Мартин Грабер

"Книга является новым руководством, предназначенным для начинающих"
Computer Book Review

SQL для простых смертных

Самое простое введение в SQL с полезными примерами и подробными объяснениями

От манипулирования значениями, таблицами и запросами до управления переменными с неявными областями значений и связанными подзапросами — все это вы найдете в этой книге

Описание всех версий стандарта SQL

Understanding SQL

MARTIN GRUBER

SQL для простых смертных

Мартин Грабер

Издательство "ЛОРИ"

Understanding SQL.
by Martin Gruber.
© Copyright All rights reserved

SQL для простых смертных.
Мартин Грабер.
Переводчик В.А.Ястребов
Научный редактор П.И.Быстров.
Верстка М.Алиевой.

Copyright © 1990 SYBEX Inc., 2021 Challenger Drive,
Alameda, CA 94501.

Перевод © Издательство «ЛОРИ», 2014

Посвящается Ли и Джанет Фесперман, предоставившим мне возможность полностью посвятить себя написанию этой книги.

БЛАГОДАРНОСТИ

Мне хотелось бы поблагодарить FFF Software за разрешение воспользоваться FirstSQL при подготовке этой книги.

Содержание

Введение xi

Глава 1. Введение в реляционные базы данных . . . 1
- Что такое реляционная база данных? 3
- Пример базы данных . . . 5
- Итоги . . . 7

Глава 2. Введение в SQL . . . 9
- Как работает SQL? . . . 10
- Различные типы данных . . . 12
- Итоги . . . 15

Глава 3. Использование SQL для выборки данных из таблиц . . . 17
- Формирование запроса . . . 18
- Определение выборки — предложение WHERE . . . 24
- Итоги . . . 26

Глава 4. Использование реляционных и булевых операторов для создания более сложных предикатов . . . 29
- Реляционные операторы . . . 30
- Булевы операторы . . . 32
- Итоги . . . 37

Глава 5. Использование специальных операторов в "условиях" . . . 39
- Оператор IN . . . 40
- Оператор BETWEEN . . . 41
- Оператор LIKE . . . 44
- Оператор IS NULL . . . 47
- Итоги . . . 49

Глава 6. Суммирование данных с помощью функций агрегирования 51
- Что такое функции агрегирования? . . . 52
- Итоги . . . 61

Глава 7. Форматирование результатов запросов . . . 63
- Строки и выражения . . . 64
- Упорядочение выходных полей . . . 67
- Итоги . . . 71

Глава 8. Использование множества таблиц в одном запросе 75
Соединение таблиц 76
Итоги 81

Глава 9. Операция соединения, операнды которой представлены одной таблицей 83
Как выполняется операция соединения двух копий одной таблицы 84
Итоги 90

Глава 10. Вложение запросов 93
Как выполняются подзапросы? 94
Итоги 105

Глава 11. Связанные подзапросы 107
Как формировать связанные подзапросы 108
Итоги 115

Глава 12. Использование оператора EXISTS 117
Как работает оператор EXISTS? 118
Использование EXISTS со связанными подзапросами 119
Итоги 124

Глава 13. Использование операторов ANY, ALL и SOME 127
Специальный оператор ANY или SOME 128
Специальный оператор ALL 135
Функционирование ANY, ALL и EXISTS при потере данных или
с неизвестными данными 139
Итоги 143

Глава 14. Использование предложения UNION 145
Объединение множества запросов в один 146
Использование UNION с ORDER BY 151
Итоги 157

Глава 15. Ввод, удаление и изменение значений полей 159
Команды обновления DML 160
Ввод значений 160
Исключение строк из таблицы 162
Изменение значений полей 163
Итоги 165

Глава 16. Использование подзапросов с командами обновления 167
Использование подзапросов в INSERT 168
Использование подзапросов с DELETE 170

Использование подзапросов с UPDATE 173
Итоги 174

Глава 17. Создание таблиц 177
Команда CREATE TABLE 178
Индексы 179
Изменение таблицы, которая уже была создана 181
Исключение таблицы 182
Итоги 183

Глава 18. Ограничения на множество допустимых значений данных 185
Ограничения в таблицах 186
Итоги 195

Глава 19. Поддержка целостности данных 197
Внешние и родительские ключи 198
Ограничения FOREIGN KEY (внешнего ключа) 199
Что происходит при выполнении команды обновления 204
Итоги 209

Глава 20. Введение в представления 211
Что такое представления? 212
Команда CREATE VIEW 212
Итоги 221

Глава 21. Изменение значений с помощью представлений 223
Обновление представлений 224
Выбор значений, размещенных в представлениях 228
Итоги 232

Глава 22. Определение прав доступа к данным 235
Пользователи 236
Передача привилегий 237
Лишение привилегий 241
Другие типы привилегий 245
Итоги 247

Глава 23. Глобальные аспекты SQL 249
Переименование таблиц 250
Каким образом база данных размещается для пользователя? 252
Когда изменения становятся постоянными? 253
Как SQL работает одновременно с множеством пользователей 255
Итоги 259

Глава 24. Как поддерживается порядок в базе данных SQL 261
Системный каталог 262
Комментарии к содержимому каталога 266
Оставшаяся часть каталога 268
Другие пользователи каталога 275
Итоги 276

Глава 25. Использование SQL с другими языками программирования (встроенный SQL) 279
Что включается во встроенный SQL? 280
Использование переменных языка высокого уровня с SQL 282
SQLCODE 288
Обновление курсоров 291
Индикаторы переменных 293
Итоги 296

Приложения

A. Ответы к упражнениям 301

B. Типы данных SQL 319
Типы ANSI 320
Эквивалентные типы данных в других языках 322

C. Некоторые общие отклонения от стандарта SQL 325
Типы данных 326
Команда FORMAT 328
Функции 330
Операции INTERSECT (пересечение) и MINUS (разность) 332
Автоматические OUTER JOINS (внешние соединения) 333
Ведение журнала 334

D. Справка по синтаксису и командам 337
Элементы SQL 338
Команды SQL 345

E. Таблицы, используемые в примерах 355

F. SQL сегодня 357
SQL сегодня 358

ВВЕДЕНИЕ

SQL (обычно произносится "SEQUEL") — структурированный язык запросов (Structured Query Language). Он позволяет создавать *реляционные* базы данных, представляющие собой набор связанных данных, хранящихся в таблицах, и оперировать ими.

Мир баз данных имеет тенденцию к постоянной интеграции, приведшей к необходимости разработки стандартного языка, пригодного для использования на множестве современных компьютерных платформ. Стандартный язык дает возможность пользователям освоить один набор команд и применять его для создания, поиска, изменения и передачи данных независимо от того, работает ли он на персональном компьютере, на рабочей станции или на большой вычислительной машине. В компьютерном мире пользователь, владеющий таким языком, имеет огромные возможности по применению и интеграции информации из множества разнообразных источников.

Благодаря своей элегантности и независимости от специфики компьютера, а также поддержке лидерами в области технологии реляционных баз данных, SQL стал и в ближайшем обозримом будущем останется таким стандартным языком. Именно по этой причине, тот, кто предполагает работать с базами данных в девяностые годы нашего столетия, должен владеть языком SQL.

Стандарт SQL определен американским национальным институтом стандартов (American National Standarts Institute) и в настоящее время принят также ISO (International Standards Organization) в качестве международного стандарта. Однако подавляющее большинство коммерческих программ, связанных с обработкой баз данных, расширяет возможности SQL за рамки того, что определено ANSI, добавляя полезные новые черты. Правда, иногда они нарушают стандарт в худшую сторону, тогда как хорошие идеи имеют тенденцию повторяться и становятся стандартом "де факто" или "рыночным" стандартом. В этой книге материал представлен в соответствии с ANSI-стандартом с учетом наиболее общих отклонений от него. Для того, чтобы обнаружить отличия от стандарта, можно воспользоваться документацией по программному обеспечению.

Кто может воспользоваться этой книгой?

Для чтения этой книги требуются минимальные знания из области компьютеров и баз данных. Использовать SQL проще, чем многие другие, менее компактные языки, поскольку при работе на SQL не определяются процедуры, необходимые для получения желаемого результата. Эта книга вводит в мир языка SQL последовательно, содержит множество примеров и упражнений к каждой главе, цель которых — отточить понима-

ние материала и мастерство. Можно выполнять полезные задания немедленно, и, по мере их выполнения, мастерство будет расти.

Поскольку SQL является частью многих программ, выполняющихся на различных компьютерах, никаких предположений относительно специфики использования языка не делается. Эта книга является самым общим пособием. Вы сможете непосредственно применить полученные знания в любой системе, использующей SQL.

Книга предназначена для новичков в области баз данных, однако SQL представлен в ней достаточно глубоко. Примеры отражают множество ситуаций, возникающих в реальных деловых областях приложения. Некоторые из них достаточно сложны, так как приводятся с целью показать все возможные варианты применения SQL.

Как организована эта книга?

Каждая глава вводит новую группу взаимосвязанных понятий и определений. Они базируются на рассмотренном ранее материале и содержат практические вопросы для закрепления полученных знаний. Ответы на практические вопросы приведены в приложении А.

Первые семь глав содержат основные понятия реляционных баз данных и SQL, за ними следуют основы запросов (queries). Запросы — команды, используемые для поиска данных в базах данных; они представляют собой наиболее общий и наиболее сложный аспект SQL. В главах с 8 по 14 техника запросов усложняется. Вводятся различные способы комбинирования запросов и запросы более чем к одной таблице. Другие аспекты SQL: создание таблиц, ввод в них значений, предоставление и закрытие доступа к созданным таблицам — рассмотрены в главах с 15 по 23. Глава 24 показывает, как получить доступ к информации о структуре базы данных. В главе 25 речь идет об использовании SQL в программах, написанных на других языках.

В зависимости от того, как будет использоваться SQL, часть информации, расположенной в конце книги, может не пригодиться. Не все пользователи создают таблицы или вводят в них значения. Эта книга построена таким образом, что каждая следующая глава продолжает предыдущую, но можно свободно пропускать те разделы, которые никогда не придется использовать. Именно по этой причине введение в запросы полностью представлено в начале книги. Запросы — это основа, необходимая для того, чтобы успешно применять большинство других функций SQL.

Во всем множестве примеров, представленных в книге, будет использоваться единый набор таблиц.

Содержимое книги по главам выглядит следующим образом:

- Глава 1 дает понятие реляционной базы данных и концепции первичных ключей (primary keys). В ней также приводятся и поясняются три таблицы, на которых базируется множество представленных в книге примеров.

- Глава 2 ориентирует вас в мире SQL. В ней рассматриваются важные вопросы структуры языка, различные типы данных, распознаваемые SQL, некоторые общие соглашения SQL и терминология.

- Глава 3 учит создавать запросы и знакомит с несколькими приемами по их уточнению. После изучения этой главы вы сможете использовать SQL с практической пользой.

- Глава 4 иллюстрирует, каким образом применяются в SQL два типа стандартных математических операторов, отношения (=, <, >, и т.д.) и булевы операции (AND, OR, NOT).

- Глава 5 вводит ряд операторов, которые используются так же, как операторы отношения, но являются специфичными для SQL. В этой главе даются разъяснения по вопросу потери данных, и определены NULL-значения.

- Глава 6 учит применять операторы, позволяющие выводить данные на основе тех, которые хранятся в таблицах, способом, отличным от простого извлечения. Это дает возможность суммировать значения данных, хранящихся в таблицах.

- Глава 7 поясняет ряд действий, возможных при выводе запроса: выполнение математических операций над данными, включение текста, сортировка.

- Глава 8 показывает, как простой запрос может извлекать информацию более чем из одной таблицы. Этот процесс определяет связь таблиц, включая способы оперирования с данными.

- Глава 9 демонстрирует технику получения ответа на запрос по множеству таблиц, применимую к установлению специальной связи для одной таблицы.

- Глава 10 научит выполнять запрос и использовать его результат в другом запросе.

- Глава 11 расширяет технику, рассмотренную в главе 10, и учит использовать вложенные запросы многократно.

- Глава 12 вводит новый тип специального оператора SQL. EXISTS — оператор, действующий на весь запрос, а не на отдельное простое значение.

- Глава 13 вводит новый тип операторов — ANY, ALL, SOME, которые, подобно оператору EXISTS, действуют на весь запрос.

- Глава 14 вводит команды, позволяющие непосредственно комбинировать результаты множественных запросов способом, отличным от их последовательного выполнения.

- Глава 15 вводит команды, позволяющие определить, какие значения хранятся в базе данных, а также команды вставки, удаления и обновления значений.

- Глава 16 расширяет мощность только что введенных команд. В ней показано, как запросы могут управлять их выполнением.

- Глава 17 учит создавать новую таблицу.

- Глава 18 детально объясняет процесс создания таблиц. Вы узнаете, как предусмотреть отказ от автоматического выполнения некоторого вида изменений.

- Глава 19 исследует логические связи, существующие между данными, на основе совпадения значений.

- Глава 20 рассказывает о представлениях, об "окне", разворачивающем таблицу, отличную от той, что хранится в базе данных.

- Глава 21 касается сложных вопросов изменения значений в представлениях, когда вы реально изменяете соответствующие таблицы. Именно с этим связана здесь необходимость рассмотрения специальных вопросов.

- Глава 22 рассказывает о привилегиях: кто имеет право обращаться с запросами к таблицам, кто имеет право изменять их содержимое, как эти права назначаются пользователям, как пользователи их лишаются и т.д.

- Глава 23 представляет некоторые ранее не рассмотренные важные моменты. Например, мы обсудим те изменения базы данных, которые становятся постоянными, а также выполнение ряда операций в SQL.

- Глава 24 описывает, как SQL поддерживает структурирование баз данных и каким образом осуществляется доступ к ним.

- Глава 25 фокусирует внимание на специальных проблемах и процедурах, связанных с вводом SQL-команд из других языков. Здесь же рассмотрены аспекты языка, специфичные для встроенной формы, например, курсоры и команда FETCH.

В приложениях вы найдете ответы на вопросы (приложение A), описание таблиц, рассматриваемых в качестве примеров (приложение B), детальные сведения о различных типах данных (приложение C), общие элементы, отличные от стандарта (приложение D), руководство по командам SQL (приложение E), взгляд на современный SQL (приложение F).

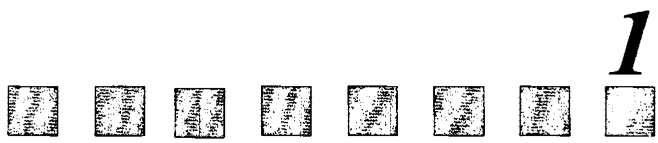
Соглашения, принятые в этой книге

SQL состоит из инструкций, которые передаются программе, управляющей работой базы данных, предлагая ей выполнить определенные действия. Эти инструкции в общем виде называют предложениями, но мы в большинстве случаев будем использовать термин "команды", чтобы показать, что они имеют область действия.

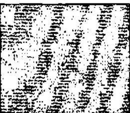
Термины выделены курсивом в тех местах, где они в первый раз встречаются. В синтаксисе команд курсив используется для того, чтобы показать, что слова имеют дополнительный смысл.

В примерах представлен текст, который следует ввести в программу обработки базы данных, и показан результат для конкретного программного продукта (FirstSQL, программа, работающая с базой данных на IBM PC). Результат, полученный с помощью других программных продуктов, может отличаться от приведенного, но основной результат (данные, полученные из базы данных) не зависит от конкретного программного продукта.

1



Введение в реляционные базы данных



Прежде чем начать использовать SQL, вы должны понять, что такое реляционная база данных. Мы намеренно не будем обсуждать в этой главе SQL, поэтому вы можете пропустить ее, если достаточно хорошо владеете основным понятиями реляционных баз данных. Однако в любом случае следует взглянуть на три таблицы, представленные в конце главы, поскольку именно они используются в большинстве примеров, приведенных в книге. Вы также можете ознакомиться с ними в приложениiE. Мы рекомендуем постоянно иметь копию этих таблиц перед глазами.

Что такое реляционная база данных?

Реляционная база данных — это связанная информация, представленная в виде двумерных таблиц. Представьте себе адресную книгу. Она содержит множество строк, каждая из которых соответствует данному индивидууму. Для каждого из них в ней представлены некоторые независимые данные, например, имя, номер телефона, адрес. Представим такую адресную книгу в виде таблицы, содержащей строки и столбцы. Каждая строка (называемая также *записью*) соответствует определенному индивидууму, каждый столбец содержит значения соответствующего типа данных: имя, номер телефона и адрес, — представленных в каждой строке. Адресная книга может выглядеть таким образом:

Name (**Имя**)	**Telephone** (**Телефон**)	**Address** (**Адрес**)
Gerry Farish	(415)365-8775	127 Primrose Ave., SF
Celia Brock	(707) 874-3553	246 #4 3rd St., Sonoma
Yves Grillet	(762)976-3665	778 Modernas, Barcelona

То, что мы получили, является основой реляционной базы данных, определенной в начале нашего обсуждения двумерной (строки и столбцы) таблицей информации. Однако, реляционная база данных редко состоит из одной таблицы, которая слишком мала по сравнению с базой данных. При создании нескольких таблиц со связанной информацией можно выполнять более сложные и мощные операции над данными. Мощность базы данных заключается, скорее, в связях, которые вы конструируете между частями информации, чем в самих этих частях.

Установление связи между таблицами

Давайте используем пример адресной книги для того, чтобы обсудить базу данных, которую можно реально использовать в деловой жизни. Предположим, что индивидуумы первой таблицы являются пациентами больницы. Дополнительную информацию о них можно хранить в другой таблице. Столбцы второй таблицы могут быть поименованы таким образом: Patient (Пациент), Doctor (Врач), Insurer (Страховка), Balance (Баланс).

Patient (Пациент)	Doctor (Врач)	Insurer (Страховка)	Balance (Баланс)
Farish	Drume	B.C./B.S.	$272.99
Grillet	Halben	None	$44.76
Brock	Halben	Health, Inc.	$9077.47

Можно выполнить множество мощных функций при извлечении информации из этих таблиц в соответствии с заданными критериями, особенно, если критерий включает связанные части информации из различных таблиц. Предположим, Dr. Halben желает получить номера телефонов всех своих пациентов. Для того чтобы извлечь эту информацию, он должен связать таблицу с номерами телефонов пациентов (адресную книгу) с таблицей, определяющей его пациентов. В данном простом примере он может мысленно проделать эту операцию и узнать телефонные номера своих пациентов Grillet и Brock, в действительности же эти таблицы вполне могут быть больше и намного сложнее. Программы, обрабатывающие реляционные базы данных, были созданы для работы с большими и сложными наборами тех данных, которые являются наиболее общими в деловой жизни общества. Даже если база данных больницы содержит десятки или тысячи имен (как это, вероятно, и бывает в реальной жизни), единственная команда SQL предоставит доктору Halben необходимую информацию практически мгновенно.

Порядок строк произволен

Для обеспечения максимальной гибкости при работе с данными строки таблицы, по определению, никак не упорядочены. Этот аспект отличает базу данных от адресной книги. Строки в адресной книге обычно упорядочены по алфавиту. Одно из мощных средств, предоставляемых реляционными системами баз данных, состоит в том, что пользователи могут упорядочивать информацию по своему желанию.

Рассмотрим вторую таблицу. Содержащуюся в ней информацию иногда удобно рассматривать упорядоченной по имени, иногда — в порядке возрастания или убывания баланса (Balance), а иногда — сгруппированной по доктору. Внушительное множество возможных порядков строк помешало бы пользователю проявить гибкость в работе с данными, поэтому строки предполагаются неупорядоченными. Именно по этой причине вы не можете просто сказать: "Меня интересует пятая строка таблицы". Независимо от порядка включения данных или какого-либо другого критерия, этой пятой строки не существует по определению. Итак, строки таблицы предполагаются расположенными в произвольном порядке.

Идентификация строк (первичный ключ)

По этой и ряду других причин, необходимо иметь столбец таблицы, который однозначно идентифицирует каждую строку. Обычно этот столбец содержит номер, например, приписанный каждому пациенту. Конечно, можно использовать для идентификации строк имя пациента, но ведь может случиться так, что имеется не-

сколько пациентов с именем Mary Smith. В подобном случае нет простого способа их различить. Именно по этой причине обычно используются номера. Такой уникальный столбец (или их группа), используемый для идентификации каждой строки и обеспечивающий различимость всех строк, называется *первичным ключом таблицы (primary key of the table)*.

Первичный ключ таблицы — жизненно важное понятие структуры базы данных. Он является сердцем системы данных: для того чтобы найти определенную строку в таблице, укажите значение ее первичного ключа. Кроме того, он обеспечивает целостность данных. Если первичный ключ должным образом используется и поддерживается, вы будете твердо уверены в том, что ни одна строка таблицы не является пустой и что каждая из них отлична от остальных. Ключи мы рассмотрим позже, после обсуждения ссылочной целостности (referential integrity) в главе 19.

Столбцы поименованы и пронумерованы

В отличие от строк, столбцы таблицы (также называемые *полями (fields)* упорядочены и поименованы. Следовательно, в нашей таблице, соответствующей адресной книге, можно сослаться на столбец "Address" как на "столбец номер три". Естественно, это означает, что каждый столбец данной таблицы должен иметь имя, отличное от других имен, для того, чтобы не возникло путаницы. Лучше всего, когда имена определяют содержимое поля. В этой книге мы будем использовать аббревиатуру для именования столбцов в простых таблицах, например: *cname* — для имени покупателя (customer name), *odate* — для даты поступления (order date). Предположим также, что таблица содержит единственный цифровой столбец, используемый как первичный ключ. В следующем разделе детально объясняются таблицы, используемые в качестве примера и их ключи.

Пример базы данных

Таблицы 1.1, 1.2, 1.3 образуют реляционную базу данных, которая достаточно мала для того, чтобы можно было понять ее смысл, но и достаточно сложна для того, чтобы иллюстрировать на ее примере важные понятия и практические выводы, связанные с применением SQL. Эти же таблицы приведены в приложении E. Поскольку в этой книге они будут использоваться для иллюстрации различных черт SQL, мы рекомендуем скопировать их и постоянно иметь перед глазами. Можно заметить, что первый столбец в каждой таблице содержит номера, не повторяющиеся от строки к строке в пределах таблицы. Как вы, наверное, догадались, это первичные ключи таблицы. Некоторые из этих номеров появляются также в столбцах других таблиц (в этом нет ничего предосудительного), что указывает на связь между строками, использующими конкретное значение первичного ключа, и той строкой, в которой это значение применяется непосредственно в первичном ключе.

Таблица 1.1. Salespeople (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. Customers (Покупатели)

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 1.3. Orders (Заказы)

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 Customers определяет, каким продавцом (salespeople) обслуживается конкретный покупатель (customer). Номер поля snum ус-

танавливает связь с таблицей Salespeople, которая дает информацию об этом продавце (salespeople). Очевидно, что продавец, который обслуживает данного покупателя, существует, т.е. значение поля snum в таблице Customers присутствует также и в таблице Salespeople. В этом случае мы говорим, что система находится в состоянии ссылочной целостности (referential integrity). Это понятие более подробно и формально объясняется в главе 19.

Сами по себе таблицы предназначены для описания реальных ситуаций в деловой жизни, когда можно использовать SQL для ведения дел, связанных с продавцами, их покупателями и заказами. Давайте зафиксируем состояние этих трех таблиц в какой-либо момент времени и уточним назначение каждого из полей таблицы.

Перед вами объяснение столбцов таблицы 1.1:

ПОЛЕ	СОДЕРЖИМОЕ
snum	Уникальный номер, приписанный каждому продавцу ("номер служащего")
sname	Имя продавца
city	Место расположения продавца
comm	Вознаграждение (комиссионные) продавца в форме с десятичной точкой

Таблица 1.2 содержит следующие столбцы:

ПОЛЕ	СОДЕРЖИМОЕ
cnum	Уникальный номер, присвоенный покупателю
cname	Имя покупателя
city	Место расположения покупателя
rating	Цифровой код, определяющий уровень предпочтения данного покупателя. Чем больше число, тем больше предпочтение
snum	Номер продавца, назначенного данному покупателю (из таблицы Salesperson)

И, наконец, столбцы таблицы 1.3:

ПОЛЕ	СОДЕРЖИМОЕ
onum	Уникальный номер, присвоенный данной покупке
amt	Количество
odate	Дата покупки
cnum	Номер покупателя, сделавшего покупку (из таблицы Customers)
snum	Номер продавца, обслужившего покупателя (из таблицы Salespeople)

Итоги

Итак, теперь вы знаете, чем является реляционная база данных. Вы также познакомились с некоторыми фундаментальными принципами структурирования таблиц, узнали, как работают строки и столбцы, как с помощью первичного ключа можно отличить одну строку таблицы от другой, и, наконец, как столбцы могут ссылаться на значения других столбцов. Вы узнали, что понятие "запись" является синонимом понятия "строка" и что понятие "поле" является синонимом понятия "столбец". Мы тоже будем использовать оба термина при обсуждении SQL в качестве синонимов.

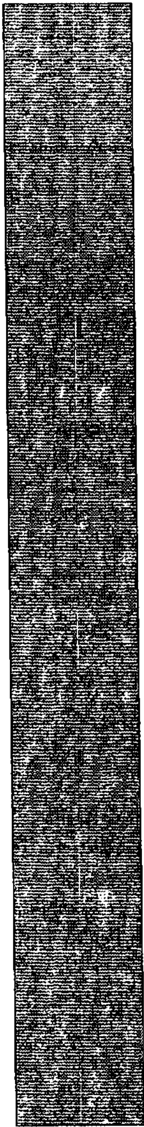
Вы уже знакомы с простыми таблицами. При всей своей краткости и простоте они вполне пригодны для демонстрации наиболее важных черт языка, в чем вы позже сами убедитесь. Иногда мы будем вводить другие таблицы или рассматривать другие данные в одной из этих таблиц для того, чтобы показать некоторые дополнительные возможности их применения.

Теперь мы готовы к непосредственному погружению в SQL. Следующая глава, к которой вам время от времени придется возвращаться, дает общее представление о языке и ориентирует вас в изложенном в книге материале.

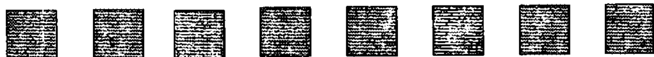
Работаем на SQL

1. Какое поле в таблице Customers является первичным ключом?
2. Дайте объяснение столбцу с номером 4 в таблице Customers?
3. Как иначе называются строка и столбец?
4. Почему нельзя попросить показать вам первые пять строк таблицы?

(Ответы см. в приложении А.)



2



Введение в SQL

В этой главе речь пойдет о структуре языка SQL, о некоторых общих вопросах, касающихся типов данных, которые могут содержаться в таблицах, а также о неясностях, существующих в SQL. Здесь же уточняется контекст специфической информации, которая будет дана в последующих главах. Вы можете войти в мир SQL, не упрощая его, и легко вернуться к нужному месту, если возникнут вопросы, благодаря тому, что этот материал расположен в начале книги.

Как работает SQL?

SQL — это язык, ориентированный специально на реляционные базы данных. Он позволяет исключить большую работу, выполняемую при использовании языка программирования общего назначения. Для создания реляционной базы данных, например на языке C, пришлось бы начать с определения объекта, называемого таблицей, который может иметь произвольное число строк, а затем создавать процедуры для ввода значений в таблицу и для поиска в ней данных. Для нахождения каких-то конкретных строк пришлось бы выполнить последовательность действий, например:

1. Посмотреть очередную строку таблицы.
2. Оттестировать ее и убедиться, что это та строка, которая Вас интересует.
3. Запомнить ее до тех пор, пока не будет просмотрена вся таблица.
4. Определить, есть ли в таблице еще строки.
5. Если в таблице еще есть строки (просмотрены не все строки), то вернуться к шагу 1.
6. Если в таблице больше нет строк (просмотрены все строки таблицы), вывести все значения, полученные на третьем этапе.

SQL освобождает от подобной работы. Команды SQL могут выполняться над целой группой таблиц, как над единственным объектом, а также могут оперировать любым количеством информации, которая извлекается или выводится из них как из единого целого.

Как осуществляется связь с ANSI-таблицей?

Стандарт SQL определен ANSI (American National Standards Institute — Американским национальным институтом стандартов). SQL не является изобретением ANSI, он — продукт исследований фирмы IBM. Однако другие компании тоже внесли свою лепту в развитие SQL; по крайней мере, компания Oracle превзошла IBM в создании популярного рыночного программного SQL-продукта.

После того, как на рынке появилось несколько конкурирующих SQL-продуктов, ANSI определила стандарт, которому все они должны удовлетворять. Однако вве-

дение стандарта post factum порождает ряд проблем. Результирующий стандарт SQL в некотором смысле ограничен: то, что определено ANSI, не всегда является наиболее полезным с точки зрения практического применения, поэтому создатели SQL-продуктов стараются разрабатывать их таким образом, чтобы они соответствовали стандарту ANSI, но не были бы слишком жестко ограничены его требованиями. Программные продукты, выполняющие обработку баз данных (системы управления базами данных — СУБД), обычно придают ANSI SQL дополнительные характерные черты и часто снимают большинство существенных практических ограничений, присущих стандарту. Поэтому наиболее общие отклонения от ANSI тоже следует проанализировать. Рассмотреть каждое отдельное исключение и отклонение от стандарта невозможно, однако полезные идеи, как правило, копируются и применяются одинаково в различных программных продуктах, даже если они не специфицированы ANSI. ANSI — это своего рода минимальный стандарт; можно делать гораздо больше, чем в нем определено, но, выполняя стандартную задачу, нужно обеспечить предусмотренные данным стандартом результаты.

Интерактивная версия встроенного SQL

Существуют два SQL: интерактивный и встроенный. В основном эти две формы SQL работают одинаково, но используются по-разному.

Интерактивный SQL применяется для выполнения действий непосредственно в базе данных с целью получить результат, который используется человеком. При применении этой формы SQL вводится команда, она выполняется, после чего можно немедленно увидеть выходные данные (если таковые есть).

Встроенный SQL состоит из команд SQL, включенных в программы, которые в большинстве случаев написаны на каком-то другом языке программирования (например, Cobol или Pascal). Такое включение может сделать программу более мощной и эффективной. Однако, несовместимость этих языков программирования со структурой SQL и присущим ему стилем управления данными требует внесения ряда расширений в интерактивный SQL. Выходные данные команд SQL во встроенном SQL "заносятся" в переменные или параметры, используемые программой, в которую включены предложения SQL.

В этой книге представлена интерактивная форма SQL, что позволит обсуждать команды и их действие, не обращая внимания на то, как они взаимодействуют с другими языками. Именно интерактивный SQL наиболее полезен для непрограммистов. Все, что характерно для интерактивного SQL, справедливо и для его встроенной формы. Изменения, которые следует выполнить в связи со встроенной формой, рассматриваются в последней главе этой книги.

Подразделы SQL

Как в интерактивном, так и во встроенном SQL имеется множество секций или подразделов. В процессе освоения SQL придется придерживаться данной терминологии, однако неудачным является то, что эти термины не используются всегда и во всех реализациях SQL. Им придается особое значение в ANSI, и они полезны на концептуальном уровне, но во многих SQL-продуктах они практически не выделены, и поэтому стали функциональными категориями SQL-команд.

Язык определения данных (Data Definition Language, DDL; в ANSI он называется также языком определения схемы (Schema Definition Language)) состоит из тех команд, которые создают объекты (таблицы, индексы, представления) в базе данных. Язык манипулирования данными (Data Manipulation Language, DML) — это множество команд, определяющих, какие данные представлены в таблицах в любой момент времени. Язык управления данными (Data Control Language, DCL) состоит из предложений, определяющих, может ли пользователь выполнить отдельное действие. Согласно ANSI, DCL является частью DDL. Важно не путать эти названия. Речь идет не о различных языках как таковых, а о разделах команд SQL, сгруппированных в соответствии с их функциональным назначением.

Различные типы данных

Не все типы значений, содержащиеся в полях таблицы, логически одинаковы. Наиболее очевидны различия между числами и текстом. Невозможно расположить числа в алфавитном порядке или извлечь одно имя из другого. Поскольку системы реляционных баз данных основаны на связях между частями информации, различные типы данных должны явно отличаться друг от друга, чтобы можно было применить подходящие способы их обработки и сравнения.

В SQL каждому полю приписывается "тип данных" (data type), который определяет, какого рода значения могут содержаться в поле. Все значения для данного поля должны быть одного типа. В таблице Customers, например, поля cname и city являются строками текста, тогда как поля rating, snum, cnum — числовые. Именно по этой причине невозможно занести значения "Highest" или "None" в поле rating, имеющее числовой тип. Это удачное ограничение, поскольку оно накладывает некоторую структуру на конкретные данные. Операцию сравнения, которая выполняется для одних строк и не выполняется для других, невозможно произвести, если значения поля имеют смешанный тип данных.

Определение этих типов данных является той областью, в которой многие коммерческие СУБД и официальный стандарт SQL имеют существенные различия. Стандарт ANSI SQL распознает только текстовый и числовой типы, тогда как многие коммерческие СУБД используют и другие специальные типы данных. Заметим, что типы DATE (дата) и TIME (время) почти de-facto являются стандартными (хотя конкретные их форматы отличаются). Некоторые СУБД поддерживают такие типы данных как MONEY (деньги) и BINARY (двоичный). (BINARY — это специальное числовое

представление, используемое компьютером. Вся информация в компьютере представлена двоичными числами, затем она преобразуется в другие системы — так ее легче использовать и понимать.)

ANSI определяет несколько различных типов числовых значений. Типы данных ANSI полностью перечислены в приложении В. Сложность числовых типов ANSI объясняется, по крайне мере частично, попыткой поддержать совместимость вложенного SQL с множеством других языков.

Два типа данных ANSI, INTEGER и DECIMAL (для которых можно использовать аббревиатуру INT и DEC соответственно), адекватны и теоретическим целям, и множеству практических приложений в деловой жизни. INTEGER отличается от DECIMAL тем, что запрещает использовать цифры справа от десятичной точки, а также саму десятичную точку.

Типом данных для текста является CHAR (CHARACTER), который относится к строке текста. Поле типа CHAR имеет фиксированную длину, равную максимальному числу букв, которые можно ввести в это поле. Большинство реализаций SQL имеет нестандартный тип, названный VARCHAR, — это текстовая строка любой длины вплоть до максимума, определяемого конкретной реализацией SQL. Значения CHAR и VARCHAR заключаются в одиночные кавычки, как, например, 'текст'. Различие между ними состоит в том, что для типа CHAR отводится участок памяти, достаточный для хранения строки максимальной длины, а для VARCHAR память выделяется по мере необходимости.

Символьные типы состоят из всех символов, которые можно ввести с клавиатуры, в том числе и цифр. Однако, число 1 не есть то же самое, что символ '1'. Символ '1' это совсем другая часть печатного текста, которая не распознается компьютером как числовое значение 1. 1 + 1 = 2, но '1' + '1' не равно '2'. Значения типа CHARACTER хранятся в компьютере как двоичные значения, но для пользователя представляются в виде печатного текста. Преобразование выполняется в соответствии с форматом, определяемым той системой, которой вы пользуетесь. Это может быть формат одного из двух стандартных типов (возможно, с расширениями), которые применяются в компьютерных системах : ASCII (используется во всех персональных и большинстве малых компьютеров) и EBCDIC (используется для больших компьютеров). Определенные операции, такие как упорядочение значений поля по алфавиту, зависят от формата. Значения этих двух форматов будут рассмотрены в главе 4.

Тип DATE будет применяться в соответствии с требованиями рынка, а не ANSI. В реализациях SQL, не распознающих тип DATE, можно объявить дату символьным или числовым полем, но это затруднит выполнение множества операций. Следует ознакомиться с документацией по программному обеспечению SQL-системы, чтобы точно определить, какие типы данных она поддерживает.

Кто такой "пользователь"?

SQL устанавливается, как правило, в компьютерных системах, имеющих не одного, а многих пользователей, которых нужно уметь различать (у семейного PC может быть любое число пользователей, но обычно не существует способа отличить их друг от друга). В типичной ситуации каждый пользователь такой системы имеет

код авторизации, который идентифицирует его или ее (в терминологии имеются различия). В начале сеанса связи с компьютером пользователь *регистрируется* в системе, сообщая компьютеру, какой именно пользователь, идентифицированный кодом авторизации ID, находится на связи. Что касается компьютера, то любое число пользователей, имеющих один и тот же ID, является для него одним пользователем; напротив, один человек может восприниматься как множество пользователей, если он (обычно в различные моменты времени) использует различные коды авторизации ID.

SQL придерживается этого правила. В большинстве SQL-систем действия приписываются определенному ID, который обычно соответствует определенному пользователю. Таблица (или другой объект) принадлежит тому пользователю, который имеет на нее (или на этот объект) полномочия. Пользователь может иметь или не иметь привилегию работы с объектами, которые ему не принадлежат. В главе 22 специально обсуждаются привилегии, пока же предположим, что любой пользователь имеет привилегию выполнять любые необходимые ему действия.

Специальное значение USER может использоваться как аргумент в команде. Он обозначает авторизационный ID пользователя, дающего команду.

Соглашения и терминология

Ключевые слова это слова, имеющие специальное значение в SQL. Они являются инструкциями, а не текстом или именами объектов. Ключевые слова будут выделяться заглавными буквами. Следует быть внимательнее и не путать ключевые слова с терминами. SQL имеет определенный набор специальных терминов, которые применяются для его описания. Среди них есть такие слова как запрос, предложение, предикат. Они важны для описания и понимания языка, но для самого SQL ничего не значат.

Команды (*commands*) или *сообщения* (*statements*) — это инструкции, которые даются базе данных SQL. Команды состоят из одной или более логически различных частей, называемых *предложениями* (*фразами*, *clauses*). Предложения начинаются с ключевого слова, по которому они обычно и называются, и состоят из ключевых слов и аргументов. Примерами предложений являются: "FROM Salespeople" и "WHERE city = 'London'". *Аргументы* заканчивают предложение или модифицируют его смысл. В приведенных примерах "Salespeople" является аргументом, а FROM - ключевым словом предложения FROM. Также "city = 'London'" является аргументом предложения WHERE. Объекты — это структуры в базе данных, которые имеют имена и хранятся в памяти. Они включают базовые таблицы, представления (то есть два вида таблиц) и индексы.

Объяснение того, как формулируются команды, будет осуществляться в основном на примерах. Однако существует более формальный метод описания команд с использованием стандартных соглашений, который иногда применяется в следующих главах. Упомянутые соглашения полезно знать в случае столкновения с ними в другой документации по SQL. Квадратные скобки ([ ]) выделяют те части, которые можно опустить, круглые скобки (...) показывают, что предшествующее им можно повторить

любое число раз. Слова, заключенные в угловые скобки (< >), — специальные термины, которые объясняются по мере того, как вводятся.

Итоги

Эта глава охватывает большое количество основной информации, дающей общее представление об SQL. Вы узнали, как он структурирован, как используется, как в нем выражаются данные, как и кем он определяется (и какие противоречия при этом возникают), а также некоторые соглашения и терминологию, используемые для описания.

В следующей главе подробно объясняются формирование и действие команд. Вы познакомитесь с командой, позволяющей извлекать информацию из таблиц и являющейся одной из наиболее часто применяемых в SQL. Вы сможете вывести сами определенную информацию из базы данных.

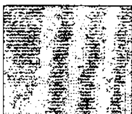
Работаем на SQL

1. Каковы основные различия между типами данных в SQL?
2. Есть ли в ANSI тип данных DATE?
3. Какой подраздел SQL используется для ввода значений в таблицы?
4. Что такое ключевое слово?

(Ответы даны в приложении А.)

3

Использование SQL для выборки данных из таблиц



Эта глава учит осуществлять поиск информации в таблицах, пропускать или переставлять столбцы, автоматически исключать избыточные данные при выводе результата. Наконец, задавать условие — критерий, который можно применять для определения строк таблицы, используемых для вывода результирующих данных. С этой особенностью SQL более подробно мы ознакомимся в последующих главах.

Формирование запроса

SQL символизирует структурированный язык запросов (Structured Query Language). Запросы являются наиболее часто используемым аспектом SQL. Есть категория пользователей SQL, которые используют язык только для формулировки запросов. Поэтому изучение SQL начинается с обсуждения запроса и того, как он выполняется в этом языке.

Что такое запрос? Это команда, которая формулируется для СУБД и требует предоставить определенную указанную информацию. Эта информация обычно выводится непосредственно на экран дисплея компьютера или используемый терминал, хотя в ряде случаев ее можно направить на принтер, сохранить в файле или использовать в качестве исходных данных для другой команды или процесса.

Как осуществляется связь запросов?

Запросы являются частью DML. Но так как они совершенно не изменяют информации в таблицах, а лишь показывают ее пользователю, предположим, что запросы являются самостоятельной категорией и определяют команды DML, воздействующие на содержимое базы данных, а не просто показывающие его.

Все запросы в SQL конструируются на базе одной команды. Структура этой команды проста, потому что ее можно расширять для того, чтобы выполнить очень сложные вычисления и обработку данных. Эта команда называется SELECT.

Команда SELECT

В простейшей форме команда SELECT дает инструкцию базе данных для поиска информации в таблице. Например, можно получить таблицу Salespeople, введя с клавиатуры следующее:

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

Выходные данные для этого запроса представлены на рис. 3.1.

Команда просто выводит все данные из таблицы. Большинство программ, как показано выше, также выводит заголовки столбцов. Некоторые программы допускают

```
SQL Execution Log
SELECT snum, sname, city, 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
Browse : ↑↓←→  PgDn PgUp —▶| |◀— Home
```

Рис. 3.1. Команда SELECT

тщательное форматирование выходных данных, но это лежит за пределами спецификаций стандарта. Далее приводится объяснение каждой части этой команды:

SELECT — Ключевое слово, которое сообщает базе данных, что команда является запросом. Все запросы начинаются с этого ключевого слова, за которым следует пробел.

snum, sname ... — Список столбцов таблицы, которые должны быть представлены в результате выполнения запроса. Столбцы, имена которых не представлены в списке, не включаются в состав выходных данных команды. Это, однако, не приводит к удалению из таблиц таких столбцов или содержащейся в них информации, потому что запрос не воздействует на информацию, представленную в таблицах: он только извлекает данные.

FROM Salespeople — FROM, так же как и SELECT, является ключевым словом, которое должно быть представлено в каждом запросе. За ним следует пробел, а затем — имя таблицы, которая используется как источник информации для запроса. В приведенном примере это таблица Salespeople.

Символ "точка с запятой" (;) используется во всех интерактивных командах SQL для сообщения базе данных, что команда сформулирована и готова к выполнению. В некоторых системах

этот символ заменен на символ "слэш обратный" ("\") в строке, которая непосредственно следует за концом команды.

Стоит заметить, что запрос по своей природе не обязательно упорядочивает выходные данные каким-либо определенным образом. Одна и та же команда, выполненная над одними и теми же данными в различные моменты времени, в результате выдает данные, упорядоченные по-разному. Обычно строки выдаются в том порядке, в котором они представлены в таблице, но этот порядок может быть совершенно произвольным. Необязательно, что данные в результате выполнения запроса будут представлены в том порядке, в котором они вводятся или хранятся. Можно упорядочить выходные данные непосредственно с помощью SQL-команд, указав специальное предложение. Позже будет объяснено, как это сделать. Сейчас же просто констатируем факт отсутствия какого-либо порядка в представлении выходных данных.

Использование клавиши возврата каретки (клавиши Enter) является произвольным. Можно ввести запрос в одной строке следующим образом:

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

Поскольку в SQL точка с запятой применяется для того, чтобы пометить конец команды, большинство SQL-программ использует клавишу "Возврат каретки" (выполняется нажатием клавиши Return или Enter) как пробел.

Выбор чего-либо простейшим способом

Если необходимо увидеть каждую колонку таблицы, существует упрощенный вариант сделать это. Можно использовать символ "*" ("звездочка"), который заменяет полный список столбцов.

```
SELECT *
   FROM Salespeople;
```

Результат выполнения этой команды тот же, что и для рассмотренной ранее.

SELECT в общем виде

Обобщая предыдущие рассуждения, следует отметить, что команда SELECT начинается с ключевого слова SELECT, за которым следует пробел. После него следует список разделенных запятыми имен столбцов, которые необходимо увидеть. Если нужно увидеть все столбцы таблицы, то можно заменить список имен столбцов символом (*) (звездочка). За звездочкой следует ключевое слово FROM, за ним — пробел и имя таблицы, к которой направляется запрос. Символ точка с запятой (;) нужно использовать для того, чтобы закончить запрос и показать, что команда готова для выполнения.

Просмотр только определенных столбцов таблицы

Мощность команды SELECT заключается в ее свойстве извлекать из таблицы лишь определенную информацию. Надо отметить возможность просмотра только указанных столбцов таблицы. Для этого достаточно пропустить столбцы, которые нет необходимости просматривать, в части команды SELECT. Например, по запросу

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

получаются выходные данные, представленные на рис. 3.2.

Существуют таблицы, включающие большое количество столбцов, содержащих данные, не все из которых требуются в определенный момент. Следовательно, возможность выбора и указания интересующих колонок весьма полезна.

```
SQL Execution Log
SELECT sname, comm
FROM Salespeople;
```

sname	comm
Peel	0.12
Serres	0.13
Motika	0.11
Rifkin	0.15
Axelrod	0.10

```
Browse  ↑↓←→  PgDn PgUp  —▶|  |◀—  Home
```

Рис. 3.2. Выбор определенных столбцов

Перестановка столбцов

Колонки таблицы упорядочены по определению, но это не значит, что их нужно извлекать в том же порядке. Звездочка (*) извлечет столбцы в соответствии с их порядком, но если указать столбцы раздельно, они выстраиваются их в любом желаемом порядке. В таблице Orders зададим такой порядок столбцов: сначала разместим столбец "дата заказа" (odate), за ним — столбец "номер продавца" (snum), затем - "номер заказа" (onum) и "количество" (amt):

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

Выходные данные, полученные по этому запросу, представлены на рис. 3.3.

Очевидно, что структура информации в таблицах является просто основой для ее реструктуризации средствами SQL.

Устранение избыточных данных

DISTINCT — аргумент, дающий возможность исключить дублирующиеся значе-

```
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

Browse : ↑↓←→ PgDn PgUp ▶| |◀ Home

Рис. 3.3. Переупорядоченные столбцы

ния из результата выполнения предложения SELECT. Предположим, необходимо узнать, какие продавцы имеют в настоящее время заказы в таблице Orders. Не имеет значения количество заказов каждого из продавцов, нужен лишь список номеров продавцов (snum). Необходимо ввести:

```
SELECT snum
    FROM Orders;
```

чтобы получить результат, представленный на рис. 3.4.

Для того чтобы получить список без повторений, который легче прочесть, нужно ввести следующую команду:

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

Выходные данные для этого запроса представлены на рис. 3.5.

DISTINCT отслеживает, какие значения появились в списке выходных данных, и исключает из него дублирующиеся значения. Это полезный способ исключить избыточные данные. Если таковых нет, не следует использовать DISTINCT, поскольку он может скрыть проблемы. Предположим, все имена покупателей различны. Если кто-то

Рис. 3.4. SELECT с повторениями

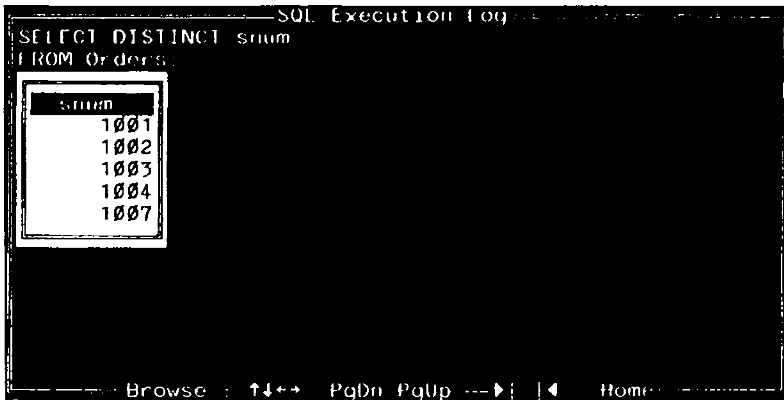


Рис. 3.5. SELECT без повторений

введет второго покупателя с фамилией Clemens в таблицу Customers при использовании SELECT DISTINCT cname, можно не заметить, что имеются дублирующиеся данные. Будут получены ошибочные сведения о Clemens, поскольку в этом случае нет информации об избыточности данных.

Параметры DISTINCT. DISTINCT можно задать только один раз для данного предложения SELECT. Если SELECT извлекает множество полей, то он исключает строки, в которых все выбранные поля идентичны. Строки, в которых некоторые значения одинаковы, а другие — различны, включаются в результат. DISTINCT, фактически, действует на всю выходную строку, а не на отдельное поле (исключение составляет его применение внутри агрегатных функций, см. главу 6), исключая возможность их повторения.

DISTINCT в сравнении с ALL. Альтернативой DISTINCT является ALL. Это ключевое слово имеет противоположное действие: повторяющиеся строки включаются в состав выходных данных. Поскольку часто бывает так, что не заданы ни DISTINCT, ни ALL, предполагается ALL; это ключевое слово имеет преимущество перед функциональным аргументом.

Определение выборки — предложение WHERE

Таблицы бывают достаточно большими с тенденцией к увеличению по мере добавления строк. В данный момент времени интересны только некоторые строки таблицы. SQL дает возможность задать критерий определения строк, которые следует включить в состав выходных данных. Предложение WHERE команды SELECT позволяет определить *предикат*, условие, которое может быть либо истинным, либо ложным для каждой строки таблицы. Команда извлекает только те строки из таблицы, для которых предикат имеет значение "истина". Предположим, необходимо узнать имена всех продавцов в Лондоне (London). В этом случае можно ввести следующую команду:

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

При наличии предложения WHERE программа обработки базы данных просматривает таблицу строка за строкой и для каждой строки проверяет, истинен ли на ней предикат. Следовательно, для записи о продавце Peel программа просмотрит текущее значение в столбце city (город), определит, что оно равно 'London', и включит эту строку в состав выходных данных. Запись о продавце Serres не включается и т.д. Выходные данные для приведенного выше запроса представлены на рис. 3.6.

Столбец city включен в результат не потому, что он указан в предложении WHERE, а потому, что имя этого столбца указано в предложении SELECT. Совершенно необязательно, чтобы столбец, используемый в предложении WHERE, был представлен в числе тех столбцов, которые необходимо видеть среди выходных данных.

Можно рассмотреть пример с использованием числового поля в предложении WHERE. Поле rating таблицы Customers предназначено для того, чтобы разделить покупателей на группы по некоторому критерию в соответствии с этим номером. Это

```
SQL Execution Log
SELECT sname, city
FROM Salespeople
WHERE city = 'London';

sname            city
Peel             London
Motika           London

Browse : ↑↓←→  PgDn PgUp  —►|  |◄—  Home
```

Рис. 3.6. SELECT с предложением WHERE

своего рода оценка кредита или оценка, основанная на значении предыдущих покупок. Такие цифровые коды могут быть полезны в реляционных базах данных как способ обобщения сложной информации. Можно выбрать всех покупателей (Customers) с рейтингом (rating) 100 следующим образом:

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

Здесь не используются одиночные кавычки, поскольку поле rating является числовым. Результат запроса представлен на рисунке 3.7.

К предложению WHERE относятся все комментарии, сделанные в этой главе ранее. Т.е. можно использовать номера столбцов, исключать повторяющиеся строки или переставлять столбцы в командах SELECT, использующих 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	1004

```
Browse : ↑↓←→ PgDn PgUp ▶| |◀ Home
```

Рис. 3.7. SELECT с числовым полем в предикате

Итоги

Мы выяснили, что существует несколько способов получения представленной в таблице информации в том виде, который вас интересует. Например, можно переставлять или исключать столбцы таблицы, а также сохранять или исключать повторяющиеся строки.

И, наконец, наиболее важно то, что можно задать предикат, который определяет, включается ли некоторая строка из множества строк в состав результирующих данных. Предикаты являются очень полезным инструментом, открывающим широкие возможности управления строками, которые должны войти в результат запроса. Именно это свойство предикатов и делает запросы SQL столь мощными. В следующих нескольких главах мы рассмотрим характерные черты и возможности предикатов. Глава 4 посвящена операторам сравнения, отличным от равенства, которые можно использовать в условиях предиката, и способам комбинирования множества условий в единственный предикат.

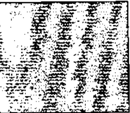
Работаем на SQL

1. Запишите команду SELECT, которая выводит порядковый номер (order number), количество (amount) и дату (date) для всех строк таблицы Order.
2. Запишите запрос, который выдает все строки таблицы Customers, где продавец имеет номер 1001.
3. Запишите запрос, который выдает строки таблицы salesperson в таком порядке: city, sname, snum, comm.
4. Запишите команду SELECT, которая выдает rating и следом за ним name каждого покупателя (customer), проживающего в San Jose.
5. Запишите запрос, позволяющий получить значения столбца snum для всех продавцов (salespeople), номера (orders) которых находятся в настоящее время в таблице Orders, причем повторения требуется исключить.

(Ответы представлены в приложении А.)

4

Использование реляционных и булевых операторов для создания более сложных предикатов



Из главы 3 выяснилось, что предикаты могут приписывать предложениям с равенством значения "истина" или "ложь", а также оценивать операторы сравнения отличные от равенства. В этой главе рассмотрены и другие операторы сравнения, применяемые в SQL, и показано, как можно использовать булевы операторы для изменения и комбинирования значений предиката. В булевом выражении единственный предикат может содержать любое количество условий, что позволяет получить очень мощные предикаты. Здесь также объясняется применение круглых скобок для структурирования сложных предикатов.

Реляционные операторы

Реляционный оператор — это математический символ, который задает определенный тип сравнения между двумя значениями. Уже известно как применяются равенства, такие как 2 + 3 = 5 или city = 'London'. Однако существуют и другие операторы сравнения. Предположим, необходимо вычислить продавцов (Salespeople), комиссионные (commissions) которых превышают заданное значение. В этом случае следует воспользоваться сравнением типа "больше или равно". SQL распознает следующие операторы сравнения:

=	Равно
>	Больше, чем
<	Меньше, чем
>=	Больше или равно
<=	Меньше или равно
<>	Неравно

Эти операторы имеют стандартное значение для числовых величин. Их определение для символьных значений зависит от используемого формата представления (ASCII или EBCDIC). SQL сравнивает символьные значения в терминах соответствующих чисел, определенных в формате преобразования. Символьные значения, представляющие числа, например, '1', необязательно равны тому числу, которое они представляют.

Операторы сравнения можно применять для того, чтобы представить алфавитный порядок; например, 'a' < 'n' означает, что 'a' предшествует 'n' в алфавитном порядке, но эта процедура ограничена параметрами формата преобразования. Как в ASCII, так и в EBCDIC, сохранен алфавитный порядок предшествования символов, представленных в одном и том же регистре. В ASCII все заглавные символы меньше, чем все строчные, значит 'Z' < 'a', а все цифры меньше, чем все символы, значит '1' < 'Z'. В EBCDIC все наоборот. Для простоты рассмотрения, предположим, что используется формат ASCII. Если точно неизвестно, с каким форматом идет работа или как работает формат, то следует обратиться к документации.

Значения, которые здесь сравниваются, называются *скалярными значениями*. Скалярные значения получаются из скалярных выражений: 1 + 2 является скалярным выражением, которое дает скалярное значение 3. Скалярные значения могут быть символами или числами, хотя только числа используются с арифметическими операторами, такими как + или *. Предикаты обычно сравнивают скалярные значения, используя операторы сравнения или специальные SQL-операторы, для того, чтобы проверить является ли результат сравнения истинным. Некоторые SQL-операторы рассмотрены в главе 5.

Предположим, необходимо увидеть всех покупателей (Customers) с рейтингом (rating) более 200. Поскольку 200 — это скалярное значение, как и все значения столбца rating, для их сравнения можно использовать оператор отношения:

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

Выходные данные для этого запроса представлены на рис. 4.1.

При необходимости увидеть всех покупателей, рейтинг (rating) которых больше или равен 200, следовало бы использовать предикат:

```
rating >= 200
```

```
SQL Execution Log
SELECT *
FROM Customers
WHERE rating > 200;
```

cnum	cname	city	rating	snum
2004	Grass	Berlin	300	1002
2008	Cisneros	San Jose	300	1007

```
Browse : ↑↓←→  PgDn PgUp  —▶|  |◀—  Home
```

Рис. 4.1. Использование "больше, чем" (>)

Булевы операторы

SQL распознает основные булевы операторы. Булевы выражения — это те выражения, относительно которых, подобно предикатам, можно сказать, истинны они или ложны. Булевы операторы связывают одно или несколько значений "истина/ложь" и в результате получают единственное значение "истина/ложь". Стандартные булевы операторы, распознаваемые SQL, — это AND, OR, NOT. Существуют и другие, более сложные булевы операторы (как, например, "исключающее ИЛИ"), но их можно построить с помощью трех простых. Булева логика "истина/ложь" представляет собой полный базис для работы цифрового компьютера. Поэтому фактически весь SQL (или какой-либо другой язык программирования) можно свести к булевой логике. Далее перечислены булевы операторы и основные принципы их действия:

- AND берет два булевых выражения (в виде A AND B) в качестве аргументов и дает в результате истину, если они оба истинны.
- OR два булевых выражения (в виде A OR B) в качестве аргументов и оценивает результат как истину, если хотя бы один из них истинен.
- NOT берет единственное булево выражение (в виде NOT A) в качестве аргумента и изменяет его значение с истинного на ложное или с ложного на истинное.

Используя предикаты с булевыми операторами, можно значительно увеличить их избирательную мощность. Предположим, необходимо увидеть всех покупателей (customers) из San Jose, чей рейтинг (rating) превышает 200:

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

Выходные данные для этого запроса представлены на рис. 4.2. Существует только один покупатель, удовлетворяющий этому условию.

При использовании OR , будут получены сведения обо всех тех покупателях (customers), которые либо проживают в San Jose, либо имеют рейтинг (rating), превышающий 200.

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

Результат выполнения этого запроса представлен на рис. 4.3.

```
SQL Execution Log
SELECT *
FROM Customers
WHERE city = 'San Jose'
AND rating > 200;

 cnum   cname       city        rating   snum
 2008   Cisneros    San Jose       300   1007

Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home
```

Рис. 4.2. SELECT с использованием AND

```
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   Cisneros    San Jose       300   1007

Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home
```

Рис. 4.3. SELECT с использованием OR

NOT дает возможность получить отрицание (противоположное значение) булева выражения. Вот пример запроса с использованием NOT:

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

```
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	Cisneros	San Jose	300	1007
2007	Pereira	Rome	100	1004

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 4.4. SELECT с использованием NOT

Результат выполнения этого запроса представлен на рис. 4.4.

Все записи, за исключением Grass, были выбраны. Grass не находится в San Jose и его рейтинг превышает 200, таким образом он не удовлетворяет обоим условиям. Каждая из других строк удовлетворяет либо первому, либо второму условию (либо каждому из них). Заметим, что оператор NOT должен предшествовать булеву выражению, значение которого он должен изменить, но не может располагаться непосредственно перед оператором сравнения, как это можно сделать во фразе на английском языке. Таким образом *некорректно* вводить

```
rating NOT > 200
```

в качестве предиката, несмотря на то, что эту фразу можно легко сформулировать по-английски. Отсюда следует ряд проблем. Например, как SQL оценит следующее?

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

Применяется ли NOT к выражению city = 'San Jose' или к двум выражениям: тому, что указано, и выражению rating > 200? В соответствии с приведенной записью правильным является первый вариант. SQL применяет NOT только к тому булеву выражению, которое непосредственно следует за ним. Можно получить другой результат по следующей команде:

```
SELECT *
  FROM Customers
  WHERE NOT (city = 'San Jose'
```

```
OR rating > 200);
```

SQL понимает круглые скобки следующим образом: все то, что расположено внутри круглых скобок, вычисляется прежде всего и рассматривается как единственное выражение по отношению к тому, что расположено за пределами круглых скобок (это соответствует стандартной интерпретации в математике). Другими словами, SQL извлекает каждую строку и определяет, выполняется ли для нее условие city = 'San Jose' или rating > 200. Если одно из этих выражений истинно, то булево выражение, расположенное в круглых скобках, тоже истинно. Однако, если булево выражение в круглых скобках истинно, предикат в целом ложен, поскольку NOT превращает истину в ложь и наоборот. Результат выполнения этого запроса представлен на рис. 4.5.

Вот преднамеренно усложненный пример. Проследим его логику (результат выполнения запроса представлен на рис. 4.6):

```
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

Browse : ↑↓←→ PgDn PgUp ▸| |◂ Home

Рис. 4.5. SELECT с использованием NOT и круглых скобок

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

Комбинации булевых операторов в сложных выражениях не столь просты, как каждый из в отдельности. Способ оценки сложного булева выражения следующий: оценить булево(ы) выражение(ия), имеющее(ие) наибольшую глубину вхождения в круглые скобки, скомбинировать результаты в одно булево выражение, а затем связать его значение со значениями выражений, имеющих меньшую глубину вхождения в круглые скобки.

Дадим детальное объяснение оценки рассмотренного выше примера. Наибольшую глубину вхождения в булево выражение имеет предикат: odate = 10/03/1990 and snum > 1002, со связкой AND, образующий булево выражение, которое оценивается как истинное для всех тех строк, которые удовлетворяют каждому из этих условий. Это составное булево выражение (которое мы назовем булево выражение номер 1 или, для краткости, B1) соединено с amt > 2000.00 (выражение B2) с помощью OR и образует третье выражение (B3), которое является истинным для данной строки в том случае, если либо B1 либо B2 истинны для этой строки. B3 полностью содержится в круглых скобках, которым предшествует NOT, и образует заключительное булево выражение (B4), которое является условием предиката. Следовательно, B4 — предикат запроса — истинен, если B3 ложен и наоборот. B3 ложен, если ложен каждый из B1 и B2. B1 ложен для строк, в которых либо order date не совпадает с заданным значением 10/03/1990, либо значение snum не превышает 1002. B2 ложен для всех строк, в которых значение поля amount не превосходит 2000.00. Любая строка с суммой, превышающей 2000.00, делает B2 истинным, отсюда B3 тоже истинно, а B4 — ложно. Следовательно, все такие строки исключаются из числа выходных данных. Остающиеся строки от 3 октября 1990 года с snum, превышающим 1002 (такой, например, является строка с onum 3001 за октябрь, 3, 1990 с snum 1007), делают B1 истинным, следовательно, и B3 истинно, а значит предикат ложен. Эти записи также исключаются из рассмотрения. Оставшиеся строки входят в состав выходных данных (см. рис. 4.6).

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 4.6. Сложный запрос

Итоги

В этой главе более полно представлены сведения из области предикатов. Показано, как можно найти значения, которые связаны с данным значением любым количеством способов, заданных с помощью различных реляционных операторов, как применяются булевы операторы AND и OR для комбинации сложных условий, каждое из которых может рассматриваться как единственный предикат. Булев оператор NOT изменяет на противоположное значение условия или группы условий. Все булевы выражения и операторы отношения управляются с помощью круглых скобок, которые определяют порядок выполнения операции. Эти операции могут иметь любой уровень сложности. Было рассмотрено, как можно разложить записанное сложное выражение на составные части, каждая из которых является простой.

В главе 5 будут представлены особые операторы языка SQL.

Работаем на SQL

1. Запишите запрос, который покажет все заявки, превышающие $1,000.
2. Запишите запрос, который покажет имена (names) и названия городов (cities) для всех продавцов в London с комиссионными (commission), превышающими .10.
3. Запишите запрос для таблицы Customers, включающий в выходные данные всех покупателей, для которых rating <= 100, в том случае, если они расположены не в Rome.
4. Каков будет результат выполнения следующего запроса?

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

5. Каков будет результат выполнения следующего запроса?

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

6. Как упростить запись следующего запроса?

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

(Ответы см. в приложении А.)

5

Использование специальных операторов в "условиях"

Кроме булевых операторов и операторов сравнения, рассмотренных в главе 4, SQL использует специальные операторы IN, BETWEEN, LIKE и IS NULL. Вы научитесь применять их, подобно операторам сравнения, для получения более выразительных и мощных предикатов. Обсуждение IS NULL касается значений пропускаемых данных и NULL-значений, фиксирующих отсутствие данных.

Оператор IN

IN полностью определяет множество, которому данное значение может принадлежать или не принадлежать. Если нужно найти всех продавцов, расположенных либо в 'Barcelona', либо в 'London', основываясь только на том, что известно к настоящему моменту, необходимо написать следующий запрос (выходные данные для него представлены на рис. 5.1):

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

Однако существует более простой способ получить ту же самую информацию:

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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 5.1. Поиск продавцов (Salespeople) в городах Barselona или London

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 5.2. SELECT с использованием IN

Выходные данные этого запроса представлены на рис. 5.2.

Как видно из примера, IN определяет множество, элементы которого точно перечисляются в круглых скобках и разделяются запятыми. Если в поле, имя которого указано слева от IN, есть одно из перечисленных в списке значений (требуется точное совпадение), то предикат считается истинным. Если элементы множества имеют числовой, а не символьный тип, то одиночные кавычки непосредственно слева и справа от значения необходимо опустить. Можно найти всех покупателей, обслуживаемых продавцами 1001, 1007, 1004. Выходные данные для следующего запроса представлены на рис. 5.3:

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

Оператор BETWEEN

Оператор BETWEEN сходен с IN. Вместо перечисления элементов множества, как это делается в IN, BETWEEN задает границы, в которые должно попадать значение, чтобы предикат был истинным. Используется ключевое слово BETWEEN, за которым следуют начальное значение, ключевое слово AND и конечное значение. Также как и IN, BETWEEN чувствителен к порядку: первое значение в предложении должно быть первым в соответствии с алфавитным или числовым порядком. (В отличие от английского языка в SQL не говорят: *значение* расположено между ("is BETWEEN") *значени-*

```
SQL Execution Log
SELECT *
FROM Customers
WHERE snum IN (1001, 1007, 1004);
```

cnum	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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 5.3. SELECT с использованием IN с числовыми значениями

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 5.4. SELECT с использованием BETWEEN

ем и *значением*, но просто *значение между* ("BETWEEN") *значением* и *значением*. Это замечание справедливо и для оператора LIKE.) Следующий запрос позволит извлечь из таблицы Salespeople всех продавцов (salespeople), комиссионные которых имеют величину в диапазоне .10 и .12 (выходные данные представлены на рис. 5.4):

```
SELECT *
   FROM Salespeople
```

```
WHERE comm BETWEEN .10 AND .12;
```

Оператор BETWEEN является включающим, т.е. граничные значения (в данном примере это .10 и .12) делают предикат истинным. SQL непосредственно не поддерживает исключающий BETWEEN. Необходимо сформулировать граничные значения так, чтобы включающая интерпретация была справедлива, либо сделать примерно следующую запись:

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

Выходные данные для этого запроса представлены на рис. 5.5.

Пусть эта запись и неуклюжа, но она показывает, как новые операторы можно комбинировать с булевыми операторами для получения более сложных предикатов. Значит, IN и BETWEEN используются, как и операторы сравнения, для сопоставления значений, одно из которых является множеством (для IN) или диапазоном (для BETWEEN).

Аналогично всем операторам сравнения, BETWEEN действует на символьных полях, представленных в двоичном (ASCII) эквиваленте, т.е. для выборки можно воспользоваться алфавитным порядком. Следующий запрос выбирает всех покупателей имена которых попадают в заданный алфавитный диапазон:

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

Выходные данные для этого запроса представлены на рис. 5.6.

```
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
Browse : ↑↓←→  PgDn PgUp  Home
```

Рис. 5.5. Выполнение исключающего BETWEEN

```
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

Browse : ↑↓←→  PgDn PgUp --▶| |◀-- Home
```

Рис. 5.6. Использование BETWEEN с выборкой в алфавитном порядке

Grass и Giovanni опущены несмотря на то, что BETWEEN является включаю- щим, так как он сравнивает строки неравной длины. Строка 'G' короче строки 'Giovanni', поэтому BETWEEN дополняет 'G' пробелами. Пробелы предшествуют символам латинского алфавита (в большинстве реализаций), поэтому Giovanni оказался невыбранным. Аналогично и для Grass. Помните об этом при использовании BETWEEN с алфавитными диапазонами. Для включения в результат выполнения запроса сведений о покупателях, фамилии которых начинаются на 'G', нужно указать следующую букву алфавита ('H') или приписать символ 'z' (несколько символов 'z', если это необходимо) после второго граничного значения.

Оператор LIKE

LIKE применим только к полям типа CHAR или VARCHAR, поскольку он используется для поиска подстрок. Другими словами, он осуществляет просмотр строки для выяснения: входит ли заданная подстрока в указанное поле. С этой же целью используются *шаблоны* — специальные символы, которые могут обозначать все, что угодно. Существует два типа шаблонов, используемых с LIKE:

- Символ "подчеркивание" (_) заменяет один любой символ. Например, образцу 'b_t' соответствуют 'bat' или 'bit', но не соответствует 'brat'.

- Символ "процент" (%) заменяет последовательность символов произвольной длины, в том числе и нулевой. Например, образцу '%p%t' соответствуют 'put', 'posit', 'opt', но не 'spite'.

Можно найти покупателей, фамилии которых начинаются на 'G' (выходные данные, соответствующие этому запросу, представлены на рис. 5.7):

```
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

Browse : ↑↓←→ PgDn PgUp ——►| |◄—— Home
```

Рис. 5.7. SELECT с использованием LIKE с символом %

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

LIKE может оказаться полезным при осуществлении поиска имени или другого значения, полное написание которого неизвестно. Предположим, не совсем понятно, как правильно записывается фамилия одного из продавцов (salespeople): Peal или Peel. Можно использовать ту часть, которая известна, и символы шаблона для нахождения всех возможных вариантов (выходные данные для этого запроса представлены на рис. 5.8):

```
SELECT *
    FROM Salespeople
    WHERE sname LIKE 'P_ _l%';
```

Каждый символ подчеркивания в шаблоне представляет единственный символ, поэтому, например, имя Prettel не вошло бы в состав выходных данных. Символ шаблона (%) в конце строки необходим в тех реализациях SQL, в которых длина поля sname превосходит количество букв в имени Peel (здесь это очевидно, потому что другие значения превышают четыре символа). В таком случае значение поля sname реально хранится как

```
SQL Execution Log
SELECT *
FROM Salespeople
WHERE sname LIKE 'P  1%';

snum    sname    city      comm
1001    Peel     London    0.12
Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home
```

Рис. 5.8. SELECT с использованием LIKE с _ (символом подчеркивания)

Peel, за ним следует ряд пробелов. Следовательно, символ 'l' не является последним в строке. Символ (%) в шаблоне заменяет все пробелы. Все вышеперечисленное не относится к полю sname типа VARCHAR.

Чтобы найти в строке символ подчеркивания или процента, в предикате LIKE любой символ можно определить как Escape-символ. Он используется в предикате непосредственно перед символом процента или подчеркивания и означает, что следующий за ним символ интерпретируется именно как обычный символ, а не как символ шаблона. Например, поиск символа подчеркивания в столбце sname можно задать следующим образом:

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

Для тех данных, которые хранятся в таблице в текущий момент времени, выходных данных нет, поскольку в именах продавцов нет подчеркиваний. Предложение ESCAPE определяет '/' как Escape-символ, который используется в LIKE-строке, за ним следуют символ процента, символ подчеркивания или сам символ '/', т.е. тот символ, поиск которого будет осуществляться в столбце и который уже не интерпретируется как символ шаблона. Escape-символ может быть единственным символом и применяться только к единственному символу, который следует за ним. В приведенном примере начальный и конечный символы процента являются символами шаблона, только символ подчеркивания представляет собой символ как таковой.

Escape-символ может использоваться и в своем собственном значении. Другими словами, если нужно найти в столбце Escape-символ, то его необходимо ввести дважды. Первый раз он действует как обычный Escape-символ и означает: "следующий

символ надо понимать буквально так, как он указан", а второй раз указывает на то, что речь идет непосредственно об Escape-символе. Далее представлен пример поиска строки '_/' в столбце sname:

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

В этом случае выходных данных нет. Просматриваемая строка состоит из любой последовательности символов (%), за которыми следуют символ подчеркивания (/_), Escape-символ (//) и любая последовательность заключительных символов (%).

Работа с NULL-значениями

Часто в таблице встречаются записи с незаданными значениями какого-либо из полей, потому что значение поля неизвестно или его просто нет. В таких случаях SQL позволяет указать в поле NULL-значение. Строго говоря, NULL-значение вовсе не представлено в поле. Когда значение поля есть NULL это значит, что программа базы данных специальным образом помечает поле, как не содержащее какого-либо значения для данной строки (записи). Дело обстоит не так в случае простого приписывания полю значения "нуль" или "пробел", которые база данных трактует как любое другое значение. Поскольку NULL не является значением как таковым, он не имеет типа данных. NULL может размещаться в поле любого типа. Тем не менее, NULL, как NULL-значение, часто используется в SQL.

Предположим, появился покупатель, которому еще не назначен продавец. Чтобы констатировать этот факт, нужно ввести значение NULL в поле snum, а реальное значение включить туда позже, когда данному покупателю будет назначен продавец.

Оператор IS NULL

Поскольку NULL фиксирует пропущенные значения, результат любого сравнения при наличии NULL-значений неизвестен. Когда NULL-значение сравнивается с любым значением, даже с NULL-значением, результат просто *неизвестен*. Булево значение "неизвестно" ведет себя также, как "ложь" — строка, на которой предикат принимает значение "неизвестно", не включается в результат запроса — при одном важном исключении: NOT от лжи есть истина (NOT (false)=true), тогда как NOT от неизвестного значения есть также неизвестное значение. Следовательно, такое выражение как "city = NULL" или "city IN (NULL)" является неизвестным независимо от значения city.

Часто необходимо различать false и unknown — строки, содержащие значения столбца, не удовлетворяющие предикату, и строки, которые содержат NULL. Для этой цели SQL располагает специальным оператором IS, который используется с ключевым словом NULL для локализации NULL-значения.

Для нахождения всех записей со значениями NULL в таблице Customers в столбце city следует ввести:

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

В данном случае выходных данных не будет, поскольку в конкретных простых таблицах нет NULL-значений.

NULL-значения чрезвычайно важны, поэтому имеет смысл вернуться к ним позже.

Использование NOT со специальными операторами

Специальные операторы, которые были рассмотрены в этой главе, могут непосредственно предшествовать булеву оператору NOT. Этим они отличаются от операторов сравнения, которые должны содержать NOT перед всем выражением. Например, если не осуществляется поиск NULL-значений, а, напротив, необходимо исключить их из выходных данных, то нужно использовать NOT для того, чтобы придать предикату противоположное значение:

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

Если NULL-значения отсутствуют (в данном случае это именно так), то в результате выполнения этого запроса будет получена вся таблица Customers, что эквивалентно вводу:

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

что тоже приемлемо.

Можно также использовать NOT и IN:

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

Другой способ выразить то же самое:

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

Выходные данные для этого запроса представлены на рис. 5.9.

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

snum   sname      city        comm
1007   Rifkin     Barcelona   0.15
1003   Axelrod    New York    0.10
Browse : ↑↓←→ PgDn PgUp ──►| |◄── Home
```

Рис. 5.9. Использование NOT с IN

Аналогичным образом можно использовать NOT BETWEEN и NOT LIKE.

Итоги

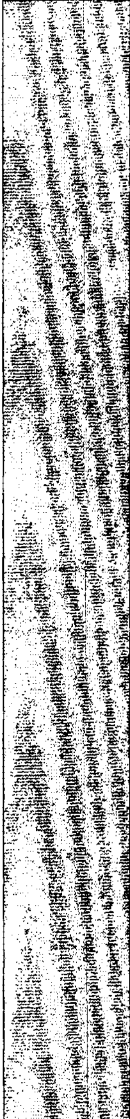
Теперь вы научились конструировать предикаты в терминах отношений, специально определенных для SQL, искать значения в определенном диапазоне (BETWEEN) или значения, принадлежащие определенному множеству (IN), искать символьные значения, удовлетворяющие заданному символьному шаблону (LIKE).

Вы поняли, как SQL реагирует на пропуски данных (вполне реальная ситуация для мира баз данных) с помощью NULL-значений. Вы умеете извлекать NULL-значения или исключать их из выходных данных, применяя оператор IS NULL (или IS NOT NULL). Полный набор стандартных математических функций и специальных операторов позволяет переходить к специальным функциям SQL, которые оперируют целыми группами, а не на единичными значениями. Этому посвящена глава 6.

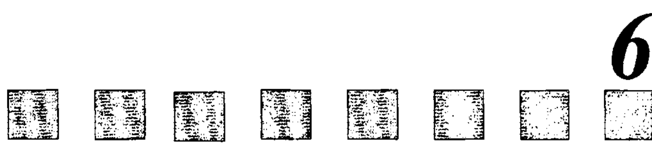
Работаем на SQL

1. Запишите два запроса, которые выдают сведения о всех заявках, принятых 3 или 4 октября 1990 года.
2. Запишите запрос, который выбирает всех покупателей, обслуживаемых Peel или Motika. (Подсказка: поле snum связывает две таблицы друг с другом.)
3. Запишите запрос, который выбирает всех покупателей, имена которых начинаются на любую из букв от 'A' до 'G'.
4. Запишите запрос, который выбирает всех покупателей, имена которых начинаются на 'C'.
5. Запишите запрос, который выбирает все заявки, у которых в поле amt (amount) указано значение 0 или NULL.

(Ответы см. в приложении А.)



6



Суммирование данных с помощью функций агрегирования

В этой главе осуществляется переход к более сложным запросам на извлечение значений из базы данных и получение информации о базе данных, основываясь на этих значениях. Это делается с помощью функций агрегирования и суммирования, которые группируют значения поля и сводят их к единственному значению. Вы узнаете, как применять эти функции, как определять группы значений, для которых они применимы и какие группы выбираются в качестве выходных данных, при каких условиях можно комбинировать значения полей с этой производной информацией в единственном запросе.

Что такое функции агрегирования?

Запросы могут обобщать не только группы значений, но и значения одного поля. Для этого применяются агрегатные функции. Они дают единственное значение для целой группы строк таблицы. Ниже приводится список этих функций:

- COUNT определяет количество строк или значений поля, выбранных посредством запроса и не являющихся NULL-значениями;
- SUM вычисляет арифметическую сумму всех выбранных значений данного поля;
- AVG вычисляет среднее значение для всех выбранных значений данного поля;
- MAX вычисляет наибольшее из всех выбранных значений данного поля;
- MIN вычисляет наименьшее из всех выбранных значений данного поля.

Как используются функции агрегирования?

Функции агрегирования используются как имена полей в предложении запроса SELECT с одним исключением: имена полей применяются как аргументы. Для SUM и AVG могут использоваться только цифровые поля. Для COUNT, MAX и MIN — цифровые и символьные поля. При употреблении с символьными полями MAX и MIN применяются к ASCII-эквивалентам: MIN предполагает минимальное (первое), а MAX — максимальное (последнее) значения в соответствии с алфавитным порядком (более детально алфавитное упорядочение рассмотрено в главе 4).

Чтобы найти сумму (SUM) всех заявок из таблицы Orders, можно ввести следующий запрос, выходные данные для которого представлены на рис. 6.1:

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

Рис. 6.1. Выбор суммы

Такая операция существенно отличается от выбора поля тем, что выходные данные содержат единственное значение независимо от количества строк в таблице. По этой причине агрегатные функции и поля не могут выбираться одновременно, если только не используется предложение GROUP BY.

Похожей операцией является поиск среднего значения (выходные данные для следующего запроса представлены на рис. 6.2):

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

Рис. 6.2. Выбор среднего значения

Специальные атрибуты в COUNT

Функция COUNT отличается от предыдущих тем, что подсчитывает количество значений в данном столбце или количество строк в таблице. Когда подсчитываются значения по столбцу, в команде используется DISTINCT для подсчета числа различных значений данного поля. Можно использовать его, например, для подсчета количества продавцов, имеющих в настоящее время заказы в таблице Orders (выходные данные представлены на рис. 6.3):

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

Использование DISTINCT. В данном примере DISTINCT вместе со следующим за ним именем поля, к которому он применяется, заключен в круглые скобки и не следует непосредственно за SELECT, как это было в примере главы 3. Такая форма применения DISTINCT с COUNT к отдельным столбцам предписывается стандартом ANSI, но многие программы не придерживаются этого требования. Можно использовать множество COUNT для DISTINCT полей в одном запросе; этот случай, рассмотренный в главе 3, отличается от случая применения DISTINCT к строкам.

Указанным способом DISTINCT можно применять с любой функцией агрегирования, но чаще всего он используется с COUNT. Применение его с MAX и MIN бесполезно; а используя SUM и AVG, необходимо включение в выходные данные повторяющихся значений, так как они влияют на сумму и среднее для значений всех столбцов.

Использование COUNT со строками, а не со значениями. Для подсчета общего количества строк в таблице следует использовать функцию COUNT со звездоч-

Рис. 6.3. Подсчет количества значений поля

Рис. 6.4. Подсчет количества строк, а не значений поля

кой вместо имени поля так, как показано в следующем примере, выходные данные для которого представлены на рис. 6.4:

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

COUNT со звездочкой включает как NULL-значения, так и повторяющиеся значения, значит DISTINCT в этом случае не применим. По этой причине в результате получается число, превышающее COUNT для отдельного поля, который исключает из этого поля все избыточные строки или NULL-значения. DISTINCT исключен для COUNT(*), поскольку он не имеет смысла для хорошо спроектированной и управляемой базы данных. В такой базе данных не должно быть ни строк, содержащих в каждом поле только NULL-значения, ни полностью повторяющихся строк (поскольку первые не содержат никаких данных, а последние полностью избыточны). С другой стороны, если имеются избыточные или содержащие одни NULL-значения строки, то нет необходимости применять COUNT для избавления от этой информации.

Использование дубликатов в агрегатных функциях. Агрегатные функции могут также (во многих реализациях) иметь аргумент ALL, который размещается перед именем поля, как и DISTINCT, но обозначает противоположное: включить дубликаты. Требования ANSI не допускают подобного для COUNT, но многие реализации игнорируют это ограничение. Различие между ALL и * при использовании COUNT заключается в следующем:

- ALL использует имя поля в качестве аргумента;
- ALL не подсчитывает NULL-значения.

Поскольку * является единственным аргументом, который включает NULL-значения и используется только с COUNT, функции, отличные от COUNT, игнорируют NULL-значения в любом случае. Следующая команда осуществляет подсчет количества значений поля rating, отличных от NULL-значений, в таблице Customers (включая повторения):

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

Агрегаты, построенные на скалярных выражениях

До сих пор были использованы агрегатные функции с одним полем в качестве аргумента. Можно использовать агрегатные функции с аргументами, которые состоят из скалярных выражений, включающих одно поле или большее количество полей. (При этом не разрешается применять DISTINCT.) Предположим, таблица Orders содержит дополнительный столбец с величиной предыдущего баланса (blnc) для каждого покупателя. Можно найти текущий баланс, добавив значение поля amount (amt) к значению поля blnc. Можно найти наибольшее значение текущего баланса:

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

В процессе выполнения этого запроса для каждой строки таблицы выполняется сложение значений двух указанных полей записи и выбирается наибольшее из полученных значений. Конечно, поскольку покупатели могут иметь несколько заказов, их окончательный баланс в данном случае оценивается отдельно для каждого заказа. Предполагается, что последняя заявка имеет наибольшее значение баланса данного покупателя. В противном случае в предыдущем примере мог быть выбран старый баланс. В SQL можно часто использовать скалярное выражение вместе с полями или вместо них.

Предложение GROUP BY

Предложение GROUP BY позволяет определять подмножество значений отдельного поля в терминах другого поля и применять функции агрегирования к полученному подмножеству. Это дает возможность комбинировать поля и агрегатные функции в одном предложении SELECT. Например, предположим, что нужно найти наибольший заказ из тех, что получил каждый из продавцов. Можно сделать отдельный запрос на каждого продавца, выбрав MAX (amt) для таблицы Orders для каждого значения поля snum и используя GROUP BY, однако, возможно объединить все в одной команде:

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

Выходные данные для этого запроса представлены на рис. 6.5.

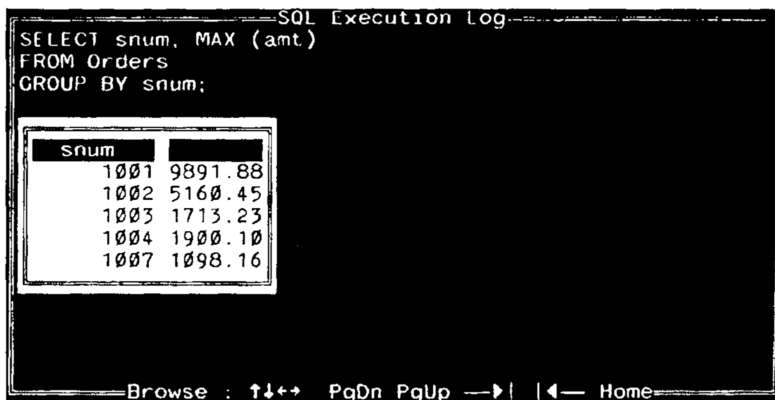


*Рис. 6.5. П*одсчет максимального количества (amounts) для каждого продавца (salesperson)

GROUP BY применяет агрегатные функции отдельно к каждой серии групп, которые определяются общим значением поля. В данном случае каждая группа состоит из всех тех строк, которые имеют одно и то же значение snum, а функция MAX применяется отдельно к каждой такой группе. Это означает, что поле, к которому применяется GROUP BY по определению имеет на выходе только одно значение на каждую группу, что соответствует применению агрегатных функций. Такая совместимость результатов и позволяет комбинировать агрегаты с полями указанным способом.

Можно также применять GROUP BY с многозначными полями. Обращаясь к предыдущему примеру, можно предположить, что необходимо увидеть наибольший заказ, сделанный каждому продавцу на каждую дату. Для этого нужно сгруппировать данные таблицы Orders по дате (date) внутри одного и того же поля salesperson и применить функцию MAX к каждой группе. В результате будет получено:

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

Выходные данные для этого запроса представлены на рис. 6.6.

Пустые группы, т.е. даты, когда данный продавец не получал заказов, в результате не представлены.

```
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
1004	10/03/1990	1900.10
1007	10/03/1990	1098.16

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 6.6. Поиск максимальных заявок (orders) для каждого продавца (salesperson) на каждый день

Предложение HAVING

Обращаясь к предыдущему примеру, можно предположить, что интересны только покупки, превышающие $3000.00. Однако использовать агрегатные функции в предложении WHERE нельзя (если только не применяется подзапрос, который будет объяснен позднее), поскольку предикаты оцениваются в терминах единственной строки, тогда как агрегатные функции оцениваются в терминах групп строк. Это значит, что нельзя формулировать запрос следующим образом:

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

Это неприемлемо с точки зрения точной интерпретации ANSI. Чтобы увидеть максимальную покупку, превышающую $3000.00, следует использовать предложение HAVING. Оно определяет критерий, согласно которому определенные группы исключаются из числа выходных данных, так же, как предложение WHERE делает это для отдельных строк. Правильная команда выглядит так:

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

Выходные данные для этого запроса представлены на рис. 6.7.

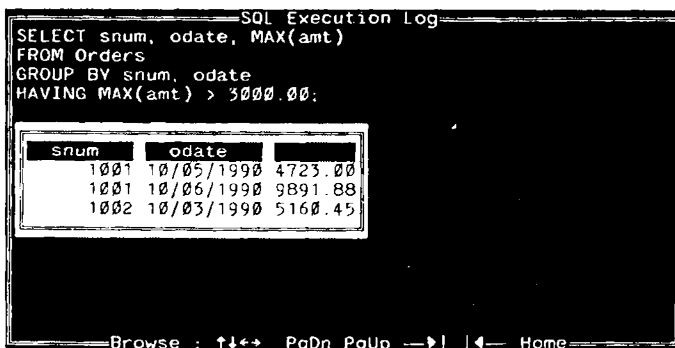


Рис. 6.7. Поглощение групп агрегатными значениями

Аргументы HAVING подчиняются тем же правилам, что и аргументы SELECT в команде, использующей GROUP BY, и должны иметь единственное значение для каждой выходной группы. Следующая команда некорректна:

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

В предложении HAVING нельзя указывать поле odate, поскольку оно может иметь (и действительно имеет) более одного значения для каждой выходной группы. HAVING должно относиться только к агрегатам и полям, выбранным по GROUP BY. Вот корректный способ формулировки приведенного запроса (выходные данные представлены на рис. 6.8):

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

Поскольку odate не является и не может быть выбранным полем, значимость полученных здесь данных, конечно, менее очевидна, чем в некоторых других примерах. Выходные данные должны были бы содержать нечто вроде следующего предложения: "Вот наибольшие заявки на 3 октября". В главе 7 будет объяснено, как вставить текст в выходные данные.

HAVING может иметь только такие аргументы, у которых единственное значение для группы выходных данных. На практике чаще всего применяются агрегатные

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

 snum
   1001   767.19
   1002  5160.45
   1004  1900.10
   1007  1098.16

Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home
```

Рис. 6.8. Максимум для каждого продавца за 3 октября 1990 г.

функции, но можно осуществлять выбор полей и с помощью GROUP BY. Например, можно взглянуть на самые большие заказы для Serres и Rifkin:

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

Выходные данные для этого запроса представлены на рис. 6.9.

Не используйте вложенные агрегаты

В версии языка SQL, определяемой ANSI, нельзя применять агрегатную функцию с агрегатом в качестве аргумента. Предположим, нужно определить в какой день было сделано наибольшее число заявок. Если ввести команду:

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

то она, вероятно, будет отвергнута. (Существуют реализации, которые не учитывают такие ограничения, что даст определенные преимущества, поскольку вложенность агрегатов может быть полезной, даже если она и вызывает сомнения.) Например, в данной команде SUM должна быть применена к каждой odate-группе, а MAX — ко всем группам, причем она выдает единственное значение для всех этих групп. В то же время предложение GROUP BY предполагает, что должна быть одна строка выходных данных на каждую группу odate.

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

snum
     1002 5160.45
     1007 1098.16

Browse : ↑↓←→ PgDn PgUp —►| |◄— Home
```

Рис. 6.9. Использование HAVING с GROUP BY

Итоги

Теперь вы научились использовать запросы иначе. Возможность выводить, а не просто локализовать значения очень важна и означает, что не надо отслеживать путь получения информации, если можно сформулировать запрос на ее вывод. Запрос дает результаты на текущий момент, тогда как таблица итогов и средних величин полезна на тот момент, когда она в последний раз обновлялась. Это не значит, что агрегатные функции во всех случаях могут полностью снять потребность проследить путь информации.

Агрегатные функции применимы к группам значений, определяемым предложением GROUP BY. Эти группы имеют общее значение поля и могут использоваться внутри других групп, имеющих общее значение поля. Предикаты нужны для определения строк, к которым применяется функция агрегирования. Такое комбинирование позволяет получить агрегаты на основе полностью определенных подмножеств значений поля. Затем вы можете задать условие исключения определенных результирующих групп с помощью предложения HAVING.

Вы уже знаете, как запросы генерируют значения. В главе 7 будет показано, эти значения можно применять.

Работаем на SQL

1. Запишите запрос, который подсчитывает все заявки за 3 октября 1990 года.
2. Запишите запрос, который подсчитывает количество различных городов (не NULL) в таблице Customers.
3. Запишите запрос, который выбирает наименьшую заявку для каждого покупателя.
4. Запишите запрос, который выбирает первого в алфавитном порядке покупателя, имя которого начинается с 'G'.
5. Запишите запрос, который выбирает максимальный рейтинг (rating) для каждого города.
6. Запишите запрос, который подсчитывает количество продавцов, получающих заказы каждый день. (Продавца, имеющего более одного заказа в день, следует включить в пересчет только один раз.)

(Ответы см. в приложении А.)

7

Форматирование результатов запросов

Назначение этой главы — расширить возможности обработки результатов запросов. Вы узнаете, как вставить текст и константы в выбранные поля, как использовать последние в математических выражениях, результаты вычисления которых станут выходными данными и, наконец, как представить выходные данные в заданной последовательности. Последнее предполагает возможность упорядочить выходные данные по любому столбцу или по любым значениям, полученным на основе данных, содержащихся в столбце.

Строки и выражения

Во многих базах данных, использующих SQL, имеются специальные средства, позволяющие оформлять результаты запросов. Естественно, в разных программных продуктах они кардинально отличаются, но эти различия здесь не обсуждаются. Однако и стандартная версия SQL имеет ряд характерных свойств, позволяющих сделать нечто большее, чем просто вывести значения полей и функций агрегирования. О них и пойдет речь в данной главе.

Скалярные выражения с выбранными полями. Предположим, необходимо выполнить простые числовые операции с данными для представления их в более удобном виде. SQL позволяет вносить скалярные выражения и константы в выбранные поля. Эти выражения могут дополнять или заменять поля в предложениях SELECT и могут содержать множество выбранных полей. Например, если вам удобнее представить комиссионные продавцов в виде процентов, а не десятичных чисел, достаточно указать:

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 7.1. Использование выражения в запросе

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

Выходные данные для этого запроса представлены на рис. 7.1.

Выходные столбцы. Последний столбец в предыдущем примере не имеет имени, поскольку является выходным столбцом. *Выходные* столбцы — это столбцы, которые создаются с помощью запроса (в тех случаях, когда в предложении запроса SELECT используются агрегатные функции, константы или выражения), а не извлекаются непосредственно из таблицы. Поскольку имена столбцов являются атрибутами таблицы, столбцы, не переходящие из таблицы в выходные данные, не имеют имен. Почти во всех ситуациях выходные столбцы отличаются от столбцов, извлекаемых из таблицы тем, что они не поименованы.

Внесение текста в выходные данные запроса. Буква 'A', не обозначающая ничего кроме самой себя, является *константой*, как и число 1. Константы, а также текст, можно включать в предложение запроса SELECT. Однако, буквенные константы, в отличие от числовых, нельзя использовать в выражениях. В SELECT-предложение можно включить 1+2, но не 'A' + 'B', поскольку 'A' и 'B' здесь просто буквы, а не переменные или символы, используемые для обозначения чего-либо отличного от них самих. Тем не менее, возможность вставить текст в выходные данные запроса вполне реальна.

Можно изменить предыдущий пример, пометив комиссионные, выраженные в процентах, символом "процент" (%), что позволяет представить их в выходных данных в виде символов и комментариев, например:

```
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         %    12.00000
  1002    Serres          San Jose       %    13.00000
  1004    Motika          London         %    11.00000
  1007    Rifkin          Barcelona      %    15.00000
  1003    Axelrod         New York       %    10.00000
=====Browse : ↑↓←→  PgDn PgUp —►|  |◄— Home=====
```

Рис. 7.2. Включение символов в выходные данные

Выходные данные для этого запроса представлены на рис. 7.2.

Аналогичный прием можно применить для того, чтобы пометить выходные данные, включив в них некоторый комментарий. Однако нужно помнить, что один и тот же комментарий будет печататься не один раз для всей таблицы, а в каждой строке выходных данных. Предположим, генерируются выходные данные для отчета, в котором фиксируется количество заказов на каждый день. Выходные данные можно пометить (см. рис. 7.3), оформив запрос следующим образом:

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

Грамматическую ошибку в выходных данных на 10/05/1990 можно исправить, но запрос при этом сильно усложнится. (Для этого пришлось бы использовать два запроса и операцию объединения UNION, которая будет рассмотрена в главе 14.) Вам может быть полезен единственный неизменный комментарий для каждой строки таблицы, но он ограничен. Иногда более элегантное и полезное решение состоит в том, чтобы выдать один и тот же комментарий для всех выходных данных в целом или разные комментарии для различных строк.

Многие программные продукты, использующие SQL, часто предоставляют пользователям генераторы отчетов, которые применяются для форматирования и улучшения формы выходных данных. Встроенный SQL тоже может употреблять средства форматирования того языка, в который он встроен. SQL предназначен прежде всего для обработки данных. Его выходными данными является информация, а программа,

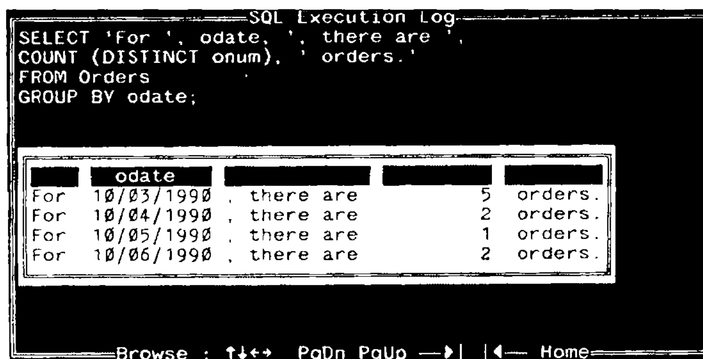


Рис. 7.3. Комбинирование текста, значений полей и агрегатов

используюшая SQL, может принимать эту информацию и выводить ее в более наглядной форме. Однако, это уже лежит за пределами самого SQL.

Упорядочение выходных полей

Таблицы являются неупорядоченными множествами, и исходящие из них данные необязательно представляются в какой-либо определенной последовательности. В SQL применяется команда ORDER BY, позволяющая внести некоторый порядок в выходные данные запроса. Она их упорядочивает в соответствии со значениями одного или нескольких выбранных столбцов. Множество столбцов упорядочиваются один внутри другого, как в случае применения GROUP BY, и можно задать возрастающую (ASC) или убывающую (DESC) последовательность сортировки для каждого из столбцов. По умолчанию принята возрастающая последовательность сортировки.

Таблица заявок (Orders), упорядоченная по номеру заявки, (обратить внимание на значения в столбце cnum) выглядит так:

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

Выходные данные представлены на рис. 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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 7.4. Упорядочение выходных данных по убыванию поля cnum

Упорядочение по множеству столбцов

Внутри уже произведенного упорядочения по полю cnum можно упорядочить таблицу и по другому столбцу, например, amt (выходные данные представлены на рис. 7.5):

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

Так можно использовать ORDER BY одновременно для любого количества столбцов. Во всех случаях столбцы, по которым выполняется сортировка, входят в число выбранных. Этому требованию стандарта ANSI удовлетворяет большинство систем. Например, следующая команда неверна:

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

Поскольку поле cnum отсутствует в списке выбранных полей, предложение ORDER BY не может его найти для упорядочения выходных данных. Даже если система позволяет это сделать, значимость такого упорядочения неочевидна, поскольку само поле, по которому выполняется сортировка, не представлено в выходных данных. Поэтому включение в них всех столбцов, используемых в предложении ORDER BY, весьма желательно.

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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 7.5. Упорядочение выходных данных по множеству полей

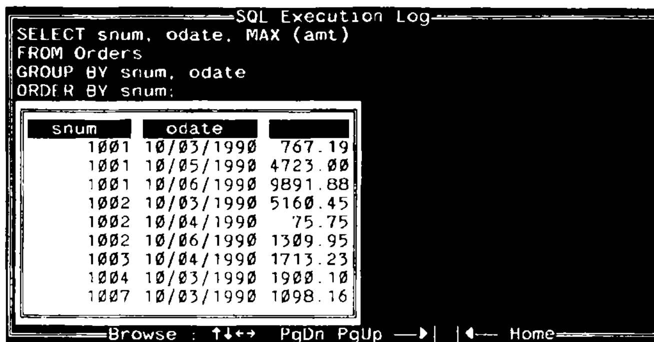
Упорядочение составных групп

ORDER BY может использоваться с GROUP BY для упорядочения групп. ORDER BY всегда выполняется последней. Вот пример из предыдущей главы с добавлением предложения ORDER BY. До этого выходные данные были сгруппированы, но порядок групп был произвольным; теперь группы выстроены в определенной последовательности:

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

Выходные данные представлены на рис. 7.6.

Поскольку в команде не указан способ упорядочения, по умолчанию применяется возрастающий.



SQL Execution Log

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

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
1004	10/03/1990	1900.10
1007	10/03/1990	1098.16

Browse : ↑↓←→ PgDn PgUp —▸| |◂— Home

Рис. 7.6. Упорядочение групп

Упорядочение результата по номеру столбца

Вместо имен столбцов для указания полей, по которым упорядочиваются выходные данные, можно использовать номера. Но ссылаясь на них, следует иметь в виду, что это номера в определении выходных данных, а не столбцов в таблице. Т.е. первое поле, имя которого указано в SELECT, является для предложения ORDER BY полем с номером 1, независимо от его расположения в таблице. Например, можно применить следующую команду, чтобы увидеть определенные поля таблицы Salespeople, упорядоченные по убыванию поля commission (comm) (выходные данные представлены на рис. 7.7):

```
SQL Execution Log
SELECT sname, comm
FROM Salespeople
ORDER BY 2 DESC;

sname        comm
Rifkin       0.15
Serres       0.13
Peel         0.12
Motika       0.11
Axelrod      0.10

Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home
```

Рис. 7.7. Упорядочение с использованием номеров столбцов

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

Мы рассматриваем это свойство ORDER BY для того, чтобы продемонстрировать возможность его использования со столбцами выходных данных; эта процедура аналогична применению ORDER BY со столбцами таблицы. Столбцы, полученные с помощью функций агрегирования, константы или выражения в предложении запроса SELECT, можно применить и с ORDER BY, если на них ссылаются по номеру. Например, чтобы подсчитать заявки (orders) для каждого продавца (salespeople) и вывести результаты в убывающем порядке, как показано на рис. 7.8:

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

В этом случае был использован номер столбца, но так как выходной столбец не имеет имени, саму функцию агрегирования применять не понадобилось. В соответствии со стандартом ANSI SQL, следующий запрос не работает, хотя в некоторых системах он воспринимается без проблем:

```
SELECT snum, COUNT (DISTINCT onum)
   FROM Orders
```

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

snum	
1001	3
1002	3
1007	2
1003	1
1004	1

Рис. 7.8. Упорядочение выходных столбцов

```
GROUP BY snum
ORDER BY COUNT (DISTINCT onum) DESC;
```

Многими системами такая команда воспринимается как ошибочная.

ORDER BY с NULL-значениями

Если в поле, которое используется для упорядочения выходных данных, существуют NULL-значения, то все они следуют в конце или предшествуют всем остальным значениям этого поля. Конкретный вариант не оговаривается стандартом ANSI, вопрос решается индивидуально для каждого программного продукта, и один из этих вариантов принимается.

Итоги

Теперь с помощью запросов вы можете получить нечто большее, чем простые значения полей и функций агрегирования для данных, представленных в таблице. Значения полей используются в выражениях: например, можно умножить числовое поле на 10 или умножить его на другое числовое поле. Кроме того, константы, в том числе и символьные, можно включать в состав выходных данных. Все это дает возможность выводить текст непосредственно в результат запроса наравне с данными, содержащи-

мися в таблице, что, в свою очередь, позволяет помечать или комментировать выходные данные различными способами.

Вы научились управлять порядком вывода результатов запроса. Несмотря на то, что сама таблица базы данных остается неупорядоченной, предложение ORDER BY позволяет управлять порядком вывода строк выходных данных конкретного запроса. Порядок представления выходных данных запроса может быть возрастающим или убывающим, и столбцы можно упорядочить один внутри другого.

В этой главе введено понятие выходных столбцов, которые можно применять для упорядочения результатов запроса, но эти столбцы не поименованы, поэтому для ссылки на них в предложении ORDER BY используется порядковый номер выходного столбца из предложения SELECT.

В главе 8 будут рассмотрены более сложные запросы. Вы узнаете, как объединить в одной единственной команде запросы к множеству таблиц базы данных, установив между ними связи.

Работаем на SQL

1. Предположим, каждый продавец имеет 12% комиссионных. Запишите запрос к таблице Orders, который выдает номер заявки (order number), номер продавца (salesperson number) и общее значение комиссионных продавца (amount of salesperson's commission). Выходные данные упорядочите по значениям последнего столбца.

2. Запишите запрос к таблице Customers, который находит максимальный рейтинг для каждого города. Представьте выходные данные в таком виде:

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

   (Для города (city) максимальный рейтинг составляет: (rating).)

3. Запишите запрос, который выдает список покупателей (customers) в порядке убывания рейтинга (rating). Поле rating в выходных данных должно быть первым, за ним следуют имя покупателя (customer's name) и номер покупателя (customer's number).

4. Запишите запрос, который подводит итоги по заказам на каждый день и представляет результаты в убывающем порядке.

(Ответы на вопросы см. в приложении А.)

8

Использование множества таблиц в одном запросе

До сих пор каждый рассматриваемый запрос базировался на единственной таблице. После изучения этой главы вы сможете формулировать запросы с помощью одной команды для любого (произвольного) количества таблиц. Это исключительно мощная процедура, поскольку осуществляется не только комбинирование выходных данных из множества таблиц, но и устанавливаются связи между ними. Вы познакомитесь с различными видами таких связей, узнаете, как они определяются и используются.

Соединение таблиц

Одна из наиболее важных черт запросов SQL состоит в их способности определять связи между множеством таблиц и отображать содержащуюся в них информацию в терминах этих связей в рамках единственной команды. Операция такого рода называется *соединением* (*join*) и является одной из самых мощных операций для реляционных баз данных. Как уже говорилось в главе 1, преимущество реляционного подхода заключается в связях (relationships), которые можно установить между элементами данных в таблице. С помощью соединений непосредственно связывается информация, содержащаяся в таблицах, независимо от их числа, а также между отдельными частями любой таблицы.

При операции соединения таблицы перечисляются в предложении запроса FROM; имена таблиц разделяются запятыми. Предикат запроса может ссылаться на любой столбец любой из соединяемых таблиц и, следовательно, может использоваться для установления связей между ними. Обычно предикат сравнивает значения в столбцах различных таблиц для того, чтобы определить, удовлетворяется ли условие WHERE.

Имена таблиц и столбцов

Полное имя столбца состоит из имени таблицы, непосредственно за которым стоит точка, а за ней — имя столбца. Приведем несколько примеров:

Salespeople.snum

Customers.city

Orders.odate

В приводимых ранее примерах имена таблиц можно было опускать, поскольку запросы адресовались только к одной таблице, и SQL выполнял подстановку имени соответствующей таблицы в качестве префикса. Даже при формулировке запроса к множеству таблиц их имена можно опустить, если все столбцы этих таблиц различны. Однако так бывает далеко не всегда.Например, есть две простые таблицы с одинаковыми именами столбцов-city. Если для них необходимо выполнить операцию соединения,

то следует указать Salespeople.city или Customers.city, что дает возможность SQL однозначно определить, о каком столбце идет речь.

Выполнение операции соединения (join)

Предположим, нужно установить связь между продавцами (Salespeople) и покупателями (Customers) в соответствии с местом их проживания, чтобы получить все возможные комбинации продавцов и покупателей из одного города. Для этого необходимо взять продавца из таблицы Salespeople и выполнить по таблице Customes поиск всех покупателей, имеющих то же значение в столбце city. Это можно сделать, введя следующую команду (выходные данные представлены на рис. 8.1):

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

Поскольку поле city присутствует в каждой из таблиц Salespeople и Customers, имена таблиц используются перед именем city в качестве префиксов. Это необходимо в том случае, когда два или более поля имеют одинаковые имена, но для ясности и полноты картины полезно включать в соединения имя таблицы. В дальнейшем имена таблиц будут использоваться там, где это необходимо, чтобы было понятно, где они нужны, а где нет.

Выполняя операцию соединения, необходимо генерировать все возможные сочетания строк для двух или более таблиц и проверять истинность предиката на каждом таком сочетании. В предыдущем примере SQL берет строку, соответствующую

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

cname	sname	city
Hoffman	Peel	London
Clemens	Peel	London
Liu	Serres	San Jose
Cisneros	Serres	San Jose
Hoffman	Motika	London
Clemens	Motika	London

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 8.1. Соединение двух таблиц

продавцу Peel из таблицы Salespeople, и комбинирует ее с каждой строкой таблицы Customers, выбирая по одной строке из этой таблицы. Если на данной комбинации строк предикат имеет значение "истинно", т.е. поле city строки таблицы Customers содержит значение London такое же, как и у Peel, то указанные в предложении SELECT поля из комбинации этих строк являются выходными данными. Те же действия предпринимаются относительно каждого продавца из таблицы Salespeople (некоторые из них не имеют покупателей, находящихся в том же городе).

Операция соединения таблиц посредством ссылочной целостности

Эта операция применяется для использования связей, встроенных в базу данных. В предыдущем примере связь между таблицами была установлена с помощью операции соединения. Но эти таблицы уже связаны по значениям полем snum. Такая связь называется состоянием ссылочной целостности, о которой упоминалось в главе 1. Стандартное применение операции соединения состоит в извлечении данных в терминах этой связи. Чтобы показать соответствие имен покупателей именам продавцов, обслуживающих этих покупателей, используется следующий запрос:

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

Выходные данные для этого запроса представлены на рис. 8.2.

```
=============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

=====Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home=====
```

Рис. 8.2. Соединение двух продавцов с их покупателями

Это также пример соединения, в котором столбцы, используемые в формулировке предиката запроса, — в данном случае это столбцы snum в обеих таблицах — опущены из выходных данных. Выходные данные показывают, какие покупатели обслуживаются какими продавцами. Значения snum, на основании которых устанавливается связь, в данном случае не представлены, поскольку здесь они не являются существенными. Однако, действуя таким образом, нужно либо иметь уверенность в том, что выходные данные сами по себе ясны, либо дать им какие-то объяснения.

Эквисоединение и другие виды соединений

Соединение, использующее предикаты, основанные на равенствах, называется *эквисоединением*. Рассмотренные в данной главе примеры относятся именно к этой категории, поскольку все условия в предложении WHERE базируются на математических выражениях, использующих символ равенства. "City='London'" и "Salespeople.snum=Orders.snum" — примеры применения символа равенства в предикатах. Эквисоединение является, по-видимому, наиболее распространенным типом соединения, но существуют и другие. Фактически в соединении можно использовать любой оператор сравнения. Вот пример соединения другого рода (выходные данные для него представлены на рис. 8.3):

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

Эта команда полезна далеко не всегда. Она генерирует все комбинации имен продавцов и покупателей так, что первые предшествуют последним в алфавитном поряд-

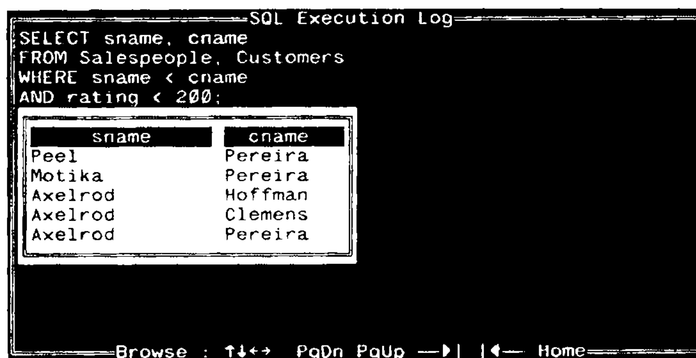


Рис. 8.3. Соединение, основанное на неравенстве

ке, а последние имеют рейтинг меньше чем 200. Обычно такие сложные связи нет необходимости конструировать, и поэтому вам полезно знать также и о других возможностях.

Соединение более чем двух таблиц

Можно конструировать запросы путем соединения более чем двух таблиц. Предположим, нужно найти все заявки покупателей, не находящихся в том же городе, что и их продавец. Для этого потребуется связать все три рассматриваемые таблицы (выходные данные представлены на рис. 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;
```

Хотя команда выглядит достаточно сложно, следуя ее логике, легко убедиться, что в выходных данных перечислены покупатели и продавцы, расположенные в разных городах (они сравниваются по полю snum), и что указанные заказы сделаны именно этими покупателями (подбор заказов устанавливается в соответствие с полями cnum и snum таблицы Orders).

```
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

Browse : ↑↓←→ PgDn PgUp —▸| |◂— Home

Рис. 8.4. Соединение трех таблиц

Итоги

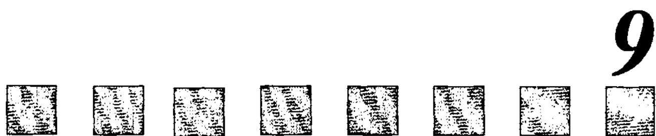
Теперь вы можете не ограничиваться рассмотрением лишь одной таблицы в некоторый момент времени, а умеете сравнивать любые поля произвольного числа таблиц и применять полученные результаты для поиска нужной информации. Эта техника настолько полезна для установления связей, что используется и для их конструирования внутри единственной таблицы. В следующей главе будет рассмотрена эффективная процедура соединения двух копий одной таблицы.

Работаем на SQL

1. Запишите запрос, который выдает каждый номер заказа, и следующее за ним имя покупателя, сделавшего этот заказ.
2. Запишите запрос, который для каждого заказа выдает после его номера имена продавцов и покупателей.
3. Запишите запрос, который выдает имена всех покупателей, обслуживаемых продавцами, имеющими комиссионные более 12%. Выходными данными должны быть имя покупателя, имя продавца и комиссионные продавца.
4. Запишите запрос, который вычисляет размер комиссионных продавца для каждого заказа покупателя с рейтингом, превышающим 100.

(Ответы см. в приложении A.)

9



Операция соединения, операнды которой представлены одной таблицей

В главе 8 было показано, как соединить две или более таблицы. Аналогичную технику можно применить для соединения двух копий одной таблицы. В настоящей главе эта процедура рассматривается подробно. Она не является простым "самоналожением", а весьма полезно для выявления определенных видов связей между элементами данных в конкретной таблице.

Как выполняется операция соединения двух копий одной таблицы

Соединение таблицы с ее же копией означает следующее: любую строку таблицы (одну в каждый момент времени) можно комбинировать с ее копией и с любой другой строкой этой же таблицы. Каждая такая комбинация оценивается в терминах предиката, как и в случае соединения нескольких различных таблиц. Это позволяет легко конструировать определенные виды связей между различными записями внутри единственной таблицы — например, осуществлять поиск пар строк с общим значением поля.

Соединение таблицы со своей копией можно представить себе следующим образом: реально таблица не копируется, но SQL выполняет команду так, как будто бы делалось именно это. Другими словами, подобный тип соединения не отличается от обычного соединения двух таблиц, за исключением того, что в данном случае они идентичны.

Алиасы

Синтаксис команды соединения таблицы с ее же копией тот же, что и для различных таблиц, с единственным исключением. В рассматриваемом случае все имена столбцов повторяются независимо от использования имени таблицы в качестве префикса. Чтобы сослаться на столбцы запроса, нужно иметь два различных имени для одной и той же таблицы. Для этого надо определить временные имена, называемые *переменными области определения*, *переменными корреляции* или просто *алиасами*. Они определяются в предложении запроса FROM. Для этого указывается имя таблицы, ставится пробел, а затем указывается имя алиаса для данной таблицы.

Приведем пример поиска всех пар продавцов, имеющих одинаковый рейтинг (выходные данные представлены на рис. 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
Browse : ↑↓←→  PgDn PgUp —▶
```

Рис. 9.1. Соединение таблицы по принципу "сама с собой"

(заметим, что на рис. 9.1 , как и в последующих примерах, видна только часть выходных данных запроса, поскольку реально все они в пределах одного окна не умещаются).

В приведенном примере команды SQL ведет себя так, как будто в операции соединения участвуют две таблицы, называемые "first" (первая) и "second" (вторая). Обе они в действительности являются таблицей Customers, но алиасы позволяют рассматривать ее как две независимые таблицы. Алиасы first и second были определены в предложении запроса FROM непосредственно за именем таблицы. Алиасы применяются также в предложении SELECT, несмотря на то, что они не определены вплоть до предложения FROM. Это совершенно оправдано. SQL сначала примет какой-либо из таких алиасов на веру, но затем отвергнет команду, если в предложении FROM запроса алиасы не определены. Время жизни алиаса зависит от времени выполнения команды. После выполнения запроса используемые в нем алиасы теряют свои значения.

Получив две копии таблицы Customers для работы, SQL выполняет операцию JOIN, как для двух разных таблиц: выбирает очередную строку из одного алиаса и соединяет ее с каждой строкой другого алиаса.

Исключение избыточности

Выходные данные включают каждую комбинацию значений дважды, причем во второй раз — в обратном порядке. Это объясняется тем, что значение появляется один раз для каждого алиаса, а предикат является симметричным. Следовательно, значение A в алиасе first выбирается в комбинации со значением B в алиасе second, и значение

A в алиасе second — в комбинации со значением B в алиасе first. В данном примере Hoffman был выбран с Clemens, а затем Clemens был выбран с Hoffman. Тоже самое произошло с Cisneros и Grass, Lie и Giovanni и т.д. Кроме того, каждая запись присоединяется сама к себе в выходных данных, например, Lie и Lie.

Простой способ исключить повторения — задать порядок для двух значений так, чтобы одно значение было меньше, чем другое, или предшествовало в алфавитном порядке. Это делает предикат ассиметричным, и одни и те же значения не извлекаются снова в обратном порядке, например:

```
SELECT first.cname, second.cname, first.rating

   FROM Customers first, Customers second

   WHERE first.rating = second.rating

      AND first.cname < second.cname;
```

Выходные данные для запроса представлены на рис. 9.2.

Hoffman предшествует Periera в алфавитном порядке, эта комбинация удовлетворяет обоим условиям предиката и появляется в составе выходных данных. Когда та же самая комбинация появляется в обратном порядке (т.е. когда Periera из таблицы с алиасом first приписывается Hoffman из таблицы с алиасом second), второе условие не выполняется. С другой стороны, Hoffman не выбирается сам по себе, как имеющий тот же рейтинг, потому что его имя не предшествует ему же самому в алфавитном поряд-

```
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
Clemens	Pereira	100
Cisneros	Grass	300

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 9.2. Исключение избыточных выходных данных при операции соединения с собственной копией

ке. Если нужно включить в запрос соединение строки с ее копией, достаточно использовать <= вместо <.

Выявление ошибок

Рассмотренное свойство SQL можно использовать для выявления ошибок определенного рода. Если посмотреть на таблицу Orders, станет ясно, что поля cnum и snum используются для определения связи. Поскольку каждому покупателю (customer) может быть назначен один и только один продавец (salesperson), в любой момент времени определенному номеру покупателя для строки из таблицы Orders соответствует строка с таким же номером продавца. Следующая команда позволяет выявить любые несоответствия такого плана:

```
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;
```

Команда выглядит сложной, но ее логика весьма прозрачна. Она берет первую строку таблицы Orders и запоминает ее под именем алиаса first, затем проверяет ее в комбинации с каждой строкой таблицы Orders под именем алиаса second. Если комбинация строк удовлетворяет предикату, она включается в состав выходных данных. В нашем случае просматривается строка, в которой поле cnum равно 2008, а поле snum равно 1007; затем осуществляется выбор каждой строки, в поле cnum которой содержится такое же значение. Если обнаруживается, что в поле snum любой из этих строк содержится другое (отличное от 1007) значение, то предикат принимает значение "истина", и в состав выходных данных включаются те поля из текущей комбинации строк, имена которых указаны в предложении SELECT. Если все значения поля snum для данного значения cnum в этой таблице одинаковы, приведенная выше команда не генерирует выходных данных.

Еще про алиасы

Хотя соединение таблиц со своими копиями — это первый встретившийся случай, когда потребовалось понятие алиаса, его употребление не лимитировано использованием для разграничения различных копий одной и той же таблицы. Алиасы можно применять при создании альтернативных имен таблиц в команде SELECT. Например, если таблицы имеют очень длинные и сложные имена, то можно определить простые, состоящие из одной буквы алиасы, например, А или В, и использовать их вместо имен таблиц в предложении SELECT и в предикате. Их можно также применять со связанными подзапросами (которые обсуждаются в главе 11).

Некоторые более сложные операции соединения

В запросе можно использовать любое количество алиасов для единственной таблицы, хотя применение более двух в одном предложении SELECT нетипично. Предположим, продавцам (salespeople) еще не назначили покупателей (customers). Политика кампании состоит в том, чтобы назначить всем продавцам по три покупателя, каждому из которых приписывается одно из трех возможных значений рейтинга. Небходимо решить, как осуществить такое распределение, и использовать следующие запросы для просмотра всех возможных комбинаций назначаемых покупателей (выходные данные представлены на рис. 9.3):

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

Этот запрос находит все возможные комбинации покупателей (customers) с тремя значениями рейтинга таким образом, что в первом столбце расположены покупатели с рейтингом 100, во втором столбце — покупатели с рейтингом 200, в третьем столбце — покупатели с рейтингом 300. Они повторены во всех возможных комбинациях. Это своего рода группирование данных, которое нельзя выполнить средствами GROUP BY или ORDER BY, поскольку они сравнивают значения только из одного столбца.

Каждый алиас или таблицы, имена которых упомянуты в предложении FROM запроса SELECT, использовать необязательно. Иногда алиас или таблица запрашиваются таким образом, что на них ссылаются предикаты запроса. Например, следующий запрос находит всех покупателей (customers), расположенных в городах, где действует

SQL Execution Log

AND c.rating = 300;

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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 9.3. Комбинирование покупателей с различными значениями рейтинга

продавец (salesperson) Serres (snum 1002) (выходные данные представлены на рис. 9.4):

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

Алиас a сделает предикат ложным, за исключением случаев, когда значение столбца snum равно 1002. Так алиас исключает всех покупателей, кроме покупателей продавца Serres. Алиас b принимает значение "истина" для всех строк с тем же значением города (city), что и текущее значение города (city) в a; в процессе выполнения запроса строка с алиасом b делает предикат истинным всякий раз, когда в поле city этой строки представлено то же значение, что и в поле city строки с алиасом a. Поиск строк алиаса b выполняется исключительно для сравнения значений с алиасом a, из строк с алиасом b никакого реального выбора данных не выполняется. Покупатели (customers) продавца Serres расположены в одном и том же городе, значит выбор их из алиаса a не является необходимым. Итак, алиас a локализует строки покупателей (customers) Serres, Liu и Grass. Алиас b находит всех покупателей (customers), расположенных в одном из городов (San Jose и Berlin соответственно), включая, конечно, самих Liu и Grass.

Можно конструировать соединения (joins), которые содержат различные таблицы и алиасы единственной таблицы. Следующий запрос соединяет таблицу Customers с ее копией для нахождения всех пар покупателей, обслуживаемых одним и тем же продавцом. В любой момент времени он соединяет покупателя (customer) с таблицей Salespeople для того, чтобы определить имя продавца (salesperson) (выходные данные для запроса представлены на рис. 9.5):

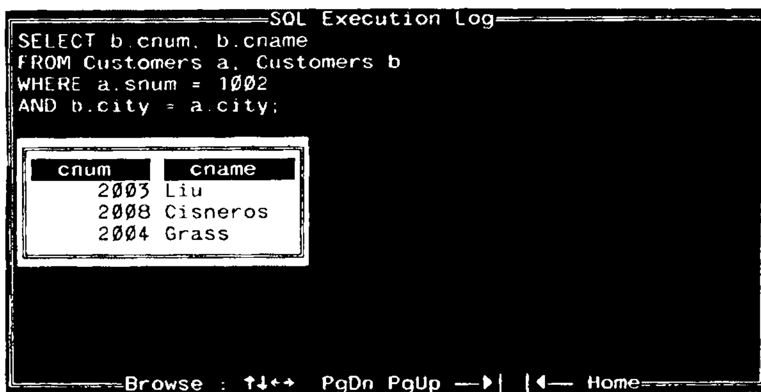


Рис. 9.4. Поиск покупателей, расположенных в тех городах, где действует продавец Serres

```
SQL Execution Log
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;

 sname          snum   cname       cname
 Serres         1002   Liu         Grass
 Peel           1001   Hoffman     Clemens

Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home
```

Рис. 9.5. Соединение таблицы с ее копией и с другой таблицей

```
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;
```

Итоги

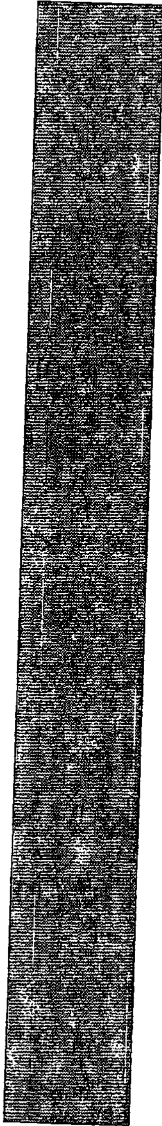
Вы узнали, что такое операции соединения (joins), как их применять для конструирования связей внутри одной таблицы, различных таблиц или двух одинаковых таблиц, где эти возможности могут оказаться полезными. Теперь вам знакомы термины — диапазон переменных, переменные связи, алиасы (в различных программных продуктах и у разных авторов изложения материала по SQL терминология варьируется, поэтому здесь объяснены все три термина). Вы больше знаете о том, как в действительности работают запросы.

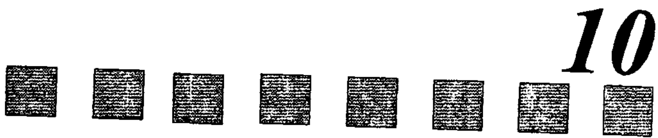
Следующий шаг после комбинации в запросе множества таблиц или множества копий единственной таблицы состоит в таком объединении множества запросов, при котором один запрос генерирует выходные данные, управляющие работой другого запроса. Подробнее об этой эффективной операции SQL мы поговорим в 10 и последующих главах.

Работаем на SQL

1. Запишите запрос, который позволяет получить все пары продавцов (Salespeople), проживающих в одном городе. Из результата необходимо исключить комбинации Salespeople с их копиями и повторяющиеся строки, отличающиеся порядком следования.
2. Запишите запрос, который генерирует все пары номеров для данного покупателя, имена покупателей и исключает повторяющиеся значения, как определено в пункте 1.
3. Запишите запрос, который позволяет получить имена и города для всех покупателей (customers), с тем же рейтингом, что и у Hoffman. Запишите запрос, использующий значение поля cnum вместо rating так, чтобы можно было использовать этот вариант запроса даже в случае изменения значения поля rating.

(Ответы см. в приложении А.)





10

Вложение запросов

В конце прошлой главы мы отметили, что одни запросы могут управлять другими. В большинстве случаев это можно сделать, размещая один запрос внутри предиката, помещенного в другом, и используя выходные данные вложенного запроса для определения истинности или ложности предиката. Из этой главы вы узнаете, какого рода операторы могут использовать подзапросы, как их применять с DISTINCT, агрегатными функциями и выражениями вывода; как использовать подзапросы с предложением HAVING и получать указатели правильного способа применения подзапросов.

Как выполняются подзапросы?

SQL позволяет вкладывать запросы друг в друга. Обычно внутренний запрос генерирует значения, которые тестируются на предмет истинности предиката. Предположим, известно имя, но мы не знаем значения поля snum для продавца Motica. Необходимо извлечь все ее заказы из таблицы Orders. Перед вами один из способов решения этой проблемы (выходные данные представлены на рис. 10.1):

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

```
==================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

```
=====Browse : ↑↓←→  PgDn PgUp —►| |◄— Home=====
```

Рис. 10.1. использование подзапроса

Чтобы оценить внешний (основной) запрос, SQL прежде всего должен оценить внутренний запрос (или подзапрос) в предложении WHERE. Эта оценка осуществляется так, как если бы запрос был единственным: просматриваются все строки таблицы Salespeople и выбираются все строки, для которых значение поля sname равно Motika, для таких строк выбираются значения поля snum.

В результате выбранной оказывается единственная строка с snum = 1004. Однако вместо простого вывода этого значения SQL подставляет его в предикат основного запроса вместо самого подзапроса, теперь предикат читается следующим образом:

```
WHERE snum = 1004
```

Затем основной запрос выполняется как обычный, и его результат точно такой же, как на рис. 10.1.

Подзапрос должен выбирать один и только один столбец, а тип данных этого столбца должен соответствовать типу значения, указанному в предикате. Часто выбранное поле и это значение имеют одно и то же имя (в данном случае, snum).

Если бы было известно значение персонального номера продавца (salesperson number) Motika, то можно было бы указать:

```
WHERE snum = 1004
```

и тем самым освободиться от подзапроса, однако использование подзапроса делает процедуру более гибкой. Вариант с подзапросом сработает и в случае изменения персонального номера продавца Motika. Простая замена имени продавца в подзапросе позволяет использовать его во множестве вариантов.

Значения, получаемые в процессе выполнения подзапросов

Весьма полезно то, что подзапрос в данном случае возвращает одно и только одно значение. Если вместо WHERE sname = 'Motika' подставить WHERE city = 'London', то в результате выполнения подзапроса получается несколько значений. Это делает невозможной оценку предиката основного запроса на предмет истинности или ложности, что приводит к оценке запроса как ошибочного.

При использовании подзапросов, основанных на операторах отношения (равенства или неравенства, рассмотренных в главе 4), нужно быть уверенным, что выходными данными подзапроса является только одна строка. Если применяется подзапрос, не генерирующий никаких значений, то это не является ошибкой, однако в настоящем случае и основной запрос не даст никаких выходных данных. Подзапросы, не генерирующие никаких выходных данных (или NULL-выход), приводят к тому, что предикат оценивается не как истинный или ложный, а как имеющий значение "неизвестно" (unknown). Предикат со значением "неизвестно" работает как и предикат со значением "ложь": основной запрос не выбирает ни од-

ной строки (информацию по поводу unknoun-предиката см. в главе 5). Попытку использовать нечто вроде

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

нельзя признать удачной.

Если в городе Barcelona есть только один продавец (salesperson) Mr.Rifkin, то подзапрос выберет единственное значение snum, и, следовательно, будет воспринят. Однако это справедливо только для текущих данных. В результате изменения состава данных таблицы Salespeople вполне реальной может стать ситуация, когда в городе (city) Barcelona появятся два продавца, тогда в результате выполнения подзапроса получим два значения, и, следовательно, запрос будет признан ошибочным.

DISTINCT с подзапросами

В некоторых случаях можно использовать DISTINCT для гарантии получения единственного значения в результате выполнения подзапроса. Предположим, нужно найти все заказы (orders), с которыми работает продавец, обслуживающий покупателя Hoffman (cnum = 2001). Вот один из вариантов решения этой задачи (выходные данные представлены на рис. 10.2):

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

Подзапрос выясняет, что значение поля snum для продавца, обслуживающего Hoffman, равно 1001; следовательно, основной запрос извлекает из таблицы Orders всех покупателей с тем же значением поля snum. Поскольку каждый покупатель обслуживается только одним продавцом, каждая строка таблицы Orders с данным значением cnum имеет то же значение поля snum. Однако, поскольку может быть любое количество таких строк, подзапрос может дать в результате множество значений snum (возможно, одинаковых) для данного cnum. Если бы подзапрос возвращал более одного значения, получилась бы ошибка в данных. Аргумент DISTINCT предотвращает такую ситуацию.

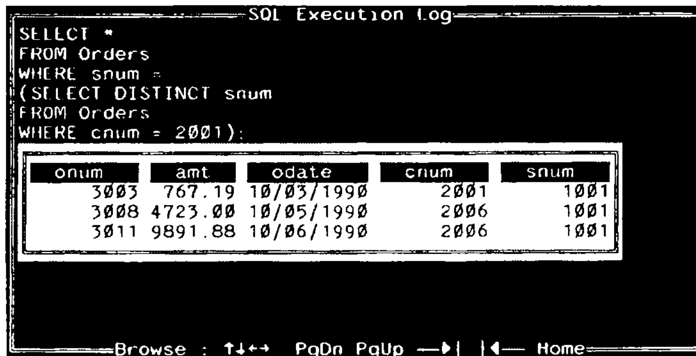


Рис. 10.2. Использование DISTINCT с целью получить единственное значение в результате выплнения подзапроса

Альтернативный способ действия — сосласться в подзапросе на таблицу Customers, а не на таблицу Orders. Поскольку cnum — первичный ключ таблицы Customer, есть гарантия, что в результате выполнения запроса будет получено единственное значение. Однако, если пользователь имеет доступ к таблице Orders, а не к таблице Customers, остается вариант, рассмотренный ранее. (SQL располагает механизмами определения того, кто и какие привилегии имеет при работе с таблицами. Эти вопросы рассмотрены в главе 22.)

Надо помнить, что приведенный в предыдущем примере прием приемлем только тогда, когда есть уверенность, что в двух различных полях содержатся одни и те же значения. Такая ситуация является скорее исключением, а не правилом для реляционных баз данных.

Предикаты с подзапросами являются неперемещаемыми

Предикаты, включающие подзапросы, используют форму *<скалярное выражение> <оператор> <подзапрос>*, а не *<подзапрос> <оператор> <скалярное выражение>* или *<подзапрос> <оператор> <подзапрос>*. Предыдущий пример нельзя записать в таком виде:

```
SELECT *
  FROM Orders
  WHERE (SELECT DISTINCT snum
    FROM Orders
```

```
        WHERE cnum = 2001)
      = snum;
```

В соответствии с соглашениями ANSI, эта запись является ошибочной, хотя некоторые программы ее понимают. Согласно ограничению ANSI запрещен также вариант команды, в котором оба значения, участвующие в сравнении, получаются в результате выполнения подзапросов.

Использование агрегатных функций в подзапросах

Одним из видов функций, которые автоматически выдают в результате единственное значение для любого количества строк, конечно, являются агрегатные функции. Любой запрос, использующий единственную агрегатную функцию без предложения GROUP BY, дает в результате единственное значение для использования его в основном предикате. Например, нужно узнать все заказы, стоимость которых превышает среднюю стоимость заказов за 4 октября 1990 г. (выходные данные представлены на рис. 10.3):

```
SELECT *
  FROM Orders
  WHERE amt >
    (SELECT AVG (amt)
        FROM Orders
```

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

onum	amt	odate	cnum	snum
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
3008	4723.00	10/05/1990	2006	1001
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

Browse : ↑↓←→ PgDn PgUp —► ◄— Home

Рис. 10.3. Выбор заявок, в которых указано количество, превышающее среднее значение для заявок, поступивших 4 октября 1990 г.

```
        WHERE odate = 10/04/1990);
```

Средняя стоимость заказов за 4 октября 1990 г. составляет 1788.98 (1713.23 + 75.75), деленное на 2, что равно 894.49. Строки, имеющие в поле amt (amount) значение, превышающее 894.49, выбираются в качестве результата запроса с вложенным подзапросом.

Сгруппированные, то есть примененные с предложением GROUP BY, агрегатные функции могут дать в результате множество значений. Поэтому их нельзя применять в подзапросах. Такие команды отвергаются в принципе, несмотря на то, что применение GROUP BY и HAVING в некоторых случаях дает единственную группу в качестве выходных данных подзапроса. Для исключения ненужных групп следует применить единственную агрегатную функцию с предложением WHERE. Например, следующий запрос, составленный с целью найти средние комиссионные (comm) для продавцов (salespeople), находящихся в Лондоне,

```
SELECT AVG (comm)
   FROM Salespeople
   GROUP BY city
   HAVING city = 'London';
```

нельзя использовать в качестве подзапроса! Это вообще не лучший способ формулировки запроса. Вот вариант, который нужен в данном случае:

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

Применение подзапросов, которые формируют множественные строки с помощью IN

Можно формулировать подзапросы, в результате выполнения которых получается любое количество строк, применяя специальный оператор IN (операторы BETWEEN, LIKE, IS NULL в подзапросах применять нельзя). IN определяет множество значений, которые тестируются на совпадение с другими значениями для определения истинности предиката. Когда IN применяется в подзапросе, SQL просто строит это множество из выходных данных подзапроса. Следовательно, можно использовать IN для выполнения подзапроса, который не работал бы с реляционным оператором, и найти все заявки (Orders) для продавцов (salespeople) из London (выходные данные представлены на рис. 10.4):

```
SELECT *
   FROM Orders
   WHERE snum IN
      (SELECT snum
```

```
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
3008	4723.00	10/05/1990	2006	1001
3011	9891.88	10/06/1990	2006	1001

```
Browse : ↑↓←→ PgDn PgUp —►| |◄— Home
```

Рис. 10.4. Использование подзапроса с IN

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

В подобной ситуации пользователю легче понять, а компьютеру проще (в конечном счете и быстрее) выполнить подзапрос, чем решать эту же задачу, применяя join:

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

Выходные данные совпадают с результатом выполнения подзапроса, но SQL просматривает все возможные комбинации строк из двух таблиц и проверяет, удовлетворяет ли каждая из них составному предикату. Проще и эффективнее извлечь из таблицы Salespeople значение поля snum для тех строк, где city = 'London', а затем выполнить поиск этих значений в таблице Orders; именно по такой схеме выполняется вариант с вложенным подзапросом. Вложенный запрос дает номера 1001 и 1004. Внешний запрос дает строки таблицы Orders, в которых значение поля snum совпадает с одним из полученных (1001 или 1004).

Эффективность варианта с использованием подзапроса зависит от *выполнения* — от особенностей реализации той программы, с которой идет работа. В любом коммерческом программном продукте есть часть программы, называемая *оптимизатором*, которая пытается найти самые эффективные способы выполнения запросов. Хороший оптимизатор преобразует версию с join в версию с подзапросом, но простого способа проверки, сделано это или нет, не существует. Поэтому при написании запросов луч-

ше использовать заведомо более эффективный вариант, чем полностью полагаться на возможности оптимизатора.

Можно применять IN и в тех ситуациях, когда есть абсолютная уверенность в получении единственного значения в результате выполнения подзапроса. IN можно также использовать там, где применим реляционный оператор сравнения. В отличие от реляционных операторов IN не приводит к ошибке выполнения команды, когда в результате выполнения подзапроса получается не одно, а несколько значений (выходных данных). В этом есть и плюсы, и минусы. Результаты выполнения подзапроса непосредственно не видны. При абсолютной уверенности в том, что в результате выполнения подзапроса будет получено только одно значение, а в действительности их получается несколько, невозможно объяснить разницу в выходных данных, полученных в результате выполнения основного запроса. Рассмотрим следующую команду, сходную с командой предыдущего примера:

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

Можно отказаться от DISTINCT, воспользовавшись IN вместо равенства. В результате получается:

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

Если допущена ошибка и один из заказов (orders) был адресован нескольким продавцам (salesperson), версия с IN выдаст все заказы для обоих продавцов. Ошибку обнаружить невозможно и отчет или решения, принятые на основе полученных данных, были бы неверными. С другой стороны, вариант с равенством, будет просто признан ошибочным, в результате чего возникнет проблема. Затем можно выполнить подзапрос сам по себе и внимательно рассмотреть его выходные данные.

Если есть уверенность в получении единственного значения в результате выполнения подзапроса, следует использовать равенство. IN подходит для тех случаев, когда запрос может генерировать одно или несколько значений, независимо от того, что именно ожидается. Предположим, нужно знать комиссионные всех продавцов (salespeople), обслуживающих покупателей (customers) в Лондоне (London):

```
SELECT comm
   FROM Salespeople
   WHERE snum IN
```

```
(SELECT snum
    FROM Customers
    WHERE city = 'London');
```

Выходные данные для запроса, представленные на рис. 10.5, показывают комиссионные продавца PEEL (snum = 1001), обслуживающего обоих лондонских покупателей. Однако, такой результат получается только на основе текущих данных. Нет (очевидной) причины, в силу которой невозможно, чтобы кого-то из лондонских покупателей обслуживал другой продавец. Следовательно, для данного запроса наиболее логичным вариантом является использование IN.

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

        comm
        0.12

Browse : ↑↓←→  PgDn PgUp —▶| |◀— Home
```

Рис. 10.5. Использование IN в подзапросе, результатом которого является единственное значение

Отказ от использования префиксов таблиц в подзапросах. Префикс имени таблицы для поля city в предыдущем примере не является необходимым, несмотря на то, что поле city есть в таблице Salespeople и в таблице Customers. SQL всегда сначала пытается найти поля в таблице (таблицах), заданной (заданных) в предложении FROM текущего запроса (подзапроса). Если поле с указанным именем здесь не найдено, то анализируется внешний запрос. В рассмотренном примере предполагается, что "city" в предложении WHERE относится к столбцу city таблицы Customers (Customers.city). Поскольку имя таблицы Customers указано в предложении FROM текущего запроса, предположение оказывается верным. От такого предположения можно избавиться, указывая имя таблицы или алиас в качестве префикса; речь об этом пойдет позднее, при обсуждении связанных подзапросов. Если есть хоть какой-то шанс допустить ошибку, то лучше всего использовать префиксы.

Подзапросы используют единственный столбец. Общая черта всех подзапросов, рассмотренных в этой главе, состоит в том, что они выбирают единственный столбец. Это существенно, так как выходные данные вложенного SELECT-предложения сравниваются с единственным значением. Из этого следует, что вариант SELECT * нельзя использовать в подзапросе. Исключением из этого правила являются подзапросы с оператором EXISTS, который рассматривается в главе 12.

Использование выражений в подзапросах. В предложении подзапроса SELECT можно использовать выражения, основанные на столбцах, а не сами столбцы. Это можно сделать, применяя операторы отношения или IN. Например, следующий запрос использует оператор отношения = (выходные данные для этого запроса представлены на рис. 10.6:

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

Запрос находит всех покупателей, для которых cnum на 1000 превосходит значение поля snum для Serres. В данном случае предполагается, что в столбце snum нет повторяющихся значений (этого можно добиться, применяя либо UNIQUE INDEX, обсуждаемый в главе 17, либо ограничение UNIQUE, обсуждаемое в главе 18); в противном случае результатом выполнения подзапроса может оказаться множество значений. За-

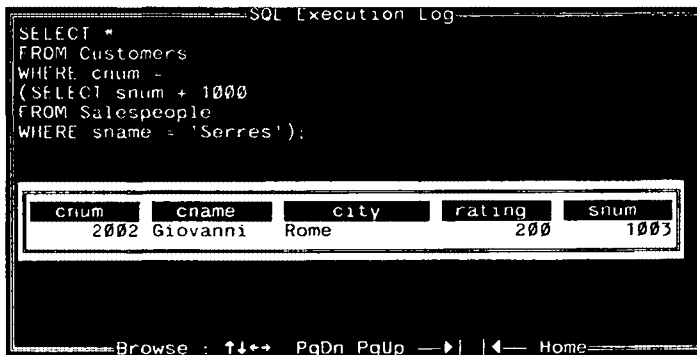


Рис. 10.6. Использование подзапроса с выражением

прос, рассмотренный в данном примере, вероятно, не очень полезен, если только не предполагается, что поля snum и cnum имеют значение, отличное от того, что просто служат первичными ключами. Но все это не проясняет сути дела.

Подзапросы с HAVING

Подзапросы можно применять также внутри предложения HAVING. В самих таких подзапросах можно использовать их собственные агрегатные функции, если они не дают множества значений, а также GROUP BY или HAVING. Например (выходные данные представлены на рис. 10.7):

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

Эта команда подсчитывает количество покупателей с рейтингом, превышающим

```
SQL Execution Log
SELECT rating, count (DISTINCT cnum)
FROM Customers
GROUP BY rating
HAVING rating >
(SELECT AVG (rating)
FROM Customers
WHERE city = 'San Jose');

rating
   300          2

Browse : ↑↓←→  PgDn PgUp —▶| |◀— Home
```

Рис. 10.7. Поиск покупателей (customers) с рейтингом (raiting), превышающим среднее значение для San Jose

среднее значение для покупателей города San Jose. Если бы были другие рейтинги, от-

личные от указанного значения 300, то каждое отдельное значение рейтинга выводилось бы вместе с указанием числа покупателей, имеющих такой рейтинг.

Итоги

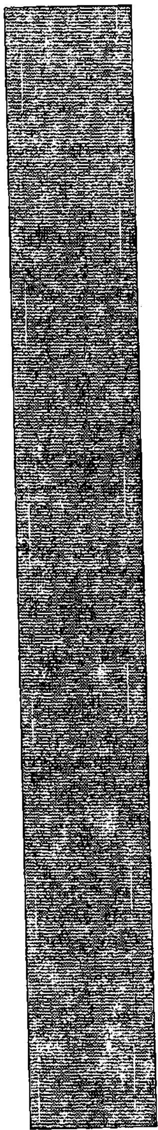
Теперь вы можете применять запросы в иерархическом порядке и знаете, насколько использование выходных данных одного запроса для управления другим облегчает выполнение многих действий, как можно использовать подзапросы с операторами отношения, а также со специальным оператором IN в предложении WHERE или HAVING внешнего запроса.

Далее будет продолжен анализ подзапросов. В следующей главе рассматриваются подзапросы, выполняемые отдельно для каждой строки, определенной во внешнем запросе. В главах 12 и 13 вводится несколько специальных операторов, действующих, как и IN, на весь подзапрос, за исключением тех случаев, когда они применяются исключительно к подзапросам.

Работаем на SQL

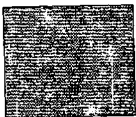
1. Запишите запрос, который использует подзапрос для получения всех заказов покупателя с именем Cisneros. Предположим, что его персональный номер неизвестен.
2. Запишите запрос, который выдает имена и рейтинги всех тех покупателей, которые сделали больше среднего числа заказов.
3. Запишите запрос, выбирающий сумму заказов каждого продавца, у которого она превышает наибольшее значение поля amount в таблице Orders.

(Ответы см. в приложении А.)



11

Связанные подзапросы



В данной главе вводятся подзапросы нового типа, которые не рассматривались ранее, — связанные подзапросы. Вы сможете использовать их в предложениях запросов WHERE и HAVING. Мы рассмотрим сходство и различия между связанными запросами и соединениями (join), а также особенности алиасов и префиксов имен таблиц (когда они необходимы и как ими пользоваться).

Как формировать связанные подзапросы

Когда в SQL используются подзапросы, во внутреннем запросе (вложенном запросе) можно ссылаться на таблицу, имя которой указано в предложении FROM внешнего запроса, тем самым формируя *связанный подзапрос (correlated subquery)*. В этом случае подзапрос выполняется повторно, по одному разу для каждой строки таблицы из основного запроса. Связанные запросы относятся, из-за сложности их оценок, к числу наиболее неясных понятий SQL. Однако, начав работать с ними, вы поймете, что они являются весьма мощным средством, так как могут выполнять очень сложные функции при достаточно компактных командах.

Вот, например, один из способов отыскать всех покупателей, сделавших заказы 3 октября 1990 года (выходные данные представлены на рис. 11.1):

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

Как работают связанные подзапросы

В данном примере "inner" и "outer" являются алиасами, о которых шла речь в главе 9. Эти имена здесь выбраны для большей ясности (inner — внутренний, outer — внешний); они относятся к значениям внутреннего и внешнего запросов соответственно. Поскольку значения в поле cnum внешнего запроса изменяются, внутренний запрос должен выполняться отдельно для каждой строки внешнего запроса. Строка внешнего запроса, для которой выполняется внутренний запрос, называется текущей строкой — *кандидатом (current candidate row)*. Процедура выполнения связанного подзапроса состоит в следующем:

1. Выбрать строку из таблицы, имя которой указано во внешнем запросе. Это текущая строка-кандидат.

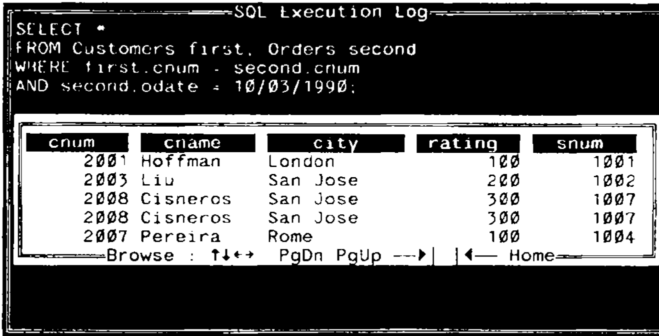


Рис. 11.1. Использование связанного подзапроса

2. Сохранить значения этой строки в алиасе, имя которого указано в предложении FROM внешнего запроса.

3. Выполнить подзапрос. Всякий раз, когда алиас, заданный для внешнего запроса, найден (в нашем случае "outer"), его значение применяется к текущей строке-кандидату. Использование в подзапросе значения из строки-кандидата внешнего запроса называется *внешней ссылкой* (*outer reference*).

4. Оценить предикат внешнего запроса на основе результатов подзапроса, выполненного на шаге 3. Это позволяет определить, будет ли строка-кандидат включена в состав выходных данных.

5. Повторять процедуру для следующей строки-кандидата таблицы до тех пор, пока не будут проверены все строки таблицы.

В предыдущем примере SQL выполняет следующую процедуру:

1. Извлекается строка Hoffman из таблицы Customers.

2. Эта строка сохраняется как текущая строка-кандидат при алиасе "outer".

3. Затем выполняется подзапрос: просматривается вся таблица Orders с целью найти строки, в которых значение поля cnum совпадает со значением outer.cnum; в данном случае это значение 2001 (значение поля cnum строки Hoffman). Затем извлекается поле odate из каждой строки таблицы Orders, для которой предикат принимает значение true (истина), и строится множество результирующих значений odate.

4. После получения множества всех значений odate, для которых cnum равно 2001, выполняется тестирование предиката основного запроса, с целью выявить, есть

ли в этом множестве 3 октября 1990 года. В случае, если такая дата обнаружена (а это так и есть), строка Hoffman выбирается для включения в состав выходных данных основного запроса.

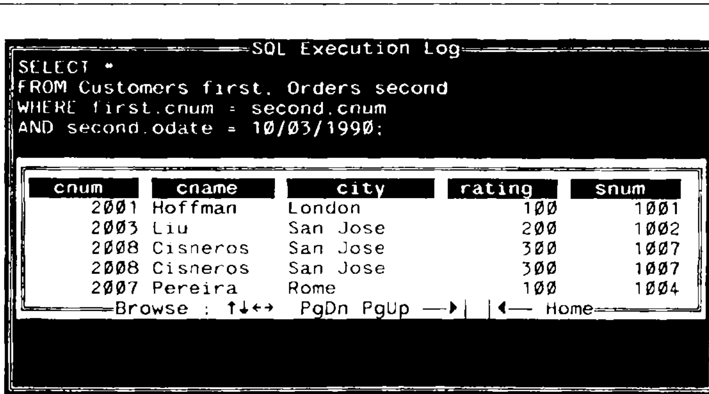
5. Процедура повторяется со строкой Giovanni, которая выбирается в качестве строки-кандидата, а затем с каждой строкой до тех пор, пока не будет проверена вся таблица Customers.

Согласно этим простым инструкциям SQL выполняет весьма сложные действия. Эту задачу можно было бы решить и с помощью операции соединения (join), например, следующим образом (выходные данные для этого запроса представлены на рис. 11.2):

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

Здесь Cisneros была выбрана дважды, по одному разу на каждый заказ, который она сделала в указанный день. Этого можно было бы избежать, используя SELECT DISTINCT вместо SELECT. Однако, такое действие не является необходимым при использовании версии с подзапросами. Оператор IN, используемый в версии с подзапросами, не делает разницы между значениями, выбираемыми в подзапросе однажды и повторно. Следовательно, в таком варианте запроса DISTINCT не является необходимым.

Предположим, нужно узнать имена и номера всех продавцов, имеющих более одного покупателя. Для этого можно воспользоваться таким запросом (выходные данные представлены на рис. 11.3):



SQL Execution Log

```
SELECT *
FROM Customers first, Orders second
WHERE first.cnum = second.cnum
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
2008	Cisneros	San Jose	300	1007
2007	Pereira	Rome	100	1004

Browse : ↑↓←→ PgDn PgUp —►| |◄— Home

Рис. 11.2. Использование соединения вместо связанного подзапроса

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

snum    sname
1001    Peel
1002    Serres

Browse : ↑↓←→  PgDn PgUp —►| |◄— Home
```

Рис. 11.3. Поиск продавца для множества покупателей

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

В этом примере предложение FROM в подзапросе не использует алиас. При отсутствии префикса в виде имени таблицы или алиаса SQL сначала предполагает, что извлекаются поля той таблицы, имя которой указано в предложении FROM текущего запроса. Если в этой таблице нет полей с указанным именем (в данном случае — snum), то SQL переходит к просмотру внешних запросов. Поэтому префиксы, содержащие имя таблицы, обычно необходимы в связанных подзапросах для того, чтобы избавится от подобных предположений. Алиасы часто применяются и для того, чтобы можно было ссылаться на одну и ту же таблицу во внутреннем и внешнем запросах без каких-либо недоразумений.

Использование связанных подзапросов для поиска ошибок

Иногда полезно выполнить запросы, предназначенные специально для поиска ошибок. В базе данных всегда может появиться ошибочная информация, и ее бывает трудно

обнаружить. Следующий запрос не генерирует выходных данных. Он проверяет таблицу Orders с целью установить, соответствует ли соотношение полей snum и cnum каждой ее строки соотношению этих же полей в таблице Customers, и осуществляет вывод любой строки, для которой такое соответствие не обнаружено. Другими словами, он позволяет контролировать корректность связей продавцов с обслуживаемыми ими покупателями (предполагается, что cnum, как первичный ключ таблицы Customers, не имеет повторяющихся значений в этой таблице).

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

Можно выявить ряд ошибок такого рода, используя механизм ссылочной целостности (обсуждается в главе 19). Однако такой механизм не всегда доступен и не во всех случаях приемлем, поэтому и полезны подзапросы, ориентированные на поиск ошибок.

Связывание таблицы со своей копией

Можно использовать подзапросы, основанные на той же таблице, что и основной запрос. Это позволяет извлекать определенные сложные виды производной информации. Например, можно найти все заказы, величина которых превышает среднюю величину заказа для данного покупателя (выходные данные представлены на рис. 11.4):

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

В маленькой таблице, рассматриваемой в качестве примера, где большинство покупателей имеют только один заказ, большинство значений совпадает со средним и, следовательно, не извлекается. Можно ввести команду по-другому (выходные данные представлены на рис. 11.5):

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

```
WHERE inner.cnum = outer.cnum);
```

Разница заключается в том, что здесь оператор отношения главного предиката включает значения, равные средней величине заказов (это обычно означает, что данные заказы являются единственными у данных покупателей).

```
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	1098.16	10/03/1990	2008	1007
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

```
Browse : ↑↓←→ PgDn PgUp —►| |◄— Home
```

Рис. 11.4. Связывание таблицы с ее копией

```
SQL Execution Log
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	1098.16	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

```
Browse : ↑↓←→ PgDn PgUp —►| |◄— Home
```

Рис. 11.5. Выбор заказов, величина которых превышает среднее значение для их покупателей или совпадает с ним

Связанные подзапросы в HAVING

В предложении HAVING могут использоваться и связанные подзапросы. В этом случае необходимо ограничить внешние ссылки теми элементами, которые могут непосредственно применяться в самом предложении HAVING. Как стало известно из главы 6, предложение HAVING может использовать только функции агрегирования из предложения SELECT или поля из предложения GROUP BY. Это и есть единственные внешние ссылки, которые можно делать, 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 выходных данных внешнего запроса, а не для каждой отдельной строки.

Предположим, необходимо суммировать значения поля amounts (amt) таблицы Orders, сгруппировав их по датам и исключив те дни, когда сумма не превышает максимальное значение, по крайней мере на 2000.00:

```
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 максимальное (MAX) значение для всех строк с одной и той же датой, совпадающей с датой, для которой сформирована очередная группа основного запроса. Это должно быть сделано, как показано в данном примере, с помощью предложения WHERE. В самом подзапросе не должно быть предложений GROUP BY и HAVING.

Связанные подзапросы и соединения

Связанные подзапросы имеют близкое сходство с соединениями, так как оба варианта включают сравнение каждой строки таблицы с каждой строкой другой (или алиасом той же самой) таблицы. Сходство заключается и в том, что многие операции, которые можно выполнить с помощью одного варианта, выполнимы и с помощью другого.

Различие в их применении заключается в ранее упомянутой необходимости использовать иногда DISTINCT в команде соединения (join), тогда как этого не требуется в подзапросах. Есть также ряд вещей, которые можно сделать только с помощью одного из этих вариантов. Подзапросы, например, могут использовать агрегатные функции в предикате, позволяя выполнить операции, подобные рассмотренной в предыдущем примере, когда извлекались заказы, величина которых превышала среднее

значение для данного покупателя. С другой стороны, соединения позволяют получать строки из двух таблиц, участвующих в сравнении, тогда как выходные данные подзапросов могут использоваться только в предикатах внешних запросов. Согласно простому правилу, лучше, вероятно, использовать ту форму запроса, которая кажется интуитивно более понятной, однако предпочтительно знать оба варианта на случай, если один из них окажется неприемлемым.

Итоги

В главе было рассмотрено одно из сложных понятий SQL — связанные подзапросы. Мы выяснили, как они соотносятся с операцией соединения, как их можно использовать с функциями агрегирования и с предложением HAVING. Вы ознакомились со всеми типами подзапросов.

Следующий шаг — введение некоторых специальных операторов, использующих, как и оператор IN, подзапросы в качестве аргументов, но в отличие от IN применяемых *только* с подзапросами. Первый из них, EXISTS, рассмотрен в главе 12.

Работаем на SQL

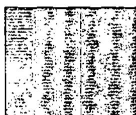
1. Запишите команду SELECT, использующую связанные подзапросы и выбирающую имена и номера всех покупателей, рейтинг которых совпадает с максимальным значением рейтинга для их города.
2. Запишите два запроса, которые выбирают (по имени и номеру) всех продавцов, проживающих в городах, где у них нет покупателей. Один запрос должен использовать операцию соединения (join), а второй — связанные подзапросы. Какое из решений является более элегантным?

   (Подсказка: один из возможных вариантов решения этой задачи — найти всех покупателей, не обслуживаемых данным продавцом, и посмотреть, не находятся ли они в одном городе.)

(Ответы см. в приложении А.)

12

Использование оператора EXISTS



В этой главе речь пойдет о специальных операторах, всегда использующих подзапросы в качестве аргументов, в частности, об операторе EXISTS.

Этот оператор применяется для образования предиката, фиксирующего, будет ли подзапрос генерировать выходные данные. Вы научитесь использовать оператор EXISTS в обычных и связанных подзапросах. Будут рассмотрены специальные случаи его употребления для агрегатов, NULL-значений и булевых значений. Вы сможете усовершенствовать навыки работы с подзапросами, рассматривая более сложные способы их применения.

Как работает оператор EXISTS?

EXISTS — оператор, генерирующий значение "истина" или "ложь", другими словами, булево выражение. Это значит, что его можно применять отдельно в предикате или комбинировать с другими булевыми выражениями с помощью операторов AND, OR и NOT. Используя подзапрос в качестве аргумента, этот оператор оценивает его как истинный, если он генерирует выходные данные, а в противном случае как ложный. В отличие от прочих операторов и предикатов, он не может принимать значение unknown. Например, нужно извлечь данные из таблицы Customers в том случае, если один (или более) покупатель из нее находится в San Jose (выходные данные для запроса представлены на рис. 12.1):

```
SELECT cnum, cname, city
    FROM Customers
    WHERE EXISTS
```

Рис. 12.1. Использование оператора EXISTS

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

Внутренний запрос выбрал все данные обо всех покупателях из San Jose. Оператор EXISTS внешнего предиката отметил, что подзапрос генерирует выходные данные, и поскольку выражение EXISTS является единственным в этом предикате, он принимает значение "истинно". Здесь подзапрос (не являющийся связанным) выполняется только один раз для всего внешнего запроса и, следовательно, имеет единственное значение для всех случаев. Поскольку EXISTS в данном примере делает предикат истинным или ложным для всех строк сразу, применять его в таком стиле для извлечения специфической информации не стоит.

Выбор столбцов с помощью EXISTS

В приведенном примере EXISTS выбирает один столбец, в отличие от с звездочки, выбирающей все столбцы, что отличается от рассмотренных ранее подзапросов, где извлекался единственный столбец. Однако несущественно, сколько столбцов извлекает EXISTS, поскольку он вообще не применяет полученных значений, а лишь фиксирует наличие выходных данных подзапроса.

Использование EXISTS со связанными подзапросами

При применении связанных подзапросов предложение EXISTS, как и другие предикатные операторы, оценивается отдельно для каждой строки таблицы, на которую есть ссылка во внешнем запросе. Это позволяет использовать EXISTS как правильный предикат, генерирующий различные ответы для каждой строки таблицы, на которую есть ссылка в основном запросе. Следовательно, при таком способе применения EXIST информация из внутреннего запроса сохраняется, если непосредственно не выводится. Например, можно сделать запрос на поиск тех продавцов, которые имеют нескольких покупателей (выходные данные такого запроса представлены на рис. 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 snum
FROM Customers outer
WHERE EXISTS
(SELECT *
FROM Customers inner
WHERE inner.snum = outer.snum
AND inner.cnum <> outer.cnum);

    snum
    1001
    1002

Browse : ↑↓←→  PgDn PgUp —►| |◄— Home
```

Рис. 12.2. Использование оператора EXISTS со связанными подзапросами

Для каждой строки-кандидата внешнего запроса (представляющей рассматриваемого в настоящий момент покупателя), внутренний запрос находит строки, имеющие соответствующее значение snum (имеет того же продавца), но не значение cnum (соответствует другому покупателю). Если строка, удовлетворяющая подобному критерию, найдена во внутреннем запросе, это означает, что различные покупатели обслуживаются данным продавцом (т.е. продавцом, обслуживающим покупателя, указанного в строке-кандидате внешнего запроса). Следовательно, предикат EXISTS истинен для текущей строки, и номер поля продавца (snum) из таблицы внешнего запроса включается в состав выходных данных. Если бы не был задан DISTINCT, каждый из таких продавцов выбирался бы один раз для каждого покупателя, которого он обслуживает.

Комбинирование EXISTS и соединений

Иногда кроме номера требуется получить о каждом продавце больше информации. Это можно сделать соединением таблиц Customers и Salespeople (выходные данные представлены на рис. 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;
```

Внутренний запрос тот же, что и в предыдущем примере, изменены только алиасы. Внешний запрос является соединением таблиц Salespeople и Customers. Новое предложение основного предиката (AND first.snum = second.snum) оценивается на том же уровне, что и предложение EXISTS. Это функциональный предикат самого соединения, сравнивающий две таблицы из внешнего запроса в терминах общего для них поля snum. Поскольку используется булев оператор AND, оба условия, сформулированные в предикатах основного запроса, должны быть истинными для истинности этого предиката. Следовательно, результаты подзапроса действуют в случаях, когда вторая часть запроса истинна и соединение выполняется. Комбинирование соединений и подзапросов указанным способом является весьма эффективным методом обработки данных.

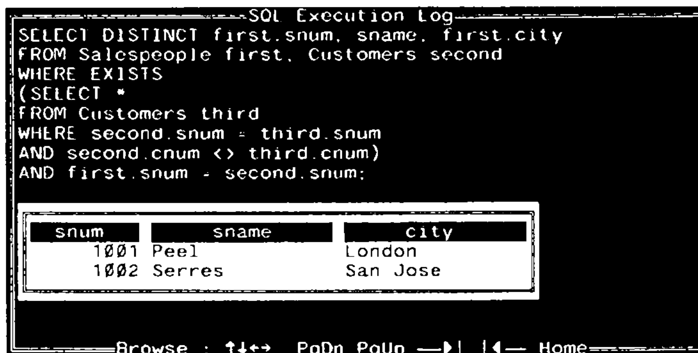


Рис. 12.3. Комбинирование EXISTS с JOIN (соединением)

Использование NOT EXISTS

Из предыдущего примера ясно, что EXISTS можно комбинировать с булевыми операторами. С EXISTS легче всего применять — и чаще всего применяется — оператор NOT. Один из способов поиска всех продавцов, имеющих только одного покупателя, — это поставить NOT перед EXISTS в предыдущем примере (см. рисунок 12.2) (выходные данные для запроса представлены на рис. 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);
```

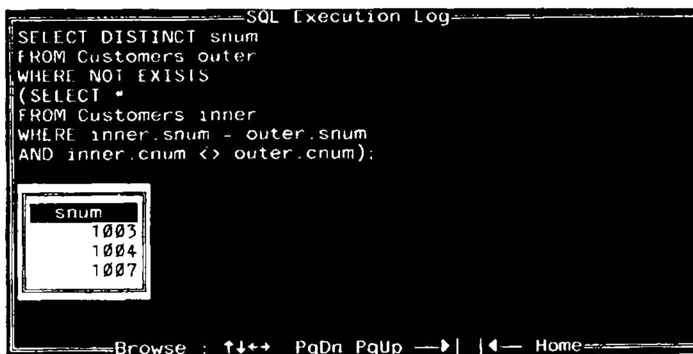


Рис. 12.4. Использование EXISTS с NOT

EXISTS и агрегаты

EXISTS не может использовать агрегатные функции в своем подзапросе. Это естественно. Если функция агрегирования находит строки для работы с ними, то EXISTS принимает значение "истина", и ему безразлично реальное значение функции; если функция не находит никаких строк, то значение EXISTS — "ложь". Попытка использовать функции агрегирования с EXISTS подобным образом свидетельствует о том, что проблема не была правильно понята.

Подзапрос для предиката EXISTS тоже может иметь в своем составе один или несколько подзапросов любого типа. Эти подзапросы, как и любые другие, входящие в их состав, могут использовать агрегатные функции, если только не существует каких-либо иных причин, по которым это невозможно. Пример, иллюстрирующий такую ситуацию, приведен в следующем разделе.

В любом случае можно получить тот же результат проще — выбрать поле, которое используется в агрегатной функции, вместо применения самой функции. Другими словами, предикат EXISTS (SELECT COUNT (DISTINCT sname) FROM salespeople) эквивалентен предикату EXISTS (SELECT sname FROM Salespeople), причем первый из них тоже допустим.

Примеры более сложных подзапросов

Возможности применения запросов весьма разнообразны. В одном запросе можно разместить один или несколько запросов, и даже один внутри другого. Перед вами пример, в котором извлекаются строки для всех продавцов, имеющих покупателей, сделавших более одного заказа. Решение этой проблемы представляет интерес с точки зрения демонстрации преимуществ логики SQL. Нужную информацию можно получить путем связывания всех трех рассматриваемых нами в примерах таблиц:

```
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));
```

Выходные данные представлены на рис. 12.5.

```
=============== SQL Execution Log ===============
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));
```

snum	sname	city	comm
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1007	Rifkin	Barcelona	0.15

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 12.5. Использование EXISTS в сложном подзапросе

Можно рассмотреть оценку данного запроса следующим образом. Взять каждую строку таблицы Salesperson в качестве строки-кандидата (внешний запрос) и выполнить подзапросы. Для каждой строки-кандидата из внешнего запроса взять каждую строку из таблицы Customers (средний запрос). Если текущая строка покупателя (customer) не соответствует текущей строке продавца (т.е. если first.snum <> second.snum), то предикат среднего запроса ложен. Как только в среднем запросе найдется покупатель, который соответствует продавцу во внешнем запросе, нужно перейти к самому внутреннему запросу, чтобы определить, истинен ли предикат среднего запроса. Самый внутренний запрос выполняет подсчет заказав (orders) для текущего покупателя (из среднего запроса). Если это число превышает 1, то предикат среднего запроса принимает значение "истина", и строка выбирается. Это делает предикат EXISTS внешнего запроса истинным для текущей строки продавца, что означает, что, по крайней мере, один из покупателей данного продавца имеет более одного заказа.

Этот запрос намного сложнее тех, с которыми мы обычно встречаемся в жизни. Основное назначение подобных примеров — показать те дополнительные средства, которые вам могут понадобиться После работы в таких сложных ситуациях простые запросы, используемые наиболее часто в SQL, покажутся элементарными.

Кроме того, этот запрос связывает три различные таблицы и выдает информацию, которую было бы трудно получить более простым способом. Возможно, на практике такая информацию вам будет нужна регулярно, например, если продавец, обеспечивший в течение недели несколько заказов одного покупателя, получает в результате вознаграждение. В этом случае понадобится применение множества команд для изменяющихся данных (для этого удобно использовать представления (view)).

Итоги

Несмотря на свою кажущуюся простоту, EXISTS относится к наиболее сложным, но весьма гибким и мощным операторам SQL. В этой главе вы познакомились с многочисленными возможностями этого оператора и существенно расширили свои знания в области логики сложных подзапросов.

Следующий шаг — рассмотрение трех других специальных операторов, использующих подзапросы в качестве аргументов: ANY, ALL, SOME. Из главы 13 вы узнаете, что они могут служить альтернативой уже известным операторам, а во многих случаях даже более полезны.

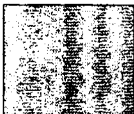
Работаем на SQL

1. Запишите запрос с EXISTS для того, чтобы извлечь всех продавцов, имеющих покупателей с рейтингом, превышающим 300.
2. Как решить эту же проблему, применяя соединение?
3. Запишите запрос с EXISTS, выбирающий всех проживающих в одном городе продавцов, а также покупателей, которых эти продавцы не обслуживают.
4. Запишите запрос, извлекающий из таблицы Customers покупателя, назначенного каждому продавцу, который уже имеет по крайней мере одного покупателя (покупатель, который выбирается, не учитывается) с заявками в таблице Orders (Подсказка: по структуре запрос сходен с трехуровневым подзапросом, рассмотренным в примере).

(Ответы см. в приложении А)

13

Использование операторов ANY, ALL и SOME



В данной главе вы познакомитесь еще с тремя специальными операторами, ориентированными на подзапросы. (Реально их два, поскольку ANY и SOME совпадают по назначению и использованию.) Этими операторами исчерпываются возможные типы предикатов SQL, используемых в подзапросах. Мы рассмотрим множество способов формулирования одного и того же запроса, применяя различные типы предикатов в подзапросах; вы узнаете преимущества и недостатки каждого подхода.

ANY, ALL и SOME так же, как и EXISTS, используют в качестве аргументов подзапросы; однако, от EXISTS они отличаются тем, что применяются в конъюнкции с операторами отношения. В этом плане они сходны с оператором IN, т.е. берут все значения, полученные в подзапросе и рассматривают их как единое целое. Однако, в отличие от IN, их можно применять только с подзапросами.

Специальный оператор ANY или SOME

Начнем с операторов ANY или SOME. Независимо от применения, они выполняются абсолютно одинаково и являются взаимозаменяемыми. Различие в терминологии отражает попытку ориентации на интуитивные представления пользователя. Однако, такой подход проблематичен, поскольку интуитивная интерпретация этих операторов может привести к ошибке.

Представляем новый способ найти продавцов с покупателями, находящимися в одних городах (выходные данные для этого запроса представлены на рис. 13.1):

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

snum	sname	city	comm
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 13.1. Использование оператора ANY

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

Оператор ANY берет все значения поля sity в таблице Customers, полученные в подзапросе, и оценивает результат как истину, если какое-либо (ANY) значение совпадает со значением поля city из текущей строки внешнего запроса. Это означает, что подзапрос должен выбирать значения того же типа, которые сравнились в основном предикате. В этом отношении ANY отличается от EXISTS, который просто определяет реально не используемые результаты.

Использование IN или EXISTS вместо ANY

Чтобы сформулировать предыдущий запрос, можно воспользоваться оператором IN:

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

Запрос генерирует выходные данные, представленные на рис. 13.2.

```
=SQL Execution Log=
SELECT *
FROM Salespeople
WHERE city IN
(SELECT city
FROM Customers);
```

snum	sname	city	comm
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11

Browse : ↑↓←→ PgDn PgUp —▸| |◂— Home

Рис. 13.2. Использование IN как альтернативы ANY

Оператор ANY может использовать другие операторы отношения кроме равенства и, следовательно, выполняет сравнения, отличные от сравнений в IN. Например, можно найти всех продавцов, имеющих покупателей, имена которых следуют в алфавитном порядке за именем продавца (выходные данные представлены на рис. 13.3):

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

Все строки, которые были выбраны, сохраняются для Serres и Rifkin, поскольку они не имеют покупателей, имена которых следуют за их именами в алфавитном порядке. Это эквивалентно следующему запросу с EXISTS, выходные данные для которого представлены на рис. 13.4:

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

Любой запрос, сформулированный с ANY (или с ALL), можно сформулировать и с EXISTS, хотя обратное утверждение неверно. Строго говоря, версии с EXISTS не совсем идентичны версиям с ANY или ALL. Различие заключается в обработке NULL-значений (эта проблема будет рассмотрена позже в этой главе). Можно обойтись без

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

snum	sname	city	comm
1001	Peel	London	0.12
1004	Motika	London	0.11
1003	Axelrod	New York	0.10

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 13.3. Использование ANY с неравенством

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

snum	sname	city	comm
1001	Peel	London	0.12
1004	Motika	London	0.11
1003	Axelrod	New York	0.10

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 13.4. Использование EXISTS как альтернативы ANY

ANY и ALL, если умело применять EXISTS (и IS NULL). Многие пользователи считают, что использовать ANY и ALL проще, чем EXISTS, который требует связанных подзапросов. В зависимости от практической реализации, ANY и ALL могут, по крайней мере теоретически, быть более эффективными по сравнению с EXISTS. Подзапрос с ANY или ALL можно выполнять один раз для каждой строки основного запроса и предоставлять выходные данные для определения предиката. С другой стороны, EXISTS использует связанный подзапрос, который требует повторного выполнения всего подзапроса для каждой строки основного запроса. SQL пытается найти наиболее эффективный способ выполнения любой команды, поэтому он может пытаться преобразовать менее действенную формулировку запроса в более эффективную (но нельзя рассчитывать на то, что при этом будет найдена наиболее эффективная формулировка).

Основная причина предложения формулировки с EXISTS, как альтернативы ANY и ALL, состоит в противоречии ANY и ALL интуиции, связанной со спецификой применения этих терминов в английском языке. Если вы научитесь применять разные способы формулировки данного запроса, у вас появится возможность разработать множество процедур для сложных или неясных случаев.

Неоднозначности при использовании ANY

ANY не является полностью интуитивно очевидным. Если запрос формулируется для выбора покупателей, имеющих рейтинг, превышающий рейтинг любого покупателя в Rome, можно получить выходные данные, которые отличаются от ожидаемых (как показано на рис. 13.5):

```
SELECT *
```

```
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

```
Browse : ↑↓←→ PgDn PgUp —►| |◄— Home
```

Рис. 13.5. Больше, чем ANY в интерпретации SQL

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

В английском языке выражение "рейтинг больше, чем *любой* (*any*) другой (где поле city равно Rome)", обычно интерпретируется так: значение данного рейтинга должно быть выше, чем значение рейтинга для *каждого* (*every*) случая, когда значение поля city равно Rome. Однако в SQL оператор ANY интерпретируется иначе. ANY оценивается как истина, если подзапрос находит *любое* (*any*) значение (любые значения), удовлетворяющее (удовлетворяющие) условию.

Если бы ANY интерпретировалось как обычное слово английского языка, покупатели с рейтингом 300 оказались бы выше Giovanni, находящегося в Rome и имеющего рейтинг 200. Однако, подзапрос с ANY находит также и Periera из Rome с рейтингом 100. Все покупатели с рейтингом 200 были выбраны (поскольку их рейтинги превышают 100) несмотря на то, что в Rome был другой покупатель (Giovanni), рейтинг которого выше 200 (то, что один из выбранных покупателей тоже находится в Rome, здесь значения не имеет). Они были выбраны, поскольку подзапрос сгенерировал значение, сделавшее предикат истинным для этих строк.

Другой пример: предположим, нужно выбрать все заказы, величина которых превосходит величину по крайней мере одного из заказов, сделанных 6 октября 1990 года:

```
SELECT *
  FROM Orders
```

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

Выходные данные для запроса представлены на рис. 13.6.

Даже если наибольшая величина заказов, представленных в таблице (9891.88), приходится на 6 октября, предшествующие строки имеют значения, превышающие значение поля amounts в другой строке таблицы за 6 октября 1990 года — 1309.95.

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 13.6. Выбор записей, значение поля amt которых превышает ANY на 6 октября

Если бы использовался оператор отношения >= вместо просто >, эта строка тоже была бы выбрана, так как она равна сама себе.

Можно использовать ANY с другими SQL-средствами, например, с соединениями. Этот запрос отыскивает все заказы, величина которых меньше величины заказа любого покупателя в San Jose (выходные данные представлены на рис. 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 минимальное значение в таблице указано в заказе из San Jose, имеется другое, превышающее его, поэтому почти все строки были выбраны. Легко запом-

```
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	767.19	10/03/1990	2001	1001
3002	1900.10	10/03/1990	2007	1004
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

```
Browse : ↑↓←→ PgDn PgUp ──▶| |◀── Home
```

Рис. 13.7. Использование ANY с JOIN (соединением)

```
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
3001	18.69	10/03/1990	2008	1007
3003	767.19	10/03/1990	2001	1001
3002	1900.10	10/03/1990	2007	1004
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

```
Browse : ↑↓←→ PgDn PgUp ──▶| |◀── Home
```

Рис. 13.8. Использование функции агрегатирования вместо ANY

нить, что < ANY означает "меньше, чем наибольшее выбранное значение", а > ANY означает "больше, чем наименьшее выбранное значение". Фактически, такую команду можно сформулировать следующим образом (выходные данные представлены на рис. 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');
```

Специальный оператор ALL

Предикат с ALL принимает значение "истина", если *каждое* (*every*) значение, выбранное в процессе выполнения подзапроса, удовлетворяет условию, заданному в предикате внешнего запроса. Если бы было нужно, чтобы в предыдущем примере в состав выходных данных включались только те покупатели, рейтинг которых превышает рейтинг каждого покупателя в Rome, то необходимо было бы ввести следующую команду для получения результата, представленного на рис. 13.9:

```
SELECT *
   FROM Customers
```

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 13.9. Использование оператора ALL

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

Это предложение проверяет значения рейтинга для всех покупателей в Риме, а затем находит тех покупателей, рейтинг которых превышает рейтинг каждого покупателя из Рима. Наибольший рейтинг в Риме имеет Giovanni, его значение составляет 200. Следовательно, в состав выходных данных включаются только те покупатели, рейтинг которых превышает 200.

EXISTS можно использовать, как в случае с ANY, для получения альтернативной формулировки того же запроса (выходные данные представлены на рис. 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

```
=====Browse : ↑↓←→ PgDn PgUp —►| |◄— Home=====
```

Рис. 13.10. Использование EXISTS как альтернативы ALL

Равенства и неравенства

ALL, как правило, используется с неравенствами, а не с равенствами, поскольку значение "равно всем" (equal to all), которое должно получиться в результате выполнения подзапроса, может быть получено в случае, если все результаты идентичны. Следующий запрос выглядит так:

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

Команда задана корректно, но в настоящем примере выходные данные не будут получены. Единственный случай, когда этот запрос продуцирует выходные данные, — значения рейтинга всех покупателей из San Jose одинаковы. Тогда запрос можно сформулировать иначе:

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

Основное различие заключается в признании последней команды ошибочной, если в результате выполнения подзапроса будет получено множество значений; в этом случае вариант с ALL просто не генерирует никаких выходных данных. Поскольку в действительности база данных постоянно изменяется, неразумно заведомо делать какие-либо предположения относительно ее содержания.

Тем не менее ALL можно эффективно использовать с неравенствами, то есть с оператором <>. Однако, фраза "значение неравно всем результатам подзапроса" выражается не так, как в привычном английском языке. Если подзапрос генерирует множество различных значений, как чаще всего это и бывает, то никакое одно значение не может быть равно всем значениям в обычном смысле. В SQL <> ALL реально означает "не равно ни одному" из результатов подзапроса. Другими словами, предикат имеет значение истина, если значение не принадлежит множеству результатов подзапроса. Следовательно, предыдущий запрос можно сформулировать, например, так (выходные данные для запроса представлены на рис. 13.11):

```
SELECT *
   FROM Customers
   WHERE rating <> ALL
      (SELECT rating
```

```
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

```
Browse : ↑↓←→ PgDn PgUp →| |← Home
```

***Рис. 13.11**.* Использование ALL с символом <>

```
            FROM Customers
            WHERE city = 'San Jose');
```

Этот подзапрос выбирает все рейтинги, для которых в поле city указано значение San Jose. В результате получается множество из двух значений: 200 (для Liu) и 300 (для Cisneros). Основной запрос затем выбирает все строки, в которых значение поля rating отличается от этих значений, т.е. все строки со значением рейтинга 100. Этот же запрос можно сформулировать с NOT IN:

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

Можно также использовать ANY:

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

Выходные данные одинаковы для трех последних запросов.

Непосредственная поддержка ANY и ALL

В языке SQL выражение "значение больше (или меньше), чем любое (ANY) из множества значений" эквивалентно выражению "значение больше (или меньше) какого-либо из множества значений". Соответственно, "значение неравно всему множеству значений (not equal ALL)" означает "в этом множестве нет значений, с которым оно совпадает".

Функционирование ANY, ALL и EXISTS при потере данных или с неизвестными данными

Ранее упоминалось, что существуют некоторые различия между EXISTS и введенными в этой главе операторами, касающиеся обработки NULL-значений. ANY и ALL также отличаются друг от друга по своей реакции, когда в результате выполнения подзапроса не получено значений, которые могут использоваться в сравнении. Если эти отличия не принять во внимание, они могут привести к неожиданным результатам.

Когда подзапрос возвращает значение EMPTY

Важное различие между ALL и ANY заключается в их реакции на ситуацию, когда подзапрос не генерирует никакого значения. Когда правильный подзапрос не генерирует выходных данных, ALL автоматически принимает значение "истина", а ANY — "ложь".

Это означает, что следующий запрос:

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

не генерирует выходных данных, тогда как запрос:

```
SELECT *
   FROM Customers
   WHERE rating > ALL
         (SELECT rating
```

```
        FROM Customers
        WHERE city = 'Boston');
```

полностью воспроизведет таблицу Customers. Поскольку в городе Boston нет никаких покупателей, все эти сравнения не имеют большого значения.

ANY и ALL вместо EXISTS с NULL-значениями

NULL-значения также создают некоторые проблемы для рассматриваемых операторов. Когда SQL сравнивает два значения, одно из которых — NULL-значение, результат принимает значение unknown (см. главу 5). Unknown-предикат, также как и false-предикат, создает ситуацию, когда строка не включается в состав выходных данных, но результат различен для различных типов запросов, в зависимости от использования в них ALL или ANY вместо EXISTS. Рассмотрим приведенные ранее примеры:

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

и:

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

Оба эти запроса ведут себя совершенно одинаково. Предположим, что в таблице Customers есть строка с NULL-значением в столбце rating:

CNUM	CNAME	CITY	RATING	SNUM
2003	Liu	San Jose	NULL	1002

В версии с ANY, когда выбирается поле rating для Mr. Liu в основном запросе, из-за NULL-значения предикат принимает значение unknown, а строка с Liu не включается в состав выходных данных. Однако, когда версия NOT EXISTS выбирает эту строку в основном запросе, NULL-значение используется в предикате подзапроса, присваивая ему всякий раз значение unknown. Это значит, что в результате выполнения подза-

проса не будет получено ни одного значения и EXISTS примет значение "ложь". Это, в свою очередь, сделает NOT EXISTS истинным. Следовательно, строка с Mr. Liu включается в состав выходных данных. Противоречие вытекает из того, что в отличие от других типов предикатов, значение EXISTS — всегда "истина" или "ложь", а никогда не unknown.

Это и является основанием для использования ANY. NULL-значение не превосходит любого реального значения. Более того, результат будет тот же, и необходимо искать меньшее значение.

Использование COUNT вместо EXISTS

Было отмечено, что ANY и ALL могут быть (приближенно) заменены на EXISTS, тогда как обратное неверно. Верно и то, что подзапросы с EXISTS и NOT EXISTS могут быть заменены теми же самыми подзапросами с COUNT(*) в предложении подзапроса SELECT. Если в состав выходных данных входит более чем 0 строк, то это эквивалентно ситуации EXISTS; в противном случае это то же самое, что NOT EXISTS. Рассмотрим пример (выходные данные для запроса представлены на рис. 13.12):

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

Его можно представить и так:

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

Выходные данные для этого запроса изображены на рис. 13.13.

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 13.12. Использование EXISTS со связанным подзапросом

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 13.13. Использование COUNT вместо EXISTS

Итоги

Эта глава содержит большой объем информации. Подзапросы — тема непростая, поэтому по мере изложения материала обсуждались и возможные варианты, и неоднозначности. Вы узнали о разных методах работы с ошибками и NULL-значениями. У вас есть теперь несколько способов решения проблемы и возможность выбора оптимального варианта.

После разбора наиболее важного и сложного аспекта SQL, оставшийся материал сравнительно прост для понимания. Следующая глава тоже посвящена запросам. Вы научитесь комбинировать выходные данные для любого количества запросов в единственном теле с целью формирования объединения множества запросов с использованием предложения UNION.

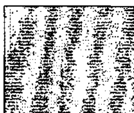
Работаем на SQL

1. Запишите запрос, выбирающий всех покупателей, рейтинг которых равен или превосходит ANY (в смысле SQL) Serres.
2. Какие выходные данные генерирует эта команда?
3. Запишите запрос, использующий ANY или ALL, который будет находить всех продавцов, не имеющих покупателей в их городе.
4. Запишите запрос, выбирающий всех покупателей, значение поля amount которых превышает любое значение (в обычном смысле) для покупателей Лондона.
5. Запишите этот же запрос с использованием MAX.

(Ответы см. в приложении А.)

14

Использование предложения UNION



В предшествующих главах обсуждались различные варианты запросов с расположением "один внутри другого". Существует другой способ комбинирования множества запросов — их объединение. В этой главе объясняются предложения UNION в SQL. *Объединения* (*unions*) отличаются от подзапросов тем, что любой из двух (или большего числа) запросов не может управлять другим запросом. В объединении все запросы выполняются независимо, но их выходные данные затем объединяются.

Объединение множества запросов в один

Можно задать множество запросов одновременно и комбинировать их выходные данные с использованием предложения UNION. UNION объединяет выходные данные двух или более SQL-запросов в единое множество строк и столбцов. Для того чтобы получить сведения обо всех продавцах (salespeople) и покупателях (customers) Лондона в виде выходных данных одного запроса, следует ввести:

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

   UNION

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

(выходные данные представлены на рис. 14.1).

Столбцы, выбранные с помощью двух команд, представлены в выходных данных так, как если бы они выбирались с помощью одного запроса. Заголовки столбцов опущены, поскольку в результат объединения входят столбцы из нескольких разных таблиц. Таким образом столбцы в выходных данных не поименованы.

Только последний запрос заканчивается точкой с запятой. Отсутствие этого знака дает возможность SQL распознать, что следует еще один запрос.

Когда можно выполнить объединение запросов?

Для того, чтобы два или более запроса можно было объединить (выполнить команду UNION), их столбцы, входящие в состав выходных данных, должны быть *совместимы по объединению* (*union compatible*). Это значит, что в каждом из запросов может быть указано одинаковое количество столбцов в таком порядке: первый, второй, тре-

```
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 Clemens

Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home
```

Рис. 14.1. Формирование объединения двух запросов

тий и т.д., — причем, первые столбцы каждого из запросов являются сравнимыми, вторые столбцы каждого из них — также сравнимы и т.д. по всем столбцам, включаемым в состав выходных данных. Значение термина "столбцы сравнимы" может меняться. ANSI определяет его очень просто: числовые поля должны иметь полностью совпадающие тип и размер. (В приложении В детально описаны числовые типы ANSI.) Символьные поля должны иметь точно совпадающее количество символов (это означает, что одинаковое количество выделено, но вовсе не обязательно заполнено).

Некоторые программные продукты SQL используют более гибкие определения. Например, нестандартные с точки зрения ANSI типы, такие как DATE и BINARY, обычно сравнимы со столбцами этих же нестандартных типов. Длина столбцов тоже является проблемой. Многие программные продукты не требуют совпадения длин, но такие столбцы нельзя использовать в UNION. С другой стороны, некоторые продукты (и ANSI) требуют, чтобы длины символьных полей точно совпадали. По этим причинам всегда следует ознакомиться с документацией по конкретному программному продукту.

Другое ограничение на сравнимость состоит в том, что если NULL-значения запрещены для любого столбца в объединении, то они должны быть запрещены для всех соответствующих столбцов в других запросах объединения. NULL-значения запрещаются с помощью ограничения NOT NULL, о котором идет речь в главе 18. Нельзя использовать UNION в подзапросах также, как и функции агрегирования в предложениях SELECT запросов в объединении (многие программные продукты смягчают эти ограничения).

UNION и устранение дублирования

UNION автоматически исключает из выходных данных дублирующиеся строки. Не возбраняется, но и не имеет особого смысла применение в SQL оператора DISTINCT в отдельных запросах для исключения повторяющихся значений. Примером может служить следующий запрос, выходные данные для которого изображены на рис. 14.2.:

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

```
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

Browse : ↑↓←→  PgDn PgUp --▶| |◀-- Home
```

Рис. 14.2. Простой запрос с повторяющимися выходными данными

Среди них есть повторяющаяся комбинация значений (1001 с London), поскольку в запросе SQL не требуется исключать дубликаты (повторяющиеся значения). Однако, если применяется UNION для комбинации этого запроса с таким же запросом для таблицы Salespeople, избыточная комбинация исключается. На рис. 14.3 представлены выходные данные для следующего запроса:

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

Можно добиться того же (в некоторых программных продуктах SQL), указав UNION ALL вместо UNION:

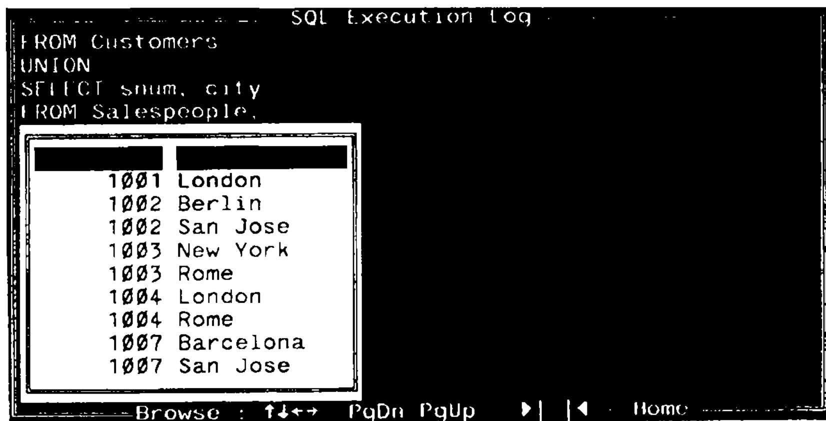


Рис. 14.3. Объединение исключает повторяющиеся выходные данные

```
SELECT snum, city
   FROM Customers

   UNION ALL

SELECT snum, city
   FROM Salespeople;
```

Использование строк и выражений с UNION

Иногда можно вставлять константы и выражения в предложения SELECT, использующие UNION. Это не соответствует в точности стандарту ANSI, но часто и оправданно применяется. Однако применяемые константы и выражения должны при этом удовлетворять стандарту сравнимости, о котором упоминалось ранее. Такая процедура может оказаться полезной, например, для формулировки комментария, определяющего, из какого конкретно запроса получена данная строка.

Предположим, необходимо сделать отчет, содержащий сведения для каждого продавца о его максимальном и минимальном заказах по каждой дате. Можно объединить два запроса, вставив соответствующий текст в качестве комментария, для того чтобы различить каждый из двух случаев (минимальный заказ и максимальный заказ).

```
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.date)

   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);
```

Выходные данные для этих команд представлены на рис. 14.4.

Необходимо добавить дополнительный пробел в строку 'Lowest on', для того чтобы длина этой строки соответствовала длине строки 'Highest on'. Peel выбран дважды, как имеющий максимальный и минимальный заказы на 5 октября 1990 г., хотя достаточно

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home
```

Рис. 14.4. Выбор наибольшей (highest) и наименьшей (lowest) заявок с поясняющими строками текста

было бы выбрать его один раз. Это произошло потому, что вставляемые строки для двух запросов различны, следовательно, строки выходных данных не удаляются автоматически как дублирующиеся.

Использование UNION с ORDER BY

До сих пор мы не придавали значения порядку представления данных множества запросов в выходных данных. Сначала представлялись выходные данные для первого запроса, затем — для второго. Нельзя считать, что данные автоматически будут следовать именно в таком порядке. Этот способ расположения (представления) выходных данных выбран исключительно для более простого восприятия результатов выполнения команды. Однако, предложение ORDER BY применяется и для упорядочения выходных данных объединения, так же как это делалось для отдельных (индивидуальных) запросов. Можно пересмотреть последний пример, предположив необходимость упорядочения выходных данных запроса. В этом случае станет ясно, почему, например, имя Peel в выходных данных предыдущего запроса многократно повторялось:

```
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
```

```
SQL Execution Log
(SELECT min (amt)
FROM orders c
WHERE c.odate = b.odate)
ORDER BY 5;

    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

Browse : ↑↓←→ PgDn PgUp →| |← Home
```

Рис. 14.5. Формирование объединения с использованием ORDER BY

```
            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)

ORDER BY 3;
```

Выходные данные для этого запроса представлены на рис. 14.5.

Поскольку способ упорядочения по возрастанию (ASC) для ORDER BY принят по умолчанию, он специально не указывается. Можно упорядочить выходные данные в соответствии со значениями нескольких полей: для каждого из полей независимо по возрастанию или убыванию (ASC или DESC), как это делалось для выходных данных одного запроса. Число 3 в предложении ORDER BY задает номер столбца в упорядоченном списке предложения SELECT. Поскольку столбцы выходных данных, полученных в результате выполнения объединения, являются непоименованными, на столбец можно сослаться только по номеру, определяющему его место расположения среди столбцов выходных данных.

Внешнее соединение

Часто бывает полезна операция объединения двух запросов, в которой второй запрос выбирает строки, исключенные первым. Обычно это приходится делать для исключения строк, не удовлетворяющих предикату при выполнении операции соединения таблиц. Это называется *внешним соединением* (*outer join*). Предположим, у некоторых из покупателей еще нет продавцов. Можно посмотреть имена и города всех покупателей с указанием имен их продавцов, не отбрасывая покупателей, еще не имеющих продавцов. Можно получить желаемые сведения, сформировав объединение двух запросов, один из которых выполняет объединение, а второй выбирает покупателей с NULL-значением в поле snum. Последний может вставлять пробелы в поле, соответствующее полю sname первого запроса.

Для идентификации запроса, с помощью которого получена данная строка, можно вставлять строки текста в выходные данные. Использование этой возможности во внешнем соединении позволяет применять предикаты для классификации выходных данных, а не для их исключения.

```
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

Browse : ↑↓←→ PgDn PgUp —▶| |◀— Home

Рис. 14.6. Внешнее соединение

Пример поиска продавцов (salespeople) с покупателями (customers), расположенными в тех же городах, рассматривался ранее. Теперь необходимо в составе выходных данных увидеть список всех продавцов и пометить тех , кто не имеет покупателей, находящихся в их городе (city), так же как и тех, кто таких покупателей имеет. Следующий запрос, выходные данные для которого представлены на рис. 14.6, позволяет сделать 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;
```

Строка 'NO MATCH' дополнена пробелами так, чтобы она соответствовала полю cname по длине (практически в этом нет необходимости для всех программных реализаций SQL). Второй запрос выбирает строки, не соответствующие предикату первого запроса.

В запрос можно добавить комментарий или выражение в качестве дополнительного поля. Для этого необходимо включить некоторый совместимый комментарий или выражение в соответствующую позицию списка имен полей предложения SELECT для каждого запроса в операторе объединения. Сравнимость по объединению предотвращает ситуации, когда дополнительное поле добавляется к одному из запросов, но не добавляется к другому. Вот пример запроса, который добавляет строки к выбранным полям. Текст этих строк сообщает о наличии для данного продавца назначенного ему покупателя из его же города ('MATCHED' — 'NO MATCH'):

```
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)
ORDERD BY 2 DESC;
```

Выходные данные для этого запроса представлены на рис. 14.7.

Это неполное внешнее соединение, поскольку оно включает только не назначенные (unmatched) поля для одной из участвующих в соединении таблиц. Полное внешнее соединение должно включать всех покупателей, которые как имеют, так и не

```
SQL Execution Log
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;

     1002 Serres     San Jose    MATCHED
     1007 Rifkin     Barcelona   NO MATCH
     1001 Peel       London      MATCHED
     1004 Motika     London      MATCHED
     1003 Axelrod    New York    NO MATCH

Browse : ↑↓←→  PgDn PgUp  Home
```

Рис. 14.7. Внешнее соединение с полями комментария

имеют продавцов в их городах. Очевидно, что это гораздо сложнее (выходные данные для следующего запроса представлены на рис. 14.8):

```
(SELECT snum, city, 'SALESPERSON - MATCHED'
   FROM Salespeople
   WHERE 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;
```

(Эта формулировка, использующая ANY, эквивалентна соединению в предыдущем примере.)

Сокращенные внешние соединения, с которых началось рассмотрение, оказываются полезными чаще, чем полные (рассмотрено в последнем примере). По поводу последнего примера можно заметить: если объединение выполняется более чем для двух

```
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
```

Рис. 14.8. Сложное внешнее соединение

запросов, то для упорядочения вычислений нужно использовать круглые скобки. Иначе говоря, вместо того, чтобы задать

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

необходимо конкретизировать либо

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

либо

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

Это необходимо, потому что UNION и UNION ALL можно комбинировать для исключения одних дубликатов без устранения других. Предложение

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

необязательно генерирует те же выходные данные, что и предложение

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

если имеются дубликаты строк, подлежащие исключению из выходных данных.

Итоги

Теперь вы знаете, как использовать предложение UNION, позволяющее комбинировать любое количество запросов в единственном теле запроса. Если есть ряд сходных таблиц, т.е. содержащих одинаковую информацию, но принадлежащих разным пользователям и имеющих различную специфику, объединение может стать легким способом слить и упорядочить выходные данные. Внешнее соединение — это новый способ применять условия, не исключая выходные данные, а только помечая или выделяя их части как удовлетворяющие и неудовлетворяющие данному условию.

Этим мы заканчиваем раздел книги, посвященный запросам. Следующая тема — введение данных в таблицы и создание новых таблиц. Вы узнаете, что запросы иногда используются не только самостоятельно, но и внутри других типов команд.

Работаем на SQL

1. Создайте объединение двух запросов, которые показывают значения полей names, cities, ratings для всех покупателей (customers). Те, у кого рейтинг (rating) 200 и выше, должны иметь комментарий "High Rating", все прочие — "Low Rating".

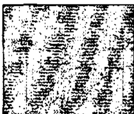
2. Запишите команду, которая выдает в качестве выходных данных значения полей name и number для каждого продавца и покупателя, имеющего более одного заказа. Результаты вывести в алфавитном порядке.

3. Сформируйте объединение трех запросов. Первый запрос выбирает значения поля snums для всех продавцов в San Jose. Второй запрос выбирает значения поля cnums для всех покупателей в San Jose; третий запрос выбирает значения поля onums для всех заявок за 3 октября 1990. Объединение должно сохранять дубликаты для двух последних запросов, но исключать любую избыточность между каждым из них (вторым и третьим) и первым.
(Замечание: в простых таблицах, используемых в примерах, такой избыточности может не оказаться, но это не изменяет сути дела.)

(Ответы см. в приложении А.)

15

Ввод, удаление и изменение значений полей



В этой главе рассматриваются команды, управляющие представленными в таблице значениями в любой момент времени. Вы научитесь размещать столбцы в таблице, исключать их и изменять отдельные значения, представленные в каждой строке. Вы познакомитесь здесь с применением запросов для генерации групп строк для вставки, а также с использованием предикатов для управления изменениями значений и удалением строк. Материал этой главы содержит большой объем сведений, необходимых для манипулирования информацией в базе данных. Некоторые более сложные способы построения предикатов будут рассмотрены в следующей главе.

Команды обновления DML

Данные заносятся в поля и исключаются из них с помощью трех команд языка манипулирования данными (Data Manipulation Language — DML: INSERT (вставить), UPDATE (обновить) и DELETE (удалить). В SQL их часто называют командами обновления *(update commands)*.

Ввод значений

Все строки в SQL вводятся при помощи команды обновления INSERT. В простейшем случае команда INSERT имеет такой синтаксис:

```
INSERT INTO <имя таблицы>
   VALUES (<значение>,<значение> ...);
```

Например, для того чтобы ввести строку в таблицу Salespeople, можно использовать следующее предложение:

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

Команды языка манипулирования данными не генерируют выходных данных, но программа должна уведомлять пользователя о том, что данные добавлены. Имя таблицы (в данном случае Salespeople) должно быть определено предварительно (до выполнения команды INSERT) с помощью команды CREATE TABLE (см. главу 17), а каждое значение в списке значений должно иметь тип данных, соответствующий типу данных столбца, в который это значение должно быть вставлено. Согласно стандарту ANSI, эти значения не могут включать выражения. Отсюда следует, что значение 3 допустимо, а 1 + 2 — недопустимо. Значения, конечно, вводятся в таблицу в порядке следования столбцов таким образом, что первое из значений, указанных в списке VALUES команды INSERT, вводится автоматически в столбец с номером 1, второе — в столбец с номером 2 и т.д.

Вставка NULL-значений

Если нужно вставить NULL-значение, необходимо указать его как обычное значение. Предположим, значение поля city для Ms.Peel еще неизвестно. В этом случае для нее можно вставить строку, указав значение NULL в столбце city, следующим образом:

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

Поскольку NULL является специальным символом, а не символьным значением, оно указано без одиночных кавычек.

Именование столбцов для INSERT

Для указания имен столбцов, в которые необходимо ввести значения, порядок столбцов в таблице неважен. Предположим, значения для таблицы Customers берутся из напечатанного отчета, в котором интересующие сведения представлены в таком порядке: city,cname,cnum. Для простоты желательно вводить значения в порядке, указанном в напечатанном отчете. Можно воспользоваться командой:

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

Столбцы с именами rating и snum опущены. Это означает, что для каждого из них автоматически назначаются значения по умолчанию. По умолчанию может быть установлено либо значение NULL, либо вполне определенное значение. Если ограничения целостности не допускают использования NULL-значения в данном столбце и для столбца не назначено значение "по умолчанию", то в такой столбец можно занести значение с помощью любой команды INSERT (информация по поводу ограничений на NULL-значения и использованию значений "по умолчанию" содержится в главе 18).

Вставка результатов запроса

Команду INSERT можно применить для того, чтобы извлечь значения из одной таблицы и разместить их в другой, воспользовавшись для этого запросом. Для этого достаточно заменить предложение VALUES на соответствующий запрос, как в примере:

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

По этой команде все значения, полученные с помощью запроса (т.е. все строки таблицы Salespeople, для которых значение поля city = 'London'), размещаются в таблице с

именем Londonstaff. Чтобы избежать проблем при выполнении команды, таблица Londonstaff должна удовлетворять следующим условиям:

- Она должна быть уже создана с помощью команды CREATE TABLE.
- Она должна иметь четыре столбца, соответствующих столбцам таблицы Salespeople в смысле типов данных: т.е. первый, второй и т.д. столбцы каждой из таблиц должны иметь один и тот же тип (использования одних и тех же имен не требуется).

Основное правило, в соответствии с которым столбцы таблицы вставляются, заключается в соответствии столбцов таблицы столбцам выходных данных запроса, в данном случае, таблице Salespeople целиком.

Londonstaff является здесь независимой таблицей, имеющей ряд значений, совпадающих со значениями таблицы Salespeople. Если значения в таблице Salespeople изменить, то эти изменения не отразятся на значениях, хранящихся в таблице Londonstaff (хотя эффект согласованного изменения данных можно создать, определив представления). Поскольку и запрос, и команда INSERT могут указывать столбцы по имени, при желании можно переставить выбираемые столбцы (указать имена в команде SELECT) или указать имена вставляемых столбцов в произвольном порядке (указать имена в команде INSERT).

Предположим, решено создать новую таблицу с именем Daytotals, которая будет отслеживать объемы заказов за каждый день. Допустим, необходимо ввести эти данные независимо от таблицы Orders, воспользовавшись при этом уже имеющимися в ней данными. Предположим, таблица Orders содержит данные по последнему финансовому году, а не за несколько дней, как в нашем примере. В этом случае преимущества использования предложения INSERT для вычисления и ввода значений вполне очевидны:

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

Заметим, что имена столбцов для двух таблиц Orders и Daytotals не совпадают. Но если date и total являются единственными столбцами таблицы и следуют в ней в указанном порядке, то имена в предложении INTO можно опустить.

Исключение строк из таблицы

Строки из таблицы можно исключить с помощью команды обновления DELETE. По этой команде исключаются только целые строки, а не отдельные значения полей; таким образом имя поля не является аргументом, необходимым для выполнения команды, и

воспринимается как ошибочный аргумент. Для исключения всех строк таблицы Salespeople, следует ввести следующее предложение:

```
DELETE FROM Salespeople;
```

В результате выполнения этой команды таблица становится пустой и ее можно удалить по команде DROP TABLE (команда объясняется в главе 17).

Обычно из таблицы требуется удалить только некоторые указанные строки. Чтобы их определить, можно, как и для запросов, использовать предикат. Например, чтобы исключить продавца Axelrod из таблицы, следует ввести:

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

Поле snum используется вместо поля sname, поскольку наилучший способ при удалении единственной строки — это указать значение ее первичного ключа. Применение первичного ключа гарантирует удаление единственной строки.

Можно употребить и предикат, выбирающий группу строк, например:

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

Изменение значений полей

По команде UPDATE можно изменять некоторые или все значения в существующей строке. Эта команда содержит предложение UPDATE, позволяющее указать имя таблицы, для которой выполняется операция, и SET предложение, определяющее изменение (изменения), которое необходимо выполнить для определенного столбца (столбцов). Например, для того чтобы изменить для всех покупателей рейтинг на 200, следует ввести:

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

Обновление только определенных строк

Замена значения столбца во всех строках таблицы, как правило, не нужна. Поэтому в команде UPDATE, как и в команде DELETE, можно использовать предикат. Для выполнения указанной замены значений столбца rating, для всех покупателей, которые обслуживаются продавцом Peel (snum = 1001), следует ввести:

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

UPDATE для множества столбцов

Нет нужды ограничиваться обновлением значения единственного столбца в результате выполнения команды UPDATE. В предложении SET можно указать любое количество значений для столбцов, разделенных запятыми. Все указанные изменения выполняются для каждой строки таблицы, удовлетворяющей предикату. В каждый момент времени обрабатывается одна строка таблицы. Предположим, Motika заменена новым продавцом (salesperson), необходимо сохранить ее персональный номер, но в соответствующую строку таблицы внести данные о новом продавце:

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

В результате выполнения команды все покупатели продавца Motika со своими заказами перейдут к Gibson, поскольку они связаны с Motika по значению поля snum.

Однако невозможно обновить множество *таблиц* с помощью единственной команды, так как нельзя использовать имя таблицы в качестве префикса имени столбца в предложении SET. Т.е. нельзя указать:

```
... SET Salespeople.sname = 'Gibson' ...
```

в команде UPDATE, можно только:

```
... SET sname = 'Gibson' ...
```

Использование выражений в UPDATE

В предложении SET команды UPDATE можно использовать скалярные выражения, указывающие способ изменения значений поля в отличие от предложения VALUES команды INSERT, в котором нельзя использовать выражения. Это весьма полезная характеристика. Предположим, решено удвоить комиссионные продавцов. Можно использовать следующее выражение:

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

Поскольку есть ссылка на значение существующего столбца в предложении SET, для каждой текущей строки выбирается значение указанного столбца, над которым выполняется заданная операция (в данном случае значение увеличивается в два раза). Можно комбинировать отдельные компоненты предложения UPDATE. Например, можно изменить значение комиссионных только для продавцов из Лондона с помощью предложения:

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

Применение UPDATE к NULL-значениям

Предложение SET не является предикатом. В нем можно указать значение NULL без использования какого-либо специального синтаксиса (например, такого как IS NULL). Таким образом, если нужно установить все рейтинги покупателей из Лондона (city = 'London') равными NULL-значению, необходимо ввести следующее предложение:

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

В результате выполнения этой команды значения рейтинга для всех покупателей из Лондона станут неопределенными (имеющими NULL-значение).

Итоги

Вы овладели средствами манипулирования содержимым базы данных с помощью трех простых команд. INSERT используется для замены содержимого строк в базе данных, DELETE — для удаления строк, UPDATE — для замены значений в строках таблицы. Вы можете применять предикаты с UPDATE и DELETE для определения конкретных строк таблицы, на которые воздействует команда. Предикаты не имеют значения для команды INSERT, поскольку рассматриваемая в команде строка не существует до тех пор, пока команда не выполнится. Однако, можно использовать запросы с INSERT для добавления множества строк в таблицу во время выполнения одной команды INSERT. Эти операции осуществляются при любом порядке расположения строк. Вы узнали, что значения, присваиваемые "по умолчанию", заносятся в те столбцы, значения которых явно не заданы. В операции может участвовать NULL-значение. Вновь обращаем ваше внимание на то, что в команде UPDATE можно использовать выражения, а в INSERT — нельзя.

Следующая глава расширит знания в области применения этих команд и использования с ними подзапросов, сходных с уже известными. В главе 16 мы обсудим также некоторые специальные вопросы и ограничения при использовании подзапросов в командах языка DML.

Работаем на SQL

1. Запишите команду, которая вводит следующие значения в заданном порядке в таблицу Salespeople: city — San Jose, name — Blanco, comm — NULL, cnum — 1100.
2. Запишите команду, которая исключает все заявки покупателя Clemens из таблицы Orders.
3. Запишите команду, которая увеличивает рейтинг всех покупателей в Rome на 100.
4. Продавец Serres покинула компанию. Переведите ее покупателей продавцу Motika.

(Ответы см. в приложении А.)

16

Использование подзапросов с командами обновления

Из этой главы вы узнаете, как применять подзапросы в командах обновления. Эта операция похожа на использование подзапросов в запросах. Зная, как подзапросы применяются в командах SELECT, довольно просто, несмотря на некоторые отличия, освоить их использование в командах обновления.

Подзапросы полностью являются командами SELECT, а не предикатами, поэтому данная операция отличается от использования предикатов с командами обновления. Простые запросы уже применялись для получения значения для INSERT, но теперь необходимо расширить эти запросы включением в них подзапросов.

Важный принцип, который нужно помнить при использовании команд обновления, состоит в том, что в предложении FROM любого подзапроса нельзя ссылаться на таблицу, изменяемую в основной команде. Это применимо ко всем трем командам обновления. Существует множество ситуаций, в которых было бы полезно использовать запросы к изменяемой таблице в процессе ее модификации, но при этом, как полагают разработчики SQL, может возникнуть двусмысленность и определенная сложность в практической реализации. Это не распространяется на текущую строку таблицы, на которую действует команда, т.е. на связанные подзапросы.

Использование подзапросов в INSERT

INSERT является простейшим из рассматриваемых случаев. Вы уже знаете, как вставить результаты запроса в таблицу. Можно использовать подзапросы внутри любого запроса, генерирующего значения для команды INSERT так же, как для других запросов — внутри предиката или предложения HAVING.

Предположим, есть таблица с именем SJpeople, определения столбцов которой полностью соответствуют определениям таблицы Salespeople. Известно, как заполнять таблицу, подобную этой, для всех покупателей (Customers), расположенных (city) в San Jose:

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

Теперь можно использовать подзапрос, для того чтобы добавить в таблицу SJpeople всех продавцов, имеющих покупателей в San Jose, независимо от места проживания продавца:

```
INSERT INTO SJpeople
   SELECT *
      FROM Salespeople
      WHERE snum = ANY
          (SELECT snum
                FROM Customers
```

```
WHERE city = 'San Jose');
```

Оба запроса в этой команде действуют так, будто они не являются частью выражения INSERT. Подзапрос отыскивает все строки для покупателей в San Jose и создает множество значений поля snum. Внешний запрос выбирает те строки из таблицы Salespeople, для которых найдены совпадающие значения snum. В данном примере в таблицу SJpeople вставляются строки для продавцов Rifkin и Serres, которым назначены покупатели из San Jose Liu и Cisneros.

Включение (предотвращение включения) одинаковых строк

Последовательность команд в предыдущем разделе может показаться проблематичной. Продавец Serres расположен в San Jose и, следовательно, будет добавлен в таблицу после выполнения первой команды. Вторая команда будет пытаться добавить его снова, поскольку он имеет покупателя в San Jose. Если имеются какие-либо ограничения для SJpeople, в соответствии с которыми значения должны быть уникальными, это второе появление (вторая вставка той же самой строки) может оказаться ошибочным. Дублирование строк — не самая удачная идея. Для того чтобы проконтролировать наличие в таблице значения, которое вставляется добавлением другого подзапроса (с использованием операторов EXISTS, IN, <>ALL и т.д.) для предиката, нужна ссылка на саму таблицу SJpeople в предложении FROM этого нового подзапроса, но нельзя ссылаться на таблицу, находящуюся в процессе формирования (как единого целого), в любом подзапросе команды обновления. В случае с INSERT, это также исключает возможность использования связанных подзапросов, базирующихся на таблице, в которую вносятся значения, что существенно, поскольку при использовании INSERT создается новая строка таблицы. "Текущая" строка не существует до тех пор, пока INSERT не закончит ее обрабатывать.

Использование подзапросов, основанных на таблицах внешних запросов

Запрет на использование ссылок на таблицу, находящуюся в процессе изменения командой INSERT, не мешает применять подзапросы, которые ссылаются на таблицу (таблицы), используемую в предложении FROM внешней команды SELECT. Таблица, из которой осуществляется выборка данных для получения значения для INSERT, не находится в состоянии изменения командой, и на нее можно ссылаться любым известным способом, как если бы это было отдельным запросом.

Предположим, существует таблица Samecity, в которой хранятся сведения о продавцах (salespeople), имеющих покупателей в их же городах (cities). Можно заполнить таблицу, применяя связанные подзапросы:

```
INSERT INTO Samecity
```

```
SELECT *
   FROM Salespeople outer
   WHERE city IN
       (SELECT city
            FROM Customers inner
            WHERE inner.snum = outer.snum);
```

Ни таблицу Salespeople, ни таблицу Samecity нельзя использовать во внешних или внутренних запросах для INSERT. Другой пример: предположим, назначено вознаграждение для каждого продавца, имеющего максимальный заказ на каждый день. Сведения о них можно формировать в таблице Bonus, содержащей значение поля snum для продавца, дату (odate) и объем заказа (amount, amt). Эту таблицу можно заполнить информацией, хранящейся в таблице Orders, 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);
```

Несмотря на то, что команда имеет подзапрос, базирующийся на той же таблице, что и внешний запрос, она не ссылается на таблицу Bonus, на которую воздействует эта команда. Следовательно, ее можно использовать. Логика запроса заключается в просмотре таблицы Orders, и для каждой строки осуществляется поиск максимальной заявки для конкретной даты. Если ее величина совпадает со значением поля amt текущей строки, то эта текущая строка и интересна, и ее данные заносятся в таблицу Bonus.

Использование подзапросов с DELETE

В предикате команды DELETE можно использовать подзапросы. Это дает возможность формулировать достаточно сложные критерии удаления строк, что требует особой внимательности. Например, если необходимо закрыть лондонский офис, можно использовать следующий запрос для исключения всех покупателей, назначенных продавцам в London:

```
DELETE
   FROM Customers
   WHERE snum = ANY
      (SELECT snum
           FROM Salespeople
```

```
            WHERE city = 'London');
```

В соответствии с этой командой из таблицы Customers будут исключены строки Hoffman, Clemens (обе назначены Peel) и строка Pereira (назначенная Motika). Желательно удостовериться в верном выполнении этой операции, прежде чем реально удалить или изменить строки с Peel и Motika.

Внимание! Когда выполняется изменение в базе данных, которое влечет другие изменения, первое желание — это выполнить сначала основное изменение, а затем — трассировку тех изменений, которые последуют в связи с первым. Этот пример показывает, почему более эффективно выполнять работу в обратной последовательности, т.е. сначала осуществить вторичные изменения. Если изменение значения поля city началось с выдачи новых значений (назначений) продавцам, то выполнение трассировки всех их покупателей окажется делом более сложным. Поскольку реальные базы данных существенно превосходят простые таблицы, которые рассматриваются здесь в качестве примера, при работе с ними могут возникнуть серьезные проблемы. Вам может оказаться полезен механизм ссылочной целостности SQL, но он не всегда применим и доступен.

Хотя нельзя сослаться в предложении FROM (подзапроса) на таблицу, из которой осуществляется удаление, в предикате можно ссылаться на текущую строку-кандидат из таблицы, т.е. на ту строку, которая в настоящее время проверяется в основном предикате. Другими словами, можно использовать связанные подзапросы. Они отличаются от применяемых с INSERT тем, что действительно базируются на строках-кандидатах из таблицы, на которую воздействует команда, а не на запросе для некоторой другой таблицы.

```
DELETE FROM Salespeople
   WHERE EXISTS
      (SELECT *
           FROM Customers
           WHERE rating = 100
           AND Salespeople.snum = Customers.snum);
```

Часть AND предиката внутреннего запроса ссылается на таблицу Salespeople. Это означает, что целый подзапрос будет выполняться отдельно для каждой строки таблицы Salespeople, как и в случае других связанных подзапросов. Команда удаляет всех продавцов, имеющих по крайней мере одного покупателя с рейтингом 100, из таблицы Salespeople. Для достижения этого существуют и другие способы. Приведем один из них:

```
DELETE FROM Salespeople
   WHERE 100 IN
         (SELECT rating
              FROM Customers
              WHERE Salespeople.snum = Customers.snum);
```

Запрос находит все рейтинги покупателей каждого продавца и удаляет продавца, имеющего покупателя с рейтингом 100.

Можно применять также обычные связанные подзапросы, т.е. связанные с таблицей, на которую есть ссылка во внешнем запросе (а не в самом предложении DELETE). Например, можно найти наименьший заказ за каждый день и удалить продавца, которому такой заказ был адресован, с помощью следующей команды:

```
DELETE FROM Salespeople
   WHERE snum IN
      (SELECT snum
           FROM Orders a
           WHERE amt =
               (SELECT MIN (amt)
                    FROM Orders b
                    WHERE a.odate = b.odate));
```

Подзапрос в предикате DELETE использует связанный подзапрос. Этот внутренний запрос находит минимальный заказ на каждую дату для каждой строки внешнего запроса. Если его величина совпадает с величиной заказа текущей строки, предикат внешнего запроса принимает значение "истина". Это значит, что текущая строка содержит минимальную заявку на данную дату. Поле snum для продавца, ответственного за эту заявку, извлекается и подставляется в основной предикат самой команды DELETE, которая затем удаляет все строки с этим значением поля snum из таблицы Salespeople. (Поскольку snum — это первичный ключ таблицы Salespeople, найдется единственная строка, подлежащая удалению в соответствии с этим запросом. Однако, если окажется, что таких строк больше, все они будут удалены.) Таким образом в данном случае будут удалены строки со следующими значениями snum: 1007 — минимум за 3 октября 1990 г.; 1002 — минимум за 4 октября 1990 г.; 1001 — минимальная и единственная заявка за 5 октября 1990 г. (команда кажется особенно жесткой, потому что удаляет Peel, как единственную заявку за 5 октября 1990 г., но она хорошо иллюстрирует суть дела).

Для того чтобы сохранить Peel, необходимо добавить другой подзапрос, как в следующем примере:

```
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));
```

В этом случае для дат, когда поступила только одна заявка, будет получено значение счетчика равное 1 во втором связанном подзапросе, что делает предикат внешнего запроса ложным, и, следовательно, эти значения поля snum не удовлетворяют основному предикату.

Использование подзапросов с UPDATE

UPDATE, как и DELETE, использует подзапросы внутри предиката. Можно применять связанные подзапросы в любой форме, приемлемой для DELETE: связанными либо с таблицей, которую следует модифицировать, либо с таблицей, на которую есть ссылка во внешнем запросе. Например, используя связанный подзапрос для таблицы, подлежащей обновлению, можно повысить комиссионные для всех продавцов, которые обслуживают по крайней мере двух покупателей:

```
UPDATE Salespeople
   SET comm = comm + .01
   WHERE 2 <=
      (SELECT COUNT (cnum)
          FROM Customers
          WHERE Customers.snum = Salespeople.snum);
```

Теперь для продавцов Peel и Serres, имеющих множество покупателей, будут увеличены комиссионные.

Приведем модификацию последнего примера для раздела, в котором рассматривалась команда DELETE. Здесь уменьшаются комиссионные для продавцов, получивших минимальные заказы:

```
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));
```

Ограничения подзапросов в командах DML

Невозможность сослаться на таблицу, изменяемую в любом подзапросе команды обновления, исключает целую категорию возможных трансформаций. Например, нельзя просто выполнить операцию удаления всех покупателей с рейтингом ниже среднего. Сначала надо выполнить запрос, получающий значение среднего рейтинга, а затем удалить все строки таблицы Customers, в которых рейтинг ниже этого значения:

Шаг 1.

```
SELECT AVG (rating)
   FROM Customers;
```

Выходными данными для этого запроса является значение 200.

Шаг 2.

```
DELETE
   FROM Customers
   WHERE rating < 200;
```

Итоги

Теперь вам известны три команды, управляющие всем содержимым базы данных. Хотелось бы только пояснить несколько основных моментов, связанных с вводом и удалением значений из таблицы: когда эти команды следует выполнять данному пользователю для данной таблицы и когда сделанные изменения станут постоянными.

Итак, команда INSERT применяется для того, чтобы добавить строки в таблицу. Можно ввести имена значений для строк в предложении VALUES (оно определяет только одну добавляемую строку), либо значения можно получить из запроса (это означает, что любое количество строк может быть добавлено с помощью одной команды). Если используется запрос, то он не может ссылаться на таблицу, в которую осуществляется вставка каким-либо из способов: либо с помощью предложения FROM, либо путем внешней ссылки (как это делается в связанных запросах). Это применимо к любым подзапросам внутри данного запроса. Запрос сохраняет свободу в применении связанных подзапросов или подзапросов, использующих имена таблиц в предложении FROM внешних запросов (это является общим случаем применения запросов).

DELETE и UPDATE используются для исключения строк из таблицы и изменения значений в них. Обе команды применяются ко всем строкам таблицы, но можно употребить и предикат для определения подмножества удаляемых или обновляемых строк. Этот предикат может содержать подзапросы, связанные с таблицей, из которой выпол-

няется удаление или для которой выполняется обновление, с использованием внешней ссылки. Но эти подзапросы не могут ссылаться на модифицируемую таблицу, имя которой указано в предложении FROM.

В следующих главах будет рассмотрен вопрос создания новых таблиц.

Работаем на SQL

1. Предположим, существует таблица с именем Multicust, определения столбцов которой полностью совпадают с определениями таблицы Salespeople. Запишите команду, которая вставляет в эту таблицу всех продавцов, имеющих более одного покупателя.

2. Запишите команду, которая удаляет всех покупателей, не имеющих в настоящее время заказов.

3. Запишите команду, увеличивающую на 20% комиссионные всех продавцов, общее количество заказов которых превышает $3,000.

(Ответы см. в приложении А.)

17

Создание таблиц

Задача этой книги — рассмотреть как можно больше проблем, связанных с обработкой информации на компьютере, начиная с наиболее часто встречающихся на практике, и постепенно переходя к более специфичным проблемам.

В этой главе обсуждаются проблемы создания, изменения и удаления таблиц. Речь пойдет об определениях таблиц, а не о данных, в них хранящихся. Реальная потребность в выполнении этих операций может и не возникнуть, но следует на концептуальном уровне иметь представление об этих операциях, что повышает компетентность пользователя в области применения SQL и в плане понимания природы используемых таблиц. Это относится к разделу SQL, называемому языком определения данных (Data Definition Language — DDL), в котором происходит создание объектов SQL.

В этой главе обсуждается и другой тип объектов данных SQL — индексы, применяемые для более эффективной организации поиска и для того, чтобы убедиться, что значения отличаются друг от друга. Подобные операции выполняются невидимо (без участия пользователя), но при попытке ввести значения в таблицу они могут отвергаться (не восприниматься), так как не являются уникальными. Это означает, что хотя две строки могут иметь в поле одинаковое значение, поле все равно будет обладать своим уникальным индексом, в противном случае происходит нарушение ограничений целостности.

Команда CREATE TABLE

Таблицы определяются с помощью команды CREATE TABLE, создающей пустую таблицу — таблицу, не имеющую строк. Значения вводятся с помощью команды DML (языка манипулирования данными) INSERT (см. главу 15). Команда CREATE TABLE определяет имя таблицы и множество поименованных столбцов в указанном порядке. Для каждого столбца устанавливаются тип и размер. Каждая таблица должна иметь, по крайней мере, один столбец. Синтаксис команды CREATE TABLE:

```
CREATE TABLE <имя таблицы>
    (<имя столбца> <тип данных> [(<размер>)],
    <имя столбца> <тип данных> [(<размер>)],...);
```

Типы данных существенно различаются в разных программных продуктах. Однако в целях совместимости со стандартом, они, как минимум, поддерживают стандартные ANSI-типы. Они приводятся в приложении В.

Поскольку пробелы используются для разделения отдельных частей команд в SQL, их нельзя использовать как часть имени таблицы (либо как часть какого-либо другого объекта, например, индекса). Символ подчеркивания (_) наиболее часто используется для разделения слов в именах таблиц.

Значение аргумента размера зависит от типа данных. Если он не указывается, то система установит значение автоматически. Вероятно, это наиболее удачное решение для числовых значений, поскольку в данном случае все поля определенного типа имеют один и тот же размер, что позволяет впоследствии не заботиться о совместимости

по объединению. Использование аргумента размера с некоторыми числовыми типами является непростой задачей. Для хранения больших чисел вы должны убедиться, что поле имеет достаточную длину.

Тип данных, для которого обязательно следует указывать размер, — это CHAR. В данном случае аргумент размера — целое число, задающее максимальное число символов, которые могут содержаться в поле. Реальное количество символов в поле может изменяться от нуля (если в поле содержится NULL-значение) до заданного максимального значения. По умолчанию это 1, т.е. в поле может содержаться единственный символ.

Таблицы принадлежат пользователю, который их создал, а имена этих таблиц должны различаться, как и имена столбцов, в пределах одной таблицы. Даже если различные таблицы принадлежат одному пользователю, они могут иметь одноименные столбцы. Примером такой ситуации является столбец с именем city, представленный в таблицах Salespeople и Customers. Пользователи, отличные от владельца таблицы, ссылаются на нее, указав имя владельца, непосредственно после которого следует точка, непосредственно за ней — имя таблицы. Например, на таблицу Employees, созданную Smith, любой другой пользователь ссылается таким образом: Smith.Employees. Предположим, в данном случае Smith является идентификатором пользователя. В SQL предполагается, что имя пользователя может использоваться в качестве идентификатора.

Следующая команда позволяет создать таблицу Salespeople:

```
CREATE TABLE Salespeople
   (snum     integer,
   sname     char(10),
   city      char(10),
   comm      decimal);
```

Порядок столбцов в определении таблицы существенен, он определяет порядок, в котором задаются значения элементов строк. Определения столбцов не должны задаваться в отдельных строках, но они должны разделяться запятыми.

Индексы

Индекс (index) — это упорядоченный (в алфавитном или числовом порядке) список содержимого столбцов или группы столбцов в таблице. Таблицы могут иметь большое количество строк и, поскольку строки задаются в любом произвольном порядке, поиск их по значению какого-либо из полей может занять достаточно много времени. Индексы предназначены для решения этой проблемы и для объединения всех значений в группы из одной или нескольких строк, отличных друг от друга. В главе 18 будет рассмотрен более простой способ унификации значений, отсутствовавший в ранних версиях SQL, в которых для этих целей использовались только индексы.

Индексы — средство SQL, созданное из коммерческих соображений. В связи с этим стандарт ANSI в настоящее время практически их не поддерживает, но они весьма полезны и используются на практике.

Когда создается индекс по значениям какого-либо поля для базы данных, создается упорядоченный список значений для этого поля. Предположим, таблица Customers имеет тысячи строк и нужно найти покупателя с номером 2999. Поскольку строки не упорядочены, программа должна просмотреть всю таблицу, строку за строкой, и выбрать ту, в которой значение поля cnum равно 2999. Если бы по полю cnum был организован индекс, программа могла бы сразу найти в нем значение 2999 и получить информацию о том, как обнаружить нужную строку таблицы. Это может значительно улучшить выполнение запросов, но управление индексами существенно замедляет время выполнения операций обновления (таких как INSERT и DELETE); кроме того, сам индекс занимает место в памяти. Следовательно, перед созданием индексов следует тщательно проанализировать ситуацию.

Индексы можно создавать по множеству полей. Если указано более одного поля для создания единственного индекса, данные упорядочиваются по значениям первого поля, по которому осуществляется индексирование. Внутри получившихся групп осуществляется упорядочение по значениям второго поля, для получившихся в результате групп осуществляется упорядочение по значениям третьего поля и т.д. Если есть первое и последнее имена для двух разных полей таблицы, можно создать индекс, который упорядочивает первое внутри последнего. Это можно сделать независимо от порядка строк в таблице.

Синтаксис команды создания индекса обычно выглядит так:

```
CREATE INDEX <имя индекса> ON <имя таблицы> (<имя столбца>
   [,<имя столбца>]..);
```

Таблица должна быть уже создана и содержать столбцы, имена которых указаны в команде. Имя индекса, определенное в команде, должно быть уникальным в базе данных, то есть оно не может использоваться для каких-либо других целей любым пользователем базы данных. Будучи однажды созданным, индекс является невидимым для пользователя. SQL сам решит, когда есть смысл воспользоваться индексом, и сделает это автоматически. Например, если часто используется таблица Customers для поиска клиентов для конкретных продавцов по значениям поля snum, следует создать индекс по полю snum таблицы Customers.

```
CREATE INDEX Clientgroup ON Customers(snum);
```

Теперь вы можете быстро найти клиентов для продавцов.

Уникальные индексы

Для индекса в предыдущем примере уникальность не нужна. Данный продавец может иметь любое количество покупателей. Это становится неприемлемым, если ключевое слово UNIQUE используется перед ключевым словом INDEX. Поле

cnum, как первичный ключ, может быть первым кандидатом на создание уникального индекса:

```
CREATE UNIQUE INDEX Custid ON Customers(cnum);
```

Замечание: Эта команда будет отвергнута, если в поле cnum имеются одинаковые значения. Наилучший способ работы с индексами — немедленное их создание после создания таблицы и перед занесением в нее значений. Уникальный индекс, создаваемый для более чем одного поля, требует, чтобы комбинация значений во всех столбцах была уникальной.

Предыдущий пример помог выяснить, можно ли использовать поле cnum в качестве первичного ключа таблицы Customers. Для базы данных можно выполнить тщательное планирование первичного и других ключей.

Удаление индексов

Основная причина именования индексов состоит в их удалении время от времени. Обычно пользователи не знают о существовании индекса. SQL автоматически определяет его необходимость и создает его, если это нужно. При исключении индекса (вы обязательно должны знать его имя) используется такой синтаксис:

```
DROP INDEX <имя индекса>;
```

Удаление индекса не изменяет содержимого поля (полей).

Изменение таблицы, которая уже была создана

Команда ALTER TABLE (изменить таблицу), не являясь частью стандарта ANSI, широко применяется. Форма команды достаточно прозрачна, хотя ее возможности изменяются в широких границах. Обычно она осуществляет добавление столбцов в таблицу, иногда может удалять столбцы или изменять их размеры — осуществлять добавление и удаление ограничений. Обычный синтаксис команды, предназначенной для добавления столбца в таблицу выглядит следующим образом:

```
ALTER TABLE <имя таблицы> ADD <имя столбца>
   <тип данных> <размер>;
```

По этой команде для существующих в таблице строк добавляется столбец, в который заносится NULL-значение. Новый столбец становится последним столбцом в таблице. Допустимо добавление в нее нескольких столбцов с помощью одной команды; в этом случае их определения разделяются запятой. Можно исключать столбцы или изменять их описания. Часто изменение столбцов связано с изменением их размеров, добавлением или удалением ограничений. Система должна предоставить пользовате-

лю средства контроля, позволяющие удостовериться, что введенные данные, хранящиеся в таблице к моменту выполнения команды ALTER TABLE, удовлетворяют заданным в команде новым ограничениям. Для этого команда отвергается (выполнение команды завершается аварийно). Однако, наилучший вариант — возможность двойного контроля ситуации. Необходимо изучить соответствующие разделы документации по конкретной системе, прежде чем приступить к выполнению этой операции. Из-за нестандартной природы команды ALTER TABLE следует постоянно обращаться к документации по конкретной системе, прежде чем приступать к внесению каких-либо изменений в таблицы.

ALTER TABLE становится неоценимой, когда возникает потребность переопределить таблицу, но база данных должна проектироваться так, чтобы, по возможности, избежать подобных ситуаций. Изменение структуры таблицы, используемой в настоящее время, дело рискованное. Представления таблицы, которые создаются на основе данных, хранящихся в реальных таблицах, могут не допустить выполнения этой команды; программы, использующие встроенный SQL, могут привести к ошибочной ситуации в процессе выполнения этой команды, либо могут отвергать эту команду. Кроме того, в процесс изменения таблицы могут оказаться вовлеченными все пользователи, имеющие дело с этой таблицей. По этой причине следует стараться проектировать таблицы с учетом перспективы их использования, а необходимость выполнения команды ALTER TABLE следует рассматривать как крайнюю меру.

Если система не поддерживает команду ALTER TABLE или если желательно избежать применения этой команды, можно создать новую таблицу с необходимыми изменениями в ее определении, а затем использовать команду INSERT с запросом SELECT * для передачи в новую таблицу существовавших ранее данных. Пользователи, имевшие доступ к старой таблице, автоматически наследуют право доступа к новой таблице.

Исключение таблицы

Необходимо быть владельцем (создателем) таблицы, чтобы иметь возможность ее удалить. Чтобы не причинить ущерба данным, хранящимся в базе данных, необходимо предварительно удалить все данные из таблицы, то есть сделать ее пустой, а затем уже исключить таблицу из базы данных. Таблица, имеющая строки, не может быть удалена. Синтаксис команды, осуществляющей удаление пустой таблицы (определения таблицы) из системы таков:

```
DROP TABLE <имя таблицы>;
```

После выполнения команды, имя таблицы больше не распознается как имя таблицы, команды не могут работать с объектом, имя которого было указано в команде DROP. Перед выполнением команды следует удостовериться, что эта таблица не со-

держит внешних ключей для какой-либо другой таблицы и что эти таблицы не используются для определения представлений

Команда реально не является частью стандарта ANSI, но поддерживается и является полезной. Она весьма проста и не имеет различий в толковании (как команда ALTER TABLE). ANSI не оговаривает способа удаления или отказа от определений таблиц.

Итоги

Эта глава вводит вас в курс определения данных. Вы можете теперь создавать, модифицировать и удалять таблицы. Поскольку только первая из перечисленных функций является частью официального SQL-стандарта, детали остальных команд существенно различаются для различных программных продуктов, особенно для команды ALTER TABLE. DROP TABLE позволяет избавиться от таблиц, потерявших свою актуальность. Она удаляет только пустые таблицы и, следовательно, не разрушает данные.

В этой главе дано общее описание индексов, процедуры их создания и удаления. SQL не предоставляет широких возможностей в плане управления процессом выполнения команд. Скорость выполнения различных команд определяется конкретной реализацией программного продукта. Индексы являются одним из средств воздействия на процесс выполнения команды в SQL. Более подробно индексы и ограничения будут рассмотрены в двух следующих главах.

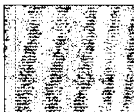
Работаем на SQL

1. Запишите предложение CREATE TABLE, которое создаст таблицу Customers.
2. Запишите команду, которая позволит пользователю быстро выбирать заказы из таблицы Orders, сгруппированными по датам.
3. Как можно добиться уникальности поля onum в предположении, что таблица Orders уже создана? (Предположим, что все существующие значения этого поля являются уникальными.)
4. Создайте индекс, который позволит каждому продавцу быстро осуществить поиск его покупателей, сгруппированных по датам.
5. Предположим, что каждый продавец должен иметь только одного покупателя с данным рейтингом. Все существующие данные удовлетворяют этому требованию. Введите команду, которая этому требованию не удовлетворяет.

(Ответы см. в приложении А.)

18

Ограничения на множество допустимых значений данных



В главе 17 мы рассказали о создании таблиц. Теперь рассмотрим в деталях, как определять ограничения на них. *Ограничения (constraints)* являются частью определения таблицы, в которой ограничиваются те значения, которые можно ввести в столбцы таблицы. До сих пор в этой книге предполагалось только одно ограничение на значения, которые можно вводить в таблицу: они должны иметь типы данных и размеры, совместимые со столбцами, в которые эти значения вводятся (как определено в команде CREATE TABLE или ALTER TABLE). Ограничения открывают более существенные возможности для управления данными.

Из этой главы вы узнаете, как задавать значения, принимаемые "по умолчанию", — те, которые автоматически подставляются в столбец таблицы, когда его значение опущено в команде INSERT для этой таблицы. Эта глава научит задавать значения "по умолчанию", среди которых наиболее часто используемое — NULL-значение. Значения, присваиваемые "по умолчанию", не являются ограничениями, но процедуры, применяемые в их определении, сходны.

Ограничения в таблицах

Когда создается таблица (или когда она изменяется), можно определить ограничения на значения, которые вводятся в поля, и SQL будет отвергать любое из них, если оно не соответствует определенному критерию. Два основных типа ограничения — это ограничения на столбцы и на таблицу. Разница между ними состоит в том, что *ограничения на столбцы (column constraints)* применимы к только к отдельным столбцам, а *ограничения на таблицу (table constraints)* применимы к группам, состоящим из одного или более столбцов.

Объявление ограничений

Ограничения на столбец добавляются в конце определения столбца после указания типа данных и перед запятой. Ограничения на таблицу размещаются в конце определения таблицы, после определения последнего столбца, перед закрывающей круглой скобкой. Команда CREATE TABLE имеет следующий синтаксис, расширенный включением ограничений:

```
CREATE TABLE <имя таблицы>
   (<имя столбца> <тип данных> <ограничения на столбец>,
   <имя столбца> <тип данных> <ограничения на столбец> ...
   <ограничения на таблицу> (<имя столбца>
   [,<имя столбца>...])...);
```

(Для большей ясности синтаксиса опущен аргумент размера, который иногда используется с типом данных.) Поля, заданные в круглых скобках после ограничений

таблицы, — это поля, на которые эти ограничения распространяются. Ограничения на столбцы применяются к тем столбцам, за которыми они следуют. Далее вы познакомитесь с описанием различных типов ограничений и их использованием.

Использование ограничений для исключения NULL-значения

Для того чтобы запретить использование NULL-значений в поле, можно применить команду CREATE TABLE, указав ключевое слово NOT NULL. Это ограничение распространяется только на множество столбцов.

NULL — специальный символ, обозначающий, что поле пусто. Но он полезен не всегда. Первичные ключи никогда не содержат NULL-значений, поскольку это нарушило бы функциональную зависимость. Во многих случаях необходимо, чтобы поля содержали определенные значения. Например, вам может потребоваться, чтобы в поле name таблицы Customers обязательно было указано имя покупателя.

Если ключевое слово NOT NULL размещается непосредственно после типа данных (включая размер) столбца, то любые попытки ввести NULL-значения в поле будут отвергнуты. В противном случае SQL разрешит использовать NULL-значения.

Например, необходимо улучшить определение таблицы Salespeople, запретив использование NULL-значений для столбцов snum и sname:

```
CREATE TABLE Salespeople
   (snum     integer NOT NULL,
   sname     char(10) NOT NULL,
   city      char(10),
   comm      decimal);
```

Важно помнить, что для каждого столбца, имеющего ограничение NOT NULL, в предложении INSERT для этой таблицы должно быть указано значение. При отсутствии NULL-значений в эти столбцы не будет введено никакого значения, если только не указано значение по умолчанию (о котором мы расскажем здесь позже).

Если система поддерживает применение команды ALTER TABLE с целью добавления столбцов в существующую таблицу, можно указать в ней и ограничения, такие как NOT NULL, для новых столбцов. Если для нового столбца задано NOT NULL, таблица должна быть пустой.

Как убедиться в том, что значения являются уникальными

В главе 17 обсуждалось использование уникальных индексов для отслеживания ситуации, когда поля имеют различные значения в различных строках. Такая практика сложилась в SQL с появлением ограничения UNIQUE. Уникальность — это свойство данных в таблице, и естественнее определить его не логическим свойством объекта данных (индексом), а ограничением на данные.

Уникальные индексы являются одним из наиболее эффективных методов поддержки уникальности. По этой причине некоторые программные продукты связывают ограничение UNIQUE с уникальными индексами: оно создает индекс, не сообщая об этом пользователю. Если ограничение уникальности задано, появляется меньше шансов попасть в непредвиденную или противоречивую ситуацию.

Уникальность как ограничение на столбец. Иногда надо удостовериться, что все значения, введенные в столбец, отличаются друг от друга. Например, когда этого требуют первичные ключи. Если при создании таблицы указывается ограничение UNIQUE для столбца, то база данных отвергнет любую попытку ввести в поле какой-либо строки значение, уже содержащееся в другой строке. Это ограничение применимо к тем полям, которые были объявлены NOT NULL, поскольку нет большого смысла в том, чтобы разрешить присутствие одного NULL-значения и затем исключить все повторяющиеся значения. Можно предложить такое определение таблицы Salespeople:

```
CREATE TABLE Salespeople
   (snum     integer NOT NULL UNIQUE,
   sname     char(10) NOT NULL UNIQUE,
   city      char(10),
   comm      decimal);
```

Объявляя поле sname уникальным, вы можете быть уверены, что две разные Mary Smith будут введены различными способами, например: Mary Smith и M. Smith. Это не является необходимым с точки зрения функциональной зависимости — поле snum, как первичный ключ, обеспечивает различие двух строк; но с точки зрения использования данных может оказаться полезным различать две личности указанием их различных имен. Столбцы, отличные от первичного ключа, для которых требуется поддерживать уникальность значений, называются *возможными ключами* (*candidate keys*) или *уникальными ключами* (*unique keys*).

Уникальность как ограничение таблицы. Можно сделать группу полей уникальными, указав в качестве ограничения таблицы UNIQUE. Определение группы столбцов уникальными отличается от определения уникальным одного столбца тем, что комбинация значений, каждое из которых не является уникальным, может быть уникальной. При объединении в группу важен порядок. Например, зна-

чения столбцов 'a', 'b' и 'b', 'a' отличаются друг от друга (дают две различные комбинации).

База данных разрабатывается в предположении, что каждому покупателю назначен только один продавец. Это означает, что каждая комбинация номера покупателя (customer number) и номера продавца (salespeople number) в таблице Customers является уникальной. Можно удостовериться в этом, задав определение таблицы Customers таким образом:

```
CREATE TABLE Customers
   (cnum      integer NOT NULL,
   cname      char(10) NOT NULL,
   city       char(10),
   rating     integer,
   snum       integer NOT NULL,
   UNIQUE     (cnum, snum));
```

Оба поля с ограничением UNIQUE в таблице имеют ограничения и для каждого из столбцов NOT NULL. Если бы использовалось ограничение UNIQUE на столбец cnum, то ограничение на таблицу не было бы необходимым. Если поле cnum различно для каждой строки, то не может быть двух строк с одинаковыми комбинациями cnum и snum. То же рассуждение было бы справедливым, если бы было объявлено уникальным поле snum, хотя для данного случая этот вариант неприемлем: один продавец может обслуживать множество покупателей. Следовательно, ограничение на таблицу UNIQUE является практически полезным, если не требуется поддерживать уникальность отдельных полей.

Предположим, разработана таблица для отслеживания всех покупок, сделанных за день, у определенного продавца. Каждая строка этой таблицы представляет итог для произвольного количества покупок, а не индивидуальную покупку. В этом случае можно исключить некоторые возможные ошибки, чтобы удостовериться, что каждый день представлен для одного продавца не более чем одной строкой, т.е. каждая комбинация snum и odate является уникальной. Следовательно, можно создать следующую таблицу, названную Salestotal:

```
CREATE TABLE Salestotal
   (snum,     integer NOT NULL,
   odate,     date NOT NULL,
   totamt,    decimal,
   UNIQUE     (snum, odate));
```

С помощью следующей команды можно внести значения в эту таблицу:

```
INSERT INTO Salestotal
   SELECT snum, odate, SUM (amt)
      FROM Orders
      GROUP BY snum, odate;
```

Ограничения для первичного ключа (PRIMARY KEY)

До сих пор мы обсуждали первичный ключ в качестве логической концепции. Известно, чем он является для любой таблицы и как он должен использоваться. Но что об этом "знает" SQL? Нужно использовать ограничение UNIQUE или уникальные индексы для первичного ключа, чтобы удостовериться в их уникальности. В ранних версиях SQL это было необходимо, сейчас же применяется по желанию. Однако сейчас SQL поддерживает первичные ключи непосредственно с помощью ограничения PRIMARY KEY. Это ограничение может быть доступно или недоступно в конкретной системе.

Ограничение PRIMARY KEY может быть применено к таблице или к множеству строк. По функциональным возможностям оно сходно с ограничением UNIQUE, за исключением того, что только один первичный ключ (состоящий из любого количества столбцов) может быть определен для данной таблицы. Следовательно, существует различие между первичным ключом и уникальными столбцами в способах их использования с внешними ключами. Синтаксис и определение уникальности зависят от ограничения уникальности (UNIQUE).

В первичном ключе недопустимо применение NULL-значений. Это означает, что, подобно полям с ограничениями UNIQUE, любое поле, используемое в ограничении PRIMARY KEY, должно быть объявлено NOT NULL. Вот усовершенствованная версия определения таблицы Salespeople:

```
CREATE TABLE Salespeople
(snum  integer NOT NULL PRIMARY KEY,
sname  char(10) NOT NULL UNIQUE,
city   char(10),
comm   decimal);
```

Ясно, что поля с атрибутом UNIQUE можно отнести к той же самой таблице. Лучший способ ввести ограничение PRIMARY KEY для полей — это ввести уникальный идентификатор строк и сохранить ограничение UNIQUE для полей, которые должны быть уникальными, исходя из логики данных (например, номера телефона или поле sname вместо использования идентификации строк).

Первичные ключи, состоящие более чем из одного поля. Ограничение PRIMARY KEY может относиться к множеству полей, дающих уникальную комбинацию значений. Предположим, первичным ключом является имя (name), а фамилия (first name) и имя (last name) хранятся в двух различных полях (таким образом можно организовать данные как единое целое). Ни имя, ни фамилия не являются уникальными (их уникальности практически невозможно добиться) Можно применить ограничение PRIMARY KEY в качестве ограничения таблицы к паре столбцов:

```
CREATE TABLE Namefield
    (firstname    char(10) NOT NULL,
```

```
lastname     char(10) NOT NULL,
city         char(10),
PRIMARY KEY  (firstname, lastname));
```

Единственная проблема, возникающая при использовании этого подхода, состоит в том, что для достижения уникальности нужно приложить усилия. Например, задавая Mary Smith и M. Smith, вы рискуете их перепутать. Проще ввести числовое поле, которое позволит различать строки, и это поле определить как PRIMARY KEY, а ограничение UNIQUE задать для двух имен полей.

Выбор значений поля

Можно определить любое количество ограничений, которым должны удовлетворять данные, вводимые в таблицы, например: попадают ли данные в заданный диапазон допустимых значений, и имеют ли они корректный формат. Для этих целей SQL предоставляет ограничение CHECK, позволяющее определить условие, которому должны удовлетворять вводимые в таблицу значения; проверка осуществляется до размещения данных в таблице. Ограничение CHECK содержит ключевое слово CHECK, за которым следует заключенный в круглые скобки предикат, применяемый к заданному для поля значению. Любая попытка обновить или заменить значения поля на те, для которых предикат принимает значение "ложь", отвергается.

Рассмотрим таблицу Salespeople. Комиссионные (столбец comm) представлены как имеющие тип decimal, следовательно, это значение можно непосредственно умножить на то, которое хранится в столбце amount, чтобы получить правильное выражение в долларах. Некто, привыкший думать о комиссионных в процентах, может не обратить на это внимания. Если он вводит значение 14 вместо .14 в качестве значения поля, содержащего величину комиссионных, то оно воспримется как 14.0, т.е. правильное десятичное значение. Для защиты от ошибок такого рода можно воспользоваться ограничением на столбец CHECK, чтобы убедиться, что значения поля comm не превышают 1.

```
CREATE TABLE Salespeople
(snum   integer NOT NULL UNIQUE,
sname   char(10) NOT NULL UNIQUE,
city    char(10),
comm    decimal CHECK (comm < 1 ) );
```

Использование CHECK для предотвращения ввода ошибочных значений. Можно использовать ограничение CHECK для обеспечения ввода в поле специфичных значений и таким образом избежать ошибок. Предположим, офисы по продаже есть только в городах London, Barcelona, San Jose и New York. Поскольку известно, что каждый из продавцов работает в одном из этих офисов, не имеет смысла разрешать ввод других значений в этот столбец (столбец city) таблицы Salespeople. Использование ограничения на множество допустимых значений с помощью явного перечисления всех

элементов этого множества позволит исключить большинство ошибочных ситуаций. Ограничение задается так:

```
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','Barcelona')),
   comm      decimal CHECK (comm < 1) );
```

Можно ввести это ограничение, если вы совершенно уверены в том, что кампания не имеет и в ближайшем будущем не предполагает иметь офисы в других городах. Изменение описания созданной таблицы — полезная, но рискованная операция, так как она не является частью стандарта ANSI. Большинство программ обработки баз данных поддерживает команду ALTER TABLE, разрешающую изменять описание таблицы, даже если эта таблица используется. Однако, изменение или удаление ограничений не всегда разрешены в этой команде, даже если она поддерживается программным продуктом. Если применяется программный продукт, который не поддерживает изменение или удаление ограничений, нужно создать с помощью команды CREATE новую таблицу и передать в нее данные из старой с учетом новых ограничений. Эта операция выполняется редко.

Перед вами определение таблицы Orders:

```
CREATE TABLE Orders
   (onum    integer NOT NULL UNIQUE,
   amt      decimal,
   odate    date NOT NULL,
   cnum     integer NOT NULL,
   snum     integer NOT NULL);
```

Тип DATE широко поддерживается, но не является частью стандарта ANSI. Что делать, если используется база данных, которая в соответствии со стандартом ANSI не распознает тип данных DATE? Если объявить, что тип данных odate должен быть числовым, то нельзя использовать ни слэш ("/"), ни дефис ("-") в качестве разделителей. Поскольку символы, выводимые на экран дисплея или на печатающее устройство, являются символами из набора ASCII, можно объявить столбец odate имеющим символьный (CHAR) тип. При этом возникает единственная проблема: необходимо использовать одиночные кавычки всякий раз, когда есть ссылка в запросе на значение поля odate. Простого решения этой проблемы не существует, и именно по этой причине тип DATE стал так популярен. Для иллюстрации можно предположить, что объявлен столбец odate типа CHAR. Можно определить формат для столбца odate в ограничении CHECK:

```
CREATE TABLE Orders
   (onum    integer NOT NULL UNIQUE,
   amt      decimal,
   odate    char(10) NOT NULL CHECK (odate LIKE '_ _/_ _/_ _ _ _'),
```

```
    cnum     integer NOT NULL,
    snum     integer NOT NULL);
```

Кроме того, можно ввести ограничения для того, чтобы убедиться, являются ли введенные символы цифровыми и лежат ли они в допустимых границах.

Выбор ограничений, основанный на множестве полей. Можно использовать CHECK как ограничение таблицы тогда, когда нужно включить более одного поля в условие ограничения. Предположим, комиссионные .15 и выше назначаются только продавцам из Барселоны (Barcelona). Это можно учесть, определив ограничение (CHECK) на таблицу:

```
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') );
```

Ясно, что два различных поля используются для проверки истинности предиката. Однако следует помнить, что здесь идет речь о двух различных полях из одной и той же строки. Даже при использовании множества полей SQL не может обрабатывать более одной строки в единицу времени. Например, не удастся просто применить ограничение CHECK для проверки того, что все комиссионные в пределах данного города одинаковы. Для выполнения этой проверки SQL должен обрабатывать несколько строк таблицы, даже если строка обновляется или вставляется, сравнивая комиссионные для записи с таким же номером. SQL не предназначен для выполнения подобной операции.

В данном случае можно использовать тщательно разработанное ограничение CHECK, если точно известны комиссионные для каждого из возможных городов. Например, можно определить ограничение таким образом:

```
CHECK ( (comm=.15 AND city='London')
    OR (comm=.14 AND city = 'Barselona)
    OR (comm=.11 AND city = 'San Jose')..)
```

Это лишь теоретическая идея. Вместо того, чтобы накладывать подобные ограничения, следует принять во внимание возможность определения представления (view) с предложением WITH CHECK OPTION, который имеет все эти ограничения в своем предикате. Пользователи могут осуществлять доступ к представлению, а не к таблице. Внесение изменений в ограничения не является трудным. WITH CHECK OPTION для представлений — это хорошая альтернатива для ограничений CHECK (см. главу 21).

Присвоение значений "по умолчанию"

Если строка вставляется в таблицу и не предоставляются значения для каждого поля, SQL должен иметь значения по умолчанию для заполнения ими значений полей, не заданных явно в команде; в противном случае команда вставки должна быть отвергнута. Наиболее распространенным значением по умолчанию является значение NULL. Это значение является значением по умолчанию для любого столбца, если для него не указано ограничение NOT NULL, либо не указано значение, присвоенное по умолчанию.

Назначение значений по умолчанию (DEFAULT) определяется, как и ограничения для столбца, командой CREATE TABLE, хотя значения DEFAULT не ограничивают множества значений, которые могут быть введены, а только определяют, что получается, если не указано никакое значение. Предположим, офис компании как и большинство ее продавцов (salespeople) базируются в Нью-Йорке (New York). Можно принять значение New York в качестве значения, присваиваемого по умолчанию, для таблицы Salespeople, чтобы сократить объем вводимых данных при добавлении строк в таблицу:

```
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) );
```

Ввод в таблицу значения New York становится элементарным всякий раз, когда назначается новый продавец; пропуск значений поля может стать автоматическим, даже если значение отлично от New York. Значение по умолчанию может быть желательным, если, например, номер, определяющий конкретный офис в таблице Orders, является длинным. Длинные числовые значения могут быть причиной ошибки. Если большинство заказов (все заказы) адресовано (адресованы) в этот конкретный офис, то предпочтительно определить его как значение, принимаемое по умолчанию.

Помимо NULL-значений, есть и другой способ использования значений по умолчанию. Поскольку NULL-значения представляют собой (по сути) отрицание любого значения, отличного от IS NULL, они имеют тенденцию исключения множества предикатов. Иногда требуется просмотреть значения пустых полей без выделения их каким-либо специальным образом. Можно определить специальное значение по умолчанию, например, пробел или нуль, которые реально меньше значения, назначаемого по умолчанию. Разница между ними и NULL-значением состоит в том, что SQL воспринимает их как и любое другое значение.

Предположим, покупателям первоначально не присвоены рейтинги. Но каждые шесть месяцев они назначаются для всех покупателей, включая тех, кому первоначально не были назначены. Если необходимо выбрать этих покупателей как группу, запрос, подобный приведенному ниже, исключит всех покупателей с рейтингами NULL:

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

Однако, если определено значение по умолчанию (DEFAULT) как 000 для поля rating, покупатели без рейтингов будут выбраны наряду с остальными. Какой из методов лучше зависит от ситуации: используется ли поле в запросах, нужно ли включать строки без значений в выходные данные запроса?

Другая характеристика значений по умолчанию этого типа состоит в том, что они позволяют объявлять поле в запросе NOT NULL. Если значения по умолчанию используются для исключения NULL-значений, то это может оказаться хорошей защитой от ошибок.

Можно также использовать с этим полем ограничения UNIQUE или PRIMARY KEY. Однако надо помнить, что только одна строка в единицу времени может иметь значение по умолчанию. Любая строка, его содержащая, будет обновляться до тех пор, пока не будет вставлена новая строка со значением по умолчанию. Поэтому ограничения UNIQUE и PRIMARY KEY (особенно последнее) обычно не размещаются в строках со значениями по умолчанию.

Итоги

В главе были рассмотрены способы контроля вводимых в таблицы значений. Ограничение NOT NULL можно использовать для исключения NULL-значений, UNIQUE — для поддержания уникальности одного или нескольких столбцов, PRIMARY KEY — для того, чтобы сделать все то же, что и с помощью UNIQUE, но с другой целью; CHECK — для определения собственного критерия, какие значения могут быть введены. Кроме того, можно использовать предложение DEFAULT для автоматической подстановки значения по умолчанию в любое поле, имя которого не указано в INSERT. Например, NULL-значения вставляются, когда предложение DEFAULT не представлено и нет ограничения NOT NULL.

Ограничения FOREIGN KEY или REFERENCES, о которых пойдет речь в главе 19, сходны с рассмотренными, за исключением того, что они относятся к группе из одного или более полей и, следовательно, воздействуют на значения, которые могут быть введены для этих групп.

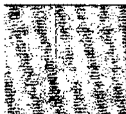
Работаем на SQL

1. Создайте таблицу Orders так, чтобы все значения столбца onum, как и все комбинации cnum и snum, были различными и NULL-значения не содержались в поле date.
2. Создайте таблицу Salespeople так, чтобы комиссионные по умолчанию составляли 10%, причем NULL-значения запрещены, поле snum является первичным ключом, а все имена расположены в алфавитном порядке между 'A' и 'M' включительно (предполагается, что все имена заданы большими буквами).
3. Создайте таблицу Orders, причем вы должны быть уверены в том, значение поля onum больше значения поля cnum, а значение поля cnum больше значения поля snum. Предположим, что NULL-значения недопустимы ни для одного из этих трех полей.

(Ответы см. в приложении A.)

19

Поддержка целостности данных



Ранее рассматривались определенные связи между некоторыми полями в таблицах. Например, поле snum таблицы Customers связывает поле snum в таблицах Salespeople и Orders. Этот тип связи называется ссылочной целостностью. И здесь мы обсудим некоторые вопросы ее использования.

В этой главе мы рассмотрим ссылочную целостность более детально, выявим ряд ограничений, которыми можно воспользоваться для управления ею и выясним, как эти ограничения поддерживаются при использовании команд обновления DML. Ограничения ссылочной целостности, подобно другим ограничениям, могут воздействовать на команды обновления, кроме того, они действуют и на другие таблицы, помимо тех, в которых расположены. Определенные функции запросов, например соединение, легко структурируются в терминах связей ссылочной целостности .

Внешние и родительские ключи

Когда все значения одного из полей таблицы должны быть представлены в поле другой таблицы, это означает,что поле *ссылается на (refers to)* или *является ссылкой (references)* на таблицу. Это свидетельствует о прямой связи между значениями двух полей. Например, каждый из покупателей таблицы Customers имеет поле snum, которое определяет продавца, назначенного ему и определенного в таблице Salespeople. Для каждой заявки в таблице Orders существует только один продавец и только один покупатель. Они определяются полями snum и cnum в таблице Orders.

Когда поле таблицы ссылается на другое поле (в другой таблице), оно называется *внешним ключом (foreign key)*; поле, на которое он ссылается, называется его *родительским ключом (parent key)*. Таким образом, поле snum в таблице Customers является внешним ключом, а поле snum, на которое он ссылается в таблице Salespeople — родительским ключом. Поля cnum и snum таблицы Orders также являются внешними ключами, ссылающимися на их родительские ключи с теми же именами в таблицах Customers и Salespeople. Имена внешнего и родительского ключей могут быть неодинаковыми; в данном случае это просто соглашение, принятое с целью сделать связи более понятными.

Внешние ключи из нескольких столбцов

Внешний ключ, как и первичный, может быть определен на любом количестве полей, которые все вместе рассматриваются как единое целое. Внешний ключ и родительский ключ, на который он ссылается, должны быть определены на одинаковом множестве полей (по количеству полей, типам полей и порядку следования полей). В простых таблицах, рассматриваемых здесь в качестве примера, используются только внешние ключи, определенные на единственном поле; возможно, они наиболее часто применяются и на практике. С целью упрощения речь пойдет о внешнем ключе как о единственном столбце. Далее, все, что будет говориться о поле, являющемся внешним

ключом, справедливо и для внешнего ключа, определенного на группе полей, если специально не оговаривается иное.

Смысл внешних и родительских ключей

Если поле является внешним ключом, то оно связано с таблицей, на которую ссылается, т.е. каждое значение в этом поле (внешнем ключе) непосредственно связано со значением в другом поле (родительском ключе). Каждое значение (каждая строка) внешнего ключа ссылается на одно значение (строку) родительского ключа. Это означает, что система находится в состоянии ссылочной целостности.

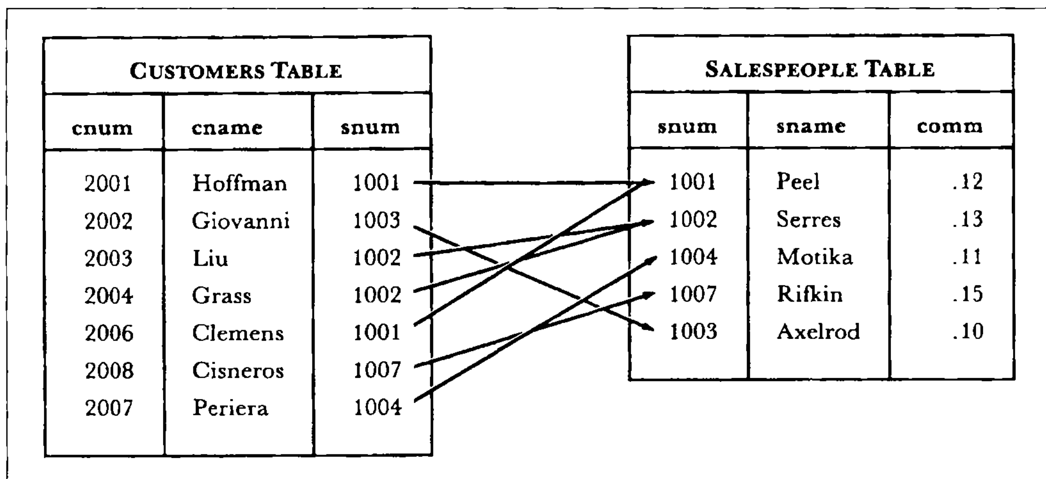
Внешний ключ snum в таблице Customers имеет значение 1001 для строк Hoffman и Clemens. Предположим, есть две строки строки в таблице Salesperson со значением поля snum = 1001. Как узнать, к какому именно из двух продавцов относятся покупатели Hoffman и Clemens? Кроме того, если бы в таблице Salespeople не было таких строк, то это значило бы, что Hoffman и Clemens назначены продавцу, которого не существует! Заключение очевидно: каждое значение внешнего ключа должно быть представлено в родительском ключе один и только один раз.

Из утверждения, что данное значение внешнего ключа может ссылаться только на одно значение родительского ключа, не следует, что верно и обратное: любое число внешних ключей может ссылаться на одно и то же значение родительского ключа. Это ясно из таблиц, рассматриваемых здесь в качестве примера. Hoffman и Clemens назначены Peel, значения их внешнего ключа соответствуют одному и тому же родительскому ключу. Значение внешнего ключа должно ссылаться только на единственное значение родительского ключа, но на это значение родительского ключа может ссылаться любое количество значений внешних ключей.

Например, значения внешнего ключа из таблицы Customers, соответствующие их родительским ключам в таблице Salespeople, показаны на рис. 19.1. Для большей наглядности рисунка опущены не относящиеся к делу поля.

Ограничения FOREIGN KEY (внешнего ключа)

SQL поддерживает ссылочную целостность с ограничением FOREIGN KEY. Это важное ограничение является новым в SQL, и поэтому еще не имеет универсальной поддержки. Более того, некоторые его программные реализации являются весьма громоздкими. Назначение FOREIGN KEY — ограничить вводимые в базу данных значения так, чтобы внешний ключ и его родительский ключ соответствовали принципам ссылочной целостности. Один из эффектов поддержки ограничения (внешнего ключа) — исключение значений для поля (полей) внешнего ключа, не представленных в данный момент в родительском ключе. Это ограничение относится и к возможности



CUSTOMERS TABLE

cnum	cname	snum
2001	Hoffman	1001
2002	Giovanni	1003
2003	Liu	1002
2004	Grass	1002
2006	Clemens	1001
2008	Cisneros	1007
2007	Periera	1004

SALESPEOPLE TABLE

snum	sname	comm
1001	Peel	.12
1002	Serres	.13
1004	Motika	.11
1007	Rifkin	.15
1003	Axelrod	.10

Рис. 19.1. Внешний ключ для таблицы Customers с родительским ключом

изменения или исключения значений родительского ключа (о чем речь пойдет позже в этой главе).

Как объявить поля внешним ключом

Ограничение FOREIGN KEY используется в команде CREATE TABLE (или в команде ALTER TABLE), содержащей поле, которое желательно объявить внешним ключом. Указывается имя родительского ключа, на который есть ссылка в ограничении внешнего ключа. Это ограничение, как и любое другое рассмотренное ранее, размещается в команде. Подобно большинству ограничений, им может быть таблица или множество столбцов, причем разрешается использовать множество полей как один внешний ключ.

FOREIGN KEY как ограничение на таблицу

Представляем синтаксис ограничения на таблицу FOREIGN KEY:

```
FOREIGN KEY <список столбцов> REFERENCES <имя таблицы>
   [<список столбцов>]
```

Первый список столбцов представляет собой заключенный в круглые скобки список из одного или более столбцов таблицы, перечисленных через запятую, создаваемых или изменяемых этой командой. Предложение REFERENCES задает имя таблицы, содержащей родительский ключ. Это может быть та же самая таблица, кото-

рая создается или изменяется в текущей команде (подробнее этот вариант рассматривается позже). Второй список столбцов представляет собой заключенный в круглые скобки список столбцов, которые формируют родительский ключ (имена столбцов указываются через запятую). Два списка столбцов должны быть сравнимы:

- Они должны иметь одинаковое количество столбцов.
- Первый, второй, третий и т.д. элемент списка столбцов внешнего ключа должны иметь тот же тип и размер, что и соответствующие элементы — первый, второй, третий и т.д. — списка родительского ключа. Столбцы в двух списках не обязаны иметь одинаковые имена, хотя в данных примерах имена совпадают; сделано это с целью обеспечения наглядности связей.

Перед вами определение таблицы Customers, в которой snum определено как внешний ключ, ссылающийся на таблицу Salespeople:

```
CREATE TABLE Customers
   (cnum     integer NOT NULL PRIMARY KEY,
   cname     char(10),
   city      char(10),
   snum      integer,
FOREIGN KEY (snum) REFERENCES Salespeople (snum) );
```

Важный момент, который следует помнить при использовании ALTER TABLE вместо CREATE TABLE для определения ограничений FOREIGN KEY, заключается в том, что значения, присутствующие в настоящий момент во внешнем и родительском ключах, должны находиться в состоянии ссылочной целостности. В противном случае команда будет отвергнута. Хотя ALTER TABLE и полезна, так как обеспечивает адаптируемость, структурные принципы, подобные ссылочной целостности, следует внедрять в систему на этапе проектирования базы данных.

FOREIGN KEY как ограничение на столбец

Версия ограничения FOREIGN KEY, как ограничения на столбец, называется ограничением REFERENCES, поскольку реально не содержит слов FOREIGN KEY, а содержит слово REFERENCES, после которого следуют имена для родительского ключа. Например:

```
CREATE TABLE Customers
   (cnum     integer NOT NULL PRIMARY KEY,
   cname     char(10),
   city      char(10),
   snum      integer REFERENCES Salespeople (snum) );
```

Команда определяет Customers.snum как внешний ключ, родительским ключом которого является Salespeople.snum. Это эквивалентно следующему ограничению на таблицу:

```
FOREIGN KEY (snum) REFERENCES Salespeople (snum)
```

Пропуск списков столбцов первичного ключа

Используя ограничение на таблицу или на столбец FOREIGN KEY, можно опустить список столбцов родительского ключа, если он имеет ограничение PRIMARY KEY. В случае использования ключей, определенных на множестве полей, порядок перечисления столбцов во внешнем и первичном ключах должен быть соответствующим, и, в любом случае, принцип совместимости между двумя ключами должен поддерживаться. Например, если разместить ограничение PRIMARY KEY на поле snum в таблице Salespeople, то можно использовать его как внешний ключ в таблице Customers (аналогично тому, как это было сделано в предыдущем примере) следующим образом:

```
CREATE TABLE Customers
   (cnum     integer NOT NULL PRIMARY KEY,
   cname     char (10),
   city      char (10),
   snum     integer REFERENCES Salespeople);
```

Средство встроено в язык с целью использования первичных ключей в качестве родительских. Далее будет кратко изложена лежащая в его основе логика.

Как ссылочная целостность ограничивает значения родительского ключа

Управление ссылочной целостностью накладывает ряд ограничений на значения, которые могут быть представлены в полях, объявленных внешним и родительским. Родительский ключ должен быть структурирован так, чтобы была уверенность, что каждое значение внешнего ключа соответствует отдельной строке. Это означает, что оно должно быть уникальным и не должно содержать NULL-значений. Недостаточно, чтобы родительский ключ полностью удовлетворял этим требованиям только в тот момент, когда объявляется внешний ключ. SQL должен гарантировать, что повторяющиеся или NULL-значения не будут введены в родительский ключ. Следовательно, необходимо иметь уверенность, что все поля, которые используются как родительские ключи, имеют либо ограничение PRIMARY KEY, либо ограничение UNIQUE, а также ограничение NOT NULL.

Сравнение PRIMARY с UNIQUE для родительских ключей

При работе с базами данных существенно, что внешние ключи ссылаются только на первичные ключи. Используя внешние ключи, возможно связать их не просто с родительскими ключами, на которые они ссылаются, а с отдельной строкой таблицы, в которой найден родительский ключ. Сам по себе он не дает никакой информации, которая не была бы уже представлена внешним ключом.

Например, значимость поля snum как внешнего ключа таблицы Customers, определяется связью, которую оно обеспечивает не с самим значением поля snum, на которое ссылается, а с другой информацией таблицы Salespeople, такой как имена продавцов, их расположение и т.д. Внешний ключ не просто связь между двумя одинаковыми значениями, а взаимодействие между двумя целыми строками рассматриваемых таблиц, основанное на совпадении двух значений. Поле snum можно использовать для связи любой информации в строке из таблицы Customers с информацией из связанной с ней строки таблицы Salespeople, например: живут ли продавцы и покупатели в одном городе, кто из них имеет более длинное имя, имеет ли продавец, обслуживающий данного покупателя, других покупателей и т.д.

Наиболее логично выбрать первичный ключ в качестве внешнего, поскольку назначение его состоит в уникальной идентификации строки. Для любого внешнего ключа, использующего уникальный ключ как родительский, нужно уметь создать внешний, применяющий первичный ключ той же таблицы с таким же эффектом. Имея внешний ключ, у которого нет никакого другого назначения кроме связывания строк, и первичный ключ, который не имеет никакого другого назначения кроме идентифицирования строк, легко сохранить структуру базы данных ясной и простой и уменьшить вероятность появления проблем.

Ограничения внешнего ключа

Внешний ключ может содержать только реально представленные значения или NULL-значения. Попытки ввести в этот ключ другие значения будут отвергнуты. Объявлять внешние ключи как NOT NULL нет необходимости, а во многих случаях и нежелательно. Предположим, при вводе сведений о покупателе еще неизвестно, кто из продавцов назначен для его обслуживания. Наилучший способ выйти из этой ситуации состоит во вводе NULL-значений, которые позднее следует заменить на реальные значения.

Что происходит при выполнении команды обновления

Предположим, все внешние ключи, встроенные в таблицы, объявлены, и для них определены ограничения FOREIGN KEY 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
   (onum     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) );
```

Использование определений таблицы

Определения таблиц имеют некоторые свойства, требующие обсуждения. Поля cnum и snum таблицы Orders выбирались в качестве одного внешнего ключа. Следовательно, вы могли быть уверены, что заявка каждого покупателя адресована именно тому продавцу, который указан в таблице Customers. Чтобы создать внешний ключ, нужно

сделать ограничение таблицы UNIQUE для двух полей таблицы Customers, несмотря на то, что оно не является необходимым для самой этой таблицы. Поскольку поле cnum в таблице имеет ограничение PRIMARY KEY, оно будет уникальным в любом случае, и, следовательно, невозможно получить неуникальную комбинацию поля cnum с любым другим полем.

При таком определении внешнего ключа поддерживается целостность базы данных, но теряется возможность сделать исключения относительно связей между продавцами и покупателем. Гарантия от ошибок часто означает устранение возможности делать какие-либо исключения, и решение об их необходимости должно приниматься на уровне управления, а не базы данных. С точки зрения управления целостностью базы данных исключения нежелательны. Но если вы хотите их разрешить и сохранить при этом целостность, можно объявить snum и cnum в таблице Orders независимыми внешними ключами одноименных полей в таблицах Salespeople и Customers соответственно.

Использование поля snum в таблице Orders не является необходимым, хотя полезно с точки зрения рассмотрения примеров. Поле cnum связывает каждый заказ с покупателем в таблице Customers, а таблицы Orders и Customers всегда могут быть связаны для поиска правильного значения snum для данного заказа (если предположить, что никакие исключения не разрешены). Часть информации — какой покупатель какому продавцу назначен — получена дважды, и можно выполнить ряд операций с целью удостовериться в том, что две версии согласуются. Если бы не было ограничений внешнего ключа, ситуация была бы проблематичной, поскольку каждая заявка должна была бы выбираться вручную (с помощью запроса), чтобы убедиться в том, что соответствующий продавец связан с соответствующим заказом. Наличие такого рода избыточной информации в базе данных называется *денормализацией (denormalization)* и является нежелательной в идеальной реляционной базе данных, хотя в практических ситуациях могут быть причины для ее разрешения. Денормализация дает возможность более быстрого выполнения ряда запросов, поскольку запрос к единственной таблице выполняется гораздо быстрее, чем соединение.

Действие ограничений

Как ограничения влияют на то, что можно и чего нельзя делать с помощью команд обновления DML? Что касается полей, определенных в качестве внешнего ключа, ответ более или менее ясен: любые значения, которые вводятся в них с помощью команды INSERT или UPDATE, должны быть уже представлены в их родительских ключах. Можно вводить NULL-значения в эти поля несмотря на то, что NULL-значения не разрешены в родительских ключах, если они не имеют NOT NULL-ограничений. Можно удалить (выполнить команду DELETE) любую строку с внешним ключом, не оказывая никакого действия на родительские ключи.

Что касается изменений значений родительского ключа, то ответ, определенный ANSI, прост, но слишком ограничен: любое значение родительского ключа, на который есть ссылка внешнего ключа, не может быть удалено или изменено. Это означает, что нельзя удалить покупателя из таблицы Customers, пока он имеет заказы в таблице

Orders. В зависимости от использования таблиц, это может оказаться либо желательно, либо хлопотно. Но намного предпочтительнее вариант, при котором было бы разрешено исключить покупателя, имеющего текущие заказы, и оставить таблицу Customers, ссылающейся на несуществующих покупателей.

Из этого следует, что создатель таблицы Orders, используя таблицы Customers и Salespeople как родительские ключи, вводит значительные ограничения на их использования. По этой причине нельзя использовать таблицу в качестве родительского ключа невладельцу таблицы, хотя владелец этой таблицы может предоставить такое право.

Существует несколько возможных способов изменения родительского ключа, не являющихся частью ANSI, их можно найти в некоторых коммерческих продуктах. Для изменения или исключения значений родительского ключа, на который имеется ссылка, есть три основные возможности:

- Можно ограничить или запретить изменение (согласно ANSI), что означает, что изменение родительского ключа *ограничено (restricted)*.

- Можно изменить родительский ключ и автоматически получить изменения внешнего ключа; это означает, что изменения выполняются *каскадно (cascades)*.

- Можно изменить родительский ключ, и тем самым установить значения внешнего ключа в NULL автоматически (предполагается, что NULL-значения разрешены во внешнем ключе); в этом случае говорят, что внешний ключ изменяется в *null*.

Нежелательно, чтобы все команды обновления в каждом из этих трех случаев выполнялись одинаково. На INSERT это, конечно, не распространяется. Если она вводит новые значения родительского ключа в таблицу, значит ни на одно из этих значений еще нет ссылок. Может понадобиться разрешить выполнение изменений каскадом, а не удалений или наоборот. Тогда лучше всего иметь возможность определять любую из трех операций независимо для команд UPDATE и DELETE. Следовательно, надо ссылаться на *эффекты обновления (update effects) и эффекты удаления (delete effects)*, которые фиксируют, что происходит при выполнении команд UPDATE или DELETE для родительского ключа. Эти эффекты, упоминавшиеся ранее, определяются ключевыми словами RESTRICTED, CASCADES, NULLS.

Реальные возможности данной системы распространяются от ограничений ANSI стандарта — эффекты обновления и удаления автоматически ограничены — до более идеальной ситуации, рассмотренной ранее. Приведем несколько примеров того, что дают эффекты обновления и удаления. Стандартный синтаксис обновления и удаления отсутствует, поскольку речь идет о нестандартных характеристиках. Синтаксис, используемый здесь, весьма прост и служит для иллюстрации действия этих эффектов.

Предположим, есть причина для изменения значения поля snum таблицы Salespeople по случаю, например, изменения районов действия продавцов. (Изменять заведенный порядок назначения первичных ключей на практике не рекомендуется.

Один из доводов в пользу этого: они не должны изменяться.) При изменении номера продавца или при удалении сведений о нем из базы данных необходимо сохранить связь со всеми его покупателями или сохранить всех его покупателей, более того, удостовериться, что им назначен новый продавец. Для этого следует определить для UPDATE эффект CASCADES, а для DELETE — эффект RESTRICTED.

```
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 попытаться удалить Peel из таблицы Salespeople, то команда не будет выполнена, пока значения поля snum для покупателей Hoffman и Clemens не будут изменены таким образом, чтобы они обслуживались другим продавцом. С другой стороны, можно изменить значение поля snum для Peel на 1009, при этом соответствующие поля для Hoffman и Clemens изменятся автоматически.

Третий эффект заключается в NULL-значениях. Когда продавцы уходят из компании, их текущие заказы остаются без адресатов. С другой стороны, необходимо автоматически отменить все заказы тех покупателей, обслуживание которых прекращено. Изменения номеров продавца или покупателя могут просто передаваться для выполнения. Создается таблица Orders, чтобы предусмотреть все перечисленные действия:

```
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);
```

Для того, чтобы в DELETE отрабатывалось действие NULLS для таблицы Salespeople, следует удалить ограничение NOT NULL для поля snum.

Внешние ключи, ссылающиеся на те самые таблицы, в которых они определены

Как упоминалось ранее, ограничение FOREIGN KEY может содержать имя своей собственной таблицы в качестве имени родительской таблицы. Такая черта на практике является весьма полезной. Предположим, есть таблица Employees с полем, названным "manager". Это поле содержит номер менеджера служащего, а поскольку каждый менеджер является и служащим, он также представлен в этой таблице. Создадим таблицу, в которой поле empno (employee number — номер служащего) объявлено первичным ключом, а поле manager (менеджер, управляющий) — внешним ключом, ссылающимся на него:

```
CREATE TABLE Employees
   (empno    integer NOT NULL PRIMARY KEY,
   name      char (10) NOT NULL UNIQUE,
   manager   integer REFERENCES Employees);
```

(Поскольку внешний ключ ссылается на первичный ключ таблицы, перечисление атрибутов столбца можно опустить.) Таблица может содержать, например, такие данные:

EMPNO	NAME	MANAGER
1003	Terrence	2007
2007	Atali	NULL
1688	McKenna	1003
2002	Collier	2007

Каждый служащий в этой таблице, кроме Atali, ссылается на другого служащего как на своего менеджера. Atali, занимающий высший пост в фирме, должен иметь в поле manager значение NULL. Здесь проявляется еще один принцип ссылочной целостности. Внешний ключ, ссылающийся на свою собственную таблицу, должен допускать NULL-значения.

Этот принцип должен соблюдаться даже в том случае, когда внешний ключ ссылается на собственную таблицу опосредованно (не непосредственно), т.е. ссылается на другую таблицу, которая, в свою очередь, ссылается на ту таблицу, в которой был определен внешний ключ. Например, таблица Salespeople содержит дополнительное поле, которое ссылается на таблицу Customers, таким образом таблицы ссылаются друг на друга; этот факт отражен в следующем предложении CREATE TABLE:

```
CREATE TABLE Salespeople
   (snum     integer NOT NULL PRIMARY KEY,
   sname     char(10) NOT NULL,
   city      char(10),
   comm      decimal,
   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 *цикличностью (circularity)* или *перекрестной ссылкой (cross referencing)*. SQL поддерживает эту процедуру теоретически, но на практике могут возникнуть проблемы. Например, какая бы из этих двух таблиц ни создавалась, первая будет ссылаться на таблицу, которой еще нет. В интересах поддержки цикличности SQL действительно допускает такую ситуацию, но ни одну из этих таблиц нельзя использовать, пока не созданы обе. С другой стороны, если эти две таблицы созданы различными пользователями, проблема становится еще сложнее. Цикличность может быть полезным инструментом, но здесь не обойтись без неоднозначных толкований и риска. В частности, предыдущий пример не очень выразителен: он ограничивает продавца одним покупателем, и здесь можно обойтись без цикличности. Цикличность необходимо использовать осторожно, детально изучив, как данная система выполняет действия обновления и удаления, а также поддержку привилегий и обработку транзакций; все это должно предшествовать созданию цикличной системы, поддерживающей ссылочную целостность.

Итоги

Основная идея ссылочной целостности заключается в том, что все значения внешних ключей отсылают к определенной строке родительского ключа. Это означает, что каждое значение внешнего ключа должно быть представлено в родительском ключе один и только один раз. Как только значение размещается во внешнем ключе, родительский ключ проверяется для того, чтобы убедиться в том, что такое значение в нем присутствует; в противном случае команда отвергается. Родительский ключ должен иметь ограничение PRIMARY KEY или UNIQUE для гарантии того, что значение не представлено более чем один раз. Попытки изменить значение родительского ключа на значение, присутствующее во внешнем ключе, отвергаются. Система, однако, может позволять иметь в качестве значения внешнего ключа NULL-значение или новое значение родительского ключа, а также задавать каждый из этих вариантов независимо для команд UPDATE и DELETE. На этом описание команды CREATE TABLE завершается. Далее будут рассмотрены другие команды типа CREATE.

Из главы 20 вы узнаете о представлениях — объектах базы данных, которые выглядят и функционируют как таблицы, но по существу являются результатами запросов. Некоторые из ограничений присущи исключительно представлениям, и, ознакомившись с содержанием следующих трех глав, вы сможете более четко формулировать ограничения в соответствии с потребностями.

Работаем на SQL

1. Создайте таблицу с именем Cityorders. В ней должны быть поля onum, amt, snum, как и в таблице Orders, и поля cnum и city, как в таблице Customers, при этом каждый заказ покупателя должен храниться вместе с названием его города. Поле onum должно быть первичным ключом таблицы Cityorders. Все поля таблицы Cityorders должны быть ограничены соответствующими строками из таблиц Customers и Orders. Предполагается, что родительские ключи этих таблиц уже имеют соответствующие ограничения.

2. Расширим проблему. Переопределим таблицу Orders таким образом: добавим новый столбец с именем prev, который идентифицирует onum предыдущего заказа для текущего пользователя. Реализуйте это как внешний ключ, ссылающийся на ту таблицу, в которой он определен (таблицу Orders). Внешний ключ должен ссылаться на поле cnum покупателя, обеспечивая связь между текущим заказом и тем, на который он ссылается.

(Ответы см. в приложении А.)

20

Введение в представления

Представление (view) — объект, который не содержит собственных данных. Это своего рода таблица, содержимое которой берется из других таблиц посредством выполнения запроса. Поскольку значения в таблицах меняются, это автоматически приводит к соответствующим изменениям в представлениях.

Из этой главы вы узнаете, что такое "представления", как они создаются, и кое-что об их ограничениях и защите. Использование представлений базируется на возможностях расширенных запросов, таких как соединения и подзапросы. Они будут рассмотрены более подробно, поскольку есть ряд существенных деталей, касающихся связи запросов и представлений.

Что такое представления?

Таблицы, о которых шла речь до сих пор, будут теперь называться *базовыми таблицами (base tables).* Это таблицы, которые содержат реальные данные. Существует и другой вид таблиц, получивший название "представления" (views). *Представления* — это таблицы, содержимое которых берется или выводится из других таблиц. Они используются в запросах и в предложениях DML так же, как и обычные таблицы, но они не содержат своих собственных данных. Представления подобны окнам, через которые просматривается информация, реально хранимая (содержащаяся) в базовых таблицах. В действительности же это запросы, выполняемые всякий раз, когда представление является объектом команды. В этот момент выходные данные запроса и становятся содержимым представления.

Команда CREATE VIEW

Представление определяется с помощью команды CREATE VIEW, состоящей из ключевых слов CREATE VIEW (создать представление), имени создаваемого представления и ключевого слова AS, после которого следует запрос.

Например:

```
CREATE VIEW Londonstaff
   AS SELECT *
      FROM Salespeople
      WHERE city = 'London';
```

В результате выполнения этой команды можно стать владельцем представления, называемого Londonstaff. Как и любую другую таблицу представление можно использовать: формулировать к нему запросы, выполнять обновление, вставку, удаление и соединение с другими таблицами и представлениями. Давайте сформулируем запрос для просмотра представления (см. рис. 20.1):

SQL Execution Log

SELECT * FROM Londonstaff;

snum	sname	city	comm
1001	Peel	London	0.1200
1004	Motika	London	0.1100

Browse : ↑↓←→ PgDn PgUp ▶| |◀ Home

Рис. 20.1. Представление Londonstaff

```
SELECT *
    FROM Londonstaff;
```

При команде выбрать все строки из представления выполняется запрос, содержащийся в определении Londonstaff, и возвращаются все его выходные данные. Если в запросе для представления указан предикат, то представление содержит только строки, вошедшие в состав выходных данных (удовлетворяющие заданному предикату).

Преимущество использования представления вместо базовой таблицы состоит в том, что оно обновляется автоматически при изменении формирующих его таблиц. Содержимое представления не фиксируется, а повторно вычисляется всякий раз, когда вы ссылаетесь на представление в команде. Если завтра добавится еще один продавец, находящийся в Лондоне (city = 'London'), то он автоматически попадет в представление.

Представления значительно расширяют возможности управления данными. Они предоставляют пользователям доступ не ко всей информации, хранящейся в таблице, а только к ее части. Если нужно, чтобы каждый продавец мог просмотреть таблицу Salespeople, но не мог видеть значения комиссионных, можно создать представление с помощью следующего предложения (результат представлен на рис. 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

Browse : ↑↓←→ PgDn PgUp ▶| |◀ Home

Рис. 20.2. Представление Salesown

Другими словами, это представление есть то же самое, что и таблица Salespeople, за исключением столбца comm, имя которого не указано в запросе и, следовательно, не включено в представление.

Обновление представлений

Это представление можно модифицировать командами обновления DML, но модификации воздействуют не на само представление, а только на лежащую в его основе таблицу:

```
UPDATE Salespeople
    SET city = 'Palo Alto'
    WHERE snum = 1004;
```

Действие команды аналогично выполнению этой же команды для таблицы Salespeople. Однако, если продавец попытается изменить величину своих комиссионных с помощью команды UPDATE:

```
UPDATE Salesown
    SET comm = .20
    WHERE snum = 1004;
```

то она будет отвергнута, поскольку в представлении нет поля comm. Важно заметить, что не все представления могут обновляться. Об этом пойдет речь в главе 21.

Именование столбцов

До сих пор в качестве имен полей представлений использовались непосредственно имена полей таблицы, лежащей в основе представления. На начальном этапе работы с представлениями это допустимо. Однако иногда требуется задать новые имена для столбцов:

- Когда некоторые столбцы являются выходными столбцами и, следовательно, не поименованы.
- Когда два или более столбцов в соединении имеют одинаковые имена в соответствующих таблицах.

Имена, которые станут именами полей, даются в круглых скобках после имени таблицы. Это делать необязательно, если они соответствуют именам полей таблицы, участвующей в запросе. Типы данных и размеры выводятся из полей запроса. Часто не требуется задавать новые имена полей, но при необходимости это можно сделать для каждого поля в представлении.

Комбинирование предикатов представлений и запросов, основанных на представлениях

Запрос для представления — это запрос к запросу. Основной метод состоит в комбинировании предикатов двух запросов в один. Можно посмотреть еще раз на представление с именем Londonstaff:

```
CREATE VIEW Londonstaff
   AS SELECT *
      FROM Salespeople
         WHERE city = 'London';
```

Если для этого представления сформулировать следующий запрос:

```
SELECT *
   FROM Londonstaff
   WHERE comm > .12;
```

то он выполняется так, как если бы был сформулирован для таблицы Salespeople:

```
SELECT *
   FROM Salespeople
   WHERE city = 'London'
```

```
    AND comm > .12;
```

Это может вызвать ряд проблем, касающихся представлений. Можно без особых проблем комбинировать два однотипных предиката и получить предикат, который работать не будет. Например, создается (CREATE) следующее представление:

```
CREATE VIEW Ratingcount (rating, number)
   AS SELECT rating, COUNT (*)
      FROM Customers
      GROUP BY rating;
```

В соответствии с запросом представления получается некоторое число покупателей для каждого уровня рейтинга. Затем для этого представления можно сформулировать запрос с целью выяснить, назначен ли какой-нибудь рейтинг для группы из трех покупателей:

```
SELECT *
   FROM Ratingcount
   WHERE number = 3;
```

Если скомбинировать два предиката, получится:

```
SELECT rating, COUNT (*)
   FROM Customers
   WHERE COUNT (*) = 3
   GROUP BY rating;
```

Это неверный запрос. Функции агрегирования, одной из которых является COUNT, нельзя использовать в предикате.
Правильный способ сформировать рассмотренный выше запрос конечно же такой:

```
SELECT rating, COUNT (*)
   FROM Customers
   GROUP BY rating;
   HAVING COUNT (*) = 3;
```

Но SQL не выполнит такое преобразование. Будет ли эквивалентный запрос о Ratingcount ошибочным? Возможно. Это область SQL, в которой техника выполнения представлений может значительно воздействовать на результаты. Наилучший способ разрешения этой проблемы, если она не отражена в документации по системе, — выполнить и проверить результат. Если команда восприметея, то можно будет использовать представления для того, чтобы получить нечто вроде ограничений SQL на синтаксис запросов.

Групповые представления

Групповые представления (grouped views) — это представления, подобные Ratingcount в предыдущем примере, которые содержат предложение GROUP BY или базируются на других групповых представлениях.

Групповые представления могут быть прекрасным способом непрерывной обработки производной информации. Предположим, что каждый день нужно отслеживать количество покупателей, имеющих заказы, количество продавцов, получивших заказы, количество заказов, среднее количество заказов и общее количество поступивших заказов. Вместо того, чтобы многократно конструировать сложный запрос, можно 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 можно получить всю необходимую информацию с помощью единственного запроса:

```
SELECT *
   FROM Totalforday;
```

Запросы SQL могут быть достаточно сложными, и, следовательно, представления являются гибким и мощным средством определения того, как будут использоваться данные. Они приносят пользу, переформатируя данные необходимым образом и исключая многочисленные повторения.

Представления и соединения

Представления не обязательно выводятся из единственной базовой таблицы, они могут получать информацию из любого количества базовых таблиц или других представлений. Например, можно определить представление, которое показывает для каждого заказа имена продавца и покупателя:

```
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 можно выбрать все заказы покупателя или продавца, либо увидеть эту информацию для любого заказа. Например, для того чтобы увидеть все заказы продавца, нужно ввести следующий запрос (выходные данные представлены на рис. 20.3):

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

Browse : ↑↓←→ PgDn PgUp ▶| |◀ Home

Рис. 20.3. Заказы для продавца Rifkin, представленные в таблице Nameorders

```
SELECT *
   FROM Nameorders
   WHERE sname = 'Rifkin';
```

Представления можно соединять с другими таблицами, как с базовыми, так и с представлениями; таким образом можно увидеть все заказы Axelrod и ее комиссионные:

```
SELECT a.sname, cname, amt * comm
   FROM Nameorders a, Salespeople b
   WHERE a.sname = 'Axelrod'
      AND b.snum = a.snum;
```

Выходные данные для запроса представлены на рис. 20.4.

В предикате можно было указать WHERE a.sname = 'Axelrod' AND b.sname = 'Axelrod', но здесь использовался предикат более общего вида. Чтобы адаптировать запрос для любого человека, нужно просто указать его имя вместо Axelrod. Кроме того, snum является первичным ключом для таблицы Salespeople, и, следовательно, по определению является уникальным. Если бы было два человека с именем Axelrod, то версия с указанием поля name комбинировала бы их данные. Предпочтительнее является вариант с использованием поля snum, который позволяет различать их.

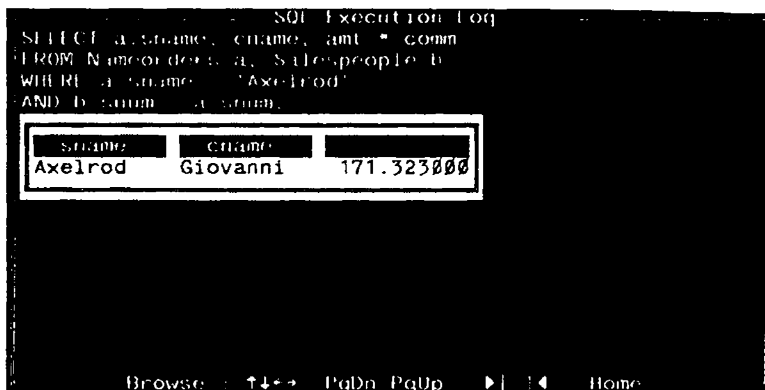


Рис. 20.4. Соединение базовой таблицы и представления

Представления и подзапросы

Представления могут также использовать подзапросы, включая и связанные подзапросы. Допустим, что компания платит вознаграждение продавцу, который имеет покупателя с наибольшим количеством заказов на заданную дату. Можно получить информацию с помощью следующего 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 вознаграждение получает только тот продавец, который имел наибольшее количество заказов по крайней мере 10 раз, то можно получить эту информацию с помощью другого представления, основанного на первом:

```
CREATE VIEW Bonus
  AS SELECT DISTINCT snum, sname
```

```
FROM Elitesalesforce a
WHERE 10 <=
   (SELECT-COUNT (*)
        FROM Elitesalesforce b
        WHERE a.snum = b.snum);
```

Извлечь из этой таблицы продавца, получившего вознаграждение, можно, введя следующий запрос:

```
SELECT *
  FROM Bonus;
```

Для того чтобы извлечь такую информацию с использованием RPG или COBOL, потребовалась бы довольно большая программа. В SQL мы использовали для этого две сравнительно сложные команды, которые хранятся как определение представления и один простой запрос. Все, что нужно делать ежедневно, — это выполнять последний элементарный запрос, поскольку информация, которую он извлекает, отражает текущее состояние базы данных.

Что нельзя сделать с помощью представлений?

Существует много типов представлений (включая примеры, рассмотренные в этой главе), которые выполняют только чтение. Это значит, что для них могут быть сформулированы запросы, но их нельзя использовать в качестве объекта для команд обновления (эта тема рассматривается подробнее в главе 21).

Существует несколько аспектов в области запросов, которые не укладываются в рамки определений представления: единственное представление должно базироваться на единственном запросе; UNION и UNION ALL недопустимы. ORDER BY тоже нельзя использовать в определении представления. Выходные данные для запроса, формирующего представление, должны быть неупорядоченными по определению.

Удаление представлений

Синтаксис исключения представления из базы данных сходен с синтаксисом для исключения базовых таблиц:

```
DROP VIEW <имя представления>
```

Здесь нет необходимости сначала удалить все содержимое, как это требовалось для базовых таблиц, поскольку содержимое представления никогда явно не определяется, а сохраняется в процессе выполнения отдельной команды. На базовые таблицы, на основе которых определено представление, команда никакого действия не оказывает. Надо помнить, что только владелец имеет полномочия.

Итоги

Теперь вы можете использовать представления. Почти все, что можно создать спонтанно с помощью запроса, определяется в виде постоянного представления. Запросы для этих представлений являются, по сути, запросами к запросам. Представления широко применяются на практике для удобства, для поддержания защиты, для форматирования и вывода значений и даже для изменения содержимого базы данных. Еще один важный вопрос, касающийся представлений, — их обновляемость, которая по мере обновления базовых таблиц возможна не во всех случаях. Подробнее мы поговорим об этом в главе 21.

Работаем на SQL

1. Создайте представление, показывающее всех покупателей с наивысшими рейтингами.
2. Создайте таблицу, которая показывает количество продавцов в каждом городе.
3. Создайте представление, которое показывает общее и среднее количество заявок для каждого продавца после его имени. Предполагается, что все имена являются уникальными.
4. Создайте представление, которое показывает каждого продавца вместе со всеми его покупателями.

(Ответы см. в приложении А.)

21

Изменение значений с помощью представлений

В этой главе речь идет о командах обновления DML — INSERT, UPDATE и DELETE — применительно к представлениям. Как упоминалось ранее, использование команд обновления в представлениях является опосредованным способом применения их для таблиц, на которые ссылаются запросы, соответствующие представлениям. Не все представления можно обновлять. Вы ознакомитесь с правилами, в соответствии с которыми можно определить, подлежит ли представление обновлению, а также с вопросами их практического применения. Вы научитесь употреблять предложение WITH CHECK OPTION, управляющее специфическими значениями, которые можно ввести в таблицу, используя представление. В некоторых случаях это может оказаться желательной альтернативой непосредственным ограничениям таблицы.

Обновление представлений

Одним из наиболее сложных и неоднозначных аспектов представлений является их применение в командах обновления DML. Эти команды непосредственно воздействуют на базовую таблицу представления, что создает своего рода противоречие. Представление состоит из результатов запроса, и при его обновлении обновляется все множество результатов запросов. Оно воздействует на значения в таблице (в таблицах), для которых выполняется запрос, и, следовательно, изменяет его выходные данные, не воздействуя на запрос как таковой. Следующее предложение создаст представление, показанное на рис. 21.1:

```
CREATE VIEW Citymatch (custcity, salescity)
   AS SELECT DISTINCT a.city, b.city
      FROM Customers a, Salespeople b
      WHERE a.snum = b.snum;
```

Представление показывает все пары покупателей с продавцами или, по крайней мере, одного покупателя в городе custcity, который обслуживается продавцом из города salescity.

Например, одна строка этой таблицы — London London — определяет, что имеется, по крайней мере, один покупатель в Лондоне, которого обслуживает продавец из Лондона. Эта строка относится к паре Hoffman с его продавцом Peel, оба они из Лондона. Однако, такое же значение получается и для пары Clemens, тоже находящейся в Лондоне со своим продавцом, которым опять же является Peel. Поскольку в запросе указано ключевое слово DISTINCT, в состав выходных данных включается только одна составная строка с этими значениями. Но какую пару из лежащих в основе таблиц она представляет?

Даже если не использовать ключевое слово DISTINCT, ответ на вопрос не будет получен, поскольку в представлении были бы две абсолютно одинаковые строки, и нельзя узнать, какая строка представления из какой реальной строки базовой таблицы

SQL Execution Log

```
SELECT *
FROM Citymatch;
```

custcity	salescity
Berlin	San Jose
London	London
Rome	London
Rome	New York
San Jose	Barcelona
San Jose	San Jose

Browse : ↑↓←→ PgDn PgUp ▶| |◀ Home

Рис. 21.1. Представление Citymatch

была получена (надо помнить, что запросы, в которых опущено предложение ORDER BY, генерируют выходные данные в произвольном порядке. Это справедливо также и для запросов, применяемых в представлениях; в таких запросах нельзя использовать ключевое слово ORDER BY. Поэтому порядок двух строк нельзя использовать для того, чтобы различить их). Это означает, что выходные строки нельзя однозначно связать со строками из таблиц, участвующих в запросе.

Что произойдет, если попытаться удалить строку London London из представления? Приведет это к удалению из таблицы Customers имен Hoffman или Clemens, или удалению обоих? Удалит ли SQL Peel из таблицы Salespeople? На эти вопросы невозможно ответить вполне определенно, поскольку удаления не разрешены для такого рода представлений. Представление Citymatch является примером представления только для чтения, в него нельзя вносить какие-либо изменения.

Определение обновляемости представления

Если команды обновления можно применить к представлению, то говорят, что представление является обновляемым (updatable); в противном случае оно является только читаемым (read-only). В соответствии с этой терминологией здесь и далее будет использоваться выражение "обновляемое представление" ("updating a view"), которое означает, что к представлению применимы все три команды обновления DML (INSERT, UPDATE и DELETE), которые могут изменять его значения.

Как определить, является ли представление обновляемым? В теории баз данных по этому поводу идут активные дебаты. Основной принцип заключается в том, что обновляемым представлением является то, для которого можно выполнить команду обновления, изменяя одну и только одну строку таблицы, лежащей в основе представления, в конкретный момент времени без какого-либо воздействия на другие строки какой-либо другой таблицы. Но проведение этого принципа в практику может быть затруднено. Более того, некоторые представления, которые являются обновляемыми в теории, могут оказаться реально необновляемыми в SQL. Приведем критерии того, является ли представление обновляемым в SQL:

- Оно должно базироваться на одной и только одной таблице.
- Оно должно включать первичный ключ таблицы (это не является ограничением ANSI стандарта, но его лучше придерживаться).
- Оно не должно содержать полей, полученных в результате применения функций агрегирования.
- Оно не может содержать спецификации DISTINCT в своем определении.
- Оно не должно использовать GROUP BY или HAVING в своем определении.
- Оно не должно использовать подзапросы (это ограничение ANSI, но в некоторых программных продуктах его не придерживаются).
- Оно может быть определено на другом представлении, но это представление должно быть также обновляемым.
- Оно не может содержать константы, строки или выражения (например, comm * 100) в списке выбираемых выходных полей.
- Для INSERT оно должно включать любые поля из лежащей в основе представления таблицы, которые имеют ограничение NOT NULL, хотя в качестве значения по умолчанию может быть указано другое значение.

Обновляемые представления в сравнении с представлениями "только для чтения"

Из этих ограничений следует, что обновляемые представления являются, по сути, окнами, через которые смотрят на таблицы, лежащие в основе представлений. Они показывают часть, но вовсе не обязательно все содержимое таблицы. Они могут быть ограничены определенными строками (при использовании предикатов) и указанными именами столбцов, но при этом представляют непосредственные значения, а не производную информацию из них, как это происходит при использовании функций агреги-

рования и выражений. Они также не выполняют сравнения строк друг с другом (как это можно сделать в соединениях и запросах с помощью ключевого слова DISTINCT).

Различия между обновляемыми представлениями и представлениями "только для чтения" не просто случайны. Цели их применения тоже часто различны. Обновляемые представления обычно используются точно так же, как базовые таблицы. Практически пользователи могут даже не знать, с каким объектом они имеют дело: с базовой таблицей или с представлением. Такие представления являются прекрасным механизмом защиты, они позволяют скрыть конфиденциальные или излишние для работы какого-либо пользователя части таблицы. (В главе 22 показано, как открыть пользователю доступ к представлению, но не к таблице, на которой оно базируется.)

С другой стороны, представления "только для чтения", позволяют осуществлять вывод данных и выполнять их переформатирование. Они предоставляют пользователю библиотеку сложных запросов, которые можно выполнять или повторно выполнять, получая производную информацию точно в том виде, какой требуется. Кроме того, можно получать результаты запросов в таблицах, которые затем можно использовать в запросах (например, в соединениях), что дает преимущества перед простым выполнением запросов. Представления "только для чтения" могут иметь приложения, связанные с защитой. Например, может понадобиться, чтобы определенные пользователи видели только агрегатные данные, такие, например, как средние комиссионные для продавца, не имея возможности увидеть индивидуальные значения комиссионных.

Способ указать, какие представления являются обновляемыми

Перед вами несколько примеров обновляемых представлений и представлений "только для чтения":

```
CREATE VIEW Dateorders (odate, ocount)
   AS SELECT odate, COUNT (*)
      FROM Orders
      GROUP BY odate;
```

Это представление "только для чтения", поскольку в него входят функции агрегирования и GROUP BY.

```
CREATE VIEW Londoncust
   AS SELECT *
      FROM Customers
      WHERE city = 'London';
```

Это представление является обновляемым.

```
CREATE VIEW SJsales (name, number, percentage)
   AS SELECT sname, snum, comm * 100
```

```
        FROM Salespeople
        WHERE city = 'San Jose';
```

Это представление является только читаемым, поскольку содержит выражение 'comm * 100'. Перестановка и переименование полей допустимы. Некоторые программные продукты разрешают удаления в представлении или обновление значений столбцов sname и snum.

```
CREATE VIEW Salesonthird
   AS SELECT *
        FROM Salespeople
        WHERE snum IN
            (SELECT snum
                FROM Orders
                WHERE odate = 10/03/1990);
```

Это представление "только для чтения" в смысле ANSI, поскольку содержит подзапрос. В некоторых программах оно может быть допустимым.

```
CREATE VIEW Someorders
   AS SELECT snum, onum, cnum
        FROM Orders
        WHERE odate IN (10/03/1990, 10/05/1990);
```

Это представление является обновляемым.

Выбор значений, размещенных в представлениях

Другая проблема, связанная с использованием представлений, состоит в том, что можно вводить значения, которые "поглощаются" лежащей в основе представления таблицей. Рассмотрим такое представление:

```
CREATE VIEW Highratings
   AS SELECT cnum, rating
        FROM Customers
        WHERE rating = 300;
```

Это обновляемое представление. Оно просто ограничивает доступ к таблице определенными строками и столбцами. Предположим, что вставляется следующая строка:

```
INSERT INTO Highratings
   VALUES (2018, 200);
```

Это правильная команда INSERT для данного представления. Строка будет вставлена через представление Highratings в таблицу Customers. Однако в данном случае она не вставляется в представление, поскольку значение рейтинга не равно 300. Здесь также возникает проблема. Приемлемым может явиться значение 200, но теперь строку из таблицы Customers не видно. Пользователь не может быть уверен в правильности того, что он ввел, поскольку строка не представлена явно, он также не может и удалить ее в любом случае.

Такую проблему можно решить, указав WITH CHECK OPTION в определении представления. Если использовать WITH CHECK OPTION в определении представления Highrating:

```
CREATE VIEW Highratings
   AS SELECT cnum, rating
      FROM Customers
      WHERE rating = 300
      WITH CHECK OPTION;
```

то рассмотренная выше вставка будет отвергнута.

WITH CHECK OPTION относится к типу средств "все или ничего". Это предложение вводится в определение представления, а не в команду DML, значит либо будут контролироваться все команды обновления представления, либо ни одна. Обычно контроль нужен, поэтому удобно иметь WITH CHECK OPTION в определении представления, если только нет особых причин для того, чтобы разрешить ввод в таблицу, лежащую в основе представления, значений, которые в самом представлении не содержатся.

Предикаты и исключаемые поля

Другая сходная проблема связана с включением в представление строк с помощью предиката, базирующегося на одном или более полях, не присутствующих в представлении. Например, может оказаться желательным определить Londonstaff таким образом:

```
CREATE VIEW Londonstaff
   AS SELECT snum, sname, comm
      FROM Salespeople
      WHERE city = 'London';
```

В самом деле, зачем включать значение поля city, если все значения этого поля одинаковы, а название представления говорит о том, какое конкретное значение поля city участвует в формировании представления? Что происходит при попытке вставить строку? Поскольку нельзя указать значение поля city, для него будет принято значение по умолчанию, возможно, значение NULL (NULL будет использоваться, если только явно не указано другое значение). Поскольку строки, добавленные в представление, не удовлетворяют условию формирования представления, они будут из него исключе-

ны, и это справедливо для любой строки, вставляемой в Londonstaff. Строки вводятся через представление Londonstaff в таблицу Salespeople, а затем исключаются из самого представления (если только точно заданное значение по умолчанию не является 'London', но это — особый случай). Пользователь будет не в состоянии вводить строки в данное представление, и все же, даже не зная об этом, сможет вводить строки непосредственно в таблицу, лежащую в основе настоящего представления. Даже если добавить WITH CHECK OPTION в определение представления:

```
CREATE VIEW Londonstaff
   AS SELECT snum, sname, comm
       FROM Salespeople
       WHERE city = 'London'
       WITH CHECK OPTION;
```

проблема не решится. Вывод может быть таким: надо иметь возможность обновлять значения в представлении, удалять строки, но не вставлять (добавлять) их. Иногда не надо разрешать пользователю выполнять добавление строк в представление. Часто полезно включать в представления все поля, на которые есть ссылка в предикате, даже если они не всегда предоставляют полезную информацию. Если присутствие этих полей в выходных данных нежелательно, можно всегда исключить их из запроса на формирование представления и использовать для запроса внутри представления. Т. е. можно определить представление Londonstaff таким образом:

```
CREATE VIEW Londonstaff
   AS SELECT *
       FROM Salespeople
       WHERE city = 'London'
       WITH CHECK OPTION;
```

В результате выполнения команды получается представление с одинаковыми значениями поля city, которые можно исключить из состава выходных данных с помощью запроса:

```
SELECT snum, sname, comm
   FROM Londonstaff;
```

Работа с представлениями, базирующимися на других представлениях

Следует вспомнить о WITH CHECK OPTION в ANSI. В нем не поддерживается каскадный принцип; это предложение действует только на то представление, в котором оно определено, но действие его не распространяется на представления, базирующиеся на этом представлении. Например, в предыдущем примере:

```
CREATE VIEW Highratings
   AS SELECT cnum, rating
```

```
        FROM Customers
        WHERE rating = 300
        WITH CHECK OPTION;
```

попытки вставить или обновить значения рейтинга, отличные от 300, будут ошибочными. Но можно создать второе представление (с тем же самым содержимым), основанное на первом представлении:

```
CREATE VIEW Myratings
    AS SELECT *
    FROM Highratings;
```

Для такого представления можно выполнить изменение рейтинга на значение, отличное от 300:

```
UPDATE Myratings
    SET rating = 200
    WHERE cnum = 2004;
```

Команда будет воспринята и выполнена.

Предложение WITH CHECK OPTION позволяет убедиться в том, что любое обновление значений в представлении осуществляется в соответствии со значениями, указанными в предикате для этого представления. Обновление других представлений, базирующихся на данном, вполне возможно, хотя ему могут препятствовать предложения WITH CHECK OPTION внутри этих представлений. Даже если такие представления присутствуют, они контролируют только предикаты представлений, в которых содержатся. Таким образом, даже если представление Myratings было бы определено следующим образом:

```
CREATE VIEW Myratings
    AS SELECT *
    FROM Highratings
    WITH CHECK OPTION;
```

проблема не была бы решена. Предложение WITH CHEC OPTION проверяет только предикат представления Myratings. Поскольку Myratings фактически не имеет предиката, то предложение WITH CHECK OPTION практически ничего не делает. Если бы в команде применялся предикат, то именно он бы и использовался для любых обновлений Myratings, а предикат, заданный в определении Highratings игнорировался бы.

Это недостаток стандарта SQL, который во многих программных продуктах корректируется. Прежде чем использовать на практике определение представления, рассмотренного в последнем примере, следует опробовать его свойства на примере. (Эта попытка может оказаться более успешной и результативной, чем попытка найти ответ на конкретный вопрос в системной документации.)

Итоги

Представления изучены полностью. Помимо правил определения, является ли данное представление обновляемым в SQL, вы узнали базовые концепции, на которых основываются правила: обновления представлений разрешены, только если SQL может однозначно определить, какие значения лежащих в основе представления таблиц следует обновить. Это означает, что команда обновления в процессе выполнения не требует ни изменения множества строк в единицу времени, ни сравнений множества строк какой-либо базовой таблицы либо выходных данных запроса. Поскольку соединения требуют сравнения строк, они запрещены. Вы научились отличать способы применения обновляемых представлений и представлений "только для чтения".

Обновляемые представления можно рассматривать как окна, показывающие данные единственной таблицы, в которой могут быть опущены или переставлены местами столбцы, а также представлены только те строки, которые удовлетворяют предикату.

С другой стороны, представления "только для чтения" могут содержать наиболее эффективные SQL-запросы, следовательно, они сами могут быть способом получения запросов, которые необходимо выполнить практически в неизменном виде. Кроме того, имея запросы, где выходные данные рассматриваются как объекты, к которым в свою очередь формулируются запросы для получения выходных данных, можно последовательно уточнять и получать интересующие данные.

Теперь вы умеете с помощью предложения WITH CHECK OPTION в определении представления предотвращать выполнение команд обновления данных представлений, ориентированное на введение в базовую таблицу тех строк, которых нет в самом представлении, а также использовать WITH CHECK OPTION вместо ограничений на таблицу, лежащую в основе данного представления.

Для отдельных запросов обычно можно без особых проблем использовать в предикате один или более столбцов, которые не представлены в списке выбранных выходных данных. Однако, если такие запросы применяются в обновляемых представлениях, то они являются проблематичными, поскольку порождают представления, которые не могут иметь вставляемых строк (хотя при отсутствии WITH CHECK OPTION строки могут содержаться в базовой таблице). Мы рассмотрели ряд возможных подходов к решению этой проблемы.

В последних двух главах мы затронули средства защиты представлений. Можно открыть пользователям доступ к представлениям, не открывая доступа к таблицам, которые являются непосредственной основой представлений. В главе 22 рассматривается вопрос доступа к данным как объектам SQL.

Работаем на SQL

1. Какие из представлений являются обновляемыми?

```
#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 = customers.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 Salespeople с именем Commissions. Это представление будет включать только поля snum и comm. Для этого представления можно вводить и изменять комиссионные, но это могут быть только значения из интервала .10 и .20.

3. Некоторые SQL приложения имеют встроенные ограничения, представляющие текущую дату, иногда называемую "CURDATE". Значит слово CURDATE может использоваться в SQL-предложениях и заменяется текущей датой тогда, когда это ключевое слово применяется в командах SELECT и INSERT. Будет использоваться представление таблицы Orders, имеющее имя Entryorders и дающее возможность вставлять строки в таблицу Orders. Создайте таблицу Orders таким образом, чтобы CURDATE автоматически подставлялось в поле odate, если никакое значение не задано. Затем создайте представление Entryorders так, чтобы никакие значения не могли быть изменены.

(Ответы смотри в приложении А.)

22

Определение прав доступа к данным

Эта глава знакомит вас с привилегиями. Как упоминалось в главе 2, SQL применяется главным образом для тех приложений, где требуется распознавать и дифференцировать пользователей системы. В обязанности администрирования базы данных входит создание других пользователей и назначение им привилегий. С другой стороны, пользователи, создающие таблицы, управляют ими. *Привилегии* (privileges) — это определения того, может или нет отдельный пользователь выполнить данную команду. Существует несколько типов привилегий, соответствующих определенным типам операций. Привилегии назначаются и отменяются с использованием двух SQL команд — GRANT и REVOKE, о применении которых рассказывается в этой главе.

Пользователи

Каждый пользователь в среде SQL имеет специальный идентификатор: имя или число. Специальная терминология различается, но, следуя ANSI, необходимо ссылаться на это имя или число как на идентификатор пользователя (authorization ID). Команды, задаваемые для базы данных, связаны с отдельным пользователем, т.е. с особым идентификатором пользователя. В базах данных SQL идентификатор пользователя — это имя пользователя, и SQL может применять специальное ключевое слово USER для ссылки на ID, связанный с текущей командой. Команды интерпретируются и разрешаются (или запрещаются) на основе информации, связанной с ID выдавшего команду.

Вход в систему

В многопользовательских системах есть процедура входа, в соответствии с которой осуществляется доступ к компьютерной системе. Эта процедура определяет, какой ID связан с текущим пользователем. Каждый человек, применяющий базу данных, имеет свой собственный ID, так что пользователи, представленные в SQL, соответствуют реальным. Часто пользователям с множеством функций назначается множество ID, либо один идентификатор может применяться несколькими пользователями. У SQL нет способа различить эти ситуации: он просто связывает пользователя с его ID.

База данных SQL может использовать собственную процедуру входа в систему, либо она может разрешить применение другой программы, такой как операционная система (основная программа, которая выполняется на компьютере), для управления входом и получения идентификатора пользователя от этой программы. Независимо от способа входа в систему существует ID, связанный с конкретными действиями и совпадающий со значением ключевого слова USER.

Передача привилегий

Каждый пользователь в базе данных SQL имеет множество привилегий. Привилегии — это те действия, которые может выполнять пользователь (вход в систему — минимальная привилегия). Привилегии могут меняться с течением времени: новые привилегии могут добавляться, старые — отменяться. Некоторые из привилегий определены в ANSI SQL, но имеются и дополнительные, весьма полезные на практике. Привилегии SQL, определенные ANSI, являются недостаточными для множества реальных ситуаций. С другой стороны, необходимые типы привилегий могут существенно отличаться от тех, которые предоставляет реальная система, — ANSI рассматривает не все из этих проблем. Привилегии, не являющиеся частью стандарта SQL, используют сходный синтаксис, который не идентичен стандарту описания вопросов конфиденциальности.

Стандартные привилегии

Привилегии SQL, определенные ANSI, являются *объектными привилегиями* (*object privileges*). Это означает, что пользователь имеет привилегию выполнить данную команду только для определенного объекта базы данных. Привилегии должны различаться для различных объектов, но системные привилегии, базирующиеся исключительно на объектных привилегиях, не могут покрыть всех потребностей SQL.

Объектные привилегии связаны как с пользователями, так и с таблицами. Привилегия дается отдельному пользователю для отдельной таблицы либо для базовой таблицы, либо для представления. Следует помнить, что пользователь, создавший таблицу, является ее владельцем. Это означает, что пользователь имеет все привилегии на таблицу и может назначать привилегии для работы с ней для других пользователей. Приведем те привилегии, которые может назначить пользователь:

SELECT — Пользователь с этой привилегией может выполнять запросы для таблицы.

INSERT — Пользователь с этой привилегией может выполнить команду INSERT для таблицы.

UPDATE — Пользователь с этой привилегией может выполнить команду UPDATE для таблицы. Привилегия ограничивается множеством столбцов таблицы.

DELETE — Пользователь с этой привилегией может выполнить команду DELETE для таблицы.

REFERENCES — Пользователь с этой привилегией может определить внешний ключ, который использует один или несколько столбцов таблицы

в качестве родительского ключа. Можно ограничить привилегию указанием столбцов. (См. главу 19 для уточнения деталей по поводу внешнего и родительского ключей.)

Кроме того, можно ввести нестандартные привилегии объектов, такие как INDEX (право на создание индексов таблицы), SYNONYM (право на создание синонимов объектов, см. главу 23) и ALTER (право на выполнение команды для таблицы ALTER TABLE). Механизм SQL позволяет выполнить назначение пользователям этих привилегий с помощью команды GRANT.

Команда GRANT

Предположим, пользователь Diane является владельцем таблицы Customers и нужно передать пользователю Adrian право на формулирование запросов к этой таблице. Для этого пользователь Diane должен ввести следующую команду:

```
GRANT SELECT ON Customers TO Adrian;
```

Теперь Adrian может выполнять запросы по таблице Customers. Без других привилегий он имеет право выполнять только выборку, а не какие-либо действия, влияющие на значения элементов таблицы Customers (включая использование Customers в качестве родительской таблицы внешнего ключа, ограничивающего изменения, которые могут быть сделаны для значений таблицы Customers).

Когда SQL выполняет команду GRANT, он выбирает привилегии пользователя, давая ему возможность определить GRANT как допустимую команду. Сам пользователь Adrian не может задать выполнение этой команды, а также передать право выполнения SELECT другому пользователю: владельцем таблицы остается Dian (хотя было показано, как Dian может передать права на выполнение команды SELECT пользователю Adrian).

Для передачи прав на другие привилегии синтаксис тот же самый. Если Adrian является владельцем таблицы Salespeople, то он может разрешить Dian вводить в нее строки; для этого следует указать команду:

```
GRANT INSERT ON Salespeople TO Diane;
```

Теперь Diane может вносить в таблицу сведения о новых продавцах.

Группы привилегий, группы пользователей. Передача привилегий не ограничивается передачей единственной привилегии единственному пользователю с помощью одной команды GRANT. Списки привилегий или пользователей с элементами, разделенными запятыми, допустимы. Например, Stephen может передать права на выполнение команд SELECT и INSERT для таблицы Orders пользователю Adrian:

```
GRANT SELECT, INSERT ON Orders TO Adrian;
```

либо пользователям Adrian и Diane:

```
GRANT SELECT, INSERT ON Orders TO Adrian, Diane;
```

При перечислении привилегий и пользователей все указанные в списке привилегии передаются всем перечисленным пользователям. В соответствии с интерпретацией ANSI, нельзя передать привилегии для множества таблиц в одной команде, но некоторые программные продукты разрешают использование нескольких имен таблиц, разделенных запятыми, таким образом, что все перечисленные пользователи получают все перечисленные привилегии для всех указанных в списке таблиц.

Ограничение привилегий для множества столбцов. Все объектные привилегии используют один и тот же синтаксис, за исключением команд UPDATE и REFERENCES, в которых, в случае необходимости, можно указывать имена столбцов. Привилегию выполнения команды UPDATE можно передать как и все остальные привилегии:

```
GRANT UPDATE ON Salespeople TO Diane;
```

Команда разрешает пользователю Diane изменять значения любого или всех столбцов таблицы Salespeople. Однако, если Adrian желает ограничить Diane возможностью изменять только значения комиссионных (значения столбца comm), то он должен вместо предыдущей команды указать:

```
GRANT UPDATE (comm) ON Salespeople TO Diane;
```

То есть пользователь просто должен указать после имени таблицы в круглых скобках имя столбца, на который распространяется привилегия UPDATE. Множество имен столбцов таблицы можно указать в любом порядке через запятую:

```
GRANT UPDATE (city, comm) ON Salespeople TO Diane;
```

REFERENCES задается в соответствии с теми же правилами. Когда привилегии REFERENCES передаются другому пользователю, он может создать внешние ключи, которые ссылаются на столбцы вашей таблицы как на родительские ключи. Подобно UPDATE, привилегия REFERENCES может содержать список из одного или более столбцов для каждой привилегии. Например, Diane может передать Stephen право использования таблицы Customers в качестве родительского ключа с помощью следующей команды:

```
GRANT REFERENCES (cname, cnum) ON Customers TO Stephen;
```

Команда дает Stephen право использовать столбцы cnum и cname в качестве родительских ключей для любых внешних ключей в этих таблицах. Stephen получает возможность управления по собственному усмотрению. В одной из его собственных таблиц он может определить (cname, cnum) или, например, (cnum, cname) как родительский ключ, определенный на двух столбцах и соответствующий внешнему, определенному на двух столбцах. Или он может создать отдельные внешние ключи для ссылки на поля индивидуально, при условии, что Diane имеет подходящим образом разработанные родительские ключи (см. главу 19). Здесь нет ограничений на число внешних ключей, которые могут базироваться на родительских ключах, и на число родительских ключей для разрешенного множества внешних ключей.

Что касается привилегии UPDATE, то можно опустить список столбцов и, следовательно, разрешить использование всех столбцов в качестве родительских ключей. Adrian может передать Diane права делать это следующим образом:

```
GRANT REFERENCES ON Salespeople TO Diane;
```

Естественно, привилегии полезны только для тех столбцов, которые удовлетворяют ограничениям, необходимым для родительских ключей.

Использование аргументов ALL и PUBLIC

SQL поддерживает два аргумента для команды GRANT, которые имеют специальное назначение: ALL PRIVILEGES (или просто ALL) и PUBLIC. ALL используется вместо имен привилегий в команде GRANT для передачи всех привилегий для таблицы. Например, пользователь Diane может передать пользователю Stephen множество привилегий для таблицы Customers с помощью команды:

```
GRANT ALL PRIVILEGES ON Customers TO Stephen;
```

(Привилегии для команд UPDATE и REFERENCES относятся ко всем столбцам.) Альтернативным способом сказать то же самое является команда:

```
GRANT ALL ON Customers TO Stephen;
```

PUBLIC — своего рода обобщающий аргумент, но он относится к пользователям, а не к привилегиям. Когда передаются привилегии с атрибутом "общедоступный" (PUBLIC), все пользователи получают их автоматически. Чаще это применяется для привилегии SELECT для определенных базовых таблиц или представлений, которые нужно предоставить каждому пользователю для рассмотрения. Разрешить каждому пользователю просматривать таблицу Orders можно, например, вводом следующей команды:

```
GRANT SELECT ON Orders TO PUBLIC;
```

Можно передавать любую привилегию или все привилегии в общее пользование, но делать это нежелательно. Все привилегии, за исключением SELECT, разрешают пользователю изменять (или, в случае REFERENCES, ограничивать) содержимое таблицы. Разрешение всем пользователям изменять содержимое таблиц влечет за собой множество проблем. Даже в небольшой компании предпочтительнее передача привилегий каждому пользователю индивидуально, а не всем сразу. При применении ключевого слова PUBLIC речь идет не только обо всех существующих пользователях. Любой новый пользователь, подключаемый к системе, автоматически получает все привилегии, назначенные с применением ключевого слова PUBLIC. Поэтому для ограничения доступа к таблице в любое время — либо только сейчас, либо в будущем — лучше передать привилегии пользователям, применяя метод, отличный от SELECT.

Передача привилегий с использованием GRANT OPTION

Иногда создатель таблицы хочет, чтобы другие пользователи имели право передавать привилегии на эту таблицу. Это реально для систем, в которых один или несколько человек могут создать большинство или все базовые таблицы базы данных,

а затем передать права по работе с этими таблицами тем, кто реально будет с ними работать. SQL позволяет это сделать с помощью предложения WITH GRANT OPTION.

Если Diane желает, чтобы Adrian имел право передавать полномочия на работу с таблицей Customers другим пользователям, то он должен передать Adrian привилегию на выполнение команды SELECT и применить предложение WITH GRANT OPTION:

```
GRANT SELECT ON Customers TO Adrian
  WITH GRANT OPTION;
```

После выполнения этой команды Adrian получает право передать привилегию выполнения команды SELECT третьим лицам. Для этого он должен выполнить команду:

```
GRANT SELECT ON Diane.Customers TO Stephen;
```

или даже:

```
GRANT SELECT ON Diane.Customers TO Stephen
  WITH GRANT OPTION;
```

Пользователь с GRANT OPTION для отдельной привилегии для данной таблицы может, в свою очередь, передать эту конкретную привилегию на эту таблицу с предложением или без предложения GRANT OPTION любому другому пользователю, что не изменяет владельца таблицы: таблицы всегда принадлежат их создателям. (В качестве префикса перед именем таблицы следует указывать идентификатор создателя таблицы, как это было продемонстрировано в примере. В следующей главе будет показан другой способ.) Пользователь с GRANT OPTION на все привилегии для данной таблицы имеет большую власть над этой таблицей.

Лишение привилегий

Как ANSI определяет команду CREATE TABLE для создания таблиц, но не предусматривает команды DROP TABLE для отказа от них, так же стандарт поддерживает команду GRANT для передачи привилегии пользователям, но не предусматривает способа лишить пользователя привилегий. Потребность лишения привилегий удовлетворяется с помощью команды REVOKE, которая де-факто стала одной из характеристик стандарта.

Синтаксис команды REVOKE повторяет синтаксис команды GRANT, но имеет противоположное значение. Таким образом, лишить Adrian привилегии выполнять команду INSERT для таблицы Orders можно с помощью команды:

```
REVOKE INSERT ON Orders FROM Adrian;
```

Списки привилегий и пользователей доступны, как и для команды GRANT, значит можно ввести следующую команду:

```
REVOKE INSERT, DELETE ON Customers FROM Adrian, Stephen;
```

Но кто имеет право лишать пользователей привилегий? Когда пользователь, имевший привилегии, лишается их, лишаются ли привилегий пользователи, получившие привилегии от него? Поскольку эта характеристика не является стандартной, однозначных ответов не существует. Наиболее распространенным подходом является следующий: привилегии отменяются пользователем, передавшим их, и отмена привилегий осуществляется *каскадно*, т.е. операция применяется последовательно ко всем пользователям, получившим привилегии от того пользователя, который их лишается в данный момент.

Использование представлений для фильтрации привилегий

С привилегиями можно работать проще, используя представления. При передаче привилегии для базовой таблицы пользователю, она автоматически применяется ко всем строкам и, с возможными исключениями для UPDATE и REFERENCES, ко всем столбцам таблицы. Создавая представление, которое ссылается на базовую таблицу, и передавая привилегии представлению, а не базовой таблице, можно ограничить их любым выражением, применимым в запросе, который формирует представление. Это во многом определяет основные возможности команды GRANT.

Кто может создавать представления? Для того, чтобы создать представление, нужно иметь привилегию использования команды SELECT для всех таблиц, на которые есть ссылка в представлении. Если представление является обновляемым, любые привилегии INSERT, UPDATE и DELETE на таблицу, лежащую в основе представления, автоматически применимы также и к представлению. Если нет привилегий обновления таблиц, лежащих в основе представления, то их нет и для создаваемых конкретных представлений, даже если сами представления являются обновляемыми. Поскольку внешние ключи в представлениях не применяются, нет необходимости использовать привилегию REFERENCES при создании представлений. Эти ограничения определены ANSI. Можно использовать также нестандартные привилегии (будут рассмотрены далее в этой главе). В следующих разделах предположим, что создатели рассматриваемых представлений являются владельцами или имеют соответствующие привилегии на все используемые базовые таблицы.

Ограничение привилегии select для определенных столбцов. Предположим, нужно дать пользователю Claire возможность видеть только столбцы snum и sname таблицы Salespeople. Это можно сделать, указав имена этих столбцов в представлении:

```
CREATE VIEW Clairesview
   AS SELECT snum, sname
      FROM Salespeople;
```

и передав Claire привилегию SELECT на представление, а не на таблицу Salespeople:

```
GRANT SELECT ON Clairesview to Claire;
```

Такие привилегии на использование столбцов можно создать, применяя и другие привилегии, но для команды INSERT это означает вставку значений по умолчанию, а для команды DELETE ограничения на столбцы будут бессмысленны. Привилегии UPDATE и REFERENCES могут быть сделаны для столбцов без изменения представления.

Ограничение области действия привилегий определенным подмножеством строк. Более полезным способом фильтрации привилегий для представлений является использование самого представления для ограничения привилегии отдельными строками. Естественно, для этого в представлении используется предикат, определяющий какие строки интересны. Например, чтобы передать привилегию на выполнение команды UPDATE для всех покупателей (таблица Customers), расположенных в Лондоне (city = 'London'), Adrian, можно создать такое представление:

```
CREATE VIEW Londoncust
   AS SELECT *
      FROM Customers
      WHERE city = 'London'
      WITH CHECK OPTION;
```

Затем можно передать привилегию UPDATE Adrian:

```
GRANT UPDATE ON Londoncust TO Adrian;
```

Эта привилегия отличается от определенной на столбцах привилегии UPDATE тем, что исключаются все строки таблицы Customers, в которых значение поля city отлично от 'London'. Предложение WITH CHECK OPTION не дает возможности Adrian изменять значение поля city.

Передача прав доступа для производных данных. Можно также предоставить пользователям доступ к производным данным, отличным от реальных данных таблицы. В этом случае могут быть полезны функции агрегирования. Можно создать представление, которое дает общее число, среднюю величину и сумму по полю amt для заказов, поступивших на каждую дату:

```
CREATE VIEW Datetotals
   AS SELECT odate, COUNT (*), SUM (amt), AVG (amt)
      FROM Orders
      GROUP BY odate;
```

Теперь Dine дается привилегия выполнять команду SELECT для представления Datetotals:

```
GRANT SELECT ON Datetotals TO Diane;
```

Использование представлений как альтернативы системе ограничений. Один из вариантов применения техники, упомянутой в главе 18, состоит в использовании представлений WITH CHECK OPTION вместо ограничений. Предположим, нужно

удостовериться, что все значения поля city в таблице Salespeople имеют одно и то же значение — название города, в котором в настоящее время расположен офис компании. Можно использовать ограничение CHECK на столбец city непосредственно, но позже будет трудно внести изменения, если компания откроет еще ряд офисов. Альтернативой является создание представления, которое исключает ошибочные значения поля city:

```
CREATE VIEW Curcities
   AS SELECT *
      FROM Salespeople
      WHERE city IN ('London', 'Rome', 'San Jose', 'Berlin')
      WITH CHECK OPTION;
```

Теперь, вместо привилегии изменять таблицу Salespeople, пользователям дается привилегия на предоставление им представления Curcities. Преимущество этого подхода заключается в следующем: при необходимости внести изменение можно исключить это представление, создать новое представление и передать пользователям привилегии на него, что гораздо проще, чем менять ограничение. Недостаток заключается в том, что владелец таблицы Salespeople также должен использовать это представление, если он хочет наверняка исключить собственные ошибки.

С другой стороны, такой метод позволяет владельцу таблицы и всем другим пользователям, имеющим право обновлять привилегии для самой таблицы, а не для представления, делать исключения для ограничений. Это весьма желательно, но невыполнимо при использовании ограничений на саму базовую таблицу. Эти исключения невидимы в представлении. Если применяется такой метод, можно создать второе представление, содержащее только ограничения:

```
CREATE VIEW Othercities
   AS SELECT *
      FROM Salespeople
      WHERE city NOT IN ('London', 'Rome', 'San Jose', 'Berlin')
      WITH CHECK OPTION;
```

При передаче пользователям только привилегии на выполнение команды SELECT для этого представления они смогут видеть исключенные строки, но не смогут занести ошибочные значения в базовую таблицу. Фактически пользователи могут сформулировать запросы к двум представлениям, сформировать их объединение и увидеть все строки за один раз.

Другие типы привилегий

Преимущественное право создавать таблицы не регламентируется ANSI. Но эту область привилегий нельзя игнорировать (все стандартные ANSI-привилегии вытекают из нее), ведь именно создатели таблиц предоставляют привилегии на объекты. Кроме того, разрешая всем пользователям создавать базовые таблицы в системе любого размера, увеличивается избыточность данных и снижается эффективность системы их обработки. Следующие вопросы связаны с этим же. Кто имеет право изменять, удалять или накладывать ограничения на таблицу? Отличается ли право создавать базовые таблицы от права создавать представления? Должны ли существовать *суперпользователи (superusers)* — пользователи, основной обязанностью которых является управление базой данных и которые имеют подавляющее множество или все привилегии и не имеют права их индивидуальной передачи?

Поскольку ANSI не касается этих вопросов, в конкретных реализациях SQL используется огромное множество подходов, и невозможно дать вполне конкретные ответы на все эти вопросы. Можно только изложить наиболее общий подход к решению этих проблем.

Привилегии, которые не определяются в терминах отдельных объектов, называются *системными привилегиями (system privileges)* или *авторским правом на базу данных (database authorities)*. В подавляющем большинстве случаев системные привилегии включают в себя право создавать объекты данных, различать базовые таблицы (обычно создаваемые силами нескольких пользователей) и представления (обычно создаваемые многими или всеми пользователями). Системная привилегия на создание представления поддерживается, в отличие от привилегии замены, объектными привилегиями, представленными в стандарте ANSI. Кроме того, в системе любого размера существует суперпользователь. Термин, чаще всего используемый для такого суперпользователя и его привилегий, — администратор базы данных (DBA — Database Administrator).

Типичная система привилегий

Обычно принято различать три типа базовых систем привилегий, которые называются CONNECT, RESOURCE и DBA. CONNECT предусматривает право входить в систему и создавать представления и синонимы (см. главу 23), если речь идет об объектных привилегиях. RESOURCE предоставляет право создавать базовые таблицы. DBA — это привилегия суперпользователя, дающая право пользователю распоряжаться базой данных по своему усмотрению (как своей собственной). Эту привилегию имеет пользователь (один или несколько) с функцией администрирования базы данных. В некоторых системах есть также специальный пользователь (иногда называемый SYSADM или просто SYS), обладающий высшим авторским правом; это специальное обозначение пользователей, в отличие от имеющих привилегию DBA. SYSADM рассматривается как идентификатор пользователя. Различия в названиях и приписанных им действиях существенны для различных систем. Мы будем ссылаться на самого высоко привилегированного пользователя (или пользователей), который занимается проектированием и управлением базой данных как на пользователя DBA, отражая тот факт, что за этим названием скрывается скорее функция чем привилегия.

Команда GRANT в своей модифицированной форме применима и к системным, и к объектным привилегиям. Начальные привилегии назначаются пользователем DBA. Например, DBA может передать привилегию создания таблицы пользователю Rodriguez таким образом:

```
GRANT RESOURCE TO Rodriguez;
```

Создание и исключение пользователей

Как был создан пользователь с именем Rodriguez? Каким образом определяется этот идентификатор автора? Во многих программных продуктах DBA создает пользователя автоматически, передавая ему привилегию CONNECT. В этом случае обычно добавляется предложение IDENTIFIED BY, позволяющее определить пароль. Например, DBA может ввести:

```
GRANT CONNECT TO Thelonius IDENTIFIED BY Redwagon;
```

Команда создает пользователя Thelonius, дает ему право входа в систему и назначает ему пароль Redwagon. Теперь, поскольку Thelonious является распознаваемым пользователем, он или DBA могут использовать эту же команду для изменения пароля Redwagon. На практике встречаются ограничения такого рода. Невозможно иметь пользователя, который не может войти в систему даже временно. Если не надо предоставлять пользователю право входа в систему, следует выполнить команду REVOKE для привилегии CONNECT, которая лишит пользователя этой привилегии. Некоторые программные продукты позволяют создавать и разрушать пользователей независимо от привилегии входа в систему.

При передаче пользователю привилегии CONNECT создается сам этот пользователь. Для этого нужно иметь привилегию DBA. Если этот пользователь должен создавать базовые таблицы, а не только представления, он должен также получить привилегию RESOURCE. Однако здесь возникает другая проблема. Если попытаться отменить привилегию CONNECT для пользователя, имеющего собственные таблицы, то эта команда будет отвергнута, поскольку в противном случае появились бы таблицы, не имеющие владельца, а такая ситуация запрещена в теории и практике баз данных. Необходимо удалить все таблицы пользователя, прежде чем лишать его привилегии CONNECT. Если эти таблицы не пусты, потребуется сохранить их содержимое в других таблицах с помощью команды INSERT, использующей запрос. Отменять привилегию RESOURCE отдельно нельзя, отмена привилегии CONNECT эквивалентна уничтожению пользователя.

Это стандартный подход к системным привилегиям, но он имеет значительные ограничения. Существуют и альтернативные подходы, которые используют более узкое определение или менее контролируемую систему привилегий.

Обсуждение этих вопросов выводит нас за границы стандарта SQL, определенного в настоящее время, а в некоторых программных продуктах касается областей, вообще не относящихся к компетенции SQL. Эти проблемы не встречаются на практике, если только пользователь не является DBA или пользователем высокого уровня. Обычным

пользователям требуются лишь знания концепции назначения системных привилегий, эти сведения можно получить из конкретной системной документации.

Итоги

Привилегии дают возможность увидеть SQL с новой точки зрения, а именно: действия SQL выполняются конкретными пользователями в конкретной системе баз данных. Команда GRANT сама по себе достаточно проста: с ее помощью можно передать одну или несколько привилегий для объекта одному или нескольким пользователям. Если пользователю передается привилегия WITH GRANT OPTION, то он может, в свою очередь, передавать эту привилегию другим пользователям.

Мы изучили привилегии пользователя на представления, их преимущества и недостатки. Системные привилегии, которые являются необходимыми, но выходят за границы стандарта SQL, рассмотрены в самом общем виде.

В главе 23 обсуждаются следующие вопросы: сохранение и отмена изменений, создание собственных имен для таблиц других пользователей. Мы рассмотрим случаи, когда различные пользователи пытаются осуществить доступ в один и тот же момент времени к одному и тому же объекту.

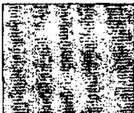
Работаем на SQL

1. Дайте Janet право изменять рейтинги (ratings) продавцов.
2. Дайте Stephen право предоставлять другим пользователям право формулировать запросы к таблице Orders.
3. Отмените привилегию выполнять команду INSERT для таблицы Salespeople для пользователя Claire и всех тех пользователей, которым Claire передал полномочия.
4. Передать Jerry право выполнять вставку и обновление таблицы Customers с учетом того, что значение рейтинга лежит в интервале от 100 до 500.
5. Разрешите Janet формулировать запросы к таблице Customers, но запретите ей доступ к тем покупателям (Customers), чей рейтинг является самым низким.

(Ответы см. в приложении А.)

23

Глобальные аспекты SQL



В этой главе обсуждаются аспекты языка SQL, имеющие общее отношение к базе данных, рассматриваемой как единое целое, включая использование множества имен для объектов данных, распределение памяти, отказ от изменений или сохранение изменений в базе данных, координацию одновременных действий со стороны многих пользователей. Этот материал поможет конфигурировать базу данных, избежать ошибок и определить, как действия, осуществляемые с базой данных одним пользователем, могут повлиять на других пользователей.

Переименование таблиц

Если в команде есть ссылка на базовую таблицу или представление, владельцем которых является другой пользователь, то нужно указать в качестве префикса перед именем таблицы имя владельца; благодаря этому SQL знает, где ее искать. Поскольку в целом ряде случаев это может оказаться неудобным, многие программные реализации SQL позволяют создавать синонимы для таблиц (эта возможность не входит в стандарт SQL). *Синоним* (synonym) — это альтернативное имя таблицы, подобное прозвищу. Создавая синоним, пользователь становится его собственником, значит перед синонимом не следует указывать другого идентификатора пользователя (имя пользователя).

Имея, по крайней мере, одну привилегию для одного или нескольких столбцов таблицы, можно создать для них синоним. (Некоторые программные реализации SQL могут иметь специальную привилегию для создания синонимов.) Adrian может создать синоним, скажем, Clients для Diane.Customers, используя команду CREATE SYNONYM следующим образом:

```
CREATE SYNONYM Clients FOR Diane.Customers;
```

Теперь Adrian может использовать имя таблицы Clients в команде точно так же, как и имя Diane.Customers. Владельцем синонима Clients является Adrian, и только он может его использовать.

Переименование с использованием того же имени

В SQL префикс пользователя является реальной частью любого имени таблицы. Когда пропускается собственное имя пользователя перед именем конкретной таблицы, SQL подставляет его сам. Следовательно, два одинаковых имени для таблиц, принадлежащих различным пользователям, реально не являются совпадающими, их нельзя перепутать (по крайней мере в SQL). Это значит, что два пользователя могут создать две совершенно несвязанные таблицы с одним и тем же именем, но это означает и то, что один пользователь может создать представление, основанное на таблице другого пользователя, им поименованное, и использовать в качестве имени представления имя таблицы. Это делают из практических соображений в тех случаях, когда представле-

ние полностью совпадает с таблицей. Например, представление полностью использует CHECK OPTION в качестве подстановки для ограничения CHECK в базовой таблице (детали рассмотрены в главе 22). Можете создать собственные синонимы, которые совпадают с оригинальными названиями таблиц. Например, Adrian может определить Customers как свой собственный синоним таблицы Diane.Customers таким образом:

```
CREATE SYNONYM Customers FOR Diane.Customers;
```

С точки зрения SQL, теперь имеются два различных имени для таблицы: Diane.Customers и Adrian.Customers. Каждый из пользователей может ссылаться на таблицу просто как Customers.

Одно имя на всех

Если таблицу Customers будет использовать множество пользователей, наилучшим может оказаться вариант, при котором все они будут применять одно и тоже имя таблицы. Это позволит, например, использовать имя для внутренних потребностей без указания префикса. Для определения единственного имени для всех пользователей, необходимо создать общедоступный (public) синоним. Например, если все пользователи должны называть таблицу Customers просто как Customers, следует ввести:

```
CREATE PUBLIC SYNONYM Customers FOR Customers;
```

Являясь владельцем таблицы Customers, вы можете не указывать перед именем этой таблицы собственное имя в качестве префикса. В общем случае доступные для всех синонимы создаются либо владельцами таблиц, либо высокопривилегированными пользователями, такими как DBA. Пользователи должны также получить привилегии на таблицу Customers для того, чтобы иметь к ней доступ. Несмотря на то, что в результате выполнения рассмотренной выше команды имя таблицы становится общедоступным, сама таблица таковой не является. Общедоступные синонимы принадлежат PUBLIC, а не тем, кто их создал.

Уничтожение синонимов

Общедоступные или другие синонимы можно удалить с помощью команды DROP SYNONYM. Синонимы удаляются их владельцами, за исключением общедоступных синонимов, которые удаляются пользователями с привилегиями DBA. Чтобы исключить свой синоним Client — поскольку теперь можно воспользоваться общедоступным синонимом Customers, — Adrian должен ввести

```
DROP SYNONYM Clients;
```

Естественно, что на саму таблицу Customers эти команды никакого действия не оказывают.

Каким образом база данных размещается для пользователя?

Таблицы и другие объекты данных хранятся в базе данных и ассоциируются с отдельными пользователями, которые являются их владельцами. Можно сказать, что они хранятся "в пространстве памяти, связанном с именем пользователя", хотя это не отражает физического расположения, а является просто логической конструкцией. Однако объекты данных должны храниться в памяти в физическом смысле и объем памяти, который можно использовать для отдельного объекта или пользователя в данный момент времени, ограничен. Ни один компьютер не предоставляет немедленного доступа к безграничному множеству устройств (диск, лента или внутренняя память) для хранения данных. Более того, работоспособность SQL повышается, если логическая структура данных поддерживается на физическом уровне, что проявляется при выполнении команд.

В больших SQL-системах (программных продуктах, основанных на использовании SQL) база данных разбивается на области, называемые *пространством базы данных (databasespaces)* или *пространством таблиц (tablespaces)*. Здесь информация содержится сконцентрированно для выполнения команд таким образом, чтобы программа не осуществляла углубленного и расширенного поиска информации, сгруппированной в пределах одной базы данных. Физические детали зависят от конкретной реализации, но работать с этими областями целесообразно непосредственно из SQL. Системы, использующие понятие пространства баз данных (далее будем использовать аббревиатуру dbspaces), разрешают их применять в качестве объектов в командах SQL. Dbspaces создаются с помощью команд CREATE DBSPACE, ACQUIRE DBSPACE или CREATE TABLESPACE в зависимости от конкретной реализации. Одно dbspace может обслуживать любое количество пользователей, а один пользователь может иметь доступ к множеству dbspaces. Привилегию создавать таблицы можно получить от пользователя со спецификацией dbspace.

Можно создать dbspace с названием Sampletables с помощью следующей команды:

```
CREATE DBSPACE Sampletables
  (pctindex  10,
  pctfree    25);
```

Параметр pctindex определяет процент пространства dbspace, выделяемый для хранения индексов таблиц. Параметр pctfree определяет процент пространства dbspace, который остается свободным, чтобы впоследствии разрешить увеличение размеров таблицы за счет добавления строк (команда ALTER TABLE может добавлять столбцы или увеличивать размеры столбцов, делая таким образом каждую строку длиннее. При подобном увеличении используется зарезервированное свободное пространство). Можно указывать и другие параметры, которые изменяются в широких пределах для различных программных продуктов. Большинство программных продуктов предоставляет

значения, автоматически назначаемые по умолчанию, значит, можно создавать dbspaces без указания параметров. Dbspace может иметь ограничения по размеру, либо ему может быть разрешен неограниченный рост вслед за увеличением размеров таблиц. После того как dbspace создано, пользователи получают право создавать в нем объекты.

После создания dbspace Sampletables пользователи получают право создавать в нем объекты. Можно дать Diane право создать таблицы в Sampletables следующим образом:

```
GRANT RESOURCE ON Sampletables TO Diane;
```

Это позволяет более рационально использовать доступный для хранения данных объем памяти. Первый dbspace, назначенный данному пользователю, является обычно участком памяти, в котором по умолчанию создаются все объекты данного пользователя. Пользователи, имеющие доступ к множеству dbspaces, должны указывать, где конкретно они хотят разместить объект.

Разделяя базу данных на участки dbspaces, нужно иметь в виду типы операций, которые будут выполняться часто. Таблицы, которые будут часто соединяться или ссылаться одна на другую с использованием внешнего ключа, лучше разместить вместе в одном dbspace. Исходя из опыта разработки простых таблиц, можно сказать, что таблица Orders будет часто соединяться с одной или двумя другими таблицами, поскольку таблица Orders использует значения из этих двух таблиц. Отсюда следует, что эти три таблицы следует разместить в одном dbspace, независимо от того, кто является их владельцем. Возможное присутствие ограничений внешнего ключа в таблице Orders свидетельствует о том, что в этом же dbspace будет храниться и другая таблица.

Когда изменения становятся постоянными?

Ошибки из-за действий человека или компьютера часто возникают, и нужно дать возможность пользователям отказаться от выполненных ими ранее действий.

Команда SQL, влияющая на содержимое или структуру базы данных, например, команда обновления языка манипулирования данными (DML) или DROP TABLE, не является необратимой. Можно определить, будет ли данная группа из одной или более команд вносить постоянные изменения в базу данных либо будет игнорировать их. С этой целью команды объединяются в группы, названные *транзакциями* (*transactions*).

Транзакция начинается всякий раз, когда инициируется сеанс работы с SQL. Все вводимые команды являются частью той же транзакции, пока она не заканчивается вводом команды COMMIT WORK или команды ROLLBACK WORK. COMMIT делает все изменения, выполняемые в транзакции, постоянными, команда ROLLBACK отменяет их. Новая транзакция начинается после каждой команды COMMIT или ROLLBACK. Этот процесс известен как обработка транзакций.

Синтаксис команды, которая делает все изменения постоянными после начала сеанса связи или после выполнения команды COMMIT или ROLLBACK, следующий:

```
COMMIT WORK;
```

Синтаксис команды, которая отменяет их:

```
ROLLBACK WORK;
```

Во многих программных продуктах устанавливается параметр, названный: AUTOCOMMIT. Он автоматически выполняет все правильные действия и отвергает приводящие к ошибкам. Если такой параметр предусмотрен в системе, можно установить режим выполнения всех действий с помощью команды, подобной этой:

```
SET AUTOCOMMIT ON;
```

Можно перейти к режиму обработки транзакций с помощью команды:

```
SET AUTOCOMMIT OFF;
```

AUTOCOMMIT может автоматически устанавливаться системой при входе в нее (в начале сеанса связи.)

Если сеанс связи пользователя завершается аварийно (например, в результате нарушений питания или в результате повторной загрузки системы пользователем), выполнение текущей транзакции автоматически отменяется. По этой причине, осуществляя транзакции вручную, желательно разделить команды на множество различных транзакций. Единственная транзакция не должна содержать множества несвязанных команд; фактически, она может состоять из единственной команды. Транзакции, которые включают целую группу несвязанных изменений, не оставляют возможности выбора, а позволяют лишь сохранять или отвергать всю эту группу, даже если нужно отменить только одно конкретное изменение. Простое правило, которого здесь следует придерживаться, — включать в состав транзакции единственную команду или тесно связанные команды.

Например, нужно удалить из базы данных продавца Motika. Прежде, чем удалить его из таблицы Salespeople, нужно что-то сделать с его заказами и покупателями. Необходимо установить значение поля snum для покупателей Motika в NULL. Тогда ни один продавец не получит комиссионных от этих заказов, а его покупатели будут переданы продавцу Peel. После этого можно удалить Motika из таблицы Salespeople:

```
UPDATE Orders
   SET snum = NULL
   WHERE snum = 1004;

UPDATE Customers
   SET snum = 1001
   WHERE snum = 1004;
DELETE FROM Salespeople
   WHERE snum = 1004;
```

Если с удалением Motika появилась проблема (например, есть другой ссылающийся на этого продавца внешний ключ, и это не было принято во внимание), может пона-

добиться отменить все изменения до принятия соответствующего решения. Следовательно, эту группу команд полезно объединить в одну транзакцию. Можно поместить перед транзакцией команду COMMIT и закончить ее командой COMMIT или ROLLBACK.

Как SQL работает одновременно с множеством пользователей

В многопользовательских системах, в которых часто применяется SQL, возможны накладки между различными выполняющимися действиями. Например, выполняется следующая команда для таблицы Salespeople:

```
UPDATE Salespeople
   SET comm = comm*2
   WHERE sname LIKE 'R%';
```

В процессе выполнения команды Diane ввел следующий запрос:

```
SELECT city, AVG (comm)
   FROM Salespeople
   GROUP BY city;
```

Будут ли средние значения, полученные Diane, отражать изменения, которые уже заданы для таблицы? Важно получить ответ на вопрос, отражаются ли в результате все изменения комиссионных. Любой промежуточный результат является случайным и непредсказуемым, зависящим от физического порядка расположения записей, но выходные данные запроса не должны зависеть от физических деталей и не должны быть случайными и непредсказуемыми.

Посмотрим на ситуацию с другой точки зрения. Предположим, найдена ошибка, но изменения после получения Diane своих выходных данных не внесены. В результате Diane имеет выходные данные с подсчитанными средними обновленными значениями, обновления позже были отменены, и, следовательно, у Diane нет информации о корректности полученных выходных данных.

Управление одновременно выполняющимися транзакциями называется *параллелизмом (concurrency)*, порождающим ряд проблем. Приведем некоторые примеры:

- Обновления должны выполняться без взаимного влияния. Например, продавец может сформулировать запрос к инвентарной таблице, найти десять единиц товара из числа хранящихся на складе и подготовить шесть из них для покупателя. Прежде чем это изменение было сделано, в соответствии с запросом другого продавца семь единиц этого же товара уже было передано его покупателям.

- От изменения базы данных можно отказаться уже после проявления результата этих изменений, как в предыдущем примере: ошибочные действия были отменены после того, как Diane получила свои выходные данные.

- Одно действие может воздействовать на частичный результат другого действия. Например: среднее значение комиссионных для Diane было установлено в то время, когда выполнялось обновление. Функции, подобные агрегатным, должны отражать состояние базы данных в точке относительной стабильности. Например, некто, проверяющий журналы регистраций, должен иметь возможность вернуться назад и определить, что средние значения для Diane существовали в какой-то момент времени и что они должны были оставаться неизменными, если после этого момента времени не вносилось никаких изменений. Этого не происходит, если обновление выполняется во время вычисления значения функции.

- Тупик. Два пользователя могут пытаться выполнить действия, которые оказывают взаимное влияние. Например, два пользователя одновременно пытаются изменить значения как внешнего, так и соответствующего ему родительского ключей.

Возникало бы множество непредсказуемых ситуаций, если бы одновременно выполняющиеся транзакции были бы неконтролируемыми. Но SQL обеспечивает *управление параллелизмом* (*concurrency* controls). Смысл его в том, что каждая из одновременно выполняющихся команд выполняются так, как если бы она выполнялась совершенно одна от самого начала до момента получения результата (включая, по мере необходимости, команды COMMIT и ROLLBACK).

Точнейшая интерпретация этой ситуации — не разрешать воздействия на таблицу более чем одной транзакции в один момент времени. Базу данных необходимо держать постоянно в состоянии доступности для множества пользователей, а это требует некоторого компромисса с управлением параллелизмом. Большинство SQL-систем предоставляет пользователям варианты, позволяющие балансировать между параллелизмом в использовании данных и доступностью базы данных. Управление такими возможностями осуществляется пользователем, DBA (администратором базы данных) либо и тем, и другим. Иногда они выполняют управление независимо от самого SQL, даже несмотря на использование операции SQL.

Механизмы SQL, которые разработчики применяют для управления совпадающими операциями, называются *замками (блокировками) (locks)*. Замки ограничивают определенные операции на базе данных, когда другие операции или транзакции являются активными. Операции, на которые накладываются ограничения, выстраиваются в очередь и выполняются, когда блокировка снимается (некоторые разработки позволяют задать NOWAIT, при этом команда не ставится в очередь, а отвергается, давая возможность выполнить какие-либо действия вместо ожидания в очереди).

Блокировки в многопользовательских системах весьма существенны. Следовательно, есть своего рода схема блокировки по умолчанию, которая применяется ко всем командам базы данных. Эта схема может быть определена для всей базы данных, либо

конкретная реализация может позволять использовать ее в качестве параметра в команде CREATE DBSPACE или в команде ALTER DBSPACE и, следовательно, определять ее индивидуально для различных dbspaces. Кроме того, системы обычно предоставляют своего рода детектор тупиковых ситуаций, когда две операции выполняют взаимные блокировки. В этом случае выполнение одной из команд откладывается, и соответствующая ей блокировка отменяется.

Терминология и определение схем блокировки изменяются в широких пределах для различных программных продуктов, поэтому в дальнейшем будут использоваться проектные решения, принятые в системе управления базами данных DB2 фирмы IBM, поскольку в ней нашли воплощение наиболее общие прикладные аспекты. IBM является лидером в этой области, и ее подходы широко воспроизводятся в других программных продуктах. Хотя различные программные продукты значительно отличаются от DB2 по синтаксису и по деталям функционирования, основные возможности практически схожи.

Типы блокировок

Существуют два основных вида блокировок: *разделяемые блокировки (share locks)* и *исключительные блокировки (exclusive locks)*. Разделяемые блокировки (или S-locks, S-блокировки) могут выполняться более чем одним пользователем в единицу времени. Это позволяет множеству пользователей осуществлять доступ к данным, не изменяя их. Исключительные блокировки (или X-locks, X-блокировки) запрещают любой доступ к данным со стороны всех пользователей, кроме того, чья блокировка в настоящий момент выполняется. Исключительные блокировки используются для команд, изменяющих содержимое или структуру таблицу. Такая блокировка держится до конца транзакции. Разделяемые блокировки используются для запросов. Продолжительность их действия зависит от уровня изоляции.

Что такое *уровень изоляции (isolation level)* для блокировки? Он определяет, какую часть таблицы охватывает блокировка. В DB2 различают три уровня изоляции: два — применимы и к разделяемой, и к исключительной блокировке, третий уровень применим к разделяемой блокировке. Они управляются командами, выходящими за рамки непосредственно SQL. Точный синтаксис команд, связанных с блокировкой, различен для различных программных продуктов. Следующее обсуждение является концептуально полезным.

Уровень изоляции *"повторное чтение" (read repeatability)* гарантирует, что внутри данной транзакции все записи, участвующие в запросе, не изменяются. Поскольку обновляемые записи в транзакции являются объектом исключительной блокировки до прекращения транзакции, их нельзя никак изменить. С другой стороны, повторное чтение означает, что можно решить заранее, какие строки желательно блокировать и выполнить запрос, который их выбирает. До прекращения выполнения текущей транзакции не будет никаких изменений этих строк. Повторное чтение защищает пользователя, выполнившего блокировку, но его работа при этом замедляется.

Уровень стабильности курсора (cursor stability) защищает каждую запись от изменения в процессе ее чтения или от ее чтения в процессе изменения записи. Такая ис-

ключительная блокировка применяется до тех пор, пока изменения не блокируются или не свертываются. Следовательно, если с помощью стабильности курсора обновляется целая группа записей, то отдельные записи остаются блокированными до завершения транзакции, что аналогично эффекту повторного чтения. Различие между двумя уровнями состоит в воздействии на запросы. На уровне стабильности курсора строки таблицы, отличные от строк, участвующих в запросе, могут быть изменены.

Третий уровень изоляции в DB2 — это уровень *"только для чтения" (read only)*. Уровень изоляции "только для чтения" делает моментальный снимок данных; реально он совершенно не блокирует таблицу. Следовательно, этот уровень изоляции нельзя использовать с командами обновления. Любое содержимое таблицы фиксируется, как единое целое, на момент выполнения команды и входит в состав выходных данных запроса. Это необязательно для уровня стабильности курсора. Блокировка "только для чтения" позволяет убедиться, что выходные данные внутренне непротиворечивы."Только для чтения" удобно для построения отчетов, которые должны быть внутренне последовательными и требовать доступа ко многим или ко всем строкам таблицы, а не к базе данных, как к единому целому.

Другие способы блокировки данных

В некоторых программных продуктах выполняется блокировка на уровне страницы, а не строки. Она может осуществляться по выбору или может быть заложена в проекте системы. Страница — это единица объема памяти, как правило, 1024 байта. Страница состоит из одной или более строк таблицы вместе с индексами и другой сопутствующей информацией, на ней могут располагаться и строки из других таблиц. Если блокировка распространяется на страницу, а не на строку, все данные этой страницы блокируются точно так же, как блокируется индивидуальная строка, в соответствии с уровнями изоляции, рассмотренными ранее.

Основное преимущество этого подхода заключается в реализации. Если SQL не должен поддерживать блокировку и разблокировку строк индивидуально, то он может работать быстрее. При этом нужно знать детали реализации системы несмотря на то, что стандарт SQL был разработан именно для выхода за пределы конкретной реализации.

Сходное средство, доступное в некоторых системах, — блокировка dbspace. Пространства баз данных превосходят размеры страниц, а значит этот подход сочетает в себе преимущества в способе реализации и логические недостатки блокировки на уровне страницы. Предпочтительнее использовать блокировку нижнего уровня.

Итоги

В этой главе были рассмотрены:

- Синонимы (как создать новые имена для объектов данных)
- Пространство базы данных (dbspace) (как разделить доступное в базе данных пространство)
- Обработка транзакций (как сохранить или отказаться от изменений в базе данных)
- Управление параллелизмом (как SQL позволяет исключить влияние команд друг на друга)

Синонимы являются объектами, имеют имена и (иногда) владельцев, но не существуют самостоятельно и независимо от таблицы, имя которой они заменяют. Они могут быть общими и, следовательно, доступными для каждого, имеющего доступ к объекту, или принадлежать только определенному пользователю.

Dbspaces — это подразделы базы данных, выделяемые пользователям. Связанные таблицы, для которых часто выполняется операция соединения, лучше хранить в одном и том же пространстве базы данных.

COMMIT и ROLLBACK — команды, применяемые для сохранения в виде группы всех изменений базы данных, начиная от предыдущей команды COMMIT или ROLLBACK или от начала сеанса, либо для отказа от них.

Управление параллелизмом определяет, в какой мере одновременные команды воздействуют друг на друга. Здесь проявляются "рабочие" различия в функционировании баз данных и способах изоляции результатов выполнения команд.

Работаем на SQL

1. Создайте пространство базы данных с именем Myspace, в котором 15 процентов пространства занимают индексы и 40 процентов используется для размещения строк.
2. Вы передали привилегию выполнения команды SELECT для таблицы Orders пользователю Diane. Введите команду, которая позволит ссылаться на эту таблицу как на таблицу Orders без указания имени Diane в качестве префикса.
3. Если произойдет сбой питания, что случится со всеми изменениями, выполненными в текущей транзакции?
4. Если нельзя просмотреть содержимое строки из-за блокировки, что можно сказать по поводу типа блокировки?
5. Если нужно знать общее число, максимальные значения и средние значения всех заказов, но вы не хотите, чтобы другие пользователи работали в это время с таблицей, то какой тип защиты может оказаться подходящим в этом случае?

(Ответы даны в приложении А.)

24

Как поддерживается порядок в базе данных SQL

Из этой главы вы узнаете, как типичная SQL-база данных поддерживает собственную организацию. Поддержка осуществляется программой, которая создает базу данных и управляет ею. Можно осуществлять доступ к этим таблицам для получения информации о привилегиях, таблицах, индексах и т.д. В этой главе мы рассмотрим типичное содержимое подобной базы данных.

Системный каталог

Для работы с базой данных SQL компьютерная система должна использовать множество объектов: таблицы, представления, индексы, синонимы, привилегии, пользователей и т.д. Существует множество способов их употребления, но наиболее эффективным, логичным, последовательным и разумным из них является реляционный подход к представлению информации в таблицах. Он позволяет компьютеру организовывать необходимую информацию и манипулировать ею, используя те же самые процедуры, что и процедуры для организации и манипулирования самими данными.

Все это относится к уровню реализации программной системы и не является частью стандарта ANSI, но в большинстве систем управления базами данных используется множество SQL-таблиц для представления внутренней информации. Это множество таблиц называется по-разному: *системный каталог (system catalog), словарь данных (data dictionary)* или просто *системные таблицы (system tables)*. (Термин *словарь данных* относится к более широкому понятию, включающему информацию о физических параметрах базы данных, но эти вопросы выходят за рамки рассмотрения SQL. Следовательно, имеется ряд систем управления базами данных, имеющих системный каталог и словарь данных.)

Таблицы системного каталога сходны с другими таблицами SQL: содержат строки и столбцы данных. Например, одна таблица каталога обычно содержит информацию о таблицах в базе данных, причем каждой таблице базы данных соответствует одна строка этой таблицы; другая таблица содержит информацию о столбцах таблиц, причем один столбец таблиц базы данных описывается одной строкой этой таблицы. Таблицы каталога создаются и принадлежат самой базе данных, которая имеет специальное имя SYSTEM. Система управления базами данных создает эти таблицы и обновляет их автоматически по мере использования; таблицы каталога нельзя применять непосредственно в командах обновления. Если отказаться от этого ограничения, то действия системы могут привести к непредсказуемым результатам и к получению совершенно неуправляемой базы данных.

Во многих системах пользователи могут обращаться к каталогу с запросами. Это позволяет выявить специфику конкретной используемой базы данных. Вся информация вообще недоступна для любого пользователя. Доступ к каталогу открыт только для пользователей с соответствующими привилегиями.

Поскольку владельцем каталога является сама система, не существует неоднозначного решения, кто имеет и кто может получить привилегии на работу с каталогом. Обычно такие привилегии передаются суперпользователю, например, системному ад-

министратору с паролем входа в систему SYSTEM или DBA. Некоторые привилегии передаются пользователям автоматически.

Типичный системный каталог

Таблицы, включенные в любой системный каталог:

Таблица	Содержащаяся в таблице информация
SYSTEMCATALOG	Таблицы (базовые и представления)
SYSTEMCOLUMNS	Столбцы таблиц
SYSTEMTABLES	Каталог представлений для SYSTEMCATALOG
SYSTEMINDEXES	Индексы для таблиц
SYSTEMUSERAUTH	Пользователи базы данных
SYSTEMTABAUTH	Объектные привилегии для пользователей
SYSTEMCOLAUTH	Привилегии для столбцов пользователей
SYSTEMSYNONS	Синонимы таблиц

Если DBA передает привилегию на просмотр SYSTEMCATALOG пользователю Stephen с помощью команды:

```
GRANT SELECT ON SYSTEMCATALOG TO Stephen;
```

то Stephen может увидеть некоторую информацию обо всех таблицах в базе данных (предположим, администратор базы данных Chris является владельцем трех таблиц, рассматриваемых в качестве примера, а пользователь Adrian владеет представлением Londoncust).

```
SELECT tname, owner, numcolumns, type, CO
   FROM SYSTEMCATALOG;
```

Выходные данные для этого запроса представлены на рис. 24.1.

Здесь каждая строка представляет таблицу. Первый столбец содержит имя таблицы, второй столбец содержит имя пользователя — владельца этой таблицы, третий столбец определяет количество столбцов в таблице; четвертый столбец содержит однобуквенный код: либо B (для базовой таблицы), либо V (для представления). Последний столбец содержит NULL-значение для случаев, когда в предыдущем столбце указан тип, отличный от V. Этот столбец определяет, следует ли указать возможность контроля.

SYSTEMCATALOG включена в приведенный каталог в качестве одной из таблиц. Для краткости остальная часть системного каталога исключена из выходных данных этой команды. Сами же таблицы, входящие в системный каталог, обычно представлены в 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	

Browse : ↑↓←→ PgDn PgUp ▶| |◀ Home

Рис. 24.1. Содержимое таблицы SYSTEMCATALOG

Использование представлений для таблиц каталога

Поскольку системный каталог SYSTEMCATALOG является таблицей, для него можно использовать представления. Предположим, имеется одно такое представление с названием SYSTEMTABLES. Это представление таблицы SYSTEMCATALOG включает только те таблицы, которые относятся к системному каталогу; обычные таблицы (таблицы базы данных), подобные Salespeople, представлены в SYSTEMCATALOG, а не в SYSTEMTABLES. Предположим, только таблицы каталога принадлежат владельцу с именем SYSTEM. При желании можно определить другое представление, которое исключает все таблицы каталога:

```
CREATE VIEW Datatables
  AS SELECT *
     FROM SYSTEMCATALOG
     WHERE owner <> 'SYSTEM';
```

Разрешение пользователям (только) просматривать их собственные объекты. Существуют другие варианты использования представлений каталога. Предположим, каждый пользователь может запрашивать информацию из каталога, относящуюся к тем таблицам, владельцем которых он является. Поскольку значение USER в командах SQL всегда определяет идентификатор конкретного пользователя, от которого получена команда, это значение можно применить для предоставления пользователю доступа к его собственным таблицам (владельцем которых он является). Можно создать следующее представление:

```
CREATE VIEW Owntables
   AS SELECT *
      FROM SYSTEMCATALOG
      WHERE Owner = USER;
```

Теперь можно передать привилегию доступа к этому представлению всем пользователям:

```
GRANT SELECT ON Owntables TO PUBLIC;
```

Теперь каждый пользователь может выполнить команду SELECT только для тех строк из SYSTEMCATALOG, которые описывают таблицы, владельцем которых он является.

Просмотр SYSTEMCOLUMNS. Можно разрешить каждому пользователю просматривать таблицу SYSTEMCOLUMNS для получения информации о столбцах тех таблиц, владельцем которых он является. Рассмотрим сначала ту часть SYSTEMCOLUMNS, в которой содержатся простые таблицы данного примера (т.е. 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

Каждая строка этой таблицы представляет один столбец таблицы базы данных. Поскольку каждый столбец данной таблицы должен иметь уникальное имя, подобно каждой таблице данного пользователя, комбинации из имени пользователя, имени таблицы и имени столбца являются уникальными. Следовательно, столбцы tname (имя таблицы), tabowner (имя владельца таблицы) и cname (имя столбца) вместе образуют первичный ключ (primary key) этой таблицы. Имя столбца datatype (тип данных) говорит о его содер-

жимом. Столбец cnumber (column number — номер столбца) определяет порядковый номер столбца (расположение столбца) в таблице. Для простоты опущены столбцы, определяющие длину (length), точность (precision) и масштаб (scale) столбцов. Эти атрибуты задают размер столбца, что особенно важно не только для символьных столбцов, но и для числовых.

В таблице SYSTEMCATALOG имеется строка со ссылкой на эту таблицу:

tname	owner	numcolumns	type CO
SYSTEMCOLUMNS	System	8	B

Некоторые SQL-продукты предоставляют больше данных, чем содержится в этих столбцах, но сейчас рассматривается базовый набор этих данных.

Имеется способ разрешить каждому пользователю увидеть информацию таблицы SYSTEMCOLUMNS, описывающую его собственные таблицы (таблицы, владельцем которых он является):

```
CREATE VIEW Owncolumns
   AS SELECT *
      FROM SYSTEMCOLUMNS
      WHERE tabowner = USER;
GRANT SELECT ON Owncolumns TO PUBLIC;
```

Комментарии к содержимому каталога

Большинство версий SQL позволяет использовать комментарии в специально отведенных для этого столбцах таблиц каталогов SYSTEMCATALOG и SYSTEMCOLUMNS, поскольку их содержимое не всегда понятно. Об этих возможностях до сих пор не упоминалось для простоты изложения материала.

Команду COMMENT ON можно применять со строкой текста, предназначенного для пометки каждого столбца в одной из таблиц. Ключевое слово TABLE адресует комментарий для SYSTEMCATALOG, а COLUMN — для SYSTEMCOLUMNS. Например:

```
COMMENT ON TABLE Chris.Orders
   IS 'Current Customer Orders';
```

Текст будет размещен в столбце примечаний таблицы SYSTEMCATALOG.

Максимальная длина примечаний не может превышать 254 символа. Комментарий относится к отдельной строке, в данном случае к той, в которой tname = Orders, a owner = Chris. Этот комментарий находится в строке для таблицы базы данных Orders в SYSTEMCATALOG:

```
SELECT tname, remarks
       FROM SYSTEMCATALOG
       WHERE tname = 'Orders'
```

```
      AND owner = 'Chris';
```

Выходные данные для этого запроса представлены на рис. 24.2.
SYSTEMCOLUMNS работает точно так же. Создаем комментарий:

```
COMMENT ON COLUMN Orders.onum
   IS 'Order #';
```

Выбираем этот столбец из SYSTEMCOLUMNS:

```
SELECT cnumber, datatype, cname, remarks
   FROM SYSTEMCOLUMNS
   WHERE tname = 'Orders'
      AND tabowner = 'Chris'
      AND cname = 'onum';
```

Выходные данные для этого запроса представлены на рис. 24.3.

Для изменения комментария нужно ввести новую команду COMMENT ON для той же самой строки. На место старого комментария заносится новый. Для исключения комментария его следует исключить из формулировки команды. Например:

```
COMMENT ON COLUMN Orders.onum
   IS '';
```

Этот пустой комментарий исключит предыдущее примечание.

```
SQL Execution Log
SELECT tname, remarks
FROM SYSTEMCATALOG
WHERE tname = 'Orders'
AND owner = 'Chris'
;

tname        remarks
Orders       Current Customer Orders

Browse : ↑↓←→ PgDn PgUp ►| |◄ Home
```

Рис. 24.2. Комментарий к SYSTEMCATALOG

```
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	Order #

```
Browse : ↑↓←→ PgDn PgUp →| |← Home
```

Рис. 24.3. Комментарий к SYSTEMCOLUMNS

Оставшаяся часть каталога

Здесь определяются оставшиеся части системных таблиц с примером запроса для каждой из них.

SYSTEMINDEXES — индексы в базе данных

Имена столбцов в таблице SYSTEMINDEXES и их описание представляют собой следующее:

Имя столбца	**Описание**
iname	Имя индекса (используется для его удаления)
iowner	Имя пользователя, создавшего индекс
tname	Имя таблицы, которая содержит индекс
cnumber	Номер столбца в таблице
tabowner	Пользователь, владеющий таблицей, содержащей индекс
numcolumns	Количество столбцов в индексе
cposition	Позиция текущего столбца среди столбцов в индексе
isunique	Является ли индекс уникальным (Y или N)

Пример запроса. Предполагается, что есть неуникальный индекс, имеющий название salesperson, определенный на столбце snum таблицы Customers:

```
SELECT iname, iowner, tname, cnumber, isunique
   FROM SYSTEMINDEXES
   WHERE iname = 'salesperson';
```

Выходные данные для этого запроса представлены на рис. 24.4.

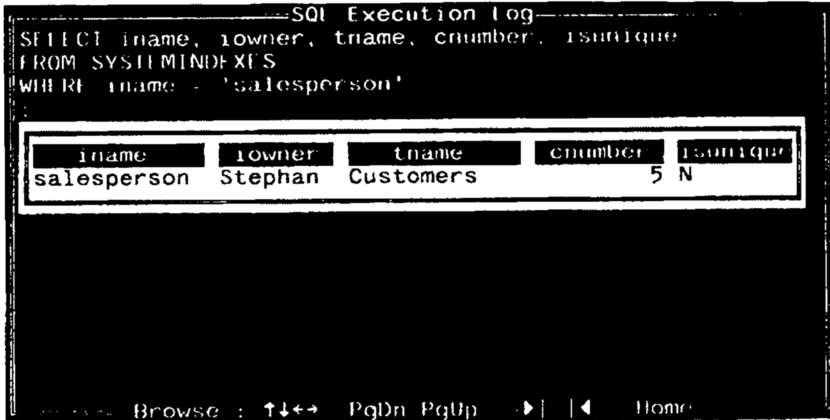


Рис. 24.4. Строка из таблицы SYSTEMINDEXES

SYSTEMUSERAUTH — пользователи и системные привилегии в базе данных

Имена столбцов для SYSTEMUSERAUTH и их описание выглядят следующим образом:

Имя столбца	**Описание**
username	Идентификатор (authorization ID) пользователя
password	Пароль пользователя для начала сеанса
resource	Имеет ли пользователь привилегию RESOURCE
dba	Имеет ли пользователь привилегию DBA

Предполагается простая система из трех привилегий, подобная той, которая рассматривалась в главе 22: CONNECT, RESOURCE и DBA. Все пользователи имеют привилегию CONNECT по определению, поэтому она не указана в таблице. Возмож-

Рис. 24.5. Пользователи, имеющие привилегию RESOURCE

ными значениями столбцов resource и dba являются Y (если пользователь имеет эту привилегию) и N (если пользователь не имеет этой привилегии). Пароли доступны только высокопривилегированным пользователям. Следовательно, эта таблица может запрашиваться с целью получения информации пользователями, имеющими системные привилегии.

Пример запроса. Для того, чтобы найти всех пользователей, имеющих привилегию RESOURCE, и увидеть, кто из них располагает привилегией DBA, можно ввести следующее предложение:

```
SELECT username, dba
   FROM SYSTEMUSERAUTH
   WHERE resource = 'Y';
```

Выходные данные для этого запроса представлены на рис. 24.5.

SYSTEMTABAUTH — привилегии объектов, которые не соответствуют специфике столбцов

Здесь представлены имена столбцов в таблице SYSTEMTABAUTH и их описания:

Имя столбца **Описание**

username	Пользователь, имеющий привилегии
grantor	Пользователь, передавший привилегии пользователю с данным именем
tname	Имя таблицы, для которой существуют привилегии
owner	Владелец tname
selauth	Имеет ли пользователь привилегию SELECT
insauth	Имеет ли пользователь привилегию INSERT
delauth	Имеет ли пользователь привилегию DELETE

В таблице представлены значения для каждой объектной привилегии (названия столбцов для них заканчиваются на auth); это может быть Y, N или G. G означает, что пользователь имеет привилегию с правом ее передачи. В каждой строке значение по крайней мере одного столбца должно быть отлично от N.

Первые четыре столбца этой таблицы формируют ее первичный ключ. Это означает, что каждая комбинация значений из столбцов tname, owner (следует помнить, что две разные таблицы, принадлежащие различным владельцам, могут иметь одно имя), user и grantor должна быть уникальной. Каждая строка в этой таблице содержит все привилегии (не на уровне столбца), переданные одним конкретным пользователем другому конкретному пользователю для отдельного объекта. UPDATE и REFERENCES являются привилегиями, которые могут быть определены для столбца и хранятся в другой таблице каталога. Если пользователь получает привилегии на таблицу не от одного, а от многих пользователей, то в ней создаются отдельные строки. При отмене привилегий это вынуждает применять процедуру каскада.

Пример запроса. Для того, чтобы найти все привилегии SELECT, INSERT и DELETE, которые Adrian передал для таблицы Customers, нужно ввести следующее предложение (выходные данные показаны на рис. 24.6):

```
SELECT username, selauth, insauth, delauth
    FROM SYSTEMTABAUTH
    WHERE grantor = 'Adrian'
        AND tname = 'Customers';
```

Из примера ясно, что Adrian передал Claire привилегии NSERT и SELECT для таблицы Customers, причем последняя привилегия передана с правом ее дальнейшей передачи. Norman он передал привилегии SELECT, INSERT и DELETE, но ни для одной из них не предоставил права передачи. Если Claire получает привилегию DELETE для таблицы Customers из какого-либо другого источника, то этот факт не будет отражен в результате данного конкретного запроса.

```
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

Browse : ↑↓←→  PgDn PgUp --▶|  |◀-- Home
```

Рис. 24.6. Пользователи, получившие привилегии от пользователя Adrian

SYSTEMCOLAUTH — привилегии, соответствующие специфике столбцов

Имя столбца	**Описание**
username	Пользователь, имеющий привилегии
grantor	Пользователь, передавший привилегии пользователю с данным именем
tname	Имя таблицы, для которой существуют привилегии
cname	Имя столбца, для которого существуют привилегии
owner	Владелец tname
updauth	Имеет ли пользователь привилегию UPDATE для данного столбца
refauth	Имеет ли пользователь привилегию REFERENCES для данного столбца

Поля updauth и refauth могут принимать значения Y, N или G; в одной и той же строке оба эти поля не могут иметь одно и то же значение N. Первые пять столбцов этой таблицы формируют первичный ключ. Он отличается от первичного ключа таблицы SYSTEMTABAUTH и включает поле cname, определяющее отдельный столбец рассматриваемой таблицы, на который распространяются одна или обе рассматривае-

мые привилегии. Отдельная строка в этой таблице существует для каждого столбца любой данной таблицы, для которого один пользователь передал права, специфицирующие столбец, другому пользователю. Как и в случае с таблицей SYSTEMTABAUTH, одна и та же привилегия может присутствовать более чем в одной строке этой таблицы, если она была передана более чем одному пользователю.

Пример запроса. Чтобы определить, для какого столбца какой таблицы имеется привилегия REFERENCES, нужно ввести следующее (выходные данные представлены на рис. 24.7):

```
SELECT owner, tname, cname
   FROM SYSTEMCOLAUTH
   WHERE refauth IN ('Y','G')
     AND username = USER
   ORDER BY 1, 2;
```

Результат иллюстрирует факт, что две таблицы, имеющие разных владельцев, но одно и то же имя, являются двумя различными таблицами (не путать с двумя синонимами для единственной 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	snum
Gillan	Customers	cnum

Browse : ↑↓←→ PgDn PgUp ▶| |◀-- Home

Рис. 24.7. Столбцы, для которых пользователи имеют привилегию

SYSTEMSYNONS — синонимы таблиц базы данных

Ниже представлены имена столбцов в таблице SYSTEMSYNONS и их описание:

Имя столбца	**Описание**
synonym	Имя синонима
synowner	Пользователь, являющийся владельцем синонима (может быть PUBLIC)
tname	Имя таблицы, используемой в качестве владельца
tabowner	Имя пользователя, владеющего таблицей

Пример запроса. Предположим, пользователь Adrian имеет синоним Clients для таблицы Customers, владельцем которой является Diane, и что для этой таблицы есть общедоступный синоним Customers. Можно сформулировать запрос по поводу всех синонимов таблицы Customers (выходные данные представлены на рис. 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
Browse : ↑↓←→  PgDn PgUp —▶|  |◀— Home
```

Рис. 24.8. Синонимы таблицы Customer

Другие пользователи каталога

Можно сформулировать более сложные запросы для системного каталога. Например, выполнить операцию соединения. Эта команда позволяет увидеть столбцы таблиц и индексы, базирующиеся на этих столбцах (выходные данные представлены на рис. 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;
```

В результате представлены два индекса для каждой из таблиц Customers и Salespeople. Последняя таблица имеет индекс с именем salesno, определенный на одном столбце snum; он был представлен первым, поскольку строки в таблице приводятся по убыванию (в соответствии с порядком, обратном алфавитному) значений столбца iname. Индекс custsale используется продавцами для поиска их покупателей. Он базируется на

```
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	snum	salesno	1
Customers	cnum	custsale	2
Customers	snum	custsale	1

Browse : ↑↓←→ PgDn PgUp - ▶| |◀ - Home

Рис. 24.9. Столбцы и их индексы

комбинации полей snum и cnum для таблицы Customers, причем, поле snum входит в состав индекса первым, о чем свидетельствует значение поля cposition.

Можно использовать и подзапросы. Ниже приводится способ просмотра данных столбцов, принадлежащих только таблицам каталога:

```
SELECT *
   FROM SYSTEMCOLUMNS
   WHERE tname IN
      (SELECT tname
           FROM SYSTEMCATALOG);
```

Для краткости опущены выходные данные для этой команды, которые содержат единственное значение для каждого столбца каждой таблицы каталога. Этот запрос можно разместить в представлении SYSTEMTABCOLS, чтобы перейти к представлению SYSTEMTABLES.

Итоги

Итак, SQL-системы используют множество таблиц, которое называется системным каталогом структуры базы данных. Для этих таблиц можно формулировать запросы, но их нельзя обновлять. Кроме того, можно добавлять столбцы комментариев в таблицы SYSTEMCATALOG и SYSTEMCOLUMNS (а также удалять их). Создание представлений для этих таблиц — превосходный способ точно определить ту информацию, к которой пользователи имеют право доступа.

На этом мы заканчиваем рассмотрение SQL в интерактивном режиме. В следующей главе будут рассмотрены вопросы применения SQL непосредственно в программах, написанных на языках программирования; такое использование позволяет извлечь преимущества взаимодействия программы с базой данных.

Работаем на SQL

1. Сформулируйте запрос к каталогу с целью получить для каждой таблицы, имеющей более четырех столбцов, имя таблицы, ее владельца, а также имена и типы данных для столбцов.
2. Сформулируйте запрос к каталогу с целью определить, сколько синонимов существует для каждой таблицы базы данных. Помните, что одноименные синонимы, принадлежащие различным владельцам, следует рассматривать как два различных синонима.
3. Определите количество таблиц, имеющих индексы для более пятидесяти процентов столбцов.

(Ответы даны в приложении A.)

25

Использование SQL с другими языками программирования (встроенный SQL)

Эта глава научит вас использовать SQL для усиления возможностей программ, написанных на других языках программирования. Непроцедурный характер SQL придает ему выразительную силу и влечет ряд ограничений. Для освобождения от ограничений можно встраивать SQL в программы, написанные на том или ином языке программирования. Для примеров в этой главе выбран язык Pascal, поскольку он является простейшим и имеет (полуофициальный) стандарт ANSI.

Что включается во встроенный SQL?

Чтобы встроить SQL в другой язык программирования, следует воспользоваться программным продуктом, позволяющим включить SQL в этот язык и применять конструкции SQL так же естественно, как и конструкции самого языка. При этом нужно знать язык программирования. Команды SQL используются для работы с таблицами базы данных, для передачи выходных данных в программу, в которую встроен SQL, и для ввода данных из программы в таблицы базы данных. Программа называется *включающей* (host program) (она может получать данные или передавать их пользователю в интерактивном режиме, а может и не делать этого).

Почему SQL называется встроенным?

В этой книге много говорилось о возможностях SQL. SQL состоит из множества отдельных команд, в то время как интерактивный SQL обычно выполняется по одной команде за один такт работы SQL-системы. Логические конструкции, используемые в большинстве структурных языков программирования, такие как if ... then, for ... do и while ... repeat, в SQL отсутствуют, и не имеется базы для принятия решения: стоит ли выполнять действие, как выполнять действие и как долго выполнять действие, основываясь на результатах его последнего выполнения. Кроме того, интерактивный SQL не может выполнять никаких действий со значениями, за исключением их ввода в таблицы, их нахождения и вывода с помощью запросов непосредственно на какое-либо устройство.

Но более традиционные языки программирования являются в этом плане более мощными. Они проектировались и разрабатывались так, чтобы программист мог начать процесс и, базируясь на его результатах, решить, какое именно действие требуется выполнить, либо повторять действие до тех пор, пока удовлетворяется некоторое условие, создавая логические пути и циклы. Значения хранятся в переменных, которые могут использоваться и изменяться любым количеством команд. Это дает возможность выдавать пользователям приглашение для ввода значений или для чтения их из файла, форматировать выходные данные подходящим образом (например, преобразовывать числовые данные в диаграммы).

Назначение встроенного SQL — комбинировать эти возможности языка программирования, давая возможность разрабатывать сложные процедурные программы, которые обращаются к базе данных через SQL, скрывая от пользователя сложности работы с таблицами в процедурном языке, не ориентированном на работу с такими структурами данных, но имеющем мощные управляющие структуры.

Как встраивается SQL?

Команды SQL размещаются в основном тексте программы на языке высокого уровня, им предшествует фраза EXEC SQL (аббревиатура от EXECute SQL — выполнить SQL). Это позволят включить команды, специфичные для встроенной формы SQL. Строго говоря, стандарт ANSI не поддерживает встроенного SQL как такового, а поддерживает *модуль (module)*, который рассматривается как множество процедур SQL, *вызываемых* из другого языка программирования, но не *встроенных* в него. Определение официального синтаксиса встроенного SQL, включающего расширенный официальный синтаксис для каждого языка, в который SQL должен быть встроен, — длительная и неблагодарная работа, которую старается избегать ANSI. Но он предоставляет четыре приложения (не являющиеся частью стандарта), определяющие синтаксис расширенного SQL для четырех языков: COBOL, Pascal, FORTRAN, PL/I. Могут использоваться язык С и другие языки программирования.

Когда команды SQL встраиваются в текст программы, написанной на каком-либо языке программирования, необходимо выполнить *предкомпиляцию (precompile*) программы перед выполнением ее компиляции. Программа, названная *предкомпилятором)* или *препроцессором — (precompiler* или *preprocessor)*, просматривает текст вашей программы и преобразует команды SQL в форму, совместимую с основным языком программирования. Затем, как обычно, применяется компилятор для преобразования программы из основного кода в выполняемый.

В соответствии с методом модульного языка программирования, определенного в ANSI, основная программа осуществляет вызовы SQL-процедур. Эти процедуры принимают значения параметров и возвращают их в основную (вызвавшую их) программу. Модуль может содержать любое количество процедур, каждая из которых состоит из единственной команды SQL. Основная идея заключается в одинаковом функционировании этих программ независимо от языка программирования, в который они встроены (из которого они вызваны) (хотя модуль должен также идентифицировать включающий язык из-за различий в типах данных для разных языков программирования).

Конкретные реализации отличаются от стандарта встроенного SQL тем, что модули могут непосредственно определяться в тексте программы. В общем случае препроцессор создает модуль, называемый *модулем доступа (access module)*. Для данной программы может существовать только один модуль, содержащий любое количество SQL-процедур. Размещение SQL-предложения непосредственно в коде включающего языка проще и более практично, чем непосредственное создание самих модулей.

Программы, использующие встроенный SQL, при выполнении также связаны с идентификатором пользователя. Идентификатор пользователя, связанный с программой, должен иметь все привилегии для выполнения операций SQL, размещенных в программе. В общем случае программа со встроенным SQL связывается с базой данных как и пользователь, выполняющий программу. Детали определяются реализацией, но необходимо включить в программу команду CONNECT или ее аналог, реализующий эту команду в полной мере.

Использование переменных языка высокого уровня с SQL

Части программы SQL и язык высокого уровня взаимодействуют в основном с помощью значений переменных. Различные языки программирования распознают различные типы переменных. ANSI определяет эквиваленты SQL для четырех языков программирования — PL/I, Pascal, COBOL, FORTRAN (детали представлены в приложении В). Эквиваленты для других языков программирования определяются разработчиками. Типы, подобные DATE, не распознаются ANSI; следовательно, в стандарте нет эквивалентных типов данных для языков высокого уровня. Более сложные типы данных языка высокого уровня, такие как матрицы, тоже не имеют эквивалентов в SQL.

Можно использовать переменные из основной программы во встроенных предложениях SQL везде, где применяются выражения значений (SQL, о котором идет речь в этой главе, — это встроенный SQL, если не оговорено нечто иное). Текущее значение переменной — это то значение, которое будет использоваться в команде. Переменные языка высокого уровня должны:

- Быть объявлены в SQL DECLARE SECTION (кратко обсуждается далее);
- Быть совместимыми по типу данных с их функциями в команде SQL (например, иметь числовой тип, если они должны быть вставлены в числовое поле);
- Иметь значение к тому моменту времени, когда они используются в SQL-команде, независимо от того, что сама SQL-команда может назначать значение;
- Перед ними должно быть указано двоеточие (:) тогда, когда на нее ссылается SQL-команда.

Поскольку переменные языка высокого уровня отличаются от имен столбцов SQL наличием двоеточия, при желании можно использовать переменные с теми же самыми именами, что и имена столбцов.

Предположим, в программе есть четыре переменные, с именами id_num, salesperson, loc, comm. Они содержат значения, которые нужно вставить в таблицу Salespeople. Можно ввести в программу следующую SQL-команду:

```
EXEC SQL INSERT INTO Salespeople
    VALUES (:id_num, :salesperson, :loc, :comm)
```

Текущие значения этих переменных будут вставлены в таблицу. Переменная comm имеет то же самое имя, что и столбец. Именно это значение будет вставлено, остальные переменные вставляться не будут. Следует также заметить, что точка с запятой в конце команды опущена, потому что соответствующее завершение для команды встроенного SQL различно для разных языков программирования. Для Pascal и PL/I — это точка с запятой; для COBOL — слово END-EXEC; для FORTRAN не используется никакого завершающего символа. В других языках программирования этот символ зависит от конкретной реализации, но мы будем использовать двоеточие для совместимости встроенного SQL с Pascal. Для Pascal встроенные команды SQL и его собственные команды заканчиваются точкой с запятой.

Можно расширить функциональные возможности рассмотренной команды, вставив ее в цикл и повторяя многократно с различными значениями переменных, например:

```
while not end-of-file (input) do
    begin
    readln (id_num, salesperson, loc,comm);
    EXEC SQL INSERT INTO Salespeople
        VALUES (:id_num, :salesperson, :loc,:comm);
    end;
```

Этот фрагмент программы Pascal определяет цикл, который будет читать значения из файла, запоминать их в четырех переменных памяти и сохранять значения этих переменных в таблице Salespeople, после этого читать следующие четыре значения, повторяя процесс до тех пор, пока не будет прочитано содержимое всего файла. Предполагается, что каждое множество значений вводится нажатием клавиши Enter (возврата каретки) (именно так принято в Pascal, функция readln читает входные данные и переходит к следующей строке). Это дает возможность передать данные из текстового файла в реляционную структуру. Перед занесением данных в реляционную структуру можно предварительно их обработать средствами языка высокого уровня, например, исключить те данные (те сведения о продавцах), для которых комиссионные меньше .12:

```
while not end-of-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 те строки, которые удовлетворяют условию comm >= .12, вставляются в таблицу. Этот пример показывает, как можно использовать и циклы, и условия в языке высокого уровня в сочетании со средствами встроенного SQL.

Объявление переменных

Все переменные, на которые есть ссылки в SQL-предложениях, прежде всего должны быть объявлены в SQL DECLARE SECTION с применением синтаксиса включающего языка. В программе может быть любое количество таких секций, и они могут располагаться в любом месте по тексту программы, но перед использованием этих переменных все прочие ограничения определяются ограничениями включающего языка. Секции объявлений должны начинаться и заканчиваться командами встроенного SQL BEGIN DECLARE SECTION и END DECLARE SECTION, которым предшествует, как и прежде, EXEC SQL. Для объявления переменных, которые использовались в предыдущем примере, можно ввести следующее:

```
EXEC SQL BEGIN DECLARE SECTION;
Var
    id-num:         integer;
    Salesperson:    packed array (1..10) of char;
    loc:            packed array (1..10) of char;
    comm:           real;
EXEC SQL END DECLARE SECTION;
```

Для тех, кто не знаком с языком программирования Pascal, следует заметить, что Var — это заголовок, который предшествует серии объявлений переменных и упакованных (представленных компактно) (или неупакованных) массивов; массив — последовательность значений одного типа, различающихся порядковым номером, указанным в круглых скобках (например, третий символ в массиве loc имеет официальное имя loc(3)). Использование точки с запятой после определения каждой переменной — это требование Pascal, а не SQL.

Поиск значений в списке переменных

Кроме ввода значений переменных в таблицу с применением SQL-команд, SQL можно применить для получения значений переменных из таблиц базы данных. Один из способов сделать это — использовать модификации команды SELECT, содержащей предложение INTO. В отличие от предыдущего примера, здесь введены значения строки, описывающей Peel, из таблицы Salespeople и сохранены в наборе переменных языка высокого уровня:

```
EXEC SQL SELECT snum, sname, city, comm
    INTO :id_num, :salesperson, :loc, :comm
    FROM Salespeople
    WHERE snum = 1001;
```

Выбранные значения размещаются (запоминаются, сохраняются) в переменных, имена которых указаны в предложении INTO в порядке, установленном в

этом предложении. Переменные, имена которых указаны в предложении INTO, должны иметь соответствующие типы для сохранения этих значений; для каждого выбираемого столбца должна быть предусмотрена переменная соответствующего типа.

За исключением представленного предложения INTO, этот запрос ничем не отличается от рассмотренных ранее. Но предложение INTO накладывает серьезное ограничение на запрос: результатом поиска в запросе может быть только одна строка. Если в результате выполнения запроса получено несколько строк, то значения всех этих строк не могут одновременно храниться в указанном наборе переменных. Команда признается ошибочной. По этой причине команду SELECT INTO можно выполнять только в том случае, если выполняются следующие условия:

- Когда предикат применяется для проверки значения, нужно иметь уверенность в том, что это значение уникально, как в рассмотренном примере. Значения, о которых точно известно, что они уникальны, — это значения, удовлетворяющие ограничению уникальности или уникального индекса (см. главы 17 и 18);

- Когда используются одна или более агрегатных функций, но не применяется GROUP BY;

- Когда для внешнего ключа используется SELECT DISTINCT с предикатом, ссылающимся на единственное значение родительского ключа (при условии, что система поддерживает ссылочную целостность), как в примере:

```
EXEC SQL SELECT DISTINCT snum
    INTO :salesnum
    FROM Customers
    WHERE snum =
        (SELECT snum
             FROM Salespeople
             WHERE sname = 'Motika');
```

В предположении, что Salespeople.sname и Salespeople.snum являются уникальным и первичным ключами соответственно для таблицы Salespeople и что Customers.snum является внешним ключом, ссылающимся на Salespeople.snum, есть гарантия того, что результатом ответа на этот запрос является единственная строка.

Существуют и другие случаи, когда можно быть твердо уверенным в том, что результат выполнения запроса — единственная строка выходных данных, но в общем случае такие ситуации плохо поддаются описанию и часто зависят от конкретных данных. Однако в программировании нельзя полагаться на случайности. На разработку программы затрачивается определенное время, и лучше быть уверенным в надежности ее работы при условии возможности изменения данных. Нет необходимости выискивать запросы, результатом выполнения которых является одна строка, как того

требует SELECT INTO. Можно использовать и запросы, результатом выполнения которых является множество строк. Для этого следует использовать курсор.

Курсор

Сила SQL заключается и в его способности работать со множеством строк таблицы, удовлетворяющих какому-либо критерию, не уточняя заранее, о каком конкретно количестве строк идет речь. Если десять строк удовлетворяют предикату, то результатом выполнения запроса будут эти десять строк. Если предикату удовлетворяют десять миллионов строк, то результатом выполнения запроса будут все десять миллионов строк. Но это трудно себе представить, если имеется в виду взаимодействие с другими языками программирования. Как можно назначить выходные данные запроса переменным, если неизвестно даже их количество? Решить эту проблему можно с помощью курсора.

Курсор — это мигающая метка текущей позиции на экране дисплея компьютера. Курсор SQL — устройство, аналогичное курсору на экране дисплея, которое помечает выходные данные запроса, хотя такую аналогию следует признать весьма натянутой. Реально, разработчики SQL ввели понятие "курсор" безотносительно к использованию этого понятия где-либо еще.

Курсор (cursor) — это переменная, связанная с запросом. Значение этой переменной — каждая строка, удовлетворяющая запросу. Подобно переменным включающего языка, курсоры должны быть описаны перед их использованием. Это делается с помощью команды DECLARE CURSOR следующим образом:

```
EXEC SQL DECLARE CURSOR Londonsales FOR
    SELECT *
        FROM Salespeople
        WHERE city = 'London';
```

Запрос не выполняется немедленно, а является лишь определением. Курсоры подобны представлениям в том, что содержат в своем определении запрос, а их содержимое — выходные данные запроса, выполняющегося в момент открытия курсора. Но, в отличие от базовых таблиц или представлений, строки курсора упорядочены, различают первую, вторую, ... последнюю строки курсора. Этот порядок может либо явно задаваться с помощью предложения ORDER BY в запросе, либо может быть произвольным, в соответствии с соглашениями, принятыми в конкретной SQL-системе.

Когда достигается место в программе, где необходимо выполнить запрос, курсор открывается с помощью команды:

```
EXEC SQL OPEN CURSOR Londonsales;
```

Значения курсора определяются на момент выполнения этой команды, а не на момент выполнения команды DECLARE или команды FETCH, которая извлекает выходные данные запроса по одной строке на одно выполнение команды FETCH:

```
EXEC SQL FETCH Londonsales INTO :id_num, :salesperson, :loc, :comm;
```

По этой команде осуществляется ввод значений из первой строки, удовлетворяющей запросу, в набор указанных переменных. В результате выполнения команды FETCH получается следующий набор значений переменных. Идея заключается в использовании команды FETCH в цикле для получения очередной строки, удовлетворяющей запросу, в выполнении необходимых действий над значениями из этой строки, в переходе к получению следующей строки курсора и сохранению ее значений в том же наборе переменных. Допустим, пользователю предоставляется возможность просмотра выходных данных запроса по одной строке в один момент времени и выдается запрос по поводу того, хочет ли он продолжить просмотр выходных данных запроса (получить на экране дисплея 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 ('Желаете ли вы продолжить просмотр данных? (Y/N)');
        readln (response);
        if response = 'N' then Look_at_more := False;
        end;
    EXEC SQL CLOSE CURSOR Londonsales;
```

В Pascal := означает "присвоить значение", тогда как = имеет свое обычное назначение. Функция writeln выводит на экран дисплея указанные в ней данные и затем выполняет установку на новую строку. Символьное значение заключается в одиночные кавычки, используется во втором writeln и в предложении if ... then. Это соглашение Pascal совпадает с соглашением SQL.

Приведенный в примере фрагмент работает таким образом: сначала логической переменной с именем Look_at_more присваивается значение true (истина), затем открывается курсор, затем работает цикл. Внутри цикла строка извлекается из курсора и выводится на экран дисплея. Затем появляется вопрос для пользователя: желает ли он получить на экране дисплея следующую строку. Если ответом не является N (НЕТ), то цикл повторяется, и извлекается следующая строка значений. Переменные Look_at_more и response должны быть объявлены с типом Boolean и char соответственно в разделе объявления переменных Pascal, но их не следует включать в раздел объявления переменных SQL. В тексте программы содержится предложение CLOSE CURSOR, соответствующее предложению CLOSE CURSOR. По этой команде разрывается связь между курсором и конкретными значениями, значит следует повторно выполнить запрос с предложением OPEN CURSOR прежде, чем станет возможным извлечение значений из курсора. Не требуется извлекать последовательно все значения курсора прежде, чем возможно будет закрыть его, хотя на практике такой вариант использова-

ния встречается очень часто. После закрытия курсора SQL не отслеживает, какие строки были извлечены. Если снова открыть курсор, то запрос выполнится повторно именно в момент повторного открытия курсора и можно начинать работу с ним заново, независимо от предыдущего использования этого курсора.

Пример не предоставляет никакого автоматического выхода из цикла после обработки всех строк курсора. Когда все строки запроса извлечены с помощью команды FETCH, следующая команда FETCH не изменяет значений переменных, указанных в предложении INTO. Следовательно, поскольку данные из курсора исчерпаны, на экран дисплея повторно выводятся те же самые значения, и это будет продолжаться до тех пор, пока пользователь не отменит выполнения цикла, введя в качестве ответа N.

SQLCODE

Вы должны знать, когда исчерпаются данные курсора; при этом можно выдать пользователю соответствующее сообщение и выйти из цикла автоматически. Это важно, так как в процессе выполнения SQL-команды может возникнуть ошибка. Переменная SQLCODE (SQLCOD — в FORTRAN) может помочь в решении этой проблемы. Ее следует определить как переменную языка высокого уровня, а ее тип данных должен соответствовать одному их из числовых типов SQL (см. приложении В). Значение SQL устанавливается всякий раз, когда выполняется команда SQL. Существуют три основные возможности:

1. Команда выполнилась без ошибок, но не дала никакого результата. Здесь ситуация зависит от конкретной команды следующим образом:

a) Для команды SELECT это свидетельствует о том, что ни одна строка не удовлетворяет запросу;

b) Для команды FETCH это свидетельствует о том, что последняя строка уже была извлечена, либо о том, что ни одна строка не удовлетворяет запросу, указанному в курсоре;

c) Для команды INSERT это свидетельствует о том, что никакие строки не были вставлены в таблицу (это может произойти, например, в том случае, когда определяются значения, предназначенные для вставки в таблицу с помощью запроса, но в результате выполнения запроса не найдено ни одной строки);

d) Для команд UPDATE и DELETE это свидетельствует о том, что ни одна строка не удовлетворяет условию, указанному в предикате, и, следовательно, в таблице не было сделано никаких изменений.

В любом из этих случаев значение SQLCODE устанавливается равным 100.

2. Команда выполняется успешно, ни один из перечисленных в пункте 1 случаев не имеет места, значение SQLCODE устанавливается равным 0.

3. Команда генерирует ошибку. В этом случае все изменения, выполненные в базе данных на протяжении выполнения текущей транзакции, игнорируются. И SQLCODE принимает одно из отрицательных значений, определенных разработчиками SQL-системы. Присвоение отрицательного значения — идентифицировать проблему наиболее точно. Система предоставляет возможность выполнить подпрограмму, которая имеет информацию о соответствии отрицательных значений реальным ситуациям, которое определено разработчиками системы. Иногда сообщения об ошибках выдаются на экран дисплея или в файл, а программа отказывается от выполненных текущей транзакцией изменений, разрывает связь с базой данных и завершает свое выполнение.

Использование SQLCODE для управления циклами

Необходимо дополнить рассмотренный ранее пример средствами автоматического выхода из цикла для случая, когда курсор оказался пустым, либо из курсора извлечены все строки, либо в процессе выполнения возникла ошибка:

```
Look_at_more := True;
EXEC SQL OPEN CURSOR Londonsales;
    while Look_at_more
    and SQLCODE = 0 do
        begin
        EXEC SQL FETCH Londonsales
             INTO :id_num, :Salesperson, :loc, :comm;
        writeln (id_num, Salesperson, loc, comm);
        writeln ('Желаете ли вы продолжить просмотр данных? (Y/N)');
        readln (response);
        if response = 'N' then Look_at_more := False;
        end;
EXEC SQL CLOSE CURSOR Londonsales;
```

WHENEVER

Прекрасно, когда есть возможность выхода из цикла после извлечения всех строк курсора, но если допущена ошибка, на нее нужно отреагировать с учетом рассмотренного пункта 3. Для этого в SQL предназначены предложения GOTO. Фактически, они дают возможность определить данные действия глобально, в результате чего программа выполняет команду GOTO автоматически, если в процессе выполнения программы вырабатывается определенное значение SQLCODE. Это можно сделать с помощью предложения WHENEVER. Два примера по этому поводу:

```
EXEC SQL WHENEVER SQLERROR GOTO Error_handler;
```

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

SQLERROR — один из способов выразить SQLCODE < 0; NOT FOUND — один из способов выразить SQLCODE = 100. (В некоторых программных продуктах для обозначения последней ситуации можно использовать SQLWARNING.) Error_handler — это имя места в программе, начиная с которого продолжается выполнение программы в том случае, если обнаружена ошибка (GOTO можно записать как одно слово или как два слова). Это имя определяется в зависимости от конкретного включающего языка, оно может быть меткой (label) в Pascal, именем секции или именем параграфа в COBOL (здесь и далее будет использоваться термин "метка"). Наиболее предпочтительным может быть указание предлагаемой разработчиками системы стандартной процедуры, которая предназначена для обработки ошибочных ситуаций и может быть включена во все программы.

CONTINUE означает, что не требуется предпринимать никаких специальных действий для обработки значения SQLCODE. Это значение принимается по умолчанию, если не используется команда WHENEVER для определения значения SQLCODE. Многократно размещая такие предложения по тексту программы, можно предпринимать какие-либо действия в программе или не предпринимать их.

Например, если программа включает серию команд INSERT, использующих запросы, которые реально выдают значения, можно напечатать специальное сообщение или сделать что-то, что дает возможность определить, какие из запросов возвращают пустые значения, и вставки значений не происходит. В этом случае можно ввести следующее:

```
EXEC SQL WHENEVER NOT FOUND GOTO No_rows;
```

No_rows — это метка кода, выполняющего соответствующее действие. Если далее в этой программе выполняется извлечение данных, то можно ввести следующее

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

поскольку повторное извлечение данных во время извлечения строк является нормальной процедурой и не требует специального управления.

Обновление курсоров

Курсоры применяются для выбора из таблицы групп строк, которые можно обновить или удалить по одной. Это дает возможность сформулировать ряд ограничений на предикаты, используемые в командах UPDATE и DELETE. Можно сослаться на рабочую таблицу, для которой сформулирован предикат запроса курсора или подзапросы этого запроса, чего нельзя сделать в самих предикатах этих команд. Стандарт SQL не позволяет воспользоваться следующей формой записи для удаления всех покупателей с рейтингами ниже среднего:

```
EXEC SQL DELETE FROM Customers
```

```
    WHERE rating <
        (SELECT AVG (rating)
            FROM Customers);
```

Но можно получить такой же эффект, используя запрос для соответствующих строк, хранящихся в этом курсоре, и выполнив команду DELETE с использованием этого курсора. Во-первых, следует определить курсор, например, так:

```
EXEC SQL DECLARE Belowavg CURSOR FOR
    SELECT *
        FROM Customers
        WHERE rating <
            (SELECT AVG (rating)
                FROM Customers);
```

Затем можно создать цикл для удаления всех покупателей, выбранных с помощью этого курсора:

```
EXEC SQL WHENEVER SQLERROR GOTO Error_handler;
EXEC SQL OPEN CURSOR Belowavg;
while not SQLCODE = 100 do
    begin
    EXEC SQL FETCH Belowavg INTO :a, :b, :c, :d, :e;
    EXEC SQL DELETE FROM Customers
        WHERE CURRENT OF Belwavg;
    end;
EXEC SQL CLOSE CURSOR Belowavg;
```

Предложение WHERE CURRENT OF означает, что DELETE применяется к строкам, найденным с помощью курсора. Это предполагает, что курсор и команда DELETE ссылаются на одну и ту же таблицу и, следовательно, запрос в курсоре не является соединением. Курсор может быть также обновляемым. Чтобы быть обновляемым, курсор должен удовлетворять тому же критерию, что и представление. ORDER BY и UNION, которые нельзя применять в представлениях, можно использовать в курсорах, но это лишает курсор свойства обновляемости. Необходимо извлекать строки из курсора в множество переменных, даже если их значения не используются. Этого требует синтаксис команды FETCH.

Команда UPDATE работает аналогично. Можно увеличить комиссионные тех продавцов, которые имеют покупателей с рейтингом 300. Во-первых, определим курсор:

```
EXEC SQL DECLARE CURSOR High_Cust AS
    SELECT *
        FROM Salespeople
        WHERE snum IN
            (SELECT snum
```

```
                    FROM Customers
                    WHERE rating = 300);
```

Затем можно вести обновления в цикле:

```
EXEC SQL OPEN CURSOR High_cust;
while SQLCODE = 0 do
    begin
    EXEC SQL 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;
```

Замечание: некоторые программные продукты требуют, чтобы в определении курсора было указано, что он будет использоваться для выполнения команд UPDATE для определенных столбцов. Это делается добавлением в определение курсора FOR UPDATE OF <*список столбцов*>. Чтобы определить курсор High_cust для выполнения обновления значений столбца comm, нужно 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 обеспечивает своего рода защиту от нежелательных изменений, которые могут привести к весьма тяжелым последствиям.

Индикаторы переменных

NULL — это специальные маркеры, определенные в SQL. Их нельзя использовать в переменных включающего языка. Попытка вставить NULL в переменную включающего языка является некорректной, поскольку включающие языки не поддерживают NULL-значений, определенных в SQL. Хотя результат попытки вставить NULL-значения во включающий язык зависит от конкретной реализации, результат, определенный в теории баз данных, создает ошибочную ситуацию: SQLCODE получает отрицательное числовое значение, и может быть вызвана программа обработки

ошибочных ситуаций. Обычная реакция — попытка избежать этого. Часто можно выбирать NULL-значения наряду со реальными, чтобы избежать аварийного завершения программы. Если программа не завершается автоматически, значения в переменных включающего языка будут некорректны, поскольку в них нет NULL-значений. Альтернативный метод обработки этой ситуации заключается в использовании индикаторов переменных.

Индикаторы переменных, как и остальные переменные, объявляются в разделе объявления переменных SQL. Они имеют тип, определенный во включающем языке и соответствующий числовому (numeric) типу SQL. Как только выполняется операция, которая может поместить NULL-значения в переменные вызывающего языка, можно использовать индикатор переменной в качестве меры предосторожности. Индикатор размещается в команде SQL непосредственно после переменной включающего языка, которую желательно защитить, без использования разделяющего пробела или запятой, хотя при желании можно вставить необязательное ключевое слово INDICATOR.

Перед выполнением команды индикатору переменной присваивается начальное значение 0. Если в процессе выполнения команды переменная получает NULL-значение, то индикатор переменной приобретает отрицательное значение. После выполнения команды можно проверить значение индикатора переменной и определить, было ли обнаружено NULL-значение. Предположим, поля city и comm таблицы Salespeople не имеют ограничения NOT NULL, и в разделе объявления переменных SQL объявлены две переменные языка Pascal типа integer, i_a и i_b. (В разделе объявления переменных ничто не говорит о том, что эти переменные будут применяться как индикаторы переменных. Они становятся индикаторами, когда их используют в своем собственном смысле.) Перед вами одна из возможностей:

```
EXEC SQL OPEN CURSOR High_cust;
while SQLCODE = 0 do
    begin
    EXEC SQL FETCH High_cust
            INTO :id_num, :salesperson,
                :loc:i_a, :commINDICATOR:i_b;
        if i_a >= 0 and i_b >= 0 then
                {no NULLS produced}
            EXEC SQL UPDATE Salespeople
                SET comm = comm + .01
                WHERE CURRENT OF High_cust;
        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 commissions');
        end;
                    {else}
        end; {while}
    EXEC SQL CLOSE CURSOR High_cust;
```

Из примера ясно, что в одном случае включено ключевое слово INDICATOR, во втором случае оно не использовалось с целью иллюстрации этих возможностей; по действию индикаторов переменных эти два варианта ничем не отличаются. Извлекается каждая строка, но команда UPDATE выполняется в том случае, если не обнаружено NULL-значение. Если получено NULL-значение, то выполняется else — часть программы, которая печатает предупреждающее сообщение, определяющее, где конкретно было обнаружено NULL-значение. Замечание: индикаторы должны проверяться во включающем языке, как в рассмотренном примере, а не в предложении WHERE команды SQL, что в общем-то законно, но может привести к непредсказуемым результатам.

Использование индикатора переменных для эмуляции NULL-значений

Можно трактовать индикаторы переменных, связанные с каждой переменной включающего языка как способ эмуляции NULL-значений SQL. Поскольку одно из этих значений используется в программе, например, в предложении if ... then, можно контролировать соответствующий индикатор переменной, чтобы увидеть, принимает ли она значение NULL. В этом случае возникает возможность различной трактовки переменных. Например, если NULL-значение было найдено в поле city для значения переменной включающего языка city, которая связана с индикатором переменной i_city, можно установить значение переменной city равным последовательности пробелов, что необходимо, когда предполагается печатать это значение; оно не имеет значения для логики конкретной программы. Конечно, i_city автоматически принимает отрицательное значение. Предположим, что в программе есть следующая конструкция if ... then:

```
if city = 'London' then
    comm := comm + .01
else comm := comm - .01;
```

Любое значение, введенное в переменную city, будет либо равно значению 'London', либо не равно ему. Следовательно, комиссионные в любом случае либо увеличиваются, либо уменьшаются. Но в SQL эквивалентная команда работает по-другому:

```
EXEC SQL UPDATE Salespeople
    SET comm = comm + .01
    WHERE city = 'London';
```

и

```
EXEC SQL UPDATE Salespeople
     SET comm = comm - .01;
     WHERE city <> 'London';
```

(Версия Pascal работает только с единственным значением, тогда как версия SQL действует на целой таблице.) Если значение city в версии SQL было равно NULL, то оба предиката принимают значение unknown (неизвестно) и, следовательно, значение comm не изменяется в обоих случаях. Во включающем языке можно добиться того же, используя индикатор переменной и задав условие, исключающее NULL-значения:

```
if i_city >= 0 then
     begin
     if city = 'London' then
          comm := comm + .01
     else comm := comm - .01;
     end;
     { begin и end необходимы в данном случае для большей наглядности }
```

В более сложной программе можно присвоить булевым переменным значение "истина", чтобы определить ситуацию, когда city имеет значение NULL. Затем можно просто проверить это значение переменной.

Другие пользователи индикатора переменных

Индикатор переменной можно использовать для назначения NULL-значений. Добавить их к именам переменных включающего языка в команде UPDATE или INSERT, как в команде SELECT. Если индикатор переменной имеет отрицательное значение, то в поле следует поместить NULL-значение. Например, следующая команда поместит NULL-значение в поле city и в поле comm таблицы Salesperson, если индикаторы переменных i_a или i_b имеют отрицательное значение; в противном случае они получат значения переменных включающего языка:

```
EXEC SQL INSERT INTO Salespeople
     VALUES (:id_num, :salesperson, :loc:i_a, :comm:i_b);
```

Индикаторы переменных используются и для определения усечения строк, когда символьное значение SQL вносится в переменную включающего языка, длина которой недостаточна для сохранения всех символов. Это специальная проблема для нестандартных типов данных VARCHAR и LONG (см. приложение C). В этом случае в переменную заносятся начальные символы строки, а те символы, которые не умещаются в переменной, просто усекаются и теряются. При использовании индикатора переменной он принимает положительное числовое значение, определяющее длину строки до ее усечения, предоставляя возможность оценить, какой длины текст был утерян. Затем,

для выяснения рассматриваемой здесь ситуации нужно выполнить проверку индикатора переменной на > 0, а не на < 0.

Итоги

SQL-команды включаются в процедурные языки программирования для комбинирования сил двух подходов. Реализации такой возможности требует некоторых расширений SQL. Команды встроенного SQL транслируются с помощью программы, названной предкомпилятором (препроцессором), для создания программы, понятной компилятору языка высокого уровня. Команды встроенного SQL заменяются вызовом подпрограмм, которые создаются с помощью встроенного препроцессора; эти подпрограммы называются модулями доступа. С помощью такого подхода ANSI поддерживает встроенный SQL для языков программирования Pascal, FORTRAN, COBOL, PL/I. Другие языки также используются разработчиками. Наиболее важным из них является C.

При описании встроенного SQL следует обратить особое внимание на следующее:

- Все встроенные команды SQL начинаются словами EXEC SQL и заканчиваются в зависимости от используемого языка высокого уровня.

- Все переменные языка высокого уровня, используемые в командах SQL, должны быть внесены в раздел описаний SQL до своего применения.

- Если в командах SQL используются переменные языка высокого уровня, перед их именами необходимо указывать двоеточие.

- Выходные данные для запросов могут храниться непосредственно в переменных языка высокого уровня с помощью INTO тогда и только тогда, когда запрос выбирает единственную строку.

- Курсоры могут применяться для хранения выходных данных запроса и для доступа к ним по одной строке за один цикл обработки. Курсоры объявляются (вместе с определением запроса, выходные данные которого содержит курсор), открываются (что соответствует выполнению запроса) и закрываются (что соответствует удалению выходных данных из курсора, разрыву связи между выходными данными и курсором). Пока курсор открыт, можно использовать команду FETCH для доступа к выходным данным запроса: по одной строке для каждого выполнения команды FETCH.

- Курсоры могут быть обновляемыми или "только для чтения". Чтобы быть обновляемым, курсор должен удовлетворять всем тем критериям, что и представление. Он не должен использовать предложения ORDER BY и UNION, которые запрещено применять в представлении. Необновляемый курсор является курсором "только для чтения".

- Если курсор является обновляемым, его можно применить для управления строками, которые используются командами встроенного SQL UPDATE и DELETE из предложения WHERE CURRENT OF. DELETE или UPDATE должны принадлежать той таблице, доступ к которой осуществляется через курсор запроса.

- SQLCODE может быть объявлен как переменная числового типа для каждой программы, использующей встроенный SQL. Значения этой переменной устанавливаются автоматически после выполнения каждой SQL-команды.

- Если команда SQL выполняется нормально, но не формирует выходных данных либо не выполняет ожидаемых изменений в базе данных, SQLCODE принимает значение 100. Если команда выдает ошибку, то SQLCODE принимает некоторое отрицательное значение, описывающее причину ошибки, в зависимости от конкретной SQL-системы. В противном случае SQLCODE равен нулю.

- Предложение WHENEVER можно использовать для определения действия, которое следует выполнить, если SQLCODE принимает значение 100 (NOT FOUND — не найдено) или отрицательное значение (SQLERROR — ошибка при выполнении SQL). Это действие заключается в переходе к некоторой определенной точке программы (GOTO <метка>) или к выполнению "пустого действия" (CONTINUE, эквивалентно понятию "ничего не делать"). По умолчанию принято "пустое действие".

- В качестве индикаторов можно использовать только числовые переменные. Переменные-индикаторы следуют за другими именами переменных в команде SQL без каких-либо разделяющих символов, за исключением слова INDICATOR.

- Обычно значение переменной-индикатора равно 0. Если команда SQL пытается разместить значение NULL в переменную языка высокого уровня, использующую этот индикатор, то он принимает отрицательное значение. Это свойство можно использовать для защиты от ошибок и в качестве флага, помечающего в SQL NULL-значения, которые будут специально интерпретироваться в основной программе.

- Переменные-индикаторы можно использовать для вставки NULL-значений в команды SQL INSERT или UPDATE. Они принимают положительные значения при возникновении ситуации усечения строк.

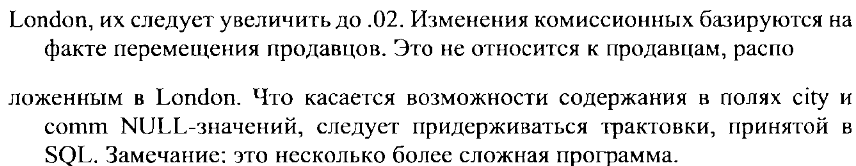
Работаем на SQL

Замечание: Ответы на эти упражнения записаны на псевдокоде, сходном с английским языком и показывающем логику программы, для того, чтобы помочь читателям, не знакомым с языком программирования Pascal (либо с каким-либо другим языком, из числа тех, что можно использовать для иллюстрации). Обратите внимание на используемые концепции, а не на особенности применения того или иного языка. Для соблюдения последовательности в изложении примеров принят стиль псевдокода, сходный с Pascal. Из программ опущены все детали, выходящие за пределы представленного в данной главе материала (определение устройств ввода/вывода, связи с базой данных и так далее). Здесь мы представляем один из многих вариантов выполнения этих упражнений.

1. Разработайте простую программу, которая выбирает все сочетания snum и cnum из таблиц Ordres и Customers и позволяет увидеть все эти комбинации в форме, близкой к письму. Если для значения из таблицы Orders не найдено соответствующего значения из таблицы Customers, то значение поля snum для этой строки заменяется на соответствующее. Можно предположить, что курсор с подзапросом является обновляемым (ограничение ANSI, которое относится и к представлениям, но редко используется на практике) и поддерживается базовая целостность базы данных, отличная от тех ошибок, которые фиксируются (первичный ключ является уникальным, все значения столбца cnums являются корректными, и т.д.). Следует предусмотреть секцию объявлений (DECLARE) и удостовериться, что все используемые курсоры объявлены.

2. Предположим, данная программа соответствует ограничению ANSI, касающемуся запрета на использование для курсов или представлений характеристики "обновляемый". Как нужно модифицировать программу в этом случае?

3. Разработайте программу, которая дает возможность пользователю изменить значения поля city для продавцов, автоматически увеличивая на .01 комиссионные для продавцов, перемещенных для обслуживания в Barcelona, и уменьшая их на .01 для продавцов, перемещенных для обслуживания в San Jose. Кроме того, продавцы, расположенные в данный момент в London, теряют .02 своих комиссионных независимо от смены города, тогда как для продавцов, не расположенных в настоящее время в



London, их следует увеличить до .02. Изменения комиссионных базируются на факте перемещения продавцов. Это не относится к продавцам, распо

ложенным в London. Что касается возможности содержания в полях city и comm NULL-значений, следует придерживаться трактовки, принятой в SQL. Замечание: это несколько более сложная программа.

(Ответы даны в приложении А.)

A
Ответы к упражнениям

ГЛАВА 1

1. cnum
2. rating
3. Запись — альтернативное название строки.
   Поле — альтернативное название столбца.
4. Потому что, по определению, порядок строк не имеет значения.

ГЛАВА 2

1. Символ (или текст) и число.
2. Нет.
3. Язык манипулирования данными (DML, Data Manipulation Language).
4. Слово, которое распознается SQL, как специальная инструкция.

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 ='San Jose';
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');
```

Можно предложить и 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);
```

и

```
   SELECT *
      FROM Orders
```

```
   WHERE odate BETWEEN 10/03/1990 AND 10/04/1990;
2. SELECT *
   FROM Customers
   WHERE snum IN (1001,1004);
3. SELECT *
   FROM Customers
   WHERE cname BETWEEN 'A' AND 'H';
```

Замечание: В системах, использующих ASCII-коды, указанные границы не включают фамилию Hoffman, поскольку после символа 'H' предполагается пробел. По той же причине в качестве второй границы нельзя указать 'G', поскольку в этом случае не будут включены имена Giovanni и Grass. Символ 'G' можно использовать в случае, если за ним следует символ 'Z' — последний символ алфавита.

```
4. SELECT *
   FROM Customers
   WHERE cname LIKE 'C%';
5. 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;
```

Алиасы могут иметь имена, отличные от указанных.

```
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;
```

Здесь используется несколько переменных, имена которых могут быть произвольными, но ответ должен содержать все указанные логические компоненты.

```
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 связанных подзапросов

```
SELECT snum, sname
 FROM Salespeople main
 WHERE city IN
     (SELECT city
          FROM Customers inner
          WHERE inner.snum < > main.snum);
```

Решение с помощью соединения:

```
SELECT DISTINCT first.snum, sname
 FROM Salespeople first, Customers second
 WHERE first.city = second.city
     AND first.snum < > second.snum;
```

Связанный подзапрос находит всех покупателей, которые не обслуживаются данным продавцом и дает возможность выявить ситуацию, когда кто-то из них находится в том же городе. Решение с помощью соединения проще и более соответствует интуитивным соображениям. Он выявляет случаи, когда значения в поле city совпадают, а значения в поле snums не совпадают. Следовательно, с помощью операции соединения получается более элегантное решение проблемы. Надо принять это во внимание, начиная с этого момента. Более элегантный подзапрос для решения этой проблемы будет рассмотрен 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 Salespeole 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.snum < > b.snum);
```

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	San Jose	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 Rating'
     FROM Customers
     WHERE rating <2 00;
```

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 двумя предложениями заключается в способе записи второго предиката. В обоих случаях строка 'Low Rating' имеет предшествующий пробел, таким образом длина этой строки совпадает с длиной строки 'High Rating'.

```
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);
```

Можно представить более наглядную версию этой команды, которая корректна только для случая, когда комиссионные (comm) не превышают 1.0 (100 процентов).

```
UPDATE Salespeople
```

```
SET comm = comm + (comm* .2)
WHERE 3000 <
    (SELECT SUM (amt)
        FROM Orders
        WHERE snum = Salespeople.snum)
AND comm + (comm*.2) < 1.0;
```

Существуют другие успешные способы решения этих проблем.

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);
```

(В примерах используются произвольные имена индексов.)

```
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,
      UNIQUE (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));
```

Первый вариант ответа является более предпочтительным.

```
2. CREATE TABLE Salespeople
   (snum    integer NOT NULL PRIMARY KEY,
```

```
     sname    char(15) CHECK (sname BETWEEN 'AA' AND 'MZ'),
     city     char(15),
     comm     decimal NOY NULL DEFAULT = .10);
3. CREATE TABLE Orders
     (onum    integer NOT NULL,
     amt      decimal,
     odate    date,
     cnum     integer NOT NULL,
     snum     integer NOT NULL,
     CHECK ((cname > sname) 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) ); 9
```

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 является необновляемым из-за использования DISTINCT.
   #2 является необновляемым, поскольку он использует соединение, функцию агрегирования и GROUP BY.
   #3 является необновляемым, поскольку он базируется на #1, который, в свою очередь, является необновляемым.
   #4 является обновляемым.
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
```

```
        WHITH 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 откатываться назад.

4. Исключительное использование.

5. Только для чтения.

ГЛАВА 24

```
1. SELECT a.tname, a.owner, b.cname, b.datatype
     FROM SYSTEMCATALOG a, SYSTEMCOLUMNS b
     WHERE a.tname = b.tname
        AND a.owner = b.owner
        AND a.numcolumns > 4;
```

Замечание: Поскольку большинство имен столбцов соединяемых таблиц различно, не все случаи использования алиасов a и b в представленной команде являются необходимыми. Имена алиасов представлены для большей ясности.

```
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;
{ требуется 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);
{ Здесь используется SQL. Рассмотренный запрос локализует строки
таблицы Ordes, которые не согласуются со строками таблицы Customers.}
EXEC SQL DECLARE Cust_assigns AS CURSOR FOR
   SELECT cnum,snum
       FROM Customers;
{ Этот курсор используется для поддержки правильных
значений столбца snum.}
begin { основная программа }
EXEC SQL OPEN CURSOR Wrong_Orders;
while SQLCODE = 0 do
{ Цикл пока значение переменной Wrong_Orders пусто.}
   begin
   EXEC SQL FETCH Wrong_Orders INTO
    (:cnum, :snum);
   if SQLCODE=0 then
        begin
{ Если значение переменной Wrong_Orders пусто, то в этом цикле ничего
не должно выполняется.}
EXEC SQL OPEN CURSOR Cust_Assigns;
  repeat
      EXEC SQL FETCH Cust_Assigns
           INTO (:custnum, :salesnum);
  until :custnum = :cnum;
{ Повторение команды FETCH until... приведет к тому, что курсор Cust_Assigns
перемещается шаг за шагом до тех пор, пока строка, содержащая текущее
значение cnum из Wrong_Orders не будет найдена.}
   EXEC SQL CLOSE CURSOR Cust_assigns;
{Таким образом, каждое повторное выполнение происходит с начала цикла.
Значение, полученное с помощью этого курсора, хранится 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. В программе, которую использовал автор, решение заключалось в простом включении onum(первичный ключ таблицы Orders в курсоре Wrong_orders). В команде UPDATE следует использовать предикат WHERE onum = :ordernum (в предположении, что объявлена переменная целого типа ordernum) вместо WHERE CURRENT OF Wrong_Orders. Результирующая программа будет выглядеть примерно так (большинство комментариев из предыдущей версии опущено):

```
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, выполняемый до тех пор, пока Wrong_Orders
пуст }
    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;
    begin { основная программа }
    EXEC SQL OPEN CURSOR Salesperson;
    EXEC SQL FETCH Salesperson
      INTO (:snum, :sname, :city:i_cit:comm:i_com);
    { Выбор (fetch) первой строки. }
    while SQLCODE = 0 do
    { Цикл выполняется пока (while) есть строки в таблице Salesperson.}
        begin
        if i_com < 0 then commnull : = true;
        if i_cit < 0 then citynull : = true;
    { Установить булевы флаги, которые соответствуют NULL-значениям }
      if citynull then
        begin
        write ('Для Salesperson не обнаружено текущее значение столбца
               city', snum,'Необходимо предпринять какие-либо действия?
               (Y(да)/N(нет))');
{ Данное приглашение соответствует случаю, когда значение city пусто (NULL)}
        read (response);
{ Значение переменной response используем позже. }
        end { if citynull }
        else { not citynull }
               begin
               if not commnull then
{Выполняем сравнение и действия для случая, когда значение
переменной commnull отлично от NULL }
                          begin
```

```
                    if city='London' then comm : = comm - .02
                            else comm : = comm + .02;
                    end;
{Для ситуации "если не commnull", begin и end указаны для ясности. }
                    write ('Текущий город для продавца ', snum, ' - ', city,'
                           Желаете изменить значение? (Y/N)');
```

Замечание: Для продавца, которому в данный момент не назначен город, комиссионные назначаются в зависимости от того, проживает он в Лондоне или нет.

```
                    read (response);
{ Переменная response содержит значение независимо от того, имеет ли
переменная citynull значение "истина" или "ложь". }
                    end; { else not citynull }
              if response = 'Y' then
                    begin
                    write ('Введите новое значение для city: ');
                    read (newcity);
                    if not commnull then
{ Эта операция может быть выполнена только для значений, отличных от NULL }
                           case newcity of:
                                begin
                                'Barcelona': comm: = comm+.01,
                                'San Jose':comm: = comm-.01
                                 end; { case and if not commnull}
                    EXEC SQL UPDATE Salespeople
                        SET city=:newcity, comm = : comm:i_com
                        WHERE CURRENT OF Salesperson;
{ Переменная-индикатор comm принимает NULL-значение в нужной ситуации}
                        end; { если response = 'Y', если response<>'Y',
                               то никаких изменений выполнять не требуется.}
                        EXEC SQL FETCH Salesperson
                            INTO (:snum,: sname,:c ity:i_cit,
                                 :comm:i_com);
{ Перейти к следующей строке }
             end; { пока SQLCODE = 0 }
         EXEC SQL CLOSE CURSOR Salesperson;
  end; { основная программа }
```

B
Типы данных SQL

Типы данных, признаваемые ANSI, представлены символьным типом (CHAR) и несколькими типами числовых значений, которые можно разбить на две категории: точные числовые значения и приближенные числовые значения. Точные числовые значения содержат цифры с десятичной точкой или без нее в традиционном представлении числовых значений. Приближенные числовые значения представлены в экспоненциальной форме (по основанию 10). Другие отличия между этими двумя числовыми типами менее существенны.

Иногда типы данных используют аргумент, названный в книге аргументом размера, точный формат и значение которого зависят от конкретного типа данных. Если аргумент размера опущен, принимаются значения по умолчанию для всех типов.

Типы ANSI

Далее представлены типы данных ANSI (название в скобках является синонимом).

ТЕКСТ (TEXT)

Тип данных	Описание
CHAR (CHARACTER)	Строка текста в формате, определенном разработчиком. Для этого типа аргумент размера — целое неотрицательное число, задающее максимальную длину строки. Значения этого типа можно заключить в одиночные кавычки, например, 'текст'. Две следующие друг за другом одиночные кавычки (''), расположенные внутри строки, задают один символ одиночной кавычки.

ТОЧНО ЧИСЛОВЫЕ (EXACT NUMERIC)

Тип данных	Описание
DEC (DECIMAL)	Десятичное число, т.е. число, которое может иметь в своем представлении десятичную точку. Соответственно, аргумент размера имеет две части: точность и масштаб. Масштаб не может превышать точность. Точность указывается на первом месте, за ней следует запятая, отделяющая аргумент масштаба. Точность показывает количество значащих десятичных разрядов. Максимальное количество разрядов для числа зависит от конкретного способа реализации, равно этому числу или пре-

	вышает его. Масштаб определяет максимальное количество разрядов справа от десятичной точки. В том случае, когда масштаб равен нулю, поле эквивалентно типу "целый" (INTEGER).
NUMERIC	Совпадает с DECIMAL, за исключением того, что максимальное количество разрядов не может превышать аргумента точности.
INT (INTEGER)	Число без явно представленной десятичной точки. Этот тип эквивалентен типу DECIMAL, не имеющему разрядов справа от десятичной точки, т.е. с масштабом, равным 0. Аргумент размера не используется (он назначается автоматически в зависимости от конкретного способа реализации).
SMALLINT	Совпадает с INTEGER, за исключением того, что в зависимости от конкретного способа реализации, размер, принятый по умолчанию для этого типа, может быть (или не быть) меньше, чем для типа INTEGER.

ПРИБЛИЖЕННЫЕ ЧИСЛОВЫЕ (APPROXIMATE NUMERIC)

Тип данных	**Описание**
FLOAT	Число с плавающей точкой, представленное в экспоненциальной форме по основанию 10. Аргумент размера содержит единственное число, задающее минимальную точность.
REAL	Совпадает с FLOAT, за исключением того, что аргумент размера не используется. Точность устанавливается по умолчанию в зависимости от конкретного способа реализации.
DOUBLE PRECISION (или DOUBLE)	Совпадает с REAL, за исключением того, что точность конкретной реализации для DOUBLE PRECISION, может превышать точность конкретной реализации для REAL.

Эквивалентные типы данных в других языках

При включении SQL в другие языки значения, используемые в командах SQL и получаемые в результате выполнения команд SQL, обычно хранятся как переменные включающего языка (см. главу 25). Эти переменные должны быть совместимы по типам данных со значениями SQL, которые они содержат. В приложениях, не являющихся частью официального стандарта SQL, ANSI предоставляет для применения в качестве включающих языков программирования следующие языки: PASCAL, PL/I, COBOL, FORTRAN. Одна из возникающих здесь проблем — необходимость определить эквивалентность типов данных для переменных, используемых в этих языках.

Далее представлены эквиваленты для четырех, определенных ANSI, языков.

PL/I

Тип в SQL	Эквивалент в PL/I
CHAR	CHAR
DECIMAL	FIXED DECIMAL
INTEGER	FIXED BINARY
FLOAT	FLOAT BINARY

COBOL

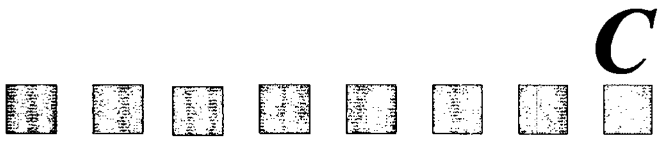
Тип в SQL	Эквивалент в COBOL
CHAR(<целое>)	PIC X(<целое>)
INTEGER	PIC S(<девятки>) USAGE COMPUTATIONAL
NUMERIC	PIC S(<*девятки с включением символа V*>) DISPLAY SIGN LEADING SEPARATE

PASCAL

Тип в SQL	Эквивалент в PASCAL
INTEGER	INTEGER
REAL	REAL
CHAR(<длина>)	PACKED ARRAY [1..<длина>] OF CHAR

FORTRAN

Тип в SQL	**Эквивалент в FORTRAN**
CHARACTER	CHARACTER
INTEGER	INTEGER
REAL	REAL
DOUBLE PRECISION	DOUBLE PRECISION



C

Некоторые общие отклонения от стандарта SQL

Существует целый ряд характерных черт языка SQL, которые не определены как составная часть ни в стандарте ANSI, ни в стандарте ISO, но являются общими для множества реализаций SQL, в силу того, что в практике они были признаны полезными. В этом приложении рассматриваются подобные характеристики. Естественно, что они отличаются для различных программных продуктов. Здесь мы знакомим вас с введением в некоторые наиболее общие подходы.

Типы данных

Типы данных, поддерживаемые стандартом SQL, рассмотрены в приложении В. Они представлены типом CHARACTER и множеством числовых типов. Фактически, разрабатываемые приложения могут быть гораздо более сложными в плане реально используемых типов данных. Обсудим некоторые нестандартные типы данных.

ТИПЫ DATE (дата) и TIME (время)

Тип данных DATE широко используется несмотря на то, что он не является частью стандарта. Мы применяли этот тип в таблице Orders, предполагая формат: mm/dd/yyyy. Такой формат представления даты принят IBM в качестве стандартного для США. Конечно, возможны и другие, и конкретные реализации часто поддерживают множество форматов, предоставляя возможность выбора в соответствии с потребностями. Реализация, поддерживающая множество форматов представления дат, преобразует их из одного формата в другой автоматически. Существуют и другие не менее важные форматы представления дат.

Стандарт	Формат	Пример
Формат ISO (International Standards Organization)	yyyy-mm-dd	1990-10-31
Формат JIS (Japanese Industrial Stadard)	yyyy-mm-dd	1990-10-31
Формат EUR (IBM European Stadard)	dd.mm.yyyy	10.31.1990

Благодаря наличию специального типа, определенного для дат, над ними можно выполнять арифметические операции. Например, можно прибавить к дате определенное количество дней и в результате получить другую дату, причем, программа сама отслеживает количество дней в месяцах, переход через границы годов и т.д. Даты можно также сравнивать, например, date A < date B означает, что дата А предшествует дате В в хронологическом порядке.

Есть множество программ, в которых, помимо дат, принят специальный тип для времени (TIME); для представления времени также определено множество форматов, включая следующие:

Стандарт	Формат	Пример
Формат ISO (International Standards Organization)	hh-mm-ss	21.04.37
Формат JIS (Japanese Industrial Standard)	hh-mm-ss	21.04.37
Формат EUR (IBM European Standard)	hh-mm-ss	21.04.37
Формат USA (IBM USA Standard)	hh.mm AM/PM	9.04 PM

Для данных, представленных в формате TIME, определены операции сложения и сравнения (как и для данных, представленных в формате DATE), причем количество секунд в минуте и часов в сутках учитывается автоматически. Кроме того, определены специальные встроенные константы, определяющие текущую дату и текущее время (CURDATE и CURTIME соответственно). Эти константы сходны с константой USER в том смысле, что их значения постоянно обновляются.

Можно ли включить время и дату в одно поле? Некоторые реализации включают только тип DATE, предполагая, что этого вполне достаточно. Напротив, есть реализации, в которых определен тип TIMESTAMP, который определяется как комбинация двух типов — DATE и TIME.

ТИПЫ TEXT STRING (строка текста)

ANSI поддерживает один тип для представления текста. Это тип CHAR. Предполагается, что любое поле этого типа имеет индивидуальную длину. Если строка, вставляемая в поле, имеет меньшую длину, то она дополняется пробелами; длина строки не может превосходить длины поля. Хотя, с точки зрения удобства разработки приложений, это определение накладывает ряд ограничений для пользователя. Например, операция UNION применима только к символьным полям одинаковой длины.

Многие разработки поддерживают строки переменной длины и для этого — типы данных VARCHAR и LONG VARCHAR (часто называемый просто LONG). Поле типа CHAR всегда занимает при размещении в основной памяти количество символов, определяемое максимальным количеством символов, хранящихся в поле, а поле VARCHAR занимает только участок памяти, необходимый для хранения реального значения поля, хотя SQL должен будет выделить дополнительно смежный участок памяти для хранения реальной длины поля. Поля типа VARCHAR могут иметь любую длину вплоть до определенного разработчиком максимума, который изменяется от 254 до 2048 символов для типа VARCHAR и вплоть до 16K символов для типа LONG. Тип LONG обычно используется для текстов более или менее одинаковой природы (например, для поясняющих текстов) или для данных, которые трудно сжать до про-

стого поля. Тип VARCHAR можно использовать для любых текстов, длина которых изменяется в достаточно широких пределах.

Не рекомендуется использовать тип VARCHAR вместо CHAR. Реализация операций поиска и обновления для полей типа VARCHAR сложнее и, следовательно, медленнее, чем для полей типа CHAR. Кроме того, тип VARCHAR использует определенный объем основной памяти для хранения длины строки. Следует оценить, какая часть множества значений поля будет отличаться по длине, а также будут ли значения поля объединяться с другими полями, прежде чем решить какому из типов отдать предпочтение: CHAR или VARCHAR. Часто тип LONG используют для хранения двоичных данных. Конечно, поля типа LONG имеют ограничения по длине, которые не всегда кажутся удачными, но тем не менее в SQL можно работать с данными такого типа. Детали следует уточнять по соответствующим руководствам.

Команда FORMAT

Набор средств для обработки результата в стандартном SQL весьма ограничен. Хотя многие реализации включают SQL в пакеты программ, предоставляющие различные средства для выполнения этой функции, некоторые реализации используют команду, подобную команде FORMAT, в самом SQL для применения определенных структур при оформлении ответа на запрос. Среди возможных функций команды FORMAT следует отметить следующие:

- установка ширины столбцов (для печати);
- определение способа представления NULL-значений;
- назначение новых имен столбцов;
- определение элементов оформления начала и конца страницы;
- применение подходящих или изменение заданных форматов, содержащих дату, время или денежную единицу;
- подведение итогов и подытогов без исключения полей, по которым оно производится, как и с помощью SUM. Один из возможных подходов к решению этой задачи в некоторых программных продуктах — использование предложения COMPUTE.

Команду FORMAT можно включить непосредственно до или после запроса, для которого эта команда применяется, в зависимости от реализации. Обычно одна команда FORMAT применяется к одному запросу, хотя и ряд команд FORMAT может применяться к одному запросу. Приведем несколько типичных команд FORMAT:

```
FORMAT NULL '_ _ _ _ _ _ _';
FORMAT BTITLE 'Заказы, сгруппированные по продавцам';
FORMAT EXCLUDE (2, 3);
```

Первое из этих предложений определяет способ печати NULL-значений в виде '_ _ _ _ _ _ _'. В соответствии со вторым предложением заголовок, указанный в кавычках, появится в конце каждой страницы. Третье предложение исключает второй и третий столбцы таблицы из выходных данных предыдущего запроса. Последний вариант команды FORMAT можно применить для выбора конкретных столбцов таблицы с целью их использования в предложении ORDER BY для упорядочения выходных данных. Поскольку особые функции команды FORMAT выполняются по-разному, полное описание ее возможных применений выходит за границы данной книги.

Существуют другие команды, которые с успехом могут пополнить список таких функций. Команда SET похожа на команду FORMAT. Она может служить альтернативой команде FORMAT либо быть командой, область действия которой распространяется на все запросы текущего сеанса пользователя, а не только на один запрос.

Некоторые реализации начинают команды с ключевого слова COLUMN, а не с ключевого слова FORMAT. Например:

```
COLUMN odate FORMAT dd-mon-yy;
```

Это предложение задает формат представления поля даты 10-Oct-90 для вывода результатов запросов.

Предложение COMPUTE, упомянутое ранее, включается в запрос таким образом:

```
SELECT odate, amt
   FROM Orders
   WHERE snum = 1001
   COMPUTE SUM (amt);
```

В соответствии с этим запросом будут получены все заказы для Peel с указанием даты (odate) и количества (amt), а затем — итог по полю amt. Некоторые реализации осуществляют подведение итогов, используя COMPUTE как команду. В такой реализации прежде всего определяется прерывание:

```
BREAK ON odate;
```

По этой команде происходит разделение результата сформулированного выше запроса на группы таким образом, что в пределах одной группы значение поля odate остается постоянным. Теперь можно ввести следующее предложение:

```
COMPUTE SUM OF amt ON odate;
```

Столбец, указанный после ключевого слова ON, должен быть предварительно использован в команде BREAK.

Функции

В ANSI-стандарте SQL можно применять функции агрегирования к столбцам и использовать их значения в скалярных выражениях, например: comm * 100. Существует и множество других полезных функций, применимых на практике. Функции SQL, отличающиеся от стандартных функций агрегирования, можно применять в предложении

SELECT запроса, точно так же, как и стандартные, но при этом они воздействуют не на единичные значения, а на их группы. В приводимой здесь таблице функции классифицированы по типам данных, на которые они воздействуют. В таблице переменные указывают именно тот тип значений, который разрешено применять в предложении SELECT, если не оговорено нечто другое.

Математические функции

Данные функции применяются к числовым значениям.

Функция	**Назначение**
ABS(X)	Абсолютное значение X (преобразует отрицательное или положительное в положительное)
CEIL(X)	X имеет десятичное значение, которое следует округлить сверху.
FLOOR(X)	X имеет десятичное значение, которое следует округлить 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 X, возведенный в степень Y.
ROUND(X,Y)	Округляет X до Y десятичных разрядов. Если Y не указан, то округляет до целого.
SIGN(X)	Возвращает минус, если X < 0, и плюс — в противном случае.
SQRT(X)	Возвращает квадратный корень из X.

Символьные функции

Эти функции могут применяться к строкам текста или столбцам, для которых определен тип данных TEXT, к заданным явно строкам текста или их комбинациям.

Функция	**Назначение**
LEFT(<*строка*>,X)	Возвращает X самых левых символов из строки.
RIGHT(<*строка*>,X)	Возвращает X самых правых символов из строки.
ASCII(<*строка*>)	Возвращает ASCII-код, который представляет строку в памяти компьютера.

CHR(<*ASCII-код*>)	Возвращает символы, соответствующие ASCII-коду.
VALUE (<*строка*>)	Возвращает математическое значение строки. Предполагается, что строка имеет тип CHAR или VARCHAR, но состоит из чисел. VALUE('3') выдает число 3 типа INTEGER.
UPPER (<*строка*>)	Все символы в строке переводятся в "большое" (заглавное, строчное) написание.
LOWER (<*строка*>)	Все символы в строке переводятся в "малое" (прописное) написание.
INITCAP (<*строка*>)	Все символы в строке переводятся в написание "в одном регистре". В некоторых реализациях функция называется PROPER.
LENGTH (<*строка*>)	Возвращает количество символов в строке.
<*строка*>||<*строка*>	Комбинирует две строки в выходных данных, таким образом, что первая из них непосредственно предшествует второй. (|| называется оператором конкатенации.)
LPAD (<*строка*>,X,'*')	Заполняет строку слева символом "*" или любым другим символом, указанным в кавычках, для того, длина строки стала равной X.
RPAD (<*строка*>,X,'*')	Совпадает с LPAD, за исключением того, что заполнение строки осуществляется справа.
SUBSTR (<*строка*>,X,Y)	Извлекает Y символов из строки, начиная с позиции X.

Функции даты и времени

Эти функции воздействуют на значения "дата" и "время".

Функция	**Назначение**
DAY(<*дата*>)	Извлекает день месяца из даты. Аналогичные функции существуют для MONTH (месяца), YEAR (года), HOUR (часа), SECOND (секунды) и т.д.
WEEKDAY (<*дата*>)	Определяет название дня недели по дате.

Прочие

Эти функции применимы к любому типу данных.

Функция	**Назначение**
NVL(<*столбец*>,) <*значение*>	NVL (NULL Value) подставляет <значение>, определяемое вторым аргументом, вместо каждого NULL-значения, обнаруженного в указанном (первым аргументом) столбце. Если в указанном столбце NULL-значений не обнаружено, то никаких изменений не происходит.

Операции INTERSECT (пересечение) и MINUS (разность)

Команда UNION связывает два запроса, объединяя результаты выполнения каждого из них в один. Два других способа комбинирования результатов отдельных запросов базируются на использовании операций INTERSECT (пересечение) и MINUS (разность). Результат выполнения операции INTERSEPT — строки, которые содержатся в результате выполнения каждого из участвующих в операции запросов. Результат выполнения операции MINUS — только те строки, которые получаются в результате выполнения одного из участвующих в операции запросов, но не содержатся в другом.

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);
```

выводят строки, выбираемые каждым их этих запросов и содержащие сведения о тех продавцах в Лондоне, которые имеют, по крайней мере, одного покупателя в своем городе.

С другой стороны:

```
SELECT *
    FROM Salespeople
    WHERE city='London'
    MINUS
SELECT *
    FROM Salespeople
    WHERE 'London' IN
      (SELECT city
            FROM Customers
            WHERE Customers.snum = Salespeople.snum);
```

в результате выполнения такой команды строки, выбранные первым запросом, удаляются из выходных данных второго, т.е. мы получаем сведения обо всех продавцах из Лондона, не имеющих покупателей в этом городе. Иногда операцию MINUS называют DEFFERENCE.

Автоматические OUTER JOINS (внешние соединения)

В главе 14 было показано, как выполнить операцию внешнего соединения, используя команду UNION. Некоторые программы, осуществляющие обработку баз данных, предусматривают непосредственные способы выполнения внешнего соединения. В отдельных реализациях указывается бинарная операция "плюс" (+) после предиката, которая говорит о том, что следует рассматривать строки, удовлетворяющие условию, и строки, условию не удовлетворяющие. Условие предиката будет содержать поле, используемое в обеих таблицах, а NULL-значения включаются, если соответствие не

было обнаружено. Предположим, нужно узнать имена продавцов, включая тех, которые в данный момент не обслуживают никаких покупателей (в рассмотренных здесь таблицах их нет, но в реальной жизни такая ситуация возможна).

```
SELECT a.snum, sname, cname
   FROM Salespeople a, Customers b
   WHERE a.snum=b.snum (+);
```

что эквивалентно следующей операции 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);
```

Предположим, символы подчеркивания используются для представления NULL-значений при выводе (см. описание команды FORMAT в этом приложении).

Ведение журнала

Реализация SQL, если она поддерживает доступ многих пользователей, скорее всего обеспечивает и какую-то возможность фиксации обращений к базе данных. Существуют два основных вида таких средств: ведение журнала и ведение отчета (аудит). Цели их применения различны.

Ведение журнала выполняется для защиты данных от сбоев в системе. Во-первых, используется зависящая от реализации процедура, которая позволяет восстановить текущее состояние базы данных: создается внешняя копия содержимого базы данных, которая может хранится где-либо (независимо от компьютера). Затем в журнале фиксируются изменения базы данных. В области памяти, отличной от основной области, где размещается база данных (причем, предпочтительным является использование другого устройства), фиксируется каждая команда, вносящая изменения в структуру базы данных или в ее содержимое. Если возникли проблемы и потеряно текущее содержимое базы данных, можно повторно выполнить все изменения, зафиксированные в журнале, для сохраненной копии базы данных и таким образом вернуть базу данных в состояние, которое она имела на момент фиксации последней записи в журнале изменений. Типичная команда, позволяющая установить режим ведения журнала:

```
SET JORNAL ON;
```

Ведение отчета выполняется для обеспечения защиты. В нем фиксируется, кто и какие действия выполнял над базой данных. Эта информация хранится в виде таблицы, доступной для ограниченного круга привилегированных пользователей. Потребность в проверке каждого отдельного действия возникает крайне редко, тогда как для хранения такого подробного отчета требуется очень большой объем памяти. Поэтому на практике можно проверять отдельных пользователей, конкретные действия и объекты данных. Одна из допустимых форм команды AUDIT:

```
AUDIT INSERT ON Salespeople BY Diane;
```

Предложения ON или BY могут быть опущены, в результате задается режим проверки для всех объектов и для всех пользователей соответственно. Если вместо AUDIT INSERT указать AUDIT ALL, будут фиксироваться все действия пользователя Diane, в том числе и с файлом базы данных Salespeople.

D

Справка по синтаксису и командам

Назначение данного приложения — дать быструю и точную справку и сжатые определения различных команд SQL. Первая часть содержит элементы, используемые для создания SQL-команд; вторая — детали синтаксиса и краткие объяснения команд.

Рассматриваются следующие стандартные соглашения (названные BNF-соглашениями):

- Ключевые слова задаются большими буквами.
- SQL и другие специальные термины указаны в угловых скобках и курсивом (< *and* >).
- Любые части команды указаны в квадратных скобках ([and]).
- Круглые скобки (...) показывают, что предшествующая часть команды может быть повторена любое число раз.
- Вертикальный разделитель (|) указывает, что все стоящее перед ним можно заменить всем тем, что за ним следует.
- Фигурные скобки ({ and }) определяют: все, что указано внутри них, должно рассматриваться как единое целое при применении к нему других символов.
- Два двоеточия и знак равенства (: : =) означают, что правая часть этой конструкции является определением ее левой части.

Кроме того, последовательность (.,..) означает: все предшествующее можно повторять произвольное число раз, причем каждое вхождение в ней отделяется запятой. Атрибуты, которые не являются частью официального стандарта, помечены звездочками (* nonstandard *).

Замечание: Терминология, используемая здесь, не является официальной терминологией ANSI. Официальная терминология представляется слишком запутанной, и ее упростили. Поэтому иногда используются термины, отличные от ANSI, либо термины ANSI, но с некоторыми отличиями от стандарта. Например, определение *<предикат>* содержит две части: одна часть — та, которая стандартно называется *<предикат>*, вторая часть — та, которая называется *<условие поиска>*.

Элементы SQL

Этот раздел содержит элементы команд SQL. Они разбиты на две категории: основные элементы языка и функциональные элементы языка. Основные элементы языка — это основные строительные блоки языка; когда SQL выполняет разбор команды, он, прежде всего, оценивает каждый символ в тексте команды в терминах этих элементов. *<Разделитель>* отделяет одну часть команды от другой; все то, что расположено

между двумя соседними *<разделителями>* считается единицей. На базе выделения таких единиц в тексте запроса SQL интерпретирует команду.

Элементы, отличные от ключевых слов, являются функциональными и интерпретируются в SQL. Части команды, расположенные между *<разделителями>*, имеют специальное значение. Некоторые из них специфичны для отдельных команд и обсуждаются вместе с соответствующими командами в этом приложении; другие — общие для множества команд.

Функциональные элементы определяются в терминах друг друга или в терминах их же самих. Например, понятие *<предикат>* определяется рекурсивно через это же понятие, поскольку для создания *<предиката>* используются AND или OR и отдельные *<предикаты>*.

Определение *<предиката>* вы найдете в отдельном разделе данного приложения из-за множественности его форм и сложности этого функционального элемента. Определение предиката следует за рассмотрением прочих функциональных элементов языка.

Основные элементы языка

Элемент	**Определение**
<разделитель>	*<комментарий>*|*<пробел>*|*<новая* строка>
<комментарий>	– – *<строка>* *<новая* строка>
<пробел>	символ пробела
<новая строка>	символ конца строки (зависит от реализации)
<идентификатор>	*<буква>*[{*<буква* или *цифра>*| *<символ* подчеркивания>}...] Замечание: В строгом соответствии со стандартом ANSI, буквы могут быть только большими и *<идентификатор>* не может содержать более 18 символов.
<символ подчеркивания>	_

Элемент	**Определение**
<символ процента>	%
<ограничитель> или	любой из следующих: , ( ) < > . : = + * – / <> >= <= *<строка>*
<строка>	[любая печатаемая строка в одиночных кавычках]

Элемент	**Определение**

	Замечание: В <*строке*> две следующие непосредственно друг за другом одиночные кавычки интерпретируются как одна.
<*терм SQL*> (* только для расширенного*)	ограничитель предложения, зависящий от включающего языка.

Функциональные элементы

В следующей таблице даны функциональные элементы команд SQL и их определения:

Элемент	**Определение**
<*запрос*>	Предложение SELECT
<*подзапрос*>	Заключенное в круглые скобки предложение SELECT, расположенное внутри другого предложения, которое, в действительности, оценивается отдельно для каждой строки-кандидата на включение в результат в соответствии с внешним предложением.
<*выражение, содержащее значения*>	<*первичный*> |<*первичный*><*оператор*><*первичный*> |<*первичный*><*оператор*><*выражение, содержащее значения*>
<*оператор*>	один из следующих: + – / *
<*первичный*>	<*имя столбца*> | <*литерал*> | <*функция агрегирования*> | <*встроенная константа*> | <*нестандартная функция*>
<*литерал*>	<*строка*> | <*математическое выражение*>
<*встроенная константа*>	USER | <*константа, зависящая от реализации*>
<*имя таблицы*>	<*идентификатор*>
<*спецификатор столбца*>	[<*имя таблицы*> | <*алиас*>.]<*имя столбца*>
<*групповой столбец*>	<*спецификатор столбца*> | <*целое*>
<*упорядоченный столбец*>	<*спецификатор столбца*> | <*целое*>

Элемент **Определение**

<ограничения на столбец>
NOT NULL | UNIQUE
| CHECK (*<предикат>*)
| PRIMARY KEY
| REFERENCES *<имя таблицы>* [(*<имя столбца>*)]

<ограничения на таблицу>
UNIQUE (*<список столбцов>*)
| CHECK (*<предикат>*)
| PRIMARY KEY (*<список столбцов>*)
| FOREIGN KEY (*<список столбцов>*)
REFERENCES *<имя таблицы>* [(*<список столбцов>*)]

<значение по умолчанию>
DEFAULT VALUE = *<выражение, содержащее значения>*

<тип данных>
Допустимый тип (см. приложение B, где перечислены типы, поддерживаемые ANSI, либо приложение C, в котором указаны наиболее общие типы).

<размер>
Значение, зависящее от типа данных (см. приложение B).

<имя курсора>
<идентификатор>

<имя индекса>
<идентификатор>

<синоним>
<идентификатор>(*нестандартный*)

<владелец>
<идентификатор для проверки полномочий>

<список столбцов>
<спецификатор столбца>.,..

<список значений>
<выражение, содержащее значения>.,..

<ссылка на таблицу>
{*<имя таблицы>*[*<алиас>*] }.,..

Предикаты

Следующее определение понятия <предикат> содержит список различных типов, которые будут объяснены далее:

<предикат> :: = [NOT]

{ <предикат сравнения>

| <в предикате>

| <NULL-предикат>

| <предикат "между">

| <предикат "как">

| <составной предикат>

| <предикат существования> }

[AND | OR <предикат>]

<Предикат> — это выражение, которое может быть истинным, ложным или неопределенным, исключение составляют *<предикат существования>* и *<NULL-предикат>*, которые могут быть только либо истинными, либо ложными. Результат получается неопределенным, если NULL-значения препятствуют получению определенного ответа. Такой результат получается, например, в том случае, если NULL-значение сравнивается с конкретным значением. Стандартные булевы операторы — AND, OR и NOT — могут использоваться с *<предикатом>*. NOT true (истина) — false, NOT false (ложь) — true, NOT unknown (неизвестно) — unknown.

AND

AND	**true**	**false**	**unknown**
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

OR

OR	**true**	**false**	**unknown**
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

Эти таблицы читаются так, как таблицы умножения: значение соответствующего булева оператора стоит на пересечении строки и столбца. В таблице AND, например, третий столбец (unknown) и первая строка (true) на пересечении дают значение результата (unknown).

Старшинство операций определяется скобками. При отсутствии скобок NOT имеет более высокий приоритет, за ним следуют AND и OR с одинаковым приоритетом. Различные типы *<предикатов>* рассмотрены по отдельности в следующих разделах.

<Предикат сравнения>

Синтаксис:

<Выражение, содержащее значения> <оператор сравнения> <выражение, содержащее значения> | <подзапрос>

<оператор сравнения> : : =

=

| <

| >

```
| <
| >=
| <>
```

Если какое-либо из <*выражений, содержащих значения*> равно NULL, то <*предикат сравнения*> имеет значение unknown. Если результат сравнения истинен, то предикат сравнения также истинен. Если результат сравнения ложен, то ложен и предикат сравнения. <*Оператор сравнения*> имеет стандартное математическое значение для числовых значений; для значений других типов результат определяется реализацией. Два <*выражения, содержащих значения*>, должны быть совместимы по типам данных. Если используется <*подзапрос*>, то он должен включать единственное выражение, содержащее значения в предложении SELECT, значение которого будет замещено вторым <*выражением, содержащим значения*> в <*предикате сравнения*> всякий раз, когда <*подзапрос*> результативно выполняется.

<*Предикат "между"*>

Синтаксис:

```
<Выражение, содержащее значения> [NOT] BETWEEN <выражение,
содержащее значения> AND <выражение, содержащее значения>
```

Предикат "между" A BETWEEN B AND C имеет то же значение, что и предикат (A>=B AND A<=C). Предикат "между" A NOT BETWEEN B AND C имеет то же значение, что и предикат NOT (A BETWEEN B AND C). <*Выражение, содержащее значение*> можно получить, используя <*подзапрос*> (*нестандартный*).

<*Предикат "в"*>

Синтаксис:

```
<выражение, содержащее значения> [NOT] IN <список значений> |
<подзапрос>
```

<*Список значений*> состоит из множества значений, заключенных в круглые скобки и разделенных запятыми. Если используется <*подзапрос*>, он должен содержать только одно <*выражение, содержащее значения*> в предложении SELECT (может быть и больше, но это не соответствует стандарту). <*Подзапрос*> выполняется отдельно для каждой строки-кандидата из основного запроса. Значения, полученные в результате, образуют <*список значений*> для строки. В любом случае <*предикат "в"*> истинен, если <*выражение, содержащее значения*> представлено в <*списке значений*> , если только не указан NOT. A NOT IN (B,C) эквивалентно NOT (A IN (B,C)).

<*Предикат "как"*>

Синтаксис:

```
<Строковое значение> [NOT] LIKE <образец> [ESCAPE <символ начала
управляющей последовательности>]
```

<Строковое значение> — это любое *<выражение, содержащее значения>* алфавитно-цифрового типа (* не соответствует стандарту *). В соответствии со стандартом, *<строковое значение>* должно содержать только *<спецификацию столбца>*. *<Образец>* состоит из строки, относительно которой проверяется, входит ли она в *<строковое значение>*. *<Символ начала управляющей последовательности>* — это единственный алфавитно-цифровой символ.

Результат сравнения положителен, если выполняются следующие условия:

- Для каждого *<символа подчеркивания>* в *<образце>*, который непосредственно не предшествует *<символу начала управляющей последовательности>*, существует один соответствующий символ в *<строковом значении>*.
- Для каждого *<символа процента>* в *<образце>*, который непосредственно не предшествует *<символу начала управляющей последовательности>*, существует ноль или более соответствующих символов в *<строковом значении>*.
- Для каждого *<символа начала управляющей последовательности>* в *<образце>*, который непосредственно не предшествует другому *<символу начала управляющей последовательности>*, нет соответствующего символа в *<строковом значении>*.
- Для любого другого символа в *<образце>*, представлен точно такой же символ в соответствующей позиции *<строкового значения>*.

Если результат сравнения положителен, *<предикат "как">* принимает значение "истина", если только не указано NOT.

Предикат "как" A NOT LIKE 'text' эквивалентен NOT (A LIKE 'text').

<NULL-предикат>

Синтаксис:

```
<Спецификатор столбца> IS [NOT] NULL
```

Предикат *<спецификатор столбца>* IS NULL принимает значение "истина", если значение NULL представлено в указанном столбце. Предикат *<спецификатор столбца>* IS NOT NULL дает тот же результат, что и предикат NOT(*<спецификатор столбца>* IS NULL).

<*Составной* предикат>

Синтаксис:

```
<Выражение, содержащее значения> <оператор отношения>
<квантор><подзапрос>
<квантор> : : = ANY | ALL| SOME
```

Предложение SELECT *<подзапроса>* должно включать одно и только одно *<выражение, содержащее значения>*. Все значения, полученные на основании *<подзапроса>* составляют *<множество результата>*. *<Выражение, содержащее значения>* сравнивается с использованием *<оператора отношения>* для каждого члена из *<множества результата>*. Это сравнение оценивается следующим образом:

- Если *<квантор>* = ALL и каждый член *<множества результата>* дает при сравнении значение "истина", то *<составной предикат>* имеет значение "истина".

- Если *<квантор>* = ANY и существует, по крайней мере, один элемент из *<множества результата>*, для которого результат сравнения имеет значение "истина", то *<составной предикат>* имеет значение "истина".

- Если *<множество результата>* пусто, то *<составной предикат>* имеет значение "истина", если имеется *<квантор>* = ALL, и значение "ложь" — в противном случае.

- Квантор SOME действует так же, как и квантор ANY.

- Если *<составной предикат>* не имеет значения ни "истина", ни "ложь", то его значение неизвестно.

<Предикат существования>

Синтаксис:

```
EXISTS (<подзапрос>)
```

Если результат выполнения *<подзапроса>* — одна или несколько строк, то *<предикат существования>* имеет значение "истина", в противном случае — значение "ложь".

Команды SQL

Этот раздел уточняет детали синтаксиса для множества команд SQL. Он дает возможность быстро отыскать команду, найти ее синтаксис и краткое описание того, как она работает.

Замечание: Команды, начинающиеся с EXEC SQL, и команды или предложения, заканчивающиеся на *<SQL-терм>*, могут использоваться только во вложенном SQL.

BEGIN DECLARE SECTION

Синтаксис

```
EXEC SQL BEGIN DECLARE SECTION <SQL-терм>;
<объявления переменных языка высокого уровня>
EXEC SQL END DECLARE SECTION <SQL-терм>;
```

Команда создает секцию программы языка высокого уровня для объявления переменных в основной программе, которые будут использоваться во встроенных SQL-предложениях. Переменная SQLCODE должна быть включена как одно из объявлений переменных языка высокого уровня.

CLOSE CURSOR

Синтаксис

```
EXEC SQL CLOSE CURSOR <имя курсора><SQL-терм>;
```

Команда определяет CURSOR (курсор) закрытым, значит из него нельзя получить никакого значения до тех пор, пока он вновь не будет открыт.

COMMIT (WORK)

Синтаксис

```
COMMIT WORK;
```

Команда делает постоянными все изменения значений, выполненные в базе данных от момента начала транзакции до текущего момента, и начинает новую транзакцию.

CREATE INDEX (нестандартная *)*

Синтаксис

```
CREATE [UNIQUE] INDEX <имя индекса>
    ON <имя таблицы> (<список столбцов>);
```

Команда открывает пути быстрого доступа к данным для обеспечения более эффективный доступ к строкам, содержащим определенные значения в столбцах. Если задано ключевое слово UNIQUE, то таблица не может содержать повторяющихся значений в этом столбце.

CREATE SYNONYM (нестандартная *)*

Синтаксис

```
CREATE [PUBLIC] SYNONYM<синоним> FOR <владелец>.<имя таблицы>;
```

Команда создает альтернативное имя для таблицы. Синоним принадлежит его создателю, а сама таблица обычно принадлежит другому пользователю. При использовании синонима его владелец не должен ссылаться на таблицу полностью, включая имя владельца таблицы. Если указано ключевое слово PUBLIC, то синоним признается имеющим статус SYSTEM и доступен всем пользователям.

CREATE TABLE

Синтаксис

```
CREATE TABLE <имя столбца>
        ({<имя столбца> <тип данных>[<размер>]
        [<ограничения на столбец>...]
        [<значение по умолчанию>]},... <ограничения на таблицу>.,..);
```

Команда создает таблицу в базе данных. Владельцем этой таблицы является ее создатель. Предполагается, что столбцы упорядочены по именам. *<Тип данных>* определяет, какого рода данные будут храниться в столбце. Стандартные *<типы данных>* описаны в приложении B; другие часто используемые типы данных обсуждаются в приложении C. Значение *<размер>* зависит от типа данных. *<Ограничения на столбцы>* и *<ограничения на таблицу>* накладывают ограничения на значения, которые можно вводить в столбцы. *<Значение по умолчанию>* определяет значение, которое автоматически проставляется, если никакое другое значение не определяется для строки (см. главу 17 для уточнения деталей команды CREATE TABLE и главы 18 и 19 для уточнения деталей, связанных с ограничениями и *<значениями по умолчанию>*).

CREATE VIEW

Синтаксис

```
CREATE VIEW <имя таблицы>
        AS <запрос>
        [WITH CHECK OPTION];
```

Представление рассматривается как любая другая таблица в командах SQL. Когда команда ссылается на *<имя таблицы>*, выполняется запрос и его результаты формируют содержимое таблицы на протяжении выполнения команды. Некоторые представления могут обновляться. Это означает, что для них можно выполнить команды обновления и распространить их действие на таблицы, указанные в *<запросе>*. Если

указано WITH CHECK OPTION, то эти обновления должны удовлетворять *<предикату>*, указанному в *<запросе>*.

DECLARE CURSOR

Синтаксис

```
EXEC SQL DECLARE <имя курсора> CURSOR FOR <запрос><SQL-терм>
```

Команда связывает имя курсора с запросом. Когда курсор открывается (см. OPEN CURSOR), выполняется запрос, и к его результату можно применить FETCHED. Если курсор является обновляемым, то таблица, на которую ссылается *<запрос>*, подвергается изменениям при выполнении операций над курсором (см. главу 25).

DELETE

Синтаксис

```
DELETE FROM <имя таблицы>
      { [WHERE <предикат>]; }
       WHERE CURRENT OF <имя курсора> <SQL-терм>
```

Если предложение WHERE отсутствует, то исключаются все строки таблицы. Если в предложении WHERE используется *<предикат>*, то исключаются строки, удовлетворяющие *<предикату>*. Если WHERE-предложение имеет аргумент CURRENT OF *<имя курсора>*, то строка из таблицы, имя которой определено в предложении FROM, исключается, если на нее ссылается *<имя курсора>*. Форма WHERE CURRENT OF может использоваться только для встроенного SQL и только с обновляемыми курсорами.

EXEC SQL

Синтаксис

```
EXEC SQL <команды встроенного SQL> <SQL-терм>
```

EXEC SQL используется для пометки начала SQL команд, встроенных в язык программирования.

FETCH

Синтаксис

```
EXEC SQL FETCH <имя курсора>
      INTO <список переменных включающего языка> <SQL-терм>
```

Команда выбирает текущую строку из результата ответа на *<запрос>* и включает ее в *<список переменных включающего языка программирования>*, затем перемещает указатель курсора на следующую строку. *<Список переменных включающего языка программирования>* может включать индикатор успешности получения списка переменных.

GRANT

Синтаксис (стандартный)

```
GRANT ALL [PRIVILEGES]
        | { SELECT
        | INSERT
        | DELETE
        | UPDATE [(<список имен столбцов>)]
        | REFERENCES [(<список имен столбцов>)] }.,..
ON <имя таблицы>.,..
TO PUBLIC <идентификатор полномочий>.,..
[WITH GRANT OPTION];
```

ALL с указанием или без указания ключевого слова PRIVILEGES включает каждую привилегию из указанного в фигурных скобках списка. PUBLIC включает настоящих и будущих пользователей. По этой команде передаются указанные в списке действия (полномочия или права доступа) с таблицей, имя которой задано с помощью ключевого слова ON. REFERENCES разрешает использовать столбцы из *<списка имен столбцов>* в качестве родительских для внешнего ключа. Другие привилегии заключаются в праве выполнять команды, указанные в таблице. Действие UPDATE, как и REFERENCES, может быть ограничено указанным множеством столбцов. GRANT OPTION наделяет правами передачи привилегий другим пользователям.

Синтаксис (общее отклонение от стандарта)

```
GRANT     DBA
        |     RESOURCE
        |     CONNECT.,..
        TO <идентификатор полномочий>.,..
        [IDENTIFIED BY > пароль>]
```

CONNECT дает определенные права, среди них право вести журнал. RESOURCE дает пользователю право создавать таблицы. DBA предоставляет практически неограниченные права. IDENTIFIED BY используется с CONNECT для изменения пароля пользователя.

INSERT

Синтаксис

```
INSERT INTO <имя таблицы> [ (<список имен столбцов>) ]
      VALUES (<список значений>) | <запрос>;
```

INSERT создает один или более новых столбцов в таблице, имя которой указано в команде. Если используется предложение VALUES, то значения, перечисленные в <*списке значений*>, вставляются в таблицу, имя которой указано в команде. Если в команде имеется запрос, то каждая строка его результата вставляется в таблицу, имя которой указано в команде. Если опущен <*список имен столбцов*>, то по умолчанию предполагается полный список имен столбцов таблицы, порядок перечисления имен полей определяется описанием структуры таблицы.

OPEN CURSOR

Синтаксис

```
EXEC SQL OPEN CURSOR <имя курсора> <SQL-терм>
```

OPEN CURSOR выполняет запрос, связанный с <*именем курсора*>. Выходные данные после выполнения этой команды можно получить построчно с помощью команды FETCH (по одной строке в результате выполнения одной команды FETCH).

REVOKE (не соответствует стандарту *)*

Синтаксис

```
REVOKE { ALL [PRIVILEGES]
      |    <привилегия>.,.. } [ON <имя таблицы>]
      FROM { PUBLIC
      |    <идентификатор полномочий>.,..};
```

<*Привилегия*> — любая привилегия из числа тех, что указаны в команде GRANT. Команду REVOKE может выполнить тот же пользователь, который выполнял соответствующую команду GRANT (одну или несколько). Предложение ON используется в том случае, когда привилегия специфична для конкретного объекта.

ROLLBACK (WORK)

Синтаксис

```
ROLLBACK WORK;
```

Команда отменяет все изменения, выполненные в базе данных за время текущей транзакции, а также заканчивает текущую транзакцию и начинает новую.

SELECT

Синтаксис

```
SELECT { [DISTINCT ALL] <выражение, содержащее значения>.,...} | *
    [INTO <список переменных включающего языка> (*только для
    встроенного варианта использования*)]
    FROM <ссылка на таблицу>.,..
    [WHERE <предикат>]
    [GROUP BY <столбец, по которому выполняется группирование>.,..]
    [HAVING <предикат>]
    [ORDER BY <столбец, по которому выполняется
    упорядочение> [ASC | DESC].,..];
```

Предложение образует запрос и выводит значения из базы данных (см. главы 3-14). Для него действуют следующие правила и определения.

- Если ни ALL ни DISTINCT не указаны, то предполагается ALL.
- *<Выражение, содержащее значения>* содержит *<спецификатор столбца>*, *<функцию агрегирования>*, *<нестандартную функцию>*, *<константу>*, либо любую их комбинацию с использованием операторов, формирующих правильное выражение.
- *<Ссылка на таблицу>* состоит из имени таблицы, включающего префикс владельца, если текущий пользователь не является ее собственником, либо синонима (* решение, отличное от стандарта *) таблицы. Таблица, на которую выполнена ссылка, может быть либо базовой таблицей, либо представлением. Можно также указать алиас, который является синонимом используемой таблицы только на время выполнения текущей команды. Имя таблицы или синоним могут быть отделены от алиаса с использованием одного или нескольких *<разделителей>*.
- Если употребляется GROUP BY, то все *<спецификаторы столбцов>*, применяемые в предложении SELECT, могут использоваться как *<столбцы, по которым выполняется группирование>*, независимо от того, входят ли они в состав *<функций агрегирования>*. Все *<столбцы, по которым выполняется группиро-*

вание> должны быть представлены в <*выражении, содержащем значения*> предложения SELECT. Для каждой отдельной комбинации значений из <*столбцов, по которым осуществляется группирование*> в результат включается одна и только одна строка.

- Если используется HAVING, то <*предикат*> применяется к каждой строке результата, формируемого в соответствии с предложением GROUP BY, если результат применения <*предиката*> для данной строки — "истина", то эта строка включается в состав выходных данных.

- Если используется ORDER BY, выходные данные имеют определенную последовательность вывода. Каждый <*идентификатор столбца*> ссылается на отдельный элемент <*выражения, содержащего значения*>, указанного в предложении SELECT. Если <*выражение, содержащее значения*> является спецификатором столбца, то <*идентификатор столбца*> совпадает со <*спецификатором столбца*>. В противном случае <*идентификатор столбца*> является положительным целым числом, указывающим расположение в <*выражении, содержащем значения*> соответствующего предложения SELECT. Выходные данные будут упорядочены в соответствии с предложением ORDER BY. В пределах одного столбца данные упорядочены по возрастанию, если не указано ключевое слово DESC. <*Идентификатор столбца*>, указанный в предложении ORDER BY первым, имеет преимущество перед остальными в завершающей последовательности строк выходных данных.

Предложение SELECT оценивает каждую возможную строку-кандидат таблицы (таблиц), полученную независимо. *Строка-кандидат* определяется следующим образом:

- Если включена только одна <*ссылка на таблицу*>, каждая строка этой таблицы рассматривается как строка-кандидат.

- Если включено более одной <*ссылки на таблицу*>, то каждая строка каждой таблицы комбинируется со всеми комбинациями строк из других таблиц. Каждая такая комбинация рассматривается как строка-кандидат.

Для каждой строки-кандидата значения подставляются в <*предикат*>, заданный в предложении WHERE; предикат принимает одно из значений: true, false, unknown. Если не используется GROUP BY, каждое <*выражение, содержащее значения*> применяется к каждой строке-кандидату, значения которой делают <*предикат*> истинным; результат этой операции добавляется к строкам выходных данных. Если используется GROUP BY, то строки-кандидаты комбинируются с использованием агрегатных функций. Если никакой предикат не указан, то каждое <*выражение, содержащее значения*> применяется к каждой строке-кандидату или к каждой группе. Если указано DISTINCT, то повторяющиеся строки исключаются из результата.

UNION

Синтаксис

```
<запрос> { UNION [ALL] <запрос> }...;
```

Выходные данные двух или более запросов объединяются. Каждый из запросов должен содержать одинаковое число элементов в <*выражении, содержащем значения*> в предложении SELECT, а элементы, стоящие на одном и том же месте в <*выражениях, содержащих значения*>, должны быть совместимы по типу данных и размеру.

UPDATE

Синтаксис

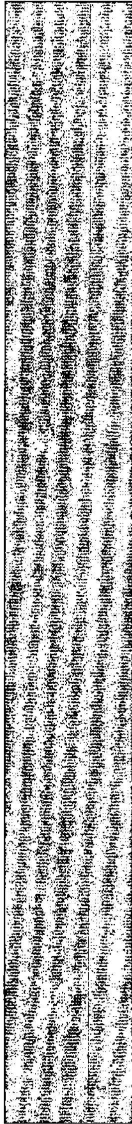
```
UPDATE <имя таблицы>
    SET { <имя столбца> = <выражение, содержащее значения> }.,..
    { [ WHERE <предикат> ];}
    |     { [ WHERE CURRENT OF <имя курсора> ]
          <SQL-терм> ] }
```

UPDATE выполняет замену значений столбцов, имена которых указаны в предложении SET, соответствующим значением из <*выражения, содержащего значения*>. Если в предложении WHERE указан <*предикат*>, то замена значений выполняется только в тех строках таблицы, на которых <*предикат*> принимает значение "истина". Если WHERE содержит предложение CURRENT OF, то происходит замена значений в текущей строке таблицы значениями из текущей строки курсора. Форма WHERE CURRENT OF обычно используется только во встроенном SQL и только с применением курсора. Если отсутствует предложение WHERE, во всех строках выполняется замена.

WHENEVER

Синтаксис

```
EXEC SQL WHENEVER <SQL-условие> <действие> <SQL-терм>
<SQL-условие> :: = SQLERROR | NOT FOUND | SQLWARNING
(в официальный стандарт не входит)
<действие> :: = CONTINUE | GOTO <адресат> GO TO <адресат>
<адресат> :: = зависит от включающего языка.
```



E

Таблицы, используемые в примерах

Таблица 1. Salespeople (продавцы)

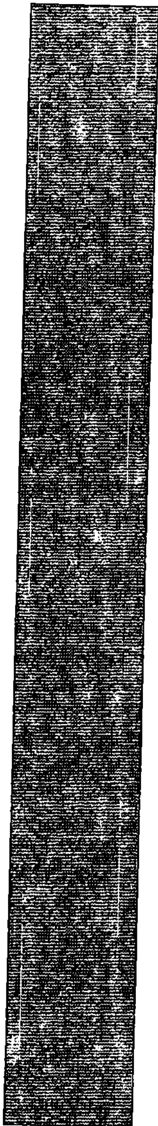
snum	sname	city	comm
1001	Peel	London	.12
1002	Serres	San Jose	.13
1004	Motika	London	.11
1007	Rifkin	Barcelona	.15
1003	Axelrod	New York	.10

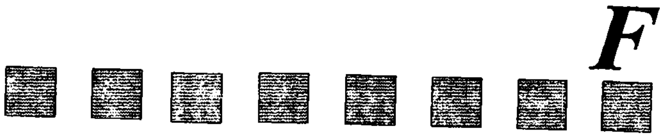
Таблица 2. Customers (покупатели)

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: Orders (заявки)

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





F

SQL сегодня

SQL сегодня

В настоящее время SQL переживает новый подъем. В качестве коммерческо- го продукта язык был впервые реализован в Oracle в 1976 году, но официального стандарта SQL не существовало до 1986 года, когда он был опубликован как результат объединенных усилий ANSI (the American National Standards Institute) и ISO (International Standards Organization). Поскольку ANSI является частью ISO, в данном приложении мы ссылаемся на обе эти организации как на ISO. Стандарт 1986 года был пересмотрен в 1989 году, в него были введены средства, обеспечивающие ссылочную целостность (referential integrity).

К тому времени, когда появился стандарт 86, ряд программных продуктов уже использовал SQL, и ISO попытался закрепить в стандарте наиболее общие черты этих реализаций для того, чтобы ввод стандарта не отразился слишком болезненно на готовых программных продуктах. ISO проанализировал все основные характеристики существовавших к тому времени программных реализаций и определил весьма минимальный стандарт. Некоторые существенные характеристики, например, как уничтожение объектов и передача привилегий, были опущены из стандарта полностью. Теперь, когда многообразный компьютерный мир стал столь коммуникабельным, разработчики и пользователи хотят без особых проблем взаимодействовать с множеством баз данных, разработанных индивидуально. В результате возникла потребность в стандартизации тех характеристик, которые ранее были отданы на усмотрение разработчика. Несколько лет эксплуатации конкретных систем и теоретических исследований дали новые идеи, которые требуют единообразия при воплощении в программных продуктах. Для удовлетворения этих потребностей ISO разработал новый стандарт SQL 92 .

Стандарт SQL 92 превышает первый стандарт SQL по объему примерно в пять раз. В нем значительно расширена область стандартизации, а также определен стандарт для ряда существовавших характеристик, которые до этого были отданы на волю разработчика, включены те моменты, которые ранее были опущены. Поскольку стандарт SQL 92, включает как подмножество стандарт 89, можно ссылаться на стандарт 92, если программный продукт удовлетворяет требованиям стандарта 89 или некоторым промежуточным требованиям.

Естественно, те продукты, которые используются, могут по-своему расширять стандарт. Для того чтобы прикладной программист мог выделять все специфические моменты, имея дело с такими расширениями, новый стандарт требует применения флаггера (flagger) — программы, проверяющей основной код и помечающей (маркирующей) все предложения SQL, не соответствующие стандарту 92. Непомеченные предложения ведут себя так, как описано в этой книге. Для помеченных предложений, конечно, следует применять системную документацию. Возможна ситуация, при которой предложение соответствует стандарту, а его поведение — нет. Такое предложение также помечается. В любом случае, этот стандарт более полон, чем предыдущий, поэтому необходимость в стандартных характеристиках для достижения функциональной полноты программного продукта практически отпала.

Пользователи, схемы и сеансы связи

Одной из областей, в которой старый стандарт отсутствовал, а новый стандарт существенно улучшен, является определение контекста применения SQL — определения пользователя, схемы и сеанса. Предыдущий стандарт отдавал эти области практически полностью на усмотрение разработчика базы данных. В этом разделе мы приводим обзор среды SQL в соответствии со стандартом 92.

В SQL имеются средства организации данных. Данные содержатся в таблицах, таблицы группируются в схемы, схемы группируются в каталоги. Каталоги могут далее группироваться в кластеры. Некоторые системы управления базами данных (СУБД) используют эти термины не так, как определено в стандарте, об этом может свидетельствовать системная документация.

С точки зрения отдельного сеанса SQL, кластер является целым миром. Он содержит все таблицы, доступные в данном сеансе, а все связанные таблицы должны находиться в одном кластере. Однако стандарт оставляет на усмотрение разработчиков возможность использования каталогов, отражающих связь кластеров.

В стандарте уточняется смысл требований SQL относительно того, кто и какие предложения может задавать. Тот, кто выдает предложения SQL, называется SQL-агентом. Им может быть пользователь, непосредственно работающий с SQL, или приложение. Агент SQL устанавливает связь с СУБД. Как только связь установлена, начинается сеанс. Реализации позволят SQL-агентам переключаться на некоторые другие связи и сеансы. В среде клиент/сервер эти связи или сеансы могут быть полностью адресованы другому серверу. В любой момент времени может существовать определенный текущий сеанс, который является активным и, возможно, несколько других, которые в этот момент не действуют, но находятся на разных стадиях выполнения. Привилегии выполнения предложения могут быть связаны с пользователями или модулем какого-то языка программирования, если таковой имеется.

Рассмотрим среду (контекст), в которой используются SQL-предложения и ряд новых особых характеристик в стандарте SQL 92.

Что нового в стандарте 92?

Стандарт 92, за редкими исключениями, полностью включает в себя стандарт 89. Если он вам известен, то достаточно ознакомиться лишь с обзором того, что было к нему добавлено.

Предложения определения схемы. Схема (shema) — это множество объектов базы данных, которые управляются единственным пользователем и могут рассматриваться как единое целое. Предыдущий стандарт SQL определял процедуры создания и удаления таблиц и других объектов, но он просто отождествлял схемы с идентификаторами пользователей (authorization IDs), как правило, понимая под пользователем его ID, что не оговорено в стандарте. Для пользователей, которые могут захотеть создать более одной схемы, новый стандарт включает предложения для создания и уда-

ления схем так же как и таблиц и других объектов. Схемы разрешается также группировать в каталоги, которые, в свою очередь, могут быть сгруппированы в кластеры.

Временные таблицы. В первом стандарте было выделено два типа таблиц: постоянные базовые таблицы (base tables) и представления (views). Базовые таблицы — это основные (базовые) данные, а представления — это, так называемые, виртуальные таблицы (virtual tables) — таблицы, выводимые из базовых с помощью запросов. Оба типа таблиц — постоянные, хотя содержимое представления не определено до тех пор, пока к нему не осуществляется доступ. Теперь в дополнение к постоянным базовым таблицам и представлениям появились три типа временных базовых таблиц. Два из них (глобальные временные таблицы и созданные локальные временные таблицы) имеют, подобно представлениям, постоянные определения, как объекты в схеме, но их содержимое постоянно не хранится в базе данных. Третий вид — локальные временные таблицы не имеет даже постоянно хранящегося определения. В отличии от представлений, временные таблицы не являются только альтернативными представлениями данных, содержащихся в базовых таблицах или выводимых из них, а содержат и свои собственные данные, которые автоматически создаются по концу сессии или транзакции. Следовательно, содержимое определения временной таблицы является определением базовой таблицы, но данные в ней не хранятся постоянно. С другой стороны, содержимым представления является запрос, используемый для того, чтобы вывести данные, когда к представлению осуществляется доступ. Временные таблицы, как правило, используются для рабочей памяти или для получения промежуточных результатов подобно переменным в программе, которые являются полезными в течение некоторого времени и не влияют на завершение выполнения программы. Представления полезны для того, чтобы придать данным базовых таблиц нужную форму, и для определения управления доступом.

Встроенные операторы JOIN. Способность выполнять соединения является частью предложения SELECT, но ранее не было механизмов встраивания SQL в автоматически генерируемые соединения различных типов. При определении множества таблиц (или многократных вхождений одной и той же таблицы) в предложении FROM для запроса, предполагающего соединение, считается, что в оставшейся части предложения определен тип соединения, который нужно выполнить. Например, новый стандарт имеет встроенные операторы для получения соединения следующих типов:

Cross	Это декартово произведение — все возможные комбинации строк, входящих в соединенные таблицы.
Natural	В принципе, это соединение внешнего ключа с родительским, на который он ссылается. В стандарте термин используется в более общем виде, как эквисоединение (соединение по равенству, по совпадению) двух или более таблиц для случая

	совпадения значений из столбцов, имеющих одинаковые имена. (Эквисоединение — наиболее общий тип, это любое соединение, базирующееся на равенстве значений из столбцов, в противоположность ситуации, когда одно значение больше другого). Иными словами, стандарт предполагает, что внешний ключ следует за родительским. В противном случае можно просто получить естественное соединение.
Inner (outer)	Это такое эквисоединение таблиц A и B, при котором каждая строка, представленная в одной таблице, имеет соответствующую строку в другой таблице.
Left (outer)	Включает все строки из таблицы A, независимо от того, сопоставимы они или нет, плюс все сопоставимые значения из B, если это возможно. Несовпадающие строки помечаются как NULL. В общем случае слово "outer" является необязательным и определяет, что несовпавшие строки представлены наряду с совпавшими.
Right (outer)	Соединение, противоположное Left: все строки из таблицы B представлены в конъюнкции с любой сопоставимой строкой из A.
Full	Комбинация левого и правого соединений. Все строки для каждой из таблиц представлены на основании того, что совпадение было обнаружено.
Union	Объединяющее соединение противоположное внутреннему соединению — оно включает только те строки из каждой таблицы, для которых не было обнаружено соответствия. Если взять полное внешнее соединение и удалить все, содержащееся в обладающем такой же структурой внутреннем соединении, то объединяющее соединение совпадает с левым. (Не путать объединяющее соединение с оператором объединения UNION, используемым для объединения выходных данных множества запросов).

Все указанные типы соединений поддерживаются специальными операторами в предложениях запросов FROM.

Курсоры типа READ-ONLY, SCROLLABLE, INSENSITIVE, DYNAMIC. *Курсор* — это объект, используемый для хранения выходных данных запроса на обработку для приложения. В стандарте SQL 89 курсор был обновляемым в принципе, то есть не существовало серии правил и предложений, защищающих от обновлений. Теперь можно объявить его "read-only" (только для чтения). Помимо расширения сек-

ретности, обновляемые курсоры ускоряют выполнение команд, уменьшая потребности в установке замков защиты данных.

Sensitivity (чувствительность) должна выполняться для того, чтобы отразить в курсоре внешние изменения данных, поскольку курсор транслирует операцию set-at-a-time в операцию item-at-a-time, обычную для языков программирования. Другими словами, курсор может быть открыт, а затем прочитан (вызван) постепенно. Это дает возможность изменять данные, на которые указывает курсор, в то время, когда они еще читаются. Что случается, если другое предложение изменяет данные, которые еще читаются открытым курсором? В соответствии со старым стандартом, ответ мог быть только один "Знает только бог!" Согласно же стандарту SQL 92, можно объявить курсор *нечувствительным (insensitive)*, и в этом случае он будет полностью игнорировать внешний мир. Нечувствительными могут быть только курсоры read-only. Нынешний стандарт не позволяет объявить курсор чувствительным для того, чтобы в нем отражались внешние изменения данных. (Можно оставить курсор неопределенным, и тогда все, что с ним произойдет, зависит только от конкретной реализации. Остается надеяться на то, что в данной реализации по крайней мере нет противоречий и случаи, связанные с неопределенностью курсора оговорены в системной документации.)

Кроме того, можно задать *прокручивающиеся курсоры (scroll)*. В общем случае строки в таблице являются неупорядоченными, а строки связанные курсором, могут иметь произвольный или определенный порядок. Ранее было необходимо оперировать строками по одной за единицу времени (за один шаг обработки данных), начиная с первой и перебирая их по одной вплоть до последней. Прокручивающийся курсор позволяет перемещаться скачками, возвращаться назад по мере необходимости и т.д. Ом должен иметь статус "только для чтения".

Последнее расширение концепции курсора — динамические курсоры т.е. курсоры динамического языка SQL, которые не поддерживались в предыдущем стандарте. В таком языке заранее неизвестно, какой запрос должен содержать курсор, поэтому содержимым курсора считается строковая переменная, которая может быть множеством текстовых данных, полученных по запросу во время его выполнения. Фактически, само имя курсора можно сделать переменным, используя предложение ALLOCATE CURSOR вместо обычного DECLARE CURSOR, поэтому вы не должны знать заранее, сколько курсоров понадобится пользователю приложения. Подобно статическим курсорам, динамические курсоры могут быть "только для чтения", нечувствительными и прокручивающимися.

Ориентация клиент/сервер. Новый стандарт позволяет управлять связями, распознавая всякую общую конфигурацию базы данных, на которой установлено программное обеспечение конечного пользователя (front-end) на одном компьютере, называемом <клиентом> (client), который пытается извлечь информацию из СУБД ("back-end"), расположенной на другом компьютере, называемом <сервером> (server). Много новых средств, относящихся к стандартизации процедур связи, схем запирания и диагностики ошибок было мотивировано желанием клиентов единообразно взаимодействовать с множеством СУБД и на множестве серверов.

Что такое архитектура клиент/сервер? В архитектуре клиент/сервер множество компьютеров объединено в сеть, в которой все компьютеры подразделяются на клиентов и серверов. Пользователи непосредственно взаимодействуют с клиентами для выполнения большей части функций конечного пользователя. Серверы выполняют различные интенсивные задания в ответ на запросы клиентов. СУБД обычно размещается на сервере и занимается обслуживанием требований клиентов. SQL хорошо соответствует таким соглашениям, поскольку, являясь декларативным языком, он очень лаконичен, и значит сеть не перегружена передачей детальных инструкций между клиентом и сервером. Такой язык является кратким конспектом, позволяющим серверу выполнить требуемую работу автоматически без последующего участия (использования) клиента. Для запросов это является усовершенствованием более старого подхода, основанного на использовании файлового сервера, где сервер может передавать клиенту целую транзакцию и предоставлять ее для извлечения данных в случае необходимости.

В архитектуре клиент/сервер важное значение имеет вопрос о связях. Клиент должен быть связан с сервером для взаимодействия с ним. При этом стандарт нейтрален относительно частностей реализации, например, относительно того, какие платформы и сети используются. Стандарт 92 определяет, что такое SQL-связь и содержит несколько основных правил регламентирующих поведение пользователей в ряде ситуаций. Он базируется на реалиях компьютерной технологии клиент-сервер, в которой клиенты стремятся взаимодействовать с множеством серверов, а серверы обычно взаимодействуют с множеством клиентов.

Это не означает, что стандарт применим только к архитектурам клиент-сервер. SQL 92, как и его предшественник, является функциональной спецификацией, приемлемой для любой конфигурации: отдельных персональных компьютеров, стандартных миникомпьютеров и mainframe компьютеров.

Более сложное управление транзакциями. *Транзакция* — это группа правильных предложений SQL, которые могут выполняться или не выполняться все вместе. Ошибка в транзакции приводит к тому, что вся последовательность операций может быть отменена (canceled) или для нее может быть выполнен откат ("rolled back"). СУБД автоматически начинает транзакцию всякий раз, когда применяется предложение, вызывающее ее выполнение, при условии, что никакая другая транзакция не является активной. Транзакции заканчиваются предложением COMMIT WORK (для того чтобы сохранить изменения) или предложением ROLLBACK (при отказе от внесения изменений или в случае сбоя или разъединения системе). Если транзакция не может быть восстановлена, то следует выполнить ее откат; ROLLBACK никогда не выполняется ошибочно.

Все это учитывалось в стандарте 86. В стандарте SQL 92 новым является следующее:

- Транзакция может быть определена "только для чтения" (read only). Это означает, что предложения внутри транзакции, ориентированные на изменение содержимого или структуры базы данных, вызовут ошибку, что способствует

улучшению выполнения текущих операций, поскольку при этом нет необходимости блокировать данные.

- Нужно сохранять проверку ограничений до конца транзакции и можно указывать желаемые ограничения. Ограничения управляют содержимым базы данных.
- Транзакции могут специфицировать уровни изоляции блокировок, накладываемых на данные.
- Транзакции могут специфицировать размер области диагностики для предложений внутри транзакции.

Уровни изоляции. Предшествующий стандарт предполагал управление транзакциями, но совсем не учитывал совпадений, т.е. ситуаций, в которых множество различных пользователей используют одни и те же данные по-разному. Естественно, что многопользовательские системы на рынке должны были иметь и имели дело с действительностью, но теперь в ISO осуществлена некоторая стандартизация. Системы имеют четыре уровня изоляции транзакций: READ UNCOMMITTED, READ COMMITTED, READ REPEATABLE, SERIALIZABLE. Кроме того, можно определить, что транзакция является "только для чтения" (read-only), а для уровня изоляции READ UNCOMMITTED это необходимо.

Привилегии владельца уровня приложения. Действия, выполняемые на базе данных, связаны с идентификатором автора, имя которого уникально внутри базы данных. Привилегии, связанные с отдельным идентификатором пользователя, определяют, какие действия могут выполняться пользователем. Например, идентификатор пользователя может иметь привилегию искать данные в таблице или использовать трансляцию набора символов. Он связан непосредственно с пользователем и определяет тип его действий: работает ли пользователь в однопользовательском режиме на SQL или запускает приложения, взаимодействующие с базой данных. В последнем случае полезно передавать привилегии приложению, а не пользователю. Так что пользователи могут выполнять предложения в приложении, не имея тех же привилегий, которые они имеют при выполнении других операций над данными. Приложения, особенно, если они используют статический SQL, могут в значительной степени регламентировать привилегии пользователей и, следовательно, предоставляют хорошие средства защиты.

Иногда говорят, что привилегии, ориентированные на приложения, воплощают *права разработчика,* а модули, выполняемые пользователями, выполняются в соответствии с их собственными привилегиями, которые называются *правами вызова*. Существуют и другие названия привилегий, поскольку некоторые программные продукты предполагают привилегии на владение приложением (application-owned), считавшиеся до недавнего времени нестандартными. Сейчас они стандартизованы по крайней мере для языка модулей, языков программирования, и это одна из причин того, что модульный подход применяется, по-видимому, чаще других. Тем не менее SQL 92 еще не поддерживает application-owned привилегии для встроенного или динамического SQL. Для однопользовательского SQL привилегии, ориентированные на приложение, несущественны. Поэтому они являются необязательными для языков

программирования. Могут также существовать пользователи, выполняющие приложения с персональными привилегиями. Эта ситуация относится к классу "либо-либо". Не может быть пользователей, выполняющих приложения с комбинацией их собственных привилегий и привилегий приложения.

Процедура стандартного связывания. В предыдущем стандарте связь SQL-предложения с идентификатором пользователя определялась конкретной реализацией. Идентификаторы пользователей имеют привилегии при выполнении определенных предложений, и, если пользователи создают объекты (такие как таблицы) то получают право управления этими объектами. Они понятны пользователям, хотя стандарт об этом явно не говорит. С точки зрения СУБД или операционной системы, "пользователь" может не иметь соответствия один к одному с каким-либо другим пользователем или пользователями реального мира. В некоторых коммерческих программных продуктах единственный идентификатор пользователя применяется несколькими реальными пользователями или отдельные пользователи могут иметь по несколько идентификаторов. Способ распознавания идентификатора пользователя СУБД — процедура связи — обычно рассматривается только на уровне конкретной реализации.

Стандарт SQL 92 уточняет способ связи пользователей с СУБД непосредственно или через приложения. Значит, возможна ситуация, в которой данный пользователь имеет несколько текущих связей, одна из которых в настоящий момент является активной. Пользователи непосредственно осуществляют переключение связи с помощью предложения SET CONNECTION. Совпадающие связи являются частью той же транзакции. Таким образом новый, стандарт ориентирован на мир клиент/сервер.

При использовании статического SQL в модульном языке программирования можно осуществить связь идентификаторов пользователей и, следовательно, привилегий с приложением, а не с пользователем, и этот подход имеет преимущества. Смотрите предыдущий раздел "Привилегии владельца уровня приложения".

Стандартизация системных таблиц. Каталог (catalog) в новом стандарте рассматривается как набор схем. Он содержит информационную схему (Information_ Schema), которая представляет собой множество таблиц, описывающих содержимое схем: какие столбцы в каких таблицах содержатся, какие представления определены, какие привилегии связаны с каждым идентификатором пользователя и т.д. В ряде коммерческих продуктов некоторые из этих таблиц получили название "каталог". Стандартная информационная схема, определенная SQL 92, разрешает как пользователям, так и приложениям применять одни и те же процедуры для получения информации о любой схеме для любой СУБД, доступной для них.

Стандартные коды ошибок и диагностики. На предшествующих этапах развития SQL информация о результате SQL-операции передавалась через переменную числового типа SQLCODE. Значение этой переменной устанавливается автоматически после

выполнения каждого предложения для определения, что произошло при выполнении предложения. Предусмотрены три возможные ситуации:

- Значение 0 определяет успешное завершение.
- Значение 100 определяет, что предложение выполнилось правильно, но не произвело никаких действий или не создало выходных данных. Например, если нет данных, удовлетворяющих запросу или была попытка удаления строки, которой нет в таблице, то в процессе выполнения этих команд SQLCODE принимает значение 100.
- Любое отрицательное значение свидетельствует об ошибке.

Идея заключалась в том, чтобы каждому отрицательному значению поставить в соответствие вполне определенную ошибку, но конкретное отображение множества ошибочных ситуаций на множество отрицательных значений в стандарте не было определено (было отдано на усмотрение разработчиков). SQLCODE поддерживается для обеспечения совместимости "снизу вверх" в существующих программных продуктах и уже разработанных приложениях, но это вызывает определенные возражения, и теперь его использование не рекомендуется. Может оказаться, что в следующем стандарте он вообще не будет поддерживаться.

Новый подход заключается в использовании другой переменной, названной SQLSTATE. Это текстовая строка длиной в пять символов со стандартными значениями для различных классов ошибок, ошибочных ситуаций, зависящих от конкретной реализации. Фактически, есть два уровня детализации ошибок: классы и подклассы. Часто сообщение об ошибке будет использовать стандартный класс и соответствующий подкласс. Таким образом стандарт определяет природу ошибки, а подкласс дает более конкретную информацию. Для подклассов стандарта нет, но если класс описывает ошибку адекватно, то подкласс можно опустить (установить в 000).

В настоящее время стандарт предоставляет собой область диагностики с множеством сообщений и кодов, которые могут иметь место в процессе выполнения единственного предложения. Доступ к ее использованию осуществляется с помощью предложения GET DIAGNOSTICS.

Поддержка динамического SQL. Динамический SQL — код SQL, генерируемый во время выполнения приложений, — специально не поддерживался в старом стандарте, хотя многие продукты используют его до сих пор. Новый стандарт учитывает эту ситуацию. Самой важной характеристикой для поддержки динамического SQL в стандарте 92 являются динамические курсоры, предложения, генерируемые из строк текста во время выполнения, области диагностики и области описания. Очень важны также новые возможности связи.

Поддержка для C, ADA и MUMPS. Официальный стандарт 86 поддерживал только основной язык, но не встроенный SQL, хотя и содержал четыре добавления, оп-

ределяющие встроенный SQL для Pascal, Fortran, COBOL, PL/I. В стандарт 92 добавлены C, Ada, MUMPS и теперь все языки являются частью официального стандарта.

Домены. Теоретики реляционной модели данных, и среди них E.F.Codd — отец теории реляционных баз данных, стимулируют применение *доменов (domains)*, исходя из того, что тип данных должен определяться более точно, чем это позволяет сделать стандарт множества типов данных. Например, тип номеров телефонов отличается от типа номеров в кодах секретности. Хотя оба типа являются числовыми, не имеет смысла непосредственно сравнивать эти значения, так как они определены на различных доменах.

SQL 92 разрешает пользователям создавать домены как объекты в схеме, а затем определять столбцы таблиц на доменах, а не типы данных для них. Определение домена содержит тип данных, но может включать в себя и предложения, которые задают значение по умолчанию, одно или более ограничений (правила, которые ограничивают значения, допустимые в отдельных столбцах) и последовательность для сравнения значений (порядок сортировки для набора символов). Однажды определенные домены могут быть впоследствии расширены или удалены.

Домены являются обычным способом воспроизведения принятых ограничений, значений по умолчанию и последовательностей для сравнения значений единообразно для всей схемы. Может оказаться полезным применение подобных ограничений на некоторые таблицы, например, использование контрольных разрядов. Их удобно применять в больших и сложных схемах или тех, которые используют сложные данные, требующие множества ограничений, что актуально при разработке некоторых приложений.

Утверждения и отсрочка ограничений. Ограничения — это правила, устанавливаемые для ограничений тех значений, которые могут содержаться в столбцах. Ранее в определения базовых таблиц были включены ограничения двух видов: ограничения на столбцы и ограничения на таблицы. Первые были частями определения столбца и контролировали предложения ориентированные на вставку или изменение значения в столбце, а вторые — частью определения таблицы и, следовательно, могли содержать правила, используемые для контроля множества столбцов таблицы. В обоих случаях ограничения могли быть либо определенного, заранее известного типа, например, NOT NULL или UNIQUE, либо CHECK-ограничениями, позволявшими создателю таблицы генерировать выражения значений из столбцов. Если выражение принимало значение FALSE, то ограничение не удовлетворялось и предложение отвергалось.

Новый стандарт дает возможность определять *утверждения* (assertions) — ограничения, которые существуют как независимые объекты в схеме, а не в таблице. Это значит, что они могут ссылаться на множество таблиц и на выражения, содержащие значения из этих таблиц. Их можно использовать для того, чтобы удостовериться в том, что таблица никогда не является пустой. Утверждения позволяют формулировать основные принципы, которым должны удовлетворять данные, например, контролировать законность выполнения операций. Утверждения могут создаваться и удаляться. Можно также размещать ограничения в домене (см. раздел "Домены" данной главы).

Старая система ограничений успешно используется до сих пор с некоторыми новыми характеристиками. Одна из них заключается в том, что ограничения должны быть поименованы. Это позволяет вносить в них добавления и удалять, не ограничивая время их существования временем существования таблицы.

Итак, с целью управления можно задавать до конца транзакции как ограничения, так и утверждения (начиная с этого момента и до конца данного приложения ограничения и утверждения будут обозначаться одним термином *ограничения (constraitions)* , если явно не оговорено нечто иное). Это может привести к некоторой неоднозначности. Ограничения проверяются в любом из следующих случаев:

1. После выполнения каждого предложения, воздействующего на таблицу, на которую есть ссылка в предложении.

2. В конце каждой транзакции, содержащей одно или более предложений, влияющих на содержимое таблиц, на которые они ссылаются.

3. Всякий раз в тот момент, когда пользователь или приложение решают, что эту операцию необходимо выполнить.

При определении ограничения устанавливается, должна ли проверка выполняться непосредственно после выполнения каждого предложения или ее можно отложить до завершения выполнения транзакции. Если выбран второй вариант, то можно уточнить, какой режим проверки принимается по умолчанию. Затем, на протяжении выполнения транзакции он может быть изменен с помощью установки *вида* ограничений (constraints mode). (Неотсроченные ограничения, по определению, должны проверяться немедленно.) Можно установить способ проверки сразу для всех ограничений или для каждого отдельно. Если установить вид ограничений в значение Immediate, то они будут проверяться немедленно. Это полезно делать в случаях, упомянутых в пункте 3 приведенного описания процедуры управления.

Существует ряд моментов, которые необходимо учесть в том случае, если ограничения отличаются от рассматриваемых. Например, необходимо иметь две таблицы, в которых внешний ключ первой ссылается на вторую, а внешний ключ второй — на первую. Предположим, каждый внешний ключ содержит ограничение NOT NULL. Следовательно, некоторое значение внешнего ключа представлено в каждой строке любой таблицы любой момент времени. При попытке добавить содержимое какой-либо таблицы в первую таблицу, необходимо на время отказаться от проверки сохранения целостности, поскольку родительский ключ отсутствует в другой таблице непосредственно после выполнения операции вставки. Это пример *цикличности* (*circularity*). Решение должно отличаться от проверки ограничения FOREIGN KEY до тех пор, пока не выполнена вставка строк в обе таблицы.

Надо помнить, что если отложить проверку ограничений до конца транзакции, можно потерять сведения о действиях, которые были выполнены в процессе транзакции. Если детали этого процесса неважны, то можно сохранять последовательность выполнения всех проверок, а можно выбрать и быстрый режим для нее, не откладывая до конца транзакции.

Добавление и удаление объектов. Согласно стандарту SQL 86, однажды созданную таблицу нельзя изменить или удалить. В реальной жизни возникают ситуации, когда следует изменить содержимое созданной таблицы или вовсе от нее отказаться. Поэтому предложения ALTER TABLE и DROP TABLE вошли в стандарт де-факто, наряду с CREATE TABLE по спецификации OSI. К сожалению эти предложения, из которых наиболее сложным является ALTER, не всегда выполнялись одинаково для различных программных продуктов. Однако теперь OSI решил сделать ALTER и DROP частью стандарта.

Можно изменять не только таблицы, но и все объекты: утверждения, домены, схемы, наборы символов, объединения, транзакции и представления. Домены можно и удалять. Изменение доменов эквивалентно изменению значений, принятых по умолчанию, добавлению или удалению ограничений. Предложение ALTER TABLE позволяет добавлять, изменять и удалять правила умолчания или ограничения, а также добавлять или удалять столбцы.

Отмена и использование привилегий. Привилегии — это то, что дает идентификатору пользователя право выполнять действия на различных объектах базы данных. По старому стандарту нельзя было лишиться однажды полученных привилегий. Однако разработчики программных продуктов ввели предложение REVOKE, отменяющее привилегии, переданные по команде GRANT. Теперь REVOKE является частью стандарта SQL 92. Оно относится к числу сложных предложений, прежде всего из-за необходимости прослеживать путь передачи привилегий от одного идентификатора пользователя к другому.

Предложение GRANT можно применять также с привилегией USAGE, чтобы открыть доступ к любому новому из числа возможных объектов схемы: к доменам, объединениям, наборам символов, способам трансляции. USAGE применимо только к этим новым объектам схемы, тогда как другие привилегии относятся только к базовым таблицам и представлениям. INSERT же определено по-новому: оно употребляется теперь и со столбцами. ISO предполагает разработать специфицированный на уровне столбца SELECT в следующем стандарте, именно для этого предусмотрено место в Information_Schema (информационной_схеме). Но в настоящее время эта забронированная в стандарте возможность не используется.

Определенные пользователем наборы символов, сравнения и трансляции. Возможность поддержки разных наборов символов (национальных алфавитов) делает SQL 92 действительно стандартом мирового сообщества. Новый стандарт позволяет разработчикам и пользователям проявлять большую гибкость в определении их собственных наборов символов. Теперь символы не обязаны иметь длину в один байт, как это было принято для символов английского языка, они могут иметь различную длину, но обязательно должны быть упорядочиваемыми. Для них должна быть определена такая последовательность сравнения (упорядочения), при которой предложение типа <символ 1> < <символ 2> может быть оценено либо как "истина", либо как "ложь", при условии, что ни символ 1, ни символ 2 не имеют значения NULL. Обычно сравнение совпадает с алфавитным порядком. В любом случае сравнение может быть переопределено, даже для стан-

дартных наборов символов. Можно указать свои собственные способы трансляции из одного набора символов в другой. Наборы символов, сравнения и способы трансляции — это объекты схемы и пользователи должны иметь привилегию USAGE для работы с ними.

Типы данных дата, время, интервал. Ранее дата и время были представлены с помощью стандартных типов символов — алфавитного и числового. Одна из возникавших при этом важных проблем связана с тем, что набор операций для этих типов отличался от набора операций, определенных для обычных чисел. Поэтому для большинства программных продуктов в стандарт добавлены два новых типа: дата и время. Они представлены следующим образом:

- Дата содержит год, месяц и день.
- Время содержит часы, минуты, секунды и доли секунды.
- Временная метка является комбинацией даты и времени. Метка времени по умолчанию состоит из четырех компонентов, например, 11:09:48.5839. По умолчанию время не включает долей секунды, но при необходимости они могут быть указаны. И время, и временная метка могут иметь индикатор зоны времени, показывающий точность времени относительно универсального координатора времени (UCT, Universal Coordinated Time).

SQL 92 поддерживает также два типа *интервалов* (*intervals*): год-месяц и день-время. Они используются для фиксации разницы между переменными даты и времени и позволяют представить дату и время в арифметическом виде. Например, если вычесть время 2:00 из времени 5:30, получится интервал 3:30 (три с половиной часа). Аналогично, можно взять значение даты "июль 1993 г.", добавить интервал — значение месяца равно трем — и в результате получить дату "октябрь 1993 г.". Интервалы имеют формат или год-месяц, или день-время, но могут быть представлены не все компоненты года или времени. Реально интервал является арифметическим и работает так, как будет указано. Существует два типа интервалов, и в общем случае нельзя сказать, сколько дней содержит месяц (это зависит от месяца), т.е. при увеличении значений дня месяца неизвестно, о каком интервале идет речь и когда можно увеличивать значение месяца.

Типы двоичных данных. Двоичный тип данных поддерживается не стандартом, а некоторыми программными продуктами. Он часто называется типом двоичных данных BLOPs (Binary Large Objects — большие двоичные объекты) и обычно используется в многоплатформных базах данных для хранения графических изображений, звуковых данных и т.д. BLOPs полезен также в научных и технических базах данных.

Существуют два типа двоичных данных: фиксированной и переменной длины. Несмотря на то, что на практике чаще нужны данные переменной длины, использование данных фиксированной длины улучшает работу системы и сокращает необходимый

объем памяти. Объекты фиксированной длины можно использовать, например, для работы с цифровыми образами. Если все образы имеют одинаковый размер и разрешающую способность (что весьма вероятно), и никакого сжатия образов не применяется (что возможно), все они будут одного размера.

Преобразования типов данных. SQL является строго типизированным языком. Все типы данных представлены в СУБД в двоичном виде, но операции не могут свободно их смешивать и использовать как двоичные или числовые, что допустимо в ряде языков программирования. Такой подход имеет свои достоинства и недостатки. Иногда полезно преобразовывать данные одного типа в данные другого типа, но могут возникнуть проблемы, если этот процесс неуправляем. Стандарт 92 разрешает выполнять преобразования типов данных с помощью выражения CAST, что дает возможность использовать инструкции СУБД для преобразования типа integer в тип символьной строки (character string) или символьной строки в двоичные данные. Это актуально для соединений и объединений, где все столбцы должны иметь одинаковые типы данных. (Можно также выполнять преобразования наборов символов или представлений наборов символов, используя TRANSLATE или CONVERT, соответственно.)

Новые встроенные системные значения. SQL представляет системные значения в форме встроенных строковых переменных, значения которых автоматически устанавливаются системой и отражают применяемый идентификатор пользователя для идентификации конкретного пользователя и его привилегий, определенных на данный момент. Определены три типа: SESSION_USER, CURRENT_USER, SYSTEM_USER.

CURRENT_USER, известный как USER, ссылается на идентификатор пользователя, который применяется для определения действий, выполняемых в настоящий момент. Это может быть идентификатор либо непосредственно пользователя, либо модуля. Последнее происходит тогда, когда выполняется модуль, имеющий собственный идентификатор пользователя (см. "Привилегии приложения" в этом разделе). В противном случае CURRENT_USER определяет пользователя. При любых обстоятельствах идентификатор пользователя связан с пользователем, который определен в переменной SESSION_USER. (Хотя SESSION_USER устанавливается автоматически, реальные SQL-системы позволяют изменять это значение с помощью предложения SET SESSION AUTORIZATION).

SYSTEM_USER — конкретный пользователь, определенный операционной системой. Наиболее часто это значение совпадает со значением SESSION_USER, однако соответствия между пользователями операционной системы и идентификаторам определяются конкретной реализацией — в двух различных СУБД они могут быть разными. Для уточнения деталей следует ознакомиться с системной документацией.

Новые операции над строками. Стандарт SQL 92 учитывает операцию конкатенации и некоторые функции для работы с текстовыми строками. Во-первых, это операции, которые выполняют действия над строками и в результате формируют

строки: оператор конкатенации (CONCATENATE) (записывается как ||), SUBSTRING, UPPER, LOWER, TRIM, TRANSLATE, CONVERT. Во-вторых, это операции, которые выполняют действия над строками, но в результате выдают числовые значения: POSITION, CHAR_LENGTH, OCTET_LENGTH, BIT_LENGTH.

В первой группе только операция CONCATENATE является *двухместной* (*diadic*), т.е. в каждой такой операции участвуют две строки. Она добавляет содержимое второй строки в конец первой. Например, в результате выполнения операции 'Jello ' || 'Biafra' получаем 'Jello Biafra'. SUBSTRING выбирает из строки то количество символов, которое необходимо извлечь, начиная с заданной позиции в этой строке, и создает, в результате, новую строку. Например, SUBSTRING ('Astarte' FROM 2 FOR 4) дает значение 'star'. UPPER и LOWER переводят все символы строки соответственно в большое (заглавными буквами) и малое (строчными буквами) написание; каждая из этих операций называется подгонкой (fold). TRIM используется для того, чтобы исключить передние или хвостовые пробелы из строки. Можно задать режим удаления передних пробелов, хвостовых пробелов, либо и тех, и других.

TRANSLATE и CONVERT поддерживают возможность использования в SQL двух наборов символов: общепринятого для конкретного пользователя и специфического. TRANSLATE осуществляет преобразование одного набора символов в другой. CONVERT, не изменяя набора символов, осуществляет переключение между различными представлениями набора символов.

POSITION находит начальное положение одной строки внутри другой, например, результат выполнения POSITION ('star' IN 'Astarte') дает значение 2. Если вхождения не обнаружено, то результат выполнения операции 0. Оставшиеся три функции сообщают длину строки, во множестве символов, октетов (8-битовая последовательность, обычно называется байтом) или битов. В обычных вариантах кодирования ASCII и EBCDIC количество символов совпадает с количеством октетов, но так как SQL предназначен для использования различных наборов символов, может потребоваться более (или менее) одного октета на символ.

Конструкторы значений строк. Предикаты в SQL сравнивают значения в терминах операторов "=" или в терминах собственных операторов SQL и возвращают значение "истина", "ложь" либо (в случае присутствия NULL-значений) "неизвестно", в зависимости от результатов сравнения. Раньше SQL-предикаты могли сравнивать одно значение с другим, а иногда и одно значение — со значениями из столбца, полученного в результате выполнения подзапроса (запрос может получать значения для использования их в другом запросе). Согласно же новому стандарту, SQL имеет дело с множествами значений, соответствующих строкам. Например, раньше можно было записать предикат:

```
WHERE c1 = 3
```

или

```
WHERE c1 = 3 AND c2 = 5
```

Согласно SQL 92, последнее выражение можно записать таким образом:

```
WHERE (c1,c2) = (3,5)
```

Это принципиально новая возможность, поскольку заключенные в круглые скобки списки значений, называющиеся *конструкторами значений строк (row value constructions)*, могут включать в себя подзапросы. Появился эффективный способ сравнения целых таблиц (т.е. любой комбинации строк и столбцов). Если вместо равенства используется неравенство, выражение является отсортированным, причем первое значение имеет наивысший приоритет, а все последующие выстроены в порядке его уменьшения. Например:

```
(1, 7, 8) < (2, 0,1)
```

имеет значение "истина", поскольку 1 < 2 (сравнение начинается с элементов с высшим приоритетом, после нахождения пары с неравными значениями оставшиеся элементы игнорируются).

Уточнение ссылочной целостности. В SQL 89 внешние ключи состоят из одного или более столбцов таблицы (таблица А), которые ссылаются на один или более столбцов другой таблицы (таблица В). Отсюда следует, что любой строке таблицы А, не имеющей NULL-значений среди значений внешнего ключа, должна соответствовать строка таблицы В с теми же значениями в столбцах, на которые осуществляется ссылка. Они являются родительским ключом (parent key). Родительский ключ таблицы В должен иметь ограничение UNIQUE либо ограничение PRIMARY KEY, что гарантирует наличие уникальных значений.

В SQL 92 все гораздо сложнее. Во-первых, совпадение внешнего ключа с родительским может быть либо частичным, либо полным. Различия проявляются при использовании составного ключа (определенного на множестве столбцов), который может содержать NULL-значения. Частичное совпадение предполагает совпадение значений, отличных от NULL-значений во внешнем ключе; полное совпадение предполагает совпадение всех значений. В ограничении внешнего ключа указывается тип применяемого совпадения.

Новый стандарт предусматривает *действия, переключаемые по ссылке (referential trigged actions)*, которые в некоторых существующих программных продуктов называются эффектами обновления и удаления. Они определяют, что произойдет при изменении значения родительского ключа, на который ссылается одно (или более) значение внешнего ключа. Можно определить этот эффект независимо для ON UPDATE и для ON DELETE. В каждом случае имеется четыре возможности:

- SET NULL. Устанавливает в NULL все значения внешнего ключа, который ссылается на удаленный или обновленный родительский ключ.

- SET DEFAULT. Устанавливает все столбцы по ссылке внешнего ключа в значение, определенное как значение по умолчанию в соответствующем предложении. Если значение по умолчанию явно не определено, используется NULL-значение.

- CASCADE. Означает автоматическое изменение значения внешнего ключа при изменении значения родительского.

Команда SELECT

```
SELECT * | { [DISTINCT | ALL] <список полей>.,.. }
   FROM {<имя таблицы> [<алиас>] }.,..
   [    WHERE <предикат> ]
   [    GROUP BY { <имя столбца> | <целое> }.,.. ]
   [    HAVING <предикат> ]
   [    ORDER BY { <имя столбца> | <целое> }.,.. ]

[ { UNION [ALL]

SELECT * | { [DISTINCT |  ALL] <список полей>.,.. }
   FROM { <имя таблицы> [<алиас>] }.,..
   [    WHERE <предикат> ]
   [    GROUP BY { <имя столбца> |  <целое> }.,.. ]
   [    HAVING <предикат> ]
   [    ORDER BY { <имя столбца> | <целое> }.,.. ]
   } ]...;
```

Элементы, используемые в команде SELECT

Элемент	**Определение**
<список полей>	Выражение, производящее значение. Оно может включать имена столбцов или состоять из них.
<имя таблицы>	Имя или синоним для таблицы или представления.
<алиас>	Временный синоним имени таблицы, определенный здесь и используемый только в этой команде.
<предикат>	Условие, которое может быть истинным или ложным для каждого столбца или комбинации столбцов из таблицы (таблиц), определенных предложением FROM.
<имя столбца>	Имя столбца таблицы.
<целое>	Число без десятичной точки. В этом случае оно определяет значение из списка выбираемых полей в предложении SELECT, указывая его расположение в этом предложении.

Команды обновления

UPDATE

```
UPDATE <имя таблицы>
    SET { | }.,..<имя столбца> = <список полей>
    [ WHERE <предикат>
    | WHERE CURRENT OF <имя курсора>
         (* только вложенный *)];
```

INSERT

```
INSERT INTO <имя таблицы> [(<имя столбца>.,..)]
{ VALUES (<список полей>).,..)}
| <запрос>;
```

DELETE

```
DELETE FROM <имя таблицы>
[ WHERE <предикат>
| WHERE CURRENT OF <имя курсора>
    (* только вложенный *)];
```

Элементы, используемые в командах обновления

Элемент	Определение
<имя курсора>	Имя курсора, используемого в этой программе.
<запрос>	Допустимая команда SELECT.

Остальные элементы определены в команде SELECT.

Символы, используемые при описании синтаксиса

<table>
<tr><th>Элемент</th><th>Определение</th></tr>
<tr><td>|</td><td>Все то, что предшествует данному символу, можоно заменить тем, что следует за ним. Этот символ используется там, где мы хотим сказать "или".</td></tr>
<tr><td>{ }</td><td>Все то, что включено в фигурные скобки, рассматривается как единое целое для применения символа | , .,.. или других.</td></tr>
<tr><td>[ ]</td><td>Все то, что заключено в квадратные скобки, является необязательным.</td></tr>
<tr><td></td><td>Все то, что предшествует этим символам, может повторяться произвольное число раз.</td></tr>
<tr><td>.,..</td><td>Все то, что предшествует этим символам, может повторяться произвольное число раз; каждое отдельное вхождение отделяется запятой.</td></tr>
</table>

Команда создания таблицы

```
CREATE TABLE <имя таблицы>
    ({<имя столбца> <тип данных> [<размер>]
[<тип столбца> ... ]} .,..);
    [<тип таблицы>] .,..);
```

Элементы, используемые в командах CREATE TABLE

Элемент	Определение
<имя таблицы>	Имя таблицы, которая создается по этой команде.
<имя столбца>	Имя столбца таблицы.
<тип данных>	Тип данных, которые будут содержаться в столбце. Можно использовать одно из значений: INTEGER, CHARACTER, DECIMAL, NUMERIC, SMALLINT, FLOAT, REAL, DOUBLE, PRECISION, LONG*, VARCHAR*, DATE*, TIME*. (Символом "*" помечены значения, не включенные в стандарт SQL).
<размер>	Значение этого элемента зависит от элемента <тип данных>.
<тип столбца>	Можно использовать одно из значений: NOT NULL, UNIQUE, PRIMARY KEY, CHECK(*<предикат>*), DEFAULT = *<список полей>*, REFERENCES *<имя таблицы>* [(*<имя столбца>*)].
<тип таблицы>	Можно использовать одно из значений: UNIQUE, PRIMARY KEY, CHECK(*<предикат>*), DEFAULT = *<список полей>*, REFERENCES *<имя таблицы>* [(*<имя столбца>*.,..)].

Ключевые слова SQL

Замечание: Следующие слова имеют специальное назначение в SQL, их нельзя использовать в качестве имен объектов.

Ключевые слова, которые не входят в официальный стандарт SQL, помечены символом "*".

ADA*	ADD*	ALL	ALTER*
AND	ANY	AS ASC	AUDIT*
AUTHORIZATION	AVG BEGIN	BETWEEN	BTITLE*
BY C*	CATALOG*	CHAR	CHARACTER
CHECK	CLOSE	COBOL	COLUMN*
COMMENT*	COMMIT	COMPUTE*	CONNECT*
CONTINUE	COUNT	CREATE	CURRENT
CURSOR	DATABASE*	DATE*	DBA*
DBSPACE*	DEC	DECIMAL	DECLARE
DEFAULT	DELETE	DESC	DISTINCT
DOUBLE	DROP*	END	ESCAPE
EXEC	EXISTS	FETCH	FLOAT
FOR	FOREIGN	FORMAT*	FORTRAN
FOUND	FROM	GO	GOTO
GRANT	GROUP	HAVING	IDENTIFIED*
IN	INDEX*	INDICATOR	INSERT
INT	INTEGER	INTO	IS
KEY	LANGUAGE	LIKE	LONG*
MAX	MIN	MODIFY*	MODULE
NOT	NULL	NUMERIC	OF
ON	OPEN	OPTION	OR
ORDER	PASCAL	PLI	PRECISION
PRIMARY	PRIVILEGES	PROCEDURE	PUBLIC
REAL	REFERENCES	RESOURCE*	REVOKE*
ROLLBACK	SCHEMA	SECTION	SELECT
SET	SMALLINT	SOME	SQL
SQLCODE	SQLERROR	SQLWARNING*	SUM
SYNONYM*	TABLE	TABLESPACE*	TIME*
TIMESTAMP*	TO	TTITLE*	UNION
UNIQUE	UPDATE	USER	VALUES
VARCHAR*	VIEW	WHENEVER	WHERE
WITH	WORK		

Мартин Грабер

SQL

для простых смертных

"SQL для простых смертных" — это полное введение в структурированный язык запросов, написанное специально для начинающих. Если вы не имеете опыта управления базами данных, благодаря этой книге вы научитесь работать с SQL легко и свободно, применяя простые запросы и сложные операции.

Для овладения SQL:

Усвойте смысл понятий, связанных с управлением базой данных, с помощью краткого и простого введения в реляционные базы данных.

Следуйте данным инструкциям по применению основных команд SQL для поиска размещенной в таблицах данных информации и работы с ней. Научитесь выбирать и суммировать данные, а также умело ими управлять.

Эффективно работайте с составными таблицами данных, применяя развитую технику запроса к более чем одной таблице одновременно, конструируя сложные запросы и подзапросы.

Создавайте новые таблицы данных для приложений в сфере торгового бизнеса. Изучайте важные принципы эффективного проектирования базы данных и технику обеспечения целостности и защиты данных.

Учитесь применять SQL с языками программирования, используя специальную главу для программистов.

"SQL для простых смертных" является необходимым руководством для любой реализации языка структурированных запросов, в которое влючены краткий справочник по стандартному SQL и описание общих свойств нестандартного SQL.

В этой книге представлены

- **Применение команд SQL, необходимых для управления данными**
- **Конструирование сложных запросов и подзапросов, а также создание базы данных из составных таблиц**
- **Проектирование эффективных баз данных, обеспечивающих целостность и защиту данных**

Об авторе

Мартин Грабер — писатель, преподаватель и консультант, работающий в регионе залива Сан-Франциско. Кроме работы над книгами, руководствами и документацией для пользователей он занимается широким кругом вопросов, связанных с компьютерами, а также поиском новых форм и способов применения баз данных.

Издательство
"ЛОРИ"
www.lory-press.ru