сделать запрос любого числа таблиц в одной команде, определив связи между ними как вы это обычно делали. Это будет темой Главы 8.

РАБОТА С SQL

1. Предположим что каждый продавец имеет 12% комиссионных. Напишите запрос к таблице Порядков который мог бы вывести номер порядка, номер продавца, и сумму комиссионных продавца для этого порядка.
2. Напишите запрос к таблице Заказчиков который мог бы найти высшую оценку в каждом городе. Вывод должен быть в такой форме:

   For the city (**city**), *the highest rating is*: (**rating**).

3. Напишите запрос который выводил бы список заказчиков в нисходящем порядке. Вывод поля оценки (rating) должден сопровождаться именем закзчика и его номером.
4. Напишите запрос который бы выводил общие порядки на каждый день и помещал результаты в нисходящем порядке.

   (См. Приложение A для ответов.)

8

ЗАПРАШИВАНИЕ МНОГОЧИСЛЕНЫХ ТАБЛИЦ ТАК ЖЕ, КАК ОДНОЙ

ДО ЭТОГО, КАЖДЫЙ ЗАПРОС КОТОРЫЙ МЫ ИССЛЕДОВАЛИ основывался на одиночной таблице. В этой главе, вы узнаете как сделать запрос любого числа таблиц с помощью одной команды. Это — чрезвычайно мощное средство потому что оно не только объединяет вывод из многочисленых таблиц, но и определяет связи между ними. Вы обучитесь различным формам которые могут использовать эти связи, а также устанавливать и использовать их чтобы удовлетворять возможным специальным требованиям.

ОБЪЕДИНЕНИЕ ТАБЛИЦ

Одна из наиболее важных особенностей запросов SQL — это их способность определять связи между многочислеными таблицами и выводить информацию из них в терминах этих связей, всю внутри одной команды.

Этот вид операции называется — *объединением*, которое является одним из видов операций в реляционных базах данных. Как установлено в

Главе 1, главное в реляционном подходе это связи которые можно создавать между позициями данных в таблицах. Используя обьединения, мы непосредственно связываем информацию с любым номером таблицы, и таким образом способны создавать связи между сравнимыми фрагментами данных.

При обьединении, таблицы представленые списком в предложении FROM запроса, отделяются запятыми. Предикат запроса может ссылаться к любому столбцу любой связанной таблицы и, следовательно, может использоваться для связи между ими. Обычно, предикат сравнивает значения в столбцах различных таблиц чтобы определить, удовлетворяет ли WHERE установленному условию.

ИМЕНА ТАБЛИЦ И СТОЛБЦОВ

Полное имя столбца таблицы фактически состоит из имени таблицы, сопровождаемого точкой и затем именем столбца. Имеются несколько примеров имен :

```
Salespeople.snum
Salespeople.city
Orders.odate
```

До этого, вы могли опускать имена таблиц потому что вы запрашивали только одну таблицу одновременно, а SQL достаточно интелектуален чтобы присвоить соответствующий префикс, имени таблицы. Даже когда вы делаете запрос многочисленых таблиц, вы еще можете опускать имена таблиц, если все ее столбцы имеют различные имена. Но это не всегда так бывает. Например, мы имеем две типовые таблицы со столбцами называемыми city.

Если мы должны связать эти столбцы (кратковременно), мы будем должны указать их с именами Salespeople.city или Customers.city, чтобы SQL мог их различать.

СОЗДАНИЕ ОБЬЕДИНЕНИЯ

Предположим что вы хотите поставить в соответствии вашему продавцу ваших заказчиков в городе в котором они живут, поэтому вы увидите все комбинации продавцов и заказчиков для этого города. Вы будете должны брать каждого продавца и искать в таблице Заказчиков всех заказчиков того же самого города. Вы могли бы сделать это, введя следующую команду (вывод показывается в Рисунке 8.1):

```
SELECT Customers.cname, Salespeople.sname, Salespeople.city
FROM Salespeople, Customers
WHERE Salespeople.city = Customers.city;
```

```
 ===============  SQL Execution Log ============
| SELECT Customers.cname, Salespeople.sname,    |
| Salespeople.city                              |
| FROM Salespeople, Customers                   |
| WHERE Salespeople.city = Customers.city       |
| ============================================= |
|   cname       cname            city           |
|  -------     --------          ----           |
|  Hoffman     Peel              London         |
|  Hoffman     Peel              London         |
|  Liu         Serres            San Jose       |
|  Cisneros    Serres            San Jose       |
|  Hoffman     Motika            London         |
|  Clemens     Motika            London         |
 ============================================
```

Рисунок 8.1: Объединение двух таблиц

Так как это поле city имеется и в таблице Продавцов и таблице Заказчиков, имена таблиц должны использоваться как префиксы. Хотя это необходимо только когда два или более полей имеют одно и то же имя, в любом случае это хорошая идея включать имя таблицы в обьединение для лучшего понимания и непротиворечивости. Несмотря на это, мы будем, в наших примерах далее, использовать имена таблицы только когда необходимо, так что будет ясно, когда они необходимы а когда нет.

Что SQL в основном делает в обьединении — так это исследует каждую комбинацию строк двух или более возможных таблиц, и проверяет эти комбинации по их предикатам. В предыдущем примере, требовалась строка продавца Peel из таблицы Продавцов и объединение ее с каждой строкой таблицы Пользователей, по одной в каждый момент времени.

Если комбинация производит значение которое делает предикат верным, и если поле city из строк таблиц Заказчика равно London, то Peel — это то запрашиваемое значение которое комбинация выберет для вывода. То же самое будет затем выполнено для каждого продавца в таблице Продавцов (у некоторых из которых небыло никаких заказчиков в этих городах).

ОБЪЕДИНЕНИЕ ТАБЛИЦ ЧЕРЕЗ СПРАВОЧНУЮ ЦЕЛОСТНОСТЬ

Эта особенность часто используется просто для эксплуатации связей встроенных в базу данных. В предыдущем примере, мы установили связь между двумя таблицами в обьединении. Это прекрасно. Но эти таблицы, уже были соединены через snum поле. Эта связь называется *состоянием справочной целостности*, как мы уже говорили в Главе 1. Используя обьединение можно извлекать данные в терминах этой связи. Например, чтобы показать имена всех заказчиков соответствующих продавцам которые их обслуживают, мы будем использовать такой запрос:

```
SELECT Customers.cname, Salespeople.sname
FROM Customers, Salespeople
WHERE Salespeople.snum = Customers.snum;
```

Вывод этого запроса показывается в Рисунке 8.2.

Это — пример обьединения, в котором столбцы используются для определения предиката запроса, и в этом случае, snum столбцы из обеих таблиц, удалены из вывода. И это прекрасно.

Вывод показывает какие заказчики каким продавцом обслуживаются; значения поля snum которые устанавливают связь — отсутствуют. Однако если вы введете их в вывод, то вы должны или удостовериться что вывод понятен сам по себе или обеспечить коментарий к данным при выводе.

```
 ===============  SQL Execution Log ============
| SELECT Customers.cname, Salespeople.sname,     |
| FROM Salespeople, Customers                    |
| WHERE Salespeople.snum = Customers.snum        |
| ============================================== |
|   cname       sname                            |
|  -------     --------                          |
|  Hoffman     Peel                              |
|  Giovanni    Axelrod                           |
|  Liu         Serres                            |
|  Grass       Serres                            |
|  Clemens     Peel                              |
|  Cisneros    Rifkin                            |
|  Pereira     Motika                            |
 ================================================
```

Рисунок 8.2: Объединение продавцов с их заказчикам

ОБЪЕДИНЕНИЯ ТАБЛИЦ ПО РАВЕНСТВУ ЗНАЧЕНИЙ В СТОЛБЦАХ И ДРУГИЕ ВИДЫ ОБЪЕДИНЕНИЙ

Обьединения которые используют предикаты основанные на равенствах называются — *объединениями по равенству*. Все наши примеры в этой главе до настоящего времени, относились именно к этой категории, потому что все условия в предложениях WHERE базировались на математических выражениях использующих знак равно (=). Строки **'city = 'London'** и **'Salespeople.snum = Orders.snum'** — примеры таких типов равенств найденных в предикатах.

Объединения по равенству — это вероятно наиболее общий вид объединения, но имеются и другие. Вы можете, фактически, использовать любой из реляционных операторов в обьединении. Здесь показан пример другого вида объединения (вывод показывается в Рисунке 8.3):

```
SELECT sname, cname
FROM Salespeople, Customers
WHERE sname < cname AND rating < 200;
```

```
 ===============  SQL Execution Log ============
| SELECT sname, cname                            |
| FROM Salespeople, Customers                    |
| WHERE sname < cname                            |
| AND rating < 200;                              |
| ============================================== |
|     sname       cname                          |
|    --------    -------                         |
|    Peel        Pereira                         |
|    Motika      Pereira                         |
|    Axelrod     Hoffman                         |
|    Axelrod     Clemens                         |
|    Axelrod     Pereira                         |
 ================================================
```

Рисунок 8.3: Обьединение основанное на неравенстве

Эта команда не часто бывает полезна. Она воспроизводит все комбинации имени продавца и имени заказчика так, что первый предшествует последнему в алфавитном порядке, а последний имеет оценку меньше чем 200. Обычно, вы не создаете сложных связей подобно этой, и, по этой причине, вы вероятно будете строить наиболее общие *объединения по равенству,* но вы должны хорошо знать и другие возможности.

ОБЪЕДИНЕНИЕ БОЛЕЕ ДВУХ ТАБЛИЦ

Вы можете также создавать запросы объединяющие более двух таблиц. Предположим что мы хотим найти все порядки заказчиков не находящихся в тех городах где находятся их продавцы. Для этого необходимо связать все три наши типовые таблицы (вывод показывается в Рисунке 8.4):

```
SELECT onum, cname, Orders.cnum, Orders.snum
FROM Salespeople, Customers,Orders
WHERE Customers.city < > Salespeople.city
      AND Orders.cnum = Customers.cnum
      AND Orders.snum = Salespeople.snum;
```

```
 ===============  SQL Execution Log ==============
| SELECT onum, cname, Orders.cnum, Orders.snum     |
| FROM Salespeople, Customers, Orders              |
| WHERE Customers.city < > Salespeople.city        |
| AND Orders.cnum = Customers.cnum                 |
| AND Orders.snum = Salespeople.snum;              |
| ================================================ |
|   onum    cname        cnum     snum             |
|  ------  -------      -----    -----             |
|   3001   Cisneros      2008     1007             |
|   3002   Pereira       2007     1004             |
|   3006   Cisneros      2008     1007             |
|   3009   Giovanni      2002     1003             |
|   3007   Grass         2004     1002             |
|   3010   Grass         2004     1002             |
 ================================================
```

Рисунок 8.4: Объединение трех таблиц

Хотя эта команда выглядит скорее как комплексная, вы можете следовать за логикой, просто проверяя — что заказчики не размещены в тех городах где размещены их продавцы (совпадение двух snum полей), и что перечисленные порядки — выпол-

нены с помощью этих заказчиков (совпадение порядков с полями cnum и snum в таблице Порядков).

РЕЗЮМЕ

Теперь вы больше не ограничиваетесь просмотром одной таблицы в каждый момент времени. Кроме того, вы можете делать сложные сравнения между любыми полями любого числа таблиц и использовать полученные результаты чтобы решать какую информацию вы бы хотели видеть. Фактически, эта методика настолько полезна для построения связей, что она часто используется для создания их внутри одиночной таблицы. Это будет правильным: вы сможете объединить таблицу с собой, а это очень удобная вещь. Это будет темой Главы 9.

РАБОТА С SQL

1. Напишите запрос, который бы вывел список номеров порядков, сопровождающихся именем заказчика, который создавал эти порядки.
2. Напишите запрос, который бы выдавал имена продавца и заказчика для каждого порядка после номера порядков.
3. Напишите запрос, который бы выводил всех заказчиков, обслуживаемых продавцом с комиссионными выше 12%. Выведите имя заказчика, имя продавца и ставку комиссионных продавца.
4. Напишите запрос, который вычислил бы сумму комиссионных продавца для каждого порядка заказчика с оценкой выше 100.

(См. Приложение A для ответов.)

9

ОБЪЕДИНЕНИЕ ТАБЛИЦЫ С СОБОЙ

В ГЛАВЕ 8 МЫ ПОКАЗАЛИ ВАМ, КАК ОБЪЕДИНЯТЬ ДВЕ или более таблиц — вместе.

Достаточно интересно то, что та же самая методика может использоваться чтобы объединять вместе две копии одиночной таблицы.

В этой главе, мы будем исследовать этот процесс. Как вы видете, объединение таблицы с самой собой, далеко не простая вещь, и может быть очень полезным способом определять определенные виды связей между пунктами данных в конкретной таблице.

КАК ДЕЛАТЬ ОБЪЕДИНЕНИЕ ТАБЛИЦЫ С СОБОЙ ?

Для объединения таблицы с собой, вы можете сделать каждую строку таблицы, одновременно, и комбинацией ее с собой и комбинацией с каждой другой строкой таблицы. Вы затем оцениваете каждую комбинацию в терминах предиката, также как в обьединениях мультитаблиц. Это позволит вам легко создавать определенные виды связей между различными позициями внутри одиночной таблицы — с помощью обнаружения пар строк со значением поля, например.

Вы можете изобразить обьединение таблицы с собой, как обьединение двух копий одной и той же таблицы. Таблица на самом деле не копируется, но SQL выполняет команду так, как если бы это было сделано.

Другими словами, это обьединение — такое же, как и любое другое обьединение между двумя таблицами, за исключением того, что в данном случае обе таблицы идентичны.

ПСЕВДОНИМЫ

Синтаксис команды для объединения таблицы с собой, тот же что и для объединения многочисленых таблиц, в одном экземпляре.

Когда вы объединяете таблицу с собой, все повторяемые имена столбца, заполняются префиксами имени таблицы. Чтобы ссылаться к этим столбцам внутри запроса, вы должны иметь два различных имени для этой таблицы.

Вы можете сделать это с помощью определения временных имен называемых — *переменными диапазона*, *переменными корреляции* или просто — ***псевдонимами.***

Вы определяете их в предложении FROM запроса. Это очень просто: вы набираете имя таблицы, оставляете пробел, и затем набираете псевдоним для нее.

Имеется пример, который находит все пары заказчиков имеющих один и тот же самый рейтинг (вывод показывается в Рисунке 9.1):

```
SELECT first.cname, second.cname, first.rating
FROM Customers first, Customers second
WHERE first.rating = second.rating;
```

```
 ===============  SQL Execution Log ==============
|    Giovanni     Giovanni                    200   |
|    Giovanni     Liu                         200   |
|    Liu          Giovanni                    200   |
|    Liu          Liu                         200   |
|    Grass        Grass                       300   |
|    Grass        Cisneros                    300   |
|    Clemens      Hoffman                     100   |
|    Clemens      Clemens                     100   |
|    Clemens      Pereira                     100   |
|    Cisneros     Grass                       300   |
|    Cisneros     Cisneros                    300   |
|    Pereira      Hoffman                     100   |
|    Pereira      Clemens                     100   |
|    Pereira      Pereira                     100   |
 ================================================
```

Рисунок 9.1: Объединение таблицы с собой

(Обратите внимание что на Рисунке 9.1, как и в некоторых дальнейших примерах, полный запрос не может уместиться в окне вывода, и следовательно будет усекаться.)

В вышеупомянутой команде, SQL ведет себя так, как если бы он соединял две таблицы называемые *'первая'* и *'вторая'*. Обе они — фактически, таблицы Заказчика, но псевдонимы разрешают им быть обработаными независимо. Псевдонимы первый и второй были установлены в предложении FROM запроса, сразу после имени копии таблицы.

Обратите внимание что псевдонимы могут использоваться в предложении SELECT, даже если они не определены в предложении FROM.

Это — очень хорошо. SQL будет сначала допускать любые такие псевдонимы на веру, но будет отклонять команду если они не определены далее в предложении FROM запроса.

Псевдоним существует только пока команда выполняется. Когда запрос заканчивается, псевдонимы, используемые в нем, больше не имеют никакого значения.

Теперь, когда имеются две копии таблицы Заказчиков, чтобы работать с ними, SQL может обрабатывать эту операцию точно также, как и любое другое обьединение — берет каждую строку из одного псевдонима и сравнивает ее с каждой строкой из другого псевдонима.

УСТРАНЕНИЕ ИЗБЫТОЧНОСТИ

Обратите внимание, что наш вывод имеет два значение для каждой комбинации, причем второй раз в обратном порядке. Это потому, что каждое значение показано первый раз в каждом псевдониме, и второй раз (симметрично) в предикате.

Следовательно, значение **A** в псевдониме сначала выбирается в комбинации со значением **B** во втором псевдониме, а затем значение **A** во втором псевдониме выбирается в комбинации со значением **B** в первом псевдониме. В нашем примере, Hoffman выбрался вместе с Clemens, а затем Clemens выбрался вместе с Hoffman. Тот же самый случай с Cisneros и Grass, Liu и Giovanni, и так далее. Кроме того каждая строка была сравнена сама с собой, чтобы вывести строки такие как — Liu и Liu.

Простой способ избежать этого состоит в том, чтобы налагать порядок на два значения, так чтобы один мог быть меньше чем другой или предшествовал ему в алфавитном порядке. Это делает предикат ассиметричным, поэтому те же самые значения в обратном порядке не будут выбраны снова, например:

```
SELECT tirst.cname, second.cname, first.rating
FROM Customers first, Customers second
WHERE first.rating = second.rating
      AND first.cname < second.cname;
```

Вывод этого запроса показывается в Рисунке 9.2.

```
 ===============  SQL Execution Log ==============
| SELECT first.cname, second.cname, first.rating  |
| FROM Customers first, Customers second          |
| WHERE first.rating = second.rating              |
| AND first.cname < second.cname                  |
| =============================================== |
|   cname      cname     rating                   |
|  -------  ---------   -------                   |
|  Hoffman    Pereira       100                   |
|  Giovanni   Liu           200                   |
|  Clemens    Hoffman       100                   |
|  Pereira    Pereira       100                   |
|  Gisneros   Grass         300                   |
 ================================================
```

Рисунок 9.2: Устранение избыточности вывода в обьединении с собой.

Hoffman предшествует Periera в алфавитном порядке, поэтому комбинация удовлетворяет обеим условиям предиката и появляется в выводе. Когда та же самая комбинация появляется в обратном порядке — когда Periera в псевдониме первой таблицы сравнтвается с Hoffman во второй таблице псевдонима — второе условие не встречается. Аналогично, Hoffman не выбирается при наличии того же рейтинга что и он сам потому что его имя не предшествует ему самому в алфавитном порядке. Если бы вы захотели включить сравнение строк с ними же в запросах подобно этому, вы могли бы просто использовать **<=** вместо **<**.

ПРОВЕРКА ОШИБОК

Таким образом мы можем использовать эту особенность SQL для проверки определенных видов ошибок. При просмотре таблицы Порядков, вы можете видеть что поля cnum и snum должны иметь постоянную связь.

Так как каждый заказчик должен быть назначен к одному и только одному продавцу, каждый раз когда определенный номер заказчика появляется в таблице Порядков, он должен совпадать с таким же номером продавца. Следующая команда будет определять любые несогласованности в этой области:

```
SELECT first.onum, first.cnum, first.snum,
second.onum, second.cnum, second.snum
FROM Orders first, Orders second
WHERE first.cnum = second.cnum
AND first.snum < > second.snum;
```

Хотя это выглядит сложно, логика этой команды достаточно проста. Она будет брать первую строку таблицы Порядков, запоминать ее под первым псевдонимом, и проверять ее в комбинации с каждой строкой таблицы Порядков под вторым псевдонимом, одну за другой. Если комбинация строк удовлетворяет предикату, она выбирается для вывода. В этом случае предикат будет рассматривать эту строку, найдет строку где поле cnum=2008 а поле snum=1007, и затем рассмотрит каждую следующую строку с тем же самым значением поля cnum. Если он находит что какая-то из них имеет значение отличное от значения поля snum, предикат будет верен, и выве-

дет выбранные поля из текущей комбинации строк. Если же значение snum с данным значением cnum в наш таблице совпадает, эта команда не произведет никакого вывода.

БОЛЬШЕ ПСЕВДОНИМОВ

Хотя обьединение таблицы с собой — это первая ситуация, когда понятно, что псевдонимы необходимы, вы не ограничены в их использовании что бы только отличать копию одной таблицы от ее оригинала. Вы можете использовать псевдонимы в любое время, когда вы хотите создать альтернативные имена для ваших таблиц в команде. Например, если ваши таблицы имеют очень длинные и сложные имена, вы могли бы определить простые односимвольные псевдонимы, типа a и b, и использовать их вместо имен таблицы в предложении SELECT и предикате. Они будут также использоваться с соотнесенными подзапросами (обсуждаемыми в Главе 11).

ЕЩЕ БОЛЬШЕ КОМПЛЕКСНЫХ ОБЪЕДИНЕНИЙ

Вы можете использовать любое число псевдонимов для одной таблицы в запросе, хотя использование более двух в данном предложении SELECT * будет излишеством.

Предположим что вы еще не назначили ваших заказчиков к вашему продавцу. Компания должна назначить каждому продавцу первоначально трех заказчиков, по одному для каждого рейтингового значения. Вы лично можете решить, какого заказчика какому продавцу назначить, но следующий запрос вы используете, чтобы увидеть все возможные комбинации заказчиков, которых вы можете назначать.

(Вывод показывается в Рисунке 9.3):

```
SELECT a.cnum, b.cnum, c.cnum
FROM Customers a, Customers b, Customers c
WHERE a.rating = 100 AND b.rating = 200 AND c.rating = 300;
```

```
 ===============  SQL Execution Log ==============
|   cnum       cnum        cnum                    |
|  -----      ------      ------                   |
|   2001       2002        2004                    |
|   2001       2002        2008                    |
|   2001       2003        2004                    |
|   2001       2003        2008                    |
|   2006       2002        2004                    |
|   2006       2002        2008                    |
|   2006       2003        2004                    |
|   2006       2003        2008                    |
|   2007       2002        2004                    |
|   2007       2002        2008                    |
|   2007       2003        2004                    |
|   2007       2003        2008                    |
 ================================================
```

Рисунок 9.3: Комбинация пользователей с различными значениями рейтинга

Как вы можете видеть, этот запрос находит все комбинации заказчиков с тремя значениями оценки, поэтому первый столбец состоит из заказчиков с оценкой 100, второй с 200, и последний с оценкой 300. Они повторяются во всех возможных комбинациях. Это — сортировка группировки которая не может быть выполнена с GROUP

BY или ORDER BY, поскольку они сравнивают значения только в одном столбце вывода.

Вы должны также понимать, что не всегда обязательно использовать каждый псевдоним или таблицу которые упомянуты в предложении FROM запроса, в предложении SELECT. Иногда, предложение или таблица становятся запрашиваемыми исключительно потому что они могут вызываться в предикате запроса. Например, следующий запрос находит всех заказчиков размещенных в городах где продавец Serres (snum 1002) имеет заказиков (вывод показывается в Рисунке 9.4):

```
SELECT b.cnum, b.cname
FROM Customers a, Customers b
WHERE a.snum = 1002
      AND b.city = a.city;
```

```
 ===============  SQL Execution Log ============
| SELECT b.cnum, b.cname                        |
| FROM Customers a, Customers b                 |
| WHERE a.snum = 1002                           |
| AND b.city = a.city;                          |
| ==============================================|
|   cnum     cname                              |
| ------   ---------                           |
|   2003     Liu                                |
|   2008     Cisneros                           |
|   2004     Grass                              |
 ===============================================
```

Рисунок 9.4: Нахождение заказчиков в городах относящихся к Serres.

Псевдоним **a** будет делать предикат неверным за исключением случая когда его значение столбца snum = 1002. Таким образом псевдоним опускает все, кроме заказчиков продавца Serres. Псевдоним **b** будет верным для всех строк с тем же самым значением города что и текущее значение города для **a**; в ходе запроса, строка псевдонима **b** будет верна один раз когда значение города представлено в **a**.

Нахождение этих строк псевдонима **b** — единственая цель псевдонима **a**, поэтоиму мы не выбираем все столбцы подряд. Как вы можете видеть, собственные заказчики Serres выбираются при нахождении их в том же самом городе что и он сам, поэтому выбор их из псевдонима **a** необязателен. Короче говоря, псевдоним назходит строки заказчиков Serres, Liu и Grass. Псевдоним **b** находит всех заказчиков размещенных в любом из их городов (San Jose и Berlin соответственно) включая, конечно, самих — Liu и Grass.

Вы можете также создать обьединение которое включает и различные таблицы и псевдонимы одиночной таблицы. Следующий запрос объединяет таблицу Пользователей с собой: чтобы найти все пары заказчиков обслуживаемых одним продавцом. В то же самое время, этот запрос объединяет заказчика с таблицей Продавцов с именем этого продавца (вывод показан на Рисунке 9.5):

```
SELECT sname, Salespeople.snum, first.cname, second.cname
FROM Customers first, Customers second, Salespeople
WHERE first.snum = second.snum
      AND Salespeople.snum = first.snum
      AND first.cnum < second.cnum;
```

```
 ===============  SQL Execution Log ==================
| SELECT cname, Salespeople.snum, first.cname          |
| second.cname                                         |
| FROM Customers first, Customers second, Salespeople  |
| WHERE first.snum  = second.snum                      |
| AND Salespeople.snum = first.snum                    |
| AND first.cnum < second.cnum;                        |
| =====================================================|
|  cname      snum        cname       cname            |
|  ------   ------      --------    --------           |
|  Serres     1002        Liu         Grass            |
|  Peel       1001        Hoffman     Clemens          |
 ======================================================
```

Рисунок 9.5: Объединение таблицы с собой и с другой таблицей

РЕЗЮМЕ

Теперь Вы понимаете возможности объединения и можете использовать их для ограничения связей с таблицей, между различными таблицами, или в обоих случаях. Вы могли видеть некоторые возможности объединения при использовании его способностей. Вы теперь познакомились с терминами *порядковые переменные*, *корреляционные переменные и предложения* (эта терминология будет меняться от изделия к изделию, так что мы предлагаем Вам познакомится со всеми тремя терминами). Кроме того Вы поняли, немного, как в действительности работают запросы.

Следующим шагом после комбинации многочисленых таблиц или многочисленых копий одной таблицы в запросе, будет комбинация многочисленных запросов, где один запрос будет производить вывод который будет затем управлять работой другого запроса. Это другое мощное средство SQL, о котором мы расскажем в Главе 10 и более тщательно в последующих главах.

РАБОТА С SQL

1. Напишите запрос, который бы вывел все пары продавцов, живущих в одном и том же городе. Исключите комбинации продавцов с ними же, а также дубликаты строк, выводимых в обратным порядке.
2. Напишите запрос, который вывел бы все пары порядков по данным заказчикам, именам этих заказчиков, и исключал дубликаты из вывода, как в предыдущем вопросе.
3. Напишите запрос, который вывел бы имена (cname) и города (city) всех заказчиков с такой же оценкой (rating) как у Hoffmana. Напишите запрос, использующий поле cnum Hoffmana а не его оценку, так чтобы оно могло быть использовано если его оценка вдруг изменится.

(См. Приложение A для ответов.)

10

ВСТАВКА ОДНОГО ЗАПРОСА ВНУТРЬ ДРУГОГО

В КОНЕЦ ГЛАВЫ 9, МЫ ГОВОРИЛИ ЧТО ЗАПРОСЫ могут управлять другими запросами. В этой главе, вы узнаете как это делается (большей частью), помещая запрос внутрь предиката другого запроса, и используя вывод внутреннего запроса в верном или неверном условии предиката. Вы сможете выяснить какие виды операторов могут использовать подзапросы и посмотреть как подзапросы работают со средствами SQL, такими как DISTINCT, с составными функциями и выводимыми выражения. Вы узнаете как использовать подзапросы с предложением HAVING и получите некоторые наставления как правильно использовать подзапросы.

КАК РАБОТАЕТ ПОДЗАПРОС?

С помощью SQL вы можете вкладывать запросы внутрь друг друга. Обычно, внутренний запрос генерирует значение, которое проверяется в предикате внешнего запроса, определяющего, верно оно или нет.

Например, предположим что мы знаем имя продавца: Motika, но не знаем значение его поля snum, и хотим извлечь все порядки из таблицы Порядков. Имеется один способ чтобы сделать это (вывод показывается в Рисунке 10.1):

```
SELECT *
FROM Orders
WHERE snum = (SELECT snum
              FROM Salespeople
              WHERE sname = 'Motika');
```

Чтобы оценить внешний (основной) запрос, SQL сначала должен оценить внутренний запрос (или подзапрос) внутри предложения WHERE. Он делает это так как и должен делать запрос имеющий единственную цель — отыскать через таблицу Продавцов все строки, где поле sname равно значению Motika, и затем извлечь значения поля snum этих строк.

Единственной найденной строкой, естественно, будет snum = 1004. Однако SQL не просто выдает это значение, а помещает его в предикат основного запроса вместо самого подзапроса, так чтобы предиката прочитал, что

```
WHERE snum = 1004
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM Orders                                     |
| WHERE snum =                                    |
| (SELECT snum                                    |
| FROM Salespeople                                |
| WHERE sname = 'Motika');                        |
|=================================================|
|   onum       amt      odate      cnum     snum  |
|  -----     -------  ----------  -----    -----  |
|   3002     1900.10  10/03/1990   2007     1004  |
 =================================================
```

Рисунок 10.1: Использование подзапроса

Основной запрос затем выполняется как обычно с вышеупомянутыми результатами. Конечно же, подзапрос должен выбрать один и только один столбец, а тип данных этого столбца должен совпадать с тем значением, с которым он будет сравниваться в предикате. Часто, как показано выше, выбранное поле и его значение будут иметь одинаковые имена (в этом случае, snum), но это необязательно.

Конечно, если бы мы уже знали номер продавца Motika, мы могли бы просто напечатать

```
WHERE snum = 1004
```

и выполнять далее с подзапросом в целом, но это было бы не так универсально. Это будет продолжать работать даже если номер Motika изменился, а с помощью простого изменения имени в подзапросе, вы можете использовать его для чего угодно.

ЗНАЧЕНИЯ, КОТОРЫЕ МОГУТ ВЫДАВАТЬ ПОДЗАПРОСЫ

Скорее всего было бы удобнее, чтобы наш подзапрос в предыдущем примере возвращал одно и только одно значение. Имея выбранным поле snum **"WHERE city = 'London'"** вместо **"WHERE sname = 'Motika'",** можно получить несколько различных значений. Это может сделать уравнение в предикате основного запроса невозможным для оценки верности или неверности, и команда выдаст ошибку.

При использовании подзапросов в предикатах основанных на реляционных операторах (уравнениях или неравенствах, как объяснено в Главе 4), вы должны убедиться, что использовали подзапрос который будет выдавать одну и только одну строку вывода. Если вы используете подзапрос, который не выводит никаких значений вообще, команда не потерпит неудачи, но основной запрос не выведет никаких значений.

Подзапросы, которые не производят никакого вывода (или нулевой вывод), вынуждают рассматривать предикат ни как верный, ни как неверный, а как *неизвестный*. Однако, неизвестный предикат имеет тот же самый эффект, что и неверный: никакие строки не выбираются основным запросом (смотри Главу 5 для подробной информации о неизвестном предикате).

Это плохая стратегия, чтобы делать что-нибудь подобное следующему:

```
SELECT *
FROM Orders
WHERE snum = ( SELECT snum
               FROM Salespeople
               WHERE city = "Barcelona" );
```

Поскольку мы имеем только одного продавца в Barcelona — Rifkin, то подзапрос будет выбирать одиночное значение snum и следовательно будет принят. Но это — только в данном случае. Большинство SQL баз данных имеют многочисленых пользователей, и если другой пользователь добавит нового продавца из Barcelona в таблицу, подзапрос выберет два значения, и ваша команда потерпит неудачу.

DISTINCT С ПОДЗАПРОСАМИ

Вы можете, в некоторых случаях, использовать DISTINCT, чтобы вынудить подзапрос генерировать одиночное значение. Предположим что мы хотим найти все порядки кредитований для тех продавцов которые обслуживают Hoffmana (cnum = 2001). Имеется один способ, чтобы сделать это (вывод показывается в Рисунке 10.2):

```
SELECT *
FROM Orders
WHERE snum = ( SELECT DISTINCT snum
               FROM Orders
               WHERE cnum = 2001 );
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM Orders                                     |
| WHERE snum =                                    |
| (SELECT DISTINCT snum                           |
| FROM Orders                                     |
| Where cnum = 2001);                             |
| ============================================== |
|   onum       amt      odate      cnum     snum  |
|  -----   ---------  ---------   ------  ------- |
|   3003      767.19  10/03/1990   2001     1001  |
|   3008     4723.00  10/05/1990   2006     1001  |
|   3011     9891.88  10/06/1990   2006     1001  |
 ================================================
```

Рисунок 10.2: Использование DISTINCT чтобы вынудить получение одного значения из подзапроса

Подзапрос установил, что значение поля snum совпало с Hoffman — 1001, и затем основной запрос выделил все порядки с этим значением snum из таблицы Порядков (не разбирая, относятся они к Hoffman или нет). Так как каждый заказчик назначен к одному и только этому продавцу, мы знаем, что каждая строка в таблице Порядков с данным значением cnum должна иметь такое же значение snum. Однако, так как там может быть любое число таких строк, подзапрос мог бы вывести много (хотя и идентичных) значений snum для данного поля cnum. Аргумент DISTINCT предотвращает это. Если наш подзапрос возвратит более одного значения, это будет указывать на ошибку в наших данных — хорошая вещь для знающих об этом.

Альтернативный подход должен быть чтобы ссылаться к таблице Заказчиков а не к таблице Порядков в подзапросе. Так как поле cnum — это первичный ключ таблицы Заказчика, запрос выбирающий его должен произвести только одно значение. Это рационально только если вы как пользователь имеете доступ к таблице Порядков но не к таблице Заказчиков. В этом случае, вы можете использовать решение которое мы показали выше. (SQL имеет механизмы которые определяют — кто имеет привилегии чтобы делать что-то в определенной таблице. Это будет объясняться в Главе 22.)

Пожалуйста учтите, что методика, используемая в предшествующем примере, применима, только когда вы знаете, что два различных поля в таблице должны всегда совпадать, как в нашем случае. Эта ситуация не является типичной в реляционных базах данных, она являеться исключением из правил.

ПРЕДИКАТЫ С ПОДЗАПРОСАМИ ЯВЛЯЮТСЯ НЕОБРАТИМЫМИ

Вы должны обратить внимание, что предикаты, включающие подзапросы, используют выражение

<скалярная форма> <оператор> <подзапрос>, а не
<подзапрос> <оператор> <скалярное выражение> или,
<подзапрос> <оператор> <подзапрос>.

Другими словами, вы не должны записывать предыдущий пример так:

```
SELECT *
FROM Orders
WHERE ( SELECT DISTINCT snum
        FROM Orders
        WHERE cnum = 2001 ) = snum;
```

В строгой ANSI реализации, это приведет к неудаче, хотя некоторые программы и позволяют делать такие вещи. ANSI также предохраняет вас от появления обеих значений при сравнении, которые нужно вывести с помощью подзапроса.

ИСПОЛЬЗОВАНИЕ АГРЕГАТНЫХ ФУНКЦИЙ В ПОДЗАПРОСАХ

Один тип функций, который автоматически может производить одиночное значение для любого числа строк, конечно же, — агрегатная функция. Любой запрос, использующий одиночную функцию агрегата без предложения GROUP BY, будет выбирать одиночное значение для использования в основном предикате. Например, вы хотите увидеть все порядки имеющие сумму приобретений выше средней на 4-е Октября (вывод показан на Рисуноке 10.3):

```
SELECT *
FROM Orders
WHERE amt > ( SELECT AVG (amt)
              FROM Orders
              WHERE odate = 10/04/1990 );
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM Orders                                     |
| WHERE amt >                                     |
| (SELECT AVG (amt)                               |
|  FROM Orders                                    |
|  WHERE odate = 01/04/1990 );                    |
| ================================================ |
|   onum       amt      odate      cnum     snum  |
|  -----    --------  ----------  -----    -----  |
|   3002     1900.10  10/03/1990   2007     1004  |
|   3005     2345.45  10/03/1990   2003     1002  |
|   3006     1098.19  10/03/1990   2008     1007  |
|   3009     1713.23  10/04/1990   2002     1003  |
|   3008     4723.00  10/05/1990   2006     1001  |
|   3010     1309.95  10/06/1990   2004     1002  |
|   3011     9891.88  10/06/1990   2006     1001  |
 ================================================
```

Рисунок 10.3: Выбор всех сумм со значением выше средней на 10/04/1990

Средняя сумма приобретений на 4 Октября — 1788.98 (1713.23 + 75.75) делится пополам, что в целом равняется = 894.49. Все строки со значением в поле amt выше этого — являются выбраными.

Имейте в виду, что сгруппированные агрегатные функции, которые являются агрегатными функциями определенными в терминах предложения GROUP BY, могут производить многочисленые значения. Они, следовательно, не позволительны в подзапросах такого характера. Даже если GROUP BY и HAVING используются таким способом, что только одна группа выводится с помощью подзапроса, команда будет отклонена в принципе. Вы должны использовать одиночную агрегатную функцию с предложением WHERE что устранит нежелательные группы. Например, следующий запрос который должен найти среднее значение комиссионных продавца в Лондоне —

```
SELECT AVG (comm)
FROM Salespeople
GROUP BY city
HAVlNG city = "London";
```

не может использоваться в подзапросе! Во всяком случае это не лучший способ формировать запрос. Другим способом может быть

```
SELECT AVG (comm)
FROM Salespeople
WHERE city = "London";
```

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ, КОТОРЫЕ ВЫДАЮТ МНОГО СТРОК С ПОМОЩЬЮ ОПЕРАТОРА IN

Вы можете использовать подзапросы которые производят любое число строк если вы используете специальный оператор IN (операторы BETWEEN, LIKE, и IS NULL не могут использоваться с подзапросами). Как вы помните, IN определяет набор значений, одно из которых должно совпадать с другим термином уравнения предиката в порядке, чтобы предикат был верным. Когда вы используете IN с подзапросом, SQL просто формирует этот набор из вывода подзапроса. Мы можем, следовательно, использовать IN чтобы выполнить такой же подзапрос, который не будет работать с реляционным оператором, и найти все атрибуты таблицы Порядков для продавца в Лондоне (вывод показывается в Рисунке 10.4):

```
SELECT *
FROM Orders
WHERE snum IN ( SELECT snum
                FROM Salespeople
                WHERE city = "LONDON" );
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM Orders                                     |
| WHERE snum IN                                   |
| (SELECT snum                                    |
|  FROM Salespeople                               |
|  WHERE city = 'London');                        |
| ================================================ |
|   onum       amt      odate      cnum     snum  |
|  -----    --------  ----------  -----    ------  |
|   3003      767.19  10/03/1990   2001     1001  |
|   3002     1900.10  10/03/1990   2007     1004  |
|   3006     1098.19  10/03/1990   2008     1007  |
|   3008     4723.00  10/05/1990   2006     1001  |
|   3011     9891.88  10/06/1990   2006     1001  |
 ================================================
```

Рисунок 10.4: Использование подзапроса с IN

В ситуации подобно этой, подзапрос более прост для пользователя чтобы понимать его и более прост для компьютера чтобы его выполнить, чем если бы Вы использовали обьединение:

```
SELECT onum, amt, odate, cnum, Orders.snum
FROM Orders, Salespeople
WHERE Orders.snum = Salespeople.snum
      AND Salespeople.city = "London";
```

Хотя это и произведет тот же самый вывод что и в примере с подзапросом, SQL должен будет просмотреть каждую возможную комбинацию строк из двух таблиц и проверить их снова по составному предикату.

Проще и эффективнее извлекать из таблицы Продавцов значения поля snum где city = "London", и затем искать эти значения в таблице Порядков, как это делается в варианте с подзапросом. Внутренний запрос дает нам snums=1001 и snum=1004. Внешний запрос, затем, дает нам строки из таблицы Порядков, где эти поля snum найдены.

Строго говоря, быстрее или нет работает вариант подзапроса, практически зависит от реализации — в какой программе вы это используете. Эта часть вашей программы, называемая ***оптимизатор***, пытается найти наиболее эффективный способ выполнения ваших запросов.

Хороший оптимизатор во всяком случае преобразует вариант обьединения в подзапрос, но нет достаточно простого способа для вас, чтобы выяснить, выполнено это или нет. Лучше сохранить ваши запросы в памяти, чем полагаться полностью на оптимизатор.

Конечно, вы можете также использовать оператор IN, даже когда вы уверены, что подзапрос произведет одиночное значение. В любой ситуации где вы можете использовать реляционный оператор сравнения (=), вы можете использовать IN. В отличие от реляционных операторов, IN не может заставить команду потерпеть неудачу если больше чем одно значение выбрано подзапросом. Это может быть или преимуществом или недостатком.

Вы не увидите непосредственно вывода из подзапросов; если вы полагаете, что подзапрос собирается произвести только одно значение, а он производит различные. Вы не сможете объяснить различия в выводе основного запроса. Например, рассмотрим команду, которая похожа на предыдущую:

```
SELECT onum, amt, odate
FROM Orders
WHERE snum = ( SELECT snum
               FROM Orders
               WHERE cnum = 2001 );
```

Вы можете устранить потребность в DISTINCT, используя IN вместо (=), подобно этому:

```
SELECT onum, amt, odate
FROM Orders
WHERE snum IN ( SELECT snum
                FROM Orders
                WHERE cnum = 2001 );
```

Что случится, если есть ошибка и один из порядков был акредитован к различным продавцам? Версия, использующая IN, будет давать вам все порядки для обоих продавцов. Нет никакого очевидного способа наблюдения за ошибкой, и поэтому сгенерированные отчеты или решения, сделанные на основе этого запроса, не будут содержать ошибки. Вариант, использующий (=), просто потерпит неудачу.

Это, по крайней мере, позволило бы вам узнать, что имеется такая проблема. Вы должны затем выполнять поиск неисправности, выполнив этот подзапрос отдельно и наблюдая значения, которые он производит.

В принципе, если вы знаете, что подзапрос должен (по логике) вывести только одно значение, вы должны использовать **=**. IN является подходящим, если запрос может ограниченно производить одно или более значений, независимо от того ожидаете вы их или нет.

Предположим, мы хотим знать комиссионные всех продавцов обслуживающих заказчиков в Лондоне:

```
SELECT comm
FROM Salespeople
WHERE snum IN ( SELECT snum
                FROM Customers
                WHERE city = "London" );
```

Выводимыми для этого запроса, показанного в Рисунке 10.5, являются значения комиссионных продавца Peel (snum = 1001), который имеет обоих заказчиков в Лондоне. Это — только для данного случая. Нет никакой причины чтобы некоторые заказчики в Лондоне не могли быть назначеными к кому-то еще. Следовательно, IN — это наиболее логичная форма чтобы использовать ее в запросе.

```
 ===============  SQL Execution Log ==============
| SELECT comm                                      |
| FROM Salespeople                                 |
| WHERE snum IN                                    |
| (SELECT snum                                     |
|  FROM Customers                                  |
|  WHERE city = 'London');                         |
| ================================================ |
|    comm                                          |
|  -------                                         |
|    0.12                                          |
 ================================================
```

Рисунок 10.5: Использование IN с подзапросом для вывода одного значения

Между прочим, префикс таблицы для поля city необязателен в предыдущем примере, несмотря на возможную неоднозначность между полями city таблицы Заказчика и таблицы Продавцов.

SQL всегда ищет первое поле в таблице обозначенной в предложении FROM текущего подзапроса. Если поле с данным именем там не найдено, проверяются внешние запросы. В вышеупомянутом примере, "city" в предложении WHERE означает что имеется ссылка к Customer.city (поле city таблицы Заказчиков).

Так как таблица Заказчиков указана в предложении FROM текущего запроса, SQL предполагает что это — правильно. Это предположение может быть отменено полным именем таблицы или префиксом псевдонима, о которых мы поговорим позже когда будем говорить об соотнесенных подзапросах. Если возможен беспорядок, конечно же, лучше всего использовать префиксы.

ПОДЗАПРОСЫ ВЫБИРАЮТ ОДИНОЧНЫЕ СТОЛБЦЫ

Смысл всех подзапросов обсужденных в этой главе тот, что все они выбирают одиночный столбец. Это обязательно, поскольку выбранный вывод сравнивается с одиночным значением. Подтверждением этому то, что SELECT * не может использоваться в подзапросе. Имеется исключение из этого, когда подзапросы используются с оператором EXISTS, который мы будем представлять в Главе 12.

ИСПОЛЬЗОВАНИЕ ВЫРАЖЕНИЙ В ПОДЗАПРОСАХ

Вы можете использовать выражение основанное на столбце, а не просто сам столбец, в предложении SELECT подзапроса. Это может быть выполнено или с помощью реляционных операторов или с IN. Например, следующий запрос использует реляционный оператор = (вывод показывается в Рисунке 10.6):

```
SELECT *
FROM Customers
WHERE cnum = ( SELECT snum + 1000
               FROM Salespeople
               WHERE sname = “Serres” );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                      |
| FROM Customers                                |
| WHERE cnum =                                  |
| (SELECT snum + 1000                           |
| WHERE Salespeople                             |
| WHERE sname = 'Serres'                        |
| ============================================= |
|   cnum     cname     city    rating    snum   |
|  -----    --------   ----    ------   -----   |
|   2002    Giovanni   Rome       200    1003   |
 ===============================================
```

Рисунок 10.6: Использование подзапроса с выражением

Он находит всех заказчиков, чье значение поля cnum равное 1000, выше поля snum Serres. Мы предполагаем, что столбец sname не имеет никаких двойных значений (это может быть предписано или UNIQUE INDEX, обсуждаемым в Главе 17, или ограничением UNIQUE, обсуждаемым в Главе 18); иначе подзапрос может произвести многочисленые значения. Когда поля snum и cnum не имеют такого простого функционального значения как например первичный ключ, что не всегда хорошо, запрос типа вышеупомянутого невероятно полезен.

ПОДЗАПРОСЫ В ПРЕДЛОЖЕНИИ HAVING

Вы можете также использовать подзапросы внутри предложения HAVING. Эти подзапросы могут использовать свои собственные агрегатные функции если они не производят многочисленых значений или использовать GROUP BY или HAVING. Следующий запрос является этому примером (вывод показывается в Рисунке 10.7):

```
SELECT rating, COUNT ( DISTINCT cnum )
FROM Customers
GROUP BY rating
HAVING rating > ( SELECT AVG (rating)
                  FROM Customers
                  WHERE city = “San Jose” );
```

```
 ===============  SQL Execution Log ============
| SELECT rating,count (DISTINCT cnum)            |
| FROM Customers                                 |
| GROUP BY rating                                |
| HAVING rating >                                |
| (SELECT AVG (rating)snum + 1000                |
|  FROM Custimers                                |
|  WHERE city = 'San Jose'                       |
|=============================================   |
|  rating                                        |
| --------    --------                           |
|   200             2                            |
 ================================================
```

Рисунок 10.7: Нахождение заказчиков с оценкой выше среднего в San Jose

Эта команда подсчитывает заказчиков с оценками выше среднего в San Jose. Так как имеются другие оценки отличные от 300, они должны быть выведены с числом номеров заказчиков которые имели эту оценку.

РЕЗЮМЕ

Теперь вы используете запросы в иерархической манере. Вы видели, что использование результата одного запроса для управления другим, расширяет возможности позволяющие выполнить большее количество функций. Вы теперь понимаете как использовать подзапросы с реляционными операторами также как и со специальным оператором IN, или в предложении WHERE или в предложении HAVING внешнего запроса.

В следующих главах, мы будем разрабатывать подзапросы. Сначала в Главе 11, мы обсудим другой вид подзапроса, который выполняется отдельно для каждой строки таблицы вызываемой во внешнем запросе. Затем, в Главе 12 и 13, мы представим вас нескольким специальным операторам которые функционируют на всех подзапросах, как это делает IN, за исключением когда эти операторы могут использоваться только в подзапросах.

РАБОТА С SQL

1. Напишите запрос, который бы использовал подзапрос для получения всех порядков для заказчика с именем Cisneros. Предположим, что вы не знаете номера этого заказчика, указываемого в поле cnum.
2. Напишите запрос, который вывел бы имена и оценки всех заказчиков, которые имеют усредненые порядки.
3. Напишите запрос, который бы выбрал общую сумму всех приобретений в порядках для каждого продавца, у которого эта общая сумма больше, чем сумма наибольшего порядка в таблице.

(См. Приложение A для ответов.)

11

СООТНЕСЕННЫЕ ПОДЗАПРОСЫ

В ЭТОЙ ГЛАВЕ, МЫ ПРЕДСТАВИМ ВАС ТИПУ подзапроса о котором мы не говорили в Главе 10 — посвященной соотнесенному подзапросу. Вы узнаете как использовать соотнесенные подзапросы в предложениях запросов WHERE и HAVING. Сходства и различия между соотнесенными подзапросами и обьединениями будут обсуждаться далее, и вы сможете повысить ваше знание псевдонимов и префиксов имени таблицы — когда они необходимы и как их использовать.

КАК СФОРМИРОВАТЬ СООТНЕСЕННЫЙ ПОДЗАПРОС

Когда вы используете подзапросы в SQL, вы можете обратиться к внутреннему запросу таблицы в предложении внешнего запроса FROM, сформировав *соотнесенный подзапрос*. Когда вы делаете это, подзапрос выполняется неоднократно, по одному разу для каждой строки таблицы основного запроса.

Соотнесенный подзапрос — один из большого количества тонких понятий в SQL из-за сложности в его оценке.

Если вы сумеете овладеть им, вы найдете что он очень мощный, потому что может выполнять сложные функции с помощью очень лаконичных указаний.

Например, имеется один способ найти всех заказчиков в порядках на 3-е Октября (вывод показывается в Рисунке 11.1):

```
SELECT *
FROM Customers outer
WHERE 10/03/1990 IN ( SELECT odate
                      FROM Orders inner
                      WHERE outer.cnum = inner.cnum );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Customers outer                          |
| WHERE 10/03/1990 IN                            |
| (SELECT odate                                  |
| FROM Orders inner                              |
| WHERE outer.cnum = inner.cnum);                |
| ============================================== |
|   cnum     cname     city    rating    snum    |
|  -----    --------   ----    ------   -----    |
|   2001    Hoffman    London     100    1001    |
|   2003    Liu        San Jose   200    1002    |
|   2008    Cisneros   San Jose   300    1007    |
|   2007    Pereira    Rome       100    1004    |
 ================================================
```

Рисунок 11.1: Использование соотнесенного подзапроса

КАК РАБОТАЕТ СООТНЕСЕННЫЙ ПОДЗАПРОС

В вышеупомянутом примере, "внутренний" (inner) и "внешний" (outer), это псевдонимы, подобно обсужденным в Главе 9. Мы выбрали эти имена для большей ясности; они отсылают к значениям внутренних и внешних запросов, соответственно. Так как значение в поле cnum внешнего запроса меняется, внутренний запрос должен выполняться отдельно для каждой строки внешнего запроса. Строка внешнего запроса для которого внутренний запрос каждый раз будет выполнен, называется — текущей *строкой-кандидатом*. Следовательно, процедура оценки выполняемой соотнесенным подзапросом — это:

1. Выбрать строку из таблицы именованной в внешнем запросе. Это будет текущая строка-кандидат.
2. Сохранить значения из этой строки-кандидата в псевдониме с именем в предложении FROM внешнего запроса.
3. Выполнить подзапрос. Везде, где псевдоним данный для внешнего запроса найден (в этом случае "внешний"), использовать значение для текущей строки-кандидата. Использование значения из строки-кандидата внешнего запроса в подзапросе называется — *внешней ссылкой.*
4. Оценить предикат внешнего запроса на основе результатов подзапроса выполняемого в шаге 3. Он определяеть — выбирается ли строка-кандидат для вывода.
5. Повторить процедуру для следующей строки-кандидата таблицы, и так далее пока все строки таблицы не будут проверены.

В вышеупомянутом примере, SQL осуществляет следующую процедуру:

1. Он выбирает строку Hoffman из таблицы Заказчиков.
2. Сохраняет эту строку как текущую строку-кандидат под псевдонимом — "внешним".
3. Затем он выполняет подзапрос. Подзапрос просматривает всю таблицу Порядков чтобы найти строки где значение cnum поле — такое же как значение outer.cnum, которое в настоящее время равно 2001, — поле cnum строки Hoffmana. Затем он извлекает поле odate из каждой строки таблицы Порядков, для которой это верно, и формирует набор значений поля odate.
4. Получив набор всех значений поля odate, для поля cnum = 2001, он проверяет предикат основного запроса чтобы видеть имеется ли значение на 3 Октября в этом наборе. Если это так (а это так), то он выбирает строку Hoffmana для вывода ее из основного запроса.
5. Он повторяет всю процедуру, используя строку Giovanni как строку-кандидата, и затем сохраняет повторно пока каждая строка таблицы Заказчиков не будет проверена.

Как вы можете видеть, вычисления которые SQL выполняет с помощью этих простых инструкций — это полный комплекс. Конечно, вы могли бы решить ту же самую проблему используя обьединение, следующего вида (вывод для этого запроса показывается в Рисунке 11.2):

```
SELECT *
FROM Customers first, Orders second
WHERE first.cnum = second.cnum AND second.odate = 10/03/1990;
```

Обратите внимание, что Cisneros был выбран дважды, по одному разу для каждого порядка, который он имел для данной даты. Мы могли бы устранить это используя SELECT DISTINCT вместо просто SELECT. Но это необязательно в варианте подзапроса. Оператор IN, используемый в варианте подзапроса, не делает никакого различия между значениями которые выбираются подзапросом один раз и значениями которые выбираются неоднократно. Следовательно DISTINCT необязателен.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Customers first, Orders second            |
| WHERE first.cnum = second.cnum                 |
| (SELECT COUNT (*)                              |
|  FROM Customers                                |
|  WHERE snum = main.snum;                       |
| ============================================== |
|   cnum     cname                               |
|  -----    --------                             |
|   1001     Peel                                |
|   1002     Serres                              |
 ================================================
```

Рисунок 11.2: Использование обьединения вместо соотнесенного подзапроса

Предположим что мы хотим видеть имена и номера всех продавцов которые имеют более одного заказчика. Следующий запрос выполнит это для вас (вывод показывается в Рисунке 11.3):

```
SELECT snum, sname
FROM Salespeople main
WHERE 1 < ( SELECT COUNT (*)
            FROM Customers
            WHERE snum = main.snum );
```

Обратите внимание, что предложение FROM подзапроса в этом примере не использует псевдоним. При отсутствии имени таблицы или префикса псевдонима, SQL может для начала принять, что любое поле выводится из таблицы с именем, указанным в предложении FROM текущего запроса. Если поле с этим именем отсутствует (в нашем случае — snum) в той таблице, SQL будет проверять внешние запросы. Именно поэтому, префикс имени таблицы обычно необходим в соотнесенных подзапросах — для отмены этого предположения. Псевдонимы также часто запрашиваются, чтобы давать вам возможность ссылаться к той же самой таблице во внутреннем и внешнем запросе без какой-либо неоднозначности.

```
 ===============  SQL Execution Log ============
| SELECT snum sname                              |
| FROM  Salespeople main                         |
| WHERE 1 <                                      |
| AND second.odate = 10/03/1990;                 |
| ============================================== |
|   cnum     cname     city    rating    snum    |
|  -----    --------   ----    ------   -----    |
|   2001    Hoffman    London     100    1001    |
|   2003    Liu        San Jose   200    1002    |
|   2008    Cisneros   San Jose   300    1007    |
|   2007    Pereira    Rome       100    1004    |
 ================================================
{!!! здесь явно глюк}
```

Рисунок 11.3: Нахождение продавцов с многочисленными заказчиками

ИСПОЛЬЗОВАНИЕ СООТНЕСЕННЫХ ПОДЗАПРОСОВ ДЛЯ НАХОЖДЕНИЯ ОШИБОК

Иногда полезно выполнять запросы, которые разработаны специально так, чтобы находить ошибки. Это всегда возможно при дефектной информации, которую можно ввести в вашу базу данных, и, если она введена, бывает трудно ее определить. Следующий запрос не должен производить никакого вывода. Он просматривает таблицу Порядков, чтобы видеть, совпадают ли поля snum и cnum в каждой строке таблицы Заказчиков и выводит каждую строку, где этого совпадения нет. Другими словами, запрос выясняет, тот ли продавец кредитовал каждую продажу (он воспринимает поле cnum, как первичный ключ таблицы Заказчиков, который не будет иметь никаких двойных значений в этой таблице).

```
SELECT *
FROM Orders main
WHERE NOT snum = ( SELECT snum
                   FROM Customers
                   WHERE cnum = main.cnum );
```

При использовании механизма справочной целостности (обсужденного в Главе 19), вы можете быть гарантированы от некоторых ошибок такого вида. Этот механизм не всегда доступен, хотя его использование желательно во всех случаях, причем поиск ошибки запроса описанный выше, может быть еще полезнее.

СРАВНЕНИЕ ТАБЛИЦЫ С СОБОЙ

Вы можете также использовать соотнесенный подзапрос, основанный на той же самой таблице, что и основной запрос. Это даст вам возможность извлечть определенные сложные формы произведенной информации. Например, мы можем найти все порядки со значениями сумм приобретений выше среднего для их заказчиков (вывод показан в Рисунке 11.4):

```
SELECT *
FROM Orders outer
WHERE amt > ( SELECT AVG amt
              FROM Orders inter
              WHERE inner.cnum = outer.cnum );
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM  Orders outer                              |
| WHERE amt >                                     |
| (SELECT AVG (amt)                               |
| FROM Orders inner                               |
| WHERE inner.cnum = outer.cnum                   |
| ============================================== |
|   onum       amt      odate      cnum     snum  |
|  -----    --------  ----------  -----   ------  |
|   3006     1098.19  10/03/1990   2008     1007  |
|   3010     1309.00  10/06/1990   2004     1002  |
|   3011     9891.88  10/06/1990   2006     1001  |
 ================================================
```

Рисунок 11.4: Соотнесение таблицы с собой

Конечно, в нашей маленькой типовой таблице, где большиство заказчиков имеют только один порядок, большинство значений являются одновременно средними и следовательно не выбираются. Давайте введем команду другим способом (вывод показывается в Рисунке 11.5):

```
SELECT *
FROM Orders outer
WHERE amt >= ( SELECT AVG (amt)
               FROM Orders inner
               WHERE inner.cnum = outer.cnum );
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM  Orders outer                              |
| WHERE amt > =                                   |
| (SELECT AVG (amt)                               |
| FROM Orders inner                               |
| WHERE inner.cnum = outer.cnum);                 |
| ============================================== |
|   onum       amt      odate      cnum     snum  |
|  -----    --------  ----------  -----   ------  |
|   3003      767.19  10/03/1990   2001     1001  |
|   3002     1900.10  10/03/1990   2007     1004  |
|   3005     5160.45  10/03/1990   2003     1002  |
|   3006     1098.19  10/03/1990   2008     1007  |
|   3009     1713.23  10/04/1990   2002     1003  |
|   3010     1309.95  10/06/1990   2004     1002  |
|   3011     9891.88  10/06/1990   2006     1001  |
 ================================================
```

Рисунок 11.5: Выбераются порядки которые >= средней сумме приобретений для их заказчиков.

Различие, конечно, в том, что реляционный оператор основного предиката включает значения которые равняются среднему (что обычно означает что они — единственые порядки для данных заказчиков).

СООТНЕСЕННЫЕ ПОДЗАПРОСЫ В ПРЕДЛОЖЕНИИ HAVING

Также как предложение HAVING может брать подзапросы, он может брать и соотнесенные подзапросы. Когда вы используете соотнесенный подзапрос в предложении HAVING, вы должны ограничивать внешние ссылки к позициям которые могли бы

непосредственно использоваться в самом предложении HAVING. Вы можете вспомнить из Главы 6 что предложение HAVING может использовать только агрегатные функции которые указаны в их предложении SELECT или поля используемые в их предложении GROUP BY. Они являются только внешними ссылками, которые вы можете делать. Все это потому, что предикат предложения HAVING оценивается для каждой группы из внешнего запроса, а не для каждой строки. Следовательно, подзапрос будет выполняться один раз для каждой группы выведеной из внешнего запроса, а не для каждой строки.

Предположим, что вы хотите суммировать значения сумм приобретений покупок из таблицы Порядков, сгруппировав их по датам, удалив все даты где бы SUM не был по крайней мере на 2000.00 выше максимальной (MAX) суммы:

```
SELECT odate, SUM (amt)
FROM Orders a
GROUP BY odate
HAVING SUM (amt) > ( SELECT 2000.00 + MAX (amt)
                     FROM Orders b
                     WHERE a.odate = b.odate );
```

Подзапрос вычисляет значение MAX для всех строк с той же самой датой что и у текущей агрегатной группы основного запроса. Это должно быть выполнено, как и ранее, с испошльзованием предложения WHERE. Сам подзапрос не должен использовать предложения GROUP BY или HAVING.

СООТНЕСЕННЫЕ ПОДЗАПРОСЫ И ОБЬЕДИНЕНИЯ

Как вы и могли предположить, соотнесенные подзапросы по природе близки к обьединениям — они оба включают проверку каждой строки одной таблицы с каждой строкой другой (или псевдонимом из той же) таблицы. Вы найдете, что большинство операций, которые могут выполняться с одним из них, будут также работать и с другим.

Однако имеется различие в прикладной программе между ними, такое как вышеупомянутая потребность в использовании DISTINCT с обьединением и его необязательность с подзапросом. Имеются также некоторые вещи, которые каждый может делать так, как этого не может другой. Подзапросы, например, могут использовать агрегатную функцию в предикате, делая возможным выполнение операций типа нашего предыдущего примера, в котором мы извлекли порядки, усредненные для их заказчиков.

Обьединения, с другой стороны, могут выводить строки из обеих сравниваемых таблиц, в то время как вывод подзапросов используется только в предикатах внешних запросов. Как правило, форма запроса которая кажется наиболее интуитивной будет вероятно лучшей в использовании, но при этом хорошо бы знать обе техники для тех ситуаций когда та или иная могут не работать.

РЕЗЮМЕ

Вы можете поздравлять себя с овладением большого куска из рассмотреных понятий в SQL — соотнесенного подзапроса. Вы видели как соотнесенный подзапрос связан с обьединение, а также, как как его можно использовать с агрегатными функциями и в предложении HAVING. В общем, вы теперь узнали все типы подзапросов полностью.

Следующий шаг — описание некоторых специальных операторов SQL. Они берут подзапросы как аргументы, как это делает IN, но в отличие от IN, они могут ис-

пользоваться только в подзапросах. Первый из их, представленный в Главе 12, — называется **EXISTS**.

РАБОТА С SQL

1. Напишите команду SELECT, использующую соотнесенный подзапрос, которая выберет имена и номера всех заказчиков с максимальными для их городов оценками.
2. Напишите два запроса, которые выберут всех продавцов (по их имени и номеру) которые, в своих городах имеют заказчиков, которых они не обслуживают. Один запрос — с использованием обьединения и один — с соотнесенным подзапросом. Которое из решений будет более изящным? (Подсказка: один из способом это сделать, состоит в том, чтобы находить всех заказчиков, не обслуживаемых данным продавцом и определить, находится ли каждый из них в городе продавца.)

(См. Приложение А для ответов.)

12

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРА EXISTS

ТЕПЕРЬ, КОГДА ВЫ ХОРОШО ОЗНАКОМЛЕНЫ С ПОДЗАПРОСАМИ, мы можем говорить о некоторых специальных операторах, которые всегда берут подзапросы как аргументы. Вы узнаете о первом из них в этой главе. Остальные будут описаны в следующей главе.

Оператор **EXISTS** используется чтобы указать предикату, производить ли подзапросу вывод или нет. В этой главе вы узнаете, как использовать этот оператор со стандартными и (обычно) соотнесенными подзапросами. Мы будем также обсуждать специальные размышления, которые перейдут в игру, когда вы будете использовать этот оператор, как относительный агрегат, как пустой указатель NULL и как оператор Буля. Кроме того, вы можете повысить ваш профессиональный уровень относительно подзапросов, исследуя их в более сложных прикладных программах чем те, которые мы видели до сих пор.

КАК РАБОТАЕТ EXISTS?

EXISTS — это оператор, который производит верное или неверное значение, другими словами, выражение Буля (см. Главу 4 для обзора этого термина). Это означает, что он может работать автономно в предикате или в комбинации с другими выражениями Буля, использующими Булевы операторы AND, OR, и NOT. Он берет подзапрос как аргумент и оценивает его как верный, если тот производит любой вывод или как неверный, если тот не делает этого. Этим он отличается от других операторов предиката, в которых он не может быть неизвестным. Например, мы можем решить, извлекать ли нам некоторые данные из таблицы Заказчиков если, и только если, один или более заказчиков в этой таблице находятся в San Jose (вывод для этого запроса показывается в Рисунке 12.1):

```
SELECT cnum, cname, city
FROM Customers
WHERE EXISTS ( SELECT *
               FROM Customers
               WHERE city = " San Jose' );
```

```
 ===============  SQL Execution Log ============
| SELECT snum, sname, city                       |
| FROM  Customers                                |
| WHERE EXISTS                                   |
| (SELECT *                                      |
| FROM Customers                                 |
| WHERE city = 'San Jose');                      |
| ============================================== |
|   cnum     cname     city                      |
|  -----    --------   ----                      |
|   2001    Hoffman    London                    |
|   2002    Giovanni   Rome                      |
|   2003    Liu        San Jose                  |
|   2004    Grass      Berlin                    |
|   2006    Clemens    London                    |
|   2008    Cisneros   San Jose                  |
|   2007    Pereira    Rome                      |
 ================================================
```

Рисунок 12.1 Использование оператора EXISTS

Внутренний запрос выбирает все данные для всех заказчиков в San Jose. Оператор EXISTS во внешнем предикате отмечает, что некоторый вывод был произведен подзапросом, и поскольку выражение EXISTS было полным предикатом, делает пре-

дикат верным. Подзапрос (не соотнесенный) был выполнен только один раз для всего внешнего запроса, и следовательно имеет одно значение во всех случаях. Поэтому EXISTS, когда используется этим способом, делает предикат верным или неверным для всех строк сразу, что это не так уж полезно для извлечения определенной информации.

ВЫБОР СТОЛБЦОВ С ПОМОЩЬЮ EXISTS

В вышеупомянутом примере, EXISTS должен быть установлен так чтобы легко выбрать один столбец, вместо того, чтобы выбирать все столбцы используя в выборе звезду (SELECT *) В этом состоит его отличие от подзапроса который (как вы видели ранее в Главе 10 мог выбрать только один столбец).

Однако, в принципе он мало отличается при выборе EXISTS столбцов, или когда выбираются все столбцы, потому что он просто замечает — выполняется или нет вывод из подзапроса — а не использует выведенные значения.

ИСПОЛЬЗОВАНИЕ EXISTS С СООТНЕСЕННЫМИ ПОДЗАПРОСАМИ

В соотнесенном подзапросе предложение EXISTS оценивается отдельно для каждой строки таблицы, имя которой указано во внешнем запросе, точно также как и другие операторы предиката, когда вы используете соотнесенный подзапрос. Это дает возможность использовать EXISTS как верный предикат, который генерирует различные ответы для каждой строки таблицы, указанной в основном запросе. Следовательно информация из внутреннего запроса будет сохранена, если выведена непосредственно, когда вы используете EXISTS таким способом. Например, мы можем вывести продавцов, которые имеют многочисленых заказчиков (вывод для этого запроса показывается в Рисунке 12.2):

```
SELECT DISTINCT snum
FROM Customers outer
WHERE EXISTS ( SELECT *
               FROM Customers inner
               WHERE inner.snum = outer.snum
                     AND inner.cnum < > outer.cnum );
```

```
 ===============  SQL Execution Log ============
| SELECT DISTINCT cnum                          |
| FROM  Customers outer                         |
| WHERE EXISTS                                  |
| (SELECT *                                     |
| FROM Customers inner                          |
| WHERE inner.snum = outer.snum                 |
| AND inner.cnum < > outer.cnum);               |
| ============================================= |
|   cnum                                        |
|  -----                                        |
|   1001                                        |
|   1002                                        |
 ===============================================
```

Рисунок 12.2: Использование EXISTS с соотнесенным подзапросом

Для каждой строки-кандидата внешнего запроса (представляющей заказчика проверяемого в настоящее время), внутренний запрос находит строки, которые сов-

падают со значением поля snum (которое имел продавец), но не со значением поля cnum (сответствующего другим заказчикам). Если любые такие строки найдены внутренним запросом, это означает, что имеются два разных заказчика, обслуживаемых текущим продавцом (т.е. продавцом заказчика в текущей строке-кандидата из внешнего запроса). Предикат EXISTS поэтому верен для текущей строки, и номер продавца — поле (snum) таблицы, указанной во внешнем запросе, будет выведен. Если был DISTINCT не указан, каждый из этих продавцов будет выбран один раз для каждого заказчика, к которому он назначен.

КОМБИНАЦИЯ ИЗ EXISTS И ОБЬЕДИНЕНИЯ

Однако для нас может быть полезнее вывести больше информации об этих продавцах, а не только их номера. Мы можем сделать это, объединив таблицу Заказчиков с таблицей Продавцов (вывод для запроса показывается в Рисунке 12.3):

```
SELECT DISTINCT first.snum, sname, first.city
FROM Salespeople first, Customers second
WHERE EXISTS ( SELECT *
               FROM Customers third
               WHERE second.snum = third.snum
                     AND second.cnum < > third.cnum )
      AND first.snum = second.snum;
```

```
 ===============  SQL Execution Log ============
| SELECT DISTINCT first.snum, sname, first.city |
| FROM  Salespeople first, Customers second     |
| WHERE EXISTS                                  |
| (SELECT *                                     |
| FROM Customers third                          |
| WHERE second.snum = third.snum                |
| AND second.cnum < > third.cnum)               |
| AND first.snum = second.snum;                 |
| ============================================= |
|   cnum     cname     city                     |
|  -----    --------   ----                     |
|   1001    Peel       London                   |
|   1002    Serres     San Jose                 |
 ===============================================
```

Рисунок 12.3: Комбинация EXISTS с обьединением

Внутренний запрос здесь — как и в предыдущем варианте, фактически сообщает, что псевдоним был изменен. Внешний запрос — это обьединение таблицы Продавцов с таблицей Заказчиков, наподобии того, что мы видели прежде. Новое предложение основного предиката (**AND first.snum = second.snum**) естественно оценивается на том же самом уровне, что и предложение EXISTS. Это — функциональный предикат самого обьединения, сравнивающий две таблицы из внешнего запроса в терминах поля snum, которое являются для них общим. Из-за Булева оператора AND, оба условия основного предиката должны быть верны в порядке для верного предиката. Следовательно, результаты подзапроса имеют смысл только в тех случаях когда вторая часть запроса верна, а обьединение — выполняемо. Таким образом комбинация объединения и подзапроса может стать очень мощным способом обработки данных.

ИСПОЛЬЗОВАНИЕ NOT EXISTS

Предыдущий пример дал понять что EXISTS может работать в комбинации с операторами Буля. Конечно, то что является самым простым способом для использования и вероятно наиболее часто используется с EXISTS — это оператор **NOT**. Один из способов которым мы могли бы найти всех продавцов только с одним заказчиком будет состоять в том, чтобы инвертировать наш предыдущий пример. (Вывод для этого запроса показывается в Рисунке 12.4.)

```
SELECT DISTINCT snum
FROM Customers outer
WHERE NOT EXISTS ( SELECT *
                   FROM Customers inner
                   WHERE inner.snum = outer.snum
                         AND inner.cnum <> outer.cnum );
```

```
 ===============  SQL Execution Log ============
| SELECT DISTINCT snum                          |
| FROM  Salespeople outer                       |
| WHERE NOT EXISTS                              |
| (SELECT *                                     |
| FROM Customers inner                          |
| WHERE inner.snum = outer.snum                 |
| AND inner.cnum < > outer.cnum);               |
| ============================================= |
|   cnum                                        |
|  -----                                        |
|   1003                                        |
|   1004                                        |
|   1007                                        |
 ===============================================
```

Рисунок 12.4: Использование EXISTS с NOT

EXISTS И АГРЕГАТЫ

Одна вещь, которую EXISTS не может сделать — взять функцию агрегата в подзапросе. Это имеет значение. Если функция агрегата находит любые строки для операций с ними, EXISTS верен, не взирая на то, что это — значение функции; если же агрегатная функция не находит никаких строк, EXISTS неправилен.

Попытка использовать агрегаты с EXISTS таким способом, вероятно покажет что проблема неверно решалась от начала до конца.

Конечно, подзапрос в предикате EXISTS может также использовать один или более из его собственных подзапросов. Они могут иметь любой из различных типов которые мы видели (или который мы будем видеть). Такие подзапросы, и любые другие в них, позволяют использовать агрегаты, если нет другой причины по которой они не могут быть использованы. Следующий раздел приводит этому пример.

В любом случае, вы можете получить тот же самый результат более легко, выбрав поле которое вы использовали в агрегатной функции, вместо использования самой этой функции. Другими словами, предикат — **EXISTS (SELECT COUNT (DISTINCT sname) FROM Salespeople)** — будет эквивалентен **EXISTS (SELECT sname FROM Salespeople)** который был позволен выше.

БОЛЕЕ УДАЧНЫЙ ПРИМЕР ПОДЗАПРОСА

Возможные прикладные программы подзапросов могут становиться многократно вкладываемыми.

Вы можете вкладывать их два или более в одиночный запрос, и даже один внутрь другого. Так как можно рассмотреть небольшой кусок чтобы получить всю картину работаты этой команды, вы можете воспользоваться способом в SQL, который может принимать различные команды из большинства других языков.

Имеется запрос, который извлекает строки всех продавцов которые имеют заказчиков с больше чем одним текущим порядком. Это не обязательно самое простое решение этой проблемы, но оно предназначено скорее показать улучшеную логику SQL. Вывод этой информации связывает все три наши типовых таблицы:

```
SELECT *
FROM Salespeople first
WHERE EXISTS (SELECT *
              FROM Customers second
              WHERE first.snum = second.snum
                    AND 1 < (SELECT COUNT (*)
                             FROM Orders
                             WHERE Orders.cnum = second.cnum));
```

Вывод для этого запроса показывается в Рисунке 12.5.

```
 ===============  SQL Execution Log ============
| FROM  Salespeople first                        |
| WHERE EXISTS                                   |
| (SELECT *                                      |
| FROM Customers second                          |
| WHERE first.snum = second.snum                 |
| AND 1 <                                        |
| (SELECT  CONT (*)                              |
| FROM Orders                                    |
| WHERE Orders.cnum = second.cnum));             |
| ============================================== |
|   cnum     cname     city         comm         |
|  -----    --------   ----       --------       |
|   1001    Peel       London         0.17       |
|   1002    Serres     San Jose       0.13       |
|   1007    Rifkin     Barselona      0.15       |
 ================================================
```

Рисунок 12.5: Использование EXISTS с комплексным подзапросом

Мы могли бы разобрать вышеупомянутый запрос примерно так:

Берем каждую строку таблицы Продавцов как строку-кандидат (внешний запрос) и выполняем подзапросы. Для каждой строки-кандидата из внешнего запроса, берем в соответствие каждую строку из таблицы Заказчиков (средний запрос). Если текущая строка заказчиков не совпадает с текущей строкой продавца (т.е. если first.snum < > second.snum), предикат среднего запроса неправилен. Всякий раз, когда мы находим заказчика в среднем запросе который совдает с продавцом во внешнем запросе, мы должны рассматривать сам внутренний запрос чтобы определить, будет ли наш средний предикат запроса верен. Внутренний запрос считает число порядков текущего заказчика (из среднего запроса). Если это число больший чем 1, предикат среднего запроса верен, и строки выбираются. Это делает EXISTS предикат внешнего запроса верным для текущей строки продавца, и означает, что по крайней мере один из текущих заказчиков продавца имеет более чем один порядок.

Если это не кажется достаточно понятным для вас в этой точке разбора примера, не волнуйтесь. Сложность этого примера — хороша независимо от того, как часто

будете Вы использовать ее в деловой ситуации. Основная цель примеров такого типа состоит в том, чтобы показать вам некоторые возможности которые могут оказаться в дальнейшем полезными. После работы со сложными ситуациями подобно этой, простые запросы которые являются наиболее часто используемыми в SQL, покажутся Вам элементарными.

Кроме того, этот запрос, даже если он кажется удобным, довольно извилистый способ извлечения информации и делает много работы. Он связывает три разных таблицы чтобы дать вам эту информацию, а если таблиц больше чем здесь указано, будет трудно получить ее напрямую (хотя это не единственный способ, и не обязательно лучший способ в SQL). Возможно вам нужно увидеть эту информацию относительно регулярной основы — если, например, вы имеете премию в конце недели для продавца который получил многочисленые порядки от одного заказчика. В этом случае, он должен был бы вывести команду, и сохранять ее чтобы использовать снова и снова по мере того как данные будут меняться (лучше всего сделать это с помощью представления, которое мы будем проходить в Главе 20).

РЕЗЮМЕ

EXISTS, хотя он и кажется простым, может быть одним из самых непонятных операторов SQL. Однако, он облажает гибкостью и мощностью. В этой главе, вы видели и овладели большинством возможностей которые EXISTS дает вам. В дальнейшем, ваше понимание улучшеной логики подзапроса расширится значительно.

Следующим шагом будет овладение тремя другими специальными операторами которые берут подзапросы как аргументы, это — ANY, ALL, и SOME. Как вы увидете в Главе 13, это — альтернативные формулировки некоторых вещей которые вы уже использовали, но которые в некоторых случаях, могут оказаться более предпочтительными.

РАБОТА С SQL

1. Напишите запрос который бы использовал оператор EXISTS для извлечения всех продавцов которые имеют заказчиков с оценкой 300.
2. Как бы вы решили предыдущую проблему используя обьединение ?
3. Напишите запрос использующий оператор EXISTS который выберет всех продавцов с заказчиками размещенными в их городах которые ими не обслуживаются.
4. Напишите запрос который извлекал бы из таблицы Заказчиков каждого заказчика назначенного к продавцу который в данный момент имеет по крайней мере еще одного заказчика (кроме заказчика которого вы выберете) с порядками в таблице Порядков (подсказка: это может быть похоже на структуру в примере с нашим трехуровневым подзапросом).

(См. Приложение A для ответов.)

13

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРОВ ANY, ALL И SOME

ТЕПЕРЬ, КОГДА ВЫ ОВЛАДЕЛИ ОПЕРАТОРОМ EXISTS, Вы узнаете приблизительно три специальных оператора ориентируемых на подзапросы. (Фактически, имеются только два, так как ANY и SOME — одно и то же.) Если вы поймете работу этих операторов, вы будете понимать все типы подзапросов предиката используемых в SQL. Кроме того, вы будете представлены различным способам где данный запрос может быть сформирован используя различные типы подзапросов предиката, и вы поймете преимущества и недостатки каждого из этих подходов.

ANY, **ALL**, и **SOME** напоминают EXISTS который воспринимает подзапросы как аргументы; однако они отличаются от EXISTS тем, что используются совместно с реляционными операторами. В этом отношении, они напоминают оператор IN когда тот используется с подзапросами; они берут все значения выведенные подзапросом и обрабатывают их как модуль. Однако, в отличие от IN, они могут использоваться только с подзапросами.

СПЕЦИАЛЬНЫЕ ОПЕРАТОРЫ ANY *или* SOME

Операторы SOME и ANY — взаимозаменяемы везде и там где мы используем ANY, SOME будет работать точно так же. Различие в терминологии состоит в том чтобы позволить людям использовать тот термин который наиболее однозначен. Это может создать проблему; потому что, как мы это увидим, наша интуиция может иногда вводить в заблуждение.

Имеется новый способ нахождения продавца с заказчиками размещенными в их городах (вывод для этого запроса показывается в Рисунке 13.1):

```
SELECT *
FROM Salespeople
WHERE city = ANY (SELECT city
                  FROM Customers );
```

Оператор ANY берет все значения выведенные подзапросом, (для этого случая — это все значения city в таблице Заказчиков), и оценивает их как верные если любой (ANY) из их равняется значению города текущей строки внешнего запроса.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Salespeople                              |
| WHERE city = ANY                               |
| (SELECT  city                                  |
| FROM Customers);                               |
| ============================================== |
|   cnum     cname     city         comm         |
|  -----    --------   ----       --------      |
|   1001    Peel       London         0.12      |
|   1002    Serres     San Jose       0.13      |
|   1004    Motika     London         0.11      |
 ================================================
```

Рисунок 13.1: Использование оператора ANY

Это означает, что подзапрос должен выбирать значения такого же типа как и те, которые сравниваются в основном предикате. В этом его отличие от EXISTS, который просто определяет, производит ли подзапрос результаты или нет, и фактически не использует эти результаты.

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРОВ IN ИЛИ EXISTS ВМЕСТО ОПЕРАТОРА ANY

Мы можем также использовать оператор IN чтобы создать запрос аналогичный предыдущему :

```
SELECT *
FROM Salespeople
WHERE city IN ( SELECT city
                FROM Customers );
```

Этот запрос будет производить вывод показанный в Рисунке 13.2.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM Salespeople                               |
| WHERE city IN (SELECT  city                    |
|                 FROM Customers);               |
| ============================================== |
|   cnum     cname     city         comm         |
|  -----    --------   ----       --------      |
|   1001    Peel       London         0.12      |
|   1002    Serres     San Jose       0.13      |
|   1004    Motika     London         0.11      |
 ================================================
```

Рисунок 13.2: Использование IN в качестве альтернативы к ANY

Однако, оператор ANY может использовать другие реляционные операторы кроме равняется (=), и таким образом делать сравнения которые являются выше возможностей IN. Например, мы могли бы найти всех продавцов с их заказчиками, которые следуют в алфавитном порядке (вывод показан на Рисунке 13.3)

```
SELECT *
FROM Salespeople
WHERE sname < ANY ( SELECT cname
                    FROM Customers );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Salespeople                              |
| WHERE sname < ANY                              |
| (SELECT  cname                                 |
| FROM Customers);                               |
| ============================================== |
|   cnum     cname     city         comm         |
|  -----    --------   ----       --------      |
|   1001    Peel       London         0.12      |
|   1004    Motika     London         0.11      |
|   1003    Axelrod    New York       0.10      |
 ================================================
```

Рисунок 13.3: Использование оператора ANY с оператором "неравно" (<)

Все строки были выбраны для Serres и Rifkin, потому что нет других заказчиков чьи имена следовали бы за ими в алфавитном порядке. Обратите внимание что это является основным эквивалентом следующему запросу с EXISTS, чей вывод показывается в Рисунке 13.4:

```
SELECT *
FROM Salespeople outer
WHERE EXISTS ( SELECT *
               FROM Customers inner
               WHERE outer.sname < inner.cname );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Salespeople outer                        |
| WHERE EXISTS                                   |
| (SELECT *                                      |
| FROM Customers inner                           |
| WHERE outer.sname < inner.cname);              |
| ============================================== |
|   cnum     cname     city         comm         |
|  -----    --------   ----       --------      |
|   1001    Peel       London         0.12      |
|   1004    Motika     London         0.11      |
|   1003    Axelrod    New York       0.10      |
 ================================================
```

Рисунок 13.4 Использование EXISTS как альтернатива оператору ANY

Любой запрос который может быть сформулирован с ANY (или, как мы увидим, с ALL), мог быть также сформулирован с EXISTS, хотя наоборот будет неверно. Строго говоря, вариант с EXISTS не абсолютно идентичен вариантам с ANY или с ALL из-за различия в том как обрабатываются пустые (NULL) значения (что будет обсуждаться позже в этой главе). Тем не менее, с технической точки зрения, вы могли бы делать это без ANY и ALL если бы вы стали очень находчивы в использовании EXISTS (и IS NULL).

Большинство пользователей, однако, находят ANY и ALL более удобными в использовании чем EXISTS, который требует соотнесенных подзапросов. Кроме того, в зависимости от реализации, ANY и ALL могут, по крайней мере в теории, быть более эффективными чем EXISTS.

Подзапросы ANY или ALL могут выполняться один раз и иметь вывод используемый чтобы определять предикат для каждой строки основного запроса. EXISTS, с другой стороны, берет соотнесенный подзапрос, который требует чтобы весь подзапрос повторно выполнялся для каждой строки основного запроса. SQL пытается найти наиболее эффективный способ выполнения любой команды, и может попробовать преобразовать менее эффективную формулу запроса в более эффективную (но вы не можете всегда рассчитывать на получение самой эффективной формулировки).

Основная причина для формулировки EXISTS как альтернативы ANY и ALL в том что ANY и ALL могут быть несколько неоднозначен, из-за способа использования этого термина в Английском языке, как вы это скоро увидите. С приходом понимания различия способов формулирования данного запроса, вы сможете поработать над процедурами которые сейчас кажутся Вам трудными или неудобными.

КАК ANY МОЖЕТ СТАТЬ НЕОДНОЗНАЧНЫМ

Как подразумевалось выше, ANY не полностью однозначен. Если мы создаем запрос, чтобы выбрать заказчиков, которые имеют больший рейтинг чем любой заказчик в Риме, мы можем получить, вывод который несколько отличался бы от того, что мы ожидали (как показано в Рисунке 13.5):

```
SELECT *
FROM Customers
WHERE rating > ANY ( SELECT rating
                     FROM Customers
                     WHERE city = Rome );
```

В Английском языке, способ которым мы обычно склонны интерпретировать оценку "больше чем ***любой*** (где city = Rome)", должен вам сообщить, что это значение оценки должно быть выше, чем значение оценки в каждом случае где значение city = Rome. Однако это не так, в случае ANY — используемом в SQL. ANY оценивает как верно, если подзапрос находит любое значение которое делает условие верным.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Customers                                |
| WHERE rating > ANY                             |
| (SELECT rating                                 |
| FROM Customers                                 |
| WHERE city = 'Rome');                          |
| ============================================== |
|   cnum     cname     city     rating   snum    |
|  -----    --------   ----     ------  ------   |
|   2002    Giovanni   Rome        200    1003   |
|   2003    Liu        San Jose    200    1002   |
|   2004    Grass      Berlin      300    1002   |
|   2008    Cisneros   San Jose    300    1007   |
 ================================================
```

Рисунок 13.5: Как оператор "*больше чем*" (**>**) интерпретируется ANY

Если мы оценим ANY способом, использующим грамматику Английского Языка, то только заказчики с оценкой 300 будут превышать Giovanni, который находится в Риме и имеет оценку 200. Однако, подзапрос ANY также находит Periera в Риме с оценкой 100. Так как все заказчики с оценкой 200 были выше этой, они будут выбраны, даже если имелся другой заказчик из Рима (Giovanni), чья оценка не была выше (фактически, то что один из выбранных заказчиков также находится в Риме несущественно). Так как подзапрос произвел по крайней мере одно значение, которое сделает предикат верным в отношении этих строк, строки были выбраны.

Чтобы дать другой пример, предположим что мы должны были выбирать все порядки сумм приоретений которые были больше чем по крайней мере один из порядков на 6-е Октября:

```
SELECT *
FROM Orders
WHERE amt > ANY ( SELECT amt
                  FROM Orders
                  WHERE odate = 10/06/1990 );
```

Вывод для этого запроса показывается в Рисунке 13.6.

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM  Orders                                    |
| WHERE amt > ANY                                 |
| (SELECT amt                                     |
| FROM Orders                                     |
| WHERE odate = 10/06/1990);                      |
| =============================================== |
|   onum       amt      odate      cnum     snum  |
|  -----    --------  ----------  -----   ------  |
|   3002     1900.10  10/03/1990   2007     1004  |
|   3005     5160.45  10/03/1990   2003     1002  |
|   3009     1713.23  10/04/1990   2002     1003  |
|   3008     4723.00  10/05/1990   2006     1001  |
|   3011     9891.88  10/06/1990   2006     1001  |
 ================================================
```

Рисунок 13.6: Выбранное значение больше чем *любое* (ANY) на 6-е Октября

Даже если самая высокая сумма приобретений в таблице (9891.88) — имелась на 6-е Октября, предыдущая строка имеет более высокое значение суммы чем другая строка на 6-е Октября, которая имела значение суммы = 1309.95. Имея реляционный оператор **">="** вместо просто **">"**, эта строка будет также выбирана, потому что она равна самой себе.

Конечно, вы можете использовать ANY с другой SQL техникой, например с техникой обьединения. Этот запрос будет находить все порядки со значением суммы меньшей чем значение любой суммы для заказчика в San Jose. (вывод показывается в Рисунке 13.7):

```
SELECT *
FROM Orders
WHERE amt < ANY ( SELECT amt
                  FROM Orders A, Customers b
                  WHERE a.cnum = b.cnum
                        AND b.city = “San Jose” );
```

Даже если нименьший порядок в таблице был для заказчика из San Jose, то был второй наибольший; следовательно почти все строки будут выбраны. Простой способ запомнить, что **< ANY** значение меньшее чем наибольшее выбранное значение, а **> ANY** значение большее чем наименьшее выбранное значение.

```
 ===============  SQL Execution Log ==============
| WHERE amt > ANY                                  |
| (SELECT amt                                      |
| FROM Orders a, Customers b                       |
| WHERE a.cnum = b.cnum                            |
| AND b.city = 'San Jose');                        |
| ================================================ |
|   onum       amt      odate      cnum     snum   |
|  -----    --------  ----------  -----   ------   |
|   3001       18.69  10/03/1990   2008     1007   |
|   3003      767.10  10/03/1990   2001     1001   |
|   3002     1900.10  10/03/1990   2007     1004   |
|   3006     1098.10  10/03/1990   2008     1007   |
|   3009     1713.23  10/04/1990   2002     1003   |
|   3007       75.10  10/04/1990   2004     1002   |
|   3008     4723.00  10/05/1990   2006     1001   |
|   3010     1309.88  10/06/1990   2004     1002   |
 ================================================
```

Рисунок 13.7: Использование ANY с объединением

Фактически, вышеуказанные команды весьма похожи на следующее — (вывод показан на Рисунке 13.8):

```
SELECT *
FROM Orders
WHERE amt < ( SELECT MAX amt
              FROM Orders A, Customers b
              WHERE a.cnum = b.cnum AND b.city = " San Jose' );
```

```
 ===============  SQL Execution Log ==============
| WHERE amt <                                      |
| (SELECT MAX (amt)                                |
| FROM Orders a, Customers b                       |
| WHERE a.cnum = b.cnum                            |
| AND b.city = 'San Jose');                        |
| ================================================ |
|   onum       amt      odate      cnum     snum   |
|  -----    --------  ----------  -----   ------   |
|   3002     1900.10  10/03/1990   2007     1004   |
|   3005     5160.45  10/03/1990   2003     1002   |
|   3009     1713.23  10/04/1990   2002     1003   |
|   3008     4723.00  10/05/1990   2006     1001   |
|   3011     9891.88  10/06/1990   2006     1001   |
 ================================================
```

Рисунок 13.8: Использование агрегатной функции вместо ANY

СПЕЦИАЛЬНЫЙ ОПЕРАТОР ALL

С помощью ALL, предикат является верным, если *каждое* значение выбранное подзапросом удовлетворяет условию в предикате внешнего запроса. Если мы хотим пересмотреть наш предыдущий пример чтобы вывести только тех заказчиков чьи оценки, фактически, выше чем у каждого заказчика в Париже, мы можем ввести следующее чтобы произвести вывод показанный в Рисунке 13.9:

```
SELECT *
FROM Customers
WHERE rating > ALL ( SELECT rating
                     FROM Customers
                     WHERE city = ‘Rome’ );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                      |
| FROM  Customers                               |
| WHERE rating > ALL                            |
| (SELECT rating                                |
| FROM Customers                                |
| WHERE city = 'Rome');                         |
| ============================================= |
|   cnum     cname     city     rating   snum   |
|  -----    --------   ----     ------  ------  |
|   2004    Grass      Berlin      300    1002  |
|   2008    Cisneros   San Jose    300    1007  |
 ===============================================
```

Рисунок 13.9: Использование оператора ALL

Этот оператор проверяет значения оценки всех заказчиков в Риме. Затем он находит заказчиков с оценкой большей чем у любого из заказчиков в Риме. Самая высокая оценка в Риме — у Giovanni (200). Следовательно, выбираются только значения выше этих 200.

Как и в случае с ANY, мы можем использовать EXISTS для производства альтернативной формулировки такого же запроса — (вывод показан на Рисунке 13.10):

```
SELECT *
FROM Customers outer
WHERE NOT EXISTS ( SELECT *
                   FROM Customers inner
                   WHERE outer.rating < = inner.rating
                         AND inner.city = ’Rome’ );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                      |
| FROM  Customers outer                         |
| WHERE NOT EXISTS                              |
| (SELECT *                                     |
| FROM Customers inner                          |
| WHERE outer rating = inner.rating             |
| AND inner.city = 'Rome');                     |
| ============================================= |
|   cnum     cname     city     rating   snum   |
|  -----    --------   ----     ------  ------  |
|   2004    Grass      Berlin      300    1002  |
|   2008    Cisneros   San Jose    300    1007  |
 ===============================================
```

Рисунок 13.10: Использование EXISTS в качестве альтернативы к ALL

РАВЕНСТВА И НЕРАВЕНСТВА

ALL используется в основном с неравенствами чем с равенствами, так как значение может быть "равным для всех" результатом подзапроса только если все результаты, фактически, идентичны. Посмотрите следующий запрос:

```
SELECT *
FROM Customers
WHERE rating = ALL ( SELECT rating
                     FROM Customers
                     WHERE city = ‘San Jose’ );
```

Эта команда допустима, но, с этими данными, мы не получим никакого вывода. Только в единственом случае вывод будет выдан этим запросом — если все значения оценки в San Jose окажутся идентичными. В этом случае, можно сказать следующее:

```
SELECT *
FROM Customers
WHERE rating = ( SELECT DISTINCT rating
                 FROM Customers
                 WHERE city = " San Jose' );
```

Основное различие в том, что эта последняя команда должна потерпеть неудачу, если подзапрос выведет много значений, в то время как вариант с ALL просто не даст никакого вывода. В общем, не самая удачная идея использовать запросы, которые работают только в определенных ситуациях, подобно этой. Так как ваша база данных будет постоянно меняться, это неудачный способ, чтобы узнать о ее содержании.

Однако, ALL может более эффективно использоваться с неравенствами, то есть с оператором "**<>**". Но учтите, что сказанное в SQL что — *значение которое не равняется всем результатам подзапроса*, — будет отличаться от того же но сказанного с учетом граматики Английского языка.

Очевидно, если подзапрос возвращает много различных значений, как это обычно бывает, ни одно отдельное значение не может быть равно им всем в обычном смысле. В SQL, выражение — **<> ALL** — в действительности соответствует *"не равен любому"* результату подзапроса. Другими словами, предикат верен, если данное значение не найдено среди результатов подзапроса. Следовательно, наш предыдущий пример противоположен по смыслу этому примеру (с выводом показанным в Рисунке 13.11):

```
SELECT *
FROM Customers
WHERE rating <> ALL ( SELECT rating
                      FROM Customers
                      WHERE city = “San Jose” );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                      |
| FROM  Customers                               |
| WHERE rating <> ALL                           |
| (SELECT rating                                |
| FROM Customers                                |
| WHERE city = 'San Jose');                     |
| ============================================= |
|   cnum     cname     city     rating   snum   |
|  -----    --------   ----     ------  ------  |
|   2001    Hoffman    London      100    1001  |
|   2006    Clemens    London      100    1001  |
|   2007    Pereira    Rome        100    1004  |
 ===============================================
```

Рисунок 13.11: Использование ALL с <>

Вышеупомянутый подзапрос выберает все оценки для города San Jose. Он выводит набор из двух значений: 200 (для Liu) и 300 (для Cisneros). Затем, основной за-

прос, выбирает все строки, с оценкой не совпадающей ни с одной из них — другими словами все строки с оценкой 100. Вы можете сформулировать тот же самый запрос используя оператор NOT IN:

```
SELECT*
FROM Customers
WHERE rating NOT IN ( SELECT rating
                      FROM Customers
                      WHERE city = “San Jose” );
```

Вы могли бы также использовать оператор ANY:

```
SELECT *
FROM Customers
WHERE NOT rating = ANY ( SELECT rating
                         FROM Customers
                         WHERE city = “San Jose” );
```

Вывод будет одинаков для всех трех условий.

ПРАВИЛЬНОЕ ПОНИМАНИЕ ANY И ALL

В SQL, сказать что — значение больше (или меньше) чем *любое* (ANY) из набора значений — тоже самое что сказать, что оно больше (или меньше) чем любое одно отдельное из этих значений. И наоборот, сказать что значение не равно *всему* (ALL) набору значений, тоже что сказать, что нет такого значения в наборе, которому оно равно.

КАК ANY, ALL, И EXIST ПОСТУПАЮТ С ОТСУТСТВУЮЩИМИ И НЕИЗВЕСТНЫМИ ДАННЫМИ

Как было сказано, имеются некоторые различия между EXISTS и операторами, представленными в этой главе, относительно того, как они обрабатывают оператор NULL. ANY и ALL также отличаются друг от друга тем, как они реагируют, если подзапрос не произвел никаких значений, чтобы использовать их в сравнении. Эти различия могут привести к непредвиденным результатам на Ваши запросы, если вы не будете их учитывать.

КОГДА ПОДЗАПРОС ВОЗВРАЩАЕТСЯ ПУСТЫМ

Одно значительное различие между ALL и ANY — способ действия в ситуации когда подзапрос не возвращает никаких значений. В принципе, всякий раз, когда допустимый подзапрос не в состоянии сделать вывод, ALL — автоматически верен, а ANY автоматически неправилен. Это означает, что следующий запрос

```
SELECT *
FROM Customers
WHERE rating > ANY ( SELECT rating
                     FROM Customers
                     WHERE city = “Boston” );
```

не произведет никакого вывода, в то время как запрос

```
SELECT
FROM Customers
WHERE rating > ALL ( SELECT rating
                     FROM Customers
                     WHERE city = 'Boston' );
```

выведет всю таблицу Заказчиков. Когда нет никаких заказчиков в Boston, естественно, ни одно из этих сравнений не имеет значення.

ANY и ALL вместо EXISTS с пустым указателем (NULL)

Значения NULL также имеют некоторые проблемы с операторами наподобии этих. Когда SQL сравнивает два значения в предикате, одно из которых пустое (NULL), то результат неизвестен (смотрите Главу 5). Неизвестный предикат, подобен неверному и является причиной того, что строка не выбирается, но работать он будет иначе в некоторых похожих запросах, в зависимости от того, используют они ALL или ANY вместо EXISTS. Рассмотрим наш предыдущий пример:

```
SELECT *
FROM Customers
WHERE rating > ANY ( SELECT rating
                     FROM Customers
                     WHERE city = 'Rome' );
```

и еще один пример:

```
SELECT *
FROM Customers outer
WHERE EXISTS ( SELECT *
               FROM Customers inner
               WHERE outer.rating > inner.rating
                     AND inner.city = 'Rome' );
```

В общем, эти два запроса будут вести себя одинаково. Но предположим, что появилось пустое (NULL) значение в столбце rating таблицы Заказчиков:

CNUM	*CNAME*	*CITY*	*RATING*	*SNUM*
2003	**Liu**	**SanJose**	**NULL**	**1002**

В варианте с ANY, где оценка Liu выбрана основным запросом, значение NULL делает предикат неизвестным, а строка Liu не выбирается для вывода. Однако, в варианте с NOT EXISTS, когда эта строка выбрана основным запросом, значение NULL используется в предикате подзапроса, делая его неизвестным в каждом случае. Это означает, что подзапрос не будет производить никаких значений, и EXISTS будет неправилен. Это, естественно, делает оператор NOT EXISTS верным. Следовательно, строка Liu будет выбрана для вывода. Это основное расхождение, в отличие от других типов предикатов, где значение EXISTS независимо от того верно оно или нет — всегда неизвестно. Все это является аргументом в пользу использования варианта формулировки с ANY. Мы не считаем что значение NULL является выше чем допустимое значение. Более того, результат будет тот же, если мы будем проверять для более низкого значения.

ИСПОЛЬЗОВАНИЕ COUNT ВМЕСТО EXISTS

Подчеркнем, что все формулировки с ANY и ALL могут быть в точности выполнены с EXISTS, в то время как наоборот будет неверно. Хотя в этом случае, также верно и то что EXISTS и NOT EXISTS подзапросы могут обманывать при выполнении тех же самых подзапросов с COUNT (*) в предложения SELECT подзапроса. Если больше чем ноль строк выводе будет подсчитано, это эквивалентно EXISTS; в противном случае это работает также как NOT EXISTS. Следующее является этому примером (вывод показывается в Рисунке 13.12):

```
SELECT *
FROM Customers outer
WHERE NOT EXISTS ( SELECT *
                   FROM Customers inner
                   WHERE outer.rating < = inner.rating
                         AND inner.city = 'Rome' );
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Customers outer                          |
| WHERE NOT EXISTS                               |
| (SELECT *                                      |
| FROM Customers inner                           |
| WHERE outer.rating <= inner.rating             |
| AND inner.city = 'Rome');                      |
| ============================================== |
|   cnum     cname     city     rating   snum    |
|  -----    --------   ----     ------  ------   |
|   2004    Grass      Berlin      300    1002   |
|   2008    Cisneros   San Jose    300    1007   |
 ================================================
```

Рисунок 13.12: Использование EXISTS с соотнесенным подзапросом

Это может также быть выполнено как

```
SELECT *
FROM Customers outer
WHERE 1 > ( SELECT COUNT (*)
            FROM Customers inner
            WHERE outer.rating < = inner.rating
                  AND inner.city = 'Rome' );
```

Вывод к этому запросу показывается в Рисунке 13.13. Теперь Вы начинаете понимать, сколько способов имеется в SQL. Если это все кажется несколько путанным на этой стадии, нет причины волноваться. Вы обучаетесь, чтобы использовать ту технику, которая лучше всего отвечает вашим требованиям и наиболее понятна для вас. Начиная с этого места, мы хотим показать Вам большое количество возможностей, что бы вы могли найти ваш собственный стиль.

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Customers outer                          |
| WHERE 1 >                                      |
| (SELECT COUNT (*)                              |
| FROM Customers inner                           |
| WHERE outer.rating <= inner.rating             |
| AND inner.city = 'Rome');                      |
| ============================================== |
|   cnum     cname     city     rating   snum    |
|  -----    --------   ----     ------  ------   |
|   2004    Grass      Berlin      300    1002   |
|   2008    Cisneros   San Jose    300    1007   |
 ================================================
```

Рисунок 13.13: Использование COUNT вместо EXISTS

РЕЗЮМЕ

Итак, вы прошли много чего в этой главе. Подзапросы не простая тема, и мы потратили много время чтобы показать их разновидности и неоднозначности. То чему Вы теперь научились, вещи достаточно глубокие. Вы знаете несколько технических решений одной проблемы, и поэтому вы можете выбрать то которое более подходит вашим целям. Кроме того, вы поняли, как различные формулировки будут обрабатывать *пустые* значения (NULL) и ошибки.

Теперь, когда вы полностью изучили запросы, наиболее важный, и вероятно наиболее сложный, аспект SQL, объем другого материала будет относительно прост для понимания.

Мы имеем еще одну главу о запросах, которая покажет вам как объединить выводы любого числа запросов в единое тело, с помощью формирования объединения многочисленых запросов используя оператор UNION.

РАБОТА С SQL

1. Напишите запрос, который бы выбирал всех заказчиков чьи оценки равны или больше чем *любая* (ANY) оценка заказчика Serres.
2. Что будет выведено вышеупомянутой командой?
3. Напишите запрос, использующий ANY или ALL, который бы находил всех продавцов, которые не имеют никаких заказчиков, размещенных в их городе.
4. Напишите запрос, который бы выбирал все порядки с суммой больше чем любая (в обычном смысле) для заказчиков в Лондоне.
5. Напишите предыдущий запрос с использованием MAX.

(См. Приложение A для ответов.)

14

ИСПОЛЬЗОВАНИЕ ПРЕДЛОЖЕНИЯ UNION

В ПРЕДШЕСТВУЮЩИХ ГЛАВАХ МЫ ОБСУЖДАЛИ различные способы, которыми запросы могут помещаться один внутрь другого. Имеется другой способ объединения многочисленых запросов — т.е. формирование их в объединение. В этой главе вы научитесь использованию предложения UNION в SQL. **UNION** отличается от подзапросов тем что в нем ни один из двух (или больше) запросов не управляются другим запросом. Все запросы выполняются независимо друг от друга, а уже вывод их — обьединяется.

ОБЪЕДИНЕНИЕ МНОГОЧИСЛЕННЫХ ЗАПРОСОВ В ОДИН

Вы можете поместить многочисленые запросы вместе и объединить их вывод, используя предложение UNION. Предложение UNION обьединяет вывод двух или более SQL запросов в единый набор строк и столбцов. Например, чтобы получить всех продавцов и заказчиков размещенных в Лондоне и вывести их как единое целое вы могли бы ввести:

```
SELECT snum, sname
FROM Salespeople
WHERE city = 'London'

UNION

SELECT cnum, cname
FROM Customers
WHERE city = 'London';
```

и получить вывод показанный в Рисунке 14.1.

Как вы можете видеть, столбцы выбранные двумя командами выведены так, как если она была одна. Заголовки столбца исключены, потому что ни один из столбцов выведенных объединением, не был извлечен непосредственно из только одной таблицы. Следовательно все эти столбцы вывода не имеют никаких имен (смотрите Главу 7, обсуждающую вывод столбцов).

Кроме того обратите внимание, что только последний запрос заканчивается точкой с запятой. Отсутствие точки с запятой дает понять SQL, что имеется еще одно или более запросов.

```
 ===============  SQL Execution Log ============
|                                               |
| SELECT snum, sname                            |
| FROM  Salespeople                             |
| WHERE city = 'London'                         |
| UNION                                         |
| SELECT cnum, cname                            |
| FROM Customers                                |
| WHERE city = 'London';                        |
| ============================================= |
|                                               |
|  -----    --------                            |
|   1001    Peel                                |
|   1004    Motika                              |
|   2001    Hoffman                             |
|   2006    Climens                             |
|                                               |
 ===============================================
```

Рисунок 14.1: Формирование объединения из двух запросов

КОГДА ВЫ МОЖЕТЕ ДЕЛАТЬ ОБЪЕДИНЕНИЕ МЕЖДУ ЗАПРОСАМИ?

Когда два (или более) запроса подвергаются объединению, их столбцы вывода должны быть совместимы для объединения. Это означает, что каждый запрос должен указывать одинаковое число столбцов и в том же порядке что и первый, второй, третий, и так далее, и каждый должен иметь тип, совместимый с каждым. Значение совместимости типов — меняется. ANSI следит за этим очень строго и поэтому числовые поля должны иметь одинаковый числовой тип и размер, хотя некоторые имена используемые ANSI для этих типов являются синонимами. (Смотрите Приложение B для подробностей об ANSI числовых типах.) Кроме того, символьные поля должны иметь одинаковое число символов (значение предназначенного номера, не обязательно такое же как используемый номер).

Хорошо, что некоторые SQL программы обладают большей гибкостью, чем это определяется ANSI. Типы не определенные ANSI, такие как DATA и BINARY, обычно должны совпадать с другими столбцами такого же нестандартного типа.

Длина строки также может стать проблемой. Большинство программ разрешают поля переменной длины, но они не обязательно будут использоваться с UNION. С другой стороны, некоторые программы (и ANSI тоже) требуют чтобы символьные поля были точно равной длины. В этих вопросах вы должны проконсультироваться с документацией вашей собственной программы.

Другое ограничение на совместимость — это когда пустые значения (NULL) запрещены в любом столбце объединения, причем эти значения необходимо запретить и для всех соответствующих столбцов в других запросах объединения. Пустые значения (NULL) запрещены с ограничением NOT NULL, которое будет обсуждаться в Главе 18. Кроме того, вы не можете использовать UNION в подзапросах, а также не можете использовать агрегатные функции в предложении SELECT запроса в объединении. (Большинство программ пренебрегают этими ограничениями.)

UNION И УСТРАНЕНИЕ ДУБЛИКАТОВ

UNION будет автоматически исключать дубликаты строк из вывода. Это нечто несвойственное для SQL, так как одиночные запросы обычно содержат DISTINCT чтобы устранять дубликаты. Например запрос, чей вывод показывается в Рисунке 14.2,

```
SELECT snum, city
FROM Customers;
```

имеет двойную комбинацию значений (snum=1001, city=London), потому что мы не указали, чтобы SQL устранил дубликаты. Однако, если мы используем UNION в комбинации этого запроса с ему подобным в таблице Продавцов, то эта избыточная комбинация будет устранена. Рисунок 14.3 показывает вывод следующего запроса.

```
SELECT snum, city
FROM Customers

UNION

SELECT snum, city
FROM Salespeople.;
```

```
 ===============  SQL Execution Log ============
| SELECT snum, city                              |
| FROM Customers;                                |
| ============================================== |
|  snum     city                                 |
|  -----    --------                             |
|   1001    London                               |
|   1003    Rome                                 |
|   1002    San Jose                             |
|   1002    Berlin                               |
|   1001    London                               |
|   1004    Rome                                 |
|   1007    San Jose                             |
 ================================================
```

Рисунок 14.2: Одиночный запрос с дублированным выводом

```
 ===============  SQL Execution Log ============
| FROM Customers                                 |
| UNION                                          |
| SELECT snum, sity                              |
| FROM Salespeople;                              |
| ============================================== |
|                                                |
|  -----    --------                             |
|   1001    London                               |
|   1002    Berlin                               |
|   1007    San Jose                             |
|   1007    New York                             |
|   1003    Rome                                 |
|   1001    London                               |
|   1003    Rome                                 |
|   1002    Barcelona                            |
|   1007    San Jose                             |
 ================================================
```

Рисунок 14.3: UNION устраняет двойной вывод

Вы можете получить нечто похожее (в некоторых программах SQL, используя UNION ALL вместо просто UNION, наподобии этого:

```
SELECT snum, city
FROM Customers

UNION ALL

SELECT snum, city
FROM Salespeople;
```

ИСПОЛЬЗОВАНИЕ СТРОК И ВЫРАЖЕНИЙ С UNION

Иногда, вы можете вставлять константы и выражения в предложения SELECT используемые с UNION. Это не следует строго указаниям ANSI, но это полезная и необычно используемая возможность. Константы и выражения которые вы используете, должны встречать совместимые стандарты которые мы выделяли ранее. Эта свойство полезно, например, чтобы устанавливать комментарии указывающие какой запрос вывел данную строку.

Предположим что вы должны сделать отчет о том, какие продавцы производят наибольшие и наименьшие порядки по датам. Мы можем объединить два запроса, вставив туда текст чтобы различать вывод для каждого из них.

```
SELECT a.snum, sname, onum, 'Highest on', odate
FROM Salespeople a, Orders b
WHERE a.snum = b.snum AND b.amt = ( SELECT MAX (amt)
                                    FROM Orders c
                                    WHERE c.odate = b.odate )

UNION

SELECT a.snum, (sname, (onum ' Lowest on', odate
FROM ( Salespeople a, Orders b
WHERE a.snum = b.snum AND b.amt = ( SELECT  MIN (amt)
                                    FROM Orders c
                                    WHERE c.odate = b.odate );
```

Вывод из этой команды показывается в Рисунке 14.4.

Мы должны были добавить дополнительный пробел в строку 'Lowest on', чтобы сделать ее совпадающей по длине со строкой 'Highest on'. Обратите внимание что Peel выбран при наличии и самого высокого и самого низкого (фактически он единственый) порядка на 5 Октября. Так как вставляемые строки двух этих запросов различны, строки не будут устранены как дубликаты.

```
 ===============  SQL Execution Log ============
| AND b.amt =                                    |
| ( SELECT min (amt)                             |
| FROM Orders c                                  |
| WHERE c.odate = b.odate);                      |
| ============================================== |
|                                                |
| -----  ------- ------  ---------- ----------- |
|  1001  Peel     3008   Highest on  10/05/1990 |
|  1001  Peel     3008   Lowest  on  10/05/1990 |
|  1001  Peel     3011   Highest on  10/06/1990 |
|  1002  Serres   3005   Highest on  10/03/1990 |
|  1002  Serres   3007   Lowest  on  10/04/1990 |
|  1002  Serres   3010   Lowest  on  10/06/1990 |
|  1003  Axelrod  3009   Highest on  10/04/1990 |
|  1007  Rifkin   3001   Lowest  on  10/03/1990 |
 ================================================
```

Рисунок 14.4: Выбор наивысших и наинизших порядков, определяемых с помощью строк

ИСПОЛЬЗОВАНИЕ UNION С ORDER BY

До сих пор, мы не оговаривали, что данные многочисленых запросов будут выводиться в каком то особом порядке. Мы просто показывали вывод сначала из одного запроса, а затем из другого. Конечно, вы не можете полагаться на вывод, приходящий в произвольном порядке. Мы как раз сделаем так, чтобы этот способ для выполнения примеров был более простым. Вы можете использовать предложение ORDER BY чтобы упорядочить вывод из объединения, точно так же как это делается в индивидуальных запросах. Давайте пересмотрим наш последний пример чтобы упорядочить имена с помощью их порядковых номеров. Это может внести противоречие, такое как повторение имени Peel в последней команде, как вы сможете увидеть из вывода показанного в Рисунке 14.5.

```
SELECT a.snum, sname, onum, 'Highest on', odate
FROM Salespeople a, Orders b
WHERE a.snum = b.snum AND b.amt = ( SELECT MAX (amt)
                                    FROM Orders c
                                    WHERE c.odate = b.odate )

UNION

SELECT a.snum, sname, onum, ' Lowest on', odate
FROM Salespeople a, Orders b
WHEREa.snum = b.snum AND b.amt = ( SELECT MIN (amt)
                                   FROM Orders c
                                   WHERE c.odate = b.odate )
ORDER BY 3;
```

```
 ===============  SQL Execution Log ============
| ( SELECT min (amt)                             |
| FROM Orders c                                  |
| WHERE c.odate = b.odate)                       |
| ORDER BY 3;                                    |
| ============================================== |
|                                                |
| -----  ------- ------  ---------- ----------- |
|  1007  Rifkin   3001   Lowest  on  10/03/1990 |
|  1002  Serres   3005   Highest on  10/03/1990 |
|  1002  Serres   3007   Lowest  on  10/04/1990 |
|  1001  Peel     3008   Highest on  10/05/1990 |
|  1001  Peel     3008   Lowest  on  10/05/1990 |
|  1003  Axelrod  3009   Highest on  10/04/1990 |
|  1002  Serres   3010   Lowest  on  10/06/1990 |
|  1001  Peel     3011   Highest on  10/06/1990 |
 ================================================
```

Рисунок 14.5: Формирование объединения с использованием ORDER BY

Пока ORDER BY используется по умолчанию, мы не должны его указывать. Мы можем упорядочить наш вывод с помощью нескольких полей, одно внутри другого и указать ASC или DESC для каждого, точно также как мы делали это для одиночных запросов. Заметьте, что номер 3 в предложении ORDER BY указывает какой столбец из предложения SELECT будет упорядочен. Так как столбцы объединения — это столбцы вывода, они не имеют имен, и следовательно, должны определяться по номеру. Этот номер указывает на их место среди других столбцов вывода. (Смотрите Главу 7, обсуждающую столбцы вывода.)

ВНЕШНЕЕ ОБЪЕДИНЕНИЕ

Операция, которая бывает часто полезна — это объединение из двух запросов, в котором второй запрос выбирает строки, исключенные первым. Наиболее часто, вы будете делать это, так чтобы не исключать строки, которые не удовлетворили предикату при объединении таблиц. Это называется *внешним обьединением*.

Предположим что некоторые из ваших заказчиков еще не были назначены к продавцам. Вы можете захотеть увидеть имена и города всех ваших заказчиков, с именами их продавцов, не учитывая тех, кто еще не был назначен. Вы можете достичь этого, формируя объединение из двух запросов, один из которых выполняет обьединение, а другой выбирает заказчиков с пустыми (NULL) значениями поля snum. Этот последний запрос должен вставлять пробелы в поля, соответствующие полю sname в первом запросе.

Как и раньше, вы можете вставлять текстовые строки в ваш вывод, чтобы идентифицировать запрос который вывел данную строку. Использование этой методики во внешнем обьединении дает возможность использовать предикаты для классификации, а не для исключения. Мы использовали пример нахождения продавцов с заказчиками размещенными в их городах и раньше. Однако вместо просто выбора только этих строк, вы возможно захотите чтобы ваш вывод перечислял всех продавцов, и указывал тех, кто не имел заказчиков в их городах, и кто имел. Следующий запрос, чей вывод показывается в Рисунке 14.6, выполнит это:

```
SELECT Salespeople.snum, sname, cname, comm
FROM Salespeople, Customers
WHERE Salespeople.city = Customers.city

UNION

SELECT snum, sname, ' NO MATCH    ', comm
FROM Salespeople
WHERE NOT city = ANY ( SELECT city
                       FROM Customers )
ORDER BY 2 DESC;
```

```
 ===============  SQL Execution Log ============
| FROM Salespeople                               |
| WHERE NOT city = ANY ( SELECT city             |
|                        FROM Customers )        |
| ORDER BY 2 DESC;                               |
| ============================================== |
|                                                |
| -----  -------  ---------     ------------     |
|  1002  Serres   Cisneros           0.1300      |
|  1002  Serres   Liu                0.1300      |
|  1007  Rifkin   NO MATCH           0.1500      |
|  1001  Peel     Clemens            0.1200      |
|  1001  Peel     Hoffman            0.1200      |
|  1004  Motika   Clemens            0.1100      |
|  1004  Motika   Hoffman            0.1100      |
|  1003  Axelrod  NO MATCH           0.1000      |
 ================================================
```

Рисунок 14.6: Внешнее обьединение

Строка 'NO MATCH' была дополнена пробелами, чтобы получить совпадение поля cname по длине (это не обязательно во всех реализациях SQL). Второй запрос выбирает даже те строки, которые исключил первый. Вы можете также добавить комментарий или выражение к вашему запросу, в виде дополнительного поля. Если вы сделаете это, вы будете должны добавить некоторый дополнительный комментарий или выражение в той же самой позиции среди выбранных полей, для каждого запроса в операции объединения. Совместимость UNION предотвращает вас от добавления дополнительного поля для первого запроса, но не для второго. Имеется запрос, который добавляет строки к выбранным полям, и указывает, совпадает ли данный продавец с его заказчиком в его городе:

```
SELECT a.snum, sname, a.city, ' MATCHED '
FROM Salespeople a, Customers b
WHERE a.city = b.city

UNION

SELECT snum, sname, city, 'NO MATCH'
FROM Salespeople
WHERE NOT city = ANY ( SELECT city
                       FROM Customers )
ORDER BY 2 DESC;
```

Рисунок 14.7 показывает вывод этого запроса.

```
 ===============  SQL Execution Log ============
| WHERE a.city = b.city                          |
| UNION                                          |
| SELECT snum,sname,city, 'NO MATCH'             |
| FROM Salespeople                               |
| WHERE NOT city = ANYate)                       |
| ( SELECT city                                  |
| FROM Customers)                                |
| ORDER BY 2 DESC;                               |
| ============================================== |
|                                                |
| -----  -------   ------------  ---------       |
|  1002  Serres     San Jose     MATCHED         |
|  1007  Rifkin     Barselona    NO MATCH        |
|  1001  Peel       London       MATCHED         |
|  1004  Motika     London       MATCHED         |
|  1003  Axelrod    New York     NO MATCH        |
 ================================================
```

Рисунок 14.7: Внешнее обьединение с полем коментария

Это не полное внешнее объединение, так как оно включает только несовпадающие поля одной из объединяемых таблиц. Полное внешнеее объединение должно включать всех заказчиков, имеющих и не имеющих продавцов в их городах. Такое условие будет более полным, как вы это сможете увидеть (вывод следующего запроса показан на Рисунке 14.8):

```
SELECT snum, city, 'SALESPERSON — MATCH'
FROM Salespeople
WHERE NOT city = ANY (SELECT city
                      FROM Customers)

UNION

SELECT snum, city, 'SALESPERSON — NO MATCH'
FROM Salespeople
WHERE NOT city = ANY (SELECT city
                      FROM Customers))

UNION

SELECT cnum, city, 'CUSTOMER — MATCHED'
FROM Customers
WHERE city = ANY (SELECT city
                  FROM Salespeople)

UNION

SELECT cnum, city, 'CUSTOMER — NO MATCH'
FROM Customers
WHERE NOT city = ANY (SELECT city
                      FROM Salespeople))

ORDER BY 2 DESC;
```

```
 =================  SQL Execution Log =============
| FROM  Salespeople)                                 |
| ORDER BY 2 DESC;                                   |
| ================================================== |
|                                                    |
|  ----   --------     ------------------------      |
|  2003   San Jose     CUSTOMER     —   MATCHED      |
|  2008   San Jose     CUSTOMER     —   MATCHED      |
|  2002   Rome         CUSTOMER     —   NO MATCH     |
|  2007   Rome         CUSTOMER     —   NO MATCH     |
|  1003   New York     SALESPERSON  —   MATCHED      |
|  1003   New York     SALESPERSON  —   NO MATCH     |
|  2001   London       CUSTOMER     —   MATCHED      |
|  2006   London       CUSTOMER     —   MATCHED      |
|  2004   Berlin       CUSTOMER     —   NO MATCH     |
|  1007   Barcelona    SALESPERSON  —   MATCHED      |
|  1007   Barcelona    SALESPERSON  —   NO MATCH     |
 ================================================
```

Рисунок 14.8: Полное внешнее обьединение

(Понятно, что эта формула, использующая ANY, — эквивалентна обьединению в предыдущем примере.)

Сокращенное внешнее обьединение, с которого мы начинали, используется чаще чем этот последний пример. Этот пример, однако, имеет другой смысл. Всякий раз, когда вы выполняете объединение более чем двух запросов, вы можете использовать круглые скобки чтобы определить порядок оценки. Другими словами, вместо просто —

```
query X UNION query Y UNION query Z;
```

вы должны указать, или

```
( query X UNION query Y )UNION query Z;
```

или

```
query X UNION ( query Y UNION query Z );
```

Это потому, что UNION и UNION ALL могут быть скомбинированны, чтобы удалять одни дубликаты, не удаляя других. Предложение

```
( query X UNION ALL query Y )UNION query Z;
```

не обязательно воспроизведет те же результаты что предложение

```
query X UNION ALL( query Y UNION query Z );
```

если двойные строки в нем, будут удалены.

РЕЗЮМЕ

Теперь вы знаете, как использовать предложение UNION, которое дает возможность объединять любое число запросов в единое тело вывода. Если вы имеете ряд подобных таблиц — таблиц, содержащих похожую информацию, но принадлежащую разным пользователям и охватывающую различные особенности, возможно — что объединение сможет обеспечить простой способ для слияния и упорядочивания вывода. Аналогично, внешние обьединения дают вам новый способ использования условий, не для исключения вывода, а для его маркировки или обработки его частей, когда встречается условие отличающееся от того, которое не выполняется.

Этим заканчиваются наши главы о запросах. Вы теперь имеете довольно полное представление о поиске данных в SQL. Следующий шаг должен включать то, как значения вводятся в таблицы и как таблицы создаются с самого начала. Как вы увидете, запросы иногда используются внутри других типов команд, также хорошо как и сами по себе.

РАБОТА С SQL

1. Создайте объединение из двух запросов которое показало бы имена, города, и оценки всех заказчиков. Те из них, которые имеют поле rating=200 и более, должны кроме того иметь слова "Высокий Рейтинг", а остальные должны иметь слова "Низкий Рейтинг".
2. Напишите команду которая бы вывела имена и номера каждого продавца и каждого заказчика, которые имеют больше чем один текущий порядок. Результат представьте в алфавитном порядке.
3. Сформируйте объединение из трех запросов. Первый выбирает поля snum всех продавцов в San Jose; второй, поля cnum всех заказчиков в San Jose; и третий поля onum всех порядков на 3 Октября. Сохраните дубликаты между последними двумя запросами, но устраните любую избыточность вывода между каждым из их и самым первым. (Примечание: в данных типовых таблицах не содержится никакой избыточности. Это только пример.)

(См. Приложение A для ответов.)

15

ВВОД, УДАЛЕНИЕ И ИЗМЕНЕНИЕ ЗНАЧЕНИЙ ПОЛЕЙ

ЭТА ГЛАВА ПРЕДСТАВЛЯЕТ КОМАНДЫ, КОТОРЫЕ управляют значениями, представляемыми в таблице. Когда вы закончите эту главу, вы будете способны помещать строки в таблицу, удалять их, и изменять индивидуальные значения, представленные в каждой строке.

Будет показано использование запросов в формировании полной группы строк для вставки, а также, как может использоваться предикат для управления изменения значений и удаления строк. Материал в этой главе составляет полный объем знаний показывающий, как создавать и управлять информацией в базе данных.

Более мощные способы проектирования предикатов будут обсуждены в следующей главе.

КОМАНДЫ МОДИФИКАЦИИ ЯЗЫКА DML

Значения могут быть помещены и удалены из полей, тремя командами языка DML (Язык Манипулирования Данными):

```
INSERT (ВСТАВИТЬ),
UPDATE (МОДИФИЦИРОВАТЬ),
DELETE (УДАЛИТЬ).
```

Не смущайтесь, все они упоминались ранее в SQL, как команды *модификации*.

ВВОД ЗНАЧЕНИЙ

Все строки в SQL вводятся с использованием команды модификации INSERT. В самой простой форме, INSERT использует следующий синтаксис:

```
INSERT INTO <table name>
VALUES ( <value>, <value> . . .);
```

Так, например, чтобы ввести строку в таблицу Продавцов, вы можете использовать следующее условие:

```
INSERT INTO Salespeople
VALUES (1001, 'Peel', 'London', .12);
```

Команды DML не производят никакого вывода, но ваша программа должна дать вам некоторое подтверждение того, что данные были использованы.

Имя таблицы (в нашем случае — Salespeople (Продавцы)), должно быть предварительно определено в команде CREATE TABLE (см. Главу 17), а каждое значение пронумерованное в предложении значений, должно совпадать с типом данных столбца, в который оно вставляется. В ANSI, эти значения не могут составлять выражений, что означает что 3 — это доступно, а выражение 2 + 1 — нет. Значения, конечно же, вводятся в таблицу в поименном порядке, поэтому первое значение с именем, автоматическми попадает в столбец 1, второе в столбец 2, и так далее.

ВСТАВКА ПУСТЫХ УКАЗАТЕЛЕЙ (NULL)

Если вам нужно ввести *пустое* значение (NULL), вы вводите его точно также как и обычное значение. Предположим, что еще не имелось поля city для мистера Peel. Вы можете вставить его строку со значением=NULL в это поле, следующим образом:

```
INSERT INTO Salespeople
VALUES (1001, 'Peel', NULL, .12);
```

Так как значение NULL — это специальный маркер, а не просто символьное значение, он не включается в одиночные кавычки.

ИМЕНОВАНИЕ СТОЛБЦА ДЛЯ ВСТАВКИ (INSERT)

Вы можете также указывать столбцы, куда вы хотите вставить значение имени. Это позволяет вам вставлять имена в любом порядке. Предположим, что вы берете значения для таблицы Заказчиков из отчета, выводимого на принтер, который помещает их в таком порядке: **city**, **cname**, и **cnum**, и для упрощения, вы хотите ввести значения в том же порядке:

```
INSERT INTO Customers (city, cname, cnum)
VALUES ('London', 'Honman', 2001);
```

Обратите внимание, что столбцы rating и snum — отсутствуют. Это значит, что эти строки автоматически установлены в значение по умолчанию. По умолчанию может быть введено или значение NULL или другое значение, определяемое как значение по умолчанию. Если ограничение запрещает использование значения NULL в данном столбце, и этот столбец не установлен как по умолчанию, этот столбец должен быть обеспечен значением для любой команды INSERT, которая относится к таблице (смотри Главу 18 для информации об ограничениях на NULL и на "по умолчанию").

ВСТАВКА РЕЗУЛЬТАТОВ ЗАПРОСА

Вы можете также использовать команду INSERT чтобы получать или выбирать значения из одной таблицы и помещать их в другую, чтобы использовать их вместе с запросом. Чтобы сделать это, вы просто заменяете предложение VALUES (из предыдущего примера) на соответствующий запрос:

```
INSERT INTO Londonstaff
SELECT *
FROM Salespeople
WHERE city = 'London';
```

Здесь выбираются все значения произведенные запросом — то-есть все строки из таблицы Продавцов со значениями city = "London" — и помещаются в таблицу называемую Londonstaff. Чтобы это работало, таблица Londonstaff должна отвечать следующим условиям:

* Она должна уже быть создана командой CREATE TABLE.

* Она должна иметь четыре столбца которые совпадают с таблицей Продавцов в терминах типа данных; то-есть первый, второй, и так далее, столбцы каждой таблицы, должны иметь одинаковый тип данных (причем они не должны иметь одинаковых имен).

Общее правило то, что всталяемые столбцы таблицы, должны совпадать со столбцами выводимыми подзапросом, в данном случае, для всей таблицы Продавцов.

Londonstaff — это теперь независимая таблица, которая получила некоторые значения из таблицы Продавцов (Salespeople). Если значения в таблице Продавцов будут вдруг изменены, это никак не отразится на таблице Londonstaff (хотя вы могли бы создать такой эффект с помощью Представления (VIEW), описанного в Главе 20).

Так как или запрос или команда INSERT могут указывать столбцы по имени, вы можете, если захотите, переместить только выбранные столбцы, а также переупорядочить только те столбцы, которые вы выбрали.

Предположим, например, что вы решили сформировать новую таблицу с именем Daytotals, которая просто будет следить за общим количеством долларов сумм приобретений упорядоченных на каждый день. Вы можете ввести эти данные независимо от таблицы Порядков, но сначала вы должны заполнить таблицу Daytotals информацией ранее представленной в таблице Порядков.

Понимая, что таблица Порядков охватывает последний финансовый год, а не только несколько дней, как в нашем примере, вы можете видеть преимущество использования следующего условия INSERT в подсчете и вводе значений

```
INSERT INTO Daytotals (date, total)
SELECT odate, SUM (amt)
FROM Orders
GROUP BY odate;
```

Обратите внимание что, как предложено ранее, имена столбцов таблицы Порядков и таблицы Daytotals — не должны быть одинаковыми. Кроме того, если дата приобретения и общее количество — это единственые столбцы в таблице, и они находятся в данном порядке, их имена могут быть исключены из вывода из-за их очевидной простоты.

УДАЛЕНИЕ СТРОК ИЗ ТАБЛИЦ

Вы можете удалять строки из таблицы командой модификации — DELETE. Она может удалять только введеные строки, а не индивидуальные значения полей, так что параметр поля является необязательным или недоступным. Чтобы удалить все содержание таблицы Продавцов, вы можете ввести следующее условие:

```
DELETE FROM Salespeople;
```

Теперь, когда таблица пуста, ее можно окончательно удалить командой DROP TABLE (это объясняется в Главе 17).

Обычно, вам нужно удалить только некоторые определенные строки из таблицы. Чтобы определить какие строки будут удалены, вы используете предикат, так же как вы это делали для запросов. Например, чтобы удалить продавца Axelrod из таблицы, вы можете ввести

```
DELETE FROM Salespeople
WHERE snum = 1003;
```

Мы использовали поле snum вместо поля sname потому, что это лучшая тактика при использовании первичных ключей когда вы хотите чтобы действию подвергалась одна и только одна строка. Для вас — это аналогично действию первичного ключа.

Конечно, вы можете также использовать DELETE с предикатом, который бы выбирал группу строк, как показано в этом примере:

```
DELETE FROM Salespeople
WHERE city = 'London';
```

ИЗМЕНЕНИЕ ЗНАЧЕНИЙ ПОЛЯ

Теперь, когда вы уже можете вводить и удалять строки таблицы, вы должны узнать, как изменять некоторые или все значения в существующей строке. Это выполняется командой **UPDATE**.

Эта команда содержит предложение UPDATE, в котором указано имя используемой таблицы и предложение SET, которое указывает на изменение, которое нужно сделать для определенного столбца. Например, чтобы изменить оценки всех заказчиков на 200, вы можете ввести

```
UPDATE Customers
SET rating = 200;
```

МОДИФИЦИРОВАНИЕ ТОЛЬКО ОПРЕДЕЛЕННЫХ СТРОК

Конечно, вы не всегда захотите указывать все строки таблицы для изменения единственного значения, так что UPDATE, наподобии DELETE, может брать предикаты. Вот как например можно выполнить изменение одинаковое для всех заказчиков продавца Peel (имеющего snum=1001):

```
UPDATE Customers
SET rating = 200
WHERE snum = 1001;
```

КОМАНДА UPDATE ДЛЯ МНОГИХ СТОЛБЦОВ

Однако, вы не должны ограничивать себя модифицированием единственного столбца с помощью команды UPDATE. Предложение SET может назначать любое число столбцов, отделяемых запятыми. Все указанные назначения могут быть сделаны для любой табличной строки, но только для одной в каждый момент времени. Предположим, что продавец Motika ушел на пенсию, и мы хотим переназначить его номер новому продавцу:

```
UPDATE Salespeople
SET sname = 'Gibson', city = 'Boston', comm = .10
WHERE snum = 1004;
```

Эта команда передаст новому продавцу Gibson, всех текущих заказчиков бывшего продавца Motika и порядки, в том виде в котором они были скомпонованы для Motika с помощью поля snum.

Вы не можете, однако, модифицировать сразу много таблиц в одной команде, частично потому, что вы не можете использовать префиксы таблицы со столбцами измененными предложением SET. Другими словами, вы не можете сказать — "SET Salespeople.sname = Gibson" в команде UPDATE, вы можете сказать только так — "SET sname = Gibson".

ИСПОЛЬЗОВАНИЕ ВЫРАЖЕНИЙ ДЛЯ МОДИФИКАЦИИ

Вы можете использовать скалярные выражения в предложении SET команды UPDATE, однако, включив его в выражение поля которое будет изменено. В этом их отличие от предложения VALUES команды INSERT, в котором выражения не могут

использоваться; это свойство скалярных выражений — весьма полезная особенность. Предположим, что вы решили удвоить комиссионные всем вашим продавцам. Вы можете использовать следующее выражение:

```
UPDATE Salespeople
SET comm = comm * 2;
```

Всякий раз, когда вы ссылаетесь к указанному значению столбца в предложении SET, произведенное значение может получится из текущей строки, прежде в ней будут сделаны какие-то изменения с помощью команды UPDATE. Естественно, вы можете скомбинировать эти особенности, и сказать, — удвоить комиссию всем продавцам в Лондоне, таким предложением:

```
UPDATE Salespeople
SET comm = comm * 2
WHERE city = 'London';
```

МОДИФИЦИРОВАНИЕ ПУСТЫХ(NULL) ЗНАЧЕНИЙ

Предложение SET — это не предикат. Он может вводить *пустые* NULL значения также, как он вводил значения, не используя какого-то специального синтаксиса (такого, например, как IS NULL). Так что, если вы хотите установить все оценки заказчиков в Лондоне в NULL, вы можете ввести следующее предложение:

```
UPDATE customers
SET rating = NULL
WHERE city = 'London';
```

что обнулит все оценки заказчиков в Лондоне.

РЕЗЮМЕ

Теперь вы овладели мастерством управления содержанием вашей базы данных с помощью трех простых команд:

```
INSERT     — используемой чтобы помещать строки в базу данных;
DELETE     — чтобы удалять их;
REFERENCES — чтобы изменять значения в уже вставленных строках.
```

Вы обучались использованию предиката с командами UPDATE и DELETE чтобы определять, на которую из строк будет воздействовать команда. Конечно, предикаты как таковые — не значимы для INSERT, потому что обсуждаемая строка не существует в таблице до окончания выполнения команды INSERT. Однако, вы можете использовать запросы с INSERT, чтобы сразу помещать все наборы строк в таблицу. Причем это, вы можете делать со столбцами в любом порядке.

Вы узнали, что значения по умолчанию, могут помещаться в столбцы, если вы не устанавливаете это значение явно. Вы также видели использование стандартного значения по умолчанию, которым является NULL. Кроме того, вы поняли, что UPDATE может использовать выражение значения, тогда как INSERT не может.

Следующая глава расширит ваше познания, показав вам, как использовать подзапросы с этими командами. Эти подзапросы напоминают те, с которыми вы уже знакомы, но имеются некоторые специальные выводы и ограничения, когда подзапросы используются в командах DML, что мы будем обсуждать в Главе 16.

РАБОТА С SQL

1. Напишите команду, которая бы поместила следующие значения, в их нижеуказанном порядке, в таблицу Продавцов:

```
city – San Jose,
   name – Bianco,
   comm – NULL,
   cnum – 1100.
```

2. Напишите команду, которая бы удалила все порядки заказчика Clemens из таблицы Порядков.
3. Напишите команду, которая бы увеличила оценку всех заказчиков в Риме на 100.
4. Продавец Serres оставил компанию. Переназначьте его заказчиков продавцу Motika.

(См. Приложение А для ответов.)

16

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ С КОМАНДАМИ МОДИФИКАЦИИ

В ЭТОЙ ГЛАВЕ, ВЫ УЗНАЕТЕ КАК ИСПОЛЬЗОВАТЬ подзапросы в командах модификации.

Вы найдете, что нечто подобное вы уже видели при использовании подзапросов в запросах. Понимание, как подзапросы используются в командах SELECT, сделает их применение в командах модификации более уверенным, хотя и останутся некоторые вопросы. Завершением команды SELECT является подзапрос, но не предикат, и поэтому его использование отличается от использования простых предикатов с командами модификации, которые вы уже выполняли ранеее с командами UPDATE и DELETE. Вы использовали простые запросы чтобы производить значения для INSERT, а теперь мы можем расширить эти запросы чтобы включять в них подзапросы.

Важный принцип, который надо соблюдать при работе с командами модификации, состоит в том, что вы не можете в предложении FROM любого подзапроса модифицировать таблицу, к которой ссылаетесь с помощью основной команды. Это относится ко всем трем командам модификации. Хотя имеется большое количество ситуаций, в которых будет полезно сделать запрос той таблицы, которую вы хотите модифицировать, причем во время ее модификации, это слишком усложняет операцию, чтобы использовать ее на практике.

Не делайте ссылки к текущей строке таблицы, указанной в команде, которая является соотнесенным подзапросом.

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ С INSERT

INSERT — это самый простой случай. Вы уже видели как вставлять результаты запроса в таблицу. Вы можете использовать подзапросы внутри любого запроса, который генерирует значения для команды INSERT тем же самым способом, которым вы делали это для других запросов — т.е. внутри предиката или предложения HAVING.

Предположим, что мы имеем таблицу с именем SJpeople, столбцы которой совпадают со столбцами нашей таблицы Продавцов. Вы уже видели как заполнять таблицу подобно этой, заказчиками в городе, например, в San Jose:

```
INSERT INTO Sjpeople
SELECT *
FROM Salespeople
WHERE city = 'San Jose';
```

Теперь мы можем использовать подзапрос чтобы добавить к таблице SJpeople всех продавцов которые имеют заказчиков в San Jose, независимо от того, находятся ли там продавцы или нет:

```
INSERT INTO Sjpeople
SELECT *
FROM Salespeople
WHERE snum = ANY ( SELECT snum
                   FROM Customers
                   WHERE city = 'San Jose' );
```

Оба запроса в этой команде функционируют также как если бы они не являлись частью выражения INSERT. Подзапрос находит все строки для заказчиков в San Jose и формирует набор значений snum. Внешний запрос выбирает строки из таблицы Salespeople, где эти значения snum найдены. В этом примере, строки для продавцов Rifkin и Serres, которые назначены заказчикам в San Jose — Liu и Cisneros, будут вставлены в таблицу SJpeople.

НЕ ВСТАВЛЯЙТЕ ДУБЛИКАТЫ СТРОК

Последовательность команд в предшествующем разделе может быть проблематичной. Продавец Serres находится в San Jose, и следовательно будет вставлен с помощью первой команды. Вторая команда попытается вставить его снова, поскольку он имеет еще одного заказчика в San Jose. Если имеются любые ограничения в таблице SJpeople, которые вынуждают ее значения быть уникальными, эта вторая вставка потерпит неудачу (как это и должно было быть). Двойные строки это плохо. (См. Главу 18 для подробностей об ограничениях.)

Было бы лучше если бы вы могли как-то выяснить, что эти значения уже были вставлены в таблицу, прежде чем вы попытаетесь сделать это снова, с помощью добавления другого подзапроса (использующего операторы типа EXISTS, IN, <> ALL, и так далее) к предикату.

К сожалению, чтобы сделать эту работу, вы должны будете сослаться на саму таблицу SJpeople в предложении FROM этого нового подзапроса, а, как мы говорили ранее, вы не можете ссылаться на таблицу которая задействована (целиком) в любом подзапросе команды модификации.

В случае INSERT, это будет также препятствовать соотнесенным подзапросам, основанным на таблице в которую вы вставляете значения. Это имеет значение, потому что, с помощью INSERT, вы создаете новую строку в таблице. "Текущая строка" не будет существовать до тех пор, пока INSERT не закончит ее обрабатывать.

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ, СОЗДАНЫХ ВО ВНЕШНЕЙ ТАБЛИЦЕ ЗАПРОСА

Запрещение на ссылку к таблице, которая модифицируется командой INSERT, не предохранит вас от использования подзапросов, которые ссылаются к таблице, используемой в предложении FROM внешней команды SELECT. Таблица, из которой вы выбираете значения, чтобы произвести их для INSERT, не будет задействована командой; и вы сможете ссылаться к этой таблице любым способом которыми вы обычно это делали, но только если эта таблица указана в автономном запросе. Предположим что мы имеем таблицу с именем Samecity в которой мы запомним продавцов с заказчиками в их городах.

Мы можем заполнить таблицу используя соотнесенный подзапрос:

```
INSERT INTO (Samecity
SELECT *
FROM (Salespeople outer
WHERE city IN ( SELECT city
                FROM Customers inner
                WHERE inner.snum = outer.snum );
```

Ни таблица Samecity, ни таблица Продавцов не должны быть использованы во внешних или внутренних запросах INSERT.

В качестве другого примера, предположим, что вы имеете премию для продавца который имеет самый большой порядок на каждый день. Вы следите за ним в таблице с именем Bonus, которая содержит поле snum продавцов, поле odate и поле amt. Вы должны заполнить эту таблицу информацией которая хранится в таблице Порядков, используя следующую команду:

```
INSERT INTO Bonus
SELECT snum, odate, amt
FROM Orders a
WHERE amt = ( SELECT MAX (amt)
              FROM Orders b
              WHERE a.odate = b.odate );
```

Даже если эта команда имеет подзапрос, который базируется на той же самой таблице, что и внешний запрос, он не ссылается к таблице Bonus, на которую воздействует команда. Что для нас абсолютно приемлемо.

Логика запроса, естественно, должна просматривать таблицу Порядков, и находить для каждой строки максимум порядка сумм приобретений для этой даты. Если эта величина — такая же как у текущей строки, текущая строка является наибольшим порядком для этой даты, и данные вставляются в таблицу Bonus.

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ С DELETE

Вы можете также использовать подзапросы в предикате команды DELETE. Это даст вам возможность определять некоторые довольно сложные критерии чтобы установить, какие строки будут удаляться, что важно, так как вы конечно же не захотите по неосторожности удалить нужную строку.

Например, если мы закрыли наше ведомство в Лондоне, мы могли бы использовать следующий запрос, чтобы удалить всех заказчиков, назначенных к продавцам в Лондоне:

```
DELETE
FROM Customers
WHERE snum = ANY ( SELECT snum
                   FROM Salespeople
                   WHERE city = 'London' );
```

Эта команда удалит из таблицы Заказчиков строки Hoffman и Clemens (назначенных для Peel), и Periera (назначенного к Motika).

Конечно, вы захотите удостовериться, правильно ли сформирована эта операция, прежде чем удалит или изменит строки Peel и Motika.

Это важно. Обычно, когда мы делаем модификацию в базе данных, которая повлечет другие модификации, наше первое желание — сделать сначало основное действие, а затем проследить другие, вторичные. Этот пример, покажет, почему более эффективно делать наоборот, выполнив сначала вторичные действия.

Если, например, вы решили изменить значение поля city ваших продавцов везде, где они переназначены, вы должны рассмотреть всех этих заказчиков более сложным способом.

Так как реальные базы данных имеют тенденцию развиваться до значительно больших размеров, чем наши небольшие типовые таблицы, это может стать серьезной проблемой. SQL может предоставить некоторую помощь в этой области, используя механизм справочной целостности (обсужденной в Главе 19), но это не всегда доступно и не всегда применимо.

Хотя вы не можете ссылаться к таблице из которой вы будете удалять строки в предложении FROM подзапроса, вы можете в предикате, сослаться на текущую строку-кандидат этой таблицы — которая является строкой, которая в настоящее время проверяется в основном предикате. Другими словами, вы можете использовать соотнесенные подзапросы. Они отличаются от тех соотнесенных подзапросов, которые вы могли использовать с INSERT, в котором они фактически базировались на строках-кандидатах таблицы, задействованой в команде, а не на запросе другой таблицы.

```
DELETE FROM Salespeople
WHERE EXISTS ( SELECT *
               FROM Customers
               WHERE rating = 100 AND
                     Salespeople.snum = Customers.snum );
```

Обратите внимание, что AND часть предиката внутреннего запроса ссылается к таблице Продавцов. Это означает что весь подзапрос будет выполняться отдельно для каждой строки таблицы Продавцов, также как это выполнялось с другими соотнесенными подзапросами. Эта команда удалит всех продавцов которые имели по меньшей мере одного заказчика с оценкой 100 в таблице Продавцов.

Конечно же, имеется другой способ сделать то же:

```
DELETE FROM Salespeople
WHERE 100 IN ( SELECT rating
               FROM Customers
               WHERE Salespeople.snum = Customers.snum);
```

Эта команда находит все оценки для каждого заказчика продавцов и удаляет тех продавцов, заказчики которых имеют оценку = 100.

Обычно *соотнесенные подзапросы* — это подзапросы, связанные с таблицей, к которой они ссылаются во внешнем запросе (а не в самом предложении DELETE) — и также часто используемы. Вы можете найти наинизший порядок на каждый день и удалить продавцов, которые произвели его, с помощью следующей команды:

```
DELETE FROM Salespeople
WHERE (snum IN ( SELECT snum
                 FROM Orders
                 WHERE amt = ( SELECT MIN (amt)
                               FROM Orders b
                               WHERE a.odate = b.odate ));
```

Подзапрос в предикате DELETE, берет соотнесенный подзапрос. Этот внутренний запрос находит минимальный порядок суммы приобретеий для даты каждой строки внешнего запроса. Если эта сумма такая же, как сумма текущей строки, предикат внешнего запроса верен, что означает, что текущая строка имеет наименьший порядок для этой даты. Поле snum продавца, ответственного за этот порядок, извлекается и передается в основной предикат команды DELETE, которая затем удаляет все строки с этим значением поля snum из таблицы Продавцов (так как snum — это первичный ключ таблицы Продавцов, то естественно там должна иметься только одна удаляемая строка для значения поля snum выведенного с помощью подзапроса. Если имеется больше одной строки, все они будут удалены.)

Поле snum = 1007, которое будет удалено, имеет наименьшее значение на 3 Октября; поле snum = 1002, наименьшее на 4 Октября; поле snum = 1001, наименьшее в порядках на 5 Октября (эта команда кажется довольно резкой, особенно когда она удаляет Peel, создавшего единственный порядок на 5 Октября, но зато это хорошая иллюстрация).

Если вы хотите сохранить Peel, вы могли бы добавить другой подзапрос, который бы это делал:

```
DELETE FROM Salespeople
WHERE snum IN ( SELECT snum
                FROM Orders a
                WHERE amt = ( SELECT MIN (amt)
                              FROM Orders b
                              WHERE a.odate = b.odate )
                      AND 1 < ( SELECT COUNT onum
                                FROM Orders b
                                WHERE a.odate = b.odate ));
```

Теперь для дня, в котором был создан только один порядок, будет произведен COUNT = 1 во втором соотнесенном подзапросе. Это сделает предикат внешнего запроса неправильным, и поля snum следовательно не будут переданы в основной предикат.

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ С UPDATE

UPDATE использует подзапросы тем же самым способом что и DELETE — внутри этого необязательного предиката. Вы можете использовать соотнесенные подзапросы или в форме пригодной для использования с DELETE — связаной или с модифицируемой таблицей или с таблицей вызываемой во внешнем запросе. Например, с помощью соотнесенного подзапроса к таблице которая будет модифицироваться, вы можете увеличить комиссионные всех продавцов которые были назначены по крайней мере двум заказчикам:

```
UPDATE Salespeople
SET comm = comm + .01
WHERE 2 <= ( SELECT COUNT (cnum)
             FROM Customers
             WHERE Customers.snum = Salespeople.snum );
```

Теперь продавцы Peel и Serres, имеющие многочисленых заказчиков, получат повышение своих комиссионных.

Имеется разновидность последнего примера из предыдущего раздела с DELETE. Он уменьшает комиссионные продавцов, которые произвели наименьшие порядки, но не стирает их в таблице:

```
UPDATE Salespeople
SET comm = comm - .01
WHERE snum IN ( SELECT snum
                FROM Orders a
                WHERE amt = ( SELECT MIN (amt)
                              FROM Orders b
                              WHERE a.odate = b.odate ));
```

СТОЛКНОВЕНИЕ С ОГРАНИЧЕНИЯМИ ПОДЗАПРОСОВ КОМАНДЫ DML

Неспособность сослаться к таблице, задействованой в любом подзапросе из команды модификации (UPDATE), устраняет целые категории возможных действий.

Например, вы не можете просто выполнить такую операцию, как удаление всех заказчиков с оценками ниже средней. Вероятно лучше всего вы могли бы сначала (Шаг 1.), выполнить запрос, получающий среднюю величину, а затем (Шаг 2.), удалить все строки с оценкой ниже этой величины:

Шаг 1.

```
SELECT AVG (rating)
FROM Customers;
```

Вывод = 200.

Шаг 2.

```
DELETE
FROM Customers
WHERE rating < 200;
```

РЕЗЮМЕ

Теперь вы овладели тремя командами, которые управляют всем содержанием вашей базы данных. Осталось только несколько общих вопросов относительно ввода и стирания значений таблицы, когда, например, эти команды могут выполниться данным пользователем в данной таблице и когда действия сделанные ими, становятся постоянными.

Подведем итог: Вы используете команду INSERT чтобы добавлять строки в таблицу. Вы можете или дать имена значениям этих строк в предложении VALUES (когда только одна строка может быть добавлена), или вывести значения с помощью запроса (когда любое число строк можно добавить одной командой). Если используется запрос, он не может ссылаться к таблице, в которую вы делаете вставку, каким бы способом Вы ее ни делали, ни в предложении FROM, ни с помощью внешней ссылки (как это делается в соотнесенных подзапросах). Все это относится к любым подзапросам внутри этого запроса.

Запрос, однако, оставляет вам свободу использования соотнесенных подзапросов или подзапросов, которые дают в предложении FROM имя таблице, которое уже было указано в предложении FROM внешнего запроса (это — общий случай для запросов).

DELETE и UPDATE используются чтобы, соответственно, удалить строки из таблицы и изменить в них значения. Оба они применимы ко всем строкам таблицы, если не используется предикат, определяющий, какие строки должны быть удалены или модифицированы. Этот предикат может содержать подзапросы, которые могут быть связаны с таблицей, удаляемой, или модифицированой, с помощью внешней ссылки. Эти подзапросы, однако, не могут ссылать к таблице модифицируемой любым предложением FROM.

Может показаться, что мы прошли материал SQL, который обладает не самым понятным логическим порядком. Сначала мы сделали запрос таблицы, которая уже заполнена данными. Потом мы показали как можно фактически помещать эти значения изначально. Но, как вы видете, полное ознакомление с запросами здесь неоценимо.

Теперь, когда мы показали вам как заполнять значениями таблицы, которые уже были созданы (по определению), мы покажем (со следующей главы), откуда появились эти таблицы.

РАБОТА С SQL

1. Предположите, что имеется таблица, называемая Multicust, с такими же именами столбцов, что и таблица Продавцов. Напишите команду, которая бы вставила всех продавцов (из таблицы Продавцов) имеющих более чем одного заказчика в эту таблицу.
2. Напишите команду, которая бы удаляла всех заказчиков, не имеющих текущих порядков.
3. Напишите команду которая бы увеличила на двадцать процентов комиссионные всех продавцов, имеющих общие текущие порядки выше чем $3,000.

(См. Приложение А для ответов.)

17

СОЗДАНИЕ ТАБЛИЦ

ВПЛОТЬ ДО ЭТОГО МЕСТА, МЫ ЗАПРАШИВАЛИ ТАБЛИЦЫ данных и выполняли команды по извлечению этих данных, считая, что эти таблицы уже были созданы кем-то до нас. Это действительно наиболее реальная ситуация, когда небольшое количество людей создают таблицы, которые затем используются другими людьми. Наша цель состоит в том, чтобы охватив информацию сначала более широко, перейти затем к более узким вопросам.

В этой главе, мы будем обсуждать *создание*, *изменение* и *удаление* таблиц. Все это относится к самим таблицам, а не к данным, которые в них содержатся. Будете или не будете Вы выполнять эти операции самостоятельно, но их концептуальное понимание увеличит ваше понимание языка SQL и природу таблиц, которые вы используете. Эта глава вводит нас в область SQL называемую — **DDL** (*Язык Определения Данных*), где создаются объекты данных SQL.

Эта глава также покажет другой вид объекта данных SQL — **Индекс**. Индексы используются, чтобы делать поиск более эффективным и, иногда, заставлять значения отличаться друга от друга.

Они обычно работают незаметно для Вас, но если вы попробуете поместить значения в таблицу и они будут отклонены из-за их неуникальности, это будет означать что другая строка имеет то же самое значение для этого поля, и что это поле имеет *уникальный индекс* или *ограничение*, которое предписывает ему уникальность.

Обсуждение вышеупомянутого, продолжится в Главе 18.

КОМАНДА СОЗДАНИЯ ТАБЛИЦЫ

Таблицы создаются командой **CREATE TABLE**. Эта команда создает пустую таблицу — таблицу без строк. Значения вводятся с помощью DML команды **INSERT** (См. Главу 15). Команда CREATE TABLE в основном определяет имя таблицы, в виде описания набора имен столбцов указанных в определенном порядке. Она также определяет типы данных и размеры столбцов. Каждая таблица должна иметь по крайней мере один столбец.

Синтаксис команды CREATE TABLE:

```
CREATE TABLE <table-name >
( <column name > <data type>[(<size>)],
<column name > <data type> [(<size>)] ... );
```

Как сказано в Главе 2, типы данных значительно меняются от программы к программе. Для совместимости со стандартом, они должны все, по крайней мере, поддерживать стандарт типа ANSI. Он описан в Приложении B.

Так как пробелы используются для разделения частей команды SQL, они не могут быть частью имени таблицы (или любого другого объекта, такого как индекс). Подчеркивание (_) обычно используется для разделения слов в именах таблиц.

Значение аргумента размера зависит от типа данных. Если вы его не указываете, ваша система сама будет назначать значение автоматически. Для числовых значений, это — лучший выход, потому что в этом случае, все ваши поля такого типа получат один и тот же размер, что освобождает вас от проблем их общей совместимости (см. Главу 14).

Кроме того, использование аргумента размера с некоторыми числовым наборами, не совсем простой вопрос. Если вам нужно хранить большие числа, вам несомненно понадобятся гарантии, что поля достаточно велики чтобы вместить их.

Один тип данных для которого вы, в основном, должны назначать размер — **CHAR**. *Аргумент размера* — это целое число которое определяет максимальное число символов которое может вместить поле. Фактически, число символов поля может быть от нуля (если поле — NULL) до этого числа. По умолчанию, аргумент разме-

ра = 1, что означает, что поле может содержать только одну букву. Это конечно не совсем то, что вы хотите.

Таблицы принадлежат пользователю, который их создал, и имена всех таблиц, принадлежащих данному пользователю должны отличаться друга от друга, как и имена всех столбцов внутри данной таблицы. Отдельные таблицы могут использовать одинаковые имена столбцов, даже если они принадлежат одному и тому же пользователю. Примером этому — столбец city в таблице Заказчиков и в таблице Продавцов. Пользователи не являющиеся владельцами таблиц могут ссылаться к этим таблицам с помощью имени владельца этих таблиц, сопровождаемого точкой; например, таблица Employees созданая Smith будет называться Smith.Employees когда она упоминается каким-то другим пользователем. Мы понимаем что Smith — это *Идентификатор Разрешения* **(ID)**, сообщаемый пользователем (ваш разрешенный ID — это ваше имя в SQL). Этот вывод обсуждался в Главе 2, и будет продолжен в Главе 22.

Эта команда будет создавать таблицу Продавцов:

```
CREATE TABLE Saleepeople
( snum  integer,
  sname char (10),
  city  char (10),
  comm  declmal );
```

Порядок столбцов в таблице определяется порядком в котором они указаны. Имя столбца не должно разделяться при переносе строки (что сделано для удобочитаемости), но отделяется запятыми.

ИНДЕКСЫ

Индекс — это упорядоченный (буквенный или числовой) список столбцов или групп столбцов в таблице. Таблицы могут иметь большое колличество строк, а, так как строки не находятся в каком-нибудь определенном порядке, на их поиск по указанному значению может потребовать время.

Индексный адрес — это и забота, и в то же время обеспечение способа объединения всех значений в группы из одной или больше строк, которые отличаются одна от другой. В Главе 18, мы будем описывать более непосредственный способ, который заставит ваши значения быть уникальными. Но этот метод не существует в ранних версиях SQL. Так как уникальность часто необходима, индексы и использовались с этой целью.

Индексы — это средство SQL, которое родил сам рынок, а не ANSI. Поэтому, сам по себе стандарт ANSI в настоящее время не поддерживает индексы, хотя они очень полезны и широко применяемы.

Когда вы создаете индекс в поле, ваша база данных запоминает соответствующий порядок всех значений этого поля в области памяти. Предположим что наша таблица Заказчиков имеет тысячи входов, а вы хотите найти заказчика с номером=2999. Так как строки не упорядочены, ваша программа будет просматривать всю таблицу, строку за строкой, проверяя каждый раз значение поля cnum на равенство значению 2999. Однако, если бы имелся индекс в поле cnum, то программа могла бы выйти на номер 2999 прямо по индексу и дать информацию о том как найти правильную строку таблицы.

В то время как индекс значительно улучшает эффективность запросов, использование индекса несколько замедляет операции модификации DML (такие как **INSERT** и **DELETE**), а сам индекс занимает объем памяти. Следовательно, каждый раз, когда вы создаете таблицу, Вы должны принять решение, индексировать ее или нет.

Индексы могут состоять из многочисленых полей. Если больше чем одно поле указывается для одного индекса, второе упорядочивается внутри первого, третье внутри второго, и так далее. Если вы имели первое и последнее имя в двух различных полях таблицы, вы могли бы создать индекс который бы упорядочил предыдущее поле внутри последующего. Это может быть выполнено независимо от способа упорядочивания столбцов в таблице.

Синтаксис для создания индекса — обычно следующий (помните, что это не ANSI стандарт):

```
CREATE INDEX <index name> ON <table name>
(<column name> [,<column name>]...);
```

Таблица, конечно, должна уже быть создана и должна содержать имя столбца. Имя индекса не может быть использовано для чего-то другого в базе данных (любым пользователем). Однажды созданый, индекс будет невидим пользователю. SQL сам решает когда он необходим чтобы ссылаться на него и делает это автоматически.

Если, например, таблица Заказчиков будет наиболее часто упоминаемой в запросах продавцов к их собственной клиентуре, было бы правильно создать такой индекс в поле snum таблицы Заказчиков.

```
CREATE INDEX Clientgroup ON Customers (snum);
```

Теперь, тот продавец, который имеет отношение к этой таблице, сможет найти собственную клиентуру очень быстро.

УНИКАЛЬНОСТЬ ИНДЕКСА

Индексу в предыдущем примере, к счастью, не предписывается уникальность, несмотря на наше замечание, что это является одним из назначений индекса. Данный продавец может иметь любое число заказчиков. Однако, этого не случится, если мы используем ключевое слово **UNIQUE** перед ключевым словом **INDEX**. Поле cnum, в качестве первичного ключа, станет первым кандидатом для уникального индекса:

```
CREATE UNIQUE INDEX Custid ON Customers (cnum);
```

ПРИМЕЧАНИЕ: эта команда будет отклонена, если уже имеются идентичные значения в поле cnum. Лучший способ иметь дело с индексами состоит в том, чтобы создавать их сразу после того, как таблица создана и прежде, чем введены любые значения. Так же обратите внимание что, для уникального индекса более чем одного поля, это — комбинация значений, каждое из которых, может и не быть уникальным.

Предыдущий пример — косвенный способ заставить поле cnum работать как первичный ключ таблицы Заказчиков. Базы данных воздействуют на первичные и другие ключи более непосредственно. Мы будем обсуждать этот вывод далее в Главах 18 и 19.

УДАЛЕНИЕ ИНДЕКСОВ

Главным признаком индекса является его имя — поэтому он может быть удален. Обычно пользователи не знают о существовании индекса. SQL автоматически определяет позволено ли пользователю использовать индекс, и если да, то разрешает использовать его. Однако, если вы хотите удалить индекс, вы должны знать его имя. Этот синтаксис используется для удаления индекса:

```
DROP INDEX <Index name>;
```

Удаление индекса не воздействует на содержание полей.

ИЗМЕНЕНИЕ ТАБЛИЦЫ ПОСЛЕ ТОГО, КАК ОНА БЫЛА СОЗДАНА

Команда **ALTER TABLE** не часть стандарта ANSI; но это — широко доступная, и довольно содержательная форма, хотя ее возможности несколько ограничены. Она используется, чтобы изменить определение существующей таблицы. Обычно, она добавляет столбцы к таблице. Иногда она может удалять столбцы или изменять их размеры, а также в некоторых программах добавлять или удалять ограничения (обсужденные в Главе 18). Типичный синтаксис чтобы добавить столбец к таблице:

```
ALTER TABLE <table name> ADD <column name> <data type> <size>;
```

Столбец будет добавлен со значением NULL для всех строк таблицы. Новый столбец станет последним по порядку столбцом таблицы. Вообще то, можно добавить сразу несколько новых столбцов, отделив их запятыми, в однй команде. Имеется возможность удалять или изменять столбцы. Наиболее часто, изменением столбца может быть просто увеличение его размера, или добавление (удаление) ограничения.

Ваша система должна убедиться, что любые изменения не противоречат существующим данным — например при попытке добавить ограничение к столбцу который уже имел значение при нарушении которого ограничение будет отклонено. Лучше всего дважды проверить это.

По крайней мере, посмотрите документацию вашей системы чтобы убедиться, гарантирует ли она что именно это было причиной. Из-за нестандартного характера команды ALTER TABLE, вам все равно необходимо посмотреть тот раздел вашей системной документации где говорится об особых случаях.

ALTER TABLE — не действует, когда таблица должна быть переопределена, но вы должны разрабатывать вашу базу данных по возможности так чтобы не слишком ей в этом передоверяться. Изменение структуры таблицы, когда она уже в использовании — опасно! Просмотрите внимательно таблицы, которые являясь вторичными таблицами с извлечеными данными из другой таблицы (смотри Главу 20), не долго правильно работают, а программы использующие вложенный SQL (Глава 25) выполняются неправильно или не всегда правильно. Кроме того, изменение может стереть всех пользователей, имеющих разрешение обращаться к таблице.

По этим причинам, вы должны разрабатывать ваши таблицы так, чтобы использовать ALTER TABLE только в крайнем случае.

Если ваша система не поддерживает ALTER TABLE, или если вы хотите избежать ее использования, вы можете просто создать новую таблицу, с необходимыми изменениями при создании, и использовать команду **INSERT** с **SELECT *** запросом чтобы переписать в нее данные из старой таблицы.

Пользователям которым был предоставлен доступ к старой таблице (см. Главу 22) должен быть предоставлен доступ к новой таблице.

УДАЛЕНИЕ ТАБЛИЦ

Вы должны быть собственником (т.е. быть создателем) таблицы, чтобы иметь возможность удалить ее. Поэтому не волнуйтесь о случайном разрушении ваших данных, SQL сначала потребует чтобы вы очистили таблицу прежде, чем удалит ее из базы данных. Таблица с находящимися в ней строками, не может быть удалена. Об-

ратитесь к Главе 15 за подробностями относительно того, как удалять строки из таблицы. Синтаксис для удаления вашей таблицы, если конечно она является пустой, следующая:

```
DROP TABLE <table name>;
```

При подаче этой команды, имя таблицы больше не распознается и нет такой команды, которая могла быть дана этому объекту. Вы должны убедиться, что эта таблица не ссылается внешним ключом к другой таблице (Внешние ключи обсуждаются в Главе 19), и что она не используется в определении Представления (Глава 20).

Эта команда фактически не является частью стандарта ANSI, но она обще поддерживаема и полезна. К счастью, она более проста, и следовательно более непротиворечива, чем ALTER TABLE. ANSI просто не имеет способа для определения разрушеных или неправильных таблиц.

РЕЗЮМЕ

Теперь Вы уже бегло ориентируетесь в основах определений данных. Вы можете создавать, изменять, и удалять таблицы. В то время как только первая из этих функций — часть официального стандарта SQL, другие будут время от времени меняться, особенно — ALTER TABLE. DROP TABLE позволяет вам избавиться от таблиц которые бесполезны. Она уничтожает только пустые таблицы, и следовательно не разрушает данные.

Вы теперь знаете об индексах а также, как их создавать и удалять. SQL не дает вам большого управления над ими, так как реализация которую вы используете довольно удачно определяет, как быстро выполняются различные команды. Индексы — это один из инструментов дающий Вам возможность воздействовать непосредственно на эффективность ваших команд в SQL. Мы рассмотрели индексы здесь чтобы отличать их от ограничений, с которыми их нельзя путать. Ограничения — это тема Главы 18 и Главы 19.

РАБОТА С SQL

1. Напишите предложение CREATE TABLE, которое бы вывело нашу таблицу Заказчиков.
2. Напишите команду, которая бы давала возможность пользователю быстро извлекать порядки сгруппированные по датам из таблицы Порядков.
3. Если таблица Порядков уже создана, как Вы можете заставить поле onum быть уникальным (если допустить что все текущие значения уникальны) ?
4. Создайте индекс который бы разрешал каждому продавцу быстро отыскивать его порядки сгруппированные по датам.
5. Предположим, что каждый продавец имеет только одного заказчика с данной оценкой, введите команду которая его извлечет.

(См. Приложение A для ответов.)

18

ОГРАНИЧЕНИЕ ЗНАЧЕНИЙ ВАШИХ ДАННЫХ

В ГЛАВЕ 17, ВЫ УЗНАЛИ КАК СОЗДАЮТСЯ ТАБЛИЦЫ. Теперь более тщательно с этого места мы покажем вам, как вы можете устанавливать ограничения в таблицах.

Ограничения — это часть определения таблицы, которое ограничивает значения, которые вы можете вводить в столбцы.

До этого места в книге, единственым ограничением на значения, которые вы могли вводить, были тип данных и размер вводимых значений, которые должны быть совместимы с теми столбцами в которые эти значения помещаются (как и определено в команде CREATE TABLE или команде ALTER TABLE). Ограничения дают вам значительно большие возможности и скоро вы это увидете. Вы также узнаете как определять значения по умолчанию в этой главе.

По умолчанию — это значение которое вставляется автоматически в любой столбец таблицы, когда значение для этого столбца отсутствует в команде INSERT для этой таблицы. NULL — это наиболее широко используемое значение *по умолчанию*, но в этой главе будет показано как определять и другие значения *по умолчанию*.

ОГРАНИЧЕНИЕ ТАБЛИЦ

Когда вы создаете таблицу (или, когда вы ее изменяете), вы можете помещать ограничение на значения, которые могут быть введены в поля. Если вы это сделали, SQL будет отклонять любые значения, которые нарушают критерии, которые вы определили. Имеется два основных типа ограничений — *ограничение столбца* и *ограничение таблицы*. Различие между ними в том, что ограничение столбца применяется только к индивидуальным столбцам, в то время как ограничение таблицы применяется к группам из одного и более столбцов.

ОБЪЯВЛЕНИЕ ОГРАНИЧЕНИЙ

Вы вставляете ограничение столбца в конец имени столбца после типа данных и перед запятой. Ограничение таблицы помещаются в конец имени таблицы после последнего имени столбца, но перед заключительной круглой скобкой. Далее показан синтаксис для команды CREATE TABLE, расширенной для включения в нее ограничения:

```
CREATE TABLE <table name>
(<column name> <data type> <column constraint>,
 <column name> <data type> <column constraint> ...
 <table constraint> ( <column name> [, <column name> ])... );
```

(Для краткости, мы опустили аргумент размера, который иногда используется с типом данных.) Поля данные в круглых скобках после ограничения таблицы — это поля к которым применено это ограничение. Ограничение столбца, естественно, применяется к столбцам, после чьих имен оно следует. Остальная часть этой глава будет описывать различные типы ограничений и их использование.

ИСПОЛЬЗОВАНИЕ ОГРАНИЧЕНИЙ ДЛЯ ИСКЛЮЧЕНИЯ ПУСТЫХ (NULL) УКАЗАТЕЛЕЙ

Вы можете использовать команду CREATE TABLE чтобы предохранить поле от разрешения в нем *пустых* (**NULL**) указателей с помощью ограничения **NOT NULL**. Это ограничение накладывается только для разнообразных столбцов.

Вы можете вспомнить что **NULL** — это специальное обозначение, которое отмечает поле как пустое. NULL может быть полезен, когда имеются случаи, когда вы хотите быть от них гарантированы. Очевидно, что первичные ключи никогда не должны быть *пустыми*, поскольку это будет подрывать их функциональные возможности. Кроме того, такие поля как имена, требуют в большинстве случаев, определенных значений. Например, вы вероятно захотите иметь имя для каждого заказчика в таблице Заказчиков. Если вы поместите ключевые слова NOT NULL сразу после типа данных (включая размер) столбца, любая попытка поместить значение NULL в это поле будет отклонена. В противном случае, SQL понимает, что NULL разрешен.

Например, давайте улучшим наше определение таблицы Продавцов, не позволяя помещать NULL значения в столбцы snum или sname:

```
CREATE TABLE Salespeople
( Snum  integer NOT NULL,
  Sname char (10) NOT NULL,
  city  char (10),
  comm  decimal);
```

Важно помнить, что любому столбцу с ограничением NOT NULL должно быть установлено значение в каждом предложении INSERT, воздействующем на таблицу. При отсутствии NULL, SQL может не иметь значений для установки в эти столбцы, если конечно значение *по умолчанию*, описанное ранее в этой главе, уже не было назначено.

Если ваша система поддерживает использование ALTER TABLE, чтобы добавлять новые столбцы к уже существующей таблице, вы можете вероятно помещать ограничение столбцов, типа NOT NULL, для этих новых столбцов. Однако, если вы предписываете новому столбцу значение NOT NULL, текущая таблица должна быть пустой.

УБЕДИТЕСЬ, ЧТО ЗНАЧЕНИЯ УНИКАЛЬНЫ

В Главе 17, мы обсудили использование уникальных индексов чтобы заставить поля иметь различные значения для каждой строки. Эта практика — осталась с прежних времен, когда SQL поддерживал ограничение UNIQUE. *Уникальность* — это свойство данных в таблице, и поэтому его более логично назвать как *ограничение этих данных*, а не просто как *свойство логического отличия*, связывающее объект данных (индекс).

Несомненно, уникальные индексы — один из самых простых и наиболее эффективных методов предписания уникальности. По этой причине, некоторые реализации ограничения UNIQUE используют уникальные индексы; то-есть они создают индекс не сообщая вам об этом. Остается фактом, что вероятность беспорядка в базе данных достаточно мала, если вы предписываете уникальность вместе с ограничением.

УНИКАЛЬНОСТЬ КАК ОГРАНИЧЕНИЕ СТОЛБЦА

Время от времени, вы хотите убедиться, что все значения введеные в столбец отличаются друг от друга. Например, первичные ключи достаточно ясно это показывают.

Если вы помещаете ограничение столбца UNIQUE в поле при создании таблицы, база данных отклонит любую попытку ввода в это поле для одной из строк, значения, которое уже представлено в другой строке. Это ограничение может применяться только к полям которые были обьявлены как *непустые* (NOT NULL), так как не имеет

смысла позволить одной строке таблицы иметь значение NULL, а затем исключать другие строки с NULL значениями как дубликаты.

Имеется дальнейшее усовершенствование нашей команды создания таблицы Продавцов:

```
CREATE TABLE Salespeople
( Snum  integer NOT NULL UNIQUE,
  Sname char (10) NOT NULL UNIQUE,
  city  char (10),
  comm  decimal );
```

Когда вы обьявляете поле sname уникальным, убедитесь, что две Mary Smith будут введены различными способами — например, Mary Smith и M. Smith. В то же время это не так уж необходимо с функциональной точки зрения — потому что поле snum в качестве первичного ключа, все равно обеспечит отличие этих двух строк — что проще для людей, использующих данные в таблицах, чем помнить что эти Smith не идентичны.

Столбцы (не первичные ключи) чьи значения требуют уникальности, называются *ключами-кандидатами* или *уникальными ключами*.

УНИКАЛЬНОСТЬ КАК ОГРАНИЧЕНИЕ ТАБЛИЦЫ

Вы можете также определить группу полей как уникальную с помощью команды ограничения таблицы — UNIQUE. Объявление группы полей уникальной, отличается от объявления уникальными индивидуальных полей, так как это комбинация значений, а не просто индивидуальное значение, которое обязано быть уникальным.

Уникальность группы — это представление порядка, так что бы пары строк со значениями столбцов "a", "b" и "b", "a" рассматривались отдельно одна от другой.

Наша база данных сделана так, чтобы каждый заказчик был назначен одному и только одному продавцу. Это означает, что каждая комбинация номера заказчика (cnum) и номера продавца (snum) в таблице Заказчиков должна быть уникальной. Вы можете убедиться в этом, создав таблицу Заказчиков таким способом:

```
CREATE TABLE Customers
( cnum   integer NOT NULL,
  cname  char (10) NOT NULL,
  city   char (10),
  rating integer,
  snum   integer NOT NULL,
  UNIQUE (cnum, snum));
```

Обратите внимание что оба поля в ограничении таблицы UNIQUE все еще используют ограничение столбца — NOT NULL. Если бы мы использовали ограничение столбца UNIQUE для поля cnum, такое ограничение таблицы было бы необязательным.

Если значения поля cnum различно для каждой строки, то не может быть двух строк с идентичной комбинацией значений полей cnum и snum. То же самое получится если мы обьявим поле snum уникальным, хотя это и не будет соответствовать нашему примеру, так как продавец будет назначен многочисленым заказчикам. Следовательно, ограничение таблицы — UNIQUE, наиболее полезно когда вы не хотите заставлять индивидуальные поля быть уникальными.

Предположим, например, что мы разработали таблицу чтобы следить за всеми порядками каждый день для каждого продавца. Каждая строка такой таблицы представляет сумму чисел любых порядков, а не просто индивидуальный порядок. В этом случае, мы могли бы устранить некоторые возможные ошибки убедившись что на ка-

ждый день имеется не более чем одна строка для данного продавца, или что каждая комбинация полей snum и odate является уникальной. Вот как например мы могли бы создать таблицу с именем **Salestotal**:

```
CREATE TABLE Salestotal
( cnum   integer NOT NULL,
  odate  date NULL,
  totamt decimal,
  UNIQUE (snum, odate));
```

Кроме того, имеется команда, которую вы будете использовать, чтобы помещать текущие данные в эту таблицу:

```
INSERT INTO Salestotal
SELECT snum, odate, SUM (amt)
FROM Orders
GROUP BY snum, odate;
```

ОГРАНИЧЕНИЕ ПЕРВИЧНЫХ КЛЮЧЕЙ

До этого мы воспринимали первичные ключи исключительно как логические понятия. Хоть мы и знаем что такое первичный ключ, и как он должен использоваться в любой таблице, мы не ведаем "*знает*" ли об этом SQL. Поэтому мы использовали ограничение UNIQUE или уникальные индексы в первичных ключах, чтобы предписывать им уникальность. В более ранних версиях языка SQL это было необходимо и могло выполняться этим способом. Однако теперь SQL поддерживает первичные ключи непосредственно с ограничением *Первичный Ключ* (**PRIMARY KEY**). Это ограничение может быть доступным или недоступным вашей системе.

PRIMARY KEY может ограничивать таблицы или их столбцы. Это ограничение работает так же как и ограничение UNIQUE, за исключением когда только один первичный ключ (для любого числа столбцов) может быть определен для данной таблицы. Имеется также различие между первичными ключами и уникальностью столбцов в способе их использоваться с внешними ключами, о которых будет рассказано в Главе 19. Синтаксис и определение их уникальности те же что и для ограничения UNIQUE.

Первичные ключи не могут позволять значений NULL. Это означает что, подобно полям в ограничении UNIQUE, любое поле, используемое в ограничении PRIMARY KEY, должно уже быть обьявлено NOT NULL.

Имеется улучшеный вариант создания нашей таблицы Продавцов:

```
CREATE TABLE Salestotal
( snum  integer NOT NULL PRIMARY KEY,
  sname char(10) NOT NULL UNIQUE,
  city  char(10),
  comm  decimal );
```

Как вы видете, уникальность (UNIQUE) полей может быть обьявлена для той же самой таблицы. Лучше всего помещать ограничение PRIMARY KEY в поле (или в поля) которое будет образовывать ваш уникальный идентификатор строки, и сохранить ограничение UNIQUE для полей которые должны быть уникальными логически (такие как номера телефона или поле sname), а не для идентификации строк.

ПЕРВИЧНЫЕ КЛЮЧИ БОЛЕЕ ЧЕМ ОДНОГО ПОЛЯ

Ограничение PRIMARY KEY может также быть применено для многочисленных полей, составляющих уникальную комбинацию значений. Предположим что ваш первичный ключ — это имя, и вы имеете первое имя и последнее имя сохраненными в двух различных полях (так что вы можете организовывать данные с помощью любого из них). Очевидно, что ни первое ни последнее имя нельзя заставить быть уникальным самостоятельно, но мы можем каждую из этих двух комбинаций сделать уникальной.

Мы можем применить ограничение таблицы PRIMARY KEY для пар:

```
CREATE TABLE Namefield
(firstname char (10) NOT NULL,
 lastname  char (10) NOT NULL
 city      char (10),
PRIMARY KEY (firstname, lastname));
```

Одна проблема в этом подходе та, что мы можем вынудить появление уникальности — например, введя Mary Smith и M. Smith. Это может ввести в заблуждение, потому что ваши служащие могут не знать кто из них кто. Обычно более надежный способ чтобы определять числовое поле которое могло бы отличать одну строку от другой, это иметь первичный ключ, и применять ограничение UNIQUE для двух имен полей.

ПРОВЕРКА ЗНАЧЕНИЙ ПОЛЕЙ

Конечно, имеется любое число ограничений, которые можно устанавливать для данных, вводимых в ваши таблицы, чтобы видеть, например, находятся ли данные в соответствующем диапазоне или правильном формате, о чем SQL естественно не может знать заранее. По этой причине, SQL обеспечивает вас ограничением **CHECK**, которое позволяет вам установить условие, которому должно удовлетворять значение, вводимое в таблицу, прежде чем оно будет принято. Ограничение CHECK состоит из ключевого слова CHECK сопровождаемого предложением предиката, который использует указанное поле. Любая попытка модифицировать или вставить значение поля, которое могло бы сделать этот предикат неверным — будет отклонена.

Давайте рассмотрим еще раз таблицу Продавцов. Столбец комиссионных выражается десятичным числом и поэтому может быть умножен непосредственно на сумму приобретений, в результате чего будет получена сумма комиссионных (в долларах) продавца с установленым справа значком доллара (**$**).

Кто-то может использовать понятие процента, однако ведь, можно об этом и не знать. Если человек введет по ошибке **14** вместо **.14** чтобы указать в процентах свои комиссионные, это будет расценено как **14.0**, что является законным десятичным значением, и будет нормально воспринято системой. Чтобы предотвратить эту ошибку, мы можем наложить ограничение столбца — CHECK чтобы убедиться что вводимое значение меньше чем 1.

```
CREATE TABLE Salespeople
(snum integer NOT NULL PRIMARY KEY,
sname char(10) NOT NULL UNIQUE,
city  char(10),
comm  decimal CHECK (comm < 1));
```

ИСПОЛЬЗОВАНИЕ CHECK, ЧТОБЫ ПРЕДОПРЕДЕЛЯТЬ ДОПУСТИМОЕ ВВОДИМОЕ ЗНАЧЕНИЕ

Мы можем также использовать ограничение CHECK чтобы защитить от ввода в поле определенных значений, и таким образом предотвратить ошибку. Например, предположим, что единственными городами в которых мы имели ведомства сбыта являются Лондон, Барселона, Сан Хосе, и Нью Йорк. Если вам известны все продавцы, работающие в каждом из этих ведомств, нет необходимости позволять ввод других значений. Если же нет, использование ограничения может предотвратить опечатки и другие ошибки.

```
CREATE TABLE Salespeople
(snum integer NOT NULL UNIQUE,
 sname char(10) NOT NULL UNIQUE,
 city  char(10)
   CHECK (city IN ('London', 'New York', 'San Jose', 'Barselona')),
 comm  decimal CHECK (comm<1));
```

Конечно, если вы собираетесь сделать это, вы должны быть уверены что ваша компания не открыла уже новых других ведомств сбыта. Большинство программ баз данных поддерживают команду **ALTER TABLE** (см. Главу 17) которая позволяет вам изменять определение таблицы, даже когда она находится в использовании. Однако, изменение или удаление ограничений не всегда возможно для этих команд, даже там, где это вроде бы поддерживается.

Если вы использовали систему, которая не может удалять ограничения, вы будете должны *создавать* (**CREATE**) новую таблицу и передавать информацию из старой таблицы в нее всякий раз, когда вы хотите изменить ограничение. Конечно же Вы не захотите делать это часто, и со временем вообще перестанете это делать.

Создадим таблицу Порядков:

```
CREATE TABLE Orders
(onum  integer NOT NULL UNIQUE,
 amt   decimal,
 odate date NOT NULL,
 cnum  integer NOT NULL,
 snum  integer NOT NULL);
```

Как мы уже говорили в Главе 2, тип **DATE** (ДАТА) широко поддерживается, но не является частью стандарта ANSI. Что же делать, если мы используем базу данных которая, следуя ANSI, не распознает тип DATE? Если мы обьявим поле odate любым типом числа, мы не сможем использовать наклонную черту вправо (/) или черточку (-) в качестве разделителя. Так как печатаемые номера — это символы ASCII, мы можем обьявить тип поля odate — **CHAR**. Основная проблема в том, что мы будем должны использовать одиночные кавычки всякий раз, когда ссылаемся на значение поля odate в запросе. Нет более простого решения этой проблемы там где тип DATE стал таким популярным. В качестве иллюстрации, давайте обьявим поле odate — типом CHAR. Мы можем по крайней мере наложить на него наш формат с ограничением CHECK:

```
CREATE TABLE Orders
(onum  integer NOT NULL UNIQUE,
 amt   decimal,
 odate char (10) NOT NULL CHECK (odate LIKE '--/--/----'),
 cnum  NOT NULL,
 snum  NOT NULL);
```

Кроме того, если вы хотите, вы можете наложить ограничение, гарантирующие что введенные символы — числа, и что они — в пределах значений нашего диапазона.

ПРОВЕРКА УСЛОВИЙ, БАЗИРУЮЩИЙСЯ НА МНОГОЧИСЛЕНЫХ ПОЛЯХ

Вы можете также использовать CHECK в качестве табличного ограничения. Это полезно в тех случаях, когда вы хотите включить более одного поля строки в условие. Предположим, что комиссионные .15 и выше будут разрешены только для продавца из Барселоны. Вы можете указать это со следующим табличным ограничением CHECK:

```
CREATE TABLE Salespeople
(snum  integer NOT NULL UNIQUE,
 sname char (10) NOT NULL UNIQUE,
 city  char(10),
 comm  decimal,
  CHECK (comm < .15 OR city = 'Barcelona'));
```

Как вы можете видеть, два различных поля должны быть проверены, чтобы определить, верен предикат или нет. Имейте в виду, что это — два разных поля одной и той же строки. Хотя вы можете использовать многочисленые поля, SQL не может проверить более одной строки одновременно. Вы не можете например использовать ограничение CHECK, чтобы удостовериться, что все комиссионные в данном городе одинаковы. Чтобы сделать это, SQL должен всякий раз, просматривая другие строки таблицы, когда вы модифицируете или вставляете строку, видеть, что значение комиссионных указано для текущего города. SQL этого делать не умеет.

Фактически, вы могли бы использовать сложное ограничение CHECK для вышеупомянутого, если бы знали заранее, каковы должны быть комиссионные в разных городах. Например, вы могли бы установить ограничение типа этого:

```
CHECK ((comm = .15 AND clty = 'London')
    OR (comm = .14 AND city = 'Barcelona')
    OR (comm = .11 AND city = 'San Jose').. )
```

Вы получили идею. Чем налагать такой комплекс ограничений, вы могли бы просто использовать *представление* с предложением **WITH CHECK OPTION**, которое имеет все эти условия в своем предикате (смотри Главу 20 и 21 для информации о *представлении* и о WITH CHECK OPTION). Пользователи могут обращаться к *представлению таблицы* вместо самой таблицы. Одним из преимуществ этого будет то, что процедура изменения в ограничении не будет такой болезненной или трудоемкой. Представление с WITH CHECK OPTION — хороший заменитель ограничению CHECK, что будет показано в Главе 21.

УСТАНОВКА ЗНАЧЕНИЙ ПОУМОЛЧАНИЮ

Когда вы вставляете строку в таблицу без указания значений в ней для каждого поля, SQL должен иметь *значение по умолчанию* для включения его в определенное поле, или же команда будет отклонена. Наиболее общим значением по умолчанию является **NULL**. Это — значение по умолчанию для любого столбца, которому не было дано ограничение NOT NULL или который имел другое назначение *по умолчанию*.

Значение **DEFAULT** (ПО УМОЛЧАНИЮ) указывается в команде CREATE TABLE тем же способом, что и ограничение столбца, хотя, с технической точки зрения, значение DEFAULT не ограничительного свойства — оно не ограничивает значения, которые вы можете вводить, а просто определяет, что может случиться если вы не введете любое из них.

Предположим что вы работаете в оффисе Нью Йорка и подавляющее большинство ваших продавцов живут в Нью Йорке. Вы можете указать Нью Йорк в качестве значения поля city, по умолчанию, для вашей таблицы Продавцов:

```
CREATE TABLE Salespeople
(snum  integer NOT NULL UNIQUE,
 sname char(10) NOT NULL UNIQUE,
 city  char(10) DEFAULT = 'New York',
 comm  decimal CHECK (comm < 1));
```

Конечно, вводить значение Нью Йорк в таблицу каждый раз когда назначается новый продавец, не такая уж необходимость, и можно просто пренебречь им (не вводя его) даже если оно должно иметь некоторое значение. *Значение по умолчанию* такого типа, более предпочтительно, чем, например, длинный конторский номер, указывающий на ваше собственное ведомство, в таблице Порядков.

Длинные числовые значения — более расположены к ошибке, поэтому если подавляющее большинство (или все) ваших порядков должны иметь ваш собственный конторский номер, желательно устанавливать для них значение *по умолчанию*.

Другой способ использовать значение по умолчанию — это использовать его как альтернативу для NULL. Так как NULL (фактически) неверен при любом сравнении, ином чем IS NULL, он может быть исключен с помощью большинства предикатов. Иногда, вам нужно видеть пустые значения ваших полей не обрабатывая их каким-то определенным образом. Вы можете установить значение по умолчанию, типа *нуль* или *пробел*, которые функционально меньше по значению чем просто не установленное значение — *пустое* значение (NULL). Различие между ними и обычным NULL в том, что SQL будет обрабатывать их также как и любое другое значение.

Предположим, что заказчикам не назначены оценки изначально. Каждые шесть месяцев, вы повышаете оценку всем вашим заказчикам, имеющим оценку ниже средней, включая и тех кто предварительно не имел никакого назначения оценки. Если вы хотите выбрать всех этих заказчиков как группу, следующий запрос исключит всех заказчиков с оценкой = NULL:

```
SELECT *
FROM Customers
WHERE rating <= 100;
```

Однако, если вы назначили *значение по умолчанию* = 000, в поле rating, заказчики без оценок будут выбраны наряду с другими. Приоритет каждого метода зависит от ситуации.

Если вы будете делать запрос с помощью поля оценки, то захотите ли Вы включить строки без значений, или исключите их?

Другая характеристика значений по умолчанию этого типа, позволит обьявить Вам поле оценки — как **NOT NULL.**

Если вы используете его по умолчанию, чтобы избежать значений = NULL, то это — вероятно хорошая защита от ошибок.

Вы можете также использовать ограничения UNIQUE или PRIMARY KEY в этом поле. Если вы сделаете это, то, имеете в виду, что только одна строка одновременно может иметь значение по умолчанию. Любую строку, которая содержит значение по умолчанию, нужно будет модифицировать прежде, чем другая строка с установкой по умолчанию будет вставлена. Это не так как вы обычно используете значения по умол-

чанию, поэтому ограничения UNIQUE и PRIMARY KEY (особенно последнее) обычно не устанавливаются для строк со значениями по умолчанию.

РЕЗЮМЕ

Вы теперь владеете несколькими способами управления значениями которые могут быть введены в ваши таблицы. Вы можете использовать ограничение NOT NULL чтобы исключать NULL, ограничение UNIQUE чтобы вынуждать все значения в группе из одного или более столбцов отличаться друг от друга, ограничение PRIMARY KEY, для того чтобы делать в основном то же самое что и UNIQUE но с различным окончанием, и наконец ограничение CHECK для определения ваших собственных сделанных на заказ условий, чтобы значения встреченные перед ними могли бы быть введены. Кроме того, вы можете использовать предложение DEFAULT, которое будет автоматически вставлять значение по умолчанию в любое поле с именем не указаным в INSERT, так же как вставляется значение NULL когда предложение DEFAULT не установлено и отсутствует ограничение NOT NULL.

FOREIGN KEY или **REFERENCES** ограничения, о которых вы узнаете в Главе 19, очень похожи на них, за исключением того, что они связывают группу из одного или более полей с другой группой, и таким образом сразу воздействуют на значения, которые могут быть введены в любую из этих групп.

РАБОТА С SQL

1. Создайте таблицу Порядков, так чтобы все значения поля onum, а также все комбинации полей cnum и snum отличались друг от друга, и так чтобы значения NULL исключались из поля даты.
2. Создайте таблицу Продавцов, так чтобы комиссионные, по умолчанию составляли 10%, не разрешались значения NULL, чтобы поле snum являлось первичным ключом, и чтобы все имена были в алфавитном порядке между A и M включительно (учитывая, что все имена будут напечатаны в верхнем регистре).
3. Создайте таблицу Порядков, будучи уверенными в том что поле onum больше чем поле cnum, а cnum больше чем snum. Запрещены значения NULL в любом из этих трех полей.

(См. Приложение A для ответов.)

19

ПОДДЕРЖКА ЦЕЛОСТНОСТИ ВАШИХ ДАННЫХ

РАНЕЕ В ЭТОЙ КНИГЕ, МЫ УКАЗЫВАЛИ НА ОПРЕДЕЛЕННЫЕ связи, которые существуют между некоторыми полями наших типовых таблиц. Поле snum таблицы Заказчиков, например, соответствует полю snum в таблице Продавцов и таблице Порядков. Поле cnum таблицы Заказчиков также соответствует полю cnum таблицы Порядков. Мы назвали этот тип связи — *справочной целостностью*; и в ходе обсуждения, вы видели как ее можно использовать.

В этой главе вы будете исследовать справочную целостность более подробно и выясним все относительно ограничений, которые вы можете использовать, чтобы ее поддерживать. Вы также увидете, как предписывается это ограничение, когда вы используете команды модификации DML. Поскольку справочная целостность включает в себя связь полей или групп полей, часто в разных таблицах, это действие может быть несколько сложнее, чем другие ограничения. По этой причине, хорошо иметь с ней полное знакомство, даже если вы не планируете создавать таблицы. Ваши команды модификации могут стать эффективнее с помощью ограничения справочной целостности (как и с помощью других ограничений, но ограничение справочной целостности может воздействовать на другие таблицы кроме тех, в которых оно определено), а определенные функции запроса, такие как обьединения, являются многократно структурированы в терминах связей справочной целостности (как подчеркивалось в Главе 8).

ВНЕШНИЙ КЛЮЧ И РОДИТЕЛЬСКИЙ КЛЮЧ

Когда все значения в одном поле таблицы представлены в поле другой таблицы, мы говорим что *первое поле ссылается на второе*. Это указывает на прямую связь между значениями двух полей. Например, каждый из заказчиков в таблице Заказчиков имеет поле snum которое указывает на продавца назначенного в таблице Продавцов. Для каждого порядка в таблице Порядков, имеется один и только этот продавец и один и только этот заказчик. Это отображается с помощью полей snum и cnum в таблице Порядков.

Когда одно поле в таблице ссылается на другое, оно называется — *внешним ключом*; а поле на которое оно ссылается, называется — *родительским ключом*. Так что поле snum таблицы Заказчиков — это внешний ключ, а поле snum на которое оно ссылается в таблице Продавцов — это родительский ключ.

Аналогично, поля cnum и snum таблицы Порядков — это внешние ключи, которые ссылаются к их родительским ключам с именами в таблице Заказчиков и таблице Продавцов. Имена внешнего ключа и родительского ключа не обязательно должны быть одинаковыми, это — только соглашение которому мы следуем чтобы делать соединение более понятным.

МНОГО-СТОЛБЦОВЫЕ ВНЕШНИЕ КЛЮЧИ

В действительности, внешний ключ не обязательно состоит только из одного поля. Подобно первичному ключу, внешний ключ может иметь любое число полей, которые все обрабатываются как единый модуль. Внешний ключ и родительский ключ, на который он ссылается, конечно же, должны иметь одинаковый номер и тип поля, и находиться в одинаковом порядке. Внешние ключи, состоящие из одного поля — те что мы использовали исключительно в наших типовых таблицах, наиболее общие.

Чтобы сохранить простоту нашего обсуждения, мы будем часто говорить о внешнем ключе как об одиночном столбце. Это не случайно. Если это не отметить, любой скажет о поле, которое является внешним ключом, что оно также относится и к группе полей, которая является внешним ключом.

СМЫСЛ ВНЕШНЕГО И РОДИТЕЛЬСКОГО КЛЮЧЕЙ

Когда поле является внешним ключом, оно определеным образом связано с таблицей, на которую он ссылается. Вы, фактически, говорите — "*каждое значение в этом поле (внешнем ключе) непосредственно привязано к значению в другом поле (родительском ключе*)." Каждое значение (каждая строка) внешнего ключа должно недвусмысленно ссылаться к одному и только этому значению (строке) родительского ключа. Если это так, то фактически ваша система, как говорится, будет *в состоянии справочной целостности*.

Вы можете увидеть это на примере. Внешний ключ snum в таблице Заказчиков имеет значение 1001 для строк Hoffman и Clemens.

Предположим, что мы имели две строки в таблице Продавцов со значением в поле snum = 1001.

Как мы узнаем, к которому из двух продавцов были назначены заказчики Hoffman и Clemens? Аналогично, если нет никаких таких строк в таблице Продавцов, мы получим Hoffman и Clemens назначенными к продавцу, которого не существует!

Понятно, что каждое значение во внешнем ключе должно быть представлено один, и только один раз, в родительском ключе.

Фактически, данное значение внешнего ключа может ссылаться только к одному значению родительского ключа не предполагая обратной возможности: т.е. любое число внешних ключей может ссылать к единственному значению родительского ключа. Вы можете увидеть это в типовых таблицах наших примеров. И Hoffman и Clemens назначены к Peel, так что оба их значения внешнего ключа совпадают с одним и тем же родительским ключом, что очень хорошо. Значение внешнего ключа должно ссылаться только к одному значению родительского ключа, зато значение родительского ключа может ссылаться с помощью любого колличества значений внешнего ключа.

В качестве иллюстрации, значения внешнего ключа из таблицы Заказчиков, совпавшие с их родительским ключом в Продавцов таблице, показываются в Рисунке 19.1. Для удобства мы не учитывали поля, не относящиеся к этому примеру.

ОГРАНИЧЕНИЕ FOREIGN KEY

SQL поддерживает справочную целостность с ограничением **FOREIGN KEY**. Хотя ограничение **FOREIGN KEY** — это новая особенность в SQL, оно еще не обеспечивает его универсальности. Кроме того, некоторые его реализации более сложны чем другие. Эта функция должна ограничивать значения, которые вы можете ввести в вашу базу данных, чтобы заставить внешний ключ и родительский ключ соответствовать принципу справочной целостности.

Одно из действий ограничения Внешнего Ключа — это отбрасывание значений для полей, ограниченных как внешний ключ, который еще не представлен в родительском ключе. Это ограничение также воздействует на вашу способность изменять или удалять значения родительского ключа (мы будем обсуждать это позже в этой главе).

КАК МОЖНО ПОЛЯ ПРЕДСТАВИТЬ В КАЧЕСТВЕ ВНЕШНИХ КЛЮЧЕЙ

Вы используете ограничение FOREIGN KEY в команде CREATE TABLE (или ALTER TABLE), которая содержит поле которое вы хотите обьявить внешним ключом. Вы даете имя родительскому ключу, на которое вы будете ссылаться внутри ограничения FOREIGN KEY. Помещение этого ограничения в команду — такое же, что для других ограничений, обсужденных в предыдущей главе.

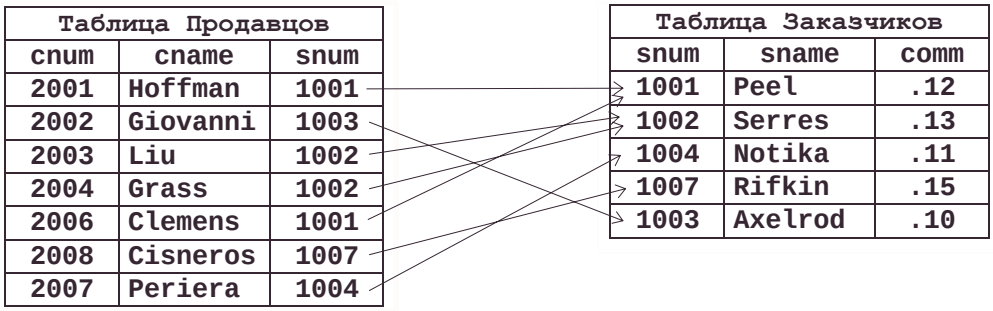

Таблица Продавцов		
cnum	cname	snum
2001	Hoffman	1001
2002	Giovanni	1003
2003	Liu	1002
2004	Grass	1002
2006	Clemens	1001
2008	Cisneros	1007
2007	Periera	1004

Таблица Заказчиков		
snum	sname	comm
1001	Peel	.12
1002	Serres	.13
1004	Notika	.11
1007	Rifkin	.15
1003	Axelrod	.10

Рисунок 19.1: Внешний Ключ таблицы Заказчиков с родительским ключом

Подобно большинству ограничений, оно может быть ограничением таблицы или столбца, в форме таблицы позволяющей использовать многочисленые поля как один внешний ключ.

ВНЕШНИЙ КЛЮЧ КАК ОГРАНИЧЕНИЕ ТАБЛИЦЫ

Синтаксис ограничения таблицы FOREIGN KEY:

```
FOREIGN KEY <column list> REFERENCES <pktable> [ <column list> ]
```

Первый список столбцов — это список из одного или более столбцов таблицы, которые отделены запятыми и будут созданы или изменены этой командой. **Pktable** — это таблица содержащая родительский ключ. Она может быть таблицей, которая создается или изменяется текущей командой. Второй список столбцов — это список столбцов которые будут составлять родительский ключ. Списки двух столбцов должны быть совместимы, т.е.:

* Они должны иметь одинаковое число столбцов.

* В данной последовательности, первый, второй, третий, и т.д., столбцы списка столбцов внешнего ключа, должны иметь одинаковые типы данных и размеры, что и первый, второй, третий, и т.д., столбцы списка столбцов родительского ключа. Столбцы в списках обоих столбцов не должны иметь одинаковых имен, хотя мы и использовали такой способ в наших примерах чтобы делать связь более понятной.

Создадим таблицу Заказчиков с полем snum определенным в качестве внешнего ключа ссылающегося на таблицу Продавцов:

```
CREATE TABLE Customers
(cnum  integer NOT NULL PRIMARY KEY
 cname char(10),
 city  char(10),
 snum  integer,
FOREIGN KEY (snum) REFERENCES Salespeople (snum));
```

Имейте в виду, что при использовании ALTER TABLE вместо CREATE TABLE, для применения ограничения FOREIGN KEY, значения которые Вы указываете во внешнем ключе и родительском ключе, должны быть в состоянии справочной целостности. Иначе команда будет отклонена. Хотя ALTER TABLE очень полезна из-за ее удобства, вы должны будете в вашей системе, по возможности каждый раз, сначала формировать структурные принципы, типа справочной целостности.

ВНЕШНИЙ КЛЮЧ КАК ОГРАНИЧЕНИЕ СТОЛБЦОВ

Вариант ограничения столбца ограничением FOREIGN KEY — по другому называется — *ссылочное ограничение* (REFERENCES), так как он фактически не содержит в себе слов FOREIGN KEY, а просто использует слово REFERENCES, и далее имя родительского ключа, подобно этому:

```
CREATE TABLE Customers
(cnum  integer NOT NULL PRIMARY KEY,
 cname char(10),
 city  char(10),
 snum  integer REFERENCES Salespeople (snum));
```

Вышеупомянутое определяет **Customers.snum** как внешний ключ у которого родительский ключ — это **Salespeople.snum**. Это эквивалентно такому ограничению таблицы:

```
FOREIGN KEY (snum) REGERENCES Salespeople (snum)
```

НЕ УКАЗЫВАТЬ СПИСОК СТОЛБЦОВ ПЕРВИЧНЫХ КЛЮЧЕЙ

Используя ограничение FOREIGN KEY таблицы или столбца, вы можете не указывать список столбцов родительского ключа если родительский ключ имеет ограничение PRIMARY KEY. Естественно, в случае ключей со многими полями, порядок столбцов во внешних и первичных ключах должен совпадать, и, в любом случае, принцип совместимости между двумя ключами все еще применим. Например, если мы поместили ограничение PRIMARY KEY в поле snum таблицы Продавцов, мы могли бы использовать его как внешний ключ в таблице Заказчиков (подобно предыдущему примеру) в этой команде:

```
CREATE TABLE Customers
(cnum  integer NOT NULL PRIMARY KEY,
 cname char(10),
 city  char(10),
 snum  integer REFERENCES Salespeople);
```

Это средство встраивалось в язык, чтобы поощрять вас использовать первичные ключи в качестве родительских ключей.

КАК СПРАВОЧНАЯ ЦЕЛОСТНОСТЬ ОГРАНИЧИВАЕТ ЗНАЧЕНИЯ РОДИТЕЛЬСКОГО КЛЮЧА

Поддержание справочной целостности требует некоторых ограничений на значения, которые могут быть представлены в полях, обьявленных как внешний ключ и родительский ключ. Родительский ключ должен быть структурен, чтобы гарантировать, что каждое значение внешнего ключа будет соответствовать одной указанной строке. Это означает, что он (ключ) должен быть уникальным и не содержать никаких пустых значений (NULL). Этого не достаточно для родительского ключа в случае выполнения такого требования как при объявлении внешнего ключа. SQL должен быть уверен что двойные значения или пустые значения (NULL) не были введены в родительский ключ. Следовательно вы должны убедиться, что все поля, которые исполь-

зуются как родительские ключи, имеют или ограничение PRIMARY KEY или ограничение UNIQUE, наподобии ограничения NOT NULL.

ПЕРВИЧНЫЙ КЛЮЧ КАК УНИКАЛЬНЫЙ ВНЕШНИЙ КЛЮЧ

Ссылка ваших внешних ключей только на первичные ключи, как мы это делали в типовых таблицах, — хорошая стратегия. Когда вы используете внешние ключи, вы связываете их не просто с родительскими ключами, на которые они ссылаются; вы связываете их с определенной строкой таблицы, где этот родительский ключ будет найден. Сам по себе родительский ключ не обеспечивает никакой информации, которая бы не была уже представлена во внешнем ключе. Смысл, например, поля snum как внешнего ключа в таблице Заказчиков — это связь, которую он обеспечивает, не к значению поля snum, на которое он ссылается, а к другой информации в таблице Продавцов, такой например как имена продавцов, их местоположение, и так далее. *Внешний ключ* — это не просто связь между двумя идентичными значениями; это — связь, с помощью этих двух значений, между двумя строками таблицы указанной в запросе.

Это поле snum может использоваться чтобы связывать любую информацию в строке из таблицы Заказчиков со ссылочной строкой из таблицы Продавцов — например чтобы узнать — живут ли они в том же самом городе, кто имеет более длинное имя, имеет ли продавец кроме данного заказчика каких-то других заказчиков, и так далее.

Так как цель первичного ключа состоит в том, чтобы идентифицировать уникальность строки, это более логичный и менее неоднозначный выбор для внешнего ключа. Для любого внешнего ключа который использует уникальный ключ как родительский ключ, вы должны создать внешний ключ который бы использовал первичный ключ той же самой таблицы для того же самого действия. Внешний ключ который не имеет никакой другой цели кроме связывания строк, напоминает первичный ключ используемый исключительно для идентификации строк, и является хорошим средством сохранить структуру вашей базы данных ясной и простой, и — следовательно создающей меньше трудностей.

ОГРАНИЧЕНИЯ ВНЕШНЕГО КЛЮЧА

Внешний ключ, в частности, может содержать только те значения которые фактически представлены в родительском ключе или пустые (NULL). Попытка ввести другие значения в этот ключ будет отклонена.

Вы можете обьявить внешний ключ как NOT NULL, но это необязательно, и в большинстве случаев, нежелательно. Например, предположим, что вы вводите заказчика не зная заранее, к какому продавцу он будет назначен. Лучший выход в этой ситуации будет, если использовать значение NOT NULL, которое должно быть изменено позже на конкретное значение.

ЧТО СЛУЧИТСЯ, ЕСЛИ ВЫ ВЫПОЛНИТЕ КОМАНДУ МОДИФИКАЦИИ

Давайте условимся, что все внешние ключи созданые в наших таблицах примеров, обьявлены и предписаны с ограничениями внешнего ключа, следующим образом:

```
CREATE TABLE Salespeople
(snum  integer NOT NULL PRIMARY KEY,
 sname char(10) NOT NULL,
 city  char(10),
 comm  decimal);

CREATE TABLE Customers
(cnum   integer NOT NULL PRIMARY KEY,
 cname  char(10) NOT NULL,
 city   char(10),
 rating integer,
 snum   integer,
 FOREIGN KEY (snum) REFERENCES Salespeople,
 UNIQUE (cnum, snum));

CREATE TABLE Orders
(cnum  integer NOT NULL PRIMARY KEY,
 amt   decimal,
 odate date NOT NULL,
 cnum  integer NOT NULL
 snum  integer NOT NULL
 FOREIGN KEY (cnum, snum) REFERENCES
 CUSTOMERS (cnum, snum));
```

ВКЛЮЧЕНИЕ ОПИСАНИЙ ТАБЛИЦЫ

Имеется несколько атрибутов таких определений, о которых нужно поговорить. Причина, по которой мы решили сделать поля cnum и snum в таблице Порядков единым внешним ключом — это гарантия того, что для каждого заказчика, содержащегося в порядках, продавец, кредитующий этот порядок — тот же, что и указаный в таблице Заказчиков. Чтобы создать такой внешний ключ, мы были бы должны поместить ограничение таблицы UNIQUE в два поля таблицы Заказчиков, даже если оно необязательно для самой этой таблицы. Пока поле cnum в этой таблица имеет ограничение PRIMARY KEY, оно будет уникально в любом случае, и следовательно невозможно получить еще одну комбинацию поля cnum с каким-то другим полем.

Создание внешнего ключа таким способом поддерживает целостность базы данных, даже если при этом вам будет запрещено внутреннее прерывание по ошибке и кредитовать любого продавца, иного чем тот который назначен именно этому заказчику.

С точки зрения поддержания целостности базы данных, внутренние прерывания (или *исключения*) конечно же нежелательны. Если вы их допускаете и в то же время хотите поддерживать целостность вашей базы данных, вы можете обьявить поля snum и cnum в таблице Порядков независимыми внешними ключами этих полей в таблице Продавцов и таблице Заказчиков, соответственно.

Фактически, использование поля snum в таблице Порядков, как мы это делали, необязательно, хотя это полезно было сделать для разнообразия. Поле cnum связывая каждый порядок заказчиков в таблице Заказчиков, в таблице Порядков и в таблице Заказчиков, должно всегда быть общим, чтобы находить правильное поле snum для данного порядка (не разрешая никаких исключений). Это означает что мы записываем фрагмент информации — какой заказчик назначен к какому продавцу — дважды, и нужно будет выполнять дополнительную работу чтобы удостовериться, что обе версии согласуются.

Если мы не имеем ограничения внешнего ключа как сказано выше, эта ситуация будет особенно проблематична, потому что каждый порядок нужно будет проверять вручную (вместе с запросом), чтобы удостовериться что именно соответствующий продавец кредитовал каждую соответствующую продажу. Наличие такого типа ин-

формационной избыточности в вашей базе данных, называется д*енормализация* (denormalization), что не желательно в идеальной реляционной базе данных, хотя практически и может быть разрешена. Денормализация может заставить некоторые запросы выполняться быстрее, поскольку запрос в одной таблице выполняется всегда значительно быстрее, чем в обьединении.

ДЕЙСТВИЕ ОГРАНИЧЕНИЙ

Как такие ограничения воздействуют на возможность и невозможность Вами использовать команды модификации DML? Для полей, определенных как внешние ключи, ответ довольно простой: любые значения которые вы помещаете в эти поля с командой INSERT или UPDATE должны уже быть представлены в их родительских кючах. Вы можете помещать пустые (NULL) значения в эти поля, несмотря на то что значения NULL не позволительны в родительских ключах, если они имеют ограничение NOT NULL. Вы можете удалять (**DELETE**) любые строки с внешними ключами не используя родительские ключи вообще.

Поскольку затронут вопрос об изменении значений родительского ключа, ответ, по определению ANSI, еще проще, но возможно несколько более ограничен: любое значение родительского ключа, ссылаемого с помощью значения внешнего ключа, не может быть удалено или изменено. Это означает, например, что вы не можете удалить заказчика из таблицы Заказчиков, пока он еще имеет порядки в таблице Порядков. В зависимости от того, как вы используете эти таблицы, это может быть или желательно или хлопотно. Однако — это конечно лучше, чем иметь систему, которая позволит вам удалить заказчика с текущими порядками и оставить таблицу Порядков ссылающейся на несуществующих заказчиков. Смысл этой системы оганичения в том, что создатель таблицы Порядков, используя таблицу Заказчиков и таблицу Продавцов как родительские ключи, может наложить значительные ограничения на действия в этих таблицах. По этой причине, вы не сможете использовать таблицу, которой вы не распоряжаетесь (т.е. не вы ее создавали и не вы являетесь ее владельцем), пока владелец (создатель) этой таблицы специально не передаст вам на это право (что объясняется в Главе 22).

Имеются некоторые другие возможные действия изменения родительского ключа, которые не являются частью ANSI, но могут быть найдены в некоторых коммерческих программах. Если вы хотите изменить или удалить текущее ссылочное значение родительского ключа, имеется по существу три возможности:

* Вы можете ограничить, или запретить, изменение (способом ANSI), обозначив, что изменения в родительском ключе — *ограничены*.

* Вы можете сделать изменение в родительском ключе и тем самым сделать изменения во внешнем ключе автоматическим, что называется — *каскадным* изменением.

* Вы можете сделать изменение в родительском ключе, и установить внешний ключ в NULL автоматически (полагая, что NULL разрешен во внешнем ключе), что называется — *пустым* изменением внешнего ключа.

Даже в пределах этих трех категорий, вы можете не захотеть обрабатывать все команды модификации таким способом. INSERT, конечно, к делу не относится. Он помещает новые значения родительского ключа в таблицу, так что ни одно из этих значений не может быть вызвано в данный момент. Однако, вы можете захотеть позволить модификациям быть *каскадными*, но без удалений, и наоборот. Лучшей может быть ситуация, которая позволит вам определять любую из трех категорий, независимо от команд UPDATE и DELETE. Мы будем следовательно ссылаться на *эффект модификации* (update effects) и *эффект удаления* (delete effects), которые

определяют, что случится, если вы выполните команды UPDATE или DELETE в родительском ключе. Эти эффекты, о которых мы говорили, называются:

Ограниченные (**RESTRICTED**) изменения,

Каскадируемые (**CASCADES**) изменения, и

Пустые (**NULL**) изменения.

Фактические возможности вашей системы должны быть в строгом стандарте ANSI — это эффекты модификации и удаления, оба, автоматически ограниченнные — для более идеальной ситуации описаной выше. В качестве иллюстрации, мы покажем несколько примеров того, что вы можете делать с полным набором эффектов модификации и удаления. Конечно, эффекты модификации и удаления, являющиеся нестандартными средствами, испытывают недостаток в стандартном госинтаксисе. Синтаксис который мы используем здесь, прост в написании и будет служить в дальнейшем для иллюстрации функций этих эффектов.

Для полноты эксперимента, позволим себе предположить что вы имеете причину изменить поле snum таблицы Продавцов в случае, когда наша таблица Продавцов изменяет разделы. (Обычно изменение первичных ключей это не то что мы рекомендуем делать практически. Просто это еще один из доводов для имеющихся первичных ключей которые не умеют делать ничего другого кроме как, действовать как первичные ключи: они не должны изменяться.) Когда вы изменяете номер продавца, вы хотите чтобы были сохранены все его заказчики. Онако, если этот продавец покидает свою фирму или компанию, вы можете не захотеть удалить его заказчиков, при удалении его самого из базы данных. Взамен, вы захотите убедиться, что заказчики назначены кому-нибудь еще. Чтобы сделать это вы должны указать **UPDATE** с *Каскадируемым* эффектом, и **DELETE** с *Ограниченным* эффектом.

```
CREATE TABLE Customers
(cnum   integer NOT NULL PRIMARY KEY,
 cname  char(10) NOT NULL,
 city   char(10),
 rating integer,
 snum   integer REFERENCES Salespeople,
 UPDATE OF Salespeople CASCADES,
 DELETE OF Salespeople RESTRICTED);
```

Если вы теперь попробуете удалить Peel из таблицы Продавцов, команда будет не допустима, пока вы не измените значение поля snum заказчиков Hoffman и Clemens для другого назначенного продавца. С другой стороны, вы можете изменить значение поля snum для Peel на 1009, и Hoffman и Clemens будут также автоматически изменены.

Третий эффект — *Пустые* (**NULL**) изменения. Бывает, что когда продавцы оставляют компанию, их текущие порядки не передаются другому продавцу. С другой стороны, вы хотите отменить все порядки автоматически для заказчиков, чьи счета вы удалите. Изменив номера продавца или заказчика, можно просто передать их ему. Пример ниже показывает, как вы можете создать таблицу Порядков с использованием этих эффектов.

```
CREATE TABLE Orders
(onum  integer NOT NULL PRIMARY KEY,
 amt   decimal,
 odate date NOT NULL
 cnum  integer NOT NULL REFERENCES Customers
 snum  integer REFERENCES Salespeople,
 UPDATE OF Customers CASCADES,
 DELETE OF Customers CASCADES,
 UPDATE OF Salespeople CASCADES,
 DELETE OF Salespeople NULLS);
```

Конечно, в команде **DELETE** с эффектом *Пустого* изменения в таблице Продавцов, ограничение NOT NULL должно быть удалено из поля snum.

ВНЕШНИЕ КЛЮЧИ, КОТОРЫЕ ССЫЛАЮТСЯ ОБРАТНО К ИХ ПОДЧИНЕНЫМ ТАБЛИЦАМ

Как было упомянуто ранее, ограничение FOREIGN KEY может представить имя этой частной таблице, как таблицы родительского ключа. Далеко не будучи простой, эта особенность может пригодиться. Предположим, что мы имеем таблицу Employees с полем manager (администратор). Это поле содержит номера каждого из служащих, некоторые из которых являются еще и администраторами.

Но так как каждый администратор — в то же время остается служащим, то он естественно будут также представлен в этой таблице. Давайте создадим таблицу, где номер служащего (столбец с именем empno), объявляется как первичный ключ, а администратор, как внешний ключ, будет ссылаться на нее:

```
CREATE TABLE Employees
(empno   integer NOT NULL PRIMARY KEY,
 name    char(10) NOT NULL UNIOUE,
 manager integer REFERENCES Employees);
```

(Так как внешний ключ это ссылаемый первичный ключ таблицы, список столбцов может быть исключен.) Имеется содержание этой таблицы:

EMPNO	NAME	MANAGER
1003	Terrence	2007
2007	Atali	NULL
1688	McKenna	1003
2002	Collier	2007

Как вы можете видеть, каждый из них (но не Atali), ссылается на другого служащего в таблице как на своего администратора. Atali, имеющий наивысший номер в таблице, должен иметь значение установленное в NULL. Это дает другой принцип справочной целостности. Внешний ключ, который ссылается обратно к частной таблице, должен позволять значения = NULL. Если это не так, как бы вы могли вставить первую строку ?

Даже если эта первая строка ссылается к себе самой, значение родительского ключа должно уже быть установлено, когда вводится значение внешнего ключа. Этот принцип будет верен, даже если внешний ключ ссылается обратно к частной таблице не напрямую, а с помощью ссылки к другой таблице, которая затем ссылается обратно к таблице внешнего ключа. Например, предположим, что наша таблица Продавцов имеет дополнительное поле, которое ссылается на таблицу Заказчиков, так, что каждая таблица ссылается на другую, как показано в следующем операторе CREATE TABLE:

```
CREATE TABLE Salespeople
(snum  integer NOT NULL PRIMARY KEY,
 sname char(10) NOT NULL,
 city  char(10),
 comm  declmal,
 cnum  integer REFERENCES Customers);

CREATE TABLE Customers
(cnum   integer NOT NULL PRIMARY KEY,
 cname  char(10) NOT NULL,
 city   char(10),
 rating integer,
 snum   integer REFERENCES Salespeople);
```

Это называется — *перекрестной* ссылкой.

SQL поддерживает это теоретически, но практически это может составить проблему. Любая таблица из этих двух, созданная первой является ссылочной таблицей которая еще не существует для другой. В интересах обеспечения перекрестной ссылки, SQL фактически позволяет это, но никакая таблица не будет пригодна для использования, пока они обе находятся в процессе создания. С другой стороны, если эти две таблицы создаются различными пользователями, проблема становится еще более трудной. Перекрестная ссылка может стать полезным инструментом, но она не без неоднозначности и опасностей. Предшествующий пример, например, не совсем пригоден для использования: потому что он ограничивает продавца одиночным заказчиком, и кроме того совсем необязательно использовать перекресную ссылку, чтобы достичь этого. Мы рекомендуем чтобы вы были осторожны в его использовании и анализировали, как ваши программы управляют эффектами модификации и удаления, а также процессами привилегий и диалоговой обработки запросов перед тем, как вы создаете перекресную систему справочной целостности. (Привилегии и диалоговая обработка запросов будут обсуждаться, соответственно, в Главах 22 и 23.)

РЕЗЮМЕ

Теперь вы имеете достаточно хороше управление справочной целостностью. Основная идея в том, что все значения внешнего ключа ссылаются к указанной строке родительского ключа. Это означает, что каждое значение внешнего ключа должно быть представлено один раз, и только один раз, в родительском ключе. Всякий раз, когда значение помещается во внешний ключ, родительский ключ проверяется, чтобы удостовериться, что его значение представлено; иначе, команда будет отклонена. Родительский ключ должен иметь Первичный Ключ (**PRIMARY KEY**) или Уникальное (**UNIQUE**) ограничение, гарантирующее, что значение не будет представлено более чем один раз. Попытка изменить значение родительского ключа, которое в настоящее время представлено во внешнем ключе, будет вообще отклонена. Ваша система может, однако, предложить вам выбор, чтобы получить значение внешнего ключа установленого в NULL или для получения нового значения родителького ключа, и указания какой из них может быть получен независимо для команд **UPDATE** и **DELETE**.

Этим завершается наше обсуждение команды **CREATE TABLE**. Далее мы представим вас другому типу команды — **CREATE**. В Главе 20 вы обучитесь представлению объектов данных, которые выглядят и действуют подобно таблице, но в действительности являются результатами запросов. Некоторые функции ограничений могут также выполняться представлениями, так что вы сможете лучше оценить вашу потребность к ограничениям, после того, как вы прочитаете следующие три главы.

1. Создайте таблицу с именем Cityorders. Она должна содержать такие же поля onum, amt, и snum что и таблица Порядков, и такие же поля cnum и city что и таблица Заказчиков, так что порядок каждого заказчика будет вводиться в эту таблицу вместе с его городом. Поле onum будет первичным ключом Cityorders. Все поля в Cityorders должны иметь ограничения при сравнении с таблицами Заказчиков и Порядков. Допускается, что родительские ключи в этих таблицах уже имеют соответствующие ограничения.
2. Усложним проблему. Переопределите таблицу Порядков следующим образом: добавьте новый столбец с именем prev, который будет идентифицирован для каждого порядка, поле onum предыдущего порядка для этого текущего заказчика. Выполните это с использованием внешнего ключа ссылающегося на саму таблицу Порядков. Внешний ключ должен ссылаться также на поле cnum заказчика, обеспечивающего определенную предписанную связь между текущим порядком и ссылаемым.

(См. Приложение A для ответов.)

20

ВВЕДЕНИЕ: ПРЕДСТАВЛЕНИЯ

ПРЕДСТАВЛЕНИЕ **(VIEW**) — ОБЪЕКТ ДАННЫХ, КОТОРЫЙ не содержит никаких данных его владельца. Это — тип таблицы, чье содержание выбирается из других таблиц с помощью выполнения запроса. Поскольку значения в этих таблицах меняются, то автоматически, их значения могут быть показаны представлением.

В этой главе вы узнаете, что такое представления, как они создаются и немного об их возможностях и ограничениях. Использование представлений, основанных на улучшеных средствах запросов, таких как обьединение и подзапрос, разработаных очень тщательно, в некоторых случаях даст больший выиигрыш по сравнениию с запросами.

ЧТО ТАКОЕ ПРЕДСТАВЛЕНИЕ?

Типы таблиц, с которыми вы имели дело до сих пор, назывались — *базовыми таблицами*. Это — таблицы, которые содержат данные. Однако имеется другой вид таблиц — представления. *Представления* — это таблицы, чье содержание выбирается или получается из других таблиц. Они работают в запросах и операторах DML точно также как и основные таблицы, но не содержат никаких собственных данных.

Представления подобны окнам, через которые вы просматриваете информацию (как она есть, или в другой форме, как вы потом увидите), которая фактически хранится в базовой таблице. *Представление* — это фактически запрос, который выполняется всякий раз, когда представление становится темой команды. Вывод запроса при этом в каждый момент становится содержанием представления.

КОМАНДА CREATE VIEW

Вы создаете представление командой **CREATE VIEW.** Она состоит из слов **CREATE VIEW** (*СОЗДАТЬ ПРЕДСТАВЛЕНИЕ*), *имени представления*, которое нужно создать, слова **AS** (*КАК*), и далее *запроса*, как в следующем примере:

```
CREATE VIEW Londonstaff
AS SELECT *
   FROM Salespeople
   WHERE city = 'London';
```

Теперь Вы имеете представление, называемое **Londonstaff**. Вы можете использовать это представление точно так же, как и любую другую таблицу. Она может быть запрошена, модифицирована, вставлена в, удалена из, и соединена с, другими таблицами и представлениями. Давайте сделаем запрос такого представления (вывод показан в Рисунке 20.1):

```
SELECT *
FROM Londonstaff;
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Londonstaff;                             |
| ===============================================|
|   snum      sname         city         comm    |
| ------    ----------   -----------   -------   |
|   1001      Peel         London       0.1200   |
|   1004      Motika       London       0.1100   |
 ================================================
```

Рисунок 20.1: Представление Londonstaff

Когда вы приказываете SQL выбрать (**SELECT**) все строки (*****) из представления, он выполняет запрос содержащий в определении — Loncfonstaff, и возвращает все из его вывода.

Имея предикат в запросе представления, можно вывести только те строки из представления, которые будут удовлетворять этому предикату. Вы могли бы вспомнить, что в Главе 15, вы имели таблицу, называемую **Londonstaff**, в которую вы вставляли это же самое содержание (конечно, мы понимаем, что таблица — не слишком велика. Если это так, вы будете должны выбрать другое имя для вашего представления). Преимущество использования представления, по сравнению с основной таблицы, в том, что представление будет модифицировано автоматически всякий раз, когда таблица, лежащая в его основе изменяется.

Содержание представления не фиксировано, и переназначается каждый раз когда вы ссылаетесь на представление в команде. Если вы добавите завтра другого, живущего в Лондоне продавца, он автоматически появится в представлении.

Представления значительно расширяют управление вашими данными. Это — превосходный способ дать публичный доступ к некоторой, но не всей информации в таблице. Если вы хотите, чтобы ваш продавец был показан в таблице Продавцов, но при этом не были показаны комиссии других продавцов, вы могли бы создать представление с использованием следующего оператора (вывод показан в Рисунке 20.2)

```
CREATE VIEW Salesown
AS SELECT snum, sname, city
   FROM Salespeople:
```

```
 ===============  SQL Execution Log ============
| SELECT *                                       |
| FROM  Salesown;                                |
| ===============================================|
|   snum      sname         city                 |
| ------    ----------   -----------             |
|   1001      Peel         London                |
|   1002      Serres       San Jose              |
|   1004      Motika       London                |
|   1007      Rifkin       Barcelona             |
|   1003      Axelrod      New York              |
 ================================================
```

Рисунок 20.2: Представление Salesown

Другими словами, это представление — такое же, как для таблицы Продавцов, за исключением того, что поле comm, не упоминалось в запросе, и следовательно не было включено в представление.

МОДИФИЦИРОВАНИЕ ПРЕДСТАВЛЕНИЙ

Представление может теперь изменяться командами модификации DML, но модификация не будет воздействовать на само представление. Команды будут на самом деле перенаправлены к базовой таблице:

```
UPDATE Salesown
SET city = 'Palo Alto'
WHERE snum = 1004;
```

Его действие идентично выполнению той же команды в таблице Продавцов. Однако, если значение комиссионных продавца будет обработано командой UPDATE

```
UPDATE Salesown
SET comm = .20
WHERE snum = 1004;
```

она будет отвергнута, так как поле comm отсутствует в представлении Salesown. Это важное замечание, показывающее что не все представления могут быть модифицированы.

Мы будем исследовать проблемы модификации представлений в Главе 21.

ИМЕНОВАНИЕ СТОЛБЦОВ

В нашем примере, поля наших представлений имеют свои имена, полученые прямо из имен полей основной таблицы. Это удобно. Однако, иногда вам нужно снабжать ваши столбцы новыми именами:

* когда некоторые столбцы являются выводимыми, и проэтому не имеющими имен.

* когда два или более столбцов в объединении имеют те же имена, что в их базовой таблице.

Имена, которые могут стать именами полей, даются в круглых скобках **()**, после имени таблиц. Они не будут запрошены, если совпадают с именами полей запрашиваемой таблицы. Тип данных и размер этих полей будут отличаются от запрашиваемых полей которые "передаются" в них. Обычно вы не указываете новых имен полей, но если вы все таки сделали это, вы должны делать это для каждого поля в представлении.

КОМБИНИРОВАНИЕ ПРЕДИКАТОВ ПРЕДСТАВЛЕНИЙ И ОСНОВНЫХ ЗАПРОСОВ В ПРЕДСТАВЛЕНИЯХ

Когда вы делаете запрос представления, вы собственно, запрашиваете запрос. Основной способ для SQL обойти это, — объединить предикаты двух запросов в один. Давайте посмотрим еще раз на наше представление с именем Londonstaff:

```
CREATE VIEW Londonstaff
AS SELECT *
FROM Salespeople
WHERE city = 'London';
```

Если мы выполняем следующий запрос в этом представлении

```
SELECT *
FROM Londonstaff
WHERE comm > .12;
```

он такой же, как если бы мы выполнили следующее в таблице Продавцов:

```
SELECT *
FROM Salespeople
WHERE city = 'London' AND comm > .12;
```

Это прекрасно, за исключением того, что появляется возможная проблема с представлением. Имеется возможность комбинации из двух полностью допустимых предикатов и получения предиката который не будет работать. Например, предположим, что мы создаем (**CREATE**) следующее представление:

```
CREATE VIEW Ratingcount (rating, number)
AS SELECT rating, COUNT (*)
   FROM Customers
   GROUP BY rating;
```

Это дает нам число заказчиков, которые мы имеем для каждого уровня оценки (rating). Вы можете затем сделать запрос этого представления чтобы выяснить, имеется ли какая-нибудь оценка, в настоящее время назначенная для трех заказчиков:

```
SELECT *
FROM Ratingcount
WHERE number = 3;
```

Посмотрим, что случится, если мы скомбинируем два предиката:

```
SELECT rating, COUNT (*)
FROM Customers
WHERE COUNT (*) = 3
GROUP BY rating;
```

Это недопустимый запрос. Агрегатные функции, такие как COUNT (*СЧЕТ*), не могут использоваться в предикате. Првильным способом при формировании вышеупомянутого запроса, конечно же будет следующий:

```
SELECT rating, COUNT (*)
FROM Customers
GROUP BY rating;
HAVING COUNT (*) = 3;
```

Но SQL может не выполнить превращения. Может ли равноценный запрос вместо запроса Ratingcount потерпеть неудачу? Да может! Это — неоднозначная область SQL, где методика использования представлений может дать хорошие результаты.

Самое лучшее что можно сделать в случае, когда об этом ничего не сказано в вашей системной документации, так это попытка в ней разобраться. Если команда допустима, вы можете использовать представления чтобы установить некоторые ограничения SQL в синтаксисе запроса.

ГРУППОВЫЕ ПРЕДСТАВЛЕНИЯ

Групповные представления — это представления, наподобии запроса **Ratingcount** в предыдущем примере, который содержит предложение GROUP BY, или который основывается на других групповных представлениях.

Групповые представления могут стать превосходным способом обрабатывать полученную информацию непрерывно. Предположим, что каждый день вы должны следить за порядком номеров заказчиков, номерами продавцов принимающих порядки, номерами порядков, средним от порядков, и общей суммой приобретений в порядках.

Чем конструировать каждый раз сложный запрос, вы можете просто создать следующее представление:

```
CREATE VIEW Totalforday
AS SELECT odate, COUNT(DISTINCT cnum), COUNT(DISTINCT snum),
          COUNT(onum), AVG(amt), SUM(amt)
FROM Orders
GROUP BY odate;
```

Теперь вы сможете увидеть всю эту информацию с помощью простого запроса:

```
SELECT *
FROM Totalforday;
```

Как мы видели, SQL запросы могут дать вам полный комплекс возможностей, так что представления обеспечивают вас чрезвычайно гибким и мощным инструментом чтобы определить точно, как ваши данные могут быть использованы.

Они могут также делать вашу работу более простой, переформатируя данные удобным для вас способом и исключив двойную работу.

ПРЕДСТАВЛЕНИЯ И ОБЬЕДИНЕНИЯ

Представления не требуют, чтобы их вывод осуществлялся из одной базовой таблицы. Так как почти любой допустимый запрос SQL может быть использован в представлении, он может выводить информацию из любого числа базовых таблиц, или из других представлений.

Мы можем, например, создать представление, которое показывало бы порядки продавца и заказчика по имени:

```
CREATE VIEW Nameorders
AS SELECT onum, amt, a.snum, sname, cname
   FROM Orders a, Customers b, Salespeople c
   WHERE a.cnum = b.cnum AND a.snum = c.snum;
```

Теперь вы можете выбрать (**SELECT**) все порядки заказчика или продавца (*****), или можете увидеть эту информацию для любого порядка.

Например, чтобы увидеть все порядки продавца Rifkin, вы должны ввести следующий запрос (вывод показан в 20.3 Рисунке):

```
SELECT *
FROM Nameorders
WHERE sname = 'Rifkin';
```

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM  Nameorders                                |
| WHERE sname = 'Rifkin';                         |
| ================================================ |
|   onum       amt       snum   sname     cname   |
|  ------   --------    -----  -------   -------  |
|   3001       18.69     1007  Rifkin    Cisneros |
|   3006     1098.16     1007  Rifkin    Cisneros |
 ================================================
```

Рисунок 20.3: Порядки Rifkin показаные в Nameorders

Вы можете также объединять представления с другими таблицами, или базовыми таблицами или представлениями, поэтому вы можете увидеть все порядки Axelroda и значения его комиссиионных в каждом порядке:

```
SELECT a.sname, cname, amt  comm
FROM Nameorders a, Salespeople b
WHERE a.sname = 'Axelrod' AND b.snum = a.snum;
```

Вывод для этого запроса показывается в Рисунке 20.4.

В предикате, мы могли бы написать — "WHERE a.sname = 'Axelrod' AND b.sname = 'Axelrod'", но предикат, который мы использовали здесь, более общеупотребительный. Кроме того, поле snum — это первичный ключ таблицы Продавцов и, следовательно, должен по определению быть уникальным.

```
 ===============  SQL Execution Log ==============
| SELECT a.sname, cname, amt * comm                |
| FROM  Nameorders a, Salespeople b                |
| WHERE a.sname = 'Axelrod' AND b.snum = a.snum;   |
| ================================================ |
|   onum       amt       snum   sname     cname   |
|  ------   --------    -----  -------   -------  |
|   3001       18.69     1007  Rifkin    Cisneros |
|   3006     1098.16     1007  Rifkin    Cisneros |
 ================================================
```

Рисунок 20.4: Обьединение основной таблицы с представлением

Если бы там, например, было два Axelrodf, вариант с именем будет объединять вместе их данные. Более предпочтительный вариант — использовать поле snum, чтобы хранить его отдельно.

ПРЕДСТАВЛЕНИЯ И ПОДЗАПРОСЫ

Представления могут также использовать и подзапросы, включая соотнесенные подзапросы. Предположим, ваша компания предусматривает премию для тех продавцов, которые имеют заказчика с самым высоким порядком для любой указанной даты. Вы можете проследить эту информацию с помощью представления:

```
CREATE VIEW Elitesalesforce
AS SELECT b.odate, a.snum, a.sname,
   FROM Salespeople a, Orders b
   WHERE a.snum = b.snum AND b.amt = (SELECT MAX (amt)
                                      FROM Orders c
                                      WHERE c.odate = b.odate);
```

Если, с другой стороны, премия будет назначаться только продавцу, который имел самый высокий порядок за последние десять лет, вам необходимо будет проследить их в другом представлении, основанном на первом:

```
CREATE VIEW Bonus
AS SELECT DISTINCT snum, sname
   FROM Elitesalesforce a
   WHERE 10 <= (SELECT COUNT (*)
                FROM Elitesalestorce b
                WHERE a.snum = b.snum);
```

Извлечение из этой таблицы продавца, который будет получать премию — выполняется простым запросом:

```
SELECT *
FROM Bonus;
```

Теперь мы видим истинную мощность SQL. Извлечение той же полученной информации программами RPG или COBOL будет более длительной процедурой. В SQL, это — только вопрос из двух комплексных команд, сохраненных, как представление совместно с простым запросом.

При самостоятельном запросе — мы должны заботится об этом каждый день, потому что информация которую извлекает запрос, непрерывно меняется чтобы отражать текущее состояние базы данных.

ЧТО НЕ МОГУТ ДЕЛАТЬ ПРЕДСТАВЛЕНИЯ

Имеются большое количество типов представлений (включая многие из наших примеров в этой главе) которые являются доступными только для чтения. Это означает, что их можно запрашивать, но они не могут подвергаться действиям команд модификации. (Мы будем рассматривать эту тему в Главе 21.)

Имеются также некоторые виды запросов, которые не допустимы в определениях представлений. Одиночное представление должно основываться на одиночном запросе; *ОБЪЕДИНЕНИЕ* (**UNION**) и *ОБЪЕДИНЕНИЕ ВСЕГО* (**UNION ALL**) не разрешаются. УПОРЯДОЧЕНИЕ ПО (**ORDER BY**) никогда не используется в определении представлений. Вывод запроса формирует содержание представления, которое напоминает базовую таблицу и является — по определению — неупорядоченным.

УДАЛЕНИЕ ПРЕДСТАВЛЕНИЙ

Синтаксис удаления представления из базы данных подобен синтаксису удаления базовых таблиц:

```
DROP VIEW <view name>
```

В этом нет необходимости, однако, сначала надо удалить все содержание, как это делается с базовой таблицей, потому что содержание представления не является созданым и сохраняется в течении определеной команды. Базовая таблица, из которой представление выводится, не эффективна, когда представление удалено.

Помните, вы должны являться владельцем представления, чтобы иметь возможность удалить его.

РЕЗЮМЕ

Теперь, когда вы можете использовать представления, ваша способность отслеживать и обрабатывать содержание вашей базы данных, значительно расширилась. Любые вещи которые вы можете создать с запросом, вы всегда сможете определить как представление. Запросы этих представлений, фактически, запрос запроса.

Использование представлений и для удобства, и для защиты также удобно, как и многие возможности представлений для форматирования и получения значений из постоянно меняющегося содержания вашей базы данных. Имеется один главный вывод относительно представлений — это способность к модификации, которую мы выбрали в отличии от Главы 21. Как показано, вы можете модифицировать представления также как и базовую таблицу, с помощью изменений, применяемых к таблице, из которой получается представление, но это не всегда возможно.

РАБОТА С SQL

1. Создайте представление которое бы показывало всех заказчиков которые имеют самые высокие оценки.
2. Создайте представление которое бы показывало номер продавца в каждом городе.
3. Создайте представление которое бы показывало усредненый и общий порядки для каждого продавца после его имени. Предполагается, что все имена — уникальны.
4. Создайте представление которое бы показывало каждого продавца с многочисленными заказчиками.

(См. Приложение A для ответов.)

21

ИЗМЕНЕНИЕ ЗНАЧЕНИЙ С ПОМОЩЬЮ ПРЕДСТАВЛЕНИЙ

ЭТА ГЛАВА РАССКАЗЫВАЕТ О КОМАНДАХ МОДИФИКАЦИИ ЯЗЫКА DML — *ВСТАВИТЬ* (**INSERT**), *ИЗМЕНИТЬ* (**UPDATE**), и *УДАЛИТЬ* (**DELETE**) — когда они применяются для представлений. Как упомянуто в предыдущей главе, использование команд модификации в представлениях — это косвенный способ использования их в ссылочных таблицах с помощью запросов представлений. Однако, не все представления могут модифицироваться.

В этой главе, мы будем обсуждать правила определяющие, является ли представление модифицируемым. Кроме того, вы обучитесь использованию предложения **WITH CHECK OPTION**, которое управляет указанными значениями, которые можно вводить в таблицу с помощью представления.

Как упомянуто в Главе 18, это, в некоторых случаях, может быть желательным вариантом непосредственного ограничения таблицы.

МОДИФИЦИРОВАНИЕ ПРЕДСТАВЛЕНИЯ

Один из наиболее трудных и неоднозначных аспектов представлений — непосредственное их использование с командами модификации DML. Как упомянуто в предыдущей главе, эти команды фактически воздействуют на значения в базовой таблице представления.

Это является некоторым противоречием. Представление состоит из результатов запроса, и когда вы модифицируете представление, вы модифицируете набор результатов запроса. Но модификация не должна воздействовать на запрос; она должна воздействовать на значения в таблице, к которой был сделан запрос, и таким образом изменять вывод запроса. Это не простой вопрос. Следующий оператор будет создавать представление, показанное на Рисунке 21.1:

```
CREATE VIEW Citymatch (custcity, salescity)
AS SELECT DISTINCT a.city, b.city
   FROM Customers a, Salespeople b
   WHERE a.snum = b.snum;
```

Это представление показывает все совпадения заказчиков с их продавцами так, что имеется по крайней мере один заказчик в **городе_заказчика**, обслуживаемый продавцом в **городе_продавца**.

Например, одна строка этой таблицы — London London — показывает, что имеется по крайней мере один заказчик в Лондоне, обслуживаемый продавцом в Лондоне. Эта строка может быть произведена при совпадении Hoffmana с его продавцом Peel, причем если оба они из Лондона.

```
 ===============  SQL Execution Log ==============
| SELECT *                                        |
| FROM Citymatch;                                 |
| ============================================== |
|   custcity    salescity                         |
|  ---------    ---------                         |
|  Berlin       San Jose                          |
|  London       London                            |
|  Rome         London                            |
|  Rome         New York                          |
|  San Jose     Barselona                         |
|  San Jose     San Jose                          |
 ================================================
```

Рисунок 21.1: Представление совпадения по городам

Однако, то же самое значение будет произведено при совпадении Clemens из Лондона, с его продавцом, который также оказался с именем — Peel. Пока отличающиеся комбинации городов выбирались конкретно, только одна строка из этих значений была произведена.

Даже если вы не получите выбора, используя отличия, вы все еще будете в том же самом положении, потому что вы будете тогда иметь две строки в представлении с идентичными значениями, то-есть с обоими столбцами равными "London London". Эти две строки представления будут отличаться друг от друга, так что вы пока не сможете сообщить, какая строка представления исходила из каких значений базовых таблиц (имейте в виду, что запросы, не использующие предложение ORDER BY, производят вывод в произвольном порядке. Это относится также и к запросам, используемым внутри представлений, которые не могут использовать ORDER BY. Таким образом, порядок из двух строк не может быть использован для их отличий. Это означает, что мы будем снова обращаться к выводу строк, которые не могут быть точно связаны с указаными строками запрашиваемой таблицы. Что если вы пробуете удалить строку ”London London” из представления? Означало бы это удаление Hoffmana из таблицы Заказчиков, удаление Clemens из той же таблицы, или удаление их обоих? Должен ли SQL также удалить Peel из таблицы Продавцов? На эти вопросы невозможно ответить точно, поэтому удаления не разрешены в представлениях такого типа. Представление Citymatch — это пример представления *только_чтение*, оно может быть только запрошено, но не изменено.

ОПРЕДЕЛЕНИЕ МОДИФИЦИРУЕМОСТИ ПРЕДСТАВЛЕНИЯ

Если команды модификации могут выполняться в представлении, представление как сообщалось будет *модифицируемым*; в противном случае оно предназначено только для чтения при запросе. Непротивореча этой терминологии, мы будем использовать выражение "*модифицируемое представление*" (updating a view), чтоы означает возможность выполнения в представление любой из трех команд модификации DML (*Вставить, Изменить* и *Удалить*), которые могут изменять значения.

Как вы определите, является ли представление модифицируемым? В теории базы данных, это — пока обсуждаемая тема. Основной ее принцип такой: *модифицирумое представление* — это представление, в котором команда модификации может выполниться, чтобы изменить одну и только одну строку основной таблицы в каждый момент времени, не воздействуя на любые другие строки любой таблицы. Использование этого принципа на практике, однако, затруднено. Кроме того, некоторые представления, которые являются модифицируемыми в теории, на самом деле не являются модифицируемыми в SQL. Критерии по которым определяют, является ли представление модифицируемым или нет, в SQL, следующие:

* Оно должно выводиться в одну и только в одну базовую таблицу.

* Оно должно содержать первичный ключ этой таблицы (это технически не предписывается стандартом ANSI, но было бы неплохо придерживаться этого).

* Оно не должно иметь никаких полей, которые бы являлись агрегатными функциями.

* Оно не должно содержать DISTINCT в своем определении.

* Оно не должно использовать GROUP BY или HAVING в своем определении.

* Оно не должно использовать подзапросы (это — ANSI ограничение, которое не предписано для некоторых реализаций)

* Оно может быть использовано в другом представлении, но это представление должно также быть модифицируемыми.

* Оно не должно использовать константы, строки, или выражения значений (например: comm * 100) среди выбранных полей вывода.

* Для INSERT, оно должно содержать любые поля основной таблицы, которые имеют ограничение NOT NULL, если другое ограничение по умолчанию не определено.

МОДИФИЦИРУЕМЫЕ ПРЕДСТАВЛЕНИЯ И ПРЕДСТАВЛЕНИЯ ТОЛЬКО_ЧТЕНИЕ

Одно из этих ограничений то, что модифицируемые представления, фактически, подобны окнам в базовых таблицах. Они показывают кое-что, но не обязательно все, из содержимого таблицы. Они могут ограничивать определенные строки (использованием предикатов), или специально именованные столбцы (с исключениями), но они представляют значения непосредственно и не выводит их информацию, с использованием составных функций и выражений.

Они также не сравнивают строки таблиц друг с другом (как в обьединениях и подзапросах, или как с DISTINCT).

Различия между модифицируемыми представлениями и представлениями только_чтение неслучайны.

Цели, для которых вы их используете, часто различны. Модифицируемые представления, в основном, используются точно так же как и базовые таблицы. Фактически, пользователи не могут даже осознать, является ли объект который они запрашивают, базовой таблицей или представлением. Это превосходный механизм защиты для сокрытия частей таблицы, которые являются конфиденциальными или не относятся к потребностям данного пользователя. (В Главе 22, мы покажем вам, как позволить пользователям обращаться к представлению, а не к базовой таблице).

Представления только_чтение, с другой стороны, позволяют вам получать и переформатировать данные более рационально. Они дают вам библиотеку сложных запросов, которые вы можете выполнить и повторить снова, сохраняя полученную вами информацию до последней минуты. Кроме того, результаты этих запросов в таблицах, которые могут затем использоваться в запросах самостоятельно (например, в обьединениях) имеют преимущество над просто выполнением запросов.

Представления только_чтение могут также иметь прикладные программы защиты. Например, вы можете захотеть, чтобы некоторые пользователи видели агрегатные данные, такие как усредненное значение комиссионных продавца, не видя индивидуальных значений комиссионных.

ЧТО ЯВЛЯЕТСЯ МОДИФИЦИРУЕМЫМ ПРЕДСТАВЛЕНИЕМ

Имеются некоторые примеры модифицируемых представлений и представлений только_чтение:

```
CREATE VIEW Dateorders (odate, ocount)
AS SELECT odate, COUNT (*)
   FROM Orders
   GROUP BY odate;
```

Это — представление только_чтение из-за присутствия в нем агрегатной функции и GROUP BY.

```
CREATE VIEW Londoncust
AS SELECT *
   FROM Customers
   WHERE city = 'London';
```

А это — представление модифицируемое.

```
CREATE VIEW SJsales (name, number, percentage)
AS SELECT sname, snum, comm 100
   FROM Salespeople
   WHERE city = 'SanJose';
```

Это — представление только_чтение из-за выражения "**comm * 100**". При этом, однако, возможны переупорядочение и переименование полей. Некоторые программы будут позволять удаление в этом представлении или в порядках столбцов snum и sname.

```
CREATE VIEW Salesonthird
AS SELECT *
   FROM Salespeople
   WHERE snum IN (SELECT snum
                  FROM Orders
                  WHERE odate = 10/03/1990);
```

Это — представление только_чтение в ANSI из-за присутствия в нем подзапроса. В некоторых программах, это может быть приемлемо.

```
CREATE VIEW Someorders
AS SELECT snum, onum, cnum
   FROM Orders
   WHERE odate IN (10/03/1990,10/05/1990);
```

Это — модифицируемое представление.

ПРОВЕРКА ЗНАЧЕНИЙ, ПОМЕЩАЕМЫХ В ПРЕДСТАВЛЕНИЕ

Другой вывод о модифицируемости представления тот, что вы можете вводить значения которые "*проглатываются*" (swallowed) в базовой таблице. Рассмотрим такое представление:

```
CREATE VIEW Highratings
AS SELECT cnum, rating
   FROM Customers
   WHERE rating = 300;
```

Это — представление модифицируемое. Оно просто ограничивает ваш доступ к определенным строкам и столбцам в таблице. Предположим, что вы *вставляете* (**INSERT**) следующую строку:

```
INSERT INTO Highratings
VALUES (2018, 200);
```

Это — допустимая команда INSERT в этом представлении. Строка будет вставлена, с помощью представления Highratings, в таблицу Заказчиков. Однако когда она появится там, она исчезнет из представления, поскольку значение оценки не равно 300. Это — обычная проблема.

Значение 200 может быть просто напечатано, но теперь строка находится уже в таблице Заказчиков где вы не можете даже увидеть ее. Пользователь не сможет понять, почему введя строку он не может ее увидеть, и будет неспособен при этом удалить ее.

Вы можете быть гарантированы от модификаций такого типа с помощью включения **WITH CHECK OPTION** (*С ОПЦИЕЙ ПРОВЕРКИ*) в определение представления. Мы можем использовать WITH CHECK OPTION в определении представления Highratmgs.

```
CREATE VIEW Highratings
AS SELECT cnum, rating
   FROM Customers
   WHERE rating = 300
   WITH CHECK OPTION;
```

Вышеупомянутая вставка будет отклонена.

WITH CHECK OPTION — производит действие *все_или_ничего* (all-or-nothing). Вы помещаете его в определение представления, а не в команду DML, так что или все команды модификации в представлении будут проверяться, или ни одна не будет проверена. Обычно вы хотите использовать опцию проверки, используя ее в определении представления, что может быть удобно.

В общем, вы должны использовать эту опцию, если у вас нет причины, разрешать представлению помещать в таблицу значения, которые он сам не может содержать.

ПРЕДИКАТЫ И ИСКЛЮЧЕННЫЕ ПОЛЯ

Похожая проблема, которую вы должны знать, включает в себя вставку строк в представление с предикатом, базирующемся на одном или более исключенных полей. Например, может показаться разумным, чтобы создать **Londonstaff** подобно этому:

```
CREATE VIEW Londonstaff
AS SELECT snum, sname, comm
   FROM Salespeople
   WHERE city = 'London';
```

В конце концов, зачем включать значение city, если все значения city будут одинаковыми.

А как будет выглядит картинка, получаемая всякий раз, когда мы пробуем вставить строку. Так как мы не можем указать значение city как значение по умолчанию, этим значением вероятно будет NULL, и оно будет введено в поле city (NULL используется если другое значение по умолчанию значение не было определено. См. Главу 18 для подробностей). Так как в этм случае поле city не будет равняться значению London, вставляемая строка будет исключена из представления. Это будет верным

для любой строки которую вы попробуете вставить в просмотр **Londonstaff**. Все они должны быть введены с помощью представления Londonstaff в таблицу Продавцов, и затем исключены из самого представления (если определением по умолчанию был не London, то это особый случай). Пользователь не сможет вводить строки в это представление, хотя все еще неизвестно, может ли он вводить строки в базовую таблицу. Даже если мы добавим WITH CHECK OPTION в определение представления

```
CREATE VIEW Londonstate
AS SELECT snum, sname, comm
   FROM Salespeople
   WHERE city = 'London'
   WITH CHECK OPTION;
```

проблема не обязательно будет решена. В результате этого мы получим представление, которое мы могли бы модифицировать или из которого мы могли бы удалять, но не вставлять в него. В некоторых случаях, это может быть хорошо; хотя возможно нет смысла пользователям, имеющим доступ к этому представлению иметь возможность добавлять строки. Но вы должны точно определить, что может произойти прежде, чем вы создадите такое представление.

Даже если это не всегда может обеспечить Вас полезной информацией, полезно включать в ваше представление все поля, на которые имеется ссылка в предикате. Если вы не хотите видеть эти поля в вашем выводе, вы всегда сможете исключить их из запроса в представлении, в противоположность запросу внутри представления. Другими словами, вы могли бы определить представление **Londonstaff** подобно этому:

```
CREATE VIEW Londonstaff
AS SELECT *
   FROM Salespeople
   WHERE city = 'London'
   WITH CHECK OPTION;
```

Эта команда заполнит представление одинаковыми значениями в поле city, которые вы можете просто исключить из вывода с помощью запроса в котором указаны только те поля которые вы хотите видеть

```
SELECT snum, sname, comm
FROM Londonstaff;
```

ПРОВЕРКА ПРЕДСТАВЛЕНИЙ, КОТОРЫЕ БАЗИРУЮТСЯ НА ДРУГИХ ПРЕДСТАВЛЕНИЯХ

Еще одно надо упоминуть относительно предложения WITH CHECK OPTION в ANSI: оно не делает *каскадированного* изменения: Оно применяется только в представлениях в которых оно определено, но не в представлениях основанных на этом представлении. Например, в предыдущем примере

```
CREATE VIEW Highratings
AS SELECT cnum, rating
   FROM Customers
   WHERE rating = 300
   WITH CHECK OPTION;
```

попытка вставить или модифицировать значение оценки не равное 300 потерпит неудачу. Однако, мы можем создать второе представление (с идентичным содержанием) основанное на первом:

```
CREATE VIEW Myratings
AS SELECT *
FROM Highratings;
```

Теперь мы можем модифицировать оценки, не равные 300:

```
UPDATE Myratings
SET rating = 200
WHERE cnum = 2004;
```

Эта команда, выполняемая так, как если бы она выполнялась как первое представление, будет допустима. Предложение WITH CHECK OPTION просто гарантирует, что любая модификация в представлении произведет значения, которые удовлетворяют предикату этого представления. Модификация других представлений, базирующихся на первом текущем, является все еще допустимой, если эти представления не защищены предложениями WITH CHECK OPTION внутри этих представлений. Даже если такие предложения установлены, они проверяют только те предикаты представлений, в которых они содержатся. Так например, даже если представление **Myratings** создавалось следующим образом

```
CREATE VIEW Myratings
AS SELECT *
   FROM Highratings
   WITH CHECK OPTION;
```

проблема не будет решена. Предложение WITH CHECK OPTION будет исследовать только предикат представления Myratings. Пока у Myratings, фактически, не имеется никакого предиката, WITH CHECK OPTION ничего не будет делать. Если используется предикат, то он будет проверяться всякий раз, когда представление Myratings будет модифицироваться, но предикат Highratings все равно будет проигнорирован.

Это — дефект в стандарте ANSI, который у большинства программ исправлен. Вы можете попробовать использовать представление наподобии последнего примера и посмотреть избавлена ли ваша система от этого дефекта. (Попытка выяснить это самостоятельно может быть иногда быть проще и яснее, чем поиск ответа в документации системы.)

РЕЗЮМЕ

Вы теперь овладели знаниями о представлениях полностью. Кроме правил определяющих, является ли данное представление модифицируемыми в SQL, вы познакомились с основными понятиями на которых эти правила базируются — т.е что модификации в представлениях допустимы только когда SQL может недвусмысленно определить, какие значения базовой таблицы можно изменять.

Это означает что команда модификации, при выполнении, не должна требовать ни изменений для многих строк сразу, ни сравнений между многочисленными строками либо базовой таблицы либо вывода запроса. Так как обьединения включают в себя сравнение строк, они также запрещены. Вы также поняли различие между некоторыми способами которые используют модификацируемые представления и представления только_чтение.

Вы научились воспринимать модифицируемые представления как окна, отображающие данные одиночной таблицы, но необязательно исключающие или реоргани-

зующие столбцы, посредством выбора только определенных строк отвечающих условию предиката.

Представления *только_чтение*, с другой стороны, могут содержать более допустимые запросы SQL; они могут следовательно стать способом хранения запросов, которые вам нужно часто выполнять в неизменной форме. Кроме того, наличие запроса, чей вывод обрабатывается как объект данных, дает вам возможность иметь ясность и удобство при создании запросов в выводе запросов.

Вы теперь можете предохранять команды модификации в представлении от создания строк в базовой таблице, которые не представлены в самом представлении с помощью предложения WITH CHECK OPTION в определении представления. Вы можете также использовать WITH CHECK OPTION как один из способов ограничения в базовой таблице. В автономных запросах, вы обычно используете один или более столбцов в предикате не представленых среди выбранных для вывода, что не вызывает никаких проблем. Но если эти запросы используются в модифицируемых представлениях, появляются проблемы, так как эти запросы производят представления, которые не могут иметь вставляемых в них строк. Вы видели некоторые подходы к этим проблемам.

В Главах 20 И 21, мы говорили, что представления имеют прикладные программы защиты. Вы можете позволить пользователям обращаться к представлениям не разрешая в тоже время обращаться к таблицам в которых эти представления непосредственно находятся. Глава 22 будет исследовать вопросы доступа к объектам данных в SQL.

РАБОТА С SQL

1. Какое из этих представлений — модифицируемое?

```
#1 CREATE VIEW Dailyorders
   AS SELECT DISTINCT cnum, snum, onum, odate
      FROM Orders;

#2 CREATE VIEW Custotals
   AS SELECT cname, SUM (amt)
      FROM Orders, Customers
      WHERE Orders.cnum = customer.cnum
      GROUP BY cname;

#3 CREATE VIEW Thirdorders
   AS SELECT *
      FROM Dailyorders
      WHERE odate = 10/03/1990;

#4 CREATE VIEW Nullcities
   AS SELECT snum, sname, city
      FROM Salespeople
      WHERE city IS NULL OR sname BETWEEN 'A' AND 'MZ';
```

2. Создайте представление таблицы Продавцов с именем Commissions (Комиссионные). Это представление должно включать только поля comm и snum. С помощью этого представления, можно будет вводить или изменять комиссионные, но только для значений между .10 и .20.
3. Некоторые SQL реализации имеют встроенную константу, представляющую текущую дату, иногда называемую "*CURDATE*". Слово CURDATE может, следовательно, использоваться в операторе SQL, и заменяться текущей датой, когда его значение станет доступным с помощью таких команд как SELECT или INSERT. Мы будем использовать представление таблицы Порядков с именем Entryorders для вставки строк в таблицу Порядков. Создайте таблицу порядков, так чтобы

CURDATE автоматически вставлялась в поле odate, если не указано другого значения. Затем создайте представление Entryorders, так чтобы значения не могли быть указаны.

(См. Приложение А для ответов.)

22

КТО ЧТО МОЖЕТ ДЕЛАТЬ В БАЗЕ ДАННЫХ

В ЭТОЙ ГЛАВЕ, ВЫ ОБУЧИТЕСЬ РАБОТЕ С ПРИВИЛЕГИЯМИ. Как сказано в Главе 2, SQL используется обычно в средах, которые требуют распознавания пользователей и различия между различными пользователями систем. Вообще говоря, администраторы баз данных, сами создают пользователей и дают им привилегии. С другой стороны пользователи которые создают таблицы, сами имеют права на управление этими таблицами. *Привилегии* — это то, что определяет, может ли указаный пользователь выполнить данную команду. Имеется несколько типов привилегий, соответствующих нескольким типам операций. Привилегии даются и отменяются двумя командами SQL: — **GRANT** (*ДОПУСК*) и **REVOKE** (*ОТМЕНА*).

Эта глава покажет вам, как эти команды используются.

ПОЛЬЗОВАТЕЛИ

Каждый пользователь в среде SQL, имеет специальное идентификационное имя или номер. Терминология везде разная, но мы выбрали (следуя ANSI) ссылку на имя или номер как на *Идентификатор* (**ID**) доступа. Команда, посланная в базе данных ассоциируется с определенным пользователем; или иначе, специальным *Идентификатором доступа*. Поскольку это относится к SQL базе данных, ID разрешения — это имя пользователя, и SQL может использовать специальное ключевое слово **USER**, которое ссылается к Идентификатору доступа связанному с текущей командой. Команда интерпретируется и разрешается (или запрещается) на основе информации связанной с Идентификатором доступа пользователя подавшего команду.

РЕГИСТРАЦИЯ

В системах с многочислеными пользователями, имеется некоторый вид процедуры входа в систему, которую пользователь должен выполнить, чтобы получить доступ к компьютерной системе. Эта процедура определяет какой ID доступа будет связан с текущим пользователем. Обычно, каждый человек использующий базу данных должен иметь свой собственный ID доступа и при регистрации превращается в действительного пользователям. Однако, часто пользователи имеющие много задач, могут регистрироваться под различными ID доступа, или наоборот один ID доступа может использоваться несколькими пользователями.

С точки зрения SQL, нет никакой разницы между этими двумя случаями; он воспринимает пользователя просто как его ID доступа.

SQL база данных может использовать собственную процедуру входа в систему, или она может позволить другой программе, типа операционной системы (основная программа, которая работает на вашем компьютере), обрабатывать файл регистрации и получать ID доступа из этой программы. Тем или другим способом, но SQL будет иметь ID доступа, чтобы связать его с вашими действиями, а для вас будет иметь значение ключевое слово **USER**.

ПРЕДОСТАВЛЕНИЕ ПРИВИЛЕГИЙ

Каждый пользователь в SQL базе данных имеет набор привилегий. Это — то, что пользователю разрешается делать (возможно это — файл регистрации, который может рассматриваться как минимальная привилегия). Эти привилегии могут изменяться со временем — новые добавляться, старые удаляться. Некоторые из этих привилегий определены в ANSI SQL, но имеются и дополнительные привилегии, которые являются также необходимыми.

SQL привилегии, как определено ANSI, не достаточны в большинстве ситуаций реальной жизни. С другой стороны, типы привилегий, которые необходимы, могут видоизменяться с видом системы, которую вы используете, относительно которой ANSI не может дать никаких рекомендаций. Привилегии, которые не являются частью стандарта SQL, могут использовать похожий синтаксис и не полностью совпадающий со стандартом.

СТАНДАРТНЫЕ ПРИВИЛЕГИИ

SQL привилегии определенные ANSI — это привелегии объекта. Это означает что пользователь имеет привилегию чтобы выполнить данную команду только на определенном объекте в базе данных. Очевидно, что привилегии должны различать эти объекты, но система привилегии основанная исключительно на привилегиях объекта не может адресовывать все что нужно SQL, как мы увидем это позже в этой главе.

Привилегии объекта связаны одновременно и с пользователями и с таблицами. То есть, привилегия дается определенному пользователю в указаной таблице, или базовой таблице или представлении. Вы должны помнить, что пользователь, создавший таблицу (любого вида), является владельцем этой таблицы. Это означает, что пользователь имеет все привилегии в этой таблице и может передавать привелегии другим пользователям в этой таблице.

Привилегии, которые можно назначить пользователю:

SELECT Пользователь с этой привилегией может выполнять запросы в таблице.

INSERT Пользователь с этой привилегией может выполнять команду INSERT в таблице.

UPDATE Пользователь с этой привилегией может выполнять команду UPDATE на таблице. Вы можете ограничить эту привилегию для определенных столбцов таблицы.

DELETE Пользователь с этой привилегией может выполнять команду DELETE в таблице.

REFERENCES Пользователь с этой привилегией может определить внешний ключ, который использует один или более столбцов этой таблицы, как родительский ключ. Вы можете ограничить эту привилегию для определенных столбцов. (Смотрите Главу 19 для подробностей относительно внешнего ключа и родительского ключа.)

Кроме того, вы столкнетесь с нестандартными привилегиями объекта, такими например как **INDEX** (*ИНДЕКС*) дающим право создавать индекс в таблице, **SYNONYM** (*СИНОНИМ*) дающим право создавать синоним для объекта, который будет объяснен в Главе 23, и **ALTER** (*ИЗМЕНИТЬ*), дающим право выполнять команду ALTER TABLE в таблице. Механизм SQL назначает пользователям эти привилегии с помощью команды **GRANT**.

КОМАНДА GRANT

Позвольте предположить, что пользователь Diane имеет таблицу Заказчиков и хочет позволить пользователю Adrian выполнить запрос к ней. Diane должна в этом случае ввести следующую команду:

```
GRANT SELECT ON Customers TO Adrian;
```

Теперь Adrian может выполнить запросы к таблице Заказчиков. Без других привилегий, он может только выбрать значения; но не может выполнить любое действие, которые бы воздействовало на значения в таблице Заказчиков (включая использование таблицы Заказчиков в качестве родительской таблицы внешнего ключа, что ограничивает изменения которые выполнять со значениям в таблице Заказчиков).

Когда SQL получает команду GRANT, он проверяет привилегии пользователя, подавшего эту команду, чтобы определить допустима ли команда GRANT.

Adrian самостоятельно не может выдать эту команду. Он также не может предоставить право SELECT другому пользователю: таблица еще принадлежит Diane (позже мы покажем как Diane может дать право Adrian предоставлять SELECT другим пользователям).

Синтаксис — тот же самый, что и для предоставления других привилегий. Если Adrian — владелец таблицы Продавцов, то он может позволить Diane вводить в нее строки с помощью следующего предложения

```
GRANT INSERT ON Salespeople TO Diane;
```

Теперь Diane имеет право помещать нового продавца в таблицу.

ГРУППЫ ПРИВЕЛЕГИЙ, ГРУППЫ ПОЛЬЗОВАТЕЛЕЙ

Вы не должны ограничивать себя предоставлением одиночной привилегии отдельному пользователю командой GRANT. Списки привилегий или пользователей, отделяемых запятыми, являются совершенно приемлемыми. Stephen может предоставить и SELECT и INSERT в таблице Порядков для Adrian

```
GRANT SELECT, INSERT ON Orders TO Adrian;
```

или и для Adrian и для Diane

```
GRANT SELECT, INSERT ON Orders TO Adrian, Diane;
```

Когда привилегии и пользователи перечислены таким образом, весь список привилегий предоставляются всем указанным пользователям. В строгой ANSI интерпретации, вы не можете предоставить привилегии во многих таблицах сразу одной командой, но в некоторых реализациях это ограничение может быть ослаблено, позволяя вам указывать несколько таблиц, отделяя их запятыми, так что бы весь список привелегий мог быть предоставлен для всех указанных таблиц.

ОГРАНИЧЕНИЕ ПРИВИЛЕГИЙ НА ОПРЕДЕЛЕННЫЕ СТОЛБЦЫ

Все привилегии объекта используют один тот же синтаксис, кроме команд UPDATE и REGERNCES в которых необязательно указывать имена столбцов. Привилегию UPDATE можно предоставлять на подобии других привилегий:

```
GRANT UPDATE ON Salespeople TO Diane;
```

Эта команда позволит Diane изменять значения в любом или во всех столбцах таблицы Продавцов. Однако, если Adrian хочет ограничить Diane в изменении например комиссионных, он может ввести

```
GRANT UPDATE (comm) ON Salespeople TO Diane;
```

Другими словами, он просто должен указать конкретный столбец, к которому привилегия UPDATE должна быть применена, в круглых скобках после имени таблицы. Имена многочисленых столбцов таблицы могут указываться в любом порядке, отделяемые запятыми:

```
GRANT UPDATE (city, comm) ON Salespeople TO Diane;
```

REFERENCES следует тому же самому правилу. Когда вы предоставите привилегию REFERENCES другому пользователю, он сможет создавать внешние ключи ссылающиеся на столбцы вашей таблицы как на родительские ключи. Подобно UPDATE, для привилегии REFERENCES может быть указан список из одного или более столбцов для которых ограничена эта привилегия. Например, Diane может предоставить Stephen право использовать таблицу Заказчиков, как таблицу родительского ключа, с помощью такой команды:

```
GRANT REFERENCES (cname, cnum) ON Customers TO Stephen;
```

Эта команда дает Stephen право использовать столбцы cnum и cname, в качестве родительских ключей по отношению к любым внешним ключам в его таблицах. Stephen может контролировать то, как это будет выполнено. Он может определить (cname, cnum) или, как в нашем случае (cnum, cname), как двух-столбцовый родительский ключ, совпадающий с помощью внешнего ключа с двумя столбцами в одной из его собственных таблиц. Или он может создать раздельные внешние ключи, чтобы ссылаться на поля индивидуально, обеспечив тем самым, чтобы Diane имела принудительное присвоение родительского ключа (см. Главу 19).

Не имея ограничений на номера внешних ключей он должден базироваться на этих родительских ключах, а родительские ключи различных внешних ключей — разрешены для совмещения (overlap).

Как и в случае с привилегией UPDATE, вы можете исключить список столбцов и таким образом позволять всем без исключения столбцам быть используемыми в качестве родительских ключей. Adrian может предоставить Diane право сделать это следующей командой:

```
GRANT REFERENCES ON Salespeople TO Diane;
```

Естественно, привилегия будет пригодна для использования только в столбцах, которые имеют ограничения требуемые для родительских ключей.

ИСПОЛЬЗОВАНИЕ АРГУМЕНТОВ ALL И PUBLIC

SQL поддерживает два аргумента для команды GRANT, которые имеют специальное значение: **ALL PRIVILEGES** (*ВСЕ ПРИВИЛЕГИИ*) или просто **ALL** и **PUBLIC** (*ОБЩИЕ*). **ALL** используется вместо имен привилегий в команде GRANT чтобы отдать все привилегии в таблице. Например, Diane может дать Stephen весь набор привилегий в таблице Заказчиков с помощью такой команды:

```
GRANT REFERENCES ON Salespeople TO Diane;
```

(привилегии UPDATE и REFERENCES естественно применяются ко всем столбцам.) А это другой способ высказать ту же мысль:

```
GRANT ALL ON Customers TO Stephen;
```

PUBLIC — больше похож на тип аргумента — *захватить все* (catch-all), чем на пользовательскую привилегию.

Когда вы предоставляете привилегии для публикации, все пользователи автоматически их получают. Наиболее часто, это применяется для привилегии SELECT в определенных базовых таблицах или представлениях которые вы хотите сделать доступными для любого пользователя. Чтобы позволить любому пользователю видеть таблицу Порядков, вы, например, можете ввести следующее:

```
GRANT SELECT ON Orders TO PUBLIC;
```

Конечно, вы можете предоставить любые или все привилегии обществу, но это видимо нежелательно. Все привилегии за исключением SELECT позволяют пользователю изменять (или, в случае REFERENCES, ограничивать) содержание таблицы. Разрешение всем пользователям изменять содержание ваших таблиц вызовет проблему.

Даже если вы имеете небольшую компанию, и в ней работают все ваши текущие пользователи, способные выполнять команды модификации в данной таблице, было бы лучше предоставить привилегии каждому пользователю индивидуально, чем одни и те же привелегии для всех.

PUBLIC не ограничен в его передаче только текущим пользователям. Любой новый пользователь, добавляемый к вашей системе, автоматически получит все привилегии, назначенные ранее всем, так что если вы захотите ограничить доступ к таблице всем, сейчас или в будущем, лучше всего предоставить привилегии, иные чем SELECT для индивидуальных пользователей.

ПРЕДОСТАВЛЕНИЕ ПРИВЕЛЕГИЙ С ПОМОЩЬЮ WITH GRANT OPTION

Иногда, создателю таблицы хочется чтобы другие пользователи могли получить привилегии в его таблице. Обычно это делается в системах, где один или более людей создают несколько (или все) базовые таблицы в базе данных а затем передают ответственность за них тем кто будет фактически с ними работать. SQL позволяет делать это с помощью предложения **WITH GRANT OPTION**.

Если Diane хотела бы чтобы Adrian имел право предоставлять привилегию SELECT в таблице Заказчиков другим пользователям, она дала бы ему привилегию SELECT с использованием предложения WITH GRANT OPTION:

```
GRANT SELECT ON Customers TO Adrian
WITH GRANT OPTION;
```

После того Adrian получил право передавать привилегию SELECT третьим лицам. Он может выдать команду

```
GRANT SELECT ON Diane.Customers TO Stephen;
```

или даже

```
GRANT SELECT ON Diane.Customers TO Stephen
WITH GRANT OPTION;
```

Пользователь с помощью GRANT OPTION в особой привилегии для данной таблицы, может, в свою очередь, предоставить эту привилегию к той же таблице, с или без GRANT OPTION, любому другому пользователю. Это не меняет принадлежности самой таблицы; как и прежде таблица принадлежат ее создателю. (поэтому пользова-

тели, получившие права, должны устанавливать префикс ID доступа владельца, когда ссылаются к этим таблицам. Следующая глава покажет вам этот способ.) Пользователь же с помощью GRANT OPTION во всех привилегиях для данной таблицы будет иметь всю полноту власти в той таблице.

ОТМЕНА ПРИВИЛЕГИЙ

Также как ANSI предоставляет команду CREATE TABLE чтобы создать таблицу, но не DROP TABLE чтобы от нее избавиться, так и команда GRANT позволяет вам давать привилегии пользователям, не предоставляя способа, чтобы отобрать их обратно. Потребность удалять привилегии сводится к команде REVOKE, фактически стандартному средству с достаточно понятной формой записи.

Синтаксис команды REVOKE — похож на GRANT, но имеет обратный смысл. Чтобы удалить привилегию INSERT для Adrian в таблице Порядков, вы можете ввести

```
REVOKE INSERT ON Orders FROM Adrian;
```

Использование списков привилегий и пользователей здесь допустимы, как и в случае с GRANT, так что вы можете ввести следующую команду:

```
REVOKE INSERT, DELETE ON Customers
FROM Adrian, Stephen;
```

Однако, здесь имеется некоторая неясность. Кто имеет право отменять привилегии? Когда пользователь с правом передавать привелегии другим, теряет это право? Пользователи которым он предоставил эти привилегии, также их потеряют ? Так как это не стандартная особенность, нет никаких авторитетных ответов на эти вопросы, но наиболее общий подход — это такой:

* Привилегии отменяются пользователем, который их предоставил, и отмена будет *каскадироваться*, то-есть она будет автоматически распространяться на всех пользователей, получивших от него эту привилегию.

ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ ДЛЯ ФИЛЬТРАЦИИ ПРИВЕЛЕГИЙ

Вы можете сделать действия привилегий более точными, используя представления. Всякий раз, когда вы передаете привилегию в базовой таблице пользователю, она автоматически распространяется на все строки, а при использовании возможных исключений UPDATE и REFERENCES, на все столбцы таблицы.

Создавая представление, которое ссылается на основную таблицу, и затем перенося привилегию на представление, а не на таблицу, вы можете ограничивать эти привилегии любыми выражениями в запросе, содержащимся в представлении. Это значительно улучшает базисные возможности команды GRANT.

КТО МОЖЕТ СОЗДАВАТЬ ПРЕДСТАВЛЕНИЯ?

Чтобы создавать представление, вы должны иметь привилегию SELECT во всех таблицах, на которые вы ссылаетесь в представлении. Если представление — модифицируемое, любая привелегия INSERT, UPDATE и DELETE, которые вы имеете в базовой таблице, будут автоматически передаваться представлению. Если вы испытываете недостаток в привилегиях на модификацию в базовых таблицах, вы не смо-

жете иметь их и в представлениях, которые создали, даже если сами эти представления — модифицируемые. Так как внешние ключи не используются в представлениях, привилегия REFERENCES никогда не используется при создании представлений. Все эти ограничения — определяются ANSI. Нестандартные привилегии системы (обсуждаемые позже в этой главе) также могут быть включены. В последующих разделах мы предположим, что создатели представлений которые мы обсуждаем, имеют частные или соответствующие привилегии во всех базовых таблицах.

ОГРАНИЧЕНИЕ ПРИВИЛЕГИИ SELECT ДЛЯ ОПРЕДЕЛЕННЫХ СТОЛБЦОВ

Предположим, вы хотите дать пользователю Claire способность видеть только столбцы snum и sname таблицы Продавцов. Вы можете сделать это, поместив имена этих столбцов в представление

```
CREATE VIEW Clairesview
AS SELECT snum, sname
   FROM Salespeople;
```

и предоставив Claire привилегию SELECT в представлении, а не в самой таблице Продавцов:

```
GRANT SELECT On Clairesview to Claire;
```

Вы можете создать привилегии специально для столбцов наподобии использования других привилегий, но, для команды INSERT, это будет означать вставку значений по умолчанию, а для команды DELETE, ограничение столбца не будет иметь значения. Привелегии REFERENCES и UPDATE, конечно, могут сделать столбцы специфическими не прибегая к представлению.

ОГРАНИЧЕНИЕ ПРИВЕЛЕГИЙ ДЛЯ ОПРЕДЕЛЕННЫХ СТРОК

Обычно, более полезный способ чтобы фильтровать привилегии с представлениями — это использовать представление чтобы привилегия относилась только к определенным строкам. Вы делаете это, естественно, используя предикат в представлении который определит, какие строки являются включенными. Чтобы предоставить пользователю Adrian, привилегию UPDATE в таблице Заказчиков, для всех заказчиков размещенных в Лондоне, вы можете создать такое представление:

```
CREATE VIEW Londoncust
AS SELECT *
   FROM Customers
   WHERE city = 'London'
   WITH CHECK OPTION;
```

Затем Вы должны передать привилегию UPDATE в этой таблице для Adrian:

```
GRANT UPDATE ON Londoncust TO Adrian;
```

В этом отличие привилегии для определенных строк от привилегии UPDATE для определенных столбцов, которая распространена на все столбцы таблицы Заказчиков, но не на строки, среди которых строки со значением поля city иным чем London не

будут учитываться. Предложение **WITH CHECK OPTION** предохраняет Adrian от замены значения поля city на любое значение кроме London.

ПРЕДОСТАВЛЕНИЕ ДОСТУПА ТОЛЬКО К ИЗВЛЕЧЕННЫМ ДАННЫМ

Другая возможность состоит в том, чтобы предлагать пользователям доступ к уже извлеченным данным, а не к фактическим значениям в таблице. Агрегатные функции, могут быть весьма удобными в применении такого способа. Вы можете создавать представление которое дает счет, среднее, и общее количество для порядков на каждый день порядка:

```
CREATE VIEW Datetotals
AS SELECT odate, COUNT (*), SUM (amt), AVG (amt)
FROM Orders
GROUP BY odate;
```

Теперь вы передаете пользователю Diane — привелегию SELECT в представлении Datetotals:

```
GRANT SELECT ON Datetotals TO Diane;
```

ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ В КАЧЕСТВЕ АЛЬТЕРНАТИВЫ К ОГРАНИЧЕНИЯМ

Одной из последних прикладных программ из серии, описанной в Главе 18, является использование представлений с WITH CHECK OPTION как альтернативы к ограничениям. Предположим что вы хотели удостовериться, что все значения поля city в таблице Продавцов находятся в одном из городов где ваша компания в настоящее время имеет ведомство. Вы можете установить ограничение CHECK непосредственно на столбец city, но позже может стать трудно его изменить, если ваша компания например откроет там другие ведомства. В качестве альтернативы, можно создать представление, которое исключает неправильные значения city:

```
CREATE VIEW Curcities
AS SELECT *
   FROM Salespeople
   WHERE city IN ('London', 'Rome', 'San Jose', 'Berlin')
   WITH CHECK OPTION;
```

Теперь, вместо того, чтобы предоставить пользователям привилегии модифицирования в таблице Продавцов, вы можете предоставить их в представлении Curcities. Преимущество такого подхода — в том, что если вам нужно сделать изменение, вы можете удалить это представление, создать новое, и предоставить в этом новом представлении привилегии пользователям, что проще, чем изменять ограничения. Недостатком является то, что владелец таблицы Продавцов также должен использовать это представление, если он не хочет, чтобы его собственные команды были отклонены.

С другой стороны, этот подход позволяет владельцу таблицы и любым другим получить привилегии модификации в самой таблице, а не в представлении, чтобы делать исключения для ограничений. Это часто бывает желательно, но не выполнимо, если вы используете ограничения в базовой таблице. К сожалению, эти исключения

нельзя будет увидеть в представлении. Если вы выберите этот подход, вам захочется создать второе представление, содержащее только исключения:

```
CREATE VIEW Othercities
AS SELECT *
   FROM Salespeople
   WHERE city NOT IN ('London', 'Rome', 'San Jose', 'Berlin')
   WITH CHECK OPTION;
```

Вы должны выбрать для передачи пользователям только привилегию SELECT в этом представлении, чтобы они могли видеть исключенные строки, но не могли помещать недопустимые значения city в базовую таблицу. Фактически, пользователи могли бы сделать запрос обоих представлений в объединении и увидеть все строки сразу.

ДРУГИЕ ТИПЫ ПРИВИЛЕГИЙ

Вы разумеется хотите знать, кто же имеет право первым создать таблицу. Эта область привилегии не относится к ANSI, но не может игнорироваться. Все стандартные привилегии ANSI вытекают из этой привилегии; привилегии создателей таблиц которые могут передавать привилегии объекта. Если все ваши пользователи будут создавать в системе базовые таблицы с разными размерами это приведет к избыточности в них и к неэффективности системы. Притягивают к себе и другие вопросы:

— Кто имеет право изменять, удалять, или ограничивать таблицы?

— Должны ли права создания базовых таблиц отличаться от прав создания представлений?

— Должен ли быть суперпользователь — пользователь который отвечает за поддержание базы данных и следовательно имеющий наибольшие, или все привилегии, которые не предоставляются индивидуально?

Пока ANSI не принимает в этом участие, а SQL используется в различных средах, мы не можем дать окончательный ответ на эти вопросы. Мы предлагаем рассмотреть здесь кусок наиболее общих выводов.

Привилегии, которые не определяются в терминах специальных объектов данных, называются *привилегиями системы*, или правами базы данных. На базисном уровне, они будут вероятно включать в себя право создавать объекты данных, вероятно отличающиеся от базовых таблиц (обычно создаваемыми несколькими пользователями) и представления (обычно создаваемые большинством пользователей). Привилегии системы для создания представлений должны дополнять, а не заменять привилегии объекта, которые ANSI требует от создателей представлений (описанных ранее в этой главе).

Кроме того, в системе любого размера всегда имеются некоторые типы суперпользователей — пользователей, которые автоматически имеют большинство или все привилегии, — и которые могут передать свой статус суперпользователя кому-нибудь с помощью привилегии или группы привилегий. *Администратор Базы Данных*, или **DBA**, является термином, наиболее часто используемым для такого суперпользователя и для привилегий, которыми он обладает.

ТИПИЧНЫЕ ПРИВИЛЕГИИ СИСТЕМЫ

При общем подходе имеется три базовых привилегии системы:

- **CONNECT** (*Подключить*),
- **RESOURCE** (*Ресурс*), *и*
- **DBA** (*Администратор Базы Данных*).

Проще, можно сказать, что **CONNECT** состоит из права зарегистрироваться и права создавать представления и синонимы (см. Главу 23), если переданы привилегии объекта. **RESOURCE** состоит из права создавать базовые таблицы. **DBA** — это привилегия суперпользователя, дающая пользователю высокие полномочия в базе данных. Один или более пользователей с функциями администратора базы данных может иметь эту привилегию. Некоторые системы кроме того имеют специального пользователя, иногда называемого **SYSADM** или **SYS** (*Системный Администратор Базы Данных*), который имеет наивысшие полномочия; это — специальное имя, а не просто пользователь со специальной DBA привилегией. Фактически только один человек имеет право зарегистрироваться с именем SYSADM, являющимся его идентификатором доступа. Различие весьма тонкое и функционирует по разному в различных системах. Для наших целей, мы будем ссылаться на высокопривилегированного пользователя, который разрабатывает и управляет базой данных имея полномочия DBA, понимая что фактически эти полномочия — та же самая привилегия. Команда GRANT, в измененной форме, является пригодной для использования с привилегиями объекта как и с системными привилегиями. Для начала передача прав может быть сделана с помощью DBA. Например, DBA может передать привилегию для создания таблицы пользователю Rodriguez следующим образом:

```
GRANT RESOURCE TO Rodriguez;
```

СОЗДАНИЕ И УДАЛЕНИЕ ПОЛЬЗОВАТЕЛЕЙ

Естественно, появляется вопрос, откуда возьмется пользователь с именем Rodriguez? Как определить его ID допуска? В большинстве реализаций, DBA создает пользователя, автоматически предоставляя ему привилегию CONNECT.

В этом случае, обычно добавляется предложение **IDENTIFIED BY**, указывающее пароль. (Если же нет, операционная система должна определить, можете ли вы зарегистрироваться в базе данных с данным ID доступа.) DBA может, например, ввести

```
GRANT CONNECT TO Thelonius IDENTIFIED BY Redwagon;
```

что приведет к созданию пользователя, с именем Thelonius, даст ему право регистрироваться, и назначит ему пароль Redwagon, и все это в одном предложении.

Раз Thelonious — уже опознаный пользователь, он или DBA могут использовать эту же команду чтобы изменить пароль Redwagon. Хотя это и удобно, но все же имеются ограничения и в этом подходе. Это невозможность иметь пользователя, который не мог бы зарегистрироваться, хотя бы временно. Если вы хотите запретить пользователю регистрироваться, вы должны использовать для REVOKE привилегию CONNECT, которая "удаляет" этого пользователя. Некоторые реализации позволяют вам создавать и удалять пользователей, независимо от их привилегий при регистрации.

Когда вы предоставляете привилегию CONNECT пользователю, вы создаете этого пользователя. При этом чтобы сделать это Вы сами, должны иметь DBA привилегию. Если этот пользователь будет создавать базовые таблицы (а не только представления), ему нужно также предоставить привилегию RESOURCE. Но это сразу порождает другую проблему.

Если вы сделаете попытку удалить привилегию CONNECT пользователя, который имеет им созданые таблицы, команда будет отклонена, потому что ее действие

оставит таблицу без владельца, а это не позволяется. Вы должны сначала удалить все таблицы созданные этим пользователем, прежде чем удалить его привилегию CONNECT. Если эти таблицы не пустые, то вы вероятно захотите передать их данные в другие таблицы с помощью команды INSERT, которая использует запрос. Вам не нужно удалять отдельно привилегию RESOURSE; достаточно удалить CONNECT чтобы удалить пользователя.

Хотя все выше сказаное — это вполне стандартный подход к привилегиям системы, он также имеет значительные ограничения. Появились альтернативные подходы, которые более конкретно определены и точнее управляют привилегиями системы.

Эти выводы несколько выводят нас за пределы стандарта SQL как это определено в настоящее время, и, в некоторых реализациях, могут полностью выйти за пределы стандарта SQL. Эти вещи вероятно не будут слишком вас касаться, если вы не DBA или не пользователь высокого уровня. Обычные пользователи просто должны быть знакомыми с привилегиями системы в принципе, справляясь со своей документации только в случае специальных сообщений.

РЕЗЮМЕ

Привилегии дают вам возможность видеть SQL под новым углом зрения, когда SQL выполняет действия через специальных пользователей в специальной системе базы данных. Сама команда GRANT достаточно проста: с ее помощью вы предоставляете те или иные привилегии объекта одному или более пользователям. Если вы предоставляете привилегию WITH GRANT OPTION пользователю, этот пользователь может в свою очередь предоставить эту привилегию другим.

Теперь вы понимаете намеки на использование привилегий в представлениях — чтобы усовершенствовать привилегии в базовых таблицах, или как альтернативы к ограничениям — и на некоторые преимущества и недостатки такого подхода. Привилегии системы, которые необходимы, но не входят в область стандарта SQL, обсуждались в их наиболее общей форме и поэтому вы будете знакомиться с ними на практике.

Глава 23 продолжит обсуждение о выводах в SQL, таких как сохранение или восстановление изменений, создание ваших собственных имен для таблиц принадлежащих другим людям, и понимание что происходит когда различные пользователи пытаются обращаться к одному и тому же объекту одновременно.

РАБОТА С SQL

1. Передайте Janet право на изменение оценки заказчика.
2. Передайте Stephan право передавать другим пользователям право делать запросы в таблице Порядков.
3. Отнимите привилегию INSERT (ВСТАВКА) в таблице Продавцов у Claire и у всех пользователей которым она была предоставлена.
4. Передайте Jerry право вставлять или модифицировать таблицу Заказчиков с сохранением его возможности оценивать значения в диапазоне от 100 до 500.
5. Разрешите Janet делать запросы в таблице Заказчиков, но запретите ему уменьшать оценки в той же таблице Заказчиков.

(См. Приложение A для ответов.)

23

ГЛОБАЛЬНЫЕ АСПЕКТЫ SQL

ЭТА ГЛАВА БУДЕТ ОБСУЖДАТЬ АСПЕКТЫ ЯЗЫКА SQL, которые имеют отношение к базе данных как к единому целому, включая использование многочисленых имен для объектов данных, размещение запоминаемых данных, восстановлние и сохранение изменений в базе данных, а также координирование одновременных действий многочисленных пользователей. Этот материал даст вам возможность конфигурации вашей базы данных, отмены действия ошибок, и определения как действия одного пользователя в базе данных будут влиять на действия других пользователей.

ПЕРЕИМЕНОВАНИЕ ТАБЛИЦ

Каждый раз, когда вы ссылаетесь в команде к базовой таблице или представлению не являющимися вашей собственностью, вы должны установить в ней префикс имени владельца, так что бы SQL знала где ее искать. Так как это со временем становится неудобным, большинство реализаций SQL позволяют вам создавать синонимы для таблиц (что не является стандартом ANSI).

Синоним — это альтернативное имя, наподобии прозвища, для таблицы. Когда вы создаете синоним, вы становитесь его собственником, так что нет никакой необходимости, чтобы он предшествовал другому пользовательскому идентификатору доступа (имени пользователя) Если вы имеете по крайней мере одну привилегию в одном или более столбцах таблицы, вы можете создать для них синоним. (Некоторое отношение к этому может иметь специальная привилегия для создания синонимов.)

Adrian может создать синоним с именем Clients, для таблицы с именем Diane.Customers, с помощью команды CREATE SYNONYM следующим образом:

```
CREATE SYNONYM Clients FOR Diane.Customers;
```

Теперь, Adrian может использовать таблицу с именем Clients в команде точно так же, как использует Diane.Customers. Синоним Clients — это собственность, используемая исключительно для Adrian.

ПЕРЕИМЕНОВАНИЕ С ТЕМ ЖЕ САМЫМ ИМЕНЕМ

Префикс (прозвище) пользователя — это фактически часть имени любой таблицы. Всякий раз, когда вы не указываете ваше собственное имя пользователя вместе с именем вашей собственной таблицы, SQL сам заполняет для вас это место. Следовательно, два одинаковых имени таблицы, но связанные с различными владельцами, становятся не идентичными и следовательно не приводят к какому-нибудь беспорядку (по крайней мере в SQL). Это означает, что два пользователя могут создать две полностью несвязанные таблицы с одинаковыми именами, но это также будет означать, что один пользователь может создать представление, основанное на имени другого пользователя, стоящем после имени таблицы. Это иногда делается когда, представление рассматривается как сама таблица — например, если представление просто использует CHECK OPTION как заменитель ограничения CHECK в базовой таблице (смотрите Главу 22 для подробностей). Вы можете также создавать ваши собственные синонимы, имена которых будут такими же, что и первоначальные имена таблиц. Например, Adrian может определить Customers, как свой синоним для таблицы Diane.Customers:

```
CREATE SYNONYM Customers FOR Diane.Customers;
```

С точки зрения SQL, теперь имеются два разных имени одной таблицы: **Diane.Customers** и **Adrian.Customers**. Однако, каждый из этих пользователей может ссылаться к этой таблице просто как к Customers, SQL как говорилось выше сам добавит к ней недостающие имена пользователей.

ОДНО ИМЯ ДЛЯ КАЖДОГО

Если вы планируете иметь таблицу Заказчиков используемую большим числом пользователей, лучше всего что бы они ссылались к ней с помощью одного и того же имени. Это даст вам возможность, например, использовать это имя в вашем внутреннем общении без ограничений. Чтобы создать единое имя для всех пользователей, вы создаете общий синоним. Например, если все пользователи будут вызывать таблицу Заказчиков с именем Customers, вы можете ввести

```
CREATE PUBLIC SYNONYM Customers FOR Customers;
```

Мы пронимаем, что таблица Заказчиков это ваша собственность, поэтому никакого префикса имени пользователя в этой команды не указывается. В основном, общие синонимы создаются владельцами объектов или привилегированными пользователями, типа DBA. Пользователям кроме того, должны еще быть предоставлены привилегии в таблице Заказчиков чтобы они могли иметь к ней доступ. Даже если имя является общим, сама таблица общей не является. Общие синонимы становятся собственными с помощью команды PUBLIC, а не с помощью их создателей.

УДАЛЕНИЕ СИНОНИМОВ

Общие и другие синонимы могут удаляться командой **DROP SYNONYM**. Синонимы удаляются их владельцами, кроме общих синонимов, которые удаляются соответствующими привилегированными личностями, обычно DBA. Чтобы удалить например синоним Clients, когда вместо него уже появился общий синоним Customers, Adrian может ввести

```
DROP SYNONYM Clients;
```

Сама таблица Заказчиков, естественно, становится не эффективной.

КАК БАЗА ДАННЫХ РАСПРЕДЕЛЕНА ДЛЯ ПОЛЬЗОВАТЕЛЕЙ?

Таблицы и другие объекты данных сохраняются в базе данных и находятся там связанными с определенными пользователями, которые ими владеют. В некотором смысле, вы могли бы сказать, что они сохраняются в "*именной области пользователя*", хотя это никак не отражает их физического расположения, но зато, как и большинство вещей в SQL, находятся в строгой логической конструкции. Однако, на самом деле, объекты данных сохраняются, в физическом смысле, и количество памяти, которое может использоваться определенным объектом или пользователем, в данное время, имеют свой предел.

В конце концов, никакой компьютер не имеет прямого доступа к бесконечному числу аппаратных средств (диску, ленте, или внутренней памяти) для хранения данных. Кроме того, эффективность SQL расширится, если логическая структура данных

будет отображаться неким физическим способом, при котором эти команды получат преимущество.

В больших SQL системах, база данных будет разделена на области, так называемые *Области Базы Данных* или *Разделы*.

Это области сохраняемой информации, которые размещены так, чтобы информация внутри них находилась близко друг к другу для выполнения команд; то-есть программа не должна искать где-то далеко информацию, сгруппированную в одиночной области базы данных. Хотя ее физические возможности зависят от аппаратного оборудования, целесообразно, чтобы команда работала в этих областях внутри самой SQL.

Системы которые используют *области базы данных* (в дальнейшем называемые — **DBS** (*Data Base Spaces*)), позволяют вам с помощью команд SQL обрабатывать эти области как объекты.

DBS создаются командами **CREATE DBSPACE** (*СОЗДАТЬ DBS*), **ACQUIRE DBSPACE** (*ПОЛУЧИТЬ DBS*) или **CREATE TABLESPACE** (*СОЗДАТЬ ТАБЛИЧНУЮ ОБЛАСТЬ*), в зависимости от используемой реализации. Одна DBS может вмещать любое число пользователей, и отдельный пользователь может иметь доступ к многим DBS. Привилегия создавать таблицы, хотя и может быть передана по всей базе данных, часто передается в конкретной DBS. Мы можем создать DBS с именем Sampletables, следующей командой:

```
CREATE DBSPACE Sampletables
(pctindex 10,
 pctfree  25);
```

Параметр pctindex определяет, какой процент DBS должен быть оставлен, чтобы сохранять в нем индексы таблиц. Pctfree — это процент DBS, который оставлен, чтобы позволить таблицам расширять размеры их строк (ALTER TABLE может добавлять столбцы или увеличивать размер столбцов, делая каждую строку длиннее. Это — расширение памяти, отводимой для этого). Имеются также другие параметры, которые вы также можете определять, и которые меняются от программы к программе. Большинство программ автоматически будут обеспечивать *значения по умолчанию*, поэтому вы можете создавать DBS, не определяя эти параметры. DBS может иметь или определенное ограничение размера, или ей может быть позволено расти неограниченно вместе с таблицами.

Если DBS создалась, пользователям предоставляются права создавать в ней объекты. Вы можете например предоставить Diane право создать таблицу Sampletables с помощью следующей команды:

```
GRANT RESOURCE ON Sampletables TO Diane;
```

Это даст вам возможность более конкретно определять место хранения данных. Первый DBS, назначаемый данному пользователю — обычно тот, где все объекты этого пользователя создаются по умолчанию.

Пользователи, имеющие доступ к многочисленым DBS, могут определить, где они хотят разместить определенный объект.

При разделении вашей базы данных на DBSы, вы должны иметь в виду типы операций, которые вы будете часто выполнять. Таблицы, которые, как вам уже известно, будут часто объединяться, или которые имеют одну таблицу, ссылающуюся на другую во внешнем ключе, должны находиться вместе в одной DBS.

Например, вы могли бы сообщить при назначении типовых таблиц, что таблица Порядков будет часто объединяться с одной или обеими из двух других таблиц, так как таблица Порядков использует значения из обеих этих таблиц. При прочих равных условиях, эти три таблицы должны входить в ту же самую область DBS, независимо

от того, кто их владелец. Возможное присутствие ограничения внешнего ключа в таблице Порядков, просто приведет к более строгому совместному использованию области DBS.

КОГДА СДЕЛАННЫЕ ИЗМЕНЕНИЯ СТАНОВЯТСЯ ПОСТОЯННЫМИ?

Визуально, среда базы данных — это картина, которая постоянно отображает для существующих пользователей постоянно вводимые и изменяемые данные, допуская, что если система правильно разработана, она будет функционировать без сбоев. Однако реально, благодаря человеческим или компьютерным сбоям, ошибки время от времени случаются, и поэтому хорошие компьютерные программы стали применять способы отмены действий, вызвавших такие ошибки.

Команда SQL, которая воздействует на содержание или структуру базы данных — например команда модификации DML или команда DROP TABLE, — не обязательно будет необратимой. Вы можете определить после окончания ее действия, останутся ли изменения, сделанные данной командой или группой команд постоянными в базеы данных, или они будут полностью проигнорированы. С этой целью, команды обрабатываются группами, называемыми *транзакциями*.

Транзакция начинается всякий раз, когда вы начинаете сеанс с SQL. Все команды которые вы введете будут частью этой транзакции, пока вы не завершите их вводом команды **COMMIT WORK** или команды **ROLLBACK WORK**. **COMMIT** может сделать все изменения постоянными с помощью транзакции, а **ROLLBACK** может откатить их братно или отменить. Новая транзакция начинается после каждой команды COMMIT или ROLLBACK. Этот процесс известен как *диалоговая обработка запросов* или ***транзакция***. Синтаксис, чтобы оставить все ваши изменения постоянными во время регистрации, или во время последнего COMMIT или ROLLBACK

```
COMMIT WORK;
```

Синтаксис отмены изменения -

```
ROLLBACK WORK;
```

В большинстве реализаций, вы можете установить параметр, называемый **AUTOCOMMIT**. Он будет автоматически запоминать все действия, которые будут выполняться. Действия, которые приведут к ошибке, всегда будут автоматически "прокручены" обратно. Если это предусмотрено в вашей системе, для фиксации всех ваших действий, вы можете использовать эту возможность с помощью команды типа:

```
SET AUTOCOMMIT ON;
```

Вы можете вернуться к обычной диалоговой обработке запросов с помощью такой команды:

```
SET AUTOCOMMIT OFF;
```

Имеется возможность установки AUTOCOMMIT, которую система выполнит автоматически при регистрации.

Если сеанс пользователя завершается аварийно — например, произошел сбой системы или выполнена перезагрузка пользователя, — то текущая транзакция выполнит автоматический откат изменений. Это — одна из причин, по которой вы можете управлять выпонением вашей диалоговой обработки запросов, разделив ваши команды на большое количество различных транзакций. Одиночная транзакция не должна

содержать много несвязанных команд; фактически, она может состоять из единственной команды.

Транзакции, которые включают всю группу несвязанных изменений, не оставляют вам фактически никакого выбора, сохранить или отклонить целую группу, если вы хотите отменить только одно определенное изменение. Хорошее правило, которому надо следовать, это делать ваши транзакции, состоящими из одной команды или нескольких близко связанных команд. Например, предположим, вы хотите удалить продавца Motika из базы данных. Прежде, чем вы удалите его из таблицы Продавцов, вы сначала должны сделать что-нибудь с его порядками и его заказчиками. (Если используются ограничения внешнего ключа, и ваша система, следуя ANSI, ограничивает изменение родительского ключа, у вас не будет выбора, делать или не делать этого. Это будет сделано обязательно.)

Одно из логических решений будет состоять в том, чтобы установить поле snum в его порядках в NULL, в следствии чего ни один продавец не получит комиссионные в этих порядках, пока комиссионые не будут предоставлены заказчикам для Peel. Затем вы можете удалить их из таблицы Продавцов:

```
UPDATE Orders
SET snum = NULL
WHERE snum = 1004;

UPDATE Cudomers
SET snum = 1001
WHERE snum = 1004;

DELETE FROM Salespeople
WHERE snum = 1004;
```

Если у вас проблема с удалением Motika (возможно, имеется другой внешний ключ, ссылающийся на него, о котором вы не знали и не учитывали), вы могли бы отменить все изменения, которые вы сделали, до тех пор, пока проблема не будет определена и решена.

Более того, это должна быть группа команд, чтобы обрабатывать ее как одиночную транзакцию. Вы можете предусмотреть это с помощью команды COMMIT, и завершить ее с помощью команды COMMIT или ROLLBACK.

КАК SQL ОБЩАЕТСЯ СРАЗУ СО МНОГИМИ ПОЛЬЗОВАТЕЛЯМИ

SQL часто используется в многопользовательских средах — в средах, где сразу много пользователей могут выполнять действия в базе данных одновременно. Это создает потенциальную возможность конфликта между различными выполняемыми действиями. Например, предположим что вы выполняете команду в таблице Продавцов:

```
UPDATE Salespeople
SET comm = comm * 2
WHERE sname LIKE 'R%';
```

и в это же время, Diane вводит такой запрос:

```
SELECT city, AVG (comm)
FROM Salespeople
GROUP BY city;
```

Может ли усредненное значение (AVG) Diane отразить изменения, которые вы делаете в таблице? Не важно, будет это сделано или нет, а важно, что бы были отражены или все, или ни одно из значений комиссионных (comm), для которых выполнялись изменения. Любой промежуточный результат является случайным или непредсказуемым, для порядка в котором значения были изменены физически. Вывод запроса не должен быть случайным и непредсказуемым.

Посмотрим на это с другой стороны. Предположим, что вы находите ошибку и прокручиваете обратно все ваши модификации уже после того, как Diane получила их результаты в виде вывода. В этом случае Diane получит ряд усредненых значений, основанных на тех изменениях, которые были позже отменены, не зная, что ее информации неточна.

Обработка одновременных транзакций называется *параллелизмом* или *совпадением* и имеет ряд возможных проблем, которые могут при этом возникать. Имеются следующие примеры:

* **Модификация может быть сделана без учета другой модификации.** Например, продавец должен сделать запрос к таблице инвентаризации, чтобы найти десять фрагментов пунктов торговцев акциями, и упорядочить шесть из них для заказчика. Прежде, чем это изменение было сделано, другой продавец делает запрос к таблице и упорядочивает семь из тех же фрагментов для своего заказчика.

ПРИМЕЧАНИЕ: Термин "упорядочить", аналогичен общепринятому — "заказать", что в принципе более соответствует логике запроса, потому что с точки зрения пользователя, он именно "заказывает" информацию в базе данных, которая упорядочивает эту информацию в соответствии с "заказом".

* **Изменения в базе данных могут быть прокручены обратно уже после того, как их действия уже были закончены.** Например если Вы отменили вашу ошибку уже после того, как Diane получила свой вывод.

* **Одно действие может воздействовать частично на результат другого действия.** Например когда Diane получает среднее от значений в то время как вы выполняете модификацию этих значений. Хотя это не всегда проблематично, в большинстве случаев действие такое же, как если бы агрегаты должны были отразить состояние базы данных в пункте относительной стабильности. Например в ревизионных книгах, должна быть возможность вернуться назад и найти это существующее усредненное значение для Diane в некоторой временной точке, и оставить его без изменений, которые могли быть сделаны начиная уже с этого места. Это будет невозможно сделать, если модификация была выполнена во время вычисления функции.

* **Тупик**. **Два пользователя могут попытаться выполнить действия, которые конфликтуют друг с другом.** Например, если два пользователя попробуют изменить и значение внешнего ключа и значение родительского ключа одновременно.

Имеется много сложнейших сценариев которые нужно было бы последовательно просматривать, если бы одновременные транзакции были неуправляемыми. К счастью, SQL обеспечивает вас средством *управления параллелизмом* для точного указания места получения результата. Что ANSI указывает для управления параллелизмом — это что все одновременные команды будут выполняться по принципу — ни одна команда не должна быть выдана, пока предыдущая не будет завершена (включая команды COMMIT или ROLLBACK).

Более точно, нужно просто не позволить таблице быть доступной более чем для одной транзакции в данный момент времени. Однако в большинстве ситуаций, необходимость иметь базу данных доступную сразу многим пользователям, приводит к некоторому компромису в управлении параллелизмом. Некоторые реализации SQL предлагают пользователям выбор, позволяя им самим находить золотую середину между согласованностью данных и доступностью к базе данных. Этот выбор доступен пользователю, DBA, или тому и другому.

На самом деле они осуществляют это управление вне SQL, даже если и воздействуют на процесс работы самой SQL.

Механизм, используемый SQL для управления параллелизмом операций, называется *блокировкой*. Блокировки задерживают определенные операции в базе данных, пока другие операции или транзакции не завершены. Задержанные операции выстраиваются в очередь и выполняются только когда блокировка снята (некоторые инструменты блокировок дают вам возможность указывать **NOWAIT**, которая будет отклонять команду вместо того, чтобы поставить ее в очередь, позволяя вам делать что-нибудь другое).

Блокировки в многопользовательских системах необходимы. Следовательно, должен быть некий тип схемы *блокировки по умолчанию*, который мог бы применяться ко всем командам в базе данных. Такая схема *по умолчанию*, может быть определена для всей базы данных, или в качестве параметра в команде CREATE DBSPACE или команде ALTER DBSPACE, и таким образом использовать их по разному в различных DBS.

Кроме того, системы обычно обеспечиваются неким типом *обнаружителя зависания*, который может обнаруживать ситуации, где две операции имеют блокировки, блокирующие друг друга. В этом случае, одна из команд будет прокручена обратно и получит сброс блокировки.

Так как терминология и специфика схем блокировок меняются от программы к программе, мы можем смоделировать наши рассуждения на примере программы базы данных **DB2** фирмы IBM. IBM — лидер в этой области (как впрочим и во многих других), и поэтому такой подход наиболее удобен. С другой стороны, некоторые реализации могут иметь значительные различия в синтаксисе и в функциях, но в основном их действия должно быть очень похожими.

ТИПЫ БЛОКИРОВОК

Имеется два базовых типа блокировок:

— ***распределяемые блокировки*** и

— ***специальльные блокировки***.

Распределяемые (или *S-блокировки*) могут быть установлены более чем одним пользователем в данный момент времени. Это дает возможность любому числу пользователей обращаться к данным, но не изменять их.

Специальные блокировки (или *X-блокировки*) не позволяют никому вообще, кроме владельца этой блокировки обращаться к данным. *Специальные* блокировки используются для команд, которые изменяют содержание или структуру таблицы. Они действуют до конца транзакции.

Общие блокировки используются для запросов. Насколько они продолжительны, зависит фактически от уровня изоляции.

Что такое *уровень изоляции* блокировки? Это — то, что определяет, сколько таблиц будет блокировано.

В DB2, имеется три уровня изоляции, два из которых можно применить и к распределеным и к специаль*ным* блокировкам, а третий, *ограниченный*, чтобы использовать эти блокировки совместно. Они управляются командами, поданными извне SQL, так что мы можем обсуждать не указывая их точного синтаксиса. Точный синтаксис команд, связанных с блокировками, различен для различных реализаций.

Следующее обсуждение полезно прежде всего на концептуальном уровне.

Уровень изоляции — *повторное чтение* — гарантирует, что внутри данной транзакции все записи, извлеченные с помощью запросов, не могут быть изменены. Поскольку записи, модифицируемые в транзакции, являются субъектами специальной

блокировки, пока транзакция не завершена, они не могут быть изменены в любом случае.

С другой стороны, для запросов *повторное чтение* означает, что вы можете решить заранее, какие строки вы хотите заблокировать и выполнить запрос, который их выберет. Выполняя запрос, вы гарантированы, что никакие изменения не будут сделаны в этих строках, до тех пор пока вы не завершите текущую транзакцию.

В то время как *повторное чтение* защищает пользователя, который поместил блокировку, она может в то же время значительно снизить производительность.

Уровень *указатель стабильности* — предохраняет каждую запись от изменений, на время когда она читается, или от чтения на время ее изменения. Последний случай — это *специальная блокировка*, и применяется, пока изменение не завершено или пока оно не отменено (т.е. на время отката изменения).

Следовательно, когда вы модифицируете группу записей, использующих *указатель стабильности*, эти записи будут заблокированы, пока транзакция не закончится, что аналогично действию, производимому уровнем *повторное чтение*. Различиие между этими двумя уровнями в их воздействии на запросы. В случае уровня *указатель стабильности*, строки таблицы, которые в данное время не используются запросом, могут быть изменены.

Третий уровень изоляции DB2 — это уровень *только чтение*.

Только чтение фиксирует фрагмент данных; хотя на самом деле он блокирует всю таблицу. Следовательно, он не может использоваться с командами модификации. Любое содержание таблицы как единое целое, в момент выполнения команды, будет отражено в выводе запроса.

Это не обязательно, так как в случае с уровнем *указатель стабильности*. Блокировка *только чтение*, гарантирует что ваш вывод будет внутренне согласован, если конечно нет необходимости во второй блокировке, не связывающей большую часть таблицы с уровнем *повторное чтение*. Блокировка т*олько чтение* удобна тогда, когда вы делаете отчеты, которые должны быть внутренне согласованны, и позволять доступ к большинству или ко всем строкам таблицы, не связывая базу данных.

ДРУГИЕ СПОСОБЫ БЛОКИРОВКИ ДАННЫХ

Некоторые реализации выполняют блокировку страницы вместо блокировки строки. Это может быть либо возможностью для вашего управления, либо нечто заложенным уже в конструкцию системы.

Страница — это блок накопления памяти, обычно равный 1024 байт. Страница может состоять из одной или более строк таблицы, возможно сопровождаемых индексами и другой периферийной информацией, а может состоять даже из нескольких строк другой таблицы. Если вы блокируете страницы вместо строк, все данные в этих страницах будут блокированы точно также как и в индивидуальных строках, согласно уровням изоляции описаным выше.

Основным преимуществом такого подхода является эффективность. Когда SQL не следит за блокированностью и разблокированностью строк индивидуально, он работает быстрее. С другой стороны, язык SQL был разработан так, чтобы максимизировать свои возможности, и произвольно блокирует строки, которые необязательно было блокировать.

Похожая возможность, доступная в некоторых системах — это *блокировка областей DBS*. Области базы данных имеют тенденцию быть больше, чем страница, так что этот подход удовлетворяет и достоинству увеличения производительности, и недостатку блокирования страниц.

Вообще то лучше отключать блокировку низкого уровня, если вам кажется, что появились значительные проблемы с эффективностью.

РЕЗЮМЕ

Ключевые определения, с которыми вы познакомились в этой главе:

* *Синонимы*, или как создавать новые имена для объектов данных.

* Области базы даных (**DBS**), или как распределяется доступная память в базе данных.

* *Транзакция*, или как сохранять или восстанавливать изменения в базе данных.

* *Управление Параллелизмом*, или как SQL предохраняет от конфликта одной команды с другой.

Синонимы — это объекты, в том смысле, что они имеют имена и (иногда) владельцев, но естественно они не могут существовать без таблицы, чье имя они замещают. Они могут быть общими и следовательно доступными каждому кто имеет доступ к объекту, или они могут принадлежать определенному пользователю.

Области DBS или просто *DBS* — это подразделы базы данных, которые распределены для пользователей. Связанные таблицы, (например таблицы, которые будут часто объединяться) лучше хранить в общей для них DBS.

COMMIT и **ROLLBACK** — это команды, используемые для выполнения изменений в базе данных, в то время когда предыдущая команда COMMIT или команда ROLLBACK, начинают сеанс и оставляют изменения, или игнорируют их как группу.

Средство Уп*равление Параллелизмом* — определяет, в какой степени одновременно поданные команды будут мешать друг другу. Оно является адаптируемым средством, находящим компромис между производительностью базы данных и изоляцией действующих команд.

РАБОТА С SQL

1. Создайте область базы данных с именем Myspace которая выделяет 15 процентов своей области для индексов, и 40 процентов на расширение строк.
2. Вы получили право SELECT в таблице Порядков продавца Diane. Введите команду так чтобы вы могли ссылаться к этой таблице как к "Orders" не используя имя "Diane" в качестве префикса.
3. Если произойдет сбой питания, что случится с всеми изменениями сделанными во время текущей транзакции?
4. Если вы не можете видеть строку из-за ее блокировки, какой это тип блокировки?
5. Если вы хотите получить общее, максимальное, и усредненое значения сумм приобретений для всех порядков, и не хотите при этом запрещать другим пользоваться таблицей, какой уровень изоляции будет этому соответствовать?

(См. Приложение A для ответов.)

24

КАК ДАННЫЕ SQL СОДЕРЖАТСЯ В УПОРЯДОЧЕННОМ ВИДЕ

В ЭТОЙ ГЛАВЕ, ВЫ УЗНАЕТЕ КАК ТИПОВАЯ SQL БАЗА данных сохраняет самоорганизованность. Не удивительно, что самоорганизованность обеспечивается реляционной базой данных, создаваемой и поддерживаемой с помощью программы. Вы можете обращаться к этим таблицам самостоятельно для получения информации о привилегиях, таблицах, индексах, и так далее. Эта глава покажет вам некоторые типы, содержащиеся в такой базе данных.

КАТАЛОГ СИСТЕМЫ

Чтобы функционировать как SQL база данных, ваша компьютерная система должна следить за многими различными вещями: таблицами, представлениями, индексами, синонимами, привилегиями, пользователями, и так далее. Имеются различные способы делать это, но ясно, что наиболее логичный, эффективный, и согласованный способ делать это в реляционной среде состоит в том, чтобы сохранять эту информацию в таблицах. Это дает возможность компьютеру размещать и управлять информацией в которой он нуждается, используя те же самые процедуры, которые он использует, чтобы размещать и управлять данными, которые он хранит для вас. Хотя это — вопрос конкретной программы, а не часть стандарта ANSI, большинство SQL баз данных используют набор SQL таблиц, хранящих служебную информацию для своих внутренних потребностей. Этот набор называется в различных публикациях как *системный каталог*, *словарь данных*, или просто *системные таблицы* (Термин "*словарь данных*" может также относится к общему архиву данных, включая информацию о физических параметрах базы данных которые хранятся вне SQL.

Следовательно, имеются программы баз данных, которые имеют и системный каталог и словарь данных.)

Таблицы системного каталога — напоминают обычные SQL таблицы: те же строки и столбцы данных. Например, одна таблица каталога обычно содержит информацию о таблицах, существующих в базе данных, по одной строке на каждую таблицу базы данных; другая содержит информацию о различных столбцах таблиц, по одной строке на столбец, и так далее. Таблицы каталога создаются и присваиваются с помощью самой базы данных и идентифицируются с помощью специальных имен, таких, например, как **SYSTEM**. База данных создает эти таблицы и модифицирует их автоматически; таблицы каталога не могут быть непосредственно подвергнуты действию команды модификации.

Если это случится, это значительно запутает всю систему и сделает ее неработоспособной. Однако, в большинстве систем, каталог может быть запрошен пользователем. Это очень полезно, потому что это дает вам возможность узнать кое-что о базе данных, которую вы используете. Конечно, вся информация не всегда доступна всем пользователям. Подобно другим таблицам, доступ к каталогу ограничен для пользователей без соответствующих привилегий.

Так как каталог принадлежит самой системе, имеется некоторая неясность относительно того, кто имеет привилегии и кто может предоставить привилегии в этом каталоге. Обычно, привилегии каталога предоставляет суперпользователь, например, администратор системы, зарегистрированый как **SYSTEM** или **DBA**. Кроме того, некоторые привилегии могут предоставляться пользователям автоматически.

ТИПИЧНЫЙ СИСТЕМНЫЙ КАТАЛОГ

Давайте рассмотрим некоторые таблицы, которые мы могли бы найти в типовом каталоге системы:

Таблицы	Содержание
SYSTEMCATALOG	Таблицы (базовые и представления)
SYSTEMCOLUMNS	Столбцы таблицы
SYSTEMTABLES	Каталог Представления в SYSTEMCATALOG
SYSTEMINDEXES	Индексы в таблице
SYSTEMUSERAUTH	Пользователи базы данных
SYSTEMTABAUTH	Объектные привилегии пользователей
SYSTEMCOLAUTH	Столбцовые привилегии пользователей
SYSTEMSYNONS	Синонимы для таблиц

Теперь, если наш DBA предоставит пользователю Stephen право просматривать SYSTEMCATALOG такой командой,

```
GRANT SELECT ON SYSTEMCATALOG TO Stephen;
```

то Stephen сможет увидеть некоторую информацию обо всех таблицах в базе данных (мы имеем здесь пользователя DBA, пользователя Chris, владельца трех наших типовых таблиц, а также Adrian, владельца представления Londoncust).

```
SELECT tname, owner, numcolumns, type, CO
FROM SYSTEMCATALOG;
```

```
 ===============  SQL Execution Log ===============
| SELECT tname, owner, numcolumns, type, CO         |
| FROM  SYSTEMCATALOG;                              |
| ================================================= |
|   tname          owner    numcolumns  type  CO    |
| -------------   -------   ----------  ----  ---   |
| SYSTEMCATALOG   SYSTEM            4     B         |
| Salespeople     Chris             4     B         |
| Customers       Chris             5     B         |
| Londoncust      Adrian            5     V    Y    |
| Orders          Chris             5     B         |
 =================================================
```

Рисунок 24.1: Содержание таблицы SYSTEMCATALOG

Как вы можете видеть, каждая строка описывает свою таблицу. Первый столбец — имя; второй — имя пользователя который владеет ею; третий — число столбцов которые содержит таблица; и четвертый — код из одного символа, это или B (для базовой таблицы) или V (для представления). Последний столбец имеет пустые (NULL) значения, если его тип не равен V; и этот столбец указывает, определена или нет возможность проверки.

Обратите внимание, что **SYSTEMCATALOG** (*СИСТЕМНЫЙ КАТАЛОГ*) представлен как одна из таблиц в вышеуказаном списке. Для простоты, мы исключили остальные каталоги системы из вывода. Таблицы системного каталога обычно показываются в **SYSTEMCATALOG**.

ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ В ТАБЛИЦАХ КАТАЛОГА

Поскольку **SYSTEMCATALOG** — это таблица, вы можете использовать ее в представлении. Фактически можно считать, что имеется такое представление с именем **SYSTEMTABLES**.

Это представление SYSTEMCATALOG содержит только те таблицы, которые входят в системный каталог; это обычно таблицы базы данных, типа таблицы Продавцов, которые показаны в SYSTEMCATALOG, но не в SYSTEMTABLES.

Давайте предположим, что только таблицы каталога являются собственностью пользователя SYSTEM. Если вы захотите, вы можете определить другое представление, которое бы специально исключало таблицы каталога из вывода:

```
CREATE VIEW Datatables
AS SELECT *
   FROM SYSTEMCATALOG
   WHERE owner <> 'SYSTEM';
```

РАЗРЕШИТЬ ПОЛЬЗОВАТЕЛЯМ ВИДЕТЬ (ТОЛЬКО) ИХ СОБСТВЕННЫЕ ОБЪЕКТЫ

Кроме того, имеются другое использование представлений каталога. Предположим вам нужно чтобы каждый пользователь был способен сделать запрос каталога, для получения информации только из таблиц которыми он владеет. Пока значение USER, в команде SQL постоянно для ID доступа пользователя выдающего команду, оно может всегда быть использоваться, чтобы давать доступ пользователям только к их собственным таблицам.

Вы можете, для начала создать следующее представление:

```
CREATE VIEW Owntables
AS SELECT *
   FROM SYSTEMCATALOG
   WHERE Owner = USER;
```

Теперь вы можете предоставить всем пользователям доступ к этому представлению:

```
GRANT SELECT ON Owntables TO PUBLIC;
```

Каждый пользователь теперь способен *выбирать* (**SELECT**) только те строки из SYSTEMCATALOG, владельцем которых он сам является.

ПРЕДСТАВЛЕНИЕ SYSTEMCOLUMNS Одно небольшое добавление к этому позволит каждому пользователю просматривать таблицу SYSTEMCOLUMNS для столбцов из его собственных таблиц. Сначала, давайте рассмотрим ту часть таблицы SYSTEMCOLUMNS, которая описывает наши типовые таблицы (другими словами, исключим сам каталог):

tname	***cname***	***datatype***	***cnumber***	***tabowner***
Salespeople	snum	integer	1	Diane
Salespeople	sname	char	2	Diane
Salespeople	city	char	3	Diane
Salespeople	comm	decimal	4	Diane
Customers	cnum	integer	1	Claire

Customers	cname	char	2	Claire
Customers	city	char	3	Claire
Customers	rating	integer	4	Claire
Customers	snum	integer	5	Claire
Orders	onum	integer	1	Diane
Orders	odate	date	2	Diane
Orders	amt	decimal	3	Diane
Orders	cnum	integer	4	Diane
Orders	snum	integer	5	Diane

Как вы можете видеть, каждая строка этой таблицы показывает столбец таблицы в базе данных. Все столбцы данной таблицы должны иметь разные имена, также как каждая таблица должна иметь данного пользователя, и наконец все комбинации пользователей, таблиц, и имен столбцов должны различаться между собой.

Следовательно табличные столбцы: tname (имя таблицы), tabowner (владелец таблицы), и cname (имя столбца), вместе составляют первичный ключ этой таблицы. Столбец datatype (тип данных) говорит сам за себя. Столбец cnumber (номер столбца) указывает на местоположение этого столбца внутри таблицы. Для упрощения, мы опустили параметры длины столбца, точности, и масштаба.

Для справки, показана строка из SYSTFMCATALOG, которая ссылается к этой таблице:

tname	***owner***	***numcolumns***	***type***	***CO***
SYSTEMCOLUMNS	System	8	B	

Некоторые SQL реализации будут обеспечивать вас большим количеством данных, чем показано в этих столбцах, но показанное являются основой для любой реализации.

Для иллюстрации процедуры, предложенной в начале этого раздела, имеется способ, позволяющий каждому пользователю видеть информацию SYSTEMCOLUMNS только для принадлежащих ему таблиц:

```
CREATE VIEW Owncolumns
AS SELECT *
   FROM SYSTEMCOLUMNS
   WHERE tabowner = USER;

GRANT SELECT ON Owncolumns TO PUBLIC;
```

КОММЕНТАРИЙ В СОДЕРЖАНИИ КАТАЛОГА

Большинство версий SQL позволяют вам помещать комментарии (ремарки) в специальные столбцы пояснений таблиц каталогов **SYSTEMCATALOG** и **SYSTEMCOLUMNS**, что удобно, так как эти таблицы не всегда могут объяснить свое содержание. Для простоты, мы пока исключали этот столбец из наших иллюстраций.

Можно использовать команду **COMMENT ON** со строкой текста, чтобы пояснить любую строку в одной из этих таблиц. Состояние — *TABLE*, для комментирования в SYSTEMCATALOG, и текст — *COLUMN*, для SYSTEMCOLUMNS.

Например:

```
COMMENT ON TABLE Chris.Orders
IS 'Current Customer Orders';
```

Текст будет помещен в столбец пояснений SYSTEMCATALOG. Обычно, максимальная длина таких пояснений — 254 символа.

Сам комментарий, указывается для конкретной строки, одна с tname=Orders, а другая owner=Chris. Мы увидим этот комментарий в строке для таблицы Порядков в SYSTEMCATALOG:

```
SELECT tname, remarks
FROM SYSTEMCATALOG
WHERE tname = 'Orders' AND owner = 'Chris';
```

Вывод для этого запроса показывается в Рисунке 24.2.

```
===============  SQL Execution Log ===============
| SELECT tname, remarks                            |
| FROM  SYSTEMCATALOG                              |
| WHERE tname = 'Orders'                           |
| AND owner = 'Chris'                              |
| ;                                                |
| ================================================ |
|   tname          remarks                         |
| -------------   ----------------------          |
| Orders          Current Customers Orders         |
 ================================================
```

Рисунок 24.2: Коментарий в SYSTEMCATALOG

SYSTEMCOLUMNS работает точно так же. Сначала, мы создаем комментарий:

```
COMMENT ON COLUMN Orders.onum
IS 'Order #';
```

затем выбираем эту строку из SYSTEMCOLUMNS:

```
SELECT cnumber, datatype, cname, remarks
FROM SYSTEMCOLUMNS
WHERE tname = 'Orders' AND tabowner = 'Chris' AND cname = onum;
```

Вывод для этого запроса показывается в Рисунке 24.3.

Чтобы изменить комментарий, вы можете просто ввести новую команду **COMMENT ON** для той же строки. Новый комментарий будет записан поверх старого. Если вы хотите удалить комментарий, напишите поверх него пустой комментарий, подобно следующему:

```
COMMENT ON COLUMN Orders.onum
IS '';
```

и этот пустой комментарий затрет предыдущий.

```
 ===============  SQL Execution Log ===============
| SELECT cnumber, datatype, cname, remarks          |
| FROM  SYSTEMCOLUMNS                               |
| WHERE tname = 'Orders'                            |
| AND tabowner = 'Chris'                            |
| AND cname = 'onum'                                |
| ;                                                 |
| ================================================= |
|   cnumber      datatype    cname    remarks       |
| ----------     ---------   ------  ------------   |
|          1     integer     onum    Orders #       |
 =================================================
```

Рисунок 24.3: Коментарий в SYSTEMCOLUMNS

ОСТАЛЬНОЕ ИЗ КАТАЛОГА

Здесь показаны оставшиеся из ваших системных таблиц определения, с типовым запросом для каждого:

SYSTEMINDEXES — ИНДЕКСАЦИЯ В БАЗЕ ДАННЫХ

Имена столбцов в таблице SYSTEMINDEXES и их описания — следующие:

СТОЛБЦЫ	***ОПИСАНИЕ***
iname	Имя индекса (используемого для его исключения)
iowner	Имя пользователя который создал индекс
tname	Имя таблицы которая содержит индекс
cnumber	Номер столбца в таблице
tabowner	Пользователь который владеет таблицей содержащей индекс
numcolumns	Число столбцов в индексе
cposition	Позиция текущего столбца среди набора индексов
isunique	Уникален ли индекс (Y или N)

ТИПОВОЙ ЗАПРОС Индекс считается неуникальным, если он вызывает продавца, в snum столбце таблицы Заказчиков:

```
SELECT iname, iowner, tname, cnumber, isunique
FROM SYSTEMINDEXES
WHERE iname = 'salesperson';
```

Вывод для этого запроса показывается в Рисунке 24.4.

```
 ===============  SQL Execution Log ================
| SELECT iname, iowner, tname, cnumber, isunique     |
| FROM  SYSTEMINDEXES                                |
| WHERE iname = 'salespeople'                        |
| ;                                                  |
| ================================================== |
|   iname       iowner   tname      cnumber  isunique |
| -----------  ------  ----------  -------  -------- |
| salesperson  Stephan  Customers        5   N       |
 ==================================================
```

Рисунок 24.4: Строка из таблицы SYSTEMINDEXES

SYSTEMUSERAUTH — ПОЛЬЗОВАТЕЛЬСКИЕ И СИСТЕМНЫЕ ПРИВИЛЕГИИ В БАЗЕ ДАННЫХ

Имена столбцов для SYSTEMUSERAUTH и их описание, следующее:

СТОЛБЦЫ	***ОПИСАНИЕ***
username	Идентификатор (ID) доступа пользователя
password	Пароль пользователя вводимый при регистрации
resource	Где пользователь имеет права RESOURCE
dba	Где пользователь имеет права DBA

Мы будем использовать простую схему системных привилегий, которая представлена в Главе 22, где были представлены три системных привилегии — **CONNECT** (*ПОДКЛЮЧИТЬ*), **RESOURCE** (*РЕСУРСЫ*) и **DBA**.

Все пользователи получают CONNECT по умолчанию при регистрации, поэтому он не описан в таблице выше. Возможные состояния столбцов **resource** и **dba** могут быть — **Y** (Да, пользователь имеет привилегии) или — **No** (Нет, пользователь не имеет привилегий).

Пароли (**password**) доступны только высоко привилегированным пользователям, если они существуют. Следовательно, запрос этой таблицы можно вообще делать только для информации относительно привилегий системы и пользователей.

ТИПОВОЙ ЗАПРОС Чтобы найти всех пользователей, которые имеют привилегию RESOURCE, и увидеть какие из них — DBA, вы можете ввести следующее условие:

```
SELECT username, dba
FROM SYSTEMUSERAUTH
WHERE resource = 'Y';
```

Вывод для этого запроса показывается в Рисунке 24.5.

SYSTEMTABAUTH — ПРИВИЛЕГИИ ОБЪЕКТА, КОТОРЫЕ НЕ ОПРЕДЕЛЯЮТ СТОЛБЦЫ

Здесь показаны имена столбцов в таблице **SYSTEMTABAUTH** и их описание:

COLUMN	***ОПИСАНИЕ***
username	Пользователь, который имеет привилегии
grantor	Пользователь, который передает привилегии по имени пользователя
tname	Имя таблицы, в которой существуют привилегии
owner	Владелец tname
selauth	Имеет ли пользователь привилегию SELECT
insauth	Имеет ли пользователь привилегию INSERT
delauth	Имеет ли пользователь привилегию DELETE

Возможные значения для каждой из перечисленных привилегий объекта (имена столбцов которых оканчиваются на auth) — Y, N, и G. G указывает, что пользователь имеет привилегию с возможностью передачи привилегий. В каждой строке, по крайней мере один из этих столбцов должен иметь состояние отличное от N (другими словами, иметь хоть какую-то привилегию).

```
 ===============  SQL Execution Log ================
| SELECT username, dba                               |
| FROM  SYSTEMUSERAUTH                               |
| WHERE resource = 'Y'                               |
| ;                                                  |
| ================================================== |
|   username    dba                                  |
| -----------  ------                                |
|  Diane         N                                   |
|  Adrian        Y                                   |
 ==================================================
```

Рисунок 24.5: Пользователи которые имеют привилегию RESOURCE

Первые четыре столбца этой таблицы составляют первичный ключ. Это означает что каждая комбинация из tname, владелец-пользователь (не забудьте, что две различные таблицы с различными владельцами могут иметь одно и тоже имя), пользователь и пользователь передающий права (*гарантор*), должна быть уникальной. Каждая строка этой таблицы содержит все привилегии (которые не являются определенным столбцом), предоставляются одним определенным пользователем другому определенному пользователю в конкретном объекте.

UPDATE и **REFERENCES** являются привилегиями, которые могут быть определенными столбцами, и находиться в различных таблицах каталога. Если пользователь получает привилегии в таблице от более чем одного пользователя, такие привилегии могут быть отдельными строками, созданными в этой таблице. Это необходимо для каскадного отслеживания при вызове привилегий.

ТИПОВОЙ ЗАПРОС Чтобы найти все привелегии SELECT, INSERT и DELETE, которые Adrian предоставляет пользователям в таблице Заказчиков, вы можете ввести следующее (вывод показан в Рисунке 24.6):

```
SELECT username, selauth, insauth, delauth
FROM SYSTEMTABAUTH
WHERE grantor = 'Adrian' AND tname = 'Customers';
```

```
 ===============  SQL Execution Log ================
| SELECT username, selauth, insauth, delauth         |
| FROM  SYSTEMTABAUTH                                |
| WHERE grantor = 'Adrian'                           |
| AND tname = 'Customers'                            |
| ;                                                  |
| ================================================== |
|   username   selauth  insauth  delauth             |
| -----------  -------  -------- --------            |
|  Claire      G        Y        N                   |
|  Norman      Y        Y        Y                   |
 ==================================================
```

Рисунок 24.6: Пользователи получившие привилегии от Adrian

Выше показано, что Adrian предоставил Claire привилегии INSERT и SELECT в таблице Заказчиков, позднее предоставив ей права на передачу привилегий. Пользователю Norman он предоставил привелегии SELECT, INSERT и DELETE, но не дал возможность передачи привилегий ни в одной из них. Если Claire имела привилегию DELETE в таблице Заказчиков от какого-то другого источника, в этом запросе это показано не будет.

SYSTEMCOLAUTH

СТОЛБЦЫ	***ОПИСАНИЕ***
username	Пользователь который имеет привилегии
grantor	Пользователь который предоставляет привилегии другому пользователю
tname	Имя таблицы в которой существуют привилегии
cname	Имя столбца в котором существуют привилегии
owner	Владелец tname
updauth	Имеет ли пользователь привилегию UPDATE в этом столбце
refauth	Имеет ли пользователь привилегию REFERENCES в этом столбце

Столбцы updauth и refauth могут быть в состоянии Y, N, или G; но не могут быть одновременно в состоянии N для одной и той же строки. Это — первые пять столбцов таблицы, которы не составляют первичный ключ. Он отличается от первичного ключа SYSTEMTABAUTH в котором содержится поле cname, указывающее на определенный столбец обсуждаемой таблицы для которой применяются одна или обе привилегии. Отдельная строка в этой таблице может существовать для каждого столбца в любой данной таблицы в которой одному пользователю передаются превилегии определенного столбца с помощью другого пользователя.

Как и в случае с SYSTEMTABAUTH та же привилегия может быть описана в более чем одной строке этой таблицы если она была передана более чем одним пользователем.

ТИПОВОЙ ЗАПРОС Чтобы выяснить, в каких столбцах какой таблицы вы имеете привилегию REFERENCES, вы можете ввести следующий запрос (вывод показывается в Рисунке 24.7)

```
SELECT owner, tname, cname
FROM SYSTEMCOLAUTH
WHERE refauth IN ('Y', 'G') AND username = USER
ORDER BY 1, 2;
```

который показывает, что эти две таблицы, которые имеют различных владельцев, но одинаковые имя, в действительности, совершенно разные таблицы (т.е. это не как два синонима для одной таблицы).

```
 ===============  SQL Execution Log ================
| SELECT OWNER, TNAME, CNAME                        |
| FROM  SYSTEMCOLAUTH                               |
| WHERE refauth IN ('Y' , 'G' )                     |
| AND username = USER                               |
| ORDER BY 1, 2                                     |
| ;                                                 |
| ================================================= |
|   owner       tname       cname                   |
| -----------  -----------  -------                 |
|  Diane       Customers     cnum                   |
|  Diane       Salespeople   sname                  |
|  Diane       Salespeople   sname                  |
|  Gillan      Customers     cnum                   |
 ================================================
```

Рисунок 24.7: Столбцы в пользователь имеет привилегию INSERT

SYSTEMSYNONS — СИНОНИМЫ ДЛЯ ТАБЛИЦ В БАЗЕ ДАННЫХ

Это — имена столбцов в таблице SYSTEMSYNONS и их описание:

СТОЛБЕЦ	***ОПИСАНИЕ***
synonym	Имя синонима
synowner	Пользователь, который является владельцем синонима (может быть PUBLIC (ОБЩИЙ))
tname	Имя таблицы используемой владельцем
tabowner	Имя пользователя который является владельцем таблицы

ТИПОВОЙ ЗАПРОС Предположим, что Adrian имеет синоним Clients для таблицы Заказчиков, принадлежащей Diane, и что имеется общий синоним Customers для этой же таблицы. Вы делаете запрос таблицы для всех синонимов в таблице Заказчиков (вывод показывается в Рисунке 24.8):

```
SELECT *
FROM SYSTEMSYNONS
WHERE tname = 'Customers'
```

```
 ===============  SQL Execution Log ================
| SELECT *                                          |
| FROM  SYSTEMSYNONS                                |
| WHERE tname = 'Customers'                         |
| ;                                                 |
| ================================================= |
| synonym       synowner    tname       tabowner    |
| -----------  -----------  ----------  ----------  |
|  Clients      Adrian      Customers   Diane       |
|  Customers    PUBLIC      Customers   Diane       |
 ===================================================
```

Рисунок 24.8: Синонимы для таблицы Заказчиков

ДРУГОЕ ИСПОЛЬЗОВАНИЕ КАТАЛОГА

Конечно, вы можете выполнять более сложные запросы в системном каталоге. Обьединения, например, могут быть очень удобны. Эта команда позволит вам увидеть столбцы таблиц и базовые индексы, установленые для каждого (вывод показывается в Рисунке 24.9):

```
SELECT a.tname, a.cname, iname, cposition
FROM SYSTEMCOLUMNS a, SYSTEMINDEXES b
WHERE a.tabowner = b.tabowner AND a.tname = b.tname AND
      a.cnumber = b.cnumber
ORDER BY 3 DESC, 2;
```

Она показывает два индекса, один для таблицы Заказчиков и один для таблицы Продавцов. Последний из них — это одностолбцовый индекс с именем salesno в поле snum; он был помещен первым из-за сортировки по убыванию (в обратном алфавитном порядке) в столбце iname. Другой индекс, custsale, используется продавцами, чтобы отыскивать своих заказчиков. Он основывается на комбинации полей snum и cnum внутри таблицы Заказчиков, с полем snum приходящим в индексе первым, как это и показано с помощью поля cposition.

```
 ===============  SQL Execution Log ================
| SELECT a.tname, a.cname, iname, cposition          |
| FROM  SYSTEMCOLUMNS a, SYSTEMINDEXES b             |
| WHERE a.tabowner = b.tabowner                      |
| AND a.tname = b.tname                              |
| AND a.cnumber = b.cnumber                          |
| ORDER BY 3 DESC, 2;                                |
| ================================================== |
|    tname       cname     iname     cposition       |
|   -----------  -------  --------  ------------     |
|   Salespeople   sname   salesno              1     |
|   Customers     cnum    custsale             2     |
|   Customers     snum    custsale             1     |
 ==================================================
```

Рисунок 24.9: Столбцы и их индексы

Подзапросы также могут быть использованы. Имеется способ увидеть данные столбца только для столбцов из таблиц каталога:

```
SELECT *
FROM SYSTEMCOLUMNS
WHERE tname IN (SELECT tname
                FROM SYSTEMCATALOG);
```

Для простоты, мы не будем показывать вывод этой команды, которая состоит из одного входа для каждого столбца каждой таблицы каталога. Вы могли бы поместить этот запрос в представление, назвав его, например, SYSTEMTABCOLS, для представления SYSTEMTABLES.

РЕЗЮМЕ

Итак, система SQL использует набор таблиц, называемый системным каталогом в структуре базы данных. Эти таблицы могут запрашиваться но модифицироваться. Кроме того, вы можете добавлять комментарии столбцов в (и удалять их из) таблицы SYSTEMCATALOG и SYSTEMCOLUMNS. Создание представлений в этих таблицах — превосходный способ точно определить, какая пользовательская информация может быть доступной.

Теперь, когда вы узнали о каталоге, вы завершили ваше обучение SQL в диалоговом режиме. Следующая глава этой книги расскажет вам, как SQL используется в программах, которые написаны прежде всего на других языках, но которые способны извлечь пользу из возможностей SQL, взаимодействуя с его таблицами базы данных.

РАБОТА С SQL

1. Сделайте запрос каталога чтобы вывести, для каждой таблицы имеющей более чем четыре столбца, имя таблицы, имя владелеца, а также имя столбцов и тип данных этих столбцов.
2. Сделайте запрос каталога чтобы выяснить, сколько синонимов существует для каждой таблицы в базе данных. Не забудьте, что один и тот же синоним принадлежащий двум различным пользователям — это фактически два разных синонима.
3. Выясните сколько таблиц имеют индексы в более чем пятьдесяти процентов их столбцов.

(См. Приложение A для ответов.)

25

ИСПОЛЬЗОВАНИЕ SQL С ДРУГИМ ЯЗЫКОМ (ВЛОЖЕННЫЙ SQL)

В ЭТОЙ ГЛАВЕ ВЫ УЗНАЕТЕ КАК SQL ИСПОЛЬЗУЕТСЯ для расширения программ написанных на других языках. Хотя непроцедурность языка SQL делает его очень мощным, в то же время это накладывает на него большое число ограничений. Чтобы преодолеть эти ограничения, вы можете включать SQL в программы написанные на том или другом процедурном языке (имеющем определенный алгоритм). Для наших примеров, мы выбрали Паскаль, считая что этот язык наиболее прост в понимании для начинающих, и еще потому, что Паскаль — один из языков, для которых ANSI имеет *полуофициальный* стандарт.

ЧТО ТАКОЕ ВЛОЖЕНИЕ SQL

Чтобы вложить SQL в другой язык, вы должны использовать пакет программ, который бы обеспечивал поддержку вложения SQL в этот язык и конечно же, поддержку самого языка. Естественно, вы должны быть знакомы с языком, который вы используете. Главным образом, вы будете использовать команды SQL для работы в таблицах базы данных, передачи результатов вывода в программу и получения ввода из программы в которую они вкладываются, обобщенно ссылаясь к главной программе (которая может или не может принимать их из диалога или посылать обратно в диалог пользователя и программы).

ЗАЧЕМ ВКЛАДЫВАТЬ SQL?

Хотя и мы потратили некоторое время на то чтобы показать что умеет делать SQL, но если вы — опытный программист, вы вероятно отметили, что сам по себе, он не очень полезен при написании программ.

Самое очевидное ограничение — это то, что в то время как SQL может сразу выполнить пакет команды, интерактивный SQL в основном выполняет по одной команде в каждый момент времени.

Типы логических конструкций типа **if ... then** ("*если ... то*"), **for ... do** ("*для ... выполнить*") и **while ... repeat** ("*пока ... повторять*") — используемых для структур большинства компьютерных программ, здесь отсутствуют, так что вы не сможете принять решение — выполнять ли, как выполнять, или как долго выполнять одно действие в результате другого действия. Кроме того, интерактивный SQL не может делать многого со значениями, кроме ввода их в таблицу, размещения или распределения их с помощью запросов, и конечно вывода их на какое-то устройство.

Более традиционные языки, однако, сильны именно в этих областях. Они разработаны так, чтобы программист мог начинать обработку данных, и основываясь на ее результатах, решать, делать ли это действие или другое, или же повторять действие до тех пор, пока не встретится некоторое условие, создавая логические маршруты и циклы. Значения сохраняются в переменных, которые могут использоваться и изменяться с помощью любого числа команд. Это дает вам возможность указывать пользователям на ввод или вывод этих команд из файла, и возможность форматировать вывод сложными способами (например, преобразовывать числовые данные в диаграммы).

Цель вложенного SQL состоит в том, чтобы объединить эти возможности, позволяющие вам создавать сложные процедурные программы, которые адресуют базу данных посредством SQL — позволяя вам устранить сложные действия в таблицах на процедурном языке, который не ориентирован на такую структуру данных, в тоже время поддерживая структурную строгость процедурного языка.

КАК ДЕЛАЮТСЯ ВЛОЖЕНИЯ SQL

Команды SQL помещаются в исходный текст главной программы, которой предшествует фраза — **EXEC SQL** (*EXECute SQL*). Далее устанавливаются некоторые команды, которые являются специальными для вложенной формы SQL и которые будут представлены в этой главе.

Строго говоря, стандарт ANSI не поддерживает вложенный SQL как таковой. Он поддерживает понятие, называемое — *модуль*, который более точно, является вызываемым набором процедур SQL, а не вложением в другой язык. Официальное определение синтаксиса вложения SQL, будет включать расширение официального синтаксиса каждого языка в который может вкладываться SQL, что весьма долгая и неблагодарная задача, которую ANSI избегает. Однако, ANSI обеспечивает четыре приложения (не являющиеся частью стандарта), которые определяют синтаксис вложения SQL для четырех языков: КОБОЛ, ПАСКАЛЬ, ФОРТРАН, и ПЛ/1. Язык C — также широко поддерживается как и другие языки. Когда вы вставляете команды SQL в текст программы, написанной на другом языке, вы должны выполнить *предкомпиляцию* прежде, чем вы окончательно ее скомпилируете.

Программа, называемая *прекомпилятором* (или *препроцессором*), будет просматривать текст вашей программы и преобразовывать команды SQL в форму, удобную для использования базовым языком. Затем вы используете обычный транслятор, чтобы преобразовывать программу из исходного текста в выполняемый код.

Согласно подходу к модульному языку определенному ANSI, основная программа вызывает процедуры SQL. Процедуры выбирают параметры из главной программы и возвращают уже обработанные значения обратно в основную программу. Модуль может содержать любое число процедур, каждая из которых состоит из одиночной команды SQL. Идея в том, чтобы процедуры могли работать тем же самым способом, что и процедуры на языке, в который они были вложены (хотя модуль еще должен идентифицировать базовый язык из-за различий в типах данных различных языков). Реализации могут удовлетворить стандарту, выполнив вложение SQL таким способом, как если бы модули уже были точно определены. Для этой цели прекомпилятор будет создавать модуль, называемый *модулем доступа*. Только один модуль, содержащий любое число процедур SQL, может существовать для данной программы. Размещение операторов SQL непосредственно в главном коде происходит более просто и более практично, чем непосредственно создание самих модулей.

Каждая из программ, использующих вложение SQL, связана с **ID** доступа во время ее выполнения. ID доступа, связанный с программой, должен иметь все привилегии, чтобы выполнять операции SQL, выполняемые в программе. Вообще то, вложенная программа SQL регистрируется в базе данных, также как и пользователь, выполняющий программу. Более подробно, это определяет проектировщик, но вероятно было бы неплохо для включить в вашу программу команду CONNECT или ей подобную.

ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ ОСНОВНОГО ЯЗЫКА В SQL

Основной способ которым SQL и части базового языка ваших программ будут связываться друг с другом — это с помощью значений переменных. Естественно, что разные языки распознают различные типы данных для переменных. ANSI определяет эквиваленты SQL для четырех базовых языков — ПЛ/1, Паскаль, КОБОЛ, и ФОРТРАН; все это подробности описаны в Приложении B. Эквиваленты для других языков — определяет проектировщик.

Имейте в виду, что типы, такие как DATE, не распознаются ANSI; и следовательно никаких эквивалентных типов данных для базовых языков не существуют в стандарте ANSI. Более сложные типы данных базового языка, такие как матрицы, не имеют эквивалентов в SQL. Вы можете использовать переменные из главной программы во вложенных операторах SQL везде, где вы будете использовать выражения значений. (SQL, используемый в этой главе, будет пониматься как к вложенный SQL, до тех пор пока это не будет оговорено особо.)

Текущим значением переменной, может быть значение, используемое в команде. Главные переменные должны -

* быть обьявленными в **SQL DECLARE SESSION** (*РАЗДЕЛ ОБЪЯВЛЕНИЙ*), который будет описан далее.

* иметь совместимый тип данных с их функциями в команде SQL (например, числовой тип, если они вставляется в числовое поле)

* быть назначеными значению во время их использования в команде SQL, если команда SQL самостоятельно не может сделать назначение.

* предшествовать двоеточию (:) когда они упоминаются в команде SQL

Так как главные переменные отличаются от имен столбцов SQL наличием у них двоеточия, вы можете использовать переменные с теми же самыми именами, что и ваши столбцы, если это конечно нужно.

Предположим что вы имеете четыре переменных в вашей программе, с именами: **id_num**, **salesperson**, **loc** и **comm**. Они содержат значения, которые вы хотите вставить в таблицу Продавцов. Вы могли бы вложить следующую команду SQL в вашу программу:

```
EXEC SQL INSERT INTO Salespeople
VALUES (:id_num, :salesperson, :loc, :comm)
```

Текущие значения этих переменных будут помещены в таблицу. Как вы можете видеть, переменная comm имеет то же самое имя что и столбец в который это значение вкладывается.

Обратите внимание, что точка с запятой в конце команды отсутствует. Это потому, что соответствующее завершение для вложенной команды SQL зависит от языка для которого делается вложение.

Для Паскаля и PL/1 это будет точка с запятой, для КОБОЛА — слово **END-EXEC**, а для ФОРТРАНА не будет никакого завершения.

В других языках это зависит от реализации, и поэтому мы договоримся что будем использовать точку с запятой (в этой книге) всегда, чтобы не противоречить интерактивному SQL и Паскалю. Паскаль завершает вложенный SQL и собственные команды одинаково — точкой с запятой.

Способ сделать команду полностью такой как описана выше, состоит в том, чтобы включать ее в цикл и повторять ее, с различными значениями переменных, как например показано в следующем примере:

```
while not end-ot-file (input) do
   begin
   readln(id_num, salesperson, loc, comm);
   EXEC SOL INSERT INTO Salespeople
        VALUES (:id_num, :salesperson, :loc, :comm);
   end;
```

Фрагмент программы на ПАСКАЛЕ, определяет цикл, который будет считывать значения из файла, сохранять их в четырех проименованных переменных, сохранять значения этих переменных в таблице Продавцов, и затем считывать следующие четыре значения, повторяя этот процесс до тех пор, пока весь входной файл не прочитается. Считается, что каждый набор значений завершается возвратом каретки (для

незнакомых с Паскалем, функция **readln** считывает вводимую информацию и переходит на следующую строку источника этой информации). Это дает вам простой способ передать данные из текстового файла в реляционную структуру.

Конечно, вы можете сначала обработать данные любыми возможными способами на вашем главном языке, например для исключения всех комиссионных ниже значения .12:

```
while not end-ot-file (input) do
  begin
  readln (id_num, salesperson, loc, comm);
  if comm >= .12 then EXEC SQL INSERT INTO Salespeople
                        VALUES (:id_num, :salesperson, :loc, :comm);
  end;
```

Только строки, которые встретят условие comm >= .12, будут вставлены в вывод. Это показывает, что можно использовать и циклы, и условия как нормальные для главного языка.

ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ

Все переменные, на которые имеется ссылка в предложениях SQL, должны сначала быть обьявлены в **SQL DECLARE SECTION** (*РАЗДЕЛЕ ОБЪЯВЛЕНИЙ*), использующем обычный синтаксис главного языка. Вы можете иметь любое число таких разделов в программе, и они могут размещаться где-нибудь в коде перед используемой переменной, подчиненной ограничениям, определенным в соответствии с главным языком. *Раздел объявлений* должен начинать и кончаться вложенными командами SQL — **BEGIN DECLARE SECTION** (*Начало Раздела Объявлений*) и **END DECLARE SECTION** (*Конец Раздела Объявлений*), которым предшествует, как обычно **EXEC SQL** (*Выполнить*).

Чтобы обьявить переменные используемые в предыдущем примере, вы можете ввести следующее:

```
EXEC SQL BEGIN DECLARE SECTION;
Var
  id-num:      integer;
  Salesperson: packed array (1..10) ot char;
  loc:         packed array (1..10) ot char;
  comm:        real;
EXEC SQL END DECLARE SECTION;
```

Для незнакомых с ПАСКАЛем, **Var** — это заголовок, который предшествует ряду обьявляемых переменных и упакованным (или распакованным) массивам, являющимся серией фиксированных переменных значений, различаемых с помощью номеров (например, третий символ loc будет loc(3)).

Использование точки с запятой после каждой переменной указывает на то, что это — Паскаль, а не SQL.

ИЗВЛЕЧЕНИЕ ЗНАЧЕНИЙ ПЕРЕМЕННЫХ

Кроме помещения значений переменных в таблицы используя команды SQL, вы можете использовать SQL, чтобы получать значения для этих переменных. Один из способов делать это — с помощью разновидности команды SELECT, которая содержит предложение **INTO**. Давайте вернемся к нашему предыдущему примеру и переместим строку Peel из таблицы Продавцов в наши переменные главного языка.

```
EXEC SQL SELECT snum, sname, city, comm
INTO :id_num, :salesperson, :loc, :comm
FROM Salespeople
WHERE snum = 1001;
```

Выбранные значения помещаются в переменные с упорядоченными именами указанными в предложении INTO. Разумееется, переменные с именами, указанными в предложении INTO, должны иметь соответствующий тип, чтобы принять эти значения, и должна быть своя переменная для каждого выбранного столбца.

Если не учитывать присутствие предложения INTO, то этот запрос похож на любой другой. Однако, предложение INTO добавляет значительное ограничение к запросу. Запрос должен извлекать не более одной строки. Если он извлекает много строк, все они не могут быть вставлены одновременно в одну и ту же переменную. Команда естественно потерпит неудачу. По этой причине, **SELECT INTO** должно использоваться только при следующих условиях:

* когда вы используете предикат, проверяющий значения, котороы как вы знаете, могут быть уникальны, как в этом примере. Значения которые, как вы знаете, могут быть уникальными — это те значения, которые имеют принудительное ограничение уникальности или уникальный индекс, как это говорилось в Главах 17 и 18.

* когда вы используете одну или более агрегатных функций и не используете GROUP BY.

* когда вы используете SELECT DISTINCT во внешнем ключе с предикатом, ссылающимся на единственное значение родительского ключа (обеспечивая вашей системе предписание справочной целостности), как в следующем примере:

```
EXEC SQL SELECT DISTINCT snum
INTO :salesnum
FROM Customers
WHERE snum = (SELECT snum
              FROM Salespeople
              WHERE sname = 'Motika');
```

Предпологалось, что Salespeople.sname и Salespeople.snum — это соответственно, уникальный и первичный ключи этой таблицы, а Customers.snum — это внешний ключ, ссылающийся на Salespeople.snum, и вы предполагали, что этот запрос произведет единственную строку.

Имеются другие случаи, когда вы точно знаете, что запрос должен произвести единственную строку вывода, но они мало известны и, в большинстве случаев, вы основываетесь на том, что ваши данные имеют целостность, которая не может быть предписана с помощью ограничений. Не полагайтесь на это! Вы создаете программу, которая, вероятно, будет использоваться в течение некоторого времени, и лучше всего проиграть ее, чтобы быть гарантированным в будущем от возможных отказов. Во всяком случае, нет необходимости группировать запросы, которые производут одиночные строки, поскольку SELECT INTO используется только для удобства.

Как вы увидите, вы можете использовать запросы выводящие многочисленные строки, используя курсор.

КУРСОР

Одна из сильных качеств SQL — это способность функционировать на всех строках таблицы, чтобы встретить определенное условие как блок-запись, не зная, сколько таких строк там может быть. Если десять строк удовлетворяют предикату, то запрос может вывести все десять строк. Если десять миллионов строк определены, все десять миллионов строк будут выведены. Это несколько затруднительно, когда вы попробуете связать это с другими языками. Как вы сможете назначать вывод запроса

для переменных, когда вы не знаете, как велик будет вывод? Решение состоит в том, чтобы использовать то, что называется *курсором*.

Вы вероятно знакомы с курсором, как с мигающей черточкой, которая отмечает вашу позицию на экране компьютера. Вы можете рассматривать SQL курсор как устройство, которое аналогично этому, отмечает ваше место в выводе запроса, хотя аналогия не полная.

Курсор — это вид переменной, которая связана с запросом. Значением этой переменной может быть каждая строка, которая выводится при запросе. Подобно главным переменным, курсоры должны быть обьявлены прежде, чем они будут использованы. Это делается командой DECLARE CURSOR, следующим образом:

```
EXEC SQL DECLARE CURSOR Londonsales FOR
SELECT *
FROM Salespeople
WHERE city = 'London';
```

Запрос не выполнится немедленно; он — только определяется. Курсор немного напоминает представление, в котором курсор содержит запрос, а содержание курсора напоминает любой вывод запроса, каждый раз когда курсор становится открытым. Однако, в отличие от базовых таблиц или представлений, строки курсора упорядочены: имеются первая, вторая... ...и последняя строка курсора. Этот порядок может быть произвольным, с явным управлением с помощью предложения ORDER BY в запросе или же по умолчанию следовать какому-то упорядочению, определяемому инструментально-определяемой схемой.

Когда вы находите точку в вашей программе, в которой вы хотите выполнить запрос, вы открываете курсор с помощью следующей команды:

```
EXEC SQL OPEN CURSOR Londonsales;
```

Значения в курсоре могут быть получены, когда вы выполняете именно эту команду, но не предыдущую команду DECLARE и не последующую команду FETCH. Затем, вы используете команду FETCH, чтобы извлечь вывод из этого запроса, по одной строке в каждый момент времени.

```
EXEC SQL FETCH Londonsales INTO :id_num, :salesperson, :loc, :comm;
```

Это выражение переместит значения из первой выбраной строки, в переменные. Другая команда **FETCH** выводет следующий набор значений. Идея состоит в том, чтобы поместить команду FETCH внутрь цикла, так чтобы выбрав строку, вы могли переместив набор значений из этой строки в переменные, возвращались обратно в цикл чтобы переместить следующий набор значений в те же самые переменные.

Например, возможно вам нужно, чтобы вывод выдавался по одной строке, спрашивая каждый раз у пользователя, хочет ли он продолжить чтобы увидеть следующую строку

```
Look_at_more:=True;
EXEC SQL OPEN CURSOR Londonsales;
while Look_at_more do
   begin
   EXEC SQL FETCH Londonsales
   INTO :id_num, :Salesperson, :loc, :comm;
   writeln (id_num, Salesperson, loc, comm);
   writeln ('Do you want to see more data? (Y/N)');
   readln (response);
   it response = 'N' then Look_at_more:=False
end;
EXEC SQL CLOSE CURSOR Londonsales;
```

В Паскале, знак **:=** означает "*является назначенным значением из*", в то время как **=** еще имеет обычное значение "*равно*". Функция **writeln** записывает ее вывод, и затем переходит к новой строке.

Одиночные кавычки вокруг символьных значений во втором writeln и в предложении **if ... then** — обычны для Паскаля, что случается при дубликатах в SQL.

В результате этого фрагмента, Булева переменная с именем **Look_at_more** должна быть установлена в состояние верно, открыт курсор, и введен цикл. Внутри цикла, строка выбирается из курсора и выводится на экран. У пользователя спрашивают, хочет ли он видеть следующую строку. Пока он не ответил **N** (Нет), цикл повторяется, и следующая строка значений будет выбрана.

Хотя переменные Look_at_more и ответ должны быть обьявлены как Булева переменная и символьная (char) переменная, соответственно, в разделе обьявлений переменных в Паскаля, они не должны быть включены в раздел обьявлений SQL, потому что они не используются в командах SQL.

Как вы можете видеть, двоеточия перед именами переменных не используются для не-SQL операторов. Далее обратите внимание, что имеется оператор **CLOSE CURSOR** соответствующий оператору **OPEN CURSOR**. Он, как вы поняли, освобождает курсор значений, поэтому запрос будет нужно выполнить повторно с оператором OPEN CURSOR, прежде чем перейти в выбору следующих значений.

Это необязательно для тех строк, которые были выбраны запросом после закрытия курсора, хотя это и обычная процедура.

Пока курсор закрыт, SQL не следит за тем, какие строки были выбраны. Если вы открываете курсор снова, запрос повторно выполняется с этой точки, и вы начинаете все сначала.

Этот пример не обеспечивает автоматический выход из цикла, когда все строки уже будут выбраны. Когда у FETCH нет больше строк, которые надо извлекать, он просто не меняет значений в переменных предложения INTO. Следовательно, если данные исчерпались, эти переменные будут неоднократно выводиться с идентичными значениями, до тех пор пока пользователь не завершит цикл, введя ответ — N.

SQL коды

Хорошо было бы знать, когда данные будут исчерпаны, так чтобы можно было сообщить об этом пользователю и цикл завершился бы автоматически. Это — даже более важно чем например знать что команда SQL выполнена с ошибкой. Переменная **SQLCODE** (называемая еще **SQLCOD** в ФОРТРАНе) предназначена чтобы обеспечить эту функцию. Она должна быть определена как переменная главного языка и должна иметь тип данных который в главном языке соответствует одному из точных числовых типов SQL, как это показано в Приложении B. Значение SQLCODE устанавливается каждый раз, когда выполняется команда SQL. В основном существуют три возможности:

1. Команда выполнилась без ошибки, но не произвела никакого действия. Для различных команд это выглядит по разному:
   а) Для SELECT, ни одна строка не выбрана запросом.
   б) Для FETCH, последняя строка уже была выбрана, или ни одной строки не выбрано запросом в курсоре.
   в) Для INSERT, ни одной строки не было вставлено (подразумевается, что запрос использовался, чтобы сгенерировать значения для вставки, и был отвергнут при попытке извлечения любой строки.
   г) Для UPDATE и DELETE, ни одна строка не ответила условию предиката, и следовательно никаких изменений сделано в таблице не будет.

В любом случае, будет установлен код SQLCODE = 100.

2. Команда выполнилась нормально, не удовлетворив ни одному из выше указанных условий. В этом случае, будет установлен код SQLCOD = 0.

3. Команда сгенерировала ошибку. Если это случилось, изменения сделанные к базе данных текущей транзакцией, будут восстановлены (см. Главу 23).

В этом случае будет установлен код SQLCODE = некоторому отрицательному числу, определяемому проектировщиком. Задача этого числа, идентифицировать проблему так точно, насколько это возможно. В принципе, ваша система должна быть снабжена подпрограммой, которая, в этом случае, должна выполниться, чтобы выдать для вас информацию, расшифровывающую значение негативного числа, определенного вашим проектировщиком. В этом случае некоторое сообщение об ошибке будет выведено на экран или записано в файл протокола, а программа в это время выполнит восстановление изменений для текущей транзакции, отключится от базы данных и выйдет из нее.

ИСПОЛЬЗОВАНИЕ SQLCODE ДЛЯ УПРАВЛЕНИЯ ЦИКЛАМИ

Теперь мы можем усовершенствовать наш предыдущий пример для выхода из цикла автоматически, при условии что курсор пуст, все строки выбраны, или произошла ошибка:

```
Look_at_more:=True;
EXEC SQL OPEN CURSOR Londonsales;
while Look_at_more and SQLCODE = O do
   begin
   EXEC SQL FETCH Londonsales
   INTO :id_num, :Salesperson, :loc, :comm;
   writeln (id_num, Salesperson, loc, comm);
   writeln ('Do you want to see more data? (Y/N)');
   readln (response);
   If response = 'N' then Look_at_more:=False;
   end;
EXEC SQL CLOSE CURSOR Londonsales;
```

ПРЕДЛОЖЕНИЕ WHENEVER

Это удобно для выхода при выполненом условии — все строки выбраны. Но если вы получили ошибку, вы должны предпринять нечто такое, что описано для третьего случая, выше. Для этой цели, SQL предоставляет предложение **GOTO**. Фактически, SQL позволяет вам применять его достаточно широко, так что программа может выполнить команду GOTO автоматически, если будет произведено определенное значение SQLCODE. Вы можете сделать это совместно с предложением **WHENEVER**. Имеется кусок из примера для этого случая:

```
EXEC SQL WHENEVER SQLERROR GOTO Error_handler;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

SQLERROR — это другой способ сообщить, что SQLCODE < 0; а **NOT FOUND** — это другой способ сообщить, что SQLCODE = 100. (Некоторые реализации называют последний случай еще как — **SQLWARNING**.)

Error_handler — это имя того места в программе, в которое будет перенесено выполнение программы, если произошла ошибка (GOTO может состоять из одного или двух слов). Такое место определяется любым способом, соответствующим для главного языка, например, с помощью метки в Паскале или имени раздела или имени параграфа в КОБОЛЕ (в дальнейшем мы будем использовать термин — *метка*). *Метка* более удачно идентифицирует стандартную процедуру распространяемую проектировщиком для включения во все программы.

CONTINUE не делает чего-то специального для значения SQLCODE. Оно также является значением по умолчанию, если вы не используете команду **WHENEVER**, определяющую значение SQLCODE. Однако, эти неактивные определения дают вам возможность переключаться вперед и назад, выполняя и не выполняя действия, в различных точках (метках) вашей программы. Например, если ваша программа включает в себя несколько команд INSERT, использующих запросы, которые реально должны производить значения, вы могли бы напечатать специальное сообщение или сделать что-то такое, что поясняло бы, что запросы возвращаются пустыми и никакие значения не были вставлены. В этом случае, вы можете ввести следующее:

```
EXEC SQL WHENEVER NOT FOUND GOTO No_rows;
```

No_rows — это метка в некотором коде, содержащем определенное действие. С другой стороны, если вам нужно сделать выборку в программе позже, вы можете ввести следующее в этой точке:

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

что бы выполнение выборки повторялось до тех пор, пока все строки не будут извлечены, что является нормальной процедурой, не требующей специальной обработки.

МОДИФИЦИРОВАНИЕ КУРСОРОВ

Курсоры могут также быть использованы, чтобы выбирать группу строк из таблицы, которые могут быть затем модифицированы или удалены одна за другой. Это дает вам возможность обходить некоторые ограничения предикатов, используемых в командах UPDATE и DELETE. Вы можете ссылаться на таблицу, задействованную в предикате запроса курсора или любом из его подзапросов, которые вы не можете выполнить в предикатах самих этих команд. Как подчеркнуто в Главе 16, стандарт SQL отклоняет попытку удалить всех пользователей с рейтингом ниже среднего, в следующей форме:

```
EXEC SQL DELETE FROM Customers
WHERE rating < (SELECT AVG (rating)
                FROM Customers);
```

Однако, вы можете получить тот же эффект, используя запрос для выбора соответствующих строк, запомнив их в курсоре, и выполнив DELETE с использованием курсора. Сначала вы должны обьявить курсор:

```
EXEC SQL DECLARE Belowavg CURSOR FOR
SELECT *
FROM Customers
WHERE rating < (SELECT AVG (rating)
                FROM Customers);
```

Затем вы должны создать цикл, чтобы удалить всех заказчиков выбранных курсором:

```
EXEC SQL WHENEVER SQLERROR GOTO Error_handler;
EXEC SQL OPEN CURSOR Belowavg;
while not SOLCODE = 100 do
   begin
   EXEC SOL FETCH Belowavg INTO :a, :b, :c, :d, :e;
   EXEC SOL DELETE FROM Customers
   WHERE CURRENT OF Belowavg;
   end;
EXEC SOL CLOSE CURSOR Belowavg;
```

Предложение **WHERE CURRENT OF** означает, что DELETE применяется к строке, которая в настоящее время выбрана курсором. Здесь подразумевается, что и курсор, и команда DELETE ссылаются на одну и ту же таблицу, и следовательно, что запрос в курсоре — это не обьединение.

Курсор должен также быть модифицируемым. Являясь модифицируемым, курсор должен удовлетворять тем же условиям что и представления (см. Главу 21). Кроме того, ORDER BY и UNION, которые не разрешены в представлениях, в курсорах — разрешаются, но предохраняют курсор от модифицируемости. Обратите внимание в вышеупомянутом примере, что мы должны выбирать строки из курсора в набор переменных, даже если мы не собирались использовать эти переменные. Этого требует синтаксис команды FETCH. UPDATE работает так же.

Вы можете увеличить значение комиссионных всем продавцам, которые имеют заказчиков с оценкой=300, следующим способом. Сначала вы обьявляете курсор:

```
EXEC SOL DECLARE CURSOR High_Cust AS
SELECT *
FROM Salespeople
WHERE snum IN (SELECT snum
               FROM Customers
               WHERE rating = 300);
```

Затем вы выполняете модификации в цикле:

```
EXEC SQL OPEN CURSOR High_cust;
while SQLCODE = 0 do
   begin
   EXEC SOL FETCH High_cust
   INTO :id_num, :salesperson, :loc, :comm;
   EXEC SQL UPDATE Salespeople
   SET comm = comm + .01
   WHERE CURRENT OF High_cust;
   end;
EXEC SQL CLOSE CURSOR High_cust;
```

Обратите внимание: что некоторые реализации требуют, чтобы вы указывали в определении курсора, что курсор будет использоваться для выполнения команды UPDATE на определенных столбцах. Это делается с помощью заключительной фразы определения курсора — FOR UPDATE <column list>. Чтобы обьявить курсор High_cust таким способом, так чтобы вы могли модифицировать командой UPDATE столбец comm, вы должны ввести следующее предложение:

```
EXEC SQL DECLARE CURSOR High_Cust AS
SELECT *
FROM Salespeople
WHERE snum IN (SELECT snum
               FROM Customers
               WHERE rating = 300)
FOR UPDATE OF comm;
```

Это обеспечит вас определенной защитой от случайных модификаций, которые могут разрушить весь порядок в базе данных.

ПЕРЕМЕННАЯ INDICATOR

Пустые (NULLS) значения — это специальные маркеры определяемые самой SQL. Они не могут помещаться в главные переменные. Попытка вставить NULL значения в главнуюпеременную будет некорректна, так как главные языки не поддерживают NULL значений в SQL, по определению. Хотя результат при попытке вставить NULL значение в главную переменную определяет проектировщик, этот результат не должен ротиворечить теории базы данных, и поэтому обязан произвести ошибку: код **SQLCODE** в виде отрицательного числа, и вызвать подпрограмму управления ошибкой. Естественно вам нужно этого избежать. Поэтому, вы можете выбрать NULL значения с допустимыми значениями, не приводящими к разрушению вашей программы. Даже если программа и не разрушится, значения в главных переменных станут неправильными, потому что они не могут иметь NULL значений. Альтернативным методом предоставляемым для этой ситуацией является — функция переменной indicator (указатель).

Переменная indicator — обьявленная в разделе объявлений SQL напоминает другие переменные. Она может иметь тип главного языка который соответствует числовому типу в SQL. Всякий раз, когда вы выполняете операцию, которая должна поместить NULL значение в переменную главного языка, вы должны использовать переменную indicator, для надежности. Вы помещаете переменную indicator в команду SQL непосредственно после переменной главного языка которую вы хотите защитить, без каких-либо пробелов или запятых, хотя вы и можете, при желании, вставить слово INDICATOR.

Переменной indicator в команде, изначально присваивается значение 0. Однако, если производится значение NULL, переменная indicator становится равной отрицательному числу. Вы можете проверить значение переменной indicator, чтобы узнать, было ли найдено значение NULL. Давайте предположим, что поля city и comm, таблицы Продавцов, не имеют ограничения NOT NULL, и что мы объявили вразделе обьявлений SQL, две ПАСКАЛЬевские переменные целого типа, i_a и i_b.

(Нет ничего такого в разделе обьявлений, что могло бы представить их как переменные indicator. Они станут переменными indicator, когда будут использоваться как переменные indicator.)

Имеется одна возможность:

```
EXEC SQL OPEN CURSOR High_cust;
while SQLCODE = O do
   begin
   EXEC SQL FETCH High_cust
   INTO :id_num, :salesperson, :loc, :i_a, :commINDlCATOR, :i_b;
   If i_a >= 0 and i_b >= 0
     then {no NULLs produced}
          begin
          EXEC SQL UPDATE Salespeople
          SET comm = comm + .01
          WHERE CURRENT OF Hlgh_cust
          end {then}
     else {one or both NULL}
          begin
          If i_a < 0 then writeln('salesperson ', id_num,
                                  ' has no city');
          If i_b < 0 then writeln('salesperson ', id_num,
                                  ' has no commission')
          end {else}
   end; {while}
EXEC SQL CLOSE CURSOR High_cust;
```

Как вы видите, мы включили, ключевое слово INDICATOR в одном случае, и исключили его в другом случае, чтобы показать, что эффект будет одинаковым в любом

случае. Каждая строка будет выбрана, но команда UPDATE выполнится только если NULL значения не будут обнаружены.

Если будут обнаружены NULL значения, выполнится еще одна часть программы, которая распечатает предупреждающее сообщение, где было найдено каждое NULL значение.

Обратите внимание: переменные indicator должны проверяться в главном языке, как указывалось выше, а не в предложении WHERE команды SQL.

Последнее в принципе не запрещено, но результат часто бывает непредвиденным.

ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННОЙ INDICATOR ДЛЯ ЭМУЛЯЦИИ NULL ЗНАЧЕНИЙ SQL

Другая возможность состоит в том, чтобы обрабатывать переменную indicator, связывая ее с каждой переменной главного языка специальным способом, эмулирующим поведение NULL значений SQL.

Всякий раз, когда вы используете одно из этих значений в вашей программе, например в предложении if ... then, вы можете сначала проверить связанную переменную indicator, является ли ее значение=NULL. Если это так, то вы обрабатываете переменную по-другому. Например, если NULL значение было извлечено из поля city для главной переменной city, которая связана с переменной indicator — i_city, вы должны установить значение city равное последовательности пробелов. Это будет необходимо, только если вы будете распечатывать его на принтере; его значение не должно отличаться от логики вашей программы. Естественно, i_city автоматически установливается в отрицательное значение. Предположим, что вы имели следующую конструкцию в вашей программе:

```
If sity = 'London' then comm: = comm + .01
                   else comm: = comm - .01
```

Любое значение, вводимое в переменную city, или будет равно "London", или не будет равно. Следовательно, в каждом случае значение комиссионных будет либо увеличено, либо уменьшено. Однако, эквивалентные команды в SQL выполняются по разному:

```
EXEC SQL UPDATE Salespeople
SET comm = comm + .01
WHERE sity = 'London';
```

и

```
EXEC SQL UPDATE Salespeople
SET comm = comm - .01;
WHERE sity <> 'London';
```

(Вариант на ПАСКАЛе работает только с единственным значением, в то время как вариант на SQL работает со всеми таблицами.)

Если значение city в варианте на SQL будет равно значению NULL, оба предиката будут неизвестны, и значение comm, следовательно, не будет изменено в любом случае.

Вы можете использовать переменную indicator, чтобы сделать поведение вашего главного языка непротиворечащим этому, с помощью создания условия, которое исключает NULL значения:

```
If i_city > = 0 then
begin
If city = 'London' then comm: = comm + .01
                   else comm: = comm - .01;
end;
{begin and end нужны здесь только для понимания}
```

ПРИМЕЧАНИЕ: Последняя строка этого примера содержит ремарку — {begin и end необходимы только для понимания}

В более сложной программе, вы можете захотеть установить Булеву переменную в "верно", чтобы указать что значение city = NULL. Затем вы можете просто проверять эту переменную всякий раз, когда вам это необходимо.

ДРУГОЕ ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННОЙ INDICATOR

Переменная indicator также может использоваться для назначения значения NULL. Просто добавьте ее к имени главной переменной в команде UPDATE или INSERT тем же способом что и в команде SELECT. Если переменная indicator имеет отрицательное значение, значение NULL будет помещено в поле. Например, следующая команда помещает значения NULL в поля city и comm, таблицы Продавцов, всякий раз, когда переменные indicator — i_a или i_b будут отрицательными; в противном случае она помещает туда значения главных переменных:

```
EXEC SQL INSERT INTO Salespeople
VALUES (:Id_num, :salesperson, :loc:i_a, :comm:i_b);
```

Переменная indicator используется также, чтобы показывать отбрасываемую строку. Это произойдет если вы вставляете значения символов SQL в главную переменную которая не достаточно длинна чтобы вместить все символы. Это особая проблема с нестандартным типами данных — VARCHAR и LONG (смотри Приложение C). В этом случае, переменная будет заполнена первыми символами строки, а последние символы будут потеряны. Если используется переменная indicator, она будет установлена в положительное значение, указывающее на длину отбрасываемой части строки, позволяя таким образом вам узнать, сколько символов было потеряно.

В этом случае, Вы можете проверить с помощью просмотра — значение переменной indicator > 0, или < 0.

РЕЗЮМЕ

Команды SQL вкладываются в процедурные языках, чтобы объединить силы двух подходов. Некоторые дополнительные средства SQL необходимы, чтобы выполнить эту работу. Вложенные команды SQL транслируемые программой, называемой прекомпилятором, в форму пригодную для использования транслятором главного языка, и используемые в этом главном языке, как вызовы процедуры к подпрограммам которые создает прекомпилятор, называются — м*одулями доступа*. ANSI поддерживает вложение SQL в языки: ПАСКАЛЬ, ФОРТРАН, КОБОЛ, и PL/I. Другие языки также используются, особенно Си. В попытке кратко описать вложенный SQL, имеются наиболее важные места в этой главе:

* Все вложенные команды SQL начинаются словами EXEC SQL и заканчиваются способом, который зависит от используемого главного языка.

* Все главные переменные, доступные в командах SQL, должны быть обьявлены в разделе объявлений SQL прежде, чем они будут использованы.

* Всем главным переменным должно предшествовать двоеточие, когда они используются в команде SQL.

* Запросы могут сохранять свой вывод непосредственно в главных переменных, используя предложение INTO, если и только если, они выбирают единственную строку.

* Курсоры могут использоваться для сохранения вывода запроса, и доступа к одной строке в каждый момент времени. Курсоры бывают обьявленными (если определяют запрос в котором будут содержаться), открытыми (если выполняют запрос), и закрытыми (если удаляют вывод запроса из курсора). Если курсор открыт, команда FETCH, используется чтобы перемещать его по очереди к каждой строке вывода запроса.

* Курсоры являются модифицируемыми или только-чтение. Чтобы стать модифицируемым, курсор должен удовлетворять всем критериям которым удовлетворяет просмотр; кроме того, он не должен использовать предложений ORDER BY или UNION, которые в любом случае не могут использоваться просмотрами. Не модифицируемый курсор является курсором только-чтение.

* Если курсор модифицируемый, он может использоваться для определения, какие строки задействованы вложенными командами UPDATE и DELETE через предложение WHERE CURRENT OF. DELETE или UPDATE должны быть вне той таблицы к которой курсор обращается в запросе.

* SQLCODE должен быть обьявлен как переменная числового типа для каждой программы которая будет использовать вложенный SQL. Его значение установливается автоматически после выполнения каждой команды SQL.

* Если команда SQL выполнена как обычно, но не произвела вывода или ожидаемого изменения в базе данных, SQLCODE = 100. Если команда произвела ошибку, SQLCODE будет равняться некоторому аппаратно-определенному отрицательному числу, которое описывает ошибку. В противном случае, SQLCODE = 0.

* Предложение WHENEVER может использоваться для определения действия которое нужно предпринять когда SQLCODE = 100 (не найдено) или когда SQLCODE равен отрицательному числу (SQLERROR). Действием может быть или переход к некоторой определенной метке в программе (GOTO <label>) или отсутствие какого-либо действия вообще (продолжить). Последнее, установлено по умолчанию.

* Числовые переменные могут также использоваться как переменные indicator. Переменные indicator следуют за другим именами переменных в команде SQL, без каких бы то ни было посторонних символов кроме (необязательного) слова INDICATOR.

* Обычно, значение переменной indicator = 0. Если команда SQL пытается поместить NULL значение в главную переменную которая использует indicator, indicator будет установлен в отрицательное значение. Этот факт можно использовать, чтобы предотвращать ошибки и для помечания NULL значений SQL для специальной обработки их в главной программе.

* Переменная indicator может использоваться для вставки NULL значений в команды SQL — INSERT или UPDATE. Она также может принимать положительное значение указывающее на длинну отбрасываемой части строки не поместившейся в предельные границы какой-нибудь переменной, куда эта строка помещалась.

РАБОТА С SQL

Обратите внимание: Ответы для этих упражнений написаны в псевдокодах, являющихся английским языком описания логики, которой должна следовать программа. Это сделано для того, чтобы помомочь читателям, которые могут быть незнакомы с Паскалем (или любым другим языком). Кроме того, это лучше сфокусирует ваше

внимание на включемых понятиях, опуская частности того или другого языка. Чтобы не противоречить нашим примерам, стиль псевдокода будет напоминать Паскаль.

Мы опустим из программ все, что не относится напрямую к рассматриваемым вопросам, например, определение устройств ввода-вывода, подключение к базе данных, и так далее. Конечно, имеется много способов чтобы выполнять такие упражнения; и совсем не обязательно, что представленые варианты решений являются самыми удачными.

1. Разработайте простую программу, которая выберет все комбинации полей snum и cnum из таблиц Порядков и Заказчиков и выясните, всегда ли предыдущая комбинация такая же как последующая. Если комбинация из таблицы Порядков не найдена в таблице Заказчиков, значе ние поля snum для этой строки будет изменено на удовлетворяющее условию совпадения. Вы должны помнить, что курсор с подзапросом — модифицируем (ANSI ограничение, также применимо к просмотрам, и что базисная целостность базы данных это не тоже самое что про верка на ошибку (т.е. первичные ключи уникальны, все поля cnums в таблице Порядков правильны, и так далее). Проверьте раздел обьявлений, и убедитесь что там обьявлены все используемые курсоры.
2. Предположим, что ваша программа предписывает ANSI запрещение курсоры или просмотры использующие модифицируемые подзапросы. Как вы должны изменить вышеупомянутую программу?
3. Разработайте программу, которая подсказывает пользователям изменить значения поля city продавца, автоматически увеличивает комиссионные на .01 для продавца, переводимого в Барселону и уменьшает их на .01 для продавца, переводимого в Сан Хосе. Кроме того, продавец, находящийся в Лондоне, должен потерять .02 из своих комиссионных, независимо от того, меняет он город или нет, в то время как продавец, не находящийся в Лондоне, должен иметь увеличение комиссионных на .02. Изменение в комиссионных, основывающееся на нахождении продавца в Лондоне, может применяться независимо от того, куда тот переводится. Выясните, могут ли поле city или поле comm содержать NULL значения, и обработайте их, как это делается в SQL. Предупреждение: эта программа имеет некоторые сокращения.

(См. Приложение A для ответов.)

Приложение A

ОТВЕТЫ ДЛЯ УПРАЖНЕНИЙ

Глава 1

1. cnum
2. rating
3. Другим словом для строки является запись. Другим словом для столбца является поле.
4. Потому что строки, по определению, находятся без какого либо определенного упорядочения.

Глава 2

1. Символ (или текст) и номер
2. Нет
3. Язык Манипулирования Данными (ЯЗЫК DML)
4. Это слово в SQL имеет специальное учебное значение

Глава 3

1.

```
SELECT onum, amt, odate
   FROM Orders;
```

2.

```
SELECT *
   FROM Customers
   WHERE snum = 1001;
```

3.

```
SELECT city, sname, snum, comm
   FROM Salespeople;
```

4.

```
SELECT rating, cname
   FROM Customers
   WHERE city = 'SanJose';
```

5.

```
SELECT DISTINCT snum
   FROM Orders;
```

Глава 4

1.

```
SELECT *
   FROM Orders
   WHERE amt > 1000;
```

2.

```
SELECT sname, city
   FROM Salespeople
   WHERE city = 'London' AND comm > .10;
```

3.

```
SELECT *
   FROM Customers
   WHERE rating > 100 OR city = 'Rome';
```

или

```
SELECT *
   FROM Customers
   WHERE NOT rating < = 100 OR city = 'Rome';
```

или

```
SELECT *
   FROM Customers
   WHERE NOT (rating < = 100 AND city < > 'Rome');
```

Могут быть еще другие решения.

4.

onum	amt	odate	cnum	snum
3001	18.69	10/03/1990	2008	1007
3003	767.19	10/03/1990	2001	1001
3005	5160.45	10/03/1990	2003	1002
3009	1713.23	10/04/1990	2002	1003
3007	75.75	10/04/1990	2004	1002
3008	4723.00	10/05/1990	2006	1001
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

5.

onum	amt	odate	cnum	snum
3001	18.69	10/03/1990	2008	1007
3003	767.19	10/03/1990	2001	1001
3006	1098.16	10/03/1990	2008	1007
3009	1713.23	10/04/1990	2002	1003
3007	75.75	10/04/1990	2004	1002
3008	4723.00	10/05/1990	2006	1001
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

6.

```
SELECT *
   FROM Salespeople;
```

Глава 5

1.

```
SELECT *
   FROM Orders
   WHERE odate IN (10/03/1990,10/04/1990);
```

2. и

```
SELECT *
   FROM Orders
   WHERE odate BETWEEN 10/03/1990 AND 10/04,1990;
```

3.

```
SELECT *
   FROM Customers
   WHERE snum IN (1001,1004);
```

4.

```
SELECT *
   FROM Customers
   WHERE cname BETWEEN 'A' AND 'H';
```

ПРИМЕЧАНИЕ: В ASCII базовой системе Hoffman не будет выведен из-за конечных пробелов после H. По той же самой причине вторая граница не может быть G, поскольку она не выведет имена Giovanni и Grass. G может использоваться в сопровождении с Z, так чтобы следовать за другими символами в алфавитном порядке, а не предшествовать им, как это делают пробелы.

5.

```
SELECT *
   FROM Customers
   WHERE cname LIKE 'C%';
```

6.

```
SELECT *
   FROM Orders
   WHERE amt <> 0 AND (amt IS NOT NULL);
```

или

```
SELECT *
   FROM Orders
   WHERE NOT (amt = 0 OR amt IS NULL);
```

Глава 6

1.

```
SELECT COUNT(*)
   FROM Orders
   WHERE odate = 10/03/1990;
```

2.

```
SELECT COUNT (DISTINCT city)
   FROM Customers;
```

3.

```
SELECT cnum, MIN (amt)
   FROM Orders
   GROUP BY cnum;
```

4.

```
SELECT MIN (cname)
   FROM Customers
   WHERE cname LIKE 'G%';
```

5.

```
SELECT city, MAX (rating)
   FROM Customers
   GROUP BY city;
```

6.

```
SELECT odate, count (DISTINCT snum)
   FROM Orders
   GROUP BY odate;
```

Глава 7

1.

```
SELECT onum, snum, amt * .12
   FROM Orders;
```

2.

```
SELECT 'For the city ', city, ', the highest rating is ', MAX (rating)
   FROM Customers
   GROUP BY city;
```

3.

```
SELECT rating, cname, cnum
   FROM Customers
   ORDER BY rating DESC;
```

4.

```
SELECT odate, SUM (amt)
   FROM Orders
   GROUP BY odate
   ORDER BY 2 DESC;
```

Глава 8

1.

```
SELECT onum, cname
   FROM Orders, Customers
   WHERE Customers.cnum = Orders.cnum;
```

2.

```
SELECT onum, cname, sname
   FROM Orders, Customers, Salespeople
   WHERE Customers.cnum = Orders.cnum AND Salespeople.snum = Orders.snum;
```

3.

```
SELECT cname, sname, comm
   FROM Salespeople, Customers
   WHERE Salespeople.snum = Customers.snum AND comm * .12;
```

4.

```
SELECT onum, comm * amt
   FROM Salespeople, Orders, Customers
   WHERE rating > 100 AND
         Orders.cnum = Customers.cnum AND
         Orders.snum = Salespeople.snum;
```

Глава 9

1.

```
SELECT first.sname, second.sname
   FROM Salespeople first, Salespeople second
   WHERE first.city = second.city AND first.sname < second.sname;
```

Псевдонимам нет необходимости иметь именно такие имена.

2.

```
SELECT cname, first.onum, second.onum
   FROM Orders first, Orders second, Customers
   WHERE first.cnum = second.cnum AND
         first.cnum = Customers.cnum AND
         first.onum < second.onum;
```

Ваш вывод может иметь некоторые отличия, но в вашем ответе все логические компоненты должны быть такими же.

3.

```
SELECT a.cname, a.city
   FROM Customers a, Customers b
   WHERE a.rating = b.rating AND b.cnum = 2001;
```

Глава 10

1.

```
SELECT *
   FROM Orders
   WHERE cnum = (SELECT cnum
                 FROM Customers
                 WHERE cname = 'Cisneros');
```

или

```
SELECT *
   FROM Orders
   WHERE cnum IN (SELECT cnum
                  FROM Customers
                  WHERE cname = 'Cisneros');
```

2.

```
SELECT DISTINCT cname, rating
   FROM Customers, Orders
   WHERE amt > (SELECT AVG (amt)
                FROM Orders)
         AND Orders.cnum = Customers.cnum;
```

3.

```
SELECT snum, SUM (amt)
   FROM Orders
   GROUP BY snum
   HAVING SUM (amt) > (SELECT MAX (amt)
                       FROM Orders);
```

Глава 11

1.

```
SELECT cnum, cname
   FROM Customers outer
   WHERE rating = (SELECT MAX (rating)
                   FROM Customers inner
                   WHERE inner.city = outer.city);
```

2. Решение с помощью соотнесенного подзапроса:

```
SELECT snum, sname
   FROM Salespeople main
   WHERE city IN (SELECT city
   FROM Customers inner
   WHERE inner.snum <> main.snum);
```

3. Решение с помощью объединения:

```
SELECT DISTINCT first.snum, sname
   FROM Salespeople first, Customers second
   WHERE first.city = second.city AND first.snum <> second.snum;
```

Соотнесенный подзапрос находит всех заказчиков, не обслуживаемых данным продавцом, и выясняет: живет ли кто-нибудь из них в его городе. Решение с помощью обьединения является более простым и более интуитивным. Оно находит случаи, где поля city совпадают, а поля snums нет. Следовательно, обьединение является более изящным решением для этой проблемы, чем то, которое мы исследовали до этого. Имеется еще более изящное решение с помощью подзапроса, с которым Вы столкнетесь позже.

Глава 12

1.

```
SELECT *
   FROM Salespeople first
   WHERE EXISTS (SELECT *
                 FROM Customers second
                 WHERE first.snum = second.snum AND rating = 300);
```

2.

```
SELECT a.snum, sname, a.city, comm
   FROM Salespeople a, Customers b
   WHERE a.snum = b.snum AND b.rating = 300;
```

3.

```
SELECT *
   FROM Salespeople a
   WHERE EXISTS (SELECT *
                 FROM Customers b
                 WHERE b.city = a.city AND a.snum <> b.snum);
```

4.

```
SELECT *
   FROM Customers a
   WHERE EXISTS (SELECT *
                 FROM Orders b
                 WHERE a.snum = b.snum AND a.cnum <> b.cnum)
```

Глава 13

1.

```
SELECT *
   FROM Customers
   WHERE rating >= ANY (SELECT rating
                        FROM Customers
                        WHERE snum = 1002);
```

2.

cnum	***cname***	***city***	***rating***	***snum***
2002	Giovanni	Rome	200	1003
2003	Liu	San Jose	200	1002
2004	Grass	Berlin	300	1002
2008	Cisneros	SanJose	300	1007

3.

```
SELECT *
   FROM Salespeople
   WHERE city <> ALL (SELECT city
                      FROM Customers);
```

ИЛИ

```
SELECT *
   FROM Salespeople
   WHERE NOT city = ANY (SELECT city
                         FROM Customers);
```

4.

```
SELECT *
   FROM Orders
   WHERE amt > ALL (SELECT amt
                    FROM Orders a, Customers b
                    WHERE a.cnum = b.cnum AND b.city = 'London');
```

5.

```
SELECT *
   FROM Orders
   WHERE amt > (SELECT MAX (amt)
                FROM Orders a, Customers b
                WHERE a.cnum = b.cnum AND b.city = 'London');
```

Глава 14

1.

```
SELECT cname, city, rating, 'High Rating'
   FROM Customers
   WHERE rating >= 200

   UNION

   SELECT cname, city, rating, ' Low Ratlng'
   FROM Customers
   WHERE rating < 200;
```

ИЛИ

```
SELECT cname, city, rating, 'High Rating'
   FROM Customers
   WHERE rating >= 200

   UNION

   SELECT cname, city, rating, ' Low Rating'
   FROM Customers
   WHERE NOT rating >= 200;
```

Различие между этими двумя предложениями — в форме второго предиката. Обратите внимание, что в обоих случаях строка "Low Rating" имеет в начале дополнительный пробел для того, чтобы совпадать со строкой "High Rating" по длине.

2.

```
SELECT cnum, cname
   FROM Customers a
   WHERE 1 < (SELECT COUNT (*)
              FROM Orders b
              WHERE a.cnum = b.cnum)

   UNION

   SELECT snum, sname
   FROM Salespeople a
   WHERE 1 < (SELECT COUNT (*)
              FROM Orders b
              WHERE a.snum = b.snum)

   ORDER BY 2;
```

3.

```
SELECT snum
   FROM Salespeople
   WHERE city = 'San Jose'

   UNION

   (SELECT cnum
   FROM Customers
   WHERE city = 'San Jose'

   UNION ALL

   SELECT onum
   FROM Orders
   WHERE odate = 10/03/1990);
```

Глава 15

1.

```
INSERT INTO Salespeople (city, cname, comm, cnum)
   VALUES ('San Jose', 'Blanco', NULL, 1100);
```

2.

```
DELETE FROM Orders WHERE cnum = 2006;
```

3.

```
UPDATE Customers
   SET rating = rating + 100
   WHERE city = 'Rome';
```

4.

```
UPDATE Customers
   SET snum = 1004
   WHERE snum = 1002;
```

Глава 16

1.

```
INSERT INTO Multicust
   SELECT *
   FROM Salespeople
   WHERE 1 < (SELECT COUNT (*)
              FROM Customers
              WHERE Customers.snum = Salespeople.snum);
```

2.

```
DELETE FROM Customers
   WHERE NOT EXISTS (SELECT *
                     FROM Orders
                     WHERE cnum = Customers.cnum);
```

3.

```
UPDATE Salespeople
   SET comm = comm + (comm * .2)
   WHERE 3000 < (SELECT SUM (amt)
                 FROM Orders
                 WHERE snum = Salespeople.snum);
```

В более сложный вариант этой команды можно было бы вставить проверку, чтобы убедиться, что значения комиссионных не превышают 1.0 (100%):

```
UPDATE Salespeople
   SET comm = comm + (comm * .2)
   WHERE 3000 < (SELECT SUM (amt)
                 FROM Orders
                 WHERE snum = Salespeople.snum)
         AND comm + (comm * .2) < 1.0;
```

Эти проблемы могут иметь другие, такие же хорошие решения.

Глава 17

1.

```
CREATE TABLE Customers
   (cnum   integer,
    cname  char(10),
    city   char(10),
    rating integer,
    snum   integer);
```

2.

```
CREATE INDEX Datesearch ON Orders(odate);
```

(Все индексные имена, используемые в этих ответах — произвольные.)

3.

```
CREATE UNIQUE INDEX Onumkey ON Orders(onum);
```

4.

```
CREATE INDEX Mydate ON Orders(snum, odate);
```

5.

```
CREATE UNIQUE INDEX Combination ON Customers(snum, rating);
```

Глава 18

1.

```
CREATE TABLE Orders
   (onum  integer NOT NULL PRIMARY KEY,
    amt   decimal,
    odate date NOT NULL,
    cnum  integer NOT NULL,
    snum  integer NOT NULL,
    UNIOUE (snum, cnum));
```

или

```
CREATE TABLE Orders
   (onum  integer NOT NULL UNIQUE,
    amt   decimal,
    odate date NOT NULL,
    cnum  integer NOT NULL,
    snum  integer NOT NULL,
    UNIQUE (snum, cnum));
```

Первое решение предпочтительнее.

2.

```
CREATE TABLE Salespeople
   (snum  integer NOT NULL PRIMARY KEY,
    sname char(15) CHECK (sname BETWEEN 'AA' AND 'MZ'),
    city  char(15),
    comm  decimal NOT NULL DEFAULT = .10);
```

3.

```
CREATE TABLE Orders
   (onum  integer NOT NULL,
    amt   decimal,
    odate date,
    cnum  integer NOT NULL,
    snum  integer NOT NULL,
    CHECK ((cnum > snum) AND (onum > cnum)));
```

Глава 19

1.

```
CREATE TABLE Cityorders
  (onum integer NOT NULL PRIMARY KEY,
   amt  decimal,
   cnum integer,
   snum integer,
   city char (15),
  FOREIGN KEY (onum, amt, snum) REFERENCES Orders (onum, amt, snum),
  FOREIGN KEY (cnum, city) REFERENCES Customers (cnum, city));
```

2.

```
CREATE TABLE Orders
  (onum  integer NOT NULL,
   amt   decimal,
   odate date,
   cnum  integer NOT NULL,
   snum  integer,
   prev  integer,
  UNIQUE (cnum, onum),
  FOREIGN KEY (cnum, prev) REFERENCES Orders (cnum,onum));
```

Глава 20

1.

```
CREATE VIEW Highratings
  AS SELECT *
     FROM Customers
     WHERE rating = (SELECT MAX (rating)
                     FROM Customers);
```

2.

```
CREATE VIEW Citynumber
  AS SELECT city, COUNT (DISTINCT snum)
     FROM Salespeople
     GROUP BY city;
```

3.

```
CREATE VIEW Nameorders
  AS SELECT sname, AVG (amt), SUM (amt)
     FROM Salespeople, Orders
     WHERE Salespeople.snum = Orders.snum
     GROUP BY sname;
```

4.

```
CREATE VIEW Multcustomers
  AS SELECT *
     FROM Salespeople a
     WHERE 1 < (SELECT COUNT (*)
                FROM Customers b
                WHERE a.snum = b.snum);
```

Глава 21

1.

#1 — не модифицируемый, потому что он использует DISTINCT.
#2 — не модифицируемый, потому что он использует обьединение, агрегатную функцию и GROUP BY.
#3 — не модифицируемый, потому что он основывается на #1, который сам по себе не модифицируемый.

2.

```
CREATE VIEW Commissions
   AS SELECT snum, comm
      FROM Salespeople
      WHERE comm BETWEEN .10 AND .20
      WITH CHECK OPTION;
```

3.

```
CREATE TABLE Orders
   (onum  integer NOT NULL PRIMARY KEY,
    amt   decimal,
    odate date DEFAULT VALUE = CURDATE,
    snum  integer,
    cnum  integer);

   CREATE VIEW Entryorders
   AS SELECT onum, amt, snum, cnum
      FROM Orders;
```

Глава 22

1.

```
GRANT UPDATE (rating) ON Customers TO Janet;
```

2.

```
GRANT SELECT ON Orders TO Stephen WITH GRANT OPTION;
```

3.

```
REVOKE INSERT ON Salespeople FROM Claire;
```

4.
Шаг 1:

```
CREATE VIEW Jerrysview
   AS SELECT *
      FROM Customers
      WHERE rating BETWEEN 100 AND 500
      WITH CHECK OPTION;
```

Шаг 2:

```
GRANT INSERT, UPDATE ON Jerrysview TO Jerry;
```

5.
Шаг 1:

```
CREATE VIEW Janetsview
   AS SELECT *
      FROM Customers
      WHERE rating = (SELECT MIN (rating)
                      FROM Customers);
```

Шаг 2:

```
GRANT SELECT ON Janetsview TO Janet;
```

Глава 23

1.

```
CREATE DBSPACE Myspace
   (pctindex 15,
    pctfree 40);
```

2.

```
CREATE SYNONYM Orders FOR Diane.Orders;
```

3. Они должны быть откатаны обратно назад

4. Блокировка взаимоисключающего доступа

5. Толко чтение

Глава 24

1.

```
SELECT a.tname, a.owner, b.cname, b.datatype
   FROM SYSTEMCATOLOG a, SYSTEMCOLUMNS b
   WHERE a.tname = b.tname AND
         a.owner = b.owner AND
         a.numcolumns > 4;
```

Обратите Внимание: из-за того, что большинство имен столбца объединяемых таблиц — различны, не все из используемых псевдонимов a и b в вышеупомянутой команде — строго обязательны. Они представлены просто для понимания.

2.

```
SELECT tname, synowner, COUNT (ALL synonym)
   FROM SYTEMSYNONS
   GROUP BY tname, synowner;
```

3.

```
SELECT COUNT (*)
   FROM SYSTEMCATALOG a
   WHERE numcolumns/2 < (SELECT COUNT (DISTINCT cnumber)
                         FROM SYSTEMINDEXES b
                         WHERE a.owner = b.tabowner AND a.tname = b.tname);
```

Глава 25

1.

```
EXEC SQL BEGIN DECLARE SECTION;
     SQLCODE  : integer;
     { требуемый всегда }
     cnum     : integer;
     snum     : integer;
     custnum  : integer;
     salesnum : integer;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL DECLARE Wrong_Orders AS CURSOR FOR
     SELECT cnum, snum
     FROM Orders a
     WHERE snum <> (SELECT snum
                    FROM Customers b
                    WHERE a.cnum = b.cnum);
  { Мы пока еще используем здесь SQL для выполнения основной работы. Запрос
  выше размещает строки таблицы Порядков, которые не согласуются с таблицей
  Заказчиков. }

  EXEC SQL DECLARE Cust_assigns AS CURSOR FOR
     SELECT cnum, snum
     FROM Customers;
  { Этот курсор используется для получения правильных значений snum }

  begin { основная программа }
  EXEC SQL OPEN CURSOR Wrong_Orders;
  while SQLCODE = 0 do { Цикл до тех пор, пока Wrong_Orders не опустеет }
     begin
     EXEC SQL FETCH Wrong_Orders INTO (:cnum, :snum);
     if SQLCODE = 0 then
        begin {Когда Wrong_Orders опустеет, мы не хотели бы продолжать
  выполнение этого цикла до бесконечности}
        EXEC SQL OPEN CURSOR Cust_Assigns;
        repeat
           EXEC SQL FETCH Cust_Assigns INTO (:custnum, :salesnum);
           until :custnum = :cnum;
  { Повторять FETCH до тех пор пока ... команда будет просматривать
  Cust_Assigns курсор до строки, которая соответствует текущему значению cnum,
  найденному в Wrong_Orders }

           EXEC SQL CLOSE CURSOR Cust_assigns;
  { Поэтому мы будем начинать новый вывод в следующий раз через цикл. Значение
  в котором мы получим из этого курсора сохраняется в переменной — salesnum. }

           EXEC SQL UPDATE Orders
           SET snum = :salesnum
           WHERE CURRENT OF Wrong_Orders;
        end; {Если SQLCODE = 0}.
     end; { Пока SQLCODE ... выполнить }
  EXEC SQL CLOSE CURSOR Wrong_Orders;
  end; { основная программа }
```

2. Для данной программы, которую я использовал, решение будет состоять в том, чтобы просто включить поле onum первичным ключом таблицы Порядков, в курсор Wrong_Orders. В команде UPDATE, вы будете затем использовать предикат WHERE onum =:ordernum (считая целую переменную — odernum, обьявленной), вместо WHERE CURRENT Of Wrong_Orders.
Результатом будет программа наподобие этой (большинство комментариев из предыдущей программы здесь исключены):

```
EXEC SQL BEGIN DECLARE SECTION;
   SQLCODE  : integer;
   odernum  : integer;
   cnum     : integer;
   snum     : integer;
   custnum  : integer;
   salesnum : integer;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE Wrong_Orders AS CURSOR FOR
   SELECT onum, cnum, snum
   FROM Orders a
   WHERE snum <> (SELECT snum
                  FROM Customers b
                  WHERE a.cnum = b.cnum);

EXEC SQL DECLARE Cust_assigns AS CURSOR FOR
   SELECT cnum, snum
   FROM Customers;

begin { основная программа }
EXEC SQL OPEN CURSOR Wrong_Orders;
while SQLCODE = 0 do {Цикл до тех пор пока Wrong_Orders не опустеет}
   begin
   EXEC SQL FETCH Wrong_Orders INTO (:odernum, :cnum, :snum);
   if SQLCODE = 0 then
      begin
      EXEC SQL OPEN CURSOR Cust_Assigns;
      repeat
         EXEC SQL FETCH Cust_Assigns INTO (:custnum, :salesnum);
         until :custnum = :cnum;
      EXEC SQL CLOSE CURSOR Cust_assigns;
      EXEC SQL UPDATE Orders
      SET snum = :salesnum WHERE CURRENT OF Wrong_Orders;
      end; { If SQLCODE = 0 }
   end; { While SQLCODE ... do }
EXEC SQL CLOSE CURSOR Wrong_Orders;
end; { основная программа }
```

3.

```
EXEC SQL BEGIN DECLARE SECTION;
   SQLCODE  : integer;
   newcity  : packed array[1..12] of char;
   commnull : boolean;
   citynull : boolean;
   response : char;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE CURSOR Salesperson AS
   SELECT * FROM SALESPEOPLE;

begln { main program }
EXEC SQL OPEN CURSOR Salesperson;
EXEC SQL FETCH Salesperson INTO (:snum, :sname, :city, :i_cit, :comm,
:i_com);
{ Выборка первой строки }
while SQLCODE = 0 do { Пока эти строки в таблице Продавцов. }
   begin
   if i_com < 0 then commnull: = true;
   if i_cit < 0 then citynull: = true;
{ Установить логические флаги, которые могут показать NULLS.}
   if citynull
      then begin
           write ('Нет текущего значения city для продавца ', snum,
                  ' Хотите предоставить хотя бы одно? (Y/N)');
{ Подсказка покажет значение city состоящее из NULL значений. }
           read (response);
{ Ответ может быть сделан позже. }
           end { если citynull }
      else begin { не citynull }
           if not commnull then
{ Чтобы выполнять сравнение и операции только для не-NULL значений связи }
                           begin
                           if city='London' then comm:=comm*.02*.02
                                           else comm:=comm+.02;
                           end;
{ Даже если значение и не - commnull, begin и end здесь для ясности. }
           write ('Текущий city для продавца', snum, 'есть', city,
                  'Хотите его изменить? (Y/N)');
{ Обратите Внимание: Продавец, не назначеный в данное время в определенный
город, не будет иметь изменений комиссионых при определении, находятся ли он
в Лондоне. }
           read (response);
{ Ответ теперь имеет значение независимо от того, что citynull верен или
неверен. }
           end; {иначе не citynull}
   if response = 'Y' then
          begin
          write ('Введите новое значение city:');
          read (newcity);
          if not commnull then
{ Эта операция может быть выполнена только для не-NULL значений. }
              case newcity of:
                 begin
                'Barcelona' : comm:= comm + .01,
                'San Jose'  : comm:= comm * .01
                 end; {случно и если не commnull}
          EXEC SQL UPDATE Salespeople
          SET city = :newcity, comm = :comm:i_com
             WHERE CURRENT OF Salesperson;
{ Переменная индикатора может поместить NULL значение в поле comm если так
назначено. }
          end; { Если ответ = 'Y', или если ответ <> 'Y', изменений не
будет. }
   EXEC SQL FETCH Salesperson INTO (:snum, :sname, :city, :i_clt, :comm,
:l_com);
{ выборка следующей строки }
```

```
   end; {если SQLCODE = 0}
EXEC SQL CLOSE CURSOR Salesperson;
end; {основной программы}
```

Приложение B

ТИПЫ ДАННЫХ В SQL

ТИПЫ ДАННЫХ, РАСПОЗНАВАЕМЫЕ С ПОМОЩЬЮ ANSI, состоят из символов и различных типов чисел, которые могут классифицироваться как *точные* числа и *приблизительные* числа.

Точные числовые типы — это номера, с десятичной точкой или без десятичной точки.

Приблизительные числовые типы — это номера в показательной (экспоненциальной по основанию 10) записи.

Для все прочих типов, отличия слишком малы чтобы их как-то классифицировать.

Иногда типы данных используют аргумент, который я называю *размером аргумента*, чей точный формат и значение меняется в зависимости от конкретного типа.

Значения по умолчанию обеспечены для всех типов, если размер аргумента отсутствует.

ТИПЫ ANSI

Ниже представлены типы данных ANSI (имена в круглых скобках — это синонимы):

TEXT
ТЕКСТ

CHAR (или CHARACTER) Строка текста в реализационно-определенном формате. Размер агрумента здесь это единственное неотрицательное целое число, которое ссылается к максимальной длине строки. Значения этого типа, должны быть заключены в одиночные кавычки, например 'text'. Две рядом стоящие одиночные кавычки ('') внутри строки будет пониматься как одна одиночная кавычка (').

ПРИМЕЧАНИЕ: Здесь и далее, фраза *Реализационно-Определенный* или Реализационно-Зависимый, указывает, что этот аргумент или формат зависит от конкретной программы, в которой реализуются данные.

EXACT NUMERIC
ТОЧНОЕ ЧИСЛО

DEC (или DECIMAL) Десятичное число; то есть, число которое может иметь десятичную точку. Здесь аргумент размера имеет две части: точность и масштаб. Масштаб не может превышать точность. Сначала указывается точность, разделительная запятая и далее аргумент масштаба. Точность указывает сколько значащих цифр имеет число. Максимальное десятичное число составляющее номер — реализационно-определенное значение, равное или большее чем этот номер. Масштаб указывает максимальное число цифр справо от десятичной точки. Масштаб = 0 делает поле эквивалентом целого числа.

NUMERIC Такое же как DECIMAL за исключением того, что максимальное десятичное не может превышать аргумента точности.

INT (или INTEGER) Число без десятичной точки. Эквивалентно DECIMAL, но без цифр справа от десятичной точки, то есть с масштабом равным

0. Аргумент размера не используется (он автоматически устанавливается в реализационно-зависимое значение).

SMALLINT Такое же как INTEGER, за исключением того, что, в зависимости от реализации, размер по умолчанию может (или не может) быть меньшее чем INTEGER.

APPROXIMATE NUMERIC
ПРИБЛИЗИТЕЛЬНОЕ ЧИСЛО

FLOAT Число с плавающей запятой на основе 10 показательной функции. Аргумент размера состоит из одного числа определяющего минимальную точность.

REAL Такое же как FLOAT, за исключением того, что никакого аргумента размера не используется. Точность установлена реализационно-зависимую по умолчанию.

DOUBLE PRECISION Такое же как REAL, за исключением того, что (или DOUBLE) реализационно-определяемая точность для DOUBLE PRECISION должна превышать реализационно-определяемую точность REAL.

ЭКВИВАЛЕНТНЫЕ ТИПЫ ДАННЫХ В ДРУГИХ ЯЗЫКАХ

Когда используется вложение SQL в другие языки, значения, используемые и произведенные командами SQL, обычно сохраняются в переменных главного языка (см. Главу 25). Эти переменные должны иметь тип данных, совместимый со значениями SQL, которые они будут получать. В дополнениях, которые не являются частью официального SQL стандарта, ANSI обеспечивает поддержку при использовании вложения SQL в четыре языка: Паскаль, PL/I, КОБОЛ, и ФОРТРАН. Между прочим, он включает определение эквивалентов SQL для данных типов переменных, используемых в этих языках.

Эквиваленты типов данных четырех языков определенных ANSI:

ПЛ/I

SQL ТИП	*ЭКВИВАЛЕНТ ПЛ/I*
CHAR	**CHAR**
DECIMAL	**FIXED DECIMAL**
INTEGER	**FIXED BINARY**
FLOAT	**FLOAT BINARY**

КОБОЛ

SQL ТИП	*ЭКВИВАЛЕНТ КОБОЛА*
CHAR(<integer>)	**PIC X (<integer>)**
INTEGER	**PIC S (<nines>) USAGE COMPUTTATIONAL**
NUMERIC	**PIC S (<nines with embedded V>) DISPLAY SING LEADING SEPERATE**

ПАСКАЛЬ

SQL ТИП	*ЭКВИВАЛЕНТ ПАСКАЛЯ*
INTEGER	**INTEGER**
REAL	**REAL**
CHAR (<length>)	**PACKED ARRAY [1..<length>] OF CHAR**

ФОРТРАН

SQL ТИП	*ЭКВИВАЛЕНТ ФОРТРАНА*
CHAR	**CHAR**
INTEGER	**INTEGER**
REAL	**REAL**
DOUBLE PRECISION	**DOUBLE PRECISION**

Приложение C

НЕКОТОРЫЕ ОБЩИЕ НЕСТАНДАРТНЫЕ СРЕДСТВА SQL

ИМЕЕТСЯ РЯД ОСОБЕННОСТЕЙ ЯЗЫКА SQL, которые пока не определены как часть стандарта ANSI или стандарта ISO (Международная Организация По Стандартизации), и являются общими для многочисленных реализаций, так как они были получены для практического использования.

Это дополнительные элементы чисел этих особенностей. Конечно, эти особенности меняются от программы к программе, и их обсуждение предназначено только чтобы показать некоторые общие подходы к ним.

ТИПЫ ДАННЫХ

Типы данных, поддерживаемые стандартом SQL, собраны в Приложении B. Это *колличество* для CHARACTER и разнообразие числовых типов. Реализация их может, фактически, быть значительно сложнее, чем показано в терминах типов, которые они фактически могут использовать. Мы будем здесь обсуждать ряд таких нестандартных типов данных.

ТИПЫ DATE И TIME

Как упомянуто в Главе 2, тип данных DATE широко поддерживается, даже если он не часть стандарта. Мы использовали ранее в нашей таблице Порядков этот тип, использующий формат mm/dd/yyyy. Это стандартный формат IBM в США. Разумеется, возможны и другие форматы, и программные реализации часто поддерживают ряд форматов, позволяя вам выбирать тот, который лучше для вас подходит. Реализация, которая предлагает эту особенность, должна быть способна преобразовывать дату одного формата в другой — автоматически.

Имеются несколько основных форматов даты, с которыми вы можете столкнуться:

Стандарт	*Формат*	*Пример*
Международная Организация По Стандартизации (ISO)	**yyyy-mm-dd**	**1990-10-31**
Японский Индустриальный Стандарт (JIS)	**yyyy-mm-dd**	**1990-10-31**
IBM Европейский Стандарт (EUR)	**dd.mm.yyyy**	**10.31.1990**

Наличие специального типа, определяемого для даты, дает возможность выполнять арифметические операция с датами. Например, вы можете добавлять число дней к дате и получать другую дату в программе, самостоятельно следящей за числом дней в месяцах, високосными годами и так далее. Даты могут также сравниваться; например фраза, дата A < дата B означает, что дата A предшествует дате B по времени.

Кроме даты, большое количество программ определяют специальный тип для *времени*, который может также быть представлен в ряде форматов, включая следующие:

Стандарт	*Формат*	*Пример*
Международная Организация По Стандартизации (ISO)	**hh-mm-ss**	**21.04.37**
Японский Индустриальный Стандарт (JIS)	**hh-mm-ss**	**21.04.37**
IBM Европейский Стандарт	**hh-mm-ss**	**21.04.37**
IBM USA Стандарт (USA)	**hh.mm AM/PM**	**9.04 PM**

Время может добавляться или сравниваться точно также как *дата*, с коррекцией числа секунд в минутах или часах автоматически. Кроме того, специально встроенные константы, указывающие текущую дату или время (**CURDATE** или **CURTIME**) являются

общими. Они похожи на константу **USER** (Пользователь), в которой их значение будет непрерывно меняться.

Можете ли вы включать *время* и *дату* в одно поле? Некоторые реализации определяют тип DATE достаточно точно, чтобы включать туда еще и TIME.

В качестве альтернативы, третий обобщающий тип, **TIMESTAMP**, может быть определен как комбинация этих двух.

ТИПЫ ТЕКСТОВОЙ СТРОКИ

ANSI поддерживает только один тип чтобы представлять текст. Это — тип **CHAR**. Любое поле такого типа должно иметь определенную длину. Если строка, вставляемая в поле, меньше, чем длина поля, она дополняется пробелами; строка не может быть длиннее, чем длина поля.

Хотя и достаточно удобное, это определение все же имеет некоторые ограничения для пользователя. Например, символьные поля должны иметь одинаковую длину, чтобы можно было выполнить команду UNION. Большинство реализаций поддерживают строки переменной длины для типов данных **VARCHAR** и **LONG VARCHAR** (или просто **LONG**). В то время как поле типа CHAR всегда может распределить память для максимального числа символов, которое может сохраняться в поле, поле VARCHAR при любом количестве символов может распределить только определенное количество памяти, чтобы сохранить фактическое содержание поля, хотя SQL может установить снаружи некоторое дополнительное пространство памяти, чтобы следить за текущей длиной поля.

Поле VARCHAR может быть любой длины, включая реализационно-определяемый максимум. Этот максимум может меняться от 254 до 2048 символов для VARCHAR, и до 16000 символов для LONG. LONG обычно используется для текста пояснительного характера или для данных, которые не могут легко сжиматься в простые значения полей; VARCHAR может использоваться для любой текстовой строки чья длина может меняться.

Между прочим, не всегда хорошо использовать VARCHAR вместо CHAR. Извлечение и модифицирование полей VARCHAR — более сложный и, следовательно, более медленный процесс, чем извлечение и модифицирование полей CHAR. Кроме того, некоторое количество памяти VARCHAR остается всегда неиспользованной (в резерве) для гарантии вмещения всей длины строки.

Вы должны просчитывать, насколько значения полей могут меняться по длине, а также, способны ли они к объединению с другими полями, перед тем как решить, использовать CHAR или VARCHAR. Часто, тип LONG используется для сохранения двоичных данных. Естественно, что использование размера такого "неуклюжего" поля будет ограничивать оперативность SQL.

Проконсультируйтесь с вашим руководством.

КОМАНДА FORMAT

Как мы подчеркивали в Главе 7, процесс вывода, выполняемого в стандарте SQL — ограничен. Хотя большинство реализаций включают SQL в пакеты, имеющие другие средства для управления этой функцией, некоторые реализации также используют команду типа **FORMAT** внутри SQL, чтобы навязывать выводу запроса определенные формы, структуры или ограничения. Среди возможных функций команды FORMAT существуют такие:

* определение ширины столбцов (при печати).
* определение представления NULL значений.
* обеспечение (новых) заголовков для столбцов.

* обеспечение заголовков внизу или вверху страниц выводимых на печать.
* навязывает присвоение или изменение форматам полей содержащих значения даты, времени или денежной суммы.
* вычисляет общие и промежуточные суммы, не исключая возможности обобщения поля, как это делает например SUM. (Альтернативным подходом к этой проблеме в некоторых программах является предложение COMPUTE.)

Команда FORMAT может вводиться сразу перед или сразу после запроса, к которому она применяется, в зависимости от реализации. Одна команда FORMAT обычно может применяться только к одному запросу, хотя любое число команд FORMAT может применяться к одному и тому же запросу. Вот некоторые примеры команды FORMAT:

```
FORMAT NULL '_ _ _ _ _ _ _';
FORMAT BTITLE 'Orders Grouped by Salesperson';
FORMAT EXCLUDE (2, 3);
```

Первая из них представляет значения NULL в виде '_ _ _ _ _ _ _' при выводе на печать; вторая вставляет заголовок 'Orders Grouped by Salesperson' в нижнюю часть каждой страницы; третья исключает второй и третий столбцы из вывода предыдущего запроса. Вы могли бы использовать последнюю из них, если вы выбираете конкретные столбцы, чтобы использовать их в предложении ORDER BY, в вашем выводе. Так как указанные функции команды FORMAT могут выполняться по разному, весь набор их приложений не может быть здесь показан.

Имеются другие команды, которые могут использоваться для выполнения тех же функций. Команда SET подобна команде FORMAT; она является вариантом или дополнением к команде, которая применяется во всех запросах текущего сеанса пользователя, а не просто в одиночном запросе. В следующей реализации, команда FORMAT начинается ключевым словом COLUMN следующим образом:

```
COLUMN odate FORMAT dd-mon-yy;
```

что навязывает формат типа 10-Oct-90 в поле даты, использующемся в выводе запроса на печать.

Предложение COMPUTE, упомянутое ранее, вставляется в запрос следующим образом:

```
SELECT odate, amt
FROM Orders
WHERE snum = 1001
COMPUTE SUM (amt);
```

Оно выводит все порядки продавца Peel, с датой и суммой приобретения по каждой дате, а в конце общую сумму приобретений.

Другая реализация выводит промежуточные суммы приобретений, используя COMPUTE в качестве команды. Сначала, она определяет разбивку

```
BREAK ON odate;
```

вывода вышеупомянутого запроса на страницы — сгруппировав их по датам, поэтому все значения odate в каждой группе — одинаковые. Затем вы можете ввести следующее предложение:

```
COMPUTE SUM OF amt ON odate;
```

Столбец в предложении ON предварительно должен быть использован в команде BREAK.

ФУНКЦИИ

Для SQL в стандарте ANSI, вы можете применять агрегатные функции для столбцов или использовать их значения в скалярных выражениях, таких например как — comm * 100. Имеется много других полезных функций, которые вы, вероятно встречали на практике.

Имеется список некоторых общих функций SQL, отличающихся от стандартных агрегатов. Они могут использоваться в предложениях SELECT запросов, точно так же как агрегатные функции, но эти функции выполняются для одиночных значений, а не для групповых. В следующем списке они классифицированны согласно типам данных, с которыми они выполняются. Если нет примечаний, то переменные в этом списке стандартизованы для любого выражения значений соответствующего типа, которые могут быть использованы в предложении SELECT:

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ

Эти функции применяются для чисел.

ФУНКЦИЯ	*ЗНАЧЕНИЕ*
ABX(X)	Абсолютное значение из X (преобразование отрицательного или положительного значений в положительное)
CEIL(X)	X является десятичным значением, которое будет округляться сверху.
FLOOR (X)	X является десятичным значением, которое будет округляться снизу.
GREATEST(X,Y)	Возвращает большее из двух значений.
LEAST(X,Y)	Возвращает меньшее из двух значений.
MOD(X,Y)	Возвращает остаток от деления X на Y.
POWER(X,Y)	Возвращает значение X в степени Y.
ROUND(X,Y)	Цикл от X до десятичного Y. Если Y отсутствует, цикл до целого числа.
SING(X)	Возвращает минус если X < 0, или плюс если X > 0.
SQRT(X)	Возвращает квадратный корень из X.

СИМВОЛЬНЫЕ ФУНКЦИИ

Эти функции могут быть применены для строк текста, либо из столбцов текстовых типов данных, либо из строк литерных текстов, или же комбинация из этих двух.

ФУНКЦИЯ	*ЗНАЧЕНИЕ*
LEFT(<string>,X)	Возвращает крайние левые (старшие) символы X из строки.
RICHT(<string>,X)	Возвращает символы X младшего разряда из строки
ASCII(<string>)	Возвращает код ASCII которым представляется строка в памяти компьютера.
CHR(<ASCIIcode>)	Возвращает принтерные символы кода ASCII.
VALUE(<string>)	Возвращает математическое значение для строки. Считается что строка имеет тип CHAR или VARCHAR, но состоит из чисел. VALUE('3') произведет число 3 типа INTEGER.
UPPER(<string>)	Преобразует все символы строки в символы верхнего регистра.

LOWER(<string>)	Преобразует все символы строки в символы нижнего регистра.
INITCAP(<string>)	Преобразует символы строки в заглавные буквы. В некоторых реализациях может иметь название — *PROPER*.
LENGTH(<string>)	Возвращает число символов в строке.
<string>||<string>	Объединяет две строки в выводе, так чтобы после первой немедленно следовала вторая. (значек || называется *оператором сцепления*).
LPAD(<string>,X,'*')	Дополняет строку слева звездочками **'*'**, или любым другим указанным символом, с колличестве, определяемом X.
RPAD(<string>,X, '')	То же самое что и LPAD, за исключением того, что дополнение делается справа.
SUBSTR(<string>,X,Y)	Извлекает Y символов из строки начиная с позиции X.

ФУНКЦИИ ДАТЫ И ВРЕМЕНИ

Эти функции выполняются только для допустимых значений даты или времени.

ФУНКЦИЯ	*ЗНАЧЕНИЕ*
DAY(<date>)	Извлекает день месяца из даты. Подобные же функции существуют для MONTH (МЕСЯЦ), YEAR (ГОД), HOUR (ЧАСЫ), SECOND (СЕКУНДЫ) и так далее.
WEEKDAY(<date>)	Извлекает день недели из даты.

ДРУГИЕ ФУНКЦИИ

Эта функция может быть применена к любому типу данных.

ФУНКЦИЯ	ЗНАЧЕНИЕ
NVL(<column>,<value>)	NVL (NULL Значение) будет меняться на значение <value> каждое NULL значение, найденое в столбце <column>. Если полученное значение <column> не =NULL, NVL ничего не делает.

INTERSECT и MINUS

Команда UNION, как вы уже видели в Главе 14, может объединить два запроса, обьединив их вывод в один. Два других обычно имеющихся способа объединения отдельных запросов — это **INTERSECT** (*Плюс*) и **MINUS** (*Минус*). **INTERSECT** выводит только строки, произведенные обоими перекресными запросами, в то время как **MINUS** выводит строки, которые производятся одним запросом, но не другим. Следовательно, следующие два запроса

```
SELECT *
FROM Salespeople
WHERE city = 'London'

INTERSECT

SELECT *
FROM Salespeople
WHERE 'London' IN (SELECT city
                   FROM Customers
                   WHERE Customers.snum = Salespeople.snum);
```

выведут строки, произведенные обоими запросами, производящими всех продавцов в Лондоне, которые имели по крайней мере одного заказчика, размещенного там также. С другой стороны, запрос

```
SELECT *
FROM Salespeople
WHERE city = 'London'

MINUS

SELECT *
FROM Salespeople
WHERE 'London' IN (SELECT sity
                   FROM Customers
                   WHERE Customers.snum = Salespeople.snum);
```

удалит строки, выбранные вторым запросом из вывода первого, и таким образом, будут выводены все продавцы в Лондоне, которые не имели там заказчиков. MINUS иногда еще называют **DIFFERENCE** (*ОТЛИЧИЕ*).

АВТОМАТИЧЕСКИЕ ВНЕШНИЕ ОБЬЕДИНЕНИЯ

В Главе 14 мы обсуждали внешнее обьединение и показывали вам, как выполнять его, используя команду UNION. Некоторые программы базы данных имеют более непосредственный способ выполнения внешних обьединений. В некоторых реализациях, вводимый знак "**+**" после предиката может выводить строки, которые удовлетворяют условию, также как и строки, которые ему не удовлетворяют. В условии предиката может содержаться поле, совпадающее для обеих таблиц, и NULL значения будут вставлены там, где такого совпадения не будет найдено.

Например, предположим, вы хотите видеть ваших продавцов и соответствующих им заказчиков, не исключая тех продавцов, которым не назначено ни одного заказчика (хотя такого нет в наших типовых таблицах, но в действительности это возможно):

```
SELECT a.snum, sname, cname
FROM Salespeople a, Customers b
WHEREa.snum = b.snum(+);
```

Это является эквивалентом следующего объединения (UNION):

```
SELECT a.snum, sname, cname
FROM Salespeople a, Customers b
WHERE a.snum = b.snum

UNION

SELECT snum, sname, '_ _ _ _ _ _ _ _ _ _'
FROM Salespeople
WHERE snum NOT IN (SELECT snum
FROM Customers);
```

Мы считаем, что подчеркивания будут отображены NULL значениями (см. команду FORMAT ранее в этом приложении, где описывалось отображение NULL значениями).

ОТСЛЕЖИВАНИЕ ДЕЙСТВИЙ

Ваша SQL реализация достаточна хороша, если она доступна многим пользователям, чтобы обеспечивать для них некий способ слежения за действиями, выполняемыми в базе данных. Имеются две основные формы, чтобы делать это:

Journaling (*Протоколирование*) и **Auditing** (*Ревизия*).

Эти формы отличаются по назначению.

Journaling применяется с целью защиты ваших данных при разрушении вашей системы. Сначала Вы используете реализационно-зависимую процедуру, чтобы архивировать текущее содержание вашей базы данных, поэтому копия ее содержания где-нибудь будет сохранена. Затем вы просматриваете протокол изменений сделанных в базе данных. Он сохраняется в некоторой области памяти, но не в главной памяти базы данных а желательно на отдельном устройстве, и содержит список всех команд которые произвели изменения в структуре или в содержании базы данных. Если у вас вдруг появились проблемы и текущее содержание вашей базы данных оказалось нарушенным, вы можете повторно выполнить все изменения зарегистрированные в протоколе на резервной копии вашей базы данных, и снова привести вашу базу данных в состояние, которое было до момента последней записи в протокол. Типичной командой, чтобы начать протоколирование, будет следующая:

```
SET JOURNAL ON;
```

Auditing используется с целью защиты. Она следит за тем, кто и какие действия выполнял в базе данных, и сохраняет эту информацию в таблице, доступной только очень немногим высоко привилегированным пользователям. Конечно, вы редко будете прибегать к процедуре ревизии, потому что очень скоро она займет много памяти и вам будет сложно работать в вашей базе данных. Но вы можете устанавливать ревизию для определенных пользователей, определенных действий или определенных объектов данных. Имеется такая форма команды AUDIT:

```
AUDIT INSERT ON Salespeople BY Diane;
```

Или предложение ON, или предложение BY могут быть исключены, устанавливая ревизию либо всех объектов, или всех пользователей, соответственно. Применение AUDIT ALL, вместо AUDIT INSERT, приведет к отслеживанию всех действий Diane в таблице Продавцов.

Приложение D

СПРАВОЧНИК ПО КОМАНДАМ И СИНТАКСИСУ

ЭТО ПРИЛОЖЕНИЕ СОДЕРЖИТ БОЛЕЕ КРАТКОЕ описание различных команд SQL. Цель состоит в том, чтобы дать вам быструю и точную ссылку и определение SQL. Первый раздел этого приложения определяет элементы, используемые для создания команд SQL; второй, подробности синтаксиса и предложения с кратким описанием самих команд. Далее показаны стандартные условные обозначения (они называются BNF условиями):

* Ключевые слова набираются в верхнем регистре.

* SQL и другие специальные условия заключаются в угловые скобки и набираются курсивом (**<and>**).

* Необязательные части команд находятся в квадратных скобках (**[and]**).

* Многоточие (**....**) указывает на то, что предшествующая часть команды может повторяться любое число раз.

* Вертикальная полоса (**|**) означает что то, что ей предшествует, может быть заменено на то, что следует за ней.

* Фигурные Скобки (**{and}**) указывают — все что внутри них, должно быть расценено как целое, для оценки других символов (например, вертикальных полос или эллипсов).

* Двойное двоеточие и равняется (**::=**) означают что то, что следует за ними, является определением того, что им предшествует.

Кроме того, мы будем использовать следующую последовательность (**.,..**) чтобы указывать, что предшествующее этому может повторяться любое число раз с индивидуальными событиями отделяемыми запятыми. Атрибуты, которые не являются частью официального стандарта, будут отмечены как (**нестандартные**) в описании.

ОБРАТИТЕ ВНИМАНИЕ: Терминология которую мы используем здесь, не официальная терминология ANSI. Официальная терминология может вас сильно запутать, поэтому мы несколько ее упростили.

По этой причине мы иногда используем условия, отличающиеся от ANSI, или используем те же самые условия, но несколько по-другому. Например, наше определение ***<predicate>*** отличается от используемой в ANSI комбинации стандартного определения ***<predicate>*** с ***<search condition>***.

SQL ЭЛЕМЕНТЫ

Этот раздел определяет элементы команд SQL. Они разделены на две категории: Основные элементы языка и Функциональные элементы языка.

Основные элементы — это создаваемые блоки языка; когда SQL исследует команду, то он сначала оценивает каждый символ в тексте команды в терминах этих элементов. Разделители ***<separator>*** отделяют одну часть команды от другой; все что находится между разделителями <*separator*> обрабатывается как модуль. Основываясь на этом разделении, SQL и интерпретирует команду.

Функциональные элементы — это разнообразные вещи, отличающиеся от ключевых слов, которые могут интерпретироваться как модули. Это — части команды, отделяемые с помощью разделителей <*separator*>, имеющих специальное значение в SQL. Некоторые из них являются специальными для определенных команд и будут описаны вместе с этими командами позже, в этом приложении. Перечисленное здесь, является общими элементы для всех описываемых команд. Функциональные элементы могут определяться в терминах друг друга или даже в собственных терминах. Например, предикат <*predicate*>, наш последний и наиболее сложный случай, содержит предикат <*predicate*> внутри собственного определения. Это потому, что предикат

<*predicate*> использующий AND или OR может содержать любое число предикатов <*predicate*> которые могут работать автономно.

Мы представляли вам предикат <*predicate*> в отдельной секции в этом приложении из-за разнообразия и сложности этого функционального элемента языка. Он будет постоянно присутствовать при обсуждении других функциональных частей команд.

БАЗОВЫЕ ЭЛЕМЕНТЫ ЯЗЫКА

ЭЛЕМЕНТ	*ОПРЕДЕЛЕНИЕ*
<separator>	<comment> | <space> | <newline>
<comment>	--<string> <newline>
<space>	пробел
<newline>	реализационно-определяемый конец символьной строки
<identifier>	<letter>[{<letter or digit> | <underscore>}...]

ИМЕЙТЕ ВВИДУ: Следуя строгому стандарту ANSI, символы должны быть набраны в верхнем регистре, а индификатор <identifier> не должен быть длиннее 18-ти символов.

<underscore>	-
<percent sign>	%
<delimiter>	любое из следующих: , ( ) < > . : = + " - | <> > = < = или <string>
<string>	[любой печатаемый текст в одиночных кавычках]

Примечание: В <string>, две последовательных одиночных кавычки ('') интерпретируются как одна (').

<SQL term> окончание, зависящее от главного языка. **(*только вложеный*)**

ФУНКЦИОНАЛЬНЫЕ ЭЛЕМЕНТЫ

Следующая таблица показывает функциональные элементы команд SQL и их определения:

ЭЛЕМЕНТ	*ОПРЕДЕЛЕНИЕ*
<query>	Предложение SELECT
<subquery>	Заключеное в круглых скобках предложение SELECT внутри другого условия, которое, фактически, оценивается отдельно для каждой строки-кандидата другого предложения.
<value expression>	*<primary>* | *<primary> <operator> <primary>* | *<primary> <operator> <value expression>*
<operator>	любое из следующих: + - / *
<primary>	*<column name>* | *<literal>* | *<aggregate function>* | *<built-in constant>* | *<nonstandard function>*
<literal>	*<string>* | *<mathematical expression>*
<built-in constant>	*USER* | *<implementation-defined constant>*
<table name>	*<identifier>*
<column spec>	*[<table name>* | *<alias>]<column n*ame>
<grouping column>	*<column spec>* | *<integer>*
<ordering column>	*<column spec>* | *<integer>*
<colconstraint>	*NOT NULL* | *UNIQUE* | *CHECK (<predicate>)* | *PRIMARY KEY* | *REFERENCES <table name>[(<column name>)]*

<tabconstraint>	*UNIQUE (<column list>) | CHECK (<predicate>) | PRIMARY KEY (<column list>) | FOREIGN KEY (<column list>) REFERENCES <table name>[(<column list>)]*
<defvalue>	ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ = *<value expression>*
<data type>	Допустимый тип данных (См. Приложение B для описания типов обеспечиваемых ANSI или Приложение C для других общих типов.)
<size>	Значение зависит от *<data type>* (См. Приложение B.)
<cursor name>	*<identifier>*
<index name>	*<identifier>*
<synonym>	*<identifier> (*nonstandard*)*
<owner>	*<Authorization ID>*
<column list>	*<column spec> .,..*
<value list>	*<value expression> .,..*
<table reference>	*{ <table name> [<alias>] } .,..*

ПРЕДИКАТЫ

Следующее определяет список различных типов предиката <predicate> описаных на следующих страницах:

```
<predicate> ::= [NOT]{ <comparison predicate> | <in predicate> | <null
predicate> | <between predicate> | <like predicate> | <quantified
predicate> | <exists predicate> } [AND | OR <predicate>]
```

<predicate> — это выражение, которое может быть верным, неверным или неизвестным, за исключением <exists predicate> и <null predicate>, которые могут быть только верными или неверными.

Будет получено *неизвестно* если NULL значения предотвращают вывод полученного ответа. Это будет случаться всякий раз, когда NULL значение сравнивается с любым значением.

Стандартные операторы Буля — **AND**, **OR** и **NOT** — могут использоваться с предикатом *<predicate>*. NOT верно = неверно, NOT неверно = верно, а NOT неизвестно = неизвестно. Результаты AND и OR в комбинации с предикатами, показаны в следующих таблицах:

AND	Верно	Неверно	Неизвестно
Верно	верно	неверно	неизвестно
Неверно	неверно	неверно	неверно
Неизвестно	неизвестно	неверно	неизвестно

OR	Верно	Неверно	Неизвестно
Верно	верно	верно	верно
Неверно	верно	неверно	неизвестно
Неизвестно	верно	неизвестно	неизвестно

Эти таблицы читаются способом на подобии таблицы умножения: вы объединяете верные, неверные, или неизвестные значения из строк с их столбцами, чтобы на перекрестье получить результат. В таблице AND, например, третий столбец (**Неизвестно**) и первая строка (**Верно**) на пересечении в верхнем правом углу дают результат — неизвестно, другими словами: **Верно AND Неизвестно = неизвестно**.

Порядок вычислений определяется круглыми скобками. Они не представляются каждый раз. NOT оценивается первым, далее AND и OR. Различные типы предикатов<*predicate*> рассматриваются отдельно в следующем разделе.

<comparison predicate> *(предикат сравнения)*

Синтаксис

```
<value expresslon> <relational op> <value expresslon> | <subquery>
<relatlonal op> ::= = | < | > | < | >=  | <>
```

Если либо <*value expression*> = NULL, либо <*comparison predicate*> = неизвестно; другими словами, это верно если сравнение верно или неверно если сравнение неверно.

<*relational op*> имеет стандартные математические значения для числовых значений; для других типов значений, эти значения определяются конкретной реализацией. Оба <*value expression*> должны иметь сравнимые типы данных. Если подзапрос <*subquery*> используется, он должен содержать одно выражение <*value expression*> в предложении SELECT, чье значение будет заменять второе выражение <*value expression*> в предикате сравнения <*comparision predicate*>, каждый раз когда <subquery> действительно выполняется.

<between predicate>

Синтаксис

```
<value expression> [NOT] BETWEEN <value expression> AND <value
expression>
```

<*between predicate*> — A BETWEEN B AND C, имеет такое же значение что и <*predicate*> — (A >= B AND <= C). <*between predicate*> для которого A NOT BETWEEN B AND C, имеет такое же значение что и NOT (BETWEEN B AND C). <*value expression*> может быть выведено с помощью нестандартного запроса <*subquery*> (*nonstandard*).

<in prediicate>

Синтаксис

```
<value expression> [NOT] IN <value list> | <subquery>
```

Список значений<*value list*> будет состоять из одного или более перечисленых значений в круглых скобках и отделяемых запятыми, которые имеют сравнимый с <*value expression*> тип данных. Если используется подзапрос <*subquery*>, он должен содержать только одно выражение <*value expression*> в предложении SELECT (возможно и больше, но это уже будет вне стандарта ANSI). Подзапрос <*subquery*> фактически, выполняется отдельно для каждой строки-кандидата основного запроса, и значения которые он выведет, будут составлять список значений <*value list*> для этой строки. В любом случае, предикат <*in predicate*> будет верен если выражение <*value expression*> представленное в списке значений <*value list*>, если не указан NOT. Фраза A NOT IN (B, C) является эквивалентом фразы NOT (A IN (B, C)).

<like predicate>

Синтаксис

```
<charvalue> [NOT] LIKE <pattern> [ESCAPE <escapechar>]
```

<*charvalue*> — это любое *нестандартное* выражение <*value expression*> алфавитно-цифрового типа. <*charvalue*> может быть, в сооветствии со стандартом, только определенным столбцом <*column spec*>. Образец <*pattern*> состоит из строки <*string*> которая будет проверена на совпадение с <*charvalue*>. Символ окончания <*escapechar*> — это одиночный алфавитно-цифровой символ. Совпадение произойдет, если верны следующие условия:

* Для каждого символа подчеркивания <*underscore*> в образце <pattern> которая не предшесивует символу окончания <*escapechar*>, имеется один соответствующий ему символ <*charvalue*>.

* Для каждого <*percent sign*> в образце <*pattern*> который не предшествует <*escapechar*>, имеются нули или более соответствующие символы в <*charvalue*>.

* Для каждого <*escapechar*> в <*pattern*> который не предшествует другому <escapechar>, нет никакого соответствующего символа в <*charvalue*>.

* Для каждого иного символа в <*pattern*>, один и тот же символ устанавливается у соответствующей отметке в <*charvalue*>.

Если совпадение произошло, <*like predicate*> — верен, если не был указан NOT. Фраза NOT LIKE 'текст' эквивалентна NOT (A LIKE 'текст').

<null predicate>

Синтаксис

```
<column spec> IS [NOT] NULL
```

<*column spec*> = IS NULL, если NULL значение представлено в этом столбце. Это сделает <*null predicate*> верным если не указан NULL. Фраза <*column spec*> IS NOT NULL имеет тот же результат, что и NOT(<*column spec*> IS NULL).

<quantified predicate>

Синтаксис

```
<value expression> <relational op> <quantifier> <subquery> <quantifier>
::= ANY | ALL | SOME
```

Предложение SELECT подзапроса <*subquery*> должно содержать одно и только одно выражение значения <*value expression*>. Все значения выведенные подзапросом <*subquery*> составляют набор результатов <*result set*>. <*value expression*> сравнивается, используя оператор связи <*relational operator*>, с каждым членом набора результатов<*result set*>. Это сравнение оценивается следующим образом:

* Если <*quantifier*> = ALL, и каждый член набора результатов <*result set*> делает это сравнение верным, <*quantified predicate*> — верен.

* Если <*quantifier*> = ANY, и имеется по крайней мере один член из набора результатов <*result set*>, который делает верным это сравнение, то <*quantified predicate*> является верным.

* Если набор результатов <*result set*> пуст, то <*quantified predicate*> верен, если <*quantifier*> = ALL, и неверен если иначе.

* Если <*quantifier*> = SOME, эффект — тот же что и для ANY.

* Если <*quantified predicate*> не верен и не неверен, он — неизвестен.

<exists predicate>

Синтаксис:

```
EXISTS (<subquery>)
```

Если подзапрос <*subquery*> выводит одну или более строк вывода, <*exists predicate*> — верен; и неверен, если иначе.

SQL КОМАНДЫ

Этот раздел подробно описывает синтаксис различных команд SQL. Это даст вам возможность быстро отыскивать команду, находить ее синтаксис и краткое описание ее работы.

ИМЕЙТЕ ВВИДУ: Команды, которые начинаются словами EXEC SQL, а также команды или предложения заканчивающиеся словом — <SQL term> могут использоваться только во вложенном SQL.

BEGIN DECLARE SECTION
(НАЧАЛО РАЗДЕЛА ОБЪЯВЛЕНИЙ)

Синтаксис

```
EXEC SQL BEGIN DECLARE SECTION <SQL term>
<host-language variable declarations>
EXEC SQL END DECLARE SECTION<SQL term>
```

Эта команда создает раздел программы главного языка для обьявления в ней главных переменных, которые будут использоваться во вкладываемых операторах SQL. Переменная SQLCODE должна быть включена как одна из обьявляемых переменных главного языка.

CLOSE CURSOR
(ЗАКРЫТЬ КУРСОР)

Синтаксис

```
EXEC SQL CLOSE CURSOR <cursor name> <SQL term>;
```

Эта команда указывает курсору закрыться, после чего ни одно значение не сможет быть выбрано из него до тех пор пока он не будет снова открыт.

COMMIT (WORK)
(ФИКСАЦИЯ (ТРАНЗАКЦИИ))

Синтаксис

```
COMMIT WORK;
```

Эта команда оставляет неизменными все изменения, сделанные в базе данных, до тех пор, пока начавшаяся транзакция не закончится, и не начнется новая транзакция.

CREATE INDEX (*NONSTANDARD*)
(СОЗДАТЬ ИНДЕКС) (*НЕСТАНДАРТНО*)

Синтаксис

```
CREATE [UNIQUE] INDEX <Index name> ON <table name> (<column list>);
```

Эта команда создает эффективный маршрут с быстрым доступом для поиска строк содержащих обозначенные столбцы. Если UNIQUE — указана, таблица не сможет содержать дубликатов (двойников) значений в этих столбцах.

CREATE SYNONYM (*NONSTANDARD*)
(СОЗДАТЬ СИНОНИМ) (*НЕСТАНДАРТНО*)

Синтаксис

```
CREATE IPUBLICl SYNONYM <synonym> FOR <owner>.<table name>;
```

Эта команда создает альтернативное (синоним) имя для таблицы. Синоним принадлежит его создателю, а сама таблица, обычно другому пользователю. Используя синоним, его владелец может не ссылаться к таблице ее полным (включая имя владельца) именем. Если PUBLIC — указан, синоним принадлежит каталогу SYSTEM и следовательно доступен всем пользователям.

CREATE TABLE
(СОЗДАТЬ ТАБЛИЦУ)

Синтаксис

```
CREATE TABLE <table name> ({<column name> <data type>[<size>]
[<colconstralnt> ...] [<defvalue>]} .,.. <tabconstraint> .,..);
```

Команда создает таблицу в базе данных. Эта таблица будет принадлежать ее создателю. Столбцы будут рассматриваться в поименном порядке.

<*data type*> определяет тип данных который будет содержать столбец. Стандарт <*data type*> описывается в Приложении B; все прочие используемые типы данных <*data type*>, обсуждались в Приложении C. Значение размера <*size*> зависит от типа данных <*data type*>.

<*colconstraint*> и <*tabconstraint*> налагают ограничения на значения, которые могут быть введены в столбце.

<*defvalue*> определяет значение (по умолчанию) которое будет вставлено автоматически, если никакого другого значения не указано для этой строки. (См. Главу 17 для подробностей о самой команде CREATE TABLE и Главы 18 И 19 для подробностей об ограничениях и о <*defvalue*>).

CREATE VIEW

(СОЗДАТЬ ПРОСМОТР)

Синтаксис

```
CREATE VIEW <table name> AS <query> [WITH CHECK OPTION];
```

Просмотр обрабатывается как любая таблица в командах SQL. Когда команда ссылается на имя таблицы <*table name*>, запрос <*query*> выполняется, и его вывод соответствует содержанию таблицы указнной в этой команде.

Некоторые просмотры могут модифицироваться, что означает, что команды модификации могут выполняться в этих просмотрах и передаваться в таблицу, на которую была ссылка в запросе <*query*>. Если указано предложение WITH CHECK OPTION, эта модификация должна также удовлетворять условию предиката <*predicate*> в запросе <*query*>.

DECLARE CURSOR
(ОБЬЯВИТЬ КУРСОР)

Синтаксис

```
EXEC SQL DECLARE <cursor name> CURSOR FOR <query><SQL term>
```

Эта команда связывает имя курсора <*cursor name*>, с запросом <*query*>. Когда курсор открыт (см. OPEN CURSOR), запрос <*query*> выполняется, и его результат может быть выбран (командой FETCH) для вывода. Если курсор модифицируемый, таблица на которую ссылается запрос <*query*>, может получить изменение содержания с помощью операции модификации в курсоре (См. Главу 25 о модифицируемых курсорах).

DELETE
(УДАЛИТЬ)

Синтаксис

```
DELETE FROM <table name> {[WHERE <predicate>];} | WHERE CURRENT OF
<cursor name><SQL term>
```

Если предложение WHERE отсутствует, ВСЕ строки таблицы удаляются. Если предложение WHERE использует предикат <*predicate*>, строки, которые удовлетворяют условию этого предиката <*predicate*> удаляются. Если предложение WHERE имеет аргумент CURRENT OF (ТЕКУЩИЙ) в имени курсора <*cursor name*>, строка из таблицы <*table name*> на которую в данный момент имеется ссылка с помощью имени курсора <*cursor name*> будет удалена. Форма WHERE CURRENT может использоваться только во вложенном SQL, и только с модифицируемыми курсорами.

EXEC SQL
(ВЫПОЛНИТЬ SQL)

Синтаксис

```
EXEC SQL <embedded SQL command> <SQL term>
```

EXEC SQL используется, чтобы указывать начало всех команд SQL, вложенных в другой язык.

FETCH
(ВЫБОРКА)

Синтаксис

```
EXEC SQL FETCH <cursorname> INTO <host-varlable llst><SQL term>
```

FETCH принимает вывод из текущей строки запроса <*query*>, вставляет ее в список главных переменных <*host-variable list*>, и перемещает курсор на следующую строку. Список <*host-variable list*> может включать переменную indicator в качестве целевой переменной (См. Главу 25.)

GRANT
(ПЕРЕДАТЬ ПРАВА)

Синтаксис (стандартный)

```
GRANT ALL [PRIVILEGES] | {SELECT | INSERT | DELETE | UPDATE [(<column
llst>)] | REFERENCES [(<column llst>)l } .,.. ON <table name> .,.. TO
PUBLIC | <Authorization ID> .,.. [WITH GRANT OPTION];
```

Аргумент **ALL** (ВСЕ), с или без **PRIVILEGES** (ПРИВИЛЕГИИ), включает каждую ривилегию в список привилегий. **PUBLIC** (ОБЩИЙ) включает всех существующих пользователей и всех созданых в будущем.

Эта команда дает возможность передать права для выполнения действий в таблице с указаным именем. **REFERENCES** позволяет дать права чтобы использовать столбцы в списке столбцов <*column list*> как родительский ключ для внешнего ключа. Другие привилегии состоят из права выполнять команды, для которых привилегии указаны их именами в таблице. UPDATE подобен REFERENCES и может накладывать ограничения на определенные столбцы. **GRANT OPTION** дает возможность передавать эти привилегии другим пользователям.

Синтаксис (нестандартный)

```
GRANT DBA | RESOURCE | CONNECT .... TO <Authorization ID> .,..
[IDENTIFIED BY <password>]
```

CONNECT дает возможность передавать право на регистрации и некоторые другие ограниченные права.

RESOURCE дает пользователю право создавать таблицы. **DBA** дает возможность передавать почти все права.

IDENTIFIED BY используется вместе с CONNECT, для создания или изменения пароля пользователя.

INSERT
(ВСТАВКА)

Синтаксис

```
INSERT INTO <table name> (<column llst>) VALUES (<value llst>) | <query>;
```

INSERT создает одну или больше новых строк в таблице с именем <*table name*>. Если используется предложение VALUES , их значения вставляются в таблицу с именем <*table name*>. Если запрос <*query*> указан, каждая строка вывода будет вставлена в таблицу с именем <*table name*>. Если список столбцов <*column list*> отсутствует, все столбцы таблицы <*table name*>, принимаются в упорядоченном виде.

OPEN CURSOR
(ОТКРЫТЬ КУРСОР)

Синтаксис

```
EXEC SQL OPEN CURSOR <cursorname> <SQL term>
```

OPEN CURSOR выполняет запрос связанный с курсором <*cursor name*>. Вывод может теперь извлекать по одной строке для каждой команды FETCH.

REVOKE (*NONSTANDARD*)
(ОТМЕНИТЬ ПОЛНОМОЧИЯ) (НЕСТАНДАРТНО)

Синтаксис

```
REVOKE {ALL [PRIVILEGES] | <privilege> .,..} [ON <table name>] FROM {
PUBLIC | <Authorization ID> .,..};
```

Привелегия <*privelege*> может быть любой из указаных в команде GRANT. Пользователь, дающий REVOKE, должен иметь те же привелегии, что и пользователь, который давал GRANT. Предложение ON может быть использовано, если используется привилегия специального типа для особого объекта.

ROLLBACK (WORK)
(ОТКАТ) (ТРАНЗАКЦИИ)

Синтаксис

```
ROLLBACK WORK;
```

Команда отменяет все изменения в базе данных, сделанные в течение текущей транзакции. Она, кроме того, заканчивает текущую и начинает новую транзакцию.

SELECT
(ВЫБОР)

Синтаксис

```
SELECT { IDISTINCT | ALL] < value expression > . , . . } / *
[INTO <host variable list> (*embedded only*)]
FROM < table reference > . , . .
[WHERE <predicate>]
[GROUP BY <grouping column> . , . .]
[HAVING <predicate>]
[ORDER BY <ordering column> [ASC | DESC] . , . . ];
```

Это предложение огранизует запрос и выводит значения из базы данных (см. Глава 3 — Глава 14). Применяются следующие правила:

* Если ни ALL, ни DISTINCT — не указаны, принимается — ALL.

* Выражение <*value expression*> состоит из <*column spec*>, агрегатной функции <*aggregate funct*>, нестандартной функции <*nonstandard function*>, постоянной <*constant*>, или любой их комбинации с операторами в допустимых выражениях.

* Ссылаемая таблица <*table reference*> состоит из имени таблицы, включая префикс владельца, если текущий пользователь не владелец, или синоним (нестандартно) для таблицы. Таблица может быть или базовой таблицей или просмотром. В принципе, псевдоним может указать, какой синонимом используется для таблицы только на время текущей команды. Имя таблицы или синоним должны отделяться от псевдонима одним или более разделительными знаками <*separator*>.

* Если используется GROUP BY, все столбцы <*column spec*> используемые в предложении SELECT, должны будут использоваться как группа столбцов <*grouping column*>, если они не содержатся в агрегатной функции <*aggregate funct*>. Вся группа столбцов <*grouping column*> должна быть представлена среди выражений <*value expressions*> указанных в предложении SELECT. Для каждой отдельной комбинации значений группы столбцов <*grouping column*>, будет иметься одна и только одна строка вывода.

* Если HAVING используется, предикат <*predicate*> применяется к каждой строке произведенной предложением GROUP BY, и те строки которые сделают этот предикат верным, будут выведены.

* Если ORDER BY используется, вывод имеет определенную последовательность. Каждый идентификатор столбца <*column identifer*> ссылается к указанному <*value expression*> в предложении SELECT. Если это <*value expression*> является указанным столбцом <*column spec*>, <*column identifier*> может быть таким же как <*column spec*>. Иначе <*column identifier*> может быть положительным целым числом, указывающим место где находится <*value expression*> в последовательности предложения SELECT. Вывод будет сформирован так чтобы помещать значения содержащиеся в <*column identifier*> в порядке возрастания, если DESC не указан. Имя идентификатора столбца <*column identifier*>, стоящее первым в предложении ORDER BY будет предшествовать позже стоящим именам в определении последовательности вывода.

Предложение SELECT оценивает каждую строку-кандидат таблицы в которой строки показаны независимо. Строка-кандидат определяется следующим образом:

* Если только одна ссылаемая таблица <*table reference*> включена, каждая строка этой таблица в свою очередь является строкой-кандидатом.

* Если более одной ссылаемой таблицы <*table reference*> включено, каждая строка каждой таблицы должна быть скомбинирована в свою очередь с каждой комбинацией строк из всех других таблиц. Каждая такая комбинация будет в свою очередь строкой-кандидатом.

Каждая строка-кандидат производит значения, которые делают предикат <*predicate*> в предложении WHERE верным, неверным, или неизвестным. Если GROUP BY не используется, каждое <*value expression*> применяется в свою очередь для каждой строки-кандидата, чье значение делает предикат верным, и результатом этой операции является вывод. Если GROUP BY используется, строки-кандидаты

комбинируются, используя агрегатные функции. Если никакого предиката <*predicate*> не установлено, каждое выражение <*value expression*> применяется к каждой строке-кандидату или к каждой группе. Если указан DISTINCT, дубликаты (двойники) строк будут удалены из вывода.

UNION
(ОБЪЕДИНЕНИЕ)

Синтаксис

```
<query> {UNION [ALL] <query> } . . . ;
```

Вывод двух или более запросов <*query*> будет объединен. Каждый запрос <*query*> должен содержать один и тот же номер <*value expression*> *в* предложение SELECT и в таком порядке что 1..n каждого, совместим по типу данных <*data type*> и размеру <*size*> с 1..n всех других.

UPDATE
(МОДИФИКАЦИЯ)

Синтаксис

```
UPDATE <table name>
SET { <column name> = <value expression> } .,..
{[ WHERE <predlcate>]; } | {[WHERE CURRENT OF <cursorname>] <SQL term>]}
```

UPDATE изменяет значения в каждом столбце с именем <*column name*> на соответствующее значение <*value expression*>. Если предложение WHERE использует предикат <*predicate*>, то только строки таблиц чьи текущие значения делают тот предикат <*predicate*> верным, могут быть изменены. Если WHERE использует предложение CURRENT OF, то значения в строке таблицы с именем <*table name*> находящиеся в курсоре с именем <*cursor name*> меняются. WHERE CURRENT OF пригодно для использования только во вложенном SQL, и только с модифицируемыми курсорами. При отсутствия предложения WHERE — все строки меняются.

WHENEVER
(ВСЯКИЙ РАЗ КАК)

Синтаксис

```
EXEC SQL WHENEVER <SQLcond> <actlon> <SQL term>
<SQLcond> ::= SQLERROR | NOT FOUND | SQLWARNING
(последнее — нестандартное)
<action> ::= CONTINUE | GOTO <target> | GOTO <target>
<target> ::= зависит от главного языка
```

Приложение E

ТАБЛИЦЫ, ИСПОЛЬЗУЕМЫЕ В SQL

ТАБЛИЦА 1: ПРОДАВЦЫ

snum	sname	city	comm
1001	Peel	London	.12
1002	Serres	San Jose	.13
1004	Motika	London	.11
1007	Rifkin	Barcelona	.15
1003	Axelrod	New York	.10

ТАБЛИЦА 2: ЗАКАЗЧИКИ

cnum	cname	city	rating	snum
2001	Hoffman	London	100	1001
2002	Giovanni	Rome	200	1003
2003	Liu	San Jose	200	1002
2004	Grass	Berlin	300	1002
2006	Clemens	London	100	1001
2008	Cisneros	San Jose	300	1007
2007	Pereira	Rome	100	1004

ТАБЛИЦА 3: ПОРЯДКИ

onum	amt	odate	cnum	snum
3001	18.69	10/03/1990	2008	1007
3003	767.19	10/03/1990	2001	1001
3002	1900.10	10/03/1990	2007	1004
3005	5160.45	10/03/1990	2003	1002
3006	1098.16	10/03/1990	2008	1007
3009	1713.23	10/04/1990	2002	1003
3007	75.75	10/04/1990	2004	1002
3008	4723.00	10/05/1990	2006	1001
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

СОДЕРЖАНИЕ