Включает полное описание синтаксиса соединений SQL

SQL
Полное руководство
Третье издание

Полное описание возможностей SQL, стандарта ANSI, вопросов применения и программирования

Включает историю, рыночные тенденции и сравнение возможностей ведущих СУБД

Обновленная информация о XML, корпоративных и специализированных базах данных (базы данных в памяти, потоковые и встраиваемые базы данных)

Джеймс Р. Грофф
Пол Н. Вайнберг
Эндрю Дж. Оппель

СКАЧАНО С WWW.SHAREWOOD.BIZ - ПРИСОЕДИНЯЙСЯ!

SQL
Полное руководство

SQL The Complete Reference

Third Edition

James Groff
Paul Weinberg
Andrew Oppel

McGraw Hill

New York Chicago San Francisco Lisbon
London Madrid Mexico City Milan New Delhi
San Juan Seoul Singapore Sydney Toronto

SQL Полное руководство

Третье издание

Джеймс Грофф
Пол Вайнберг
Эндрю Оппель

ДИАЛЕКТИКА

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Г89
УДК 681.3.07

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

Грофф, Джеймс Р., **Вайнберг**, Пол Н., **Оппель**, Эндрю Дж.

Г89 SQL: полное руководство, 3-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 960 с. : ил. — Парал. тит. англ.
ISBN 978-5-907114-26-5

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Osborne Media.

Authorized translation from the English language edition published by McGraw-Hill Companies, Copyright © 2010

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Научно-популярное издание

Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Оппель

SQL: полное руководство

3-е издание

Подписано в печать 11.06.2019. Формат 70x100/16
Гарнитура Times
Усл. печ. л. 77,4. Уч.-изд. л. 54,3
Доп. тираж 200 экз. Заказ № 4895

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

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

ISBN 978-5-907114-26-5 (рус.)
ISBN 978-0-07-159255-0 (англ.)

© ООО "Диалектика", 2019
© by The McGraw-Hill Companies, 2010

Оглавление

Благодарности 22
Введение 23

ЧАСТЬ I. ОБЗОР SQL 27
Глава 1. Введение 29
Глава 2. Краткий обзор SQL 41
Глава 3. Перспективы SQL 49
Глава 4. Реляционные базы данных 77

ЧАСТЬ II. ВЫБОРКА ДАННЫХ 95
Глава 5. Основы SQL 97
Глава 6. Простые запросы 119
Глава 7. Многотабличные запросы (соединения) 155
Глава 8. Итоговые запросы 203
Глава 9. Подзапросы и выражения с запросами 229

ЧАСТЬ III. ОБНОВЛЕНИЕ ДАННЫХ 273
Глава 10. Внесение изменений в базу данных 275
Глава 11. Целостность данных 293
Глава 12. Обработка транзакций 331

ЧАСТЬ IV. СТРУКТУРА БАЗЫ ДАННЫХ 367
Глава 13. Создание базы данных 369
Глава 14. Представления 413
Глава 15. SQL и безопасность 435
Глава 16. Системный каталог 461

ЧАСТЬ V. ПРОГРАММИРОВАНИЕ И SQL 491
Глава 17. Встроенный SQL 493
Глава 18. Динамический SQL* 543
Глава 19. SQL API 591

ЧАСТЬ VI. SQL СЕГОДНЯ И ЗАВТРА 687
Глава 20. Хранимые процедуры SQL 689
Глава 21. SQL и хранилища данных 739
Глава 22. SQL и серверы приложений 757
Глава 23. Сети и распределенные базы данных 779
Глава 24. SQL и объекты 819
Глава 25. SQL и XML 855
Глава 26. Специализированные базы данных 893
Глава 27. Будущее SQL 907

ЧАСТЬ VII. ПРИЛОЖЕНИЯ 923
Приложение А. Учебная база данных 925
Приложение Б. Производители СУБД 931
Приложение В. Синтаксис SQL 945
Предметный указатель 953

Содержание

Об авторах 21
О техническом редакторе 21
Благодарности 22
Введение 23
Как организована книга 24
Соглашения, принятые в книге 25
Для кого предназначена эта книга 25
ЧАСТЬ I. ОБЗОР SQL 27
Глава 1. Введение 29
Язык SQL 30
Роль SQL 32
Преимущества SQL 34
Независимость от конкретных СУБД 35
Межплатформенная переносимость 35
Стандарты языка SQL 35
Поддержка со стороны IBM 36
Поддержка со стороны Microsoft 36
Основанность на реляционной модели 36
Высокоуровневая структура, напоминающая естественный язык 37
Интерактивные запросы 37
Программный доступ к базе данных 37
Различные представления данных 37
Полноценный язык для работы с базами данных 37
Динамическое определение данных 38
Архитектура "клиент/сервер" 38
Поддержка приложений уровня предприятия 38
Расширяемость и поддержка объектно-ориентированных технологий 39
Возможность доступа к данным в Интернете 39
Интеграция с языком Java (протокол JDBC) 39
Поддержка открытого кода 40
Промышленная инфраструктура 40

Глава 2. Краткий обзор SQL **41**
Простая база данных 41
Выборка данных 42
Получение итоговых данных 44
Добавление данных 45
Удаление данных 46
Обновление данных 46
Защита данных 46
Создание базы данных 47
Резюме 48
Глава 3. Перспективы SQL **49**
SQL и эволюция управления базами данных 49
Краткая история SQL 50
Первые годы 52
Первые реляционные СУБД 52
Продукты IBM 53
Коммерческое признание 54
Стандарты SQL 56
Стандарты ANSI/ISO 56
Другие ранние стандарты SQL 59
ODBC и консорциум SQL Access Group 59
JDBC и серверы приложений 60
SQL и переносимость 61
SQL и сети 63
Централизованная архитектура 63
Архитектура файлового сервера 64
Архитектура "клиент/сервер" 64
Многоуровневая архитектура 66
Влияние SQL 67
SQL и мэйнфреймы 68
SQL и мини-компьютеры 68
SQL и UNIX 68
SQL и персональные компьютеры 69
SQL и обработка транзакций 70
SQL и базы данных для рабочих групп 71
SQL, хранилища данных и интеллектуальные ресурсы предприятия 72
SQL и интернет-приложения 74
Резюме 75
Глава 4. Реляционные базы данных **77**
Ранние модели данных 77
Системы управления файлами 77

Иерархические базы данных 79
Сетевые базы данных 81
Реляционная модель данных 83
Учебная база данных 84
Таблицы 85
Первичные ключи 86
Взаимоотношения 88
Внешние ключи 89
Двенадцать правил Кодда для реляционных баз данных* 90
Резюме 93

ЧАСТЬ II. ВЫБОРКА ДАННЫХ 95

Глава 5. Основы SQL 97

Инструкции 97
Имена 103
Имена таблиц 104
Имена столбцов 105
Типы данных 105
Константы 111
Числовые константы 111
Строковые константы 112
Константы даты и времени 112
Символьные константы 113
Выражения 114
Встроенные функции 115
Отсутствующие данные (значения NULL) 117
Резюме 118

Глава 6. Простые запросы 119

Инструкция SELECT 119
Предложение SELECT 121
Предложение FROM 122
Результаты запроса 122
Простые запросы 125
Вычисляемые столбцы 125
Выборка всех столбцов (SELECT *) 128
Повторяющиеся строки (DISTINCT) 129
Отбор строк (WHERE) 130
Условия отбора 132
Сравнение (=, <>, <, <=, >, >=) 132
Проверка на принадлежность диапазону (BETWEEN) 135
Проверка наличия во множестве (IN) 137
Проверка на соответствие шаблону (LIKE) 139

Проверка на равенство NULL (IS NULL) 141
Составные условия отбора (AND, OR и NOT) 142
Сортировка результатов запроса (ORDER BY) 145
Правила выполнения однотабличных запросов 148
Объединение результатов нескольких запросов (UNION)* 149
Объединение и повторяющиеся строки* 151
Объединение и сортировка* 152
Вложенные объединения* 152
Резюме 154

Глава 7. Многотабличные запросы (соединения) 155

Пример двухтабличного запроса 155
Простое соединение таблиц 158
Запросы с использованием отношения "предок-потомок" 159
Еще один способ определения соединений 161
Соединения с условиями отбора строк 162
Несколько связанных столбцов 163
Естественные соединения 164
Запросы к трем и более таблицам 165
Прочие соединения по равенству 168
Соединение по неравенству 170
Особенности многотабличных запросов 171
Квалифицированные имена столбцов 171
Выборка всех столбцов 173
Самосоединения 173
Псевдонимы таблиц 176
Производительность при обработке многотабличных запросов 177
Внутренняя структура соединения таблиц 179
Умножение таблиц 179
Правила выполнения многотабличных запросов на выборку 180
Внешние соединения 182
Левое и правое внешние соединения 185
Старая запись внешнего соединения * 188
Соединения и стандарт SQL 190
Внутренние соединения в стандарте SQL 191
Внешние соединения в стандарте SQL* 192
Перекрестные соединения в стандарте SQL* 193
Многотабличные соединения в стандарте SQL 196
Резюме 203

Глава 8. Итоговые запросы 204

Агрегирующие функции 204
Вычисление суммы значений столбца 206
Вычисление среднего значений столбца 207

Вычисление предельных значений 208
Подсчет количества данных 209
Статистические функции в списке возвращаемых столбцов 210
Статистические функции и значения NULL 213
Удаление повторяющихся строк (DISTINCT) 215
Запросы с группировкой (GROUP BY) 215
Несколько столбцов группировки 219
Ограничения на запросы с группировкой 221
Значения NULL в столбцах группировки 223
Условия отбора групп (HAVING) 224
Ограничения на условия отбора групп 227
Значения NULL и условия отбора групп 228
Предложение HAVING без GROUP BY 228
Резюме 229

Глава 9. Подзапросы и выражения с запросами 230

Применение подзапросов 230
Что такое подзапрос 231
Подзапросы в предложении WHERE 233
Внешние ссылки 235
Условия отбора в подзапросе 235
Сравнение с результатом подзапроса (=, <>, <, <=, >, >=) 236
Проверка на принадлежность результатам подзапроса (IN) 238
Проверка существования (EXISTS) 239
Многократное сравнение (предикаты ANY и ALL)* 242
Подзапросы и соединения 246
Вложенные подзапросы 248
Коррелированные подзапросы* 249
Подзапросы в предложении HAVING* 252
Резюме по подзапросам 254
Сложные запросы* 255
Выражения со скалярными значениями 257
Выражения со строками таблиц 263
Табличные выражения 266
Выражения запросов 270
Резюме по SQL-запросам 273

ЧАСТЬ III. ОБНОВЛЕНИЕ ДАННЫХ 274

Глава 10. Внесение изменений в базу данных 276

Добавление новых данных 277
Однострочная инструкция INSERT 277
Многострочная инструкция INSERT 281
Программы пакетной загрузки 284

Удаление существующих данных 285
Инструкция DELETE 285
Удаление всех строк 287
Инструкция DELETE с подзапросом* 287
Обновление существующих данных 289
Инструкция UPDATE 289
Обновление всех строк 292
Инструкция UPDATE с подзапросом* 292
Резюме 293

Глава 11. Целостность данных 294

Условия целостности данных 294
Обязательность данных 296
Условия на значения 297
Ограничения на значения столбца 298
Домены 299
Целостность таблицы 300
Прочие условия уникальности столбцов 301
Уникальность и значения NULL 301
Ссылочная целостность 302
Проблемы, связанные со ссылочной целостностью 304
Правила удаления и обновления* 306
Каскадные удаления и обновления* 310
Ссылочные циклы* 313
Внешние ключи и значения NULL* 316
Расширенные возможности ограничений 318
Утверждения 319
Типы ограничений SQL 320
Отложенная проверка ограничений 321
Бизнес-правила 324
Что такое триггер 325
Триггеры и ссылочная целостность 327
Преимущества и недостатки триггеров 328
Триггеры и стандарты SQL 329
Резюме 329

Глава 12. Обработка транзакций 332

Что такое транзакция 332
Модель транзакции ANSI/ISO SQL 335
Инструкции START TRANSACTION и SET TRANSACTION 336
Инструкции SAVEPOINT и RELEASE SAVEPOINT 337
Инструкции COMMIT и ROLLBACK 338
Транзакции: что за сценой* 340
Транзакции и работа в многопользовательском режиме 342

Проблема пропавшего обновления 342
Проблема промежуточных данных 343
Проблема несогласованных данных 345
Проблема строк-призраков 346
Параллельные транзакции 347
Блокировка* 349
Уровни блокировки 350
Блокировка с обеспечением совместного доступа и исключающая блокировка 352
Усовершенствованные методы блокировки* 355
Управление версиями* 360
Управление версиями в действии* 361
Преимущества и недостатки управления версиями* 364
Резюме 365

ЧАСТЬ IV. СТРУКТУРА БАЗЫ ДАННЫХ 367

Глава 13. Создание базы данных 369

Язык определения данных 369
Создание базы данных 371
Определения таблиц 372
Создание таблицы (CREATE TABLE) 373
Удаление таблицы (DROP TABLE) 383
Изменение определения таблицы (ALTER TABLE) 384
Определения ограничений 387
Утверждения 388
Домены 388
Псевдонимы, или синонимы (CREATE/DROP ALIAS) 389
Индексы (CREATE/DROP INDEX) 391
Управление другими объектами базы данных 395
Структура базы данных 398
Архитектура с одной базой данных 399
Архитектура с несколькими базами данных 400
Архитектура с каталогами 402
Базы данных на нескольких серверах 404
Структура базы данных и стандарт ANSI/ISO 404
Каталоги 407
Схемы 407
Резюме 411

Глава 14. Представления 413

Что такое представление 413
Как СУБД работает с представлениями 415
Преимущества представлений 416

Недостатки представлений 416
Создание представлений (CREATE VIEW) 417
Горизонтальные представления 418
Вертикальные представления 419
Смешанные представления 421
Сгруппированные представления 421
Соединенные представления 423
Обновление представлений 425
Обновление представлений и стандарт ANSI/ISO 426
Обновление представлений в коммерческих СУБД 427
Контроль над обновлением представлений (CHECK OPTION) 428
Удаление представления (DROP VIEW) 430
Материализованные представления* 431
Резюме 433

Глава 15. SQL и безопасность 435

Принципы защиты данных, применяемые в SQL 435
Идентификаторы пользователей 437
Защищаемые объекты 441
Привилегии 442
Представления и безопасность SQL 445
Предоставление привилегий (GRANT) 448
Привилегии для работы со столбцами 449
Передача привилегий (GRANT OPTION) 450
Отмена привилегий (REVOKE) 452
REVOKE и GRANT OPTIONS 455
REVOKE и стандарт ANSI/ISO 457
Безопасность на основе ролей 458
Резюме 460

Глава 16. Системный каталог 461

Что такое системный каталог 461
Системный каталог и средства формирования запросов 462
Системный каталог и стандарт ANSI/ISO 463
Содержимое системного каталога 464
Информация о таблицах 465
Информация о столбцах 470
Информация о представлениях 473
Примечания 475
Информация об отношениях между таблицами 476
Информация о пользователях 478
Информация о привилегиях 480
Информационная схема SQL 481
Прочая информация каталога 488
Резюме 488

ЧАСТЬ V. ПРОГРАММИРОВАНИЕ И SQL 491

Глава 17. Встроенный SQL 493

Методы программного SQL 493
Обработка инструкций в СУБД 495
Основные концепции встроенного SQL 497
Разработка программы со встроенным SQL 499
Выполнение программы со встроенным SQL 502
Простые инструкции встроенного SQL 504
Объявления таблиц 507
Обработка ошибок 507
Использование базовых переменных 515
Выборка данных с помощью встроенного SQL 521
Запросы, возвращающие одну запись 522
Многострочные запросы 528
Удаление и обновление данных на основе курсоров 536
Курсоры и обработка транзакций 540
Резюме 542

Глава 18. Динамический SQL* 543

Недостатки статического SQL 543
Концепции динамического SQL 545
Динамическое выполнение инструкций (EXECUTE IMMEDIATE) 547
Динамическое выполнение в два этапа 549
Инструкция PREPARE 552
Инструкция EXECUTE 553
Динамические запросы 560
Инструкция DESCRIBE 565
Инструкция DECLARE CURSOR 567
Динамическая инструкция OPEN 568
Динамическая инструкция FETCH 570
Динамическая инструкция CLOSE 571
Диалекты динамического SQL 572
Динамический SQL в Oracle* 572
Динамический SQL и стандарт SQL 576
Базовые динамические инструкции SQL 577
Стандартная SQLDA 579
Стандарт SQL и динамические запросы на выборку 584
Резюме 588

Глава 19. SQL API 591

Концепции API 592
dblib API (SQL Server) 594
Основы работы с SQL Server 595

Запросы на выборку в SQL Server 602
Позиционные обновления 609
Динамические запросы на выборку 610
ODBC API и стандарт SQL/CLI 617
Стандартизация CLI 617
Структуры CLI 622
Обработка инструкций в CLI 626
Ошибки CLI и диагностическая информация 644
Атрибуты CLI 646
Информационные функции CLI 647
ODBC API 648
Структура ODBC 649
ODBC и независимость от СУБД 650
Функции ODBC для работы с системными каталогами 651
Расширенные возможности ODBC 652
Oracle Call Interface (OCI) 656
Дескрипторы OCI 657
Подключение к серверу Oracle 659
Выполнение инструкций 660
Обработка результатов запроса 661
Управление описателями 661
Управление транзакциями 661
Обработка ошибок 662
Получение информации из системного каталога 662
Работа с большими объектами 662
Java Database Connectivity (JDBC) 663
История и версии JDBC 664
Реализация JDBC и типы драйверов 665
JDBC API 669
Базовая обработка инструкций в JDBC 671
Обработка простых запросов 673
Использование подготовленных инструкций в JDBC 676
Использование вызываемых инструкций в JDBC 678
Обработка ошибок в JDBC 681
Курсоры произвольного доступа в JDBC 682
Получение метаданных в JDBC 683
Расширенные возможности JDBC 685
Резюме 686

ЧАСТЬ VI. SQL СЕГОДНЯ И ЗАВТРА 687

Глава 20. Хранимые процедуры SQL 689

Концепции хранимых процедур 690
Простейший пример 692

Использование хранимых процедур 693
Создание хранимой процедуры 694
Вызов хранимой процедуры 696
Переменные хранимых процедур 697
Блоки инструкций 700
Функции 701
Возврат значений через параметры 703
Условное выполнение 705
Циклы 707
Другие управляющие конструкции 709
Циклы с курсорами 710
Обработка ошибок 713
Преимущества хранимых процедур 715
Производительность хранимых процедур 716
Системные хранимые процедуры 717
Внешние хранимые процедуры 718
Триггеры 719
Преимущества и недостатки триггеров 720
Триггеры в диалекте Transact-SQL 720
Триггеры в диалекте Informix 722
Триггеры в диалекте Oracle PL/SQL 724
Дополнительные вопросы, связанные с использованием триггеров 726
Хранимые процедуры и стандарт SQL 726
Стандарт SQL/PSM для хранимых процедур 727
Стандарт SQL/PSM для триггеров 736
Резюме 737

Глава 21. SQL и хранилища данных 739

Концепции хранилищ данных 740
Компоненты хранилища данных 742
Эволюция хранилищ данных 743
Архитектура баз данных для хранилищ 744
Кубы фактов 744
Схема звезды 746
Многоуровневые измерения 748
Расширения SQL для хранилищ данных 750
Производительность хранилищ данных 751
Скорость загрузки данных 751
Производительность запросов 753
Резюме 754

Глава 22. SQL и серверы приложений 757

SQL и веб-сайты: ранние реализации 757
Серверы приложений и трехуровневые архитектуры веб-сайтов 759

Доступ серверов приложений к базам данных 761
Типы EJB 762
Доступ к базе данных со стороны session bean 763
Доступ к базе данных со стороны entity bean 766
Усовершенствования EJB 2.0 770
Усовершенствования EJB 3.0 771
Разработка приложений с открытым кодом 773
Серверы приложений и кеширование 773
Резюме 776

Глава 23. Сети и распределенные базы данных 779

Проблемы управления распределенными данными 780
Практические подходы к управлению распределенными базами данных 785
Доступ к удаленным базам данных 786
Прозрачность доступа к удаленным данным 789
Дублирование таблиц 791
Репликация таблиц 793
Двунаправленная репликация 795
Затраты на репликацию 797
Типичные схемы репликации 798
Доступ к распределенным базам данных 801
Удаленные запросы 802
Удаленные транзакции 803
Распределенные транзакции 804
Распределенные запросы 805
Протокол двухфазного завершения транзакций* 807
Сетевые приложения и архитектура баз данных 810
Приложения "клиент/сервер" и архитектура баз данных 811
Приложения "клиент/сервер" с хранимыми процедурами 812
Корпоративные приложения и кеширование данных 813
Управление базами данных в Интернете 815
Резюме 817

Глава 24. SQL и объекты 819

Объектно-ориентированные базы данных 820
Характеристики объектно-ориентированной базы данных 820
"Плюсы" и "минусы" объектно-ориентированных баз данных 822
Влияние объектных технологий на рынок баз данных 823
Объектно-реляционные базы данных 824
Поддержка больших объектов 825
Большие объекты в реляционной модели 826
Специализированная обработка больших объектов 827
Абстрактные (структурированные) типы данных 830

Определение абстрактных типов данных 832
Использование абстрактных типов данных 834
Наследование 835
Табличное наследование: реализация классов 837
Множества, массивы и коллекции 840
Определение коллекций 841
Коллекции и запросы на выборку 845
Работа с коллекциями данных 846
Коллекции и хранимые процедуры 847
Пользовательские типы данных 849
Методы и хранимые процедуры 850
Поддержка объектов в стандарте SQL 853
Резюме 854

Глава 25. SQL и XML 855

Что такое XML 855
Азы XML 857
XML для данных 859
XML и SQL 860
Элементы и атрибуты 862
Использование XML с базами данных 864
Вывод XML 865
Ввод XML 869
Обмен XML-данными 871
Хранение и интеграция XML-данных 871
XML и метаданные 876
DTD 877
XML Schema 879
XML и запросы 885
Концепции XQuery 886
Обработка запросов в XQuery 888
Базы данных на основе XML 890
Резюме 891

Глава 26. Специализированные базы данных 893

Низкие задержки и базы данных в памяти 893
Анатомия баз данных в памяти 895
Реализация баз данных в памяти 897
Кеширование с базами данных в памяти 897
Сложные базы данных для обработки событий
и потоковые базы данных 898
Непрерывные запросы в потоковых базах данных 900
Реализации потоковых баз данных 901
Компоненты потоковых баз данных 901

Встраиваемые базы данных 903
Характеристики встраиваемых баз данных 903
Реализации встраиваемых баз данных 904
Мобильные базы данных 904
Роли мобильных баз данных 905
Реализации мобильных баз данных 905
Резюме 906

Глава 27. Будущее SQL 907

Тенденции на рынке баз данных 908
Насыщение рынка корпоративных баз данных 908
Сегментация рынка СУБД 909
Пакеты корпоративных приложений 910
Программное обеспечение в виде служб 911
Повышение производительности аппаратного обеспечения 912
Специализированные серверы баз данных 913
Стандартизация SQL 914
SQL в следующем десятилетии 915
Распределенные базы данных 915
Массивные хранилища данных для оптимизации бизнеса 916
Сверхпроизводительные базы данных 916
Интеграция Интернета и сетевых служб 917
Встраиваемые базы данных 918
Интеграция с объектно-ориентированными технологиями 919
Горизонтально масштабируемые базы данных 920
Резюме 921

ЧАСТЬ VII. ПРИЛОЖЕНИЯ 923

Приложение А. Учебная база данных 925

Приложение Б. Производители СУБД 931

Приложение В. Синтаксис SQL 945

Инструкции DDL 946
Инструкции управления доступом 947
Основные инструкции DML 948
Инструкции обработки транзакций 948
Инструкции для работы с курсорами 948
Выражения запросов 949
Условия отбора 951
Выражения 951
Элементы инструкций 952
Простые элементы 952

Предметный указатель 953

Об авторах

Джеймс Р. Грофф (James R. Groff) является исполнительным директором компании PBworks. Ранее Грофф занимал ту же должность в компании TimesTen, ведущем производителе баз данных с хранением информации в оперативной памяти. Он возглавлял TimesTen с первых дней ее основания в течение восьми лет, до тех пор пока компания не была в 2005 году приобретена корпорацией Oracle, где после этого он работал в качестве старшего вице-президента, и Oracle TimesTen стала флагманской базой данных реального времени Oracle. Грофф вместе с Полем Вайнбергом (Paul Weinberg) был соучредителем Network Innovations Corporation, разработчика сетевого программного обеспечения на базе SQL. Кроме данной книги, ими в соавторстве написана книга *Understanding UNIX: A Conceptual Guide*. Грофф занимал также высокие должности в Apple Computer и Hewlett-Packard. Он имеет степень бакалавра математики Массачусетсского технологического института и степень магистра делового администрирования Гарвардского университета.

Пол Н. Вайнберг (Paul N. Weinberg) — старший вице-президент компании SAP. До того как занять эту должность в SAP, Вайнберг был президентом A2i, Inc., которая была приобретена SAP в 2004 году. Вместе с Гроффом он был соучредителем компании Network Innovations Corporation, пионера в области баз данных "клиент/сервер", которая в 1988 году была приобретена Apple Computer. Кроме данной книги, ими в соавторстве написана книга *Understanding UNIX: A Conceptual Guide*. Вайнберг работал также в Bell Laboratories, Hewlett-Packard и Plexus Computers. В 1981 году был соавтором бестселлера года — книги *The Simple Solution to Rubik's Cube* тиражом свыше 6 млн экземпляров. Пол имеет степень бакалавра Мичиганского университета и магистра Станфордского университета — обе в области теории вычислительных машин и систем.

Эндрю Дж. Оппель (Andrew J. (Andy) Oppel) — ведущий специалист по моделированию данных в Blue Shield. Кроме того, он уже более 20 лет преподает теорию баз данных в Калифорнийском университете в Беркли. Им разработаны и реализованы сотни баз данных для широкого диапазона приложений, включая медицину, банковское дело, страхование, телекоммуникации и многое другое. Он автор книг *Databases Demystified*, *SQL Demystified* и *Databases: A Beginner's Guide*, а также соавтор книги *SQL: A Beginner's Guide*. Эндрю имеет степень бакалавра в области теории вычислительных машин и систем Трансильванского университета (штат Кентукки).

О техническом редакторе

Аарон Давенпорт (Aaron Davenport) работает с реляционными технологиями более десяти лет. В настоящее время он является ведущим сотрудником LCS Technologies, Inc., фирмы, специализирующейся на консультациях по настройке производительности баз данных, разработке приложений и архитектуре баз данных. До этого он работал в Yahoo!, Gap Inc. и Blue Shield.

Благодарности

Особая благодарность Энди Оппелю, нашему новому соавтору третьего издания книги *SQL: полное руководство*. Его знание материала помогло сделать эту книгу еще лучше, и мы счастливы, что нам удалось привлечь его к этой работе.

Джим и Пол

Я горжусь тем, что меня пригласили в команду авторов третьего издания книги *SQL: полное руководство*. Я благодарен команде издательства McGraw-Hill за ее поддержку всех наших усилий. В особенности я благодарен техническому редактору Аарону Давенпорту за его постоянное внимание к деталям, которые так много значат для качества книги.

Энди

Введение

В третьем издании книги *SQL: полное руководство* содержится исчерпывающее, глубокое и детальное описание языка SQL. Предназначена она как для пользователей, программистов и специалистов в области обработки данных, так и для менеджеров, которые хотят узнать, какое влияние оказывает SQL на компьютерный рынок. В книге рассматриваются основные понятия, необходимые для изучения SQL и применения его на практике, описываются история развития и стандарты этого языка, а также рассказывается о роли, которую играет SQL в разных сегментах компьютерной индустрии — от корпоративной обработки данных до хранилищ данных и архитектуры веб-сайтов. В этом новом издании имеются главы, рассматривающие роль SQL в архитектуре серверов приложений, а также интеграцию SQL с XML и объектно-ориентированными технологиями.

Свойства и особенности SQL раскрываются постепенно, шаг за шагом; изложение сопровождается многочисленными иллюстрациями и реальными примерами, облегчающими усвоение материала. Кроме того, в книге проводится сравнение различных СУБД от ведущих поставщиков и оцениваются их преимущества и недостатки; это поможет вам выбрать для своего приложения наиболее подходящую СУБД.

Большинство примеров в книге использует учебную базу данных, описанную в приложении А, "Учебная база данных". Эта учебная база данных содержит информацию для поддержки простого приложения обработки заказов небольшой торговой компании. В приложении А, "Учебная база данных", содержатся также инструкции по получению SQL-текстов для создания и заполнения учебной базы данных в различных СУБД — Oracle, SQL Server, MySQL и DB2. Это позволит вам самостоятельно повторить все примеры из книги и получить опыт написания и выполнения инструкций SQL.

В некоторых главах материал, в зависимости от сложности, разбит на две части: вначале раскрывается основная тема главы, а затем рассматриваются некоторые "секреты" работы SQL, представляющие интерес для профессионалов. Разделы, в которых излагается информация повышенной сложности, помечены звездочкой (*). Для того чтобы освоить практическую работу с SQL, эти разделы читать не обязательно.

Как организована книга

Данная книга состоит из шести частей, в каждой из которых раскрываются определенные аспекты языка SQL.

- Часть I, "Обзор SQL". В этой части содержится введение в язык SQL и рассказывается об истории его применения в качестве языка для работы с базами данных. Четыре главы посвящены истории SQL, эволюции его стандартов и описанию связи SQL с реляционной моделью данных, а также с более ранними технологиями построения баз данных. Кроме того, здесь дан краткий обзор всего языка и проиллюстрированы наиболее важные его особенности.
- Часть II, "Выборка данных". Здесь рассказывается о возможностях SQL по выполнению запросов на выборку информации из базы данных. В первой главе этой части описывается общая структура языка SQL. В четырех последующих главах рассматриваются запросы на выборку, начиная с самых простых и заканчивая более сложными — запросами к нескольким таблицам, — итоговыми и подчиненными.
- Часть III, "Обновление данных". В этой части объясняется, как с помощью SQL добавлять в базу новые данные, а также удалять устаревшие и модифицировать уже имеющиеся данные. Кроме того, здесь рассматривается проблема целостности базы данных, возникающая при изменении данных, и способы решения этой проблемы с помощью SQL. В последней из трех глав этой части раскрывается понятие транзакции и рассказывается об обработке транзакций в многопользовательской среде.
- Часть IV, "Структура базы данных". Рассказывается о том, как с помощью SQL можно создавать и изменять структуру базы данных. В четырех главах этой части объясняется, как создавать таблицы, представления и индексы, которые образуют структуру реляционной базы данных. Рассматриваются схема безопасности баз данных, предотвращающая несанкционированный доступ к данным, и системный каталог SQL, содержащий описание структуры базы данных. Кроме того, здесь обсуждаются существенные различия, имеющиеся в структурах баз данных, поддерживаемых различными СУБД.
- Часть V, "Программирование и SQL". В этой части рассказывается об использовании SQL в приложениях, предназначенных для работы с базами данных. Здесь обсуждается встроенный SQL, соответствующий стандарту ANSI/ISO и применяемый в большинстве СУБД ведущих фирм (таких, как IBM, Oracle, Informix и др.). Кроме того, описывается динамический SQL, который используется для создания приложений общего назначения, таких как генераторы отчетов и программы просмотра баз данных. Наконец, в этой части описываются популярные интерфейсы программирования SQL-приложений, такие как ODBC, стандартный интерфейс

с языком программирования Java JDBC, а также интерфейсы разных производителей, в частности Oracle OCI API.

- Часть VI, "SQL сегодня и завтра". В этой части рассматриваются нынешнее состояние реляционных СУБД и тенденции их развития. В двух главах описывается применение хранимых процедур и триггеров SQL для оперативной обработки транзакций, а также применение SQL для хранилищ данных. Четыре дополнительные главы посвящены распределенным базам данных, влиянию на SQL объектных технологий, специализированным базам данных и интеграции технологий SQL и XML. В последней главе рассматривается будущее SQL и некоторые из наиболее важных тенденций в управлении данными на базе SQL.

Соглашения, принятые в книге

В книге описываются возможности SQL, реализованные в наиболее распространенных СУБД и определенные в стандарте ANSI/ISO для SQL. Как правило, синтаксис инструкций SQL, приводимых в книге и используемых в примерах, применим ко всем диалектам языка. Если же в диалектах имеются отличия, о них рассказывается в тексте, а в примерах используется наиболее распространенный вариант. В таких случаях может возникнуть необходимость слегка изменить инструкции SQL для того, чтобы их можно было применить в конкретной СУБД.

При первом упоминании или определении технического термина он выделяется *курсивом*. Элементы языка SQL, включая ключевые слова, названия столбцов и таблиц, а также примеры инструкций SQL напечатаны `ПРОПИСНЫМИ БУКВАМИ МОНОШИРИННЫМ ШРИФТОМ`. Имена API-функций SQL напечатаны `строчными буквами моноширинным шрифтом`. В примерах программ используются соглашения конкретного языка программирования (прописные буквы для программ на языках COBOL и FORTRAN и строчные для программ на C). Отметим, что указанные соглашения используются исключительно для улучшения восприятия текста; почти во всех реализациях SQL разрешается набирать инструкции как прописными, так и строчными буквами. В большинстве примеров сразу же за инструкцией приводятся результаты запросов, как это обычно бывает при интерактивной работе с SQL. Если результат запроса является слишком длинным, то приводится лишь несколько первых строк, а вместо остальных строк ставится многоточие.

Для кого предназначена эта книга

Эта книга будет полезной всем, кто хочет ознакомиться с SQL и изучить его: пользователям баз данных, специалистам по обработке данных, программистам, студентам и руководителям. Материал изложен простым и доступным языком; большое количество иллюстраций и примеров помогают понять суть языка SQL и особенности его применения на практике. В данной книге не делается акцент на каком-то одном варианте или диалекте SQL, а рассматривается стандартный вариант языка. И только по мере необходимости описываются различия между наиболее

популярными СУБД, включая Oracle, Microsoft SQL Server, IBM DB2 и Informix, Sybase и MySQL. В книге также рассказывается о важности стандартов на базе SQL, таких как ODBC и JDBC, а также стандартов ANSI/ISO для SQL и связанных технологий. Третье издание книги содержит новые главы и разделы с описанием последних новинок, таких как объектно-реляционные технологии, XML и архитектуры серверов приложений.

Новичок в SQL найдет в этой книге исчерпывающее описание языка, начинающееся с простых запросов и постепенно раскрывающее все более сложные понятия. Структура книги позволяет любому читателю, имеющему основные представления о базах данных, быстро начать работать с SQL и изучить все его особенности. Книга полезна и при изучении более сложных возможностей языка. Вы можете создать учебную базу данных при помощи сценариев SQL, доступных на веб-сайте McGraw-Hill (см. приложение А, "Учебная база данных"), и использовать ее для практической работы с книгой.

Для руководителей и специалистов по обработке данных книга полезна тем, что освещает влияние SQL на все сегменты компьютерного рынка, от систем оперативной обработки транзакций до хранилищ данных, веб-сайтов и распределенных баз данных. В первых главах рассказывается об истории SQL, его положении на рынке и эволюции со времени появления первых баз данных. В заключительных главах рассматриваются перспективы развития SQL и новых технологий обработки данных, включая распределенные базы данных, объектно-ориентированные расширения SQL, а также интеграция языка XML.

Программисту эта книга предлагает полный курс обучения программированию на SQL. В отличие от справочных руководств по различным СУБД, в данной книге излагается общая концепция программирования на SQL, объясняется, зачем и как разрабатываются приложения, в которых используется SQL. Кроме того, здесь сравниваются программные интерфейсы SQL, используемые во всех основных СУБД, включая встраиваемый SQL, динамический SQL, ODBC, JDBC, а также API конкретных СУБД, например Oracle Call Interface. Такого описания и сравнения вы не найдете ни в одной другой книге.

Тот, кому необходимо выбрать СУБД, найдет здесь подробное описание особенностей и преимуществ диалектов SQL, предлагаемых различными поставщиками СУБД. Различия между ведущими СУБД рассматриваются не только с технической стороны, но и с точки зрения их конкурентоспособности на компьютерном рынке. Учебная база данных может послужить для испытания конкретных СУБД на пригодность для ваших целей.

Короче говоря, пользу от этой книги получит любой — и технический специалист, и тот, кто далек от техники. Это наиболее полный источник информации об языке SQL, его возможностях и достоинствах, популярных продуктах, истории — словом, обо всем на свете, что хоть как-то связано с SQL.

ЧАСТЬ I

Обзор SQL

Глава 1
Введение

Глава 2
Краткий обзор SQL

Глава 3
Перспективы SQL

Глава 4
Реляционные базы данных

Первые четыре главы данной книги являются введением в SQL. В главе 1, "Введение", описывается, что такое язык SQL, и раскрываются его основные возможности и преимущества. Глава 2, "Краткий обзор SQL", бегло ознакомит вас со многими возможностями SQL с помощью простых примеров. В главе 3, "Перспективы SQL", вы ознакомитесь с историей SQL, его стандартами и основными производителями продуктов на базе SQL, а также с перспективами его развития. В главе 4, "Реляционные базы данных", описывается реляционная модель данных, на которой основан SQL, и выполняется ее сравнение с более ранними моделями данных.

ГЛАВА 1

Введение

Язык SQL и СУРБД, основанные на нем, составляют одну из наиболее важных базовых технологий в информатике. В течение трех последних десятилетий SQL вырос от первого робкого коммерческого применения до целого сегмента рынка, стоящего десятки миллиардов долларов, и стал *стандартом* языка баз данных. Его поддерживают сотни баз данных, работающих на различных платформах, — от мэйнфреймов до персональных компьютеров. Базы данных с поддержкой SQL могут даже быть встроены в ваш мобильный телефон или КПК или, например, в ваш автомобиль. Официальный международный стандарт SQL принимался и расширялся несколько раз. По сути, любой крупный программный продукт для работы с данными использует SQL, и именно SQL представляет собой ядро флагманских баз данных Microsoft, Oracle и IBM, трех крупнейших программных компаний в мире. Кроме того, SQL является сердцем СУРБД с открытым кодом, таких так MySQL и Postgres, способствующих распространению популярности Linux и движения в поддержку открытого кода. Изначально SQL представлял собой скромный исследовательский проект IBM, но со временем он стал широко известен и в качестве важной компьютерной технологии, и в качестве мощного рыночного фактора.

Но, собственно говоря, что такое SQL? Почему он так важен? Что он позволяет делать и как он работает? Если SQL — настоящий стандарт, то почему существует так много различных версий и диалектов? Чем отличаются друг от друга такие популярные СУБД, как SQL Server, Oracle, MySQL, Sybase и DB2? Как SQL соотносится со стандартами Microsoft, такими как ODBC и .NET? Как JDBC связывает SQL с миром Java и объектной технологией? Какую роль он играет в сервис-ориентированной архитектуре (Service-Oriented Architecture, SOA) и веб-службах крупных ИТ-организаций? Действительно ли SQL может работать и на мэйнфреймах, и в наладонниках? Обеспечивает ли он производительность, достаточную для обработки транзакций большого объема? Как SQL влияет на применение компьютеров и как получить максимальную пользу от этого важного инструмента

СКАЧАНО С САЙТА WWW.SHKOLNIK.BY ДЛЯ ОБЩЕНИЯ ШКОЛЬНИКОВ

для работы с данными? Книга, которую вы держите в руках, отвечает на все эти вопросы и обеспечивает читателя солидным багажом знаний о SQL.

Язык SQL

SQL является инструментом, предназначенным для организации, управления, выборки и обработки информации, содержащейся в базе данных. Изначально в IBM этому языку было дано имя *Structured English Query Language*, сокращенно *SEQUEL*. Однако в тот момент владельцем торговой марки SEQUEL была компания Hawker Siddeley Aircraft Company из Великобритании, так что название языка было сокращено до SQL (Structured Query Language, *язык структурированных запросов*), а слово English было удалено из названия (для соответствия названия аббревиатуре). В настоящее время аббревиатура SQL читается либо как "сиквел", либо как "эс-кю-эль", причем корректны оба произношения, хотя предпочтительно второе. Как следует из названия, SQL является *языком программирования*, который применяется для организации взаимодействия пользователя с базой данных. На самом деле SQL работает только с базами данных определенного типа, называемыми *реляционными базами данных*, которые представляют собой основной способ организации данных в широком диапазоне приложений.

На рис. 1.1 показана схема работы SQL. Согласно этой схеме, в вычислительной системе имеется *база данных*, в которой хранится важная информация. Если вычислительная система относится к сфере бизнеса, то в базе данных могут содержаться сведения о материальных ценностях, выпускаемой продукции, объемах продаж и зарплате. В базе данных на персональном компьютере может находиться информация о выписанных чеках, телефонах и адресах или информация, извлеченная из более крупной вычислительной системы. Компьютерная программа, которая управляет базой данных, называется *системой управления базой данных*, или СУБД.

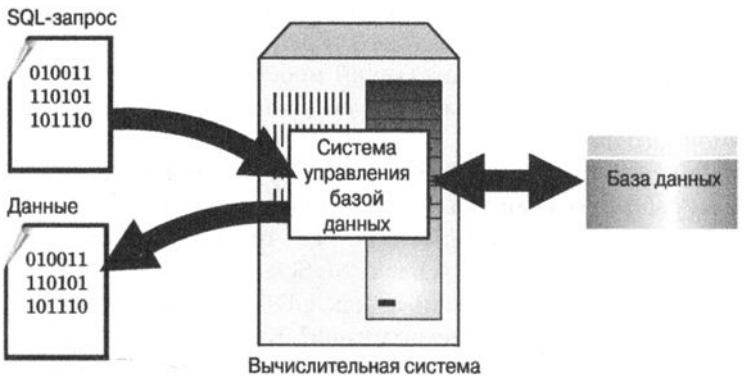


Рис. 1.1. Применение SQL для обращения к базе данных

Для того чтобы получить информацию из базы данных, вы запрашиваете ее у СУБД с помощью SQL. СУБД обрабатывает запрос, находит требуемые данные и возвращает их вам. Процесс запроса данных у базы данных и получения резуль-

тата называется *запросом* к базе данных (вот почему в названии языка имеется слово *"запрос"* — язык структурированных *запросов*).

Однако это название не совсем соответствует действительности. Во-первых, сегодня SQL представляет собой нечто большее, чем просто инструмент создания запросов, хотя именно для этого он и был первоначально разработан. Несмотря на то что выборка данных по-прежнему остается одной из наиболее важных функций SQL, сейчас этот язык используется для реализации всех функциональных возможностей, которые СУБД предоставляет пользователю.

- **Определение данных**. SQL позволяет пользователю определить структуру и организацию хранимых данных и взаимоотношения между элементами сохраненных данных.
- **Выборка данных**. SQL дает пользователю или приложению возможность извлекать из базы содержащиеся в ней данные и пользоваться ими.
- **Обработка данных**. SQL позволяет пользователю или приложению изменять базу данных, т.е. добавлять в нее новые данные, а также удалять или обновлять уже имеющиеся в ней данные.
- **Управление доступом**. С помощью SQL можно ограничить возможности пользователя по выборке, добавлению и изменению данных и защитить их от несанкционированного доступа.
- **Совместное использование данных**. SQL применяется для координации совместного использования данных пользователями, работающими одновременно, с тем чтобы изменения, вносимые одним пользователем, не приводили к непреднамеренному уничтожению изменений, вносимых примерно в то же время иным пользователем.
- **Целостность данных**. SQL позволяет обеспечить целостность базы данных, защищая ее от разрушения из-за несогласованных изменений или отказа системы.

Таким образом, SQL является достаточно мощным языком для управления СУБД и взаимодействия с ней.

Во-вторых, SQL — это не полноценный компьютерный язык типа COBOL, C, C++ или Java. SQL является подъязыком баз данных, в который входит около сорока инструкций, предназначенных для решения задач управления базами данных. Эти инструкции SQL могут быть встроены в другой язык, такой как COBOL или C, и расширяют его, давая возможность получать доступ к базам данных. Кроме того, из такого языка, как C, C++ или Java, инструкции SQL можно посылать СУБД в явном виде, используя *интерфейс на уровне вызовов функций* (call-level interface) или отправляя сообщения по вычислительной сети.

SQL отличается от других языков программирования, поскольку он описывает, *что* пользователь хочет от компьютера, а не *как* компьютер должен это сделать. (Говоря технически, SQL является декларативным, или описательным, а не процедурным языком.) В SQL нет инструкции `IF` для проверки выполнения условия, нет инструкций `GOTO`, `DO` или `FOR` для управления потоком выполнения. Инструкции SQL описывают, как организован набор данных или какие данные должны быть

выбраны или добавлены в базу данных. Последовательность шагов для решения этих задач определяется самой СУБД.

Наконец, SQL — это слабо структурированный язык, особенно по сравнению с такими высокоструктурированными языками, как C, Pascal или Java. Инструкции SQL напоминают обычные предложения естественного языка и содержат "слова-пустышки", не влияющие на смысл инструкции, но облегчающие ее чтение. В SQL почти нет нелогичностей, к тому же имеется ряд специальных правил, предотвращающих создание инструкций, которые выглядят как абсолютно правильные, но не имеют смысла.

Несмотря на не совсем точное название, SQL на сегодняшний день является *стандартом* языка для работы с реляционными базами данных. SQL — это достаточно мощный и в то же время относительно легкий для изучения язык. Краткое введение в SQL, представленное в следующей главе, познакомит вас с основными возможностями этого языка.

Роль SQL

Сам по себе SQL не является ни системой управления базами данных, ни отдельным программным продуктом. Приобрести SQL нельзя ни в магазине, ни на сайте. SQL — это неотъемлемая часть СУБД, язык и инструмент для связи с ней. На рис. 1.2 показаны некоторые компоненты типичной СУБД и как SQL объединяет их в единое целое.

Сердцем СУБД является *механизм базы данных* (database engine, часто называемый просто *движком*); он отвечает за структурирование данных, сохранение и получение их из базы данных. Он принимает SQL-запросы от других компонентов СУБД (таких, как генератор отчетов или модуль запросов), от пользовательских приложений и даже от других вычислительных систем. Как видно из рисунка, SQL выполняет много различных функций.

- SQL — *интерактивный язык запросов*. Для получения данных и вывода их на экран пользователи вводят команды SQL в интерактивных программах. Это удобный способ выполнения специальных запросов.
- SQL — *язык программирования баз данных*. Чтобы получить доступ к базе данных, программисты вставляют в свои прикладные программы команды SQL. Эта методика используется как в программах, написанных пользователями, так и в служебных программах баз данных (например, в таких, как генераторы отчетов).
- SQL — *язык администрирования баз данных*. Администратор базы данных, находящейся на рабочей станции или на сервере, использует SQL для определения структуры базы данных и управления доступом к данным.

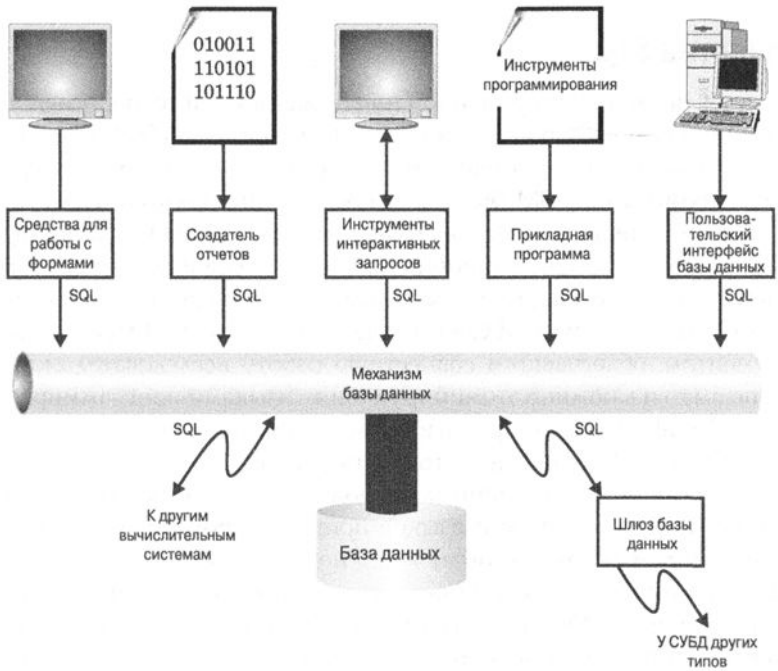


Рис. 1.2. Компоненты типичной системы управления базой данных

- SQL — *язык создания приложений "клиент/сервер"*. В программах для персональных компьютеров SQL используется как средство организации связи по сети с серверами баз данных, в которых хранятся совместно используемые данные. Архитектура "клиент/сервер" используется во многих популярных приложениях корпоративного уровня.
- SQL — *язык доступа к данным в Интернете*. Веб-серверы Интернета, взаимодействующие с корпоративными данными и серверами приложений Интернета, используют SQL в качестве стандартного языка доступа к корпоративным базам данных, зачастую путем внедрения SQL-доступа в популярные языки сценариев наподобие Perl или PHP.
- SQL — *язык распределенных баз данных*. В системах управления распределенными базами данных SQL помогает распределять данные между несколькими соединенными вычислительными системами. Программное обеспечение каждой системы посредством SQL связывается с другими системами, посылая им запросы на доступ к данным.
- SQL — *язык шлюзов баз данных*. В вычислительных сетях с различными СУБД SQL часто используется в *шлюзовой программе*, которая позволяет СУБД одного типа связываться с СУБД другого типа.

Таким образом, SQL — полезный и мощный инструмент, обеспечивающий пользователям, программам и вычислительным системам доступ к информации, содержащейся в реляционных базах данных.

Преимущества SQL

SQL является чрезвычайно успешной информационной технологией. Вспомните, каким был рынок компьютеров в средине 1980-х годов, когда SQL только начинался. Тогда доминировали корпоративные мэйнфреймы и мини-компьютеры. Первый персональный компьютер IBM был создан всего лишь несколько лет назад, а его пользовательским интерфейсом была командная строка MS DOS. Мэйнфреймы IBM работали под управлением операционных систем от Digital Equipment, Data General, Hewlett-Packard и др. Компьютеры соединялись друг с другом при помощи закрытых сетей типа IBM SNA или DECnet от Digital Equipment. Интернет был не более чем инструментом, облегчавшим совместную работу исследовательских лабораторий, а веба не было и в помине. Доминирующими языками программирования были COBOL, C и Pascal; объектно-ориентированное программирование только-только делало первые шаги, а Java еще предстояло изобрести.

Во всех областях информационных технологий — от аппаратного обеспечения до операционных систем, сетей и языков программирования — ключевые технологии средины 1980-х годов уступили место новым решениям и методам. Но в мире управления данными реляционные базы данных и SQL никому не уступили свое место под солнцем. Они развивались, чтобы соответствовать новому аппаратному и программному обеспечению, операционным системам, сетям и языкам программирования. Несмотря на массу попыток свергнуть их с престола, реляционная модель и SQL преуспевают и остаются *единственной доминирующей силой* в области управления данными. Вот некоторые основные свойства SQL, обеспечивающие такой небывалый его успех в течение последних десятилетий.

- Независимость от конкретных СУБД
- Межплатформенная переносимость
- Наличие стандартов
- Поддержка со стороны компании IBM
- Поддержка со стороны компании Microsoft
- Построение на реляционной модели
- Высокоуровневая структура, напоминающая естественный язык
- Возможность выполнения специальных интерактивных запросов
- Обеспечение программного доступа к базам данных
- Возможность различного представления данных
- Полноценность в качестве языка, предназначенного для работы с базами данных
- Возможность динамического определения данных
- Поддержка архитектуры клиент/сервер
- Поддержка приложений уровня предприятия
- Расширяемость и поддержка объектно-ориентированных технологий
- Возможность доступа к данным в Интернете

- Интеграция с языком Java (протокол JDBC)
- Поддержка открытого кода
- Промышленная инфраструктура

Ниже эти факторы рассмотрены более подробно.

Независимость от конкретных СУБД

Все ведущие поставщики СУБД используют SQL, и в течение последнего десятилетия ни одна новая СУБД, не поддерживающая SQL, не может рассчитывать на успех. Реляционную базу данных и программы, которые с ней работают, можно перенести с одной СУБД на другую с минимальными доработками и переподготовкой персонала. Программные средства, входящие в состав СУБД для персональных компьютеров, такие как программы для создания запросов, генераторы отчетов и генераторы приложений, работают с реляционными базами данных многих типов. Таким образом, SQL обеспечивает независимость от конкретных СУБД, что является одной из наиболее важных причин его популярности.

Межплатформенная переносимость

Реляционные СУБД выполняются на различных вычислительных системах — от мэйнфреймов и систем среднего уровня до персональных компьютеров, рабочих станций и переносных ПК. Они функционируют на отдельных компьютерах, в локальных и корпоративных сетях и даже на уровне Интернета. Приложения, созданные с помощью SQL и рассчитанные на однопользовательские системы, по мере своего развития могут быть перенесены на более крупные системы. Информация из корпоративных реляционных баз данных может быть загружена в базы данных отдельных подразделений или в персональные базы данных пользователей. Наконец, экономичные персональные компьютеры могут использоваться для тестирования прототипа приложения базы данных с использованием SQL, перед тем как перенести его на дорогую многопользовательскую систему.

Стандарты языка SQL

Официальный стандарт языка SQL был опубликован Американским национальным институтом стандартов (American National Standards Institute, ANSI) и Международной организацией по стандартизации (International Standards Organization, ISO) в 1986 году, после чего был расширен в 1989 году, а затем — в 1992, 1999, 2003 и 2006 годах. Кроме того, SQL является федеральным стандартом США в области обработки информации (Federal Information Processing Standard, FIPS), и, следовательно, соответствие ему является одним из основных требований, содержащихся в больших правительственных контрактах на разработки в компьютерной промышленности. В течение многих лет свой вклад в стандартизацию различных составляющих SQL, таких как интерфейсы программирования и объектно-ориентированные расширения, вносили многие международные, правительственные и промышленные группы. Со временем часть подобных инициатив стала составной частью

стандарта ANSI/ISO. Все эти стандарты служат как бы официальной печатью, одобряющей SQL, и они ускорили завоевание им рынка.

Поддержка со стороны IBM

Изначально SQL разрабатывался исследователями IBM и быстро стал стратегическим продуктом, подтверждением чему служит флагманская СУБД DB2 компании IBM. Поддержка SQL имеется для всех основных семейств компьютеров компании IBM — от персональных компьютеров до мощных мэйнфреймов. Работа IBM ясно указала направление развития для других поставщиков баз данных и программных систем. Позже поддержка IBM существенно ускорила принятие SQL рынком. В 1970-х годах IBM была доминирующей силой на рынке вычислительной техники, так что ее поддержку SQL трудно переоценить.

Поддержка со стороны Microsoft

Компания Microsoft рассматривает подсистему доступа к базам данных как ключевую часть архитектуры Windows для персональных компьютеров. Как настольная, так и серверная версия Windows предоставляет стандартизованный доступ к реляционным базам данных посредством ODBC (Open Database Connectivity, открытый доступ к базам данных), программного интерфейса, основанного на SQL. Протокол ODBC поддерживается наиболее распространенными приложениями Windows (электронными таблицами, текстовыми процессорами, базами данных и т.п.), разработанными как самой компанией Microsoft, так и другими ведущими поставщиками, а кроме того, все ведущие SQL-базы данных обеспечивают доступ посредством ODBC. Со временем Microsoft расширила поддержку ODBC более высокоуровневыми объектно-ориентированными надстройками, включая поддержку управления данными в .NET. Но все эти новые технологии всегда могут работать с реляционными базами данных на более низком уровне ODBC/SQL. Когда в конце 1980-х годов Microsoft начала предпринимать усилия по превращению Windows в жизнеспособную серверную операционную систему, она представила SQL Server в качестве новой собственной SQL-базы данных. SQL Server и сегодня является флагманским продуктом Microsoft и ключевым компонентом архитектуры Microsoft .NET для веб-служб.

Основанность на реляционной модели

SQL является языком реляционных баз данных, поэтому он стал популярным тогда, когда популярной стала реляционная модель представления данных. Табличная структура реляционной базы данных со строками и столбцами интуитивно понятна пользователям, поэтому язык SQL является простым и легким для изучения. Реляционная модель имеет солидный теоретический фундамент, послуживший основой для эволюции и реализации реляционных баз данных. На волне популярности, вызванной успехом реляционной модели, SQL стал, по сути, *единственным* языком для реляционных баз данных.

Высокоуровневая структура, напоминающая естественный язык

Инструкции SQL похожи на обычные предложения английского языка, что упрощает их изучение и понимание. Частично это обусловлено тем, что инструкции SQL описывают *данные*, которые необходимо получить, а *не способ* их поиска. Таблицы и столбцы в реляционной базе данных могут иметь длинные описательные имена. В результате большинство инструкций SQL означает именно то, что точно соответствует их именам, поэтому их можно читать как простые, естественные предложения.

Интерактивные запросы

SQL является языком интерактивных запросов, который обеспечивает пользователям немедленный доступ к данным. С помощью SQL пользователь может в интерактивном режиме получить ответы на самые сложные запросы в считанные минуты или секунды, тогда как программисту потребовались бы дни или недели, чтобы написать соответствующую программу. Из-за того что SQL допускает интерактивное формирование запросов, данные становятся более доступными и могут помочь в принятии решений, делая их более обоснованными. Интерактивность SQL на ранних этапах его эволюции была важным преимуществом над нереляционными базами данных и позже осталась таковым по отношению к чисто объектно-ориентированным базам данных.

Программный доступ к базе данных

Программисты пользуются языком SQL при создании приложений, обращающихся к базам данных. Одни и те же инструкции SQL используются как для интерактивного, так и для программного доступа, поэтому части программ, содержащие обращения к базе данных, можно вначале тестировать в интерактивном режиме, а затем встраивать в программу. В традиционных базах данных для программного доступа используются одни программные средства, а для выполнения интерактивных запросов — другие, без какой-либо связи между этими двумя режимами доступа.

Различные представления данных

С помощью SQL создатель базы данных может сделать так, что различные пользователи базы данных будут видеть различные *представления* ее структуры и содержимого. Например, базу данных можно спроектировать таким образом, что каждый пользователь будет видеть только данные, относящиеся к его подразделению или торговому региону. Кроме того, данные из различных частей базы данных могут быть скомбинированы и представлены пользователю в виде одной простой таблицы. Следовательно, представления можно использовать для усиления защиты базы данных и ее настройки под конкретные требования отдельных пользователей.

Полноценный язык для работы с базами данных

Первоначально SQL был задуман как язык интерактивных запросов, но сейчас он вышел далеко за рамки выборки данных. SQL является полноценным, самосо-

гласованным и логичным языком, предназначенным для создания базы данных, управления ее защитой, изменения ее содержимого, выборки данных и совместного их использования несколькими параллельно работающими пользователями. Концепции, освоенные при изучении одной части языка, могут затем применяться в других командах, что повышает производительность работы пользователей.

Динамическое определение данных

С помощью SQL можно динамически изменять и расширять структуру базы данных даже в то время, когда пользователи обращаются к ее содержимому. Это большое преимущество перед статическими языками определения данных, которые запрещают доступ к базе данных во время изменения ее структуры. Таким образом, SQL обеспечивает максимальную гибкость, так как дает базе данных возможность адаптироваться к изменяющимся требованиям, не прерывая работу приложения, выполняющегося в реальном масштабе времени.

Архитектура "клиент/сервер"

SQL — естественное средство для реализации приложений, использующих распределенную архитектуру "клиент/сервер". В этой роли SQL служит связующим звеном между клиентской системой, взаимодействующей с пользователем, и серверной системой, управляющей базой данных, позволяя каждой из них сосредоточиться на выполнении своих функций. Кроме того, SQL дает возможность персональным компьютерам функционировать в качестве клиентов по отношению к сетевым серверам или более крупным базам данных, установленным на мэйнфреймах; это позволяет получать доступ к корпоративным данным из приложений, работающих на персональных компьютерах.

Поддержка приложений уровня предприятия

Все крупные приложения уровня предприятия, поддерживающие ежедневную деятельность больших компаний и организаций, используют для хранения и организации информации SQL-базы данных. В 1990-х годах, в связи с так называемой "проблемой 2000 года", большие фирмы массово перешли от собственных доморощенных систем к приложениям от таких производителей, как SAP, Oracle, PeopleSoft, Siebel и др. Данные, обрабатываемые такими приложениями (заказы, продажи, клиенты, склады и т.п.), обычно имеют структурированный вид записей с полями, который легко преобразуется в формат строк и столбцов SQL. Создавая свои приложения на базе SQL-баз данных уровня предприятия, производители программного обеспечения избегают необходимости разрабатывать собственные системы управления данными и пользуются всеми преимуществами существующих инструментов и опытом программистов. Поскольку все основные приложения уровня предприятия для своей работы требуют SQL-базы данных, увеличение продаж таких приложений автоматически влечет повышение спроса на новое программное обеспечение баз данных.

Расширяемость и поддержка объектно-ориентированных технологий

Основным вызовом доминированию языка SQL в качестве стандарта баз данных стало появление объектно-ориентированного программирования и языков программирования наподобие Java и C++. Как следствие общей направленности компьютерного рынка в сторону объектно-ориентированных технологий, появились объектные базы данных. В ответ на это поставщики реляционных СУБД начали постепенно расширять и модернизировать SQL, добавляя в него различные объектные возможности. Появившиеся в результате "объектно-реляционные" базы данных, основанные, как и ранее, на SQL, стали более популярной альтернативой "чисто объектным" базам данных, обеспечив тем самым дальнейшее доминирование SQL в течение последнего десятилетия. Новейшие веяния объектных технологий, воплотившиеся в XML и архитектурах веб-служб, вновь "покусились" на доминирование SQL посредством "XML-баз данных" и альтернативных языков запросов в начале 2000-х годов. И вновь ведущие производители сумели отреагировать на угрозу путем добавления в язык XML-расширений. Пока что история SQL дает основания надеяться, что так же будут преодолены и новые вызовы, которые могут появиться в будущем.

Возможность доступа к данным в Интернете

Рост популярности Интернета и Веб привел к тому, что к концу 1990-х годов SQL стал рассматриваться и как стандартный язык для доступа к данным в Интернете. Первоначально, в эпоху зарождения веб, разработчики, занимавшиеся отображением на веб-страницах информации, извлеченной из баз данных, применяли SQL как средство взаимодействия со шлюзами баз данных. Позднее, с появлением трехуровневой архитектуры Интернета с четким разделением на тонкие клиенты, серверы приложений и серверы баз данных, SQL стал связующим звеном между вторым и третьим уровнями. Роль SQL в многоуровневой архитектуре в настоящее время начинает выходить за рамки серверных баз данных и включает кеширование и управление данными в реальном времени на уровне приложений или вблизи него.

Интеграция с языком Java (протокол JDBC)

Основной областью разработки SQL в последние пять–десять лет являлась интеграция SQL с Java. Осознав необходимость связи языка программирования Java с существующими реляционными базами данных, Sun Microsystems (создатель Java) предложил интерфейс JDBC (Java Database Connectivity), стандартный интерфейс прикладного программирования, позволяющий программистам на Java пользоваться SQL для обращения к базам данных. JDBC получил дальнейшую поддержку, когда был принят в качестве стандарта доступа к данным в спецификации Java2 Enterprise Edition (J2EE), которая определяет операционную среду, предоставляемую большинством ведущих серверов приложений Интернета. В дополнение к роли Java в качестве языка программирования, имеющего доступ к базам данных, многие ведущие производители реализовали поддержку Java *в* своих системах баз данных, тем самым обеспечив применение Java в качестве языка хранимых процедур и бизнес-логики в самих базах данных. Такая тенденция интеграции Java и SQL обеспечивает важность SQL и в новую эру Java-программирования.

Поддержка открытого кода

Одним из новейших важных событий в компьютерной индустрии стало применение подхода "открытого кода" в построении сложных программных систем. При этом подходе исходный текст является открытым и доступным бесплатно, так что многие программисты могут вносить в него свой вклад, добавлять в него новые возможности, исправлять ошибки, улучшать функциональность и т.п. Такое сообщество программистов, которые могут работать в различных организациях по всему миру, при определенной координации становится мощным двигателем новой технологии. Программное обеспечение с открытым кодом обычно доступно по невысокой цене (или вовсе бесплатно), что только добавляет ему привлекательности. В течение последнего десятилетия разработан ряд успешных проектов SQL-баз данных с открытым кодом, одна из которых, MySQL, стала стандартным компонентом набора программного обеспечения с открытым кодом LAMP, в который, кроме MySQL, входят операционная система Linux, веб-сервер Apache и язык сценариев PHP. Широкая доступность бесплатных SQL-баз данных с открытым кодом способствует дальнейшему росту популярности SQL.

Промышленная инфраструктура

Пожалуй, наиболее важным фактором растущей популярности SQL можно считать появление целой промышленной инфраструктуры, основанной на SQL. СУРБД на базе SQL являются важной частью этой инфраструктуры. Еще одной важной частью являются приложения уровня предприятия, использующие SQL и требующие наличия SQL-базы данных. Кроме того, можно упомянуть массу инструментов, таких как генераторы отчетов, инструменты для ввода данных, проектирования баз данных, программный инструментарий и многое другое, что облегчает применение SQL. Критической частью этой инфраструктуры является большое количество опытных программистов SQL. Еще одной важной частью является легкодоступное обучение и помощь при работе с SQL. Целые фирмы специализируются на консультировании по вопросам SQL, оптимизации и повышении производительности кода. Все эти части усиливают друг друга и обеспечивают успех SQL. Проще говоря, при наличии задач управления данными простейшие, наименее рискованные и наиболее дешевые решения получаются при применении SQL.

ГЛАВА 2

Краткий обзор SQL

Перед тем как погрузиться в детали SQL, в общих чертах познакомимся с языком, его перспективами и принципами работы. В данной главе приводится краткий обзор основных его возможностей и функций. Его цель не в том, чтобы сделать вас специалистом по написанию инструкций SQL, — это задача части II, "Выборка данных", настоящей книги. Данная глава дает лишь общее представление о возможностях языка.

Простая база данных

Примеры, приводимые в этой главе, основаны на простой реляционной базе данных маленькой торговой компании. Структура этой базы данных изображена на рис. 2.1. В ней хранится информация, необходимая для реализации небольшого приложения по обработке заказов. Инструкции по созданию этой простой базы данных вы найдете в приложении А, "Учебная база данных", так что можете проверять работу описываемых запросов на практике прямо во время чтения. В базе хранится следующая информация:

- о *клиентах*, которые покупают товары компании;
- о *заказах*, сделанных клиентами;
- о *служащих* компании, которые продают товары клиентам;
- об *офисах*, где работают служащие.

Эта база данных, как и большинство других, является моделью "реального мира". Данные, содержащиеся в ней, представляют реальные сущности: клиенты, заказы, служащие компании и офисы. Для каждой сущности имеется собственная отдельная таблица. Например, в таблице SALESREPS каждый продавец представлен отдельной строкой, а каждый столбец хранит определенную информацию о продавце, такую как его имя или офис, где он работает. Запросы к базе данных, создаваемые с помощью SQL, отражают события, происходящие в реальном мире: клиенты делают, от-

СКАЧАНО С САЙТА: WWW.SHKOLNIK.BY ПРИСОЕДИНЯЙСЯ!

меняют и изменяют заказы, владелец компании нанимает и увольняет служащих и т.д. Давайте посмотрим, что можно делать с этими данными с помощью SQL.

Таблица ORDERS

ORDER_NUM	CUST	PRODUCT	QTY	AMOUNT
112961	2117	2A44L	7	$31,500.00
113012	2111	41003	35	$3,745.00
112989	2101	114	6	$1,458.00
113051	2118	XK47	4	$1,420.00
112968	2102	41004	34	$3,978.00
113036	2107	4100Z	9	$22,500.00
113045	2112	2A44R	10	$45,000.00
112963	2103	41004	28	$3,276.00
113013	2118	41003	1	$652.00
113058	2108	112	10	$1,480.00
112997	2124	41003	1	$652.00
112983	2103	41004	6	$702.00
113024	2114	XK47	20	$7,100.00
113062	2124	114	10	$2,430.00
112979	2114	4100Z	6	$15,000.00
113027	2103	41002	54	$4,104.00
113007	2112	773C	3	$2,925.00
113069	2109	775C	22	$31,350.00
113034	2107	2A45C	8	$632.00
112992	2118	41002	10	$760.00
112975	2111	2A44G	6	$2,100.00
113055	2108	4100X	6	$150.00
113048	2120	779C	2	$3,750.00
112993	2106	2A45C	24	$1,896.00
113065	2106	XK47	6	$2,130.00
113003	2108	779C	3	$5,625.00
113049	2118	XK47	2	$776.00
112987	2103	4100Y	11	$27,500.00
113057	2111	4100X	24	$600.00
113042	2113	2A44R	5	$22,500.00

Таблица CUSTOMERS

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2111	JCP Inc.	103	$50,000.00
2102	First Corp.	101	$65,000.00
2103	Acme Mfg.	105	$50,000.00
2123	Carter & Sons	102	$40,000.00
2107	Ace International	110	$35,000.00
2115	Smithson Corp.	101	$20,000.00
2101	Jones Mfg.	106	$65,000.00
2112	Zetacorp	108	$50,000.00
2121	QMA Assoc.	103	$45,000.00
2114	Orion Corp.	102	$20,000.00
2124	Peter Brothers	107	$40,000.00
2108	Holm & Landis	109	$55,000.00
2117	J.P. Sinclair	106	$35,000.00
2122	Three-Way Lines	105	$30,000.00
2120	Rico Enterprises	102	$50,000.00
2106	Fred Lewis Corp.	102	$65,000.00
2119	Solomon Inc.	109	$25,000.00
2118	Midwest Systems	108	$60,000.00
2113	Ian & Schmidt	104	$20,000.00
2109	Chen Associates	107	$25,000.00
2105	AAA Investments	101	$45,000.00

Таблица SALESREPS

NAME	REP_OFFICE	QUOTA	SALES
Bill Adams	13	$350,000.00	$367,911.00
Mary Jones	11	$300,000.00	$392,725.00
Sue Smith	21	$350,000.00	$474,050.00
Sam Clark	11	$275,000.00	$299,912.00
Bob Smith	12	$200,000.00	$142,594.00
Dan Roberts	12	$300,000.00	$305,673.00
Tom Snyder	NULL	NULL	$75,985.00
Larry Fitch	21	$350,000.00	$361,865.00
Paul Cruz	12	$275,000.00	$286,775.00
Nancy Angelli	22	$300,000.00	$186,042.00

Таблица OFFICES

OFFICE	CITY	REGION	TARGET	SALES
22	Denver	Western	$300,000.00	$186,042.00
11	New York	Eastern	$575,000.00	$692,637.00
12	Chicago	Eastern	$800,000.00	$735,042.00
13	Atlanta	Eastern	$350,000.00	$367,911.00
21	Los Angeles	Western	$725,000.00	$835,915.00

Рис. 2.1. Простая реляционная база данных

Выборка данных

Вначале просмотрим список офисов с указанием города, где размещается офис, и объема продаж офиса с начала года по текущий день. Инструкция SQL, которая извлекает информацию из базы данных, называется SELECT. Приведенная ниже инструкция SQL выбирает из базы данных интересующую вас информацию.

```
SELECT CITY, OFFICE, SALES
  FROM OFFICES;

CITY          OFFICE        SALES
-----------   ------   -----------
Denver            22   $186,042.00
New York          11   $692,637.00
Chicago           12   $735,042.00
Atlanta           13   $367,911.00
Los Angeles       21   $835,915.00
```

Инструкция SELECT запрашивает для каждого офиса три вида данных: город, номер офиса и объем продаж. Еще она определяет, что данные находятся в таблице OFFICES, в которой хранится информация об офисах. Результаты запроса приведены сразу после рассматриваемой инструкции в форме таблицы. Заметим, что форматирование вывода результатов запроса зависит от конкретной реализации SQL и может изменяться при переходе от одной СУБД к другой.

Инструкция SELECT применяется во всех SQL-запросах на выборку данных. Так, далее приведен запрос, который запрашивает список имен и текущих объемов продаж по всем служащим в базе данных. Кроме того, в запросе приводится планируемый объем продаж и номер офиса, где работает служащий. В этом случае данные извлекаются из таблицы SALESREPS.

```
SELECT NAME, REP_OFFICE, SALES, QUOTA
  FROM SALESREPS;

NAME           REP_OFFICE        SALES        QUOTA
-------------  ----------   -----------  -----------
Bill Adams             13   $367,911.00  $350,000.00
Mary Jones             11   $392,725.00  $300,000.00
Sue Smith              21   $474,050.00  $350,000.00
Sam Clark              11   $299,912.00  $275,000.00
Bob Smith              12   $142,594.00  $200,000.00
Dan Roberts            12   $305,673.00  $300,000.00
Tom Snyder           NULL    $75,985.00         NULL
Larry Fitch            21   $361,865.00  $350,000.00
Paul Cruz              12   $286,775.00  $275,000.00
Nancy Angelli          22   $186,042.00  $300,000.00
```

Значения NULL в строке продавца Тома Снайдера представляют отсутствующие или неизвестные данные. Он — новичок в компании и пока не получил ни назначения в определенный офис, ни планируемый объем продаж. Но тем не менее он уже осуществил некоторое количество продаж, о чем свидетельствуют выведенные данные в его строке.

SQL позволяет также запрашивать вычисляемые результаты. Например, можно попросить вычислить сумму, на которую каждый служащий опережает план или отстает от него.

```
SELECT NAME, SALES, QUOTA, (SALES - QUOTA)
  FROM SALESREPS;

NAME                 SALES        QUOTA  (SALES-QUOTA)
-------------   -----------  -----------  -------------
Bill Adams      $367,911.00  $350,000.00     $17,911.00
Mary Jones      $392,725.00  $300,000.00     $92,725.00
Sue Smith       $474,050.00  $350,000.00    $124,050.00
Sam Clark       $299,912.00  $275,000.00     $24,912.00
Bob Smith       $142,594.00  $200,000.00    -$57,406.00
Dan Roberts     $305,673.00  $300,000.00      $5,673.00
Tom Snyder       $75,985.00         NULL           NULL
Larry Fitch     $361,865.00  $350,000.00     $11,865.00
Paul Cruz       $286,775.00  $275,000.00     $11,775.00
Nancy Angelli   $186,042.00  $300,000.00   -$113,958.00
```

Запрашиваемые данные (включая вычисленную разницу между объемом продаж и планом) снова представлены в виде строк и столбцов таблицы. Возможно, вы хотели бы получить сведения о служащих, которые не выполняют план. SQL позволяет легко получить такую информацию, добавив в предыдущий запрос операцию сравнения.

```
SELECT NAME, SALES, QUOTA, (SALES - QUOTA)
  FROM SALESREPS
 WHERE SALES < QUOTA;

NAME                 SALES        QUOTA  (SALES-QUOTA)
-------------   -----------  -----------  ------------
Bob Smith       $142,594.00  $200,000.00    -$57,406.00
Nancy Angelli   $186,042.00  $300,000.00   -$113,958.00
```

С помощью этого же приема можно получить список больших заказов и определить, кто сделал конкретный заказ, какие товары и в каких количествах были заказаны. SQL, кроме того, позволяет упорядочить заказы по их стоимости.

```
SELECT ORDER_NUM, CUST, PRODUCT, QTY, AMOUNT
  FROM ORDERS
 WHERE AMOUNT > 25000.00
 ORDER BY AMOUNT;

ORDER_NUM   CUST   PRODUCT   QTY       AMOUNT
---------   ----   -------   ---   ----------
   112987   2103   4100Y      10   $27,500.00
   113069   2109   775C       22   $31,350.00
   112961   2117   2A44L       7   $31,500.00
   113045   2112   2A44R      10   $45,000.00
```

Получение итоговых данных

SQL можно использовать не только для выборки данных, но и для получения итоговых значений по содержимому базы данных. Какова средняя стоимость заказов в базе данных? Следующий запрос обеспечивает вычисление средней стоимости.

```
SELECT AVG(AMOUNT)
  FROM ORDERS;

AVG(AMOUNT)
-----------
  $8,256.37
```

Можно также узнать среднюю стоимость всех заказов, сделанных конкретным клиентом.

```
SELECT AVG(AMOUNT)
  FROM ORDERS
 WHERE CUST = 2103;

AVG(AMOUNT)
-----------
  $8,895.50
```

Наконец, давайте найдем общую стоимость всех заказов, сделанных каждым клиентом. Для этого сгруппируем заказы по номерам клиентов, а затем просуммируем их по каждому клиенту.

```
SELECT CUST, SUM(AMOUNT)
  FROM ORDERS
 GROUP BY CUST;

CUST      SUM(AMOUNT)
----      -----------
2101        $1,458.00
2102        $3,978.00
2103       $35,582.00
2106        $4,026.00
2107       $23,132.00
2108        $7,255.00
2109       $31,350.00
2111        $6,445.00
2112       $47,925.00
2113       $22,500.00
2114       $22,100.00
2117       $31,500.00
2118        $3,542.00
2120        $3,750.00
2124        $3,082.00
```

Добавление данных

SQL можно использовать и для добавления в таблицы новых данных. Предположим, что вы открыли в Далласе новый офис "Western" с плановым объемом продаж $275000. Ниже приведена инструкция INSERT, которая добавляет в соответствующую таблицу новый офис с номером 23.

```
INSERT INTO OFFICES (CITY, REGION, TARGET, SALES, OFFICE)
   VALUES ('Dallas', 'Western', 275000.00, 0.00, 23);

1 row inserted.
```

Аналогично, если служащая Мери Джонс (номер 109) заключает договор с новым клиентом, компанией Acme Industries, приведенная ниже инструкция INSERT добавит в соответствующую таблицу имя клиента с номером 2125 и лимитом кредита в $25000.

```
INSERT INTO CUSTOMERS (COMPANY, CUST_REP, CUST_NUM, CREDIT_LIMIT)
     VALUES ('Acme Industries', 109, 2125, 25000.00);

1 row inserted.
```

Как можно заметить, механизм базы данных возвращает сообщение (1 row inserted) о том, что инструкция сработала. Точный текст и форматирование ответа варьируются от одной реализации SQL к другой.

Удаление данных

Точно так же, как инструкция INSERT добавляет в таблицу новые данные, инструкция DELETE удаляет данные из таблицы. Если через несколько дней компания Acme Industries решит отказаться от ваших услуг и уйти к конкуренту, вы сможете удалить из базы данных информацию о ней с помощью следующей инструкции.

```
DELETE FROM CUSTOMERS
 WHERE COMPANY = 'Acme Industries';

1 row deleted.
```

Если вы решите уволить всех служащих, чей объем продаж меньше плана, то сможете удалить их имена из базы данных с помощью такой инструкции.

```
DELETE FROM SALESREPS
 WHERE SALES < QUOTA;

2 rows deleted.
```

Обновление данных

SQL можно использовать и для обновления информации, уже содержащейся в базе данных. Например, чтобы увеличить лимит кредита для компании First Corp. до $75000, можно воспользоваться следующей инструкцией UPDATE.

```
UPDATE CUSTOMERS
   SET CREDIT_LIMIT = 75000.00
 WHERE COMPANY  = 'First Corp.';

1 row updated.
```

С помощью инструкции UPDATE можно вносить в базу данных несколько обновлений одновременно. Например, следующая инструкция UPDATE увеличивает план для всех продавцов на $15000.

```
UPDATE SALESREPS
   SET QUOTA = QUOTA + 15000.00;

8 rows updated.
```

Защита данных

Важной задачей базы данных является защита информации от несанкционированного доступа. Предположим, ваш секретарь Мери прежде не имела разрешения на ввод в базу данных сведений о новых клиентах. С помощью следующей инструкции SQL можно дать ей такое разрешение.

```
GRANT INSERT
   ON CUSTOMERS
   TO MARY
```

```
Privilege granted.
```

Аналогично приведенная ниже инструкция дает Мери разрешение на изменение данных о клиентах и чтение информации о них с помощью SELECT.

```
GRANT UPDATE, SELECT
   ON CUSTOMERS
   TO MARY;

Privilege granted.
```

Если вы решите запретить Мери добавлять в базу данных сведения о новых клиентах, для этого достаточно воспользоваться инструкцией REVOKE.

```
REVOKE INSERT
    ON CUSTOMERS
  FROM MARY;

Privilege revoked.
```

Точно так же следующая инструкция REVOKE отменит все привилегии Мери на доступ и модифицирование данных о клиентах.

```
REVOKE ALL
    ON CUSTOMERS
  FROM MARY;

Privilege revoked.
```

Создание базы данных

Для того чтобы в базе данных можно было хранить информацию, сначала необходимо определить ее структуру. Предположим, что вы хотите расширить нашу учебную базу данных, добавив в нее таблицу с информацией о товарах, которые продает ваша компания. Для каждого товара должны храниться следующие данные:

- идентификатор производителя — три символа;
- идентификатор товара — пять символов;
- описание — до тридцати символов;
- цена товара;
- количество товара.

Приведенная далее инструкция CREATE TABLE определяет новую таблицу для хранения указанных данных о товарах.

```
CREATE TABLE PRODUCTS
     (MFR_ID CHAR(3),
  PRODUCT_ID CHAR(5),
 DESCRIPTION VARCHAR(30),
       PRICE DECIMAL(9,2),
 QTY_ON_HAND INTEGER);

Table created.
```

Хотя инструкция CREATE TABLE не столь очевидна, как инструкции, рассматривавшиеся ранее, тем не менее она все же довольно проста. Эта инструкция присваивает новой таблице имя PRODUCTS и определяет для каждого из пяти ее столбцов имя и тип данных, хранимых в нем. Идентификаторы производителя и товара хранятся в виде последовательностей символов фиксированной длины, описание товара — в виде строки переменной длины, цена представляет собой десятичные данные (действительное число), а количество — целочисленное значение.

После того как таблица создана, ее можно заполнять данными. Вот инструкция INSERT, предназначенная для ввода данных о новой партии изделия Size 7 Widget (товар ACI-41007) в количестве 250 единиц по цене $225 за штуку.

```
INSERT INTO PRODUCTS (MFR_ID, PRODUCT_ID, DESCRIPTION,
                      PRICE, QTY_ON_HAND)
              VALUES ('ACI', '41007', 'Size 7 Widget',
                      225.00, 250);

1 row inserted.
```

Наконец, если позднее вы решите, что в базе данных больше не требуется хранить информацию о товарах, то можете удалить таблицу (вместе со всеми данными, которые в ней содержатся) с помощью инструкции DROP TABLE.

```
DROP TABLE PRODUCTS;

Table dropped.
```

Резюме

В данной главе показаны основные возможности SQL и на примере наиболее распространенных инструкций SQL продемонстрирован синтаксис языка. Итак, подведем итоги.

- SQL используется для *выборки* информации из базы данных с помощью инструкции SELECT. Можно извлечь все данные из таблицы или лишь часть из них, отсортировать их и получить итоговые значения, вычисляя суммы и средние величины.
- SQL используется для *изменения* информации в базе данных. Инструкция INSERT добавляет данные, инструкция DELETE удаляет их, а инструкция UPDATE обновляет существующие данные.
- SQL используется для *управления доступом* к базе данных. С помощью инструкций SQL предоставляются и отменяются разного рода привилегии для различных пользователей.
- SQL используется для *создания и изменения* базы данных путем определения структуры новых таблиц и удаления таблиц, ставших ненужными, для чего применяются инструкции CREATE и DROP.

ГЛАВА 3

Перспективы SQL

Сегодня SQL является стандартным языком управления базами данных. Что это означает? Как SQL стал стандартом? Какую роль играет официальный стандарт SQL? Насколько он поддерживается и почему, несмотря на стандарт, имеются диалекты SQL? Насколько сильно SQL влияет на различные сегменты компьютерного рынка? Чтобы ответить на эти вопросы, в настоящей главе прослеживается история развития SQL и рассказывается о его нынешней роли на рынке компьютерных технологий.

SQL и эволюция управления базами данных

Одной из основных задач вычислительной системы является хранение и обработка данных. В конце 1960-х–начале 70-х годов стали появляться специализированные компьютерные программы для решения этой задачи, известные под названием *системы управления базами данных* (СУБД). СУБД помогала пользователям компьютеров организовывать и структурировать данные и позволяла вычислительной системе играть более активную роль в обработке данных. Хотя изначально СУБД использовались на больших ЭВМ (мэйнфреймах), их популярность быстро распространилась на мини-компьютеры, а затем и на рабочие станции, персональные компьютеры и специализированные серверы.

Системы управления базами данных играли ключевую роль в стремительном развитии компьютерных сетей и Интернета. Ранние СУБД работали в крупных монолитных вычислительных комплексах, где данные, программное обеспечение СУБД и прикладные программы, осуществлявшие доступ к базе данных, работали в единой системе. В 1980-х–1990-х годах получила распространение архитектура "клиент/сервер", в которой пользователь персонального компьютера или прикладная программа посредством локальной сети выполняли обращение к базе данных, расположенной в другой системе. В конце 90-х годов растущая популярность Интернета и Веб оказала влияние на архитектуру управления данными. Сегодня пользователю зачастую достаточно иметь лишь веб-браузер, чтобы получить

доступ к базам данных, расположенным не только в его организации, но и в любой точке земного шара. Такие интернет-архитектуры обычно включают три и более компьютерных систем: одна, где работает веб-браузер, обеспечивает взаимодействие с пользователем и подключена через Интернет ко второй — серверу приложений, — где работает прикладное программное обеспечение, а та, в свою очередь, подключена к третьей системе, где работает СУБД.

Сегодня рынок СУБД — это очень большой бизнес. Независимые компании по производству программного обеспечения и крупные поставщики продают программы для управления базами данных на миллиарды долларов ежегодно. Практически все компьютерные приложения уровня предприятия, поддерживающие деятельность крупных компаний и иных организаций, используют базы данных. Эти приложения включают некоторые быстрорастущие категории приложений, такие как ERP (Enterprise Resource Planning, управление ресурсами предприятия), CRM (Customer Relationship Management, система управления взаимосвязями с клиентами и партнерами), SCM (Supply Chain Management, управление цепочками поставок), SFA (Sales Force Automation, автоматизация процесса продаж), а также финансовые приложения. Специализированные высокопроизводительные серверы, оптимизированные для работы большинства популярных баз данных, образуют многомиллиардный рынок, и еще большие миллиарды добавляет рынок дешевых серверов. Базы данных работают "за сценой" большинства транзакционно-ориентированных веб-сайтов и используются для хранения и анализа пользовательских транзакций. Таким образом, управление данными пронизывает все сегменты компьютерного рынка.

С конца 1980-х годов произошел стремительный взлет популярности СУБД конкретного типа — системы управления *реляционными* базами данных (СУРБД). С тех пор реляционная база данных стала, по сути, *стандартом* баз данных. Информация в реляционной базе данных хранится в простом табличном виде, что дает реляционным базам данных много преимуществ по сравнению с базами данных более ранних разработок. SQL предназначен, в первую очередь, для работы именно с реляционными базами данных.

Краткая история SQL

История SQL тесно связана с развитием реляционных баз данных. В табл. 3.1 перечислены основные вехи его сорокалетней истории. Понятие реляционной базы данных было введено доктором Э. Ф. Коддом (Edgar Frank "Ted" Codd), научным сотрудником компании IBM. В июне 1970 года доктор Кодд опубликовал в журнале *Communications of the Association for Computing Machinery* статью "Реляционная модель для больших банков совместно используемых данных" ("A Relational Model of Data for Large Shared Data Banks"), в которой в общих чертах была изложена математическая теория хранения данных в табличной форме и их обработки. От этой статьи и берут свое начало реляционные базы данных и SQL.

Таблица 3.1. Основные этапы развития SQL

Год	Событие
1970	Доктор Кодд создает модель реляционной базы данных
1974	Начинается разработка проекта System/R компании IBM
1974	Первая статья с описанием языка SEQUEL
1978	Опытная эксплуатация проекта System/R
1979	Появляется первая коммерческая СУРБД компании Oracle
1981	Компания Relational Technology выпускает СУБД Ingres
1981	Компания IBM создает СУБД SQL/DS
1982	ANSI формирует комитет по стандартизации языка SQL
1983	Компания IBM объявляет о создании СУБД DB2
1986	ANSI принимает стандарт SQL1
1986	Компания Sybase создает СУРБД для обработки транзакций
1987	ISO одобряет стандарт SQL1
1988	Компании Ashton-Tate и Microsoft объявляют о выпуске СУБД SQL Server для операционной системы OS/2
1989	Опубликован первый тест производительности TPC (TPC-A)
1990	Опубликован тест производительности TPC-B
1991	Консорциум SQL Access Group публикует спецификацию доступа к базам данных
1992	Компания Microsoft публикует спецификацию протокола ODBC
1992	ANSI принимает стандарт SQL2 (SQL-92)
1992	Опубликован тест производительности TPC-C (OLTP)
1993	Первые поставки систем обслуживания хранилищ данных
1993	Первые поставки программных продуктов, поддерживающих протокол ODBC
1994	Коммерческие поставки серверов баз данных, поддерживающих параллельную обработку
1995	Первый выпуск СУРБД с открытым кодом MySQL
1996	Опубликован стандарт API-функций для доступа к базам данных OLAP и тест производительности OLAP-систем
1997	Компания IBM выпускает СУБД DB2 Universal Database, унифицировав ее архитектуру для работы на платформах других поставщиков
1997	Ведущие поставщики СУБД объявили о поддержке Java-технологий
1998	Компания Microsoft выпустила СУБД SQL Server 7, обеспечив поддержку корпоративных баз данных для платформы Windows NT
1998	Выпущена СУБД Oracle 8i, ознаменовавшая отход от архитектуры "клиент/сервер" и обеспечившая интеграцию баз данных с Интернетом
1998	Первый выпуск базы данных, полностью размещающейся в оперативной памяти
1999	J2EE стандартизовал JDBC-доступ к базам данных со стороны серверов приложений

Окончание табл 3.1

Год	Событие
1999	Принимается стандарт ANSI/ISO SQL:1999 с добавлением в язык объектно-ориентированных конструкций
2000	Oracle выпускает серверы приложений с интегрированным кешированием баз данных
2000	Microsoft выпускает SQL Server 2000, предназначенный для приложений уровня предприятия
2001	В основных СУРБД появляется возможность поддержки XML
2001	IBM покупает Informix
2002	IBM обходит Oracle и становится производителем баз данных №1
2003	Принят стандарт ANSI/ISO SQL:2003, в который добавлен SQL/XML
2006	Принят стандарт ANSI/ISO SQL:2006 с существенным расширением SQL/XML и объектно-ориентированных конструкций
2006	Исследования показывают, что Oracle лидирует на рынке
2008	Sun Microsystems покупает MySQL AB
2008	Принят стандарт ANSI/ISO SQL:2008

Первые годы

Статья доктора Кодда вызвала волну исследований в области реляционных баз данных, включая большой исследовательский проект компании IBM. Цель этого проекта, названного System/R, заключалась в том, чтобы доказать работоспособность реляционной модели и приобрести опыт реализации реляционной СУБД. Работа над проектом System/R началась в середине 70-х годов в лаборатории Санта-Тереза компании IBM в городе Сан-Хосе, штат Калифорния.

В 1974-1975 годах, на первом этапе выполнения проекта System/R, был создан минимальный прототип реляционной СУБД. Кроме разработки самой СУБД, в рамках проекта System/R проводилась работа над созданием языков запросов к базе данных. Один из этих языков был назван SEQUEL (Structured English Query Language — структурированный английский язык запросов). В 1976-1977 годах разработанный прототип проекта System/R был полностью переделан, и в 1978-1979 годах новая реализация проекта System/R была установлена на компьютерах нескольких заказчиков компании IBM для опытной эксплуатации. Эта эксплуатация принесла пользователям первый реальный опыт работы с СУБД System/R и ее языком базы данных, который по юридическим соображениям был переименован в SQL. В 1979 году исследовательский проект System/R завершился, и IBM сделала заключение, что реляционные базы данных не только вполне работоспособны, но и могут служить основой для создания коммерческих программных продуктов.

Первые реляционные СУБД

Проект System/R и созданный в его рамках язык работы с базами данных под названием SQL были подробно описаны в технических журналах 1970-х годов. Семинары по технологии баз данных характеризовались дебатами о достоинствах

новой "еретической" реляционной модели. Уже в 1976 году было ясно, что IBM стала энтузиастом реляционной технологии баз данных и прилагает значительные усилия для развития языка SQL.

Сообщения о проекте System/R привлекли внимание группы инженеров из города Менлоу Парк, штат Калифорния, которые решили, что исследования компании IBM предвещают значительный рынок сбыта для реляционных баз данных. В 1977 году они организовали компанию Relational Software, Inc., чтобы создать реляционную СУБД, основанную на SQL. Поставки этой СУБД, названной Oracle, начались в 1979 году. Oracle стала первой реляционной СУБД на компьютерном рынке. Она на целых два года опередила появление первой реляционной СУБД компании IBM и предназначалась для мини-компьютеров VAX компании Digital, которые были дешевле больших ЭВМ компании IBM. Эта компания агрессивно продвигала новый реляционный стиль управления базами данных и в конечном счете в качестве нового имени приняла название своей разработки. Сегодня Oracle Corporation является ведущим поставщиком реляционных СУБД с годовым оборотом свыше десяти миллиардов долларов.

Профессора из компьютерных лабораторий Калифорнийского университета (город Беркли) также исследовали реляционные базы данных в середине 1970-х годов. Подобно исследовательской группе компании IBM, они создали прототип реляционной СУБД и назвали свою систему Ingres. Проект Ingres включал в себя язык запросов QUEL, который был более "структурированным", но менее похожим на английский, чем язык SQL. Многие специалисты по базам данных, разработчики и основатели компаний, работающих в этой области, начинали свою деятельность с проекта Ingres.

В 1980 году несколько профессоров покинули Беркли и основали компанию Relational Technology, Inc., чтобы создать коммерческую версию системы Ingres, поставки которой на рынок начались в 1981 году. Ingres и Oracle сразу же вступили в острую конкурентную борьбу, но это соперничество лишь привлекло внимание к технологии реляционных баз данных. Несмотря на техническое превосходство во многих областях, Ingres вскоре стала сдавать свои позиции, не выдержав конкуренции с возможностями, предлагаемыми языком SQL, а также по причине агрессивной маркетинговой политики компании Oracle. В 1986 году первоначальный язык запросов QUEL был заменен на SQL. Это являлось свидетельством того, что стандарт SQL стал важным рыночным фактором. В середине 90-х годов технология Ingres была продана компании Computer Associates, ведущему производителю программного обеспечения для мэйнфреймов (которая продала свою долю в Ingres в 2005 году).

Продукты IBM

В то время как Oracle и Ingres становились коммерческими продуктами, компания IBM также предпринимала усилия по превращению проекта System/R в коммерческую разработку, получившую название SQL/Data System (SQL/DS). В 1981 году IBM объявила о создании СУБД SQL/DS, а в 1982 году начала ее поставки на рынок. В 1983 году IBM анонсировала версию SQL/DS для операцион-

ной системы VM/CMS, часто используемой на больших ЭВМ компании IBM в корпоративных информационных центрах.

В 1983 году IBM разработала еще одну реляционную СУБД для своих больших ЭВМ — Database 2 (DB2). Эта СУБД функционировала под управлением операционной системы MVS, которая являлась "рабочей лошадкой" в крупных центрах по обработке данных на мэйнфреймах. Поставки на рынок первой версии DB2 начались в 1985 году, и представители компании IBM назвали ее своим стратегическим программным продуктом. С этого времени DB2 стала флагманом реляционных СУБД компании IBM. Благодаря значительному влиянию IBM на рынок вычислительных систем, язык SQL этой СУБД фактически стал стандартом языка управления базами данных. Технология, реализованная в DB2, затем была использована в программных продуктах всех направлений компании IBM, от персональных компьютеров до сетевых серверов и мэйнфреймов. В 1997 году IBM пошла еще дальше, объявив о создании версий DB2 для компьютерных систем своих конкурентов — компаний Sun Microsystems и Hewlett-Packard. DB2 для мэйнфреймов остается центральным элементом стратегии IBM в области баз данных.

Коммерческое признание

В течение первой половины 1980-х годов поставщики реляционных баз данных боролись за коммерческое признание своих продуктов. По сравнению с традиционными архитектурами баз данных, реляционные программные продукты имели несколько недостатков. Производительность реляционных баз данных была ниже, чем традиционных. За исключением продуктов компании IBM, реляционные базы данных поставлялись на рынок мелкими, начинающими, поставщиками. И, опять же, за исключением продуктов IBM, реляционные базы данных предназначались для мини-компьютеров, а не для мэйнфреймов.

Однако у реляционных продуктов было большое преимущество. Реализованные в них языки реляционных запросов (SQL, QUEL и другие) позволяли выполнять запросы к базе данных без написания программ и немедленно получать результаты. В результате реляционные базы данных постепенно стали использоваться в качестве инструментов для поддержки принятия решений. В мае 1985 года компания Oracle с гордостью объявила о том, что количество инсталляций реляционных продуктов ее производства превысило одну тысячу. Ingres к тому времени также была инсталлирована на сравнимом количестве компьютеров. Постепенно начали получать признание и продукты DB2 и SQL/DS, общее число инсталляций которых тоже превысило тысячу.

Во второй половине 1980-х годов реляционные базы данных уже стали считаться технологией баз данных будущего. Резко увеличилась производительность реляционных баз данных. В частности, каждая новая версия Ingress и Oracle превосходила предшественницу в два-три раза. На увеличении этого показателя сказался и общий рост быстродействия компьютеров.

В конце 1980-х годов росту популярности SQL начали способствовать и рыночные тенденции. Компания IBM продвигала на рынок систему DB2 как лучшее решение 1990-х годов. Опубликование в 1986 году стандарта SQL, принятого ANSI/ISO, официально придало SQL статус стандартного языка баз данных. Кро-

ме того, SQL стал стандартом для компьютерных систем на базе UNIX, популярность которых также росла ускоренными темпами в конце 1980-х годов. По мере увеличения мощности персональных компьютеров и объединения их в локальные сети возникла необходимость в более сложных СУБД. Поставщики таких СУБД при создании систем нового поколения для персональных компьютеров взяли за основу SQL, а поставщики СУБД для мини-компьютеров, чтобы выдержать конкуренцию со стороны персональных компьютеров, вышли на зарождающийся рынок локальных вычислительных сетей.

В начале 1990-х годов усовершенствование реализаций SQL и резкое возрастание мощи процессоров сделали SQL практичным решением для приложений обработки транзакций. И наконец, SQL стал ключевой частью архитектуры "клиент/сервер", использующей персональные компьютеры, локальные сети и сетевые серверы для построения дешевых систем обработки информации. С развитием Интернета для SQL нашлась новая роль — в качестве языка баз данных для интернет-приложений и электронной коммерции.

Однако SQL не испытывал и недостатка в конкуренции. К началу 1990-х годов объектно-ориентированное программирование зарекомендовало себя как наиболее перспективный метод разработки приложений, особенно для персональных компьютеров и в области пользовательских графических интерфейсов. Объектная модель данных, со своими объектами, классами, методами и наследованием, плохо соотносилась с реляционной моделью таблиц, строк и столбцов данных. Ранние "объектные базы данных" включали Gemstone от Servio Logic, Gbase от Graphael и Vbase от Ontologic. В средине 1990-х годов выросло новое поколение компаний, рассчитывавших на то, что объектные базы данных вытеснят с рынка реляционные базы данных и их производителей, так же как в свое время SQL поступил с нереляционными базами данных. Здесь можно упомянуть такие продукты, как ITASCA от Itasca Systems, Jasmine от Fujitsu, Matisse от Matisse Software, Objectivity/DB от Objectivity, ONTOS от Ontos, Inc. (переименованного из Ontologic), O2 от O2 Technology, а также с полдесятка других. Но SQL и реляционная модель более чем устояли перед этим натиском. Некоторые из упомянутых продуктов остались на рынке и сегодня, но большинство было просто куплено или сгинуло во мгле времени. Например, O2 Technology постигла судьба некоторых других компаний, купленных Informix, а Informix, в свою очередь, позже был приобретен IBM. Общие годовые доходы объектно-ориентированных баз данных измеряются миллионами долларов, в то время как рынок SQL, СУРБД, соответствующих инструментов и служб оценивается в десятки миллиардов долларов в год.

По мере развития SQL этот язык стал применяться для решения множества задач, связанных с управлением данными, и постепенно, к концу 90-х годов, рынок баз данных перестал быть монолитным, разделившись на ряд специализированных сегментов. Одним из наиболее быстрорастущих среди них стал сегмент хранилищ данных, где базы данных применяются для анализа огромных объемов информации на предмет выявления скрытых тенденций и моделей развития. Другим направлением является внедрение в SQL новых типов данных (в частности, мультимедийных) и объектно-ориентированных принципов. Третий важный сегмент — это "мобильные" базы данных для переносных персональных компьютеров, взаи-

модействующие с централизованными базами данных как в оперативном, так и в автономном режиме. Еще одним сегментом оказываются базы данных, работающие с оперативной памятью, разрабатывающиеся как очень высокопроизводительные, ориентированные на работу с потоком базы данных, предназначенные для управления сетевыми потоками данных.

Несмотря на появление различных сегментов рынка, SQL по-прежнему остается их общим знаменателем. И через сорок лет после его возникновения позиции SQL так же сильны, как и ранее. SQL остается *стандартом* в области баз данных. Всегда будут появляться новые проблемы — например, сейчас это необходимость включения поддержки XML и его иерархической модели данных, а также необходимость поддержки огромных массивов данных в масштабе Интернета. Но история последних сорока лет дает все основания рассчитывать на то, что SQL и реляционная модель обладают изрядным запасом сил и способны справиться с новыми требованиями к управлению данными.

Стандарты SQL

Одним из наиболее важных шагов на пути к признанию SQL на рынке стало появление стандартов этого языка. Обычно при упоминании "стандарта SQL" имеют в виду официальный стандарт, утвержденный Американским национальным институтом стандартов (American National Standards Institute, ANSI) и Международной организацией по стандартизации (International Standards Organization, ISO). Однако существуют и другие важные стандарты, включая стандарт *де-факто*, каковым является SQL, реализованный в семействе продуктов DB2 компании IBM, и Oracle-диалект SQL, доминирующий на рынке.

Стандарты ANSI/ISO

Работа над официальным стандартом SQL началась в 1982 году, когда ANSI поставил перед своим комитетом X3H2 задачу по созданию стандарта языка управления реляционными базами данных. Вначале в комитете обсуждались преимущества различных предложенных языков. Однако, поскольку к тому времени на рынке стандартом де-факто стал SQL, комитет X3H2 остановил свой выбор на нем и занялся его стандартизацией.

Разработанный в результате стандарт в большей степени был основан на диалекте SQL системы DB2, хотя и содержал в себе ряд существенных отличий от этого диалекта. После нескольких доработок в 1986 году стандарт был официально утвержден как стандарт ANSI номер X3.135, а в 1987 году — в качестве стандарта ISO. Затем стандарт ANSI/ISO был принят правительством США как федеральный стандарт США в области обработки информации (Federal Information Processing Standard, FIPS). Этот стандарт, незначительно пересмотренный в 1989 году, обычно называют стандартом SQL1 или SQL-89.

Многие из членов комитетов ANSI и ISO представляли фирмы-поставщики различных СУБД, в каждой из которых был реализован собственный диалект SQL. Как и диалекты человеческого языка, диалекты SQL были, в основном, похожи

друг на друга, однако несовместимы в деталях. Во многих случаях комитет просто обошел существующие различия и не стандартизировал некоторые части языка, определив, что они реализуются по усмотрению разработчика. Этот подход позволил объявить большое число реализаций SQL совместимыми со стандартом, однако сделал сам стандарт относительно слабым.

Чтобы заполнить эти пробелы, комитет ANSI продолжил свою работу и создал проект нового, более жесткого, стандарта SQL2. В отличие от стандарта 1989 года, проект SQL2 предусматривал возможности, выходящие за рамки возможностей, уже реализованных в реальных коммерческих продуктах. А для следующего за ним стандарта SQL3 были предложены еще более глубокие изменения. Кроме того, была предпринята попытка официально стандартизировать те части языка, на которые давно существовали "собственные" стандарты в различных СУБД. В результате предложенные стандарты SQL2 и SQL3 оказались более противоречивыми, чем исходный стандарт. Стандарт SQL2 прошел процесс утверждения в ANSI и был окончательно принят в октябре 1992 года. В то время как первый стандарт 1986 года занимает не более ста страниц, стандарт SQL2 (официально называемый SQL-92) содержит около шестисот.

Существенным нововведением стало официальное утверждение трех уровней совместимости со стандартом SQL2. На самом нижнем, начальном, уровне (Entry Level) от СУБД требуются лишь минимальные дополнительные возможности в сравнении со стандартом SQL-89. Промежуточный уровень (Intermediate Level) представляет собой значительный шаг вперед по отношению к стандарту SQL-89, хотя и не затрагивает наиболее сложных и системно-зависимых аспектов языка SQL. Третий, самый высокий, уровень (Full Level) требует от СУБД реализации всех возможностей стандарта SQL2. В самом стандарте определение каждой возможности сопровождается описанием того, как она должна быть реализована на каждом из трех уровней. Сегодня специализированные базы данных, такие как используемые во встраиваемых приложениях или в приложениях с открытым кодом, в ряде областей обладают начальным уровнем соответствия стандарту SQL, но все основные базы данных уровня предприятия полностью поддерживают стандарт SQL-92.

После принятия SQL-92 работа над стандартами SQL шла в разных направлениях. Единый комитет разделился на ряд подкомитетов, каждый из которых работал над различными расширениями языка. Некоторые из них, например сохраняемые процедуры, уже имелись во многих коммерческих SQL-базах данных. Другие, такие как предложенные объектные расширения SQL, пока что не были широко доступны или полностью реализованы. Новые версии стандарта были выпущены в 1999, 2003, 2006 и 2008 годах. Стандарт 2006 года включал существенные дополнения к XML-части стандарта.

В процессе работы стандарт ANSI/ISO был разделен на 14 частей. Одни из них после некоторой активности в данном направлении оказались заброшены, некоторые вошли в другие части, а над остальными продолжается активная работа.

- **Часть 1 — SQL/Framework** содержит общие определения и служит "оглавлением" для других частей.
- **Часть 2 — SQL/Foundation** представляет собой наибольшую часть и содержит определения основных инструкций SQL для определения структуры базы данных и управления данными. Это потомок версий SQL-89

и SQL-92 стандарта. Данная часть существенно расширена путем включения структур для бизнес-анализа.

- **Часть 3 — SQL/CLI** (Call Level Interface, интерфейс уровня вызовов) описывает интерфейс уровня вызова процедур, хорошо известный как стандарт Microsoft ODBC. Он появился в 1995 году.
- **Часть 4 — SQL/PSM** (Persistent Stored Modules, постоянные хранимые модули) описывает процедурные расширения SQL, аналогичные возможностям, имеющимся в популярных процедурных SQL-языках наподобие PL/SQL в Oracle.
- **Часть 5 — SQL/Bindings** описывает встраивание SQL в другие процедурные языки. Эта часть была объединена с частью 2 в версии SQL:2003 стандарта.
- **Часть 6 — SQL/Transaction** была посвящена вопросам распределенных транзакций, но затем работа в данном направлении была остановлена.
- **Часть 7 — SQL/Temporal** была посвящена расширениям SQL для работы с календарными и временными данными, но затем работа в данном направлении была прекращена.
- **Часть 8 — SQL/Objects** в процессе работы над SQL3 содержала объектно-ориентированные расширения SQL. Эти расширения были внесены в часть 2 в стандарте SQL:1999.
- **Часть 9 — SQL/MED** (Management of External Data, управление внешними данными) добавляет в язык возможности по работе с нереляционными источниками данных; появилась в стандарте SQL:2003.
- **Часть 10 — SQL/OLB** (Object Language Bindings, связи с объектными языками) описывает обращение к SQL из языка программирования Java. Она связана с JDBC и SQL, встроенным в Java, и появилась в стандарте SQL:2003.
- **Часть 11 — SQL/Schemata** содержит стандарты для "каталога базы данных", или таблиц с самоописывающей базу данных системной информацией. Эта спецификация находилась в стандарте SQL:1999 в части 2, но была вынесена в отдельную часть в стандарте SQL:2003.
- **Часть 12 — SQL/Replication** изначально определяла стандарты копирования из одной SQL-базы данных в другую, но затем работа в данном направлении была прекращена.
- **Часть 13 — SQL/JRT** (Java Routines and Types, подпрограммы и типы Java) описывает подпрограммы и типы, используемые языком программирования Java для доступа к SQL-базам данных; впервые появилась в стандарте SQL:2003.
- **Часть 14 — SQL/XML** описывает интеграцию XML (Extensible Markup Language, расширяемый язык разметки) в язык SQL. Впервые появилась в стандарте SQL:2003 и с тех пор была значительно расширена.

С нескольких сотен страниц, описывающих базовые возможности языка SQL в 1986 году, стандарт ANSI/ISO SQL значительно вырос — как по объему, так и в смысле сложности и охвата различных тем. "Реальный" стандарт SQL, конечно, представляет собой SQL, реализованный в продуктах, занимающих важное место на рынке. Программисты и пользователи, как правило, предпочитают придерживаться тех частей языка, которые одинаковы у большинства продуктов. Большинство новых расширений SQL начинаются как новшества крупных производителей баз данных. Некоторые из них так и не получают поддержку и со временем исчезают из языка. Другие годами остаются в языке с единственной целью обеспечить обратную совместимость. Третьи же, наиболее удачные в коммерческом смысле, попадают "в струю", широко распространяются среди баз данных и в конечном счете оказываются в официальном стандарте.

Другие ранние стандарты SQL

Хотя стандарт ANSI/ISO наиболее широко распространен, он не был единственным стандартом SQL во времена его становления. Европейская группа поставщиков X/OPEN также приняла SQL в качестве одного из своих стандартов для среды переносимых приложений на основе UNIX. Стандарты группы X/OPEN играли важную роль на европейском компьютерном рынке, где ключевой задачей являлась переносимость приложений между компьютерными системами различных производителей.

Компания IBM также включила SQL в свою спецификацию Systems Application Architecture (архитектуры прикладных систем) в 1990-х годах и пообещала, что все ее продукты в конечном счете будут переведены на этот диалект SQL. Хотя данная спецификация и не оправдала надежд на унификацию линии продуктов компании IBM, движение в сторону унификации IBM SQL продолжается. Система DB2 остается основной СУБД компании IBM для мэйнфреймов, однако компания выпустила реализацию DB2 и для OS/2, собственной операционной системы для персональных компьютеров, и для линии серверов и рабочих станций RS/6000, работающих под управлением UNIX. Распространение DB2 (не только на разные аппаратные системы, но и для разных типов данных) воплотилось в названии одной из последних реализаций DB2 — Universal Database (универсальная база данных, UDB).

ODBC и консорциум SQL Access Group

В технологии баз данных существует важная область, которую не затрагивали ранние официальные стандарты. Это *взаимодействие баз данных* — методы, с помощью которых различные базы данных могут обмениваться информацией, обычно по сети. В 1989 году несколько производителей программного обеспечения сформировали консорциум SQL Access Group специально для решения этой проблемы. В 1991 году консорциум опубликовал спецификацию RDA (Remote Database Access, удаленный доступ к базам данных). К сожалению, эта спецификация была тесно связана с сетевыми протоколами OSI, которые проиграли сетевую битву протоколам TCP/IP, поэтому она никогда не была широко реализована.

Второй стандарт от SQL Access Group повлиял на рынок существенно сильнее. В результате настойчивых требований компании Microsoft консорциум SQL Access Group обратил свое внимание на интерфейс уровня вызовов. Спецификация CLI (Call Level Interface, интерфейс уровня вызовов), основанная на разработках компании Microsoft, увидела свет в 1992 году. В этом же году был опубликован и протокол ODBC (Open Database Connectivity, открытый доступ к базам данных) компании Microsoft, основанный на спецификации CLI. Благодаря рыночному влиянию Microsoft и благословению, полученному "открытым стандартом" от SQL Access Group, ODBC оказался стандартом де-факто для интерфейсов доступа к SQL-базам данных на персональных компьютерах. Весной 1993 года компании Apple и Microsoft объявили о соглашении относительно поддержки ODBC в Macintosh и Windows, что закрепило за этим протоколом статус промышленного стандарта в обеих популярных средах с графическим пользовательским интерфейсом. Вскоре появилась реализация ODBC для платформы UNIX. В 1995 году, с публикацией стандарта SQL/Call-Level Interface, интерфейс ODBC стал стандартом ANSI/ISO.

В течение последних десяти лет ODBC продолжает развиваться, но существенно медленнее. Microsoft продолжает поддерживать ODBC, но основные усилия перенесены в область создания более высокоуровневых, объектно-ориентированных интерфейсов для универсального доступа к базам данных. Тем не менее ODBC продолжает играть главную роль в обеспечении переносимости баз данных для приложений уровня предприятия и инструментария баз данных. Достаточно распространена ситуация, когда инструментарий базы данных или приложение уровня предприятия поддерживает "драйвер", оптимизирующий непосредственный доступ к базам данных Oracle, DB2 или SQL Server с помощью их специфических интерфейсов уровня вызова. Такое приложение обычно включает дополнительный драйвер, использующий в качестве способа поддержки широкого диапазона других баз данных ODBC. Поскольку этот подход принят очень многими приложениями и инструментами, практически все производители баз данных предоставляют доступ с применением ODBC, иногда в качестве основного интерфейса уровня вызовов, а иногда как дополнение к высокопроизводительному интерфейсу, специфичному для данной базы данных.

JDBC и серверы приложений

Стремительный рост популярности Интернета привел к дальнейшему развитию стандартов обращения к базам данных, с тем чтобы они поддерживали работу с объектно-ориентированным языком программирования Java. Java стал, по сути, стандартным языком для создания интернет-приложений, работающих на серверах приложений на базе Java. Компания Sun Microsystems, разработчик Java, приложила немалые усилия по стандартизации применения Java для серверов приложений в спецификации Java2 Enterprise Edition (J2EE). J2EE включает Java Database Connectivity (JDBC) в качестве стандарта доступа к реляционным базам данных из Java. В отличие от доступа к базам данных из языка программирования C, где ODBC многие годы предшествовали интерфейсы уровня вызовов конкретных баз данных, стандарт JDBC был разработан относительно рано, на пике роста популярности Java. В результате частные интерфейсы для подключения к Java так и не появились, и *единственным* стандартом SQL-доступа со стороны Java стал JDBC.

SQL и переносимость

Появление стандарта SQL вызвало довольно много восторженных заявлений о переносимости SQL и использующих его приложений. Для иллюстрации того, как любое приложение, используя SQL, может работать с любой СУБД на основе SQL, часто приводят диаграммы, подобные изображенной на рис. 3.1. На самом деле различия между существующими диалектами SQL достаточно значительны, так что при переводе приложения под другую СУБД его, как правило, приходится модифицировать. Со временем ядро языка становится все более стандартным, но в то же время производители баз данных добавляют в язык новые возможности, часто с расширениями, специфичными для конкретной базы данных. Рассмотрим примеры областей, в которых возникают такие отличия.

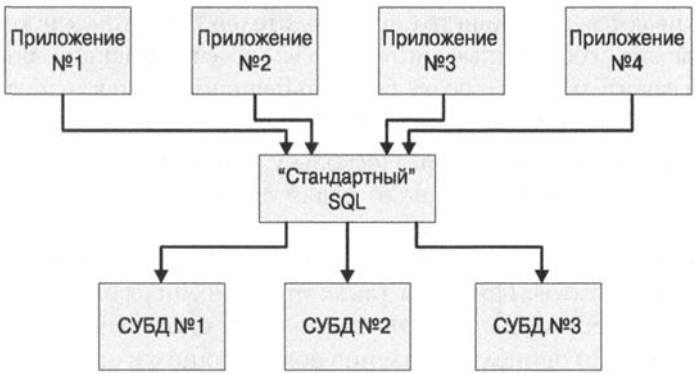


Рис. 3.1. Миф о переносимости SQL

- **Типы данных**. В стандарте SQL определен достаточно широкий набор типов данных, однако производители постоянно добавляют новые типы. Даже старые типы данных могут помешать переносимости — например, специфичный для Oracle тип данных `NUMBER` широко используется для представления числовых данных в базах данных Oracle, — и при этом только в них.
- **Обратная совместимость**. Не такая уж редкость встретить приложение уровня предприятия, работающее один-два десятка лет после того, как оно было написано, когда все программисты, создававшие его, давно уволены (или уволились). Такие программы становятся "неприкасаемыми", поскольку детальное знание о том, как они функционируют, оказывается утраченным. Большие разделы этих программ могут зависеть от старых, специфичных для данной базы данных, свойств SQL, так что производители баз данных вынуждены поддерживать обратную совместимость, иначе имеется риск, что старые приложения перестанут работать. Такие увековеченные отличия диалектов препятствуют переносимости.
- **Системные таблицы**. В стандарте SQL описание системных таблиц, в которых содержится информация о структуре самой базы данных, появилось только в версии SQL-92. Поэтому каждый производитель создавал собствен-

ные системные таблицы, которые продолжают развиваться и эволюционировать и зачастую содержат информацию, которая выходит за указанные в стандарте пределы. Приложения, использующие такие специфичные для конкретной базы данных системные таблицы, непереносимы.

- **Программный интерфейс**. В раннем стандарте SQL определен абстрактный способ использования SQL в приложениях, написанных на таких языках программирования, как COBOL, C, FORTRAN и другие, который не был широко принят сообществом программистов. Стандарт SQL/CLI 1995 года окончательно определил программный доступ к SQL, но к тому времени коммерческие СУБД уже популяризовали собственные интерфейсы и внедрили их в сотни тысяч пользовательских приложений и прикладных пакетов. Хотя в настоящее время стандартные API и широко поддерживаются, большинство производителей баз данных продолжают предоставлять собственные интерфейсы, обеспечивающие более высокую производительность и более богатую функциональность, побочным действием которых является привязка к конкретной базе данных.
- **Семантические отличия**. Поскольку некоторые элементы определены в стандартах как зависящие от реализации, может возникнуть ситуация, когда в результате выполнения одного и того же запроса в двух отвечающих стандарту реализациях SQL будут получены два различных набора результатов. Примеры таких отличий могут быть найдены, например, в обработке значений `NULL`, в разных статистических функциях и в несовпадении процедур удаления повторяющихся строк.
- **Репликация и зеркальное копирование данных**. Многие промышленные базы данных содержат таблицы, которые дублируются в двух или большем количестве географически разнесенных баз данных, чтобы обеспечить высокую степень доступности или восстановления после аварий, для распределения нагрузки или снижения задержек при работе в сети. Методы, используемые для определения таких схем репликации и управления ими, у каждой базы данных свои, и от попыток стандартизовать процессы репликации пришлось отказаться.
- **Коды ошибок**. Коды, возвращаемые инструкциями SQL при возникновении ошибок, появились в стандарте SQL-92, но все популярные СУБД к настоящему времени давно используют собственные коды ошибок. Даже при использовании режима со стандартными кодами ошибок расширения в конкретных базах данных могут генерировать собственные коды, выходящие за рамки, определенные стандартом.
- **Структура базы данных**. В стандарте SQL-89 определен язык SQL, который используется уже после того, как база данных открыта и подготовлена к работе. Детали наименования баз данных и установки первоначального подключения к тому времени уже сильно отличались. Стандарт SQL-92 в некоторой степени унифицирует этот процесс, но не может полностью скрыть детали реализации.

Несмотря на перечисленные отличия, в начале 1990-х годов стали появляться (и популярны по сей день) коммерческие инструменты для работы с базами данных, реализующие переносимость между рядом различных СУБД. На практике такие программы всегда включают специальные драйвера для работы с определенными СУБД. Эти драйвера генерируют код в соответствии с определенным диалектом SQL, выполняют преобразования типов данных, трансляцию кодов ошибок и т.д.

SQL и сети

Резкий рост компьютерных сетей в 1990-х годах оказал большое влияние на управление базами данных и придал SQL новые возможности. По мере распространения сетей, приложения, которые традиционно работали на центральном мини-компьютере или мэйнфрейме, переводятся на серверы и рабочие станции ЛВС. В таких сетях SQL играет важнейшую роль и связывает приложение, выполняющееся на рабочей станции с графическим пользовательским интерфейсом, и СУБД, управляющую совместно используемыми данными на сервере. Рост популярности Интернета и Веб еще больше усилил влияние SQL в сфере сетевых технологий. С появлением трехуровневой архитектуры Интернета язык SQL стал связующим звеном между прикладной логикой (работающей на среднем уровне, сервере приложений или веб-сервере) и базой данных (третий уровень). В следующих подразделах мы поговорим о развитии архитектур сетевого управления базами данных и о роли, которую SQL играет в каждой из них.

Централизованная архитектура

На рис. 3.2 изображена традиционная централизованная архитектура баз данных, использовавшаяся в DB2 и первоначальных базах данных для мини-компьютеров, таких как Oracle и Ingres. В этой архитектуре и СУБД, и сами физические данные размещаются на центральном мини-компьютере или мэйнфрейме вместе с приложением, принимающим входную информацию с пользовательского терминала и отображающим на нем же данные. Прикладная программа "общается" с СУБД с помощью SQL.

Рис. 3.2. Управление базами данных в централизованной архитектуре

Предположим, что пользователь вводит запрос, который требует последовательного просмотра базы данных (например, запрос на вычисление среднего значения суммы сделок по всем заказам). СУБД получает этот запрос, просматривает базу данных, выбирая с диска каждую запись, вычисляет среднее значение и отображает результат на экране. Приложение и СУБД работают на одном компьютере, так что этот тип запросов (как и всех прочие) выполняется весьма эффективно.

Недостатки централизованной архитектуры проявляются при масштабировании. Поскольку система обслуживает много различных пользователей, каждый из них ощущает снижение быстродействия по мере увеличения нагрузки на систему.

Архитектура файлового сервера

Появление персональных компьютеров и локальных вычислительных сетей привело к разработке архитектуры *файлового сервера* (рис. 3.3). При такой архитектуре приложение, выполняемое на персональном компьютере, может получить прозрачный доступ к файловому серверу, на котором хранятся совместно используемые файлы. Когда приложение, работающее на персональном компьютере, запрашивает данные из такого файла, сетевое программное обеспечение автоматически считывает требуемый блок данных с сервера. Архитектура файловых серверов поддерживалась первыми СУБД для персональных компьютеров, такими как dBASE и позднее Microsoft Access, при этом на каждом персональном компьютере работала своя копия СУБД.

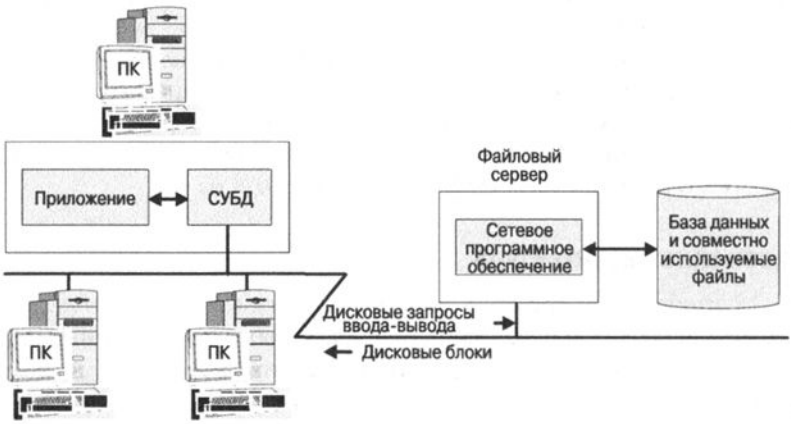


Рис. 3.3. Управление базами данных в архитектуре файлового сервера

При выполнении типичных запросов, когда требуется получение одной или небольшого количества строк из базы данных, эта архитектура обеспечивает великолепную производительность, поскольку в распоряжении каждой копии СУБД находятся все ресурсы персонального компьютера. Однако рассмотрим запрос из приведенного выше примера. Поскольку запрос требует последовательного просмотра базы данных, СУБД постоянно запрашивает новые записи из базы данных, которая физически расположена на сервере. В конечном счете СУБД запросит и получит по сети *все* блоки файла. Очевидно, что при выполнении запросов такого типа эта архитектура создает слишком большую нагрузку на сеть и уменьшает производительность работы.

Архитектура "клиент/сервер"

На рис. 3.4 изображена очередная стадия эволюции сетевых баз данных — архитектура *клиент/сервер*. В этой схеме персональные компьютеры объединены

в локальную сеть, в которой имеется *сервер баз данных*, хранящий общие базы данных. Функции СУБД разделены на две части: пользовательские программы, такие как приложения для формирования интерактивных запросов, генераторы отчетов и прикладные программы, выполняются на клиентском компьютере, а ядро базы данных, которое хранит данные и управляет ими, работает на сервере. В этой архитектуре, популярность которой существенно возросла в течение 1990-х годов, SQL стал стандартным языком, обеспечивающим взаимодействие между пользовательскими программами и ядром базы данных.

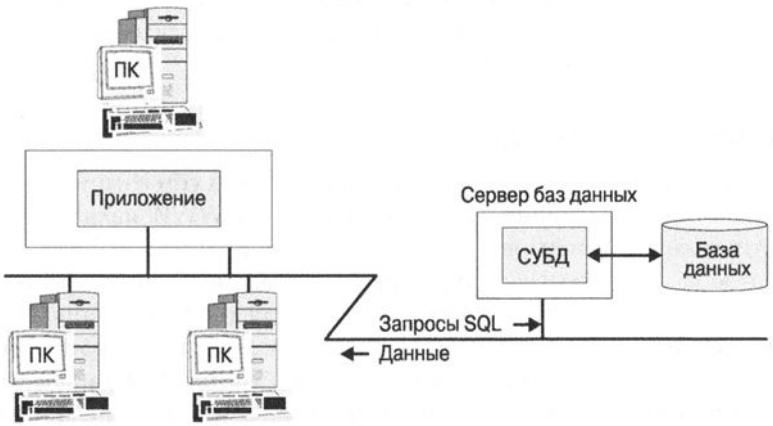


Рис. 3.4. Управление базами данных в архитектуре "клиент/сервер"

Давайте еще раз рассмотрим пример с вычислением средней стоимости заказа. При архитектуре "клиент/сервер" запрос передается по сети на сервер баз данных в виде SQL-запроса. Ядро базы данных на сервере обрабатывает запрос и просматривает базу данных, которая также расположена на сервере. После вычисления результата ядро базы данных посылает его обратно по сети клиентскому приложению, которое отображает его на экране персонального компьютера.

Архитектура "клиент/сервер" снижает сетевой трафик и распределяет процесс загрузки базы данных. Функции для работы с пользователем, такие как обработка ввода и отображение данных, выполняются на персональном компьютере пользователя. Функции для работы с данными, такие как дисковый ввод-вывод и выполнение запросов, выполняются сервером баз данных. Наиболее важно то, что SQL обеспечивает четко определенный интерфейс между клиентской и серверной системами, эффективно передавая запросы на доступ к базе данных.

Преимущества данной архитектуры сделали ее наиболее популярной схемой при разработке новых приложений в середине 1990-х годов. Все ведущие СУБД — Oracle, Informix, Sybase, SQL Server, DB2 и многие другие — стали предлагать клиент-серверные возможности. Многие компании начали выпускать средства разработки приложений "клиент/сервер". Некоторые из них разрабатывались производителями баз данных, иные — сторонними производителями.

У архитектуры "клиент/сервер", как и у всех остальных, есть свои недостатки. Наиболее серьезный из них — проблема управления прикладным программным обеспечением, расположенным на тысячах персональных компьютеров, а не на од-

ной центральной машине. Обновление какого-либо приложения одновременно на тысяче персональных компьютеров в крупной компании требовало от его информационного подразделения огромных усилий. Все становилось еще хуже, если изменения в прикладной программе должны были быть синхронизированы с изменениями в других приложениях или в самой СУБД. Кроме того, пользователи часто самостоятельно инсталлировали вспомогательное программное обеспечение и настраивали установленные программы на свой манер, что, безусловно, в значительной степени усложняло задачу администрирования. Компании разрабатывали специальные стратегии борьбы с этими проблемами, но все равно к концу 1990-х годов возникла необходимость пересмотреть концепции управления приложениями "клиент/сервер" в больших распределенных системах.

Многоуровневая архитектура

С развитием Интернета и особенно Веб архитектура сетевого управления базами данных получила дальнейшее развитие. Поначалу WWW являлась средой просмотра статических документов и развивалась независимо от рынка СУБД. Но когда веб-браузеры получили широкое распространение, разработчики пришли к выводу, что это очень удобный способ обеспечения доступа к корпоративным базам данных. Предположим, к примеру, что торговая компания располагает собственным веб-сайтом, на котором клиенты могут найти информацию о товарах, выпускаемых компанией, включая текстовое и графическое их описание. Естественным следующим шагом будет предоставление клиентам доступа к информации о наличии выбранного товара на складе, причем посредством того же интерфейса веб-браузера. Для этого требуется связать последний с базой данных, хранящей такую (постоянно меняющуюся) информацию.

Методы связывания веб-серверов и СУБД стремительно развивались в конце 1990-х–начале 2000-х годов, и в итоге это вылилось в трехуровневую сетевую архитектуру, показанную на рис. 3.5. Интерфейсом пользователя является веб-браузер, работающий на персональном компьютере или некотором другом "тонком клиенте", например таком, как смартфон. Браузер взаимодействует с веб-сервером на прикладном уровне. Если пользователь запрашивает нечто большее, чем просто статические веб-страницы, веб-сервер переадресует запрос серверу приложений, роль которого заключается в применении бизнес-логики, необходимой для обработки запроса. Зачастую запрос включает обращение к старой системе, работающей на мэйнфрейме, либо к корпоративной базе данных. Это серверный уровень модели.

Как и в случае архитектуры "клиент/сервер", SQL является стандартным языком баз данных для обмена информацией между сервером приложений и серверами баз данных. Все программные продукты прикладного уровня для обращения к базам данных оснащены API на основе SQL. Поскольку большинство коммерческих серверов приложений использует стандарт Java2 Enterprise Edition (J2EE), ведущим API для доступа сервера приложений к базе данных является Java Database Connectivity (JDBC).

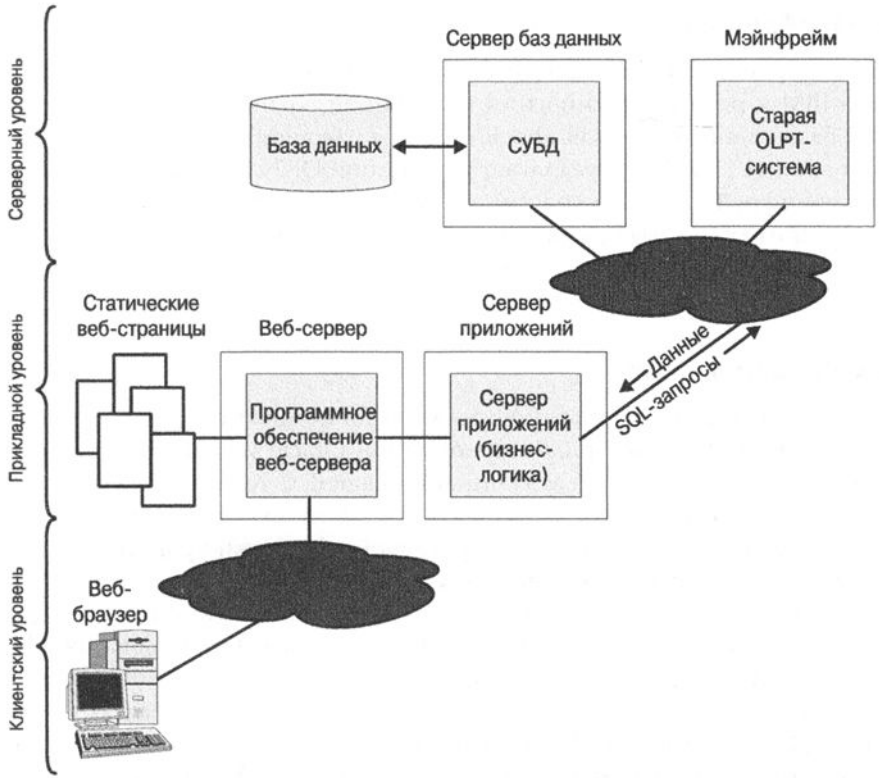


Рис. 3.5. Управление базами данных в трехуровневой интернет-архитектуре

Влияние SQL

Будучи стандартным языком доступа к реляционным базам данных, SQL оказывает большое влияние на все сегменты компьютерного рынка. SQL-база данных IBM DB2 доминирует в области управления данными на мэйнфреймах. База данных Oracle контролирует рынок компьютерных систем и серверов под управлением UNIX. SQL Server от Microsoft — ведущая SQL-база данных для серверных операционных систем Windows для рабочих групп и ведомственных приложений. MySQL доминирует на рынке баз данных с открытым кодом. SQL принят в качестве технологии для оперативной обработки транзакций (online transaction processing, OLTP), что полностью опровергает мнение 1980-х годов о том, что реляционные базы данных никогда не станут достаточно производительны для применения в приложениях обработки транзакций. Хранилища данных и приложения для извлечения информации на базе SQL используются для изучения пользовательского спроса, с тем чтобы предоставлять продукты и услуги, в большей степени соответствующие требованиям рынка. В Интернете базы данных на основе SQL служат основой для более персонализированных продуктов, служб и информационных сервисов, что является ключевым преимуществом электронной коммерции.

SQL и мэйнфреймы

Хотя иерархическая база данных IMS от IBM все еще предлагается для мэйнфреймов IBM и работает со многими высокопроизводительными приложениями, ведущей базой данных для мэйнфреймов уже более чем два десятилетия является SQL-база данных DB2. IBM предлагает реализации DB2 для разных архитектур, но DB2 для мэйнфреймов можно рассматривать как "плавбазу для всей эскадры" баз данных IBM. Любые новые разработки баз данных для мэйнфреймов используют DB2, укрепляя доминирующую роль SQL в области обработки данных на мэйнфреймах.

SQL и мини-компьютеры

Сегмент рынка реляционных СУБД для мини-компьютеров начал развиваться одним из первых. Первые продукты компаний Oracle и Ingres предназначались для мини-компьютеров VAX/VMS компании Digital. С тех пор обе СУБД были перенесены на множество других платформ. СУБД компании Sybase, появившаяся позднее и предназначавшаяся для оперативной обработки транзакций, работала на нескольких платформах, включая VAX.

Кроме того, на протяжении 1980-х годов поставщики мини-компьютеров разрабатывали собственные реляционные СУБД на основе SQL. Компания Digital на каждую систему VAX/VMS устанавливала собственную СУБД Rdb/VMS. Компания Hewlett-Packard поставляла Allbase — СУБД, поддерживающую как HPSQL (диалект SQL от компании Hewlett-Packard), так и нереляционный интерфейс. Компания Data General поменяла свои старые нереляционные базы данных на СУБД DG/SQL. К тому же многие из поставщиков мини-компьютеров перепродавали реляционные СУБД независимых поставщиков. Все это помогло языку SQL утвердиться в качестве важной технологии в системах среднего уровня.

Со средины 1990-х годов SQL-продукты поставщиков мини-компьютеров, в основном, исчезли, уступив место мультиплатформенным разработкам компаний Oracle, Informix, Sybase и др. Oracle приобрела Rdb компании Digital; другие продукты постепенно оказались заброшены. Одновременно с этим угасло и влияние специализированных операционных систем для мини-компьютеров — всех их заменила операционная система UNIX. Вчерашний рынок реляционных продуктов для мини-компьютеров стал сегодняшним рынком серверов баз данных для платформы UNIX.

SQL и UNIX

SQL был однозначно признан лучшим решением в области управления данными для компьютерных систем на базе платформы UNIX. Операционная система UNIX, которая изначально была разработана в Bell Laboratories, в 1980-х годах стала стремительно завоевывать популярность как независимая от производителя стандартная операционная система. Она работает на разнообразных компьютерных системах, начиная от рабочих станций и заканчивая мэйнфреймами, и стала стандартной операционной системой для высококачественных серверных систем, включая серверы баз данных.

В начале 1980-х годов были доступны четыре большие СУБД для UNIX-систем. Две из них, производства компаний Oracle и Ingres, были UNIX-версиями продуктов для мини-компьютеров компании DEC. Две другие СУБД, Informix и Unify, были созданы специально для UNIX. Вначале ни одна из них не предлагала поддержку SQL, но к 1985 году Unify ввела эту поддержку в свою СУБД, а Informix полностью переписала свою СУБД, предложив Informix-SQL с полной поддержкой SQL.

На сегодняшний день в данном сегменте рынка лидирует СУБД Oracle, доступная для всех ведущих серверных платформ UNIX. Компания Informix была приобретена IBM, которая продолжает предлагать продукты для своих собственных и чужих UNIX-систем. Серверы баз данных на базе UNIX (и все в большей степени на базе Linux) являются ведущим звеном в архитектуре "клиент/сервер" и в трехуровневой архитектуре Интернета. Требования к повышению производительности реляционных баз данных являлись одной из основных движущих сил на пути развития аппаратного обеспечения для платформы UNIX. Сюда можно отнести принятие симметричной мультипроцессорности (Symmetric Multiprocessing, SMP) в качестве базовой серверной архитектуры, разработку многоядерных микропроцессоров (что можно рассматривать как SMP на уровне микросхемы), а также использование RAID (Redundant Array of Independent Disk, избыточный массив недорогих дисков), технологии, обеспечивающей резкое повышение производительности ввода-вывода.

SQL и персональные компьютеры

С появлением первых моделей IBM PC базы данных стали приобретать популярность на рынке персональных компьютеров. СУБД dBASE компании Ashton-Tate была инсталлирована более чем на миллионе персональных компьютеров, работавших под управлением MS DOS. Хотя в большинстве СУБД для персональных компьютеров данные хранились в табличной форме, эти СУБД не обладали полной мощью реляционных баз данных и не поддерживали SQL. Первые СУБД для персональных компьютеров представляли собой соответствующим образом переработанные версии известных СУБД для мини-компьютеров и с трудом "умещались" на персональных компьютерах. Например, система Professional Oracle для IBM PC, анонсированная в 1984 году, требовала двух мегабайтов памяти — намного больше типичных для того времени 640 Кбайт.

С появлением в апреле 1987 года операционной системы OS/2, созданной компаниями IBM и Microsoft, начался рост популярности SQL на персональных компьютерах. Кроме стандартной версии OS/2, компания IBM выпустила собственную расширенную редакцию OS/2 (OS/2 Extended Edition, OS/2 EE) со встроенной поддержкой SQL-базы данных и коммуникаций. Сделав SQL частью операционной системы, компания IBM тем самым вновь подтвердила свою приверженность этому языку.

Появление OS/2 EE стало проблемой для компании Microsoft. Поскольку она была разработчиком стандартной OS/2 и продавала ее другим производителям персональных компьютеров, потребовалась альтернатива OS/2 EE. Ответом Microsoft стала покупка лицензии на СУБД компании Sybase, разработанной для VAX, и перенос этой СУБД в систему OS/2. В январе 1988 года Microsoft и Ashton-Tate (в то время — лидер рынка баз данных для персональных компьютеров со

своей dBASE) неожиданно объявили, что они будут совместно продавать новую СУБД, получившую название SQL Server. Компания Microsoft станет поставлять SQL Server вместе с OS/2 производителям компьютеров, а компания Ashton-Tate займется распространением SQL Server по розничным каналам пользователям персональных компьютеров. В сентябре 1989 года компания Lotus Development (еще один член большой тройки производителей программного обеспечения для персональных компьютеров того времени) внесла свой вклад в SQL Server, сделав инвестицию в компанию Sybase. В том же году компания Ashton-Tate отказалась от исключительных прав на распространение этой СУБД и продала свою долю компании Lotus.

Хотя успех СУБД SQL Server для OS/2 был ограниченным (как, к сожалению, и успех самой весьма неплохой операционной системы), компания Microsoft в типичном для нее стиле продолжала вкладывать деньги в эту СУБД и перенесла ее на платформу Windows NT. В течение некоторого времени компании Microsoft и Sybase оставались партнерами: первая сосредоточила свои усилия на рынке персональных компьютеров, ЛВС и Windows NT, а вторая — на рынке мини-компьютеров и UNIX-серверов. Но по мере того как Windows NT и UNIX становились конкурентами в качестве платформ для серверов баз данных, взаимоотношения между компаниями начали переходить из области сотрудничества в область конкуренции. В конце концов пути компаний окончательно разошлись. Теперь они предлагают совершенно разные продукты, хотя некоторые сходные SQL-расширения (например, хранимые процедуры) выдают в них общее прошлое.

На сегодняшний день СУБД SQL Server является ведущей СУБД для серверов под управлением Windows. Новые версии этой СУБД выходят каждые два-три года, причем с большими изменениями и дополнениями в таких областях, как работа с XML, специальные данные, полнотекстовый поиск, хранилища и анализ данных и высокая доступность. В то время как на очень больших серверах баз данных доминируют UNIX и Oracle, благодаря SQL Server серверы под управлением Windows смогли занять нишу систем среднего уровня.

SQL и обработка транзакций

В процессе своего развития SQL и реляционные базы данных почти не применялись в приложениях, предназначенных для оперативной обработки транзакций (Online Transaction Processing, OLTP). Поскольку в реляционных базах данных упор делается на запросы, такие базы данных традиционно использовались в приложениях, служащих для поддержки принятия решений, и в приложениях с малым объемом транзакций, где их низкое быстродействие не было недостатком. В области оперативной обработки транзакций, когда требовалось обеспечить одновременный доступ к данным сотням пользователей и время ожидания каждого из них не должно было превышать доли секунды, доминировала нереляционная СУБД IMS (Information Management System, система управления информацией) компании IBM.

В 1986 году компания Sybase, новичок на рынке СУБД, представила реляционную СУБД, предназначенную специально для оперативной обработки транзакций. СУБД компании Sybase работала на мини-компьютерах VAX/VMS и рабочих стан-

циях Sun и обеспечивала уровень быстродействия, необходимый для обработк и больших объемов транзакций. Вскоре вслед за этим компании Oracle Corporation и Relational Technology объявили, что также выпустят версии своих продуктов Oracle и Ingres для оперативной обработки транзакций. На рынке UNIX-систем компания Informix анонсировала OLTP-версию своей СУБД под названием Informix-Turbo.

В апреле 1988 года компания IBM присоединилась к поставщикам реляционных СУБД для OLTP, выпустив систему DB2 Version 2. Тесты показали, что на больших мэйнфреймах эта система могла обрабатывать до 250 транзакций в секунду. Компания IBM утверждала, что теперь быстродействие DB2 позволяет использовать ее во всех OLTP-приложениях, кроме наиболее требовательных к быстродействию, и поощряла клиентов устанавливать ее вместо IMS. После этого тесты OLTP стали стандартным маркетинговым инструментом для реляционных СУБД, вопреки серьезным сомнениям в том, насколько они отражают быстродействие реальных приложений.

Развитие реляционных технологий и появление более мощных компьютеров привели к тому, что роль SQL в OLTP-приложениях резко возросла, как возросли и показатели производительности СУБД. Теперь поставщики СУБД начали позиционировать свои продукты в зависимости от показателей OLTP-производительности, и на несколько лет рынок баз данных погрузился в войну тестов производительности. Независимая организация Transaction Processing Council (TPC, совет по средствам обработки транзакций) опубликовала серию собственных тестов (TPC-A, TPC-B и TPC-C), чем только подогрела интерес к данной области со стороны поставщиков.

В начале 2000-х годов реляционные базы данных на базе SQL на UNIX-серверах высокого класса преодолели рубеж 1000 транзакций в секунду. Системы "клиент/сервер", включающие реляционные СУБД, стали признанной архитектурой для реализации OLTP-приложений. Таким образом, рассматриваемый ранее как "непригодный для оперативной обработки транзакций", язык SQL вырос до промышленного стандарта в области построения OLTP-приложений.

SQL и базы данных для рабочих групп

Стремительный рост популярности локальных сетей на базе персональных компьютеров в 1980-е–90-е годы создал предпосылки для распространения новой архитектуры систем управления базами данных масштаба подразделений, или "рабочих групп". Первоначально СУБД, предназначенные для этого сегмента рынка, работали под управлением операционной системы OS/2 компании IBM. Даже СУБД SQL Server, ныне играющая ключевую роль в стратегии компании Microsoft, связанной с платформой Windows, в свое время дебютировала как продукт для OS/2. В середине 90-х годов фирма Novell тоже изо всех сил старалась сделать свою сетевую операционную систему NetWare привлекательной платформой для серверов баз данных рабочих групп. На заре эры локальных сетей NetWare прочно утвердилась в качестве доминирующей операционной системы для файл/серверов и серверов печати. Объединившись с Oracle и другими разработчиками, Novell рассчитывала распространить свое лидерство и на серверы рабочих групп.

Однако появление операционной системы Windows NT, специализированной версии Windows для применения на серверах, вызвало настоящий переворот. Если в качестве файл/сервера NetWare явно превосходила NT по производительности,

то NT имела более надежную и универсальную архитектуру, во многом подобную архитектуре операционных систем для мини-компьютеров. Microsoft успешно позиционировала NT как более привлекательную платформу для работы приложений рабочих групп (в качестве сервера приложений) и баз данных рабочих групп. SQL Server компании Microsoft продавался (а часто просто встраивался) с NT в качестве тесно интегрированной платформы разработки приложений для рабочих групп. Поначалу информационные отделы компаний отнеслись к этой пока еще новой и непроверенной технологии с большой осторожностью, но предложение было слишком уж заманчивым: комбинация Windows NT и SQL Server позволяла различным подразделениям разрабатывать собственные небольшие проекты, не прибегая к помощи центрального информационного отдела компании. Поэтому она не только была принята, но и стимулировала интенсивный рост всего сегмента рынка баз данных для рабочих групп.

Сегодня SQL стал общепризнанным стандартом в качестве языка работы с базами данных для рабочих групп. В дополнение к Windows, популярной платформой для серверов рабочих групп стал Linux. Большая часть рынка поделена между Microsoft SQL Server и Oracle, но в последнее время достаточно сильным (и дешевым) конкурентом для них начинает становиться база данных с открытым кодом MySQL. Еще одним участником гонки можно назвать другую базу данных с открытым кодом — Postgres, разработанную в Калифорнийском университете в Беркли.

SQL, хранилища данных и интеллектуальные ресурсы предприятия

Несколько лет усилий по внедрению технологии SQL в сферу разработки OLTP-приложений сделали свое дело, и реляционные базы данных стали цениться не только за удобство выполнения запросов и поддержку принятия решений. Тесты производительности и отчаянное соперничество между ведущими СУБД переместились в область простых транзакций, таких как добавление новых заказов в базу данных или определение баланса счетов клиента. Благодаря мощи реляционной модели те же базы данных, которые используются для выполнения текущих деловых операций, могут служить и для анализа растущих объемов накапливаемых данных. На профессиональных конференциях менеджеров информационных служб предприятий часто можно было услышать о том, что накопленные деловые данные (разумеется, хранящиеся в реляционных базах данных) должны рассматриваться как один из ценнейших активов предприятия и служить повышению эффективности принятия деловых решений.

Хотя теоретически реляционные базы данных без труда можно использовать и для оперативной обработки транзакций, и для поддержки принятия решений, на практике возникает ряд очень важных проблем. Рабочая нагрузка OLTP-систем состоит из множества коротких транзакций, выполняемых с большой частотой, причем важным фактором является малое время реакции системы. В противоположность этому запросы, связанные с принятием решений (например, "Какова средняя стоимость заказов для данного региона?" или "Насколько увеличился сбыт продукции по сравнению с аналогичным периодом прошлого года?"), могут требовать сканирования огромных таблиц и выполняться минутами или даже часами. Если экономист-аналитик попытается выполнить подобный запрос во время

пика деловых транзакций, это может вызвать серьезное снижение производительности всей системы, что для оперативной обработки транзакций просто недопустимо. Еще одну проблему составляет размещение данных, необходимых для получения ответов на вопросы из области делового анализа: они могут быть распределены между многими базами данных, причем, как правило, еще и поддерживаемыми разными СУБД и на разных компьютерных платформах.

Необходимость в полноценном использовании информационного потенциала предприятия, т.е. накопленных им данных, а также практические проблемы, связанные с производительностью OLTP-систем, породили новое технологическое решение, получившее название "хранилище данных" (data warehouses). Идею хранилища данных иллюстрирует схема на рис. 3.6. Бизнес-данные извлекаются из OLTP-систем предприятия, форматируются, проверяются и помещаются в отдельную базу данных, предназначенную исключительно для делового анализа ("хранилище"). Извлечение и форматирование данных можно выполнять на периодической основе в пакетном режиме во время наименьшей загрузки системы. В идеале из транзакционных баз данных извлекаются только новые и измененные данные, что позволяет сократить общее количество данных, обрабатываемых во время ежемесячной, еженедельной или ежедневной операции обновления хранилища. Таким образом, длительные запросы, связанные с деловым анализом, адресуются только к хранилищу данных и не требуют участия систем оперативной обработки транзакций.

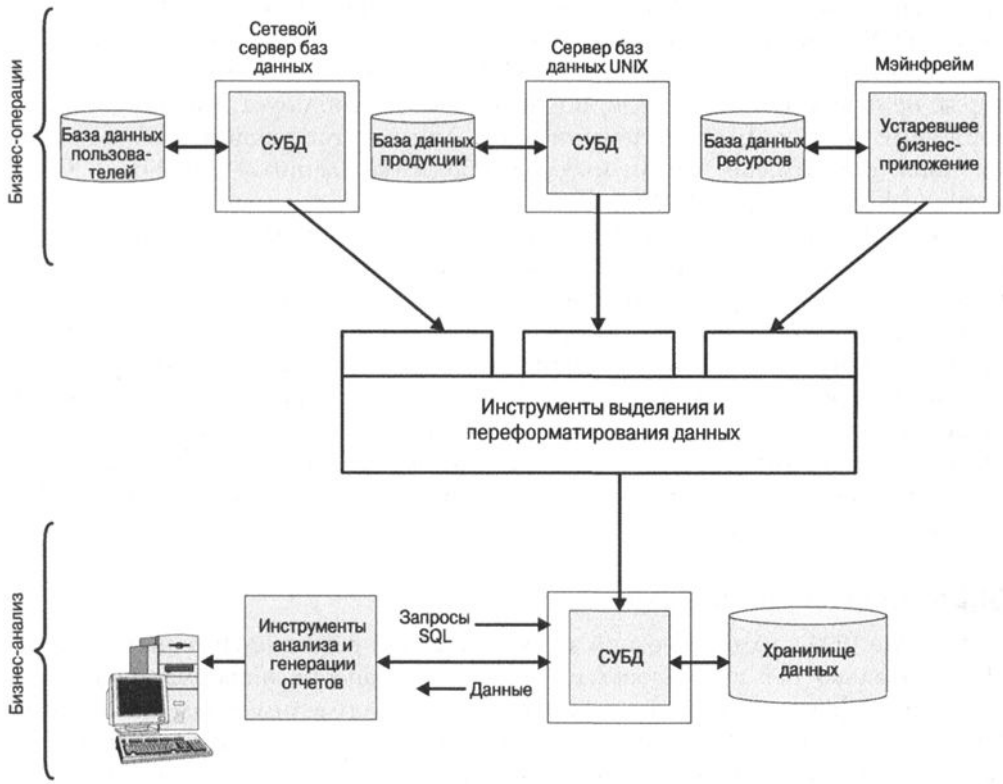


Рис. 3.6. Концепция хранилищ данных

Благодаря своей гибкости в обработке запросов, реляционные СУБД идеально подходили для организации хранилищ данных. Появились новые компании, взявшиеся за разработку программных средств для извлечения данных из OLTP-систем, их преобразования, загрузки в хранилище и последующего выполнения запросов к хранилищу. Не тратили времени зря и производители СУБД, сконцентрировавшие свое внимание на типичных запросах делового анализа. Эти запросы, как правило, бывают большими и сложными, как, например, анализ десятков или сотен миллионов отдельных операций оплаты товара для выявления наиболее распространенных схем оплаты, и часто связаны с обработкой хронологических данных — скажем, анализ ежемесячного изменения объема продаж или доли компании в своем сегменте рынка. Кроме того, в таких запросах чаще всего фигурируют статистические результаты — суммарный объем продаж, средняя стоимость заказов, процент роста и т.п., а не отдельные элементы данных.

Для удовлетворения специфических потребностей приложений, работающих с хранилищами данных и часто называемых приложениями для оперативной аналитической обработки данных (Online Analytical Processing, OLAP), стали появляться специализированные СУБД. Их производительность оптимизирована для выполнения сложных запросов, связанных только с чтением данных. Они поддерживают расширенный набор статистических функций и имеют встроенные средства для других распространенных типов анализа данных, например хронологического. За счет заблаговременного вычисления статистических данных эти СУБД могут существенно ускорять получение средних и суммарных величин. Среди специализированных СУБД есть некоторое количество таких, которые не используют язык SQL, но большинство использует именно его, благодаря чему получил распространение еще один сопутствующий термин — реляционная оперативная аналитическая обработка данных (Relational Online Analytical Processing, ROLAP).

Эволюция рынка хранилищ данных привела к тому, что новым важным сегментом рынка стали инструменты для работы с хранилищами, часто именуемые интеллектуальными ресурсами (*business intelligence*). Три ведущие компании в этой области достигли немалого успеха на рынке до того, как были куплены промышленными гигантами. Компания Business Objects была приобретена SAP, ведущим производителем приложений уровня предприятия. Компания Hyperion досталась Oracle, а Cognos — IBM. Как и во многих других сегментах рынка, достоинства SQL в качестве стандарта сделали его важным фактором развития и обеспечили закрепление на рынке SQL-хранилищ данных и аналитического инструментария.

SQL и интернет-приложения

В конце 1990-х годов основной движущей силой развития Интернета стал Веб. Первоначально Веб использовался для распространения информации в форме текста и графики и практически не применялся в области управления данными. Однако в средине 1990-х годов большая часть содержимого на корпоративных веб-сайтах была из корпоративных SQL-баз данных. Например, на коммерческом веб-сайте страницы содержат информацию о товарах, ценах, доступности товара на

складе, специальных предложениях и т.п., причем обычно такие страницы создаются по запросу, на основе данных, выбранных из SQL-базы данных. Подавляющее большинство страниц на сайтах аукционов или туристических агентств также основано на информации из SQL-баз данных, преобразованной в формат HTML-страниц. Осуществляется поток информации и в другом направлении, когда введенные пользователем данные в формы на веб-страницах вносятся в SQL-базы данных, являющиеся частью архитектуры веб-сайта.

В начале 2000-х годов технологии Интернета стали применяться для соединения компьютерных приложений друг с другом. Такие архитектуры распределенных приложений получили широкое распространение под общим названием *веб-служб* (web services). В соответствии с давними неписаными правилами компьютерной индустрии, появление новой технологии сопровождается появлением двух лагерей с различными стандартами и языками для их реализации. В данном случае это лагерь, возглавляемый Microsoft под флагом .NET Framework, и лагерь сторонников Java и J2EE-серверов приложений. Обе архитектуры признают ключевую роль XML, стандарта обмена структурированными данными наподобие тех, что находятся в SQL-базах данных.

В ответ на поворот к веб-службам масса продуктов анонсировала поддержку XML в SQL-базах данных. Разработчики баз данных объявили о выпуске новых продуктов с поддержкой XML, доказывая, что именно в их базе данных идеально реализована встроенная возможность обмена данными в XML-формате через Интернет. На это откликнулись известные производители реляционных баз данных, добавляя в свои продукты возможность ввода-вывода в формате XML и соответствующие типы данных. Тесная интеграция XML и SQL остается областью активных исследований всех основных производителей баз данных.

Подход Интернета к масштабируемости также оказывает большое влияние на программное обеспечение баз данных. Многие элементы программного обеспечения в Интернете используют горизонтальное масштабирование, когда рабочая нагрузка распределяется между десятками или сотнями недорогих серверов. Поисковый механизм Google может служить одним из наиболее выдающихся примеров такой архитектуры, когда даже один простой запрос может быть распределен между десятками серверов, а общий объем поиска — десятки тысяч серверов в Интернете. Имеется масса трудностей при применении такого подхода к управлению базами данных, и в настоящее время в этой области ведутся активные исследовательские работы.

Резюме

В этой главе описано развитие SQL и его роль в качестве стандартного языка управления реляционными базами данных.

- SQL был разработан научными сотрудниками компании IBM, и поддержка языка со стороны IBM является главной составляющей его успеха.

- Существуют официальные стандарты SQL, утвержденные ANSI/ISO, которые существенно выросли как по сложности, так и по количеству охватываемых тем со времени первой версии 1986 года.
- Вопреки наличию стандартов, имеется множество коммерческих диалектов SQL. Ни один из них не соответствует в точности другому.
- SQL стал стандартным языком управления базами данных в широком диапазоне компьютерных систем и приложений, включая мэйнфреймы, рабочие станции, персональные компьютеры, системы оперативной обработки транзакций, системы "клиент/сервер", хранилища данных и Интернет.

ГЛАВА 4

Реляционные базы данных

СУБД организует и структурирует данные таким образом, чтобы пользователи и прикладные программы могли их сохранять и выбирать из базы данных. Структуры данных и способы доступа к ним, обеспечиваемые конкретной СУБД, называются ее *моделью данных*. Модель данных определяет как "индивидуальность" СУБД, так и круг приложений, для которых она подходит наилучшим образом.

SQL представляет собой язык для работы с реляционными базами данных и основан на *реляционной модели данных*. Что это такое? В каком виде информация хранится в реляционной базе данных? Чем реляционные базы данных отличаются от более ранних баз данных, таких как иерархические и сетевые? Какими преимуществами и недостатками обладает реляционная модель? В данной главе описана реляционная модель данных, поддерживаемая языком SQL, и приведено ее сравнение с более ранними стратегиями организации баз данных.

Ранние модели данных

Когда в 1970-80-х годах стали популярны базы данных, появилось множество различных моделей данных. Каждая из них имела свои преимущества и недостатки, которые сыграли ключевую роль в развитии реляционной модели данных, появившейся во многом благодаря стремлению упростить и упорядочить ранние модели данных. Чтобы понять роль SQL и реляционных баз данных и оценить их вклад в развитие СУБД, следует кратко изучить ряд моделей данных, предшествовавших появлению SQL.

Системы управления файлами

До появления СУБД все данные, которые содержались в компьютерной системе постоянно, хранились в виде отдельных файлов. *Система управления файлами*, которая обычно являлась частью операционной системы, следила за именами фай-

лов и их размещением. Системы управления файлами широко используются и сегодня — вероятно, вы знакомы со структурой папок и файлов, предоставляемой файловыми системами операционных систем Microsoft Windows или Macintosh компании Apple. Аналогичные файловые системы используются и в UNIX-серверах и всех коммерческих вычислительных системах.

В системах управления файлами модели данных, как правило, отсутствуют; эти системы ничего не знают о внутреннем содержимом файлов. В лучшем случае файловая система поддерживает информацию о "типе файла" наряду с его именем, позволяя отличить документ текстового редактора от файла, содержащего данные о начисленной зарплате. Знание о содержимом файла — какие данные в нем хранятся и как они организованы — удел прикладных программ, использующих этот файл, как показано на рис. 4.1. В этом приложении начисления зарплаты каждая из программ на языке программирования COBOL, работающая с основным файлом с информацией о сотрудниках, содержит *описание файла* (file description, FD), в котором указана схема размещения данных в файле. Если структура данных изменяется — например, при решении хранить некоторую дополнительную информацию о каждом сотруднике — должны быть соответствующим образом модифицированы все программы, работающие с данным файлом. Это не слишком большая проблема в случае файла с документом текстового редактора или электронных таблиц, которые обычно обрабатываются одной программой. Но при корпоративной работе с данными файлы зачастую совместно используются десятками, а то и сотнями программ (см. рис. 4.1). При увеличении количества файлов и программ отделу обработки данных придется тратить больше усилий на поддержание работоспособности старых программ, чем на разработку новых.

Проблемы сопровождения больших систем, основанных на файлах, привели в конце 1960-х годов к появлению СУБД. В основе СУБД лежала простая идея: изъять из отдельных программ определение структуры содержимого файла и хранить это определение вместе с данными, в базе данных. Используя информацию, хранящуюся в базе данных, СУБД может играть существенно более активную роль как в управлении данными, так и в изменениях структуры данных. Кроме того, СУБД представляют собой расширения систем управления файлами, а не их замену. СУБД используют системы управления файлами (обычно входящими в состав операционных систем) для хранения структур баз данных. Затем пользователь базы данных обращается к СУБД, которая работает с деталями физического хранения информации. Это тот уровень абстракции, который обеспечивает физическую независимость данных.

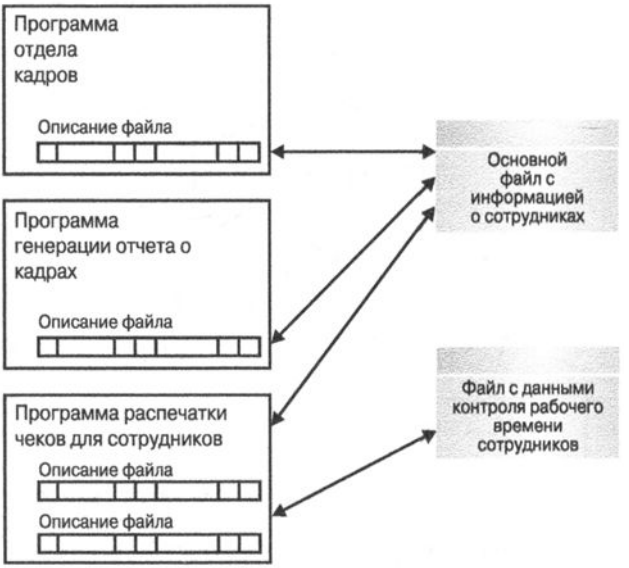


Рис. 4.1. Приложение для начисления зарплаты, использующее систему управления файлами

Иерархические базы данных

Одной из наиболее важных сфер применения первых СУБД было планирование производства для компаний, занимающихся выпуском продукции. Например, если автомобильная компания хотела выпустить 10000 машин одной модели и 5000 машин другой модели, ей необходимо было знать, сколько деталей следует заказать у своих поставщиков. Чтобы ответить на этот вопрос, необходимо выяснить, из каких частей состоит изделие, затем определить, из каких деталей состоят эти части, и т.д. Например, машина состоит из двигателя, корпуса и ходовой части; двигатель состоит из клапанов, цилиндров, свечей и т.д. Для обработки таких списков частей идеально подходят компьютеры.

Список составных частей изделия по своей природе является иерархической структурой. Для хранения данных, имеющих такую структуру, была разработана *иерархическая* модель данных, которую иллюстрирует рис. 4.2. В этой модели каждая *запись* базы данных представляла конкретную деталь. Между записями существовали отношения *предок-потомок*, связывающие каждую часть с деталями, входящими в нее.

При доступе к информации, содержащейся в базе данных, программа могла выполнить следующие задачи:

- найти конкретную деталь (такую, как левая дверь) по ее номеру;
- перейти "вниз" к первому потомку (ручка двери);
- перейти "вверх" к предку (кузов);
- перейти "в сторону" к другому потомку (правая дверь).

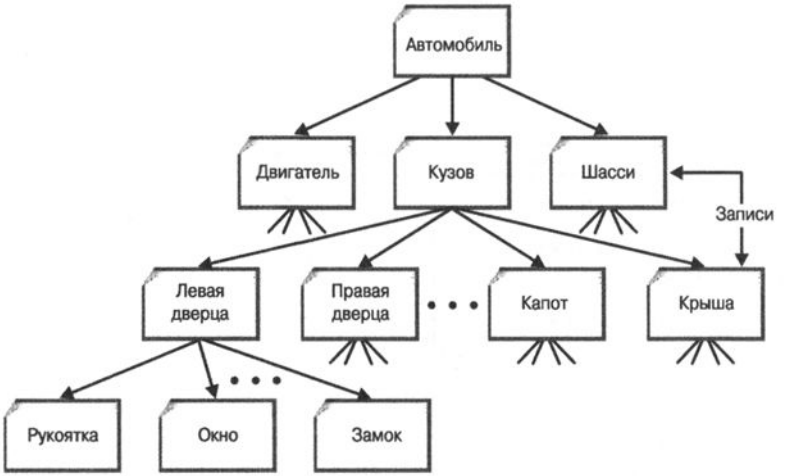


Рис. 4.2. Иерархическая база данных, содержащая информацию о составных частях

Таким образом, для чтения информации из иерархической базы данных требовалась возможность перемещения по записям, за один шаг переходя на одну запись вверх, вниз или в сторону.

Одной из наиболее популярных иерархических СУБД была Information Management System (IMS) компании IBM, появившаяся в 1968 году. Ниже перечислены преимущества IMS и реализованной в ней иерархической модели.

- **Простая структура**. Организацию базы данных IMS легко понять. Иерархия базы данных напоминает структуру компании или генеалогическое дерево.
- **Использование отношений "предок-потомок"**. СУБД IMS позволяла легко представлять отношения "предок-потомок", такие как "А является частью В" или "А принадлежит В".
- **Производительность**. В СУБД IMS отношения "предок-потомок" реализованы в виде физических указателей из одной записи на другую, вследствие чего перемещение по базе данных выполняется очень быстро.

СУБД IMS все еще остается одной из распространенных СУБД для мэйнфреймов компании IBM. Обладающая очень высокой производительностью, она идеально подходит для приложений, связанных с обработкой большого числа транзакций, таких как транзакции с кредитными карточками или резервирование авиабилетов. Хотя за последние пару десятилетий производительность реляционных баз данных возросла столь существенно, что описанное преимущество IMS стало не столь важным, большое количество корпоративных данных, хранящихся в базах данных IMS, и множество старых приложений, работающих с этими данными, гарантируют, что СУБД IMS будет использоваться еще много лет.

Сетевые базы данных

Если структура данных оказывалась сложнее, чем обычная иерархия, простота организации иерархической базы данных становилась ее недостатком. Например, в базе данных для хранения заказов один заказ может участвовать в трех *различных* отношениях "предок-потомок", связывающих заказ с покупателем, разместившим его, продавцом, принявшим его, и с заказанным товаром, что проиллюстрировано на рис. 4.3. Такие структуры данных не соответствовали строгой иерархии IMS.

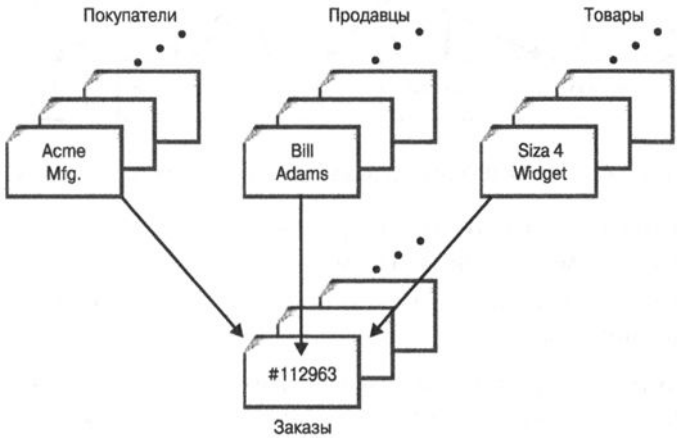


Рис. 4.3. Множественные отношения "предок-потомок"

В связи с этим для таких приложений, как обработка заказов, была разработана новая, *сетевая*, модель данных. Она расширила иерархическую модель, позволяя одной записи участвовать в нескольких отношениях "предок-потомок", именуемых *множествами* (set) (рис. 4.4). В 1971 году на конференции по языкам обработки данных (Conference on Data Systems Languages, CODASYL) был опубликован официальный стандарт сетевых баз данных, который известен как модель CODASYL. Компания IBM не стала разрабатывать собственную сетевую СУБД, но в 1970-х годах независимые производители программного обеспечения реализовали сетевую модель в таких продуктах, как IDMS компании Cullinet, Total компании Cincom и СУБД Adabas, которые приобрели большую популярность. Однако IBM усовершенствовала IMS, обеспечив путь обхода правила единственного предка в классических иерархических структурах, в котором дополнительные предки рассматриваются как логические. Эта модель данных, ставшая известной как расширенная иерархическая модель, сделала базу данных IMS конкурентом сетевых СУБД.

С точки зрения программиста, доступ к сетевой базе данных был очень похож на доступ к иерархической базе данных. Прикладная программа могла сделать следующее:

- найти определенную запись предка по ключу (такому, как номер клиента);
- перейти к первому потомку в определенном множестве (первый заказ, размещенный клиентом);

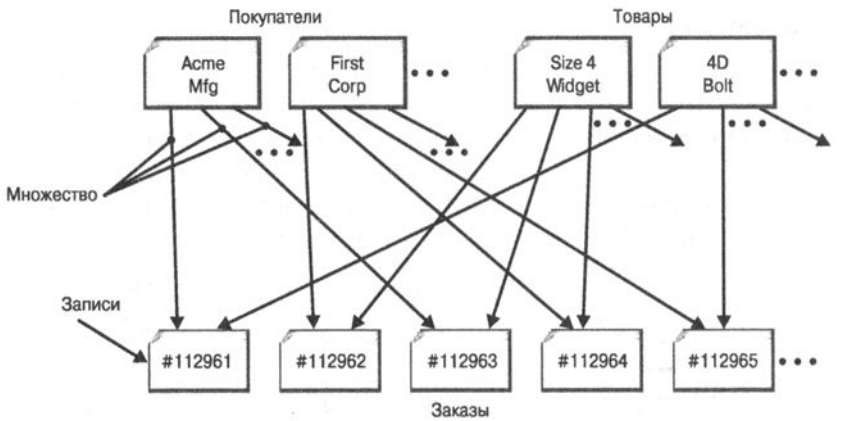


Рис. 4.4. Сетевая (CODASYL) база данных для работы с заказами

- перейти от одного потомка к другому в определенном множестве по горизонтали (следующий заказ, сделанный этим же клиентом);
- перейти вверх от потомка к его предку в другом множестве (служащий, принявший заказ).

И опять программисту приходилось искать информацию в базе данных, последовательно перебирая записи, но указывая при этом не только направление, но и требуемое отношение.

Сетевые базы данных обладали рядом преимуществ.

- **Гибкость**. Множественные отношения "предок-потомок" позволяли сетевой базе данных хранить информацию, структура которой не укладывалась в простую иерархию.
- **Стандартизация**. Появление стандарта CODASYL увеличило популярность сетевой модели, упрощая программистам переход от одной СУБД к другой.
- **Производительность**. Отношения "предок-потомок" были представлены указателями на физические записи, что обеспечивало высокую скорость перемещения по базе данных.

Конечно, у сетевых баз данных имелись недостатки. Подобно своим иерархическим предкам, сетевые базы данных были очень "жесткими". Наборы отношений и структура записей должны были быть заданы наперед. Изменение структуры базы данных обычно означало полную перестройку последней.

И иерархическая, и сетевая база данных были инструментами программистов. Чтобы получить ответ на вопрос *какой товар наиболее часто заказывает компания X?* или *сколько всего заказано единиц товара Y?*, программисту приходилось писать программу для навигации по базе данных, выборки нужных записей и подсчета результата. Реализация пользовательских запросов часто затягивалась на недели и месяцы, и к моменту появления программы возвращаемая ею информация часто оказывалась бесполезной.

Недостатки иерархической и сетевой моделей привели к повышенному интересу к новой *реляционной* модели данных, впервые описанной доктором Коддом в 1970 году. Поначалу она представляла лишь академический интерес. Сетевые базы данных продолжали оставаться важной технологией на протяжении 1970-х и в начале 1980-х годов, особенно в мини-компьютерных системах, переживавших пик популярности. Однако в середине 1980-х годов начался взлет реляционной модели. В начале 1990-х годов сетевые базы данных утратили популярность и сегодня не играют значительной роли на рынке баз данных.

Реляционная модель данных

Реляционная модель данных, предложенная Коддом, была попыткой упростить структуру базы данных. В ней отсутствовала явная структура "предок-потомок", а все данные были представлены в виде простых таблиц, разбитых на строки и столбцы. На рис. 4.5 показана реляционная версия рассмотренной выше сетевой базы данных, содержащей информацию о заказах (рис. 4.4).

Таблица **PRODUCTS**

DESCRIPTION	PRICE	QTY_ON_HAND
Size 3 Widget	$107.00	207
Size 4 Widget	$117.00	139
Hinge Pin	$350.00	14
.		
.		
.		

Таблица **ORDERS**

ORDER_NUM	COMPANY	PRODUCT	QTY
112963	Acme Mfg.	41004	28
112975	JCP Inc.	2A44G	6
112983	Acme Mfg.	41004	6
113012	JCP Inc.	41003	35
.			
.			
.			

Таблица **CUSTOMERS**

COMPANY	CUST_REP	CREDIT_LIMIT
Acme Mfg.	105	$50,000.00
JCP Inc.	103	$50,000.00
.		
.		
.		

Рис. 4.5. Реляционная база данных для работы с заказами

Работа Кодда дает точное, математическое определение реляционной базы данных, а также теоретический фундамент для операций, которые могут быть выполнены над ней. Однако более полезно следующее неформальное определение реляционной базы данных.

Реляционной называется база данных, в которой все данные, доступные пользователю, организованы в виде таблиц, а все операции базы данных выполняются над этими таблицами

Приведенное определение не оставляет места пользовательским структурам, таким как встроенные указатели иерархических и сетевых СУБД. Реляционная СУБД

способна реализовать отношения "предок-потомок", однако эти отношения представлены исключительно значениями, содержащимися в таблицах базы данных.

Учебная база данных

На рис. 4.6 показана маленькая реляционная база данных для приложения, выполняющего обработку заказов. Большинство примеров в данной книге построено на ее основе. Полное описание структуры и содержимого учебной базы данных, изображенной на рис. 4.6, приведено в приложении А, "Учебная база данных". Здесь представлено только по нескольку строк каждой таблицы.

Таблица SALESREPS

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	2006-02-12	104	$350,000.00	$367,911.00
109	Mary Jones	31	11	Sales Rep	2007-10-12	106	$300,000.00	$392.725.00
102	Sue Smith	48	21	Sales Rep	2004-12-10	108	$350,000.00	$474,050.00

Таблица PRODUCTS

MFR_ID	PRODUCT_IDD	ESCRIPTION	PRICE	QTY_ON_HAND
REI	2A45C	RatchetL ink	$79.00	210
ACI	4100Y	Widget Remover	$2,750.00	25
QSA	Xk47	Reducer	$355.00	38

Таблица ORDERS

ORDER_NUM	ORDER_DATE	CUST	REPM	FR	PRODUCT	QTY	AMOUNT
112961	2007-12-17	2117	106R	EI	2A44L	7	$31,500.00
113012	2008-01-11	2111	105A	CI	41003	35	$3,745.00
112989	2008-01-03	2101	106F	EA	114X	6	$1,458.00

Таблица OFFICES

OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300,000.00	$186,042.00
11	NewY ork	Eastern	106	$575,000.00	$692,637.00
12	Chicago	Eastern	104	$800,000.00	$735,042.00

Таблица CUSTOMERS

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2111	JCPI nc.	103	$50,000.00
2102	First Corp.	101	$65,000.00
2103	Acme Mfg.	105	$50,000.00

Рис. 4.6. Учебная база данных (представлена частично)

В учебной базе данных содержится пять таблиц. В каждой таблице хранится информация об одном конкретном *типе* данных:

- в таблице SALESREPS хранятся данные о каждом служащем, включая его идентификатор, имя, возраст, объем продаж с начала года и другую информацию;
- в таблице PRODUCTS хранятся данные о каждом товаре, предлагаемом на продажу, такие как его производитель, идентификатор, описание и цена;
- в таблице ORDERS хранятся все заказы клиентов с указанием идентификатора служащего, принявшего заказ, идентификатора заказанного то-

вара, его количества, стоимости заказа и т.д.; для простоты в одном заказе может упоминаться только один товар;

- в таблице OFFICES хранятся данные о каждом офисе, включая город, в котором расположен офис, область, к которой он принадлежит, и т.д.;
- в таблице CUSTOMERS хранятся данные о каждом клиенте, такие как название компании, лимит кредита и идентификатор служащего, работающего с этим клиентом.

Таблицы

В реляционной базе данных информация организована в виде прямоугольных *таблиц*, разделенных на строки и столбцы, на пересечении которых содержатся значения данных. Каждая таблица имеет уникальное *имя*, описывающее ее содержимое. (На практике, как показано в главе 5, "Основы SQL", каждый пользователь может присваивать собственным таблицам имена, не беспокоясь о том, какие имена выберут для своих таблиц другие пользователи.)

Более наглядно структуру таблицы иллюстрирует рис. 4.7, на котором изображена таблица OFFICES. Каждая горизонтальная *строка* этой таблицы представляет отдельную физическую сущность — один офис. Пять строк таблицы вместе представляют все пять офисов компании. Все данные, содержащиеся в конкретной строке таблицы, относятся к офису, который описывается этой строкой.

Таблица **OFFICES**

OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300,000.00	$186,042.00
11	New York	Eastern	106	$575,000.00	$692,637.00
12	Chicago	Eastern	104	$800,000.00	$735,042.00
13	Atlanta	Eastern	105	$350,000.00	$367,911.00
21	Los Angeles	Western	108	$725,000.00	$835,915.00

Данные в этой строке относятся к этому офису

Данные в этой строке относятся к этому офису

Город, в котором находится офис

Идентификатор управляющего офисом

Продажи офиса за текущий год

Рис. 4.7. Структура реляционной таблицы

Каждый вертикальный *столбец* таблицы OFFICES представляет один элемент данных для каждого из офисов. Например, в столбце CITY содержатся названия

городов, в которых расположены офисы. В столбце SALES содержатся объемы продаж, обеспечиваемые офисами. В столбце MGR содержатся идентификаторы управляющих офисами.

На пересечении строки и столбца таблицы содержится только одно значение данных. Например, в строке, представляющей нью-йоркский офис, в столбце CITY содержится значение "New York". В столбце SALES этой же строки находится значение $692637.00, которое является объемом продаж нью-йоркского офиса с начала года.

Все значения, содержащиеся в одном и том же столбце, являются данными одного типа. Например, в столбце CITY содержатся только слова, в столбце SALES — денежные суммы, а в столбце MGR — целые числа, представляющие идентификаторы служащих. Множество значений, которые могут содержаться в столбце, называется *доменом* этого столбца. Доменом столбца CITY является множество всех названий городов. Домен столбца SALES — это любая денежная сумма. Домен столбца REGION состоит всего из двух значений, "Eastern" и "Western", поскольку у компании всего два торговых региона.

У каждого столбца в таблице есть свое *имя*, которое обычно служит заголовком столбца. Все столбцы в одной таблице должны иметь уникальные имена, однако разрешается присваивать одинаковые имена столбцам, расположенным в различных таблицах. На практике такие имена столбцов, как NAME (имя), ADDRESS (адрес), PRICE (цена) и тому подобные, часто встречаются в различных таблицах одной базы данных.

Столбцы таблицы упорядочены слева направо, и их порядок определяется при создании таблицы. В любой таблице всегда есть как минимум один столбец. В стандарте ANSI/ISO максимально допустимое число столбцов в таблице не указывается; однако почти во всех коммерческих СУБД такой предел существует, но он редко бывает меньше 255 столбцов.

В отличие от столбцов, строки таблицы *не имеют* определенного порядка. Это значит, что если последовательно выполнить два одинаковых запроса для отображения содержимого таблицы, нет гарантии, что оба раза строки будут перечислены в одном и том же порядке. Конечно, можно попросить SQL-запрос отсортировать строки перед выводом, однако порядок сортировки не имеет совершенно ничего общего с фактическим расположением строк в таблице.

В таблице может содержаться любое количество строк. По очевидным причинам допускается существование таблицы с нулевым количеством строк. Такая таблица называется *пустой*. Пустая таблица сохраняет структуру, определенную ее столбцами, просто в ней не содержатся данные. Стандарт ANSI/ISO не накладывает ограничений на количество строк в таблице, и во многих СУБД размер таблиц ограничен лишь свободным дисковым пространством компьютера. В других СУБД имеется максимальный предел, однако обычно он весьма высок, — два миллиарда строк, а то и больше.

Первичные ключи

Поскольку строки в реляционной таблице не упорядочены, нельзя выбрать строку по ее номеру в таблице. В таблице нет "первой", "последней" или

"тринадцатой" строки. Тогда каким же образом можно выбрать в таблице конкретную строку, например строку для офиса, расположенного в Денвере?

В правильно построенной реляционной базе данных в каждой таблице есть столбец (или комбинация столбцов), для которого значения во всех строках различны. Этот столбец (столбцы) называется *первичным ключом* (primary key) таблицы. Давайте вновь посмотрим на базу данных, изображенную на рис. 4.7. На первый взгляд, первичным ключом таблицы OFFICES могут служить и столбец OFFICE, и столбец CITY. Однако в случае, если компания будет расширяться и откроет в каком-либо городе второй офис, столбец CITY больше не сможет исполнять роль первичного ключа. На практике в качестве первичных ключей таблиц обычно следует выбирать идентификаторы, такие как идентификатор офиса (OFFICE в таблице OFFICES), служащего (EMPL_NUM в таблице SALESREPS) и клиента (CUST_NUM в таблице CUSTOMERS). В случае с таблицей ORDERS у нас нет выбора — единственным столбцом, содержащим уникальные значения, является номер заказа (ORDER_NUM).

Таблица PRODUCTS, фрагмент которой показан на рис. 4.8, является примером таблицы, в которой первичный ключ представляет собой *комбинацию* столбцов. Такой первичный ключ называется *составным*. Столбец MFR_ID содержит идентификаторы производителей всех товаров, перечисленных в таблице, а столбец PRODUCT_ID содержит номера, присвоенные товарам производителями. Может показаться, что столбец PRODUCT_ID мог бы и один исполнять роль первичного ключа, однако ничто не мешает двум различным производителям присвоить своим изделиям одинаковые номера. Таким образом, в качестве первичного ключа таблицы PRODUCTS необходимо использовать комбинацию столбцов MFR_ID и PRODUCT_ID. Для каждого из товаров, содержащихся в таблице, комбинация значений в этих столбцах будет уникальной.

Таблица **PRODUCTS**

MFR_ID	PRODUCT_ID	DESCRIPTION	PRICE	QTY_ON_HAND
.				
.				
.				
ACI	41003	Size 3 Widget	$107.00	207
ACI	41004	Size 4 Widget	$117.00	139
BIC	41003	Handle	$652.00	3
.				
.				
.				

Первичный ключ

Рис. 4.8. Таблица с составным первичным ключом

Первичный ключ у каждой строки таблицы является уникальным в пределах таблицы, поэтому в таблице с первичным ключом нет двух совершенно одинаковых строк. Таблица, в которой все строки отличаются друг от друга, в математических терминах называется *отношением* (relation). Именно этому термину реляци-

онные базы данных и обязаны своим названием, поскольку в их основе лежат отношения, т.е. таблицы с отличающимися друг от друга строками.

Хотя первичные ключи являются важной частью реляционной модели данных, в первых реляционных СУБД (System/R, DB2, Oracle и других) явная их поддержка обеспечена не была. Как правило, проектировщики базы данных сами следили за тем, чтобы у всех таблиц были первичные ключи; в самих СУБД не было возможности задать для таблицы первичный ключ. И только СУБД DB2 Version 2, появившаяся в апреле 1988 года, была первым коммерческим SQL-продуктом с поддержкой первичных ключей. После этого подобная поддержка была добавлена в стандарт ANSI/ISO, и сегодня практически все СУРБД предоставляют такую возможность.

Взаимоотношения

Одним из отличий реляционной модели от ранних моделей представления данных было то, что в ней отсутствовали явные указатели, такие как использовавшиеся для реализации отношений "предок-потомок" в иерархической модели данных. Однако вполне очевидно, что такие отношения существуют и в реляционных базах данных. Например, в нашей учебной базе данных каждый из служащих закреплен за конкретным офисом, поэтому ясно, что между строками таблицы `OFFICES` и таблицы `SALESREPS` существует отношение. Не приводит ли отсутствие явных указателей в реляционной модели к потере информации?

Как следует из рис. 4.9, ответ на этот вопрос должен быть отрицательным. На рисунке изображено несколько строк из таблиц `OFFICES` и `SALESREPS`. Обратите внимание на то, что в столбце `REP_OFFICE` таблицы `SALESREPS` содержится идентификатор офиса, в котором работает служащий. Доменом этого столбца (множеством значений, которые могут в нем храниться) является множество идентификаторов офисов, содержащихся в столбце `OFFICE` таблицы `OFFICES`. Узнать, в каком офисе работает Мери Джонс, можно, определив значение столбца `REP_OFFICE` в строке таблицы `SALESREPS` для Мери Джонс (число 11), а затем отыскав в таблице `OFFICES` строку с таким же значением в столбце `OFFICE` (это строка для офиса в Нью-Йорке). Таким же образом, чтобы найти всех служащих нью-йоркского офиса, следует запомнить значение столбца `OFFICE` для Нью-Йорка (число 11), а потом просмотреть таблицу `SALESREPS` и найти все строки, в столбце `REP_OFFICE` которых содержится число 11 (это строки для Мери Джонс и Сэма Кларка).

Отношение "предок-потомок", существующее между офисами и работающими в них людьми, в реляционной модели не утеряно; просто оно реализовано в виде *одинаковых значений данных*, хранящихся в двух таблицах, а не в виде явного указателя. Таким способом реализуются все отношения, существующие между таблицами реляционной базы данных. Одним из главных преимуществ языка SQL является возможность извлекать связанные между собой данные, используя эти отношения.

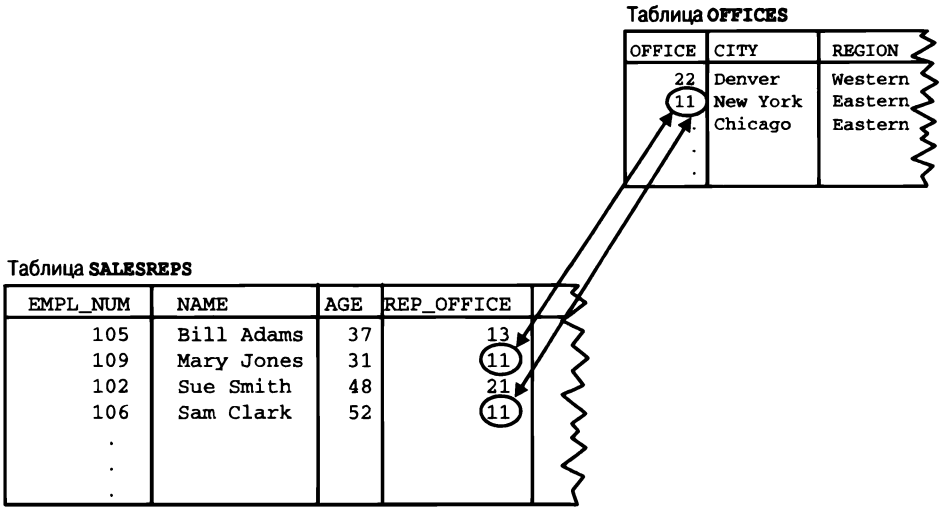


Рис. 4.9. Отношение "предок-потомок" в реляционной базе данных

Внешние ключи

Столбец одной таблицы, значения в котором совпадают со значениями столбца, являющегося первичным ключом другой таблицы, называется *внешним ключом* (foreign key). На рис. 4.9 столбец REP_OFFICE представляет собой внешний ключ для таблицы OFFICES. Хотя столбец REP_OFFICE и находится в таблице SALESREPS, значения, содержащиеся в нем, представляют собой идентификаторы офисов. Эти значения соответствуют значениям в столбце OFFICE, который является первичным ключом таблицы OFFICES. Первичный и внешний ключи создают между таблицами, в которых они содержатся, такое же отношение "предок-потомок", как и в иерархической базе данных.

Внешний ключ, как и первичный, тоже может представлять собой комбинацию столбцов. Фактически внешний ключ *всегда* будет составным (состоящим из нескольких столбцов), если он ссылается на составной первичный ключ в другой таблице. Очевидно, что количество столбцов и их типы данных в первичном и внешнем ключах совпадают.

Если таблица связана с несколькими другими таблицами, она может иметь несколько внешних ключей. На рис. 4.10 показаны три внешних ключа таблицы ORDERS из учебной базы данных:

- столбец CUST является внешним ключом для таблицы CUSTOMERS и связывает каждый заказ с клиентом, разместившим его;
- столбец REP является внешним ключом для таблицы SALESREPS и связывает каждый заказ со служащим, принявшим его;
- столбцы MFR и PRODUCT представляют собой составной внешний ключ для таблицы PRODUCTS, который связывает каждый заказ с заказанным товаром.

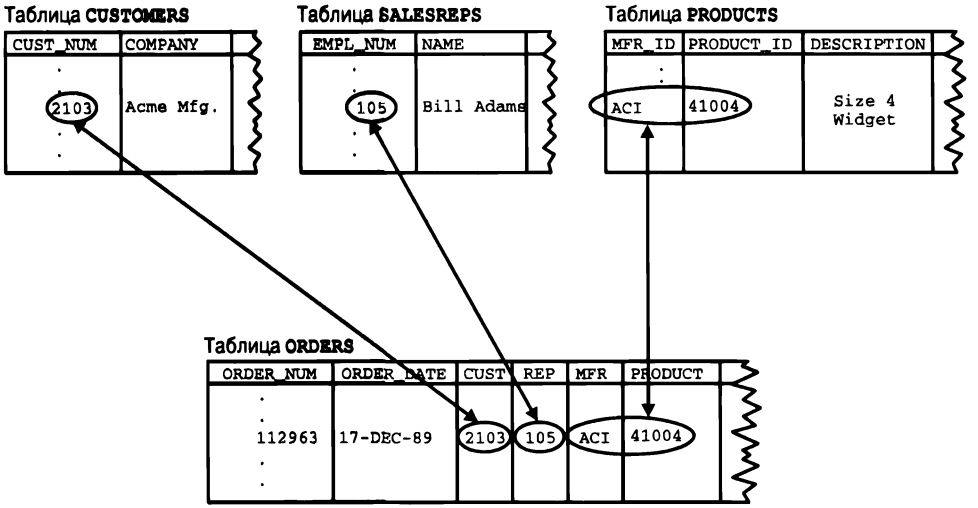


Рис. 4.10. Множественные отношения "предок-потомок" в реляционной базе данных

Отношения "предок-потомок", созданные с помощью трех внешних ключей в таблице ORDERS, могут показаться вам знакомыми. И действительно, это те же самые отношения, что и в сетевой базе данных, представленной на рис. 4.4. Как показывает данный пример, реляционная модель данных обладает всеми возможностями сетевой модели по части выражения сложных отношений.

Внешние ключи являются фундаментальной частью реляционной модели, поскольку реализуют отношения между таблицами базы данных. К сожалению, как и в случае с первичными ключами, поддержка внешних ключей отсутствовала в первых реляционных СУБД. Она была реализована в DB2 Version 2, а затем добавлена в стандарт ANSI/ISO и теперь имеется во всех основных коммерческих СУБД.

Двенадцать правил Кодда для реляционных баз данных*

Когда в средине 1980-х годов реляционная модель стала очень популярной, почти все производители СУБД стали добавлять слово "реляционный" в описание своих продуктов. Но ряд из них был не более чем тонким слоем SQL-подобного языка на поверхности сетевой или иерархической базы данных. Некоторые реализовывали только рудиментарную табличную структуру, даже не пытаясь реализовать язык запросов. Вскоре вопрос *так что же такое* ***настоящая*** *реляционная база данных?* стал подниматься все чаще и чаще, а производители СУБД стали утверждать, что их продукты "реляционнее", чем продукты их конкурентов.

В 1985 году Тед Кодд (чья статья 15-летней давности определила реляционную модель данных) задался этим вопросом и ответил на него в журнале Computerworld (*Is Your DBMS Really Relational?* (Действительно ли ваша СУБД реляционная?, 14.10.1985) и *Does Your DBMS Run By the Rules?* (Работает ли ваша СУБД по правилам?, 21.10.1985)). Здесь он изложил двенадцать правил, которым должна соответствовать настоящая реляционная база данных.

1. *Правило представления информации.* Вся информация в реляционной базе данных должна быть представлена исключительно на логическом уровне и только одним способом — в виде значений, содержащихся в таблицах.
2. *Правило гарантированного доступа.* Логический доступ ко всем и каждому элементу данных (атомарному значению) в реляционной базе данных должен обеспечиваться путем использования комбинации имени таблицы, значения первичного ключа и имени столбца.
3. *Систематическая трактовка значения NULL.* В настоящей реляционной базе данных должна быть реализована полная поддержка значений NULL (которые отличаются от строки символов нулевой длины, строки пробельных символов, а также от нуля или любого другого числа), которые используются для представления отсутствующей и неприменимой информации систематическим образом независимо от типа этих данных.
4. *Правило динамического каталога, основанного на реляционной модели.* Описание базы данных на логическом уровне должно быть представлено в том же виде, что и обычные данные, чтобы пользователи, обладающие соответствующими правами, могли работать с ним с помощью того же реляционного языка, который они применяют для работы с основными данными.
5. *Правило исчерпывающего подъязыка данных.* Реляционная система может поддерживать несколько языков и режимов взаимодействия с пользователем. Однако должен существовать по крайней мере один язык, инструкции которого можно представить в виде строк символов в соответствии с некоторым точно определенным синтаксисом и который в полной мере поддерживает все следующие элементы:
   - определение данных;
   - определение представлений;
   - обработку данных (интерактивную и программную);
   - ограничения целостности данных;
   - авторизацию;
   - границы транзакций (начало, фиксацию и откат).
6. *Правило обновления представлений.* Все представления, которые теоретически можно обновить, должны быть доступны для обновления системой.
7. *Правило высокоуровневого добавления, обновления и удаления.* Операции вставки, обновления и удаления должны применяться к отношению в целом.
8. *Правило физической независимости данных.* Прикладные программы и утилиты для работы с данными на логическом уровне должны оставаться неизменными при любых изменениях способов хранения данных или методов доступа к ним.
9. *Правило логической независимости данных.* Прикладные программы и утилиты для работы с данными должны на логическом уровне оставаться нетронутыми при внесении в базовые таблицы любых изменений, которые

теоретически позволяют сохранить нетронутыми содержащиеся в этих таблицах данные.

10. *Правило независимости контроля целостности.* Должна существовать возможность определять условия целостности, специфичные для конкретной реляционной базы данных, на подъязыке этой базы данных и хранить их в каталоге, а не в прикладной программе.
11. *Правило независимости распространения.* Реляционная база данных должна быть переносима не только в пределах системы, но и по сети.
12. *Правило согласования языковых уровней.* Если в реляционной системе есть низкоуровневый язык (обрабатывающий одну запись за один раз), то он не должен иметь возможность обходить правила и условия целостности данных, выраженные на реляционном языке высокого уровня (обрабатывающем несколько записей за один раз).

Хотя дискуссии по этому вопросу давно завершились, эти 12 правил интересны, как минимум, с исторической точки зрения, поскольку они раз и навсегда разрешили все вопросы и представляют собой хорошее неформальное определение реляционной базы данных. Правило 1 напоминает неформальное определение реляционной базы данных, приведенное ранее; остальные правила уточняют и дополняют его.

Правило 2 указывает на роль первичных ключей при поиске информации в базе данных. Имя таблицы позволяет найти требуемую таблицу, имя столбца — требуемый столбец, а первичный ключ — строку, содержащую искомый элемент данных. Правило 3 требует, чтобы отсутствующие данные можно было представить с помощью значения NULL, описываемого в главе 5, "Основы SQL".

Правило 4 гласит, что реляционная база данных должна описывать сама себя. Другими словами, база данных должна содержать набор *системных таблиц*, описывающих структуру самой базы данных. Эти таблицы рассматриваются в главе 16, "Системный каталог".

Правило 5 требует, чтобы СУБД использовала язык реляционной базы данных, например SQL, хотя явно SQL в правиле не упомянут. Такой язык должен поддерживать все основные функции СУБД, а не только выборку данных.

Правило 6 касается представлений, которые являются *виртуальными таблицами*, позволяющими показывать различным пользователям различные фрагменты структуры базы данных. Представления рассматриваются в главе 14, "Представления".

Правило 7 акцентирует внимание на том, что реляционные базы данных по своей природе ориентированы на работу с множествами. Оно требует, чтобы операции добавления, удаления и обновления можно было выполнять над множествами строк. Это правило предназначено для того, чтобы запретить реализации таких СУБД, в которых поддерживаются только операции над одной строкой.

Правила 8 и 9 изолируют пользователей и прикладные программы от низкоуровневой реализации базы данных и даже от изменений в структуре таблиц.

Правило 10 гласит, что язык базы данных должен поддерживать возможность определения ограничений на вводимые данные и изменения базы данных, которые могут быть выполнены.

Правило 11 говорит о том, что язык базы данных должен обеспечивать возможность работы с распределенными данными, расположенными в различных компьютерных системах.

И наконец, правило 12 предотвращает использование других средств работы с базой данных, помимо ее подъязыка, поскольку это может нарушить ее целостность.

Резюме

SQL основан на реляционной модели данных, в которой данные организованы в виде коллекции таблиц.

- Каждая таблица имеет уникальное имя.
- В каждой таблице есть один или несколько именованных столбцов, расположенных в определенном порядке слева направо.
- В каждой таблице есть нуль или более строк, каждая из которых содержит одно значение данных в каждом столбце; строки в таблице не упорядочены.
- Все значения данных в одном столбце имеют одинаковый тип данных и входят в набор допустимых значений, который называется доменом столбца.

Отношения между таблицами реализуются с помощью содержащихся в них данных. В реляционной модели данных для представления этих отношений используются первичные и внешние ключи.

- Первичным ключом может быть столбец или комбинация столбцов таблицы, значения которых уникальным образом идентифицируют каждую строку таблицы. У таблицы есть только один первичный ключ.
- Внешним ключом является столбец или группа столбцов таблицы, значения которых совпадают со значениями первичного ключа другой таблицы. Таблица может содержать несколько внешних ключей, связывающих ее с одной или несколькими другими таблицами.
- Пара "первичный ключ–внешний ключ" создает отношение "предок-потомок" между таблицами, содержащими их.

II

ЧАСТЬ

Выборка данных

Глава 5
Основы SQL

Глава 6
Простые запросы

Глава 7
Многотабличные запросы (соединения)

Глава 8
Итоговые запросы

Глава 9
Подзапросы и выражения с запросами

Запросы — сердце SQL, и многие используют SQL просто как инструмент для запросов к базам данных. В следующих пяти главах будут всесторонне и глубоко рассмотрены запросы SQL. В главе 5, "Основы SQL", описаны основные структуры SQL, используемые для формирования инструкций SQL. В главе 6, "Простые запросы", рассказывается о простых запросах, позволяющих извлекать данные из одной таблицы. Глава 7, "Многотабличные запросы (соединения)", посвящена многотабличным запросам. В главе 8, "Итоговые запросы", описаны запросы, позволяющие получить итоговые данные. И, наконец, в главе 9, "Подзапросы и выражения с запросами", рассказывается о подзапросах, которые используются для создания сложных запросов.

ГЛАВА 5

Основы SQL

В этой главе дается подробное описание SQL. Здесь рассматриваются структура инструкций языка, а также его базовые элементы, такие как ключевые слова, типы данных и выражения. Кроме того, в главе рассказывается, как SQL обрабатывает отсутствующие данные с помощью значений NULL. Несмотря на фундаментальность этих элементов языка, их реализации в различных популярных SQL-продуктах немного отличаются, и во многих случаях в язык вносятся существенные расширения по сравнению со стандартом ANSI/ISO. Эти отличия также рассматриваются в настоящей главе.

Инструкции

В SQL имеется около сорока инструкций (наиболее важные и часто используемые из них представлены в табл. 5.1). Каждая из них "просит" СУБД выполнить определенное действие, например извлечь данные, создать таблицу или добавить в таблицу новые данные. Все инструкции SQL имеют подобную структуру, которая изображена на рис. 5.1.

Таблица 5.1. Основные инструкции SQL

Инструкция	Описание
Обработка данных	
SELECT	Извлекает данные из базы данных
INSERT	Добавляет новые строки в базу данных
UPDATE	Обновляет данные, имеющиеся в базе данных
MERGE	Добавляет/обновляет/удаляет новые и старые строки на основе условий
DELETE	Удаляет строки из базы данных

Окончание табл. 5.1

Инструкция	Описание
Определение данных	
CREATE TABLE	Добавляет новую таблицу в базу данных
DROP TABLE	Удаляет таблицу из базы данных
ALTER TABLE	Изменяет структуру существующей таблицы
CREATE VIEW	Добавляет новое представление в базу данных
DROP VIEW	Удаляет представление из базы данных
CREATE INDEX	Создает индекс для столбца
DROP INDEX	Удаляет индекс столбца
CREATE SCHEMA	Добавляет новую схему в базу данных
DROP SCHEMA	Удаляет схему из базы данных
CREATE DOMAIN	Добавляет новый домен значений данных
ALTER DOMAIN	Изменяет определение домена
DROP DOMAIN	Удаляет домен из базы данных
Управление доступом	
GRANT	Предоставляет пользователю привилегии доступа
REVOKE	Отменяет пользовательские привилегии доступа
CREATE ROLE	Добавляет в базу данных новую роль
GRANT ROLE	Предоставляет роль, содержащую привилегии доступа
DROP ROLE	Удаляет роль из базы данных
Управление транзакциями	
COMMIT	Завершает текущую транзакцию
ROLLBACK	Отменяет текущую транзакцию
SET TRANSACTION	Определяет характеристики доступа к данным в текущей транзакции
START TRANSACTION	Явно начинает новую транзакцию
SAVEPOINT	Устанавливает точку восстановления транзакции
Программный SQL	
DECLARE	Определяет курсор запроса
EXPLAIN	Возвращает описание плана доступа к данным в запросе
OPEN	Открывает курсор для получения результатов запроса
FETCH	Извлекает строку из результатов запроса
CLOSE	Закрывает курсор
PREPARE	Подготавливает инструкцию SQL к динамическому выполнению
EXECUTE	Динамически выполняет инструкцию SQL
DESCRIBE	Описывает подготовленный запрос

Рис. 5.1. Структура инструкции SQL

Каждая инструкция SQL начинается с *команды*, т.е. ключевого слова, описывающего действие, выполняемое инструкцией. Типичными командами являются CREATE (создать), INSERT (добавить), DELETE (удалить) и COMMIT (зафиксировать). После команды идет одно или несколько *предложений*. Предложение может описывать данные, с которыми работает инструкция, или содержать уточняющую информацию о действии, выполняемом инструкцией. Каждое предложение также начинается с ключевого слова, такого как WHERE (где), FROM (откуда), INTO (куда) или HAVING (имеющий). Одни предложения в инструкции являются обязательными, а другие — нет. Конкретная структура и содержимое предложения могут изменяться. Многие предложения содержат имена таблиц или столбцов; некоторые из них могут содержать дополнительные ключевые слова, константы и выражения.

В стандарте ANSI/ISO определен набор зарезервированных (а также незарезервированных) ключевых слов, которые используются в инструкциях SQL. В соответствии со стандартом, зарезервированные ключевые слова нельзя использовать для именования объектов базы данных, таких как таблицы, столбцы и пользователи. Во многих реализациях СУБД этот запрет ослаблен, но тем не менее следует избегать использования ключевых слов в качестве имен таблиц и столбцов. В табл. 5.2 перечислены ключевые слова, включенные в стандарт ANSI/ISO SQL:2006.

Таблица 5.2. Зарезервированные ключевые слова SQL:2006

ABS	ALL	ALLOCATE	ALTER
AND	ANY	ARE	ARRAY
AS	ASENSITIVE	ASYMMETRIC	AT
ATOMIC	AUTHORIZATION	AVG	BEGIN
BETWEEN	BIGINT	BINARY	BLOB
BOOLEAN	BOTH	BY	CALL
CALLED	CARDINALITY	CASCADED	CASE
CAST	CEIL	CEILING	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CLOB	CLOSE	COALESCE	COLLATE
COLLECT	COLUMN	COMMIT	CONDITION
CONNECT	CONSTRAINT	CONVERT	CORR
CORRESPONDING	COUNT	COVAR_POP	COVAR_SAMP

Продолжение табл. 5.2

CREATE	CROSS	CUBE	CUME_DIST
CURRENT	CURRENT_DATE	CURRENT_DEFAULT_TRANSFORM_GROUP	CURRENT_PATH
CURRENT_ROLE	CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_TRANSFORM_GROUP_FOR_TYPE
CURRENT_USER	CURSOR	CYCLE	DATE
DAY	DEALLOCATE	DEC	DECIMAL
DECLARE	DEFAULT	DELETE	DENSE_RANK
DEREF	DESCRIBE	DETERMINISTIC	DISCONNECT
DISTINCT	DOUBLE	DROP	DYNAMIC
EACH	ELEMENT	ELSE	END
END-EXEC	ESCAPE	EVERY	EXCEPT
EXEC	EXECUTE	EXISTS	EXP
EXTERNAL	EXTRACT	FALSE	FETCH
FILTER	FLOAT	FLOOR	FOR
FOREIGN	FREE	FROM	FULL
FUNCTION	FUSION	GET	GLOBAL
GRANT	GROUP	GROUPING	HAVING
HOLD	HOUR	IDENTITY	IN
INDICATOR	INNER	INOUT	INSENSITIVE
INSERT	INT	INTEGER	INTERSECT
INTERSECTION	INTERVAL	INTO	IS
JOIN	LANGUAGE	LARGE	LATERAL
LEADING	LEFT	LIKE	LN
LOCAL	LOCALTIME	LOCALTIMESTAMP	LOWER
MATCH	MAX	MEMBER	MERGE
METHOD	MIN	MINUTE	MOD
MODIFIES	MODULE	MONTH	MULTISET
NATIONAL	NATURAL	NCHAR	NCLOB
NEW	NO	NONE	NORMALIZE
NOT	NULL	NULLIF	NUMERIC
OCTET_LENGTH	OF	OLD	ON
ONLY	OPEN	OR	ORDER
OUT	OUTER	OVER	OVERLAPS
OVERLAY	PARAMETER	PARTITION	PERCENT_RANK
PERCENTILE_CONT	PERCENTILE_DISC	POSITION	POWER
PRECISION	PREPARE	PRIMARY	PROCEDURE
RANGE	RANK	READS	REAL
RECURSIVE	REF	REFERENCES	REFERENCING
REGR_AVGX	REGR_AVGY	REGR_COUNT	REGR_INTERCEPT
REGR_R2	REGR_SLOPE	REGR_SXX	REGR_SXY
REGR_SYY	RELEASE	RESULT	RETURN

Окончание табл. 5.2

RETURNS	REVOKE	RIGHT	ROLLBACK
ROLLUP	ROW	ROW_NUMBER	ROWS
SAVEPOINT	SCOPE	SCROLL	SEARCH
SECOND	SELECT	SENSITIVE	SESSION_USER
SET	SIMILAR	SMALLINT	SOME
SPECIFIC	SPECIFICTYPE	SQL	SQLEXCEPTION
SQLSTATE	SQLWARNING	SQRT	START
STATIC	STDDEV_POP	STDDEV_SAMP	SUBMULTISET
SUBSTRING	SUM	SYMMETRIC	SYSTEM
SYSTEM_USER	TABLE	TABLESAMPLE	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRANSLATE	TRANSLATION
TREAT	TRIGGER	TRIM	TRUE
UESCAPE	UNION	UNIQUE	UNKNOWN
UNNEST	UPDATE	UPPER	USER
USING	VALUE	VALUES	VAR_POP
VAR_SAMP	VARCHAR	VARYING	WHEN
WHENEVER	WHERE	WIDTH_BUCKET	WINDOW
WITH	WITHIN	WITHOUT	YEAR

Следует также избегать при именовании объектов баз данных незарезервированных ключевых слов, поскольку они являются кандидатами в зарезервированные в будущих версиях стандарта. В табл. 5.3 перечислены ключевые слова, включенные в стандарт ANSI/ISO SQL:2006.

Таблица 5.3. Незарезервированные ключевые слова SQL:2006

ABSOLUTE	ACTION	ADA	ADD
ADMIN	AFTER	ALWAYS	ASC
ASSERTION	ASSIGNMENT	ATTRIBUTE	ATTRIBUTES
BEFORE	BERNOULLI	BREADTH	CASCADE
CATALOG	CATALOG_NAME	CHAIN	CHARACTER_SET_ CATALOG
CHARACTER_SET_NAME	CHARACTER_SET_ SCHEMA	CHARACTERISTICS	CHARACTERS
CLASS_ORIGIN	COBOL	COLLATION	COLLATION_CATALOG
COLLATION_NAME	COLLATION_SCHEMA	COLUMN_NAME	COMMAND_FUNCTION
COMMAND_FUNCTION_ CODE	COMMITTED	CONDITION_NUMBER	CONNECTION
CONNECTION_NAME	CONSTRAINT_CATALOG	CONSTRAINT_NAME	CONSTRAINT_SCHEMA
CONSTRAINTS	CONSTRUCTOR	CONTAINS	CONTINUE
CURSOR_NAME	DATA	DATETIME_INTERVAL_ CODE	DATETIME_INTERVAL_ PRECISION
DEFAULTS	DEFERRABLE	DEFERRED	DEFINED

Продолжение табл. 5.3

DEFINER	DEGREE	DEPTH	DERIVED
DESC	DESCRIPTOR	DIAGNOSTICS	DISPATCH
DOMAIN	DYNAMIC_FUNCTION	DYNAMIC_FUNCTION_CO DE	EQUALS
EXCEPTION	EXCLUDE	EXCLUDING	FINAL
FIRST	FOLLOWING	FORTRAN	FOUND
GENERAL	GENERATED	GO	GOTO
GRANTED	IMMEDIATE	IMPLEMENTATION	INCLUDING
INCREMENT	INITIALLY	INPUT	INSTANCE
INSTANTIABLE	INVOKER	ISOLATION	KEY
KEY_MEMBER	KEY_TYPE	LAST	LENGTH
LEVEL	LOCATOR	MAP	MATCHED
MAXVALUE	MESSAGE_LENGTH	MESSAGE_OCTET_LENGT H	MESSAGE_TEXT
MINVALUE	MORE	MUMPS	NAME
NAMES	NESTING	NEXT	NORMALIZED
NULLABLE	NULLS	NUMBER	OBJECT
OCTETS	OPTION	OPTIONS	ORDERING
ORDINALITY	OTHERS	OUTPUT	OVERRIDING
PAD	PARAMETER_MODE	PARAMETER_NAME	PARAMETER_ORDINAL_ POSITION
PARAMETER_SPECIFIC_ CATALOG	PARAMETER_ SPECIFIC_NAME	PARAMETER_SPECIFIC_ SCHEMA	PARTIAL
PASCAL	PATH	PLACING	PLI
PRECEDING	PRESERVE	PRIOR	PRIVILEGES
PUBLIC	READ	RELATIVE	REPEATABLE
RESTART	RESTRICT	RETURNED_CARDINALIT Y	RETURNED_LENGTH
RETURNED_OCTET_ LENGTH	RETURNED_SQLSTATE	ROLE	ROUTINE
ROUTINE_CATALOG	ROUTINE_NAME	ROUTINE_SCHEMA	ROW_COUNT
SCALE	SCHEMA	SCHEMA_NAME	SCOPE_CATALOG
SCOPE_NAME	SCOPE_SCHEMA	SECTION	SECURITY
SELF	SEQUENCE	SERIALIZABLE	SERVER_NAME
SESSION	SETS	SIMPLE	SIZE
SOURCE	SPACE	SPECIFIC_NAME	STATE
STATEMENT	STRUCTURE	STYLE	SUBCLASS_ORIGIN
TABLE_NAME	TEMPORARY	TIES	TOP_LEVEL_COUNT
TRANSACTION	TRANSACTION_ACTIVE	TRANSACTIONS_ COMMITTED	TRANSACTIONS_ ROLLED_BACK
TRANSFORM	TRANSFORMS	TRIGGER_CATALOG	TRIGGER_NAME
TRIGGER_SCHEMA	TYPE	UNBOUNDED	UNCOMMITTED
UNDER	UNNAMED	USAGE	USER_DEFINED_TYPE_ CATALOG

Окончание табл. 5.3

USER_DEFINED_TYPE_ CODE	USER_DEFINED_TYPE_ NAME	USER_DEFINED_TYPE_ SCHEMA	VIEW
WORK	WRITE	ZONE	

В данной книге допустимые формы инструкций SQL иллюстрируются с помощью синтаксических диаграмм, таких как показанная на рис. 5.2. Чтобы создать правильную инструкцию или предложение, необходимо пройти по синтаксической диаграмме вдоль линий до точки, которая служит концом диаграммы. На синтаксической диаграмме и в примерах ключевые слова всегда напечатаны прописными буквами (как слова DELETE и FROM на рис. 5.2), но почти во всех СУБД ключевые слова можно набирать как прописными, так и строчными буквами, и часто последнее оказывается более удобным.

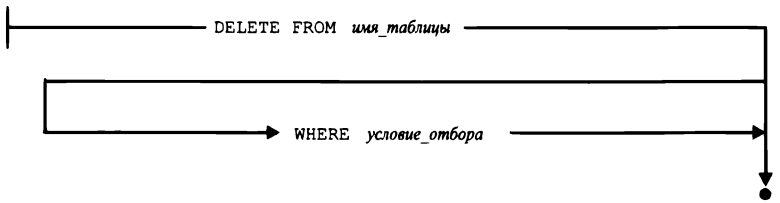


Рис. 5.2. Пример синтаксической диаграммы

Изменяемые элементы инструкции SQL в синтаксической диаграмме указаны строчными буквами и выделены курсивом (имя таблицы и условие отбора на рис. 5.2). При создании инструкции программист должен самостоятельно определить каждый из таких элементов. Необязательные предложения и ключевые слова (например, предложение WHERE на рис. 5.2) на синтаксической диаграмме показаны с помощью дополнительных линий. Если имеется возможность необязательного выбора из нескольких ключевых слов, тогда то из них, которое принято по умолчанию (т.е. действие, выполняемое инструкцией, если не указано ни одно из ключевых слов, выполняется, как если бы было указано слово по умолчанию), подчеркивается.

Имена

У каждого объекта в базе данных есть уникальное имя. Имена используются в инструкциях SQL и указывают, над каким объектом базы данных должно быть выполнено действие. Фундаментальными именованными объектами в реляционной базе данных являются имена таблиц, столбцов и пользователей; правила их именования были определены еще в стандарте SQL1. В последующих версиях стандарта этот список был значительно расширен и теперь включает схемы (коллекции таблиц), ограничения (ограничительные условия, накладываемые на содержимое таблиц и их отношения), домены (допустимые наборы значений, которые могут быть занесены в столбец) и ряд других типов объектов. Во многих СУБД существуют дополнительные виды именованных объектов, например хра-

нимые процедуры, отношения "первичный ключ–внешний ключ", формы для ввода данных и схемы репликации данных.

В соответствии с первоначальным стандартом ANSI/ISO, имена в SQL должны содержать от 1 до 18 символов, начинаться с буквы и не могут содержать пробельные символы или специальные символы пунктуации. В стандарте SQL2 максимальное число символов в имени увеличено до 127 (дословно в стандарте сказано "менее 128"), и это количество остается неизменным до последнего стандарта SQL:2006. На практике поддержка имен в различных СУБД реализована по-разному. Чаще всего приходится сталкиваться с ограничениями на имена, связанные с другим программным обеспечением вне базы данных (например, имена пользователей, которые могут совпадать с именами, используемыми операционной системой). Различные продукты отличаются и по отношению к применению в именах специальных символов. С точки зрения переносимости, лучше воздержаться от длинных имен и не применять в них никаких специальных символов, за исключением символа подчеркивания, который используется для разделения слов в именах SQL.

Имена таблиц

Если в инструкции указано имя таблицы, SQL предполагает, что происходит обращение к одной из ваших собственных таблиц (т.е. таблиц, которые создали вы). Обычно таблицам присваиваются короткие, но описательные имена.

Имена таблиц в учебной базе данных (`ORDERS`, `CUSTOMERS`, `OFFICES`, `SALESREPS`) могут служить хорошими примерами. В персональных базах данных или базах данных небольших отделов выбором имен для таблиц обычно занимается разработчик или проектировщик базы данных.

В более крупных корпоративных базах данных могут существовать определенные корпоративные стандарты именования таблиц, позволяющие избежать конфликтов имен. Кроме того, большинство СУБД позволяет различным пользователям создавать таблицы с одинаковыми именами (например, и пользователь Joe, и пользователь Sam могут создать таблицу `BIRTHDAYS`). СУБД обращается к необходимой таблице в зависимости от того, кто из пользователей запрашивает данные. При наличии соответствующих прав можно обращаться к таблицам, владельцами которых являются другие пользователи, с помощью *полного*, или *квалифицированного, имени таблицы*. Оно состоит из имен владельца таблицы и собственно таблицы, разделенных точкой. Например, квалифицированное имя таблицы `BIRTHDAYS`, владельцем которой является пользователь `SAM`, имеет такой вид.

```
SAM.BIRTHDAYS
```

В общем случае в инструкциях SQL везде, где должно использоваться имя таблицы, можно использовать ее квалифицированное имя.

Стандарт ANSI/ISO SQL еще больше обобщает понятие квалифицированного имени таблицы. Он разрешает создавать именованное множество таблиц, называемое *схемой*. Вы можете обращаться к таблице определенной схемы с использованием квалифицированного имени. Например, обращение к таблице `BIRTHDAYS` в схеме `EMPLOYEE_INFO` имеет следующий вид.

```
EMPLOYEE_INFO.BIRTHDAYS
```

В главе 13, "Создание базы данных", имеется больше информации о схемах, пользователях и иных аспектах структуры SQL-базы данных. Пока что просто помните, что пользователи и схемы — это не одно и то же, и в действительности один пользователь может быть владельцем нескольких схем.

Имена столбцов

Если в SQL-инструкции указано имя столбца, обычно SQL сам в состоянии определить, в какой из указанных в этой же инструкции таблиц содержится данный столбец. Однако если в инструкцию требуется включить два столбца из различных таблиц, но с одинаковыми именами, необходимо указать *квалифицированные имена столбцов*, которые однозначно определяют их местонахождение. Такое квалифицированное имя столбца состоит из имени таблицы, содержащей столбец, и имени столбца, разделенных точкой. Например, полное имя столбца SALES из таблицы SALESREPS имеет такой вид.

```
SALESREPS.SALES
```

Если столбец находится в таблице, владельцем которой является другой пользователь, то в квалифицированном имени столбца следует использовать квалифицированное имя таблицы. Например, полное имя столбца BIRTH_DATE в таблице BIRTHDAYS, владельцем которой является пользователь SAM, имеет следующий вид.

```
SAM.BIRTHDAYS.BIRTH_DATE
```

Квалифицированное имя столбца можно использовать вместо короткого имени в инструкциях SQL там, где используется простое (неквалифицированное) имя; об исключениях говорится при описании конкретных инструкций SQL.

Типы данных

В стандарте ANSI/ISO SQL описываются различные типы данных, которые могут храниться в SQL-базе данных и обрабатываться с помощью SQL. Исходный стандарт SQL1 определял лишь минимальный набор типов данных. Последующие версии стандарта расширили список, добавив в него строки переменной длины, дату и время, битовые строки, XML и другие типы данных. Современные коммерческие СУБД в состоянии работать с данными самых разных типов, причем между наборами типов данных у разных СУБД имеются существенные отличия. Ниже рассмотрим типичные типы данных.

- **Целые числа**. Обычно это данные о ценах, количествах, возрасте и т.п. Целочисленные столбцы часто используются также для хранения идентификаторов, таких как идентификатор клиента, служащего или номер заказа.
- **Десятичные числа**. Числа, имеющие дробную часть, но которые вычисляются точно, — например, курсы валют или проценты. Зачастую это денежные величины.

- **Числа с плавающей точкой**. Величины, которые можно вычислить приблизительно, такие как вес или расстояние. Числа с плавающей точкой могут представлять более широкий диапазон значений, чем десятичные числа, но при работе с ними возможны погрешности округления.
- **Строки символов постоянной длины**. Обычно это строки, имеющие одну и ту же длину, — например, почтовые коды, аббревиатуры стран или штатов, краткие описания и т.п. Если реальная строка оказывается короче, чем строки, хранящиеся в данном столбце, она дополняется пробелами, с тем чтобы ее длина соответствовала указанной в описании типа данных.
- **Строки символов переменной длины**. Строки символов, длина которых изменяется от строки к строке до некоторого максимального значения. (В стандарте SQL1 были определены только строки постоянной длины, которые проще в работе, но требуют значительно больше пространства для хранения.) Столбцы с этим типом данных обычно хранят имена людей или названия компаний, адреса, описания и т.д. В отличие от строк постоянной длины, здесь нет дополнения пробелами — хранится только необходимое количество символов, а также длина строки.
- **Денежные величины**. Во многих СУБД поддерживается тип данных MONEY или CURRENCY, который обычно хранится в виде десятичного числа или числа с плавающей точкой. Наличие отдельного типа данных для представления денежных величин позволяет правильно форматировать их при выводе на экран. Однако в стандарте SQL такие типы данных не определяются.
- **Дата и время**. Поддержка значений даты/времени также широко распространена в различных СУБД, хотя способы ее реализации довольно сильно отличаются друг от друга, поскольку до принятия стандарта различные производители по-разному реализовывали этот тип данных. Обычно поддерживаются различные комбинации дат, времени, временных интервалов и арифметических действий над этими величинами. Стандарт SQL включает детальную спецификацию типов данных DATE, TIME, TIMESTAMP и INTERVAL, а также поддержку часовых поясов и точного времени (например, десятые или сотые доли секунды).
- **Логические величины**. Одни СУБД, в частности Microsoft SQL Server, явно поддерживают логические значения (TRUE или FALSE), а другие разрешают выполнять в инструкциях SQL логические операции (сравнение, логическое И/ИЛИ и другие) над данными.
- **Длинные символьные объекты**. В стандарт SQL:1999 добавлен тип данных CLOB, который обеспечивает хранение больших символьных строк, до предельных размеров, которые обычно составляют несколько гигабайтов. Это позволяет хранить в базе данных целые документы, описания товаров, технические статьи, резюме и другие неструктурированные текстовые данные. Ряд SQL-баз данных поддерживает собственные типы данных (добавленные до введения стандарта SQL:1999) для хранения

длинных текстовых строк (обычно до 32000 или 65000 символов, а в ряде случаев — и того больше). Обычно СУБД ограничивает применение таких столбцов в интерактивных запросах и поисках.

- **Большие бинарные объекты**. В стандарт SQL:1999 добавлен тип данных BLOB, который поддерживает хранение неструктурированных последовательностей байтов переменной длины. Столбцы, содержащие такие данные, используются для хранения видеоизображений, выполнимого кода и прочей неструктурированной информации. До публикации стандарта разработчики реализовывали собственные решения, такие как типы данных IMAGE в SQL Server или LONG RAW в Oracle, которые могут хранить до 2 Гбайт данных.
- **Азиатские символы**. С ростом поддержки базами данных глобальных приложений производители СУБД добавляют поддержку в строках фиксированной и переменной длины многобайтовых символов, используемых для представления иероглифического письма или арабских символов. Для этого использовались собственные типы данных, такие как GRAPHIC и VARGRAPHIC в SQL Server. В настоящее время стандарт ANSI/ISO определяет различные символьные типы данных для наборов национальных символов (NCHAR, NVARCHAR и NCLOB). Хотя большинство современных баз данных поддерживает сохранение и выборку таких символов (зачастую с применением UNICODE для их представления), поддержка поиска и сортировки достаточно сильно варьируется в различных базах данных.

В табл. 5.4 перечислены типы данных, определенные в стандарте ANSI/ISO.

Таблица 5.4. Типы данных ANSI/ISO SQL

Тип данных	Сокращение	Описание
CHARACTER (*длина*)	CHAR	Строки символов постоянной длины
CHARACTER VARYING (*длина*)	CHAR VARYING, VARCHAR	Строки символов переменной длины
CHARACTER LARGE OBJECT (*длина*)	CLOB	Большие строки символов переменной длины
NATIONAL CHARACTER (*длина*)	NATIONAL CHAR, NCHAR	Строки символов постоянной длины с наборами национальных символов
NATIONAL CHARACTER VARYING (*длина*)	NATIONAL CHAR VARYING, NCHAR	Строки символов переменной длины с наборами национальных символов
NATIONAL CHARACTER LARGE OBJECT (*длина*)	NCLOB	Большие строки символов переменной длины с наборами национальных символов
BIT (*длина*)		Битовые строки постоянной длины
BIT VARYING (*длина*)		Битовые строки переменной длины
INTEGER	INT	Целые числа
SMALLINT		Малые целые числа
NUMERIC (*точность, масштаб*)		Десятичные числа
DECIMAL (*точность, масштаб*)	DEC	Десятичные числа

Окончание табл. 5.4

Тип данных	Сокращение	Описание
FLOAT (*точность*)		Числа с плавающей точкой
REAL		Числа с плавающей точкой малой точности
DOUBLE PRECISION		Числа с плавающей точкой большой точности
DATE		Календарные даты
TIME (*точность*)		Время
TIME WITH TIME ZONE (*точность*)		Поясное время
TIMESTAMP (*точность*)		Дата и время
TIMESTAMP WITH TIME ZONE (*точность*)		Дата и поясное время
INTERVAL		Временные интервалы
XML (*модификатор типа* [*вторичный модификатор типа*])		Символьные данные в XML-формате

Различия в поддержке типов данных в разных реализациях SQL препятствуют переносимости SQL-приложений. Причины подобных различий следует искать в самом пути, по которому развивались реляционные базы данных. Вот типичная схема такого пути.

- Производитель СУБД добавил в свой продукт новый тип данных, который обеспечивает полезные новые возможности для определенной группы пользователей.
- Другие производители ввели поддержку того же типа данных, но с небольшими модификациями, чтобы их нельзя было обвинить в слепом копировании.
- По прошествии нескольких лет роста популярности нового типа данных он появляется в большинстве ведущих СУБД, став частью "джентльменского набора" базовых типов данных, поддерживаемых основными реализациями SQL.
- Далее этой идеей начинают интересоваться комитеты по стандартизации, задачей которых является устранение произвольных различий в реализациях ведущих СУБД. Но чем больше таких различий, тем труднее найти компромисс. Как правило, результатом деятельности комитета является вариант, который не соответствует в точности *ни одной* из реализаций.
- Производители СУБД начинают потихоньку внедрять поддержку полученного стандартизированного типа данных, но поскольку они располагают обширной базой уже инсталлированных продуктов, то вынуждены сопровождать и свой собственный старый вариант типа данных.
- По прошествии длительного времени (обычно включающего выпуск нескольких новых версий СУБД) пользователи, наконец, полностью переходят к использованию стандартного варианта рассматриваемого типа данных и производитель СУБД может начать процесс удаления поддержки старого варианта из своего продукта.

В качестве отличного примера рассмотрим форматы представления даты и времени. Так, DB2 давно поддерживает представление дат и времени при помощи трех различных типов данных.

- DATE — Хранит дату наподобие "June 30, 2008"
- TIME — Хранит время суток наподобие "12:30:00 p.m."
- TIMESTAMP — Представляет конкретный момент времени с точностью до наносекунд

Значения даты и времени можно представлять в виде строковых констант. Кроме того, поддерживаются арифметические операции над значениями даты. Ниже приведен пример с использованием дат DB2, в котором предполагается, что в столбце HIRE_DATE содержатся данные типа DATE.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE >= '05/30/2007' + 15 DAYS;
```

В СУБД SQL Server введен единый тип данных для представления даты и времени — DATETIME, который напоминает тип данных TIMESTAMP из DB2. Если столбец HIRE_DATE имеет тип DATETIME, в этой СУБД можно выполнить такой запрос.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE >= '06/14/2007';
```

Поскольку в запросе не указано конкретное время 14 июня 2007 года, SQL Server по умолчанию считает, что время соответствует полуночи. Таким образом, запрос для SQL Server *в действительности* означает следующее.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE >= '06/14/2007 12:00AM';
```

Кроме того, SQL Server поддерживает арифметические операции над датами с помощью набора встроенных функций. Так, рассматриваемый выше запрос из DB2 можно переписать для SQL Server следующим образом.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE >= DATEADD(DAY, 15, '05/30/2007');
```

Это, конечно же, значительно отличается от синтаксиса DB2.

СУБД Oracle издавна поддерживает единственный тип данных для представления даты и времени, который называется DATE (заметим, что начиная с Oracle 9*i* в нем поддерживаются стандартные типы SQL DATETIME и TIMESTAMP). Как и тип данных DATETIME в SQL Server, тип данных DATE в Oracle фактически соответствует типу данных TIMESTAMP из DB2. Как и в SQL Server, временная часть значения типа DATE по умолчанию принимается равной полуночи. Формат даты, принятый в Oracle по умолчанию, отличается от форматов, принятых в DB2 и SQL Server, поэтому версия запроса для Oracle имеет следующий вид.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE >= '14-JUN-07';
```

Oracle также, хотя и с некоторыми ограничениями, поддерживает арифметические операции над датами, поэтому исходный запрос можно повторить, но с тем отличием, что здесь не используется ключевое слово DAYS из DB2.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE >= '30-MAY-07' + 15;
```

Заметим, однако, что эта инструкция требует от СУБД неявного преобразования строки в соответствующий тип данных, прежде чем будет выполнено суммирование, и что не все реализации SQL поддерживают такое преобразование. Например, Oracle сообщит об ошибке, если перед выполнением арифметических действий не будет применена функция TO_DATE или CAST, преобразующая символьную строку в тип данных Oracle DATE или DATETIME.

К счастью, в связи с наступлением 2000 года большинство производителей СУБД добавили универсальную поддержку дат в инструкциях SQL с годом из четырех цифр в стандартном формате YYYY-MM-DD, который мы и будем использовать в большинстве примеров в данной книге. В случае Oracle формат по умолчанию остается тем же, что и в приведенных ранее примерах, но его можно изменить одной командой в сессии базы данных или пользователя. Если вы работаете с Oracle и хотите выполнить пример из данной книги, просто введите приведенную команду для изменения формата даты по умолчанию.

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD';
```

При формировании запроса поиска точной даты с применением оператора равенства (=) следует быть осторожным, поскольку соответствующие данные хранят еще и время суток. Рассмотрим следующий пример.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE = '06/14/2007';
```

Если информация о дате приема служащего на работу была сохранена в базе данных в полдень 14 июня 2007 года, то строка, содержащая сведения об этом человеке, не попадет в результаты запроса в случае баз данных Oracle или SQL Server. СУБД полагает, что к строке даты в инструкции SQL следует добавить время, соответствующее полночи, а так как полдень и полночь — "две большие разницы", данная строка не будет отобрана. С другой стороны, она будет отобрана в DB2 при хранении данных в формате DATE.

Наконец, начиная с SQL2 в стандарт ANSI/ISO был введен набор типов данных для работы с датой и временем, основанных на рассмотренных типах данных из DB2, но не идентичных им. В дополнение к типам данных DATE, TIME и TIMESTAMP стандарт определяет тип данных INTERVAL, предназначенный для хранения значений интервалов времени (например, продолжительность какого-то процесса, измеренная в днях, часах, минутах и секундах). В стандарте определены тщательно продуманные сложные методы для выполнения арифметических операций над

значениями даты и времени, принципы задания точности вычисления интервалов времени, учета разницы между часовыми поясами и т.д. Большинство реализаций SQL в настоящее время поддерживает эти стандартные типы данных. Одно важное исключение состоит в том, что SQL Server давно использует тип данных TIMESTAMP для совершенно иной цели, так что поддержка спецификации ANSI/ISO в настоящее время весьма сомнительна.

Как иллюстрируют приведенные примеры, незначительные отличия в реализации типов данных приводят к значительным отличиям в синтаксисе инструкций SQL.

Эти отличия могут даже привести к тому, что, выполнив один и тот же запрос в различных СУБД, можно получить немного отличающиеся результаты. Таким образом, повсеместно восхваляемая переносимость SQL существует только в самом общем смысле. Приложение и в самом деле можно перенести с одной СУБД на другую, и оно может быть высокопереносимо только при использовании основных, широко распространенных, возможностей SQL. Однако небольшие отличия в реализациях SQL приводят к тому, что типы данных и инструкции SQL при переносе почти всегда приходится несколько видоизменять. Чем сложнее приложение, тем вероятнее его зависимость от возможностей и нюансов конкретной СУБД и тем менее оно переносимо.

Константы

В некоторых инструкциях SQL необходимо указывать значения чисел, строк или даты в текстовом виде. Например, рассмотрим следующую инструкцию INSERT, которая добавляет в базу данных имя нового служащего.

```
INSERT INTO SALESREPS(EMPL_NUM, NAME, QUOTA,
                      HIRE_DATE, SALES)
     VALUES(115, 'Dennis Irving', 175000.00,
            '2008-06-21', 0.00);
```

Здесь в предложении VALUES указаны значения для каждого столбца во вновь вставляемой строке. Константы используются и в выражениях, как в приведенной ниже инструкции SELECT.

```
SELECT CITY
  FROM OFFICES
 WHERE TARGET > (1.1 * SALES) + 10000.0;
```

В стандарте ANSI/ISO определен формат числовых и строковых констант, или *литералов*, которые представляют конкретные значения данных. Этот формат используется в большинстве реализаций SQL.

Числовые константы

Целые и десятичные константы (известные также как *точные числовые литералы*) записываются в инструкциях SQL как обычные десятичные числа с необязательным ведущим знаком плюс (+) или минус (-).

```
21   -375   2000.00   +497500.8778
```

В числовых константах нельзя ставить символы разделения разрядов между цифрами; кроме того, не все диалекты SQL разрешают ставить перед числом знак плюс, так что лучше этого избегать. В случае с данными, представляющими денежные величины, в большинстве реализаций SQL используются просто целые или десятичные константы, хотя в некоторых из них перед константой можно указывать символ денежной единицы.

```
$0.75   $5000.00   $-567.89
```

Константы с плавающей точкой (известные также как *приблизительные числовые литералы*) определяются с помощью символа E и имеют такой же формат, как и в распространенных языках программирования типа C или FORTRAN. Ниже приведены примеры констант с плавающей точкой.

```
1.5E3   -3.14159E1   2.5E-7   0.783926E21
```

Символ E читается как "умножить на десять в степени", так что первая константа представляет число "1.5 умножить на десять в степени 3", или 1500.

Строковые константы

В соответствии со стандартом ANSI/ISO строковые константы в SQL должны быть заключены в одинарные кавычки ('...'), как показано в следующих примерах.

```
'Jones, John J.'   'New York'   'Western'
```

Если необходимо включить в строковую константу одинарную кавычку, вместо нее следует поставить две одинарные кавычки. Таким образом, константа

```
'I can''t'
```

представляет семисимвольную строку "I can't".

В некоторых реализациях SQL, например в SQL Server, допускаются строковые константы в двойных кавычках.

```
"Jones, John J."   "New York"   "Western"
```

К сожалению, употребление двойных кавычек вызывает проблемы при переносе программ в другие SQL-продукты. В стандарте SQL предоставляется дополнительная возможность определения строковых констант с применением национальных (например, французского или немецкого) или пользовательских наборов символов. Однако средства поддержки пользовательских наборов символов в ведущих СУБД практически не реализованы.

Константы даты и времени

В SQL-продуктах, которые поддерживают данные, представляющие собой дату и время, значения даты, времени и интервалов времени указываются как строковые константы. Форматы этих констант в различных СУБД отличаются друг от друга. Кроме того, способы записи даты и времени меняются в зависимости от страны.

СУБД IBM DB2 поддерживает несколько различных международных форматов для записи даты и времени (табл. 5.5). Выбор формата осуществляется при инстал-

ляции системы. Кроме того, в DB2 для представления интервалов времени используются специальные константы, как показано в следующем примере.

```
HIRE_DATE + 30 DAYS
```

Обратите внимание на то, что информацию об интервале времени нельзя сохранить в базе данных, поскольку в DB2 нет явного типа данных DURATION.

Таблица 5.5. Форматы даты и времени в SQL DB2

Формат	Формат даты	Пример даты	Формат времени	Пример времени
Американский	mm/dd/yyyy	5/19/1963	hh:mm am/pm	2:18 PM
Европейский	dd.mm.yyyy	19.5.1963	hh.mm.ss	14.18.08
Японский	yyyy-mm-dd	1963-5-19	hh:mm:ss	14:18:08
ISO	yyyy-mm-dd	1963-5-19	hh.mm.ss	14.18.08

СУБД SQL Server также поддерживает различные форматы констант даты и времени. Она автоматически воспринимает все форматы, так что вы даже можете при желании использовать все их одновременно. Ниже приведен ряд корректных в SQL Server констант даты

```
March 15, 1990   Mar 15 1990   3/15/1990   3-15-90   1990 MAR 15
```

и констант времени.

```
15:30:25   3:30:25 PM   3:30:25 pm   3 PM
```

Значения дат и времени в Oracle также записываются в виде строковых констант с использованием следующего формата.

```
15-MAR-90
```

Кроме того, в Oracle можно использовать встроенную функцию TO_DATE() для преобразования констант даты, записанных в другом формате.

```
SELECT NAME, AGE
  FROM SALESREPS
 WHERE HIRE_DATE = TO_DATE('JUN 14 1990', 'MON DD YYYY');
```

В стандарте SQL2 определен формат констант даты и времени, совпадающий с форматом ISO в табл. 5.5, за исключением того, что в константах времени для разделения часов, минут и секунд используются двоеточия, а не точки. Стандартный тип данных SQL TIMESTAMP, не показанный в таблице, имеет формат yyyy-mm-dd-hh.mm.ss.nnnnnn — например, "1963-05-19-19.18.08.048632" представляет примерно 7 часов 18 минут пополудни 19 мая 1963 года.

Символьные константы

Кроме пользовательских констант, в SQL имеются специальные именованные константы, возвращающие значения, хранимые в самой СУБД. Например, значение константы CURRENT_DATE, реализованной в ряде СУБД, всегда равно текущей

дате и может использоваться в таких запросах, как показанный ниже, где выполняется поиск сотрудников, принятых на работу в будущем.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE > CURRENT_DATE;
```

В стандарте SQL1 определена только одна символьная константа (константа USER, которая будет рассмотрена в главе 15, "SQL и безопасность"), но в большинстве СУБД их количество гораздо больше. В общем случае именованную константу можно применять в любом месте инструкции SQL, в котором разрешается использовать обычную пользовательскую константу того же типа. В стандарт SQL2 вошли наиболее полезные константы из различных реализаций SQL, в частности константы CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, а также USER, SESSION_USER и SYSTEM_USER.

В некоторых SQL-продуктах, включая SQL Server, доступ к системным значениям обеспечивается с помощью не именованных констант, а встроенных функций. Версия предыдущего запроса для SQL Server имеет такой вид.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE HIRE_DATE > GETDATE();
```

Встроенные функции описаны ниже в настоящей главе.

Выражения

Выражения в SQL используются для выполнения операций над значениями, извлекаемыми из базы данных или используемыми при поиске в базе данных. Например, в следующем запросе вычисляется процентное соотношение объема и плана продаж для каждого офиса.

```
SELECT CITY, TARGET, SALES, (SALES/TARGET) * 100
  FROM OFFICES;
```

А этот запрос возвращает список офисов, объем продаж которых превышает план на $50000 или больше.

```
SELECT CITY
  FROM OFFICES
 WHERE SALES > TARGET + 50000.00;
```

В соответствии со стандартом ANSI/ISO в выражениях можно использовать четыре арифметические операции: сложение (X+Y), вычитание (X-Y), умножение (X*Y) и деление (X/Y). Для формирования сложных выражений можно использовать скобки.

```
(SALES * 1.05) - (TARGET * .95)
```

Строго говоря, в приведенном выше выражении скобки не требуются, поскольку в соответствии со стандартом ANSI/ISO умножение и деление имеют более высокий приоритет, чем сложение и вычитание. Однако вы всегда должны использо-

вать скобки, чтобы сделать ваше выражение однозначным. Дело в том, что различные диалекты SQL могут использовать разные правила. Скобки, кроме того, повышают удобочитаемость инструкций и облегчают сопровождение SQL-кода.

В стандарте ANSI/ISO определено также, что преобразование целых чисел в десятичные и десятичных чисел в числа с плавающей точкой должно происходить автоматически. Таким образом, в одном выражении можно использовать числовые данные разных типов.

Во многих реализациях SQL допускается выполнение операций над датами и строками. Стандарт SQL определяет оператор конкатенации строк, записываемый в виде двух вертикальных черточек (||), который поддерживается в большинстве реализаций (важным исключением является SQL Server, в которой для этой цели применяется оператор плюс (+)). Если в двух столбцах с именами FIRST_NAME и LAST_NAME содержатся значения "Jim" и "Jackson", выражение DB2

```
('Mr./Mrs. ' || FIRST_NAME || ' ' || LAST_NAME)
```

вернет строку "Mr./Mrs. Jim Jackson". Как уже упоминалось, DB2 и многие другие реализации поддерживают также операции сложения и вычитания значений типа DATE, TIME и TIMESTAMP, когда эти операции имеют смысл. Эта возможность включена и в стандарт SQL.

Встроенные функции

В стандарте SQL определено множество *встроенных функций*; при этом в большинстве реализаций SQL добавлены собственные встроенные функции. Многие из них выполняют различные преобразования типов данных. Например, встроенные функции MONTH() и YEAR() из СУБД DB2 принимают в качестве аргумента значения DATE или TIMESTAMP и возвращают целое число, представляющее соответственно месяц или год из заданного аргумента. Запрос, приведенный ниже, перечисляет имена и месяцы приема на работу каждого служащего, данные о котором содержатся в нашей учебной базе данных.

```
SELECT NAME, MONTH(HIRE_DATE)
  FROM SALESREPS;
```

А этот запрос возвращает список служащих, нанятых на работу в 2006 году.

```
SELECT NAME, MONTH(HIRE_DATE)
  FROM SALESREPS
 WHERE YEAR(HIRE_DATE) = 2006;
```

Кроме того, встроенные функции часто используются для форматирования данных. Например, встроенная функция TO_CHAR() из СУБД Oracle принимает в качестве аргументов значение типа DATE и спецификацию формата, а возвращает строку, содержащую значение даты, отформатированное в соответствии со спецификацией. В результатах, возвращаемых запросом

```
SELECT NAME, TO_CHAR(HIRE_DATE, 'DAY MONTH DD, YYYY')
  FROM SALESREPS
```

благодаря использованию встроенной функции TO_CHAR() все даты приема на работу будут иметь формат "Wednesday June 14, 2007".

В общем случае встроенную функцию можно использовать в любом месте инструкции SQL, в котором можно использовать константу того же типа данных. Здесь невозможно перечислить все встроенные функции, поддерживаемые распространенными диалектами SQL, поскольку их слишком много. В DB2 их около двух десятков, столько же, но своих встроенных функций, и в Oracle, а в SQL Server — еще больше. В стандарт SQL2 вошли наиболее полезные функции из этих реализаций SQL, зачастую с несколько отличным синтаксисом. Эти функции перечислены в табл. 5.6.

Таблица 5.6. Стандартные встроенные функции SQL

Функция	Возвращает
BIT_LENGTH(*строка*)	Количество битов в битовой *строке*
CAST(*значение* AS *тип данных*)	*Значение*, преобразованное в указанный *тип данных* (например, дата, преобразованная в строку)
CHAR_LENGTH(*строка*)	Длина *строки* символов
CONVERT(*строка* USING *функция*)	*Строка*, преобразованная в соответствии с указанной *функцией*
CURRENT_DATE	Текущая дата
CURRENT_TIME(*точность*)	Текущее время с указанной *точностью*
CURRENT_TIMESTAMP(*точность*)	Текущие дата и время с указанной *точностью*
EXTRACT(*часть* FROM *значение*)	Указанная *часть* (DAY, HOUR и т.д.) из *значения* типа DATETIME
LOWER(*строка*)	*Строка*, переведенная в нижний регистр
OCTET_LENGTH(*строка*)	Число 8-битовых байтов в *строке* символов
POSITION(*подстрока* IN *строка*)	Позиция, с которой начинается вхождение *подстроки* в *строку*
SUBSTRING(*строка* FROM n FOR *длина*)	Часть *строки*, начинающаяся с *n*-го символа и имеющая указанную *длину*
TRANSLATE(*строка* USING *функция*)	*Строка*, транслированная с помощью указанной *функции*
TRIM(BOTH *символ* FROM *строка*)	*Строка*, из которой удалены ведущие и конечные указанные *символы*
TRIM(LEADING *символ* FROM *строка*)	*Строка*, из которой удалены ведущие указанные *символы*
TRIM(TRAILING *символ* FROM *строка*)	*Строка*, из которой удалены конечные указанные *символы*
UPPER(*строка*)	*Строка*, переведенная в верхний регистр

Отсутствующие данные (значения NULL)

Поскольку база данных обычно представляет собой модель реального мира, отдельные элементы данных в ней неминуемо будут отсутствовать, будут неизвестными или неприменимыми. Например, в столбце QUOTA таблицы SALESREPS содержатся плановые объемы продаж для каждого служащего. Однако для одного из новых служащих план еще не был утвержден; соответствующая информация в базе данных отсутствует. Конечно, в столбец QUOTA для нового служащего можно ввести значение 0, однако это исказит ситуацию. План служащего не равен нулю, а просто "пока неизвестен".

Аналогично столбец MANAGER в таблице SALESREPS для каждого служащего содержит идентификатор его менеджера. Однако Сэм Кларк (Sam Clark), вице-президент по торговым операциям, не подчиняется никому из отдела сбыта. К нему этот столбец не относится. Можно ввести в столбец MANAGER для вице-президента число 0 или 9999, но ни одно из них не будет настоящим идентификатором начальника Сэма Кларка. Для этой строки не подходит ни одно число.

SQL поддерживает обработку отсутствующих, неизвестных или неприменимых данных с помощью концепции *отсутствующего значения*. Это значение является *индикатором*, который показывает, что в конкретной строке определенный элемент данных отсутствует или что столбец вообще не подходит для этой строки. Для удобства говорят, что значение такого элемента данных равно NULL. Однако NULL не является значением данных, как 0, 48300 или "Sam Clark". Напротив, это признак того, что точное значение данных неизвестно или отсутствует. На рис. 5.3 показано содержимое таблицы SALESREPS. Обратите внимание на то, что в столбцах QUOTA и REP_OFFICE для Тома Снайдера (Tom Snyder) и в столбце MANAGER для Сэма Кларка (Sam Clark) содержатся значения NULL. Обратите внимание, что разные SQL-инструменты по-разному выводят это значение: одни — как NULL, другие — как пустую или заполненную пробелами строку.

Таблица SALESREPS

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	12-FEB-88	104	$350,000.00	$367,911.00
109	Mary Jones	31	11	Sales Rep	12-OCT-89	106	$300,000.00	$392,725.00
102	Sue Smith	48	21	Sales Rep	10-DEC-86	108	$350,000.00	$474,050.00
106	Sam Clark	52	11	VP Sales	14-JUN-88	NULL	$275,000.00	$299,912.00
104	Bob Smith	33	12	Sales Mgr	19-MAY-87	106	$200,000.00	$142,594.00
101	Dan Roberts	45	12	Sales Rep	20-OCT-86	104	$300,000.00	$305,673.00
110	Tom Snyder	41	NULL	Sales Rep	13-JAN-90	101	NULL	$75,985.00
108	Larry Fitch	62	21	Sales Mgr	12-OCT-89	106	$350,000.00	$361,865.00
103	Paul Cruz	29	12	Sales Rep	14-NOV-88	108	$275,000.00	$286,775.00
107	Nancy Angelli	49	22	Sales Rep	14-NOV-88	108	$300,000.00	$186,042.00

Значение неизвестно

Значение неприменимо

Значение неизвестно

Рис. 5.3. Значения NULL в таблице SALESREPS

Во многих ситуациях значения NULL требуют от СУБД специальной обработки. Например, если пользователь просит вычислить сумму по столбцу QUOTA, что

СУБД должна делать со значениями NULL при вычислении суммы? Ответ на этот вопрос дает набор правил обработки значений NULL в различных инструкциях и предложениях SQL. Из-за необходимости включения таких правил в синтаксис языка SQL многие теоретики реляционных баз данных считают, что значения NULL использовать не следует. Другие, включая доктора Кодда, отстаивают использование различных значений NULL для ситуаций, когда данные неизвестны или неприменимы.

Независимо от академических споров, значения NULL стали частью стандарта ANSI/ISO и реализованы почти во всех коммерческих СУБД. Они играют важную практическую роль в производстве SQL-баз данных. О специальных правилах обработки значений NULL (и отличиях, существующих при обработке таких значений в различных СУБД) рассказывается позже в этой книге.

Резюме

В настоящей главе были описаны основные элементы SQL.

- Язык SQL включает около тридцати инструкций, каждая из которых состоит из команды и одного или нескольких предложений. Каждая инструкция выполняет одно конкретное действие.
- В SQL-базах данных могут храниться значения различных типов, включая текстовые данные, целые и десятичные числа, числа с плавающей точкой и данные ряда других типов, введенных конкретным производителем.
- Инструкции SQL могут включать в себя выражения, в которых над именами столбцов, константами и встроенными функциями выполняются арифметические и другие операции.
- Разнообразие типов данных, констант и встроенных функций делает переносимость инструкций SQL более сложной задачей, чем это может показаться на первый взгляд.
- Значения NULL используются для обработки отсутствующих или неприменимых элементов данных в SQL.

ГЛАВА 6

Простые запросы

Сердцем языка SQL являются запросы. Инструкция SELECT, которая используется для создания SQL-запросов, является наиболее мощной и сложной из всех инструкций SQL. Несмотря на богатство возможностей этой инструкции, ее изучение можно начать с создания простейших запросов, а затем постепенно увеличивать их сложность. В этой главе рассказывается о самых простых SQL-запросах — запросах на чтение данных из отдельных строк одной таблицы базы данных. Если вам не приходилось заниматься этим раньше, то изучение материала будет более эффективно, если вы создадите учебную базу данных в своей системе и будете пытаться выполнить все рассматриваемые здесь запросы. Руководство по созданию учебной базы данных можно найти в приложении А, "Учебная база данных".

Инструкция SELECT

Инструкция SELECT извлекает информацию из базы данных и возвращает ее в виде результатов запроса. Точный вид результатов зависит от конкретной используемой СУБД. В кратком введении в SQL в главе 2, "Краткий обзор SQL", уже приводились примеры инструкций SELECT. Вот еще несколько образцов запросов, извлекающих данные об офисах.

Вывести список офисов с их плановыми и фактическими объемами продаж.

```
SELECT CITY, TARGET, SALES
  FROM OFFICES;

CITY                TARGET          SALES
-----------    -----------    -----------
Denver         $300,000.00    $186,042.00
New York       $575,000.00    $692,637.00
Chicago        $800,000.00    $735,042.00
Atlanta        $350,000.00    $367,911.00
Los Angeles    $725,000.00    $835,915.00
```

Вывести список офисов, расположенных в восточном регионе, с их плановыми и фактическими объемами продаж.

```
SELECT CITY, TARGET, SALES
  FROM OFFICES
 WHERE REGION = 'Eastern';

CITY             TARGET           SALES
--------     -----------      -----------
New York     $575,000.00      $692,637.00
Chicago      $800,000.00      $735,042.00
Atlanta      $350,000.00      $367,911.00
```

Вывести список офисов в восточном регионе, в которых фактические объемы продаж превысили плановые; отсортировать список в алфавитном порядке по названиям городов.

```
SELECT CITY, TARGET, SALES
  FROM OFFICES
 WHERE REGION = 'Eastern'
   AND SALES > TARGET
 ORDER BY CITY;

CITY             TARGET           SALES
--------     -----------      -----------
Atlanta      $350,000.00      $367,911.00
New York     $575,000.00      $692,637.00
```

В случае простых запросов инструкция SELECT очень похожа на предложение на английском языке. Когда запросы становятся сложнее, требуется использовать большее количество возможностей инструкции SELECT, чтобы точно указать, что именно запрашивается.

На рис. 6.1 приведена синтаксическая диаграмма инструкции SELECT. Инструкция состоит из шести предложений. Предложения SELECT и FROM являются обязательными; четыре остальные включаются в инструкцию только при необходимости. Ниже перечислены функции каждого из предложений.

- В предложении SELECT перечисляются элементы данных, которые должны быть выбраны инструкцией SELECT. Это могут быть либо столбцы базы данных, либо столбцы, вычисляемые при выполнении запроса. Предложение SELECT описано далее в настоящей главе.
- В предложении FROM указывается список таблиц и представлений, которые содержат элементы данных, извлекаемые запросом (представления рассматриваются в главе 14, "Представления"). Запросы, извлекающие данные из одной таблицы, описаны в настоящей главе. Более сложные запросы, извлекающие данные из двух или более таблиц, рассматриваются в главе 7, "Многотабличные запросы (соединения)".
- Предложение WHERE указывает, что в результаты запроса следует включать только некоторые строки. Для отбора строк, включаемых в результаты запроса, используется *условие отбора*. Основные возможности этого предложения описаны ниже в настоящей главе. Использование в предложении WHERE вложенных подзапросов рассматривается в главе 9, "Подзапросы и выражения с запросами".

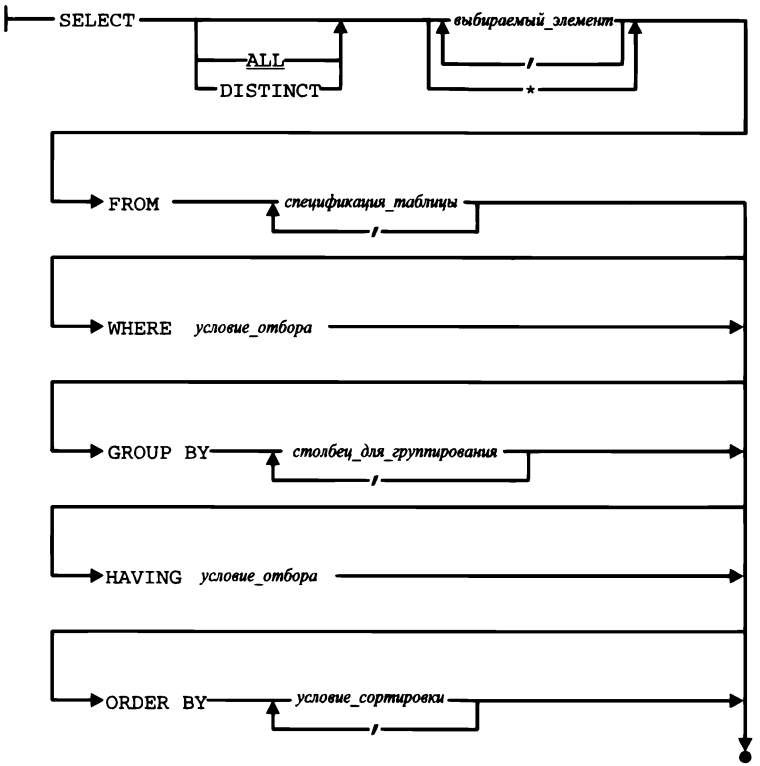


Рис. 6.1. Синтаксическая диаграмма инструкции SELECT

- Предложение GROUP BY позволяет создать итоговый запрос. Вместо генерации одной строки результата для каждой строки данных в базе данных, итоговый запрос вначале группирует строки базы данных по определенному признаку, а затем включает в результаты запроса одну итоговую строку для каждой группы. Итоговые запросы рассматриваются в главе 8, "Итоговые запросы".
- Предложение HAVING указывает, что в результаты запроса следует включать только некоторые из групп, созданных с помощью предложения GROUP BY. В этом предложении, как и в предложении WHERE, для отбора включаемых групп используется условие отбора. Предложение HAVING описано в главе 8, "Итоговые запросы".
- Предложение ORDER BY сортирует результаты запроса на основании данных, содержащихся в одном или нескольких столбцах. Если это предложение отсутствует, результаты запроса не будут отсортированы. Предложение ORDER BY рассматривается далее в настоящей главе.

Предложение SELECT

В предложении SELECT, с которого начинаются все инструкции SELECT, необходимо указать элементы данных, которые будут возвращены запросом. Эти эле-

менты задаются в виде *списка выбора*, состоящего из *выбираемых элементов*, разделенных запятыми. Для каждого элемента из этого списка в таблице результатов запроса будет создан один столбец; при этом столбцы в таблице результатов будут расположены в том же порядке слева направо, что и элементы списка возвращаемых столбцов. Возвращаемый элемент может представлять собой следующее.

- *Имя столбца*, идентифицирующее один из столбцов, содержащихся в таблицах, которые перечислены в предложении FROM. СУБД просто берет значение этого столбца для каждой из строк таблицы и помещает его в соответствующую строку таблицы результатов запроса.
- *Константу*, показывающую, что в каждой строке результатов запроса должно содержаться одно и то же значение.
- *Выражение*, указывающее, что SQL должен вычислить значение, помещаемое в результаты запроса, по формуле, определенной в выражении.

Все типы возвращаемых столбцов описаны далее в настоящей главе.

Предложение FROM

Предложение FROM состоит из ключевого слова FROM, за которым следует список спецификаций таблиц, разделенных запятыми. Каждая спецификация таблицы идентифицирует таблицу или представление, содержащие данные, которые извлекает запрос. Эти таблицы называются *исходными таблицами* запроса (и инструкции SELECT), поскольку они являются источниками всех данных, содержащихся в таблице результатов запроса. Во всех запросах, рассматриваемых в настоящей главе, в предложении FROM указана одна таблица.

Результаты запроса

Результатом SQL-запроса на выборку всегда является таблица, содержащая данные и ничем не отличающаяся от таблиц базы данных. Если пользователь набирает инструкцию SQL в интерактивном режиме, СУБД выводит результаты запроса (которые некоторые производители именуют *результирующим набором* (result set)) на экран в табличной форме. Если программа посылает запрос СУБД с помощью программного SQL, то СУБД возвращает таблицу результатов запроса программе. В любом случае результаты запроса всегда имеют такой же формат, как и обычные таблицы, содержащиеся в базе данных, как показано на рис. 6.2. Обычно результаты запроса представляют собой таблицу с несколькими строками и столбцами. Например, запрос, приведенный ниже, возвращает таблицу из трех столбцов (поскольку запрашиваются три элемента данных) и десяти строк (по количеству служащих).

Вывести список имен, офисов и дат приема на работу всех служащих.

```
SELECT NAME, REP_OFFICE, HIRE_DATE
  FROM SALESREPS;

NAME            REP_OFFICE   HIRE_DATE
```

```
-------------   ----------   ---------
Bill Adams              13  2006-02-12
Mary Jones              11  2007-10-12
Sue Smith               21  2004-12-10
Sam Clark               11  2006-06-14
Bob Smith               12  2005-05-19
Dan Roberts             12  2004-10-20
Tom Snyder            NULL  2008-01-13
Larry Fitch             21  2007-10-12
Paul Cruz               12  2005-03-01
Nancy Angelli           22  2006-11-14
```

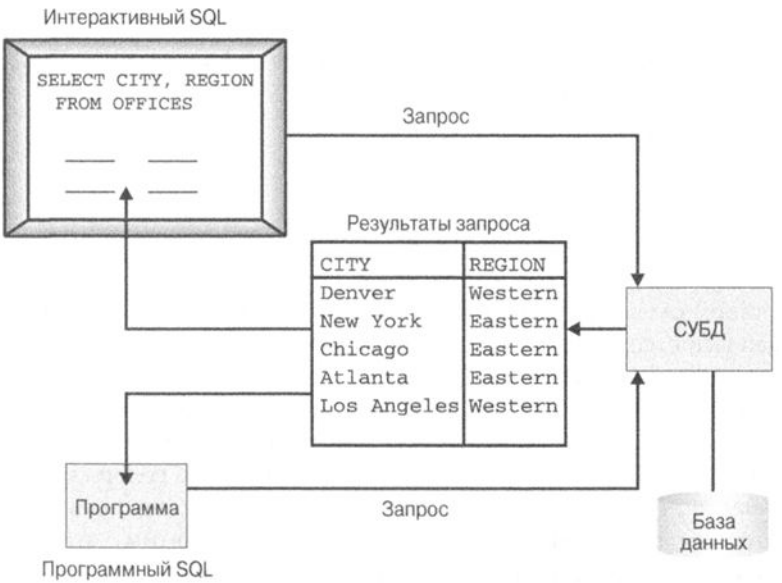


Рис. 6.2. Табличная структура результатов SQL-запроса

В отличие от запроса, показанного выше, следующий запрос возвращает только одну строку, так как есть всего один служащий, имеющий указанный идентификатор. Хотя результаты этого запроса, содержащие всего одну строку, имеют не такой "табличный" вид, как результаты, содержащие несколько строк, SQL все равно считает их таблицей, состоящей из трех столбцов и одной строки.

Имя, плановый и фактический объемы продаж служащего с идентификатором 107.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS
 WHERE EMPL_NUM = 107;

NAME                 QUOTA         SALES
-------------   -----------   -----------
Nancy Angelli   $300,000.00   $186,042.00
```

В некоторых случаях результатом запроса может быть единственное значение, как в следующем примере.

Среднее значение фактических объемов продаж по всем служащим компании.

```
SELECT AVG(SALES)
  FROM SALESREPS;

 AVG(SALES)
-----------
$289,353.20
```

Эти результаты запроса также являются таблицей, которая состоит из одного столбца и одной строки.

И наконец, запрос может вернуть результаты, содержащие *нуль* строк, как в следующем примере.

Список имен и дат приема на работу всех служащих, фактический объем продаж которых превышает $500 000.

```
SELECT NAME, HIRE_DATE
  FROM SALESREPS
 WHERE SALES > 500000.00;

NAME         HIRE_DATE
----------   ---------
```

Даже в таком случае результаты запроса считаются таблицей. Пустая таблица, приведенная выше, содержит два столбца и нуль строк.

Обратите внимание на то, что поддержка отсутствующих данных в SQL распространяется и на результаты запроса. Если один из элементов данных в таблице имеет значение NULL, то оно попадет в результаты запроса при извлечении этого элемента. Например, в таблице SALESREPS значение NULL содержится в столбцах QUOTA и MANAGER. Приведенный далее запрос возвращает эти значения во втором и третьем столбцах таблицы результатов запроса. Заметим, что не все SQL-продукты выводят значения NULL таким образом — Oracle и DB2, например, встретив значение NULL, не выводят ничего.

Список служащих с их плановыми объемами продаж и менеджерами.

```
SELECT NAME, QUOTA, MANAGER
  FROM SALESREPS;

NAME                 QUOTA    MANAGER
-------------   -----------   -------
Bill Adams      $350,000.00       104
Mary Jones      $300,000.00       106
Sue Smith       $350,000.00       108
Sam Clark       $275,000.00      NULL
Bob Smith       $200,000.00       106
Dan Roberts     $300,000.00       104
Tom Snyder             NULL       101
Larry Fitch     $350,000.00       106
Paul Cruz       $275,000.00       104
Nancy Angelli   $300,000.00       108
```

То, что SQL-запрос всегда возвращает таблицу данных, очень важно. Это означает, что результаты запроса можно сохранить в базе данных в виде таблицы. Это

означает также, что результаты двух подобных запросов можно объединить в одну таблицу. И наконец, это говорит о том, что результаты запроса сами могут стать предметом дальнейших запросов. Таким образом, табличная структура реляционной базы данных тесно связана с реляционными запросами SQL. Таблицам можно посылать запросы, а запросы возвращают таблицы.

Простые запросы

Наиболее простые запросы извлекают данные из столбцов, расположенных в одной таблице базы данных. Например, следующий запрос извлекает из таблицы OFFICES три столбца.

Вывести для каждого из офисов список городов, регионов и объемов продаж.

```
SELECT CITY, REGION, SALES
  FROM OFFICES;

CITY          REGION         SALES
-----------  -------   -----------
Denver       Western   $186,042.00
New York     Eastern   $692,637.00
Chicago      Eastern   $735,042.00
Atlanta      Eastern   $367,911.00
Los Angeles  Western   $835,915.00
```

Инструкция SELECT для простых запросов, таких как показанный выше, состоит только из двух обязательных предложений. В предложении SELECT перечисляются имена требуемых столбцов; в предложении FROM указываются имена таблиц и представлений, содержащих эти столбцы.

Концептуально SQL обрабатывает запрос путем построчного просмотра таблицы, указанной в предложении FROM. Для каждой строки таблицы берутся значения из указанных в запросе столбцов и создается одна строка результатов запроса. Таким образом, таблица результатов простого запроса на выборку содержит одну строку данных для каждой строки исходной таблицы базы данных.

Вычисляемые столбцы

Кроме столбцов, значения которых извлекаются непосредственно из базы данных, SQL-запрос на выборку может содержать *вычисляемые столбцы*, значения которых определяются на основании значений, хранящихся в базе данных. Чтобы получить вычисляемый столбец, в списке возвращаемых столбцов необходимо указать выражение. Как было сказано в главе 5, "Основы SQL", выражения могут включать в себя операции сложения, вычитания, умножения и деления. Для построения более сложных выражений можно использовать скобки. Конечно, столбцы, участвующие в арифметическом выражении, должны содержать числовые данные. При попытке сложить, вычесть, умножить или разделить столбцы, содержащие текстовые данные, SQL выдаст сообщение об ошибке.

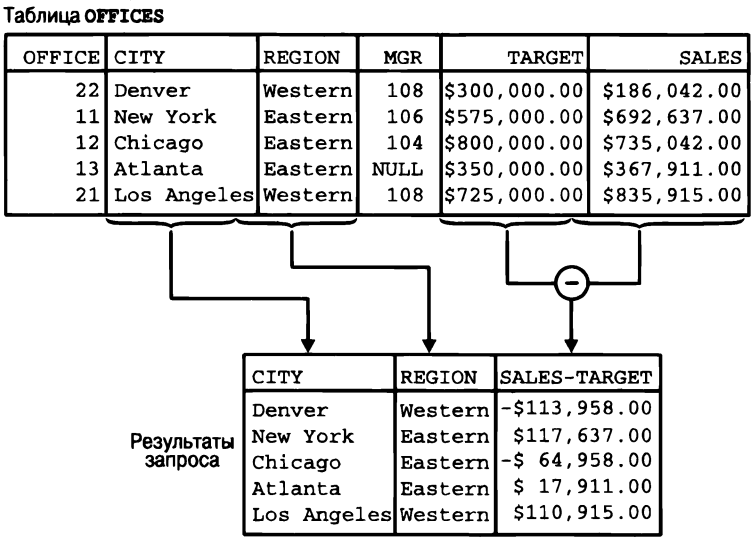
В следующем запросе будет получен простой вычисляемый столбец.

Выдать для каждого офиса список городов, регионов и сумм, на которые был перевыполнен/недовыполнен план по продажам.

```
SELECT CITY, REGION, (SALES - TARGET)
  FROM OFFICES;

CITY           REGION  (SALES-TARGET)
-----------    -------  -------------
Denver         Western    -$113,958.00
New York       Eastern     $117,637.00
Chicago        Eastern     -$64,958.00
Atlanta        Eastern      $17,911.00
Los Angeles    Western     $110,915.00
```

При выполнении этого запроса для каждой строки таблицы OFFICES генерируется одна строка результатов, как показано на рис. 6.3. Значения первых двух столбцов результатов запроса извлекаются непосредственно из таблицы OFFICES. Третий столбец для каждой строки результатов запроса вычисляется на основании значений столбцов текущей строки таблицы OFFICES.



OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300,000.00	$186,042.00
11	New York	Eastern	106	$575,000.00	$692,637.00
12	Chicago	Eastern	104	$800,000.00	$735,042.00
13	Atlanta	Eastern	NULL	$350,000.00	$367,911.00
21	Los Angeles	Western	108	$725,000.00	$835,915.00

CITY	REGION	SALES-TARGET
Denver	Western	-$113,958.00
New York	Eastern	$117,637.00
Chicago	Eastern	-$ 64,958.00
Atlanta	Eastern	$ 17,911.00
Los Angeles	Western	$110,915.00

Рис. 6.3. Выполнение запроса, содержащего вычисляемый столбец

Далее приведены другие примеры запросов, в которых используются вычисляемые столбцы.

Показать общую стоимость по каждому товару (показаны только 8 строк результирующего набора).

```
SELECT MFR_ID, PRODUCT_ID, DESCRIPTION, (QTY_ON_HAND * PRICE)
  FROM PRODUCTS;

MFR_ID  PRODUCT_ID  DESCRIPTION      (QTY_ON_HAND*PRICE)
------- ----------- --------------- --------------------
REI      2A45C      Ratchet Link              $16,590.00
ACI      4100Y      Widget Remover            $68,750.00
QSA      XK47       Reducer                   $13,490.00
```

```
BIC      41672      Plate                          $0.00
IMM      779C       900-lb Brace              $16,875.00
ACI      41003      Size 3 Widget             $22,149.00
ACI      41004      Size 4 Widget             $16,263.00
BIC      41003      Handle                     $1,956.00
```

Что получится, если увеличить плановый объем продаж для каждого служащего на 3% от его фактического объема продаж?

```
SELECT NAME, QUOTA, (QUOTA + (.03 * SALES))
  FROM SALESREPS;

NAME                 QUOTA  (QUOTA + (.03 * SALES))
-------------   -----------  -----------------------
Bill Adams      $350,000.00              $361,037.33
Mary Jones      $300,000.00              $311,781.75
Sue Smith       $350,000.00              $364,221.50
Sam Clark       $275,000.00              $283,997.36
Bob Smith       $200,000.00              $204,277.82
Dan Roberts     $300,000.00              $309,170.19
Tom Snyder             NULL                     NULL
Larry Fitch     $350,000.00              $360,855.95
Paul Cruz       $275,000.00              $283,603.25
Nancy Angelli   $300,000.00              $305,581.26
```

Как было сказано в главе 5, "Основы SQL", во многих СУБД реализованы дополнительные арифметические операции, операции над строками символов и встроенные функции, которые можно применять в выражениях SQL. Их также можно использовать в выражениях в списке возвращаемых столбцов, как в следующем примере для DB2, в котором из даты извлекается значение месяца и года.

Вывести список имен, а также месяц и год приема на работу всех служащих. (В случае базы данных Oracle вместо функций MONTH и YEAR следует применить функцию TO_CHAR.)

```
SELECT NAME, MONTH(HIRE_DATE), YEAR(HIRE_DATE)
  FROM SALESREPS;
```

Кроме того, в списке возвращаемых столбцов можно использовать константы. Это может пригодиться для создания результатов запроса, которые более удобны для восприятия, как в следующем примере.

Список объемов продаж для каждого города.

```
SELECT CITY, 'has sales of', SALES
  FROM OFFICES;

CITY          HAS SALES OF        SALES
-----------   ------------  -----------
Denver        has sales of  $186,042.00
New York      has sales of  $692,637.00
Chicago       has sales of  $735,042.00
Atlanta       has sales of  $367,911.00
Los Angeles   has sales of  $835,915.00
```

Создается впечатление, что результаты запроса состоят из отдельных предложений, каждое из которых относится к одному из офисов, но на самом деле они представляют собой таблицу, содержащую три столбца. Первый и третий столбцы содержат значения из таблицы OFFICES. Во втором столбце для всех строк содержится одна и та же текстовая строка из двенадцати символов.

Выборка всех столбцов (SELECT *)

Иногда требуется получить содержимое всех столбцов таблицы. На практике такая ситуация может возникнуть, когда вы впервые сталкиваетесь с новой базой данных и необходимо быстро получить представление о ее структуре и хранимых в ней данных. С учетом этого в SQL разрешается использовать вместо списка возвращаемых столбцов символ звездочки (*), который означает, что требуется извлечь все столбцы.

Показать все данные, содержащиеся в таблице OFFICES.

```
SELECT *
  FROM OFFICES;

OFFICE    CITY         REGION     MGR        TARGET         SALES
------    -----------  -------    ---    -----------    -----------
22        Denver       Western    108    $300,000.00    $186,042.00
11        New York     Eastern    106    $575,000.00    $692,637.00
12        Chicago      Eastern    104    $800,000.00    $735,042.00
13        Atlanta      Eastern    105    $350,000.00    $367,911.00
21        Los Angeles  Western    108    $725,000.00    $835,915.00
```

Результаты запроса содержат все шесть столбцов таблицы OFFICES, которые расположены в том же порядке, что и в исходной таблице.

Символ выборки всех столбцов очень удобно использовать в интерактивном SQL. Использования же его в программном SQL следует избегать, поскольку изменения в структуре базы данных могут привести к краху приложения. Предположим, например, что таблица OFFICES была удалена из базы данных, а затем создана вновь, при этом был изменен порядок столбцов и добавлен новый, седьмой, столбец. Если программа ожидает, что запрос SELECT * FROM OFFICES возвратит результат, содержащий шесть столбцов определенных типов, она почти наверняка перестанет работать после изменения порядка столбцов и добавления нового столбца.

Этих сложностей можно избежать, если в программах запрашивать требуемые столбцы по именам. Например, приведенный ниже запрос возвращает те же результаты, что и запрос SELECT * FROM OFFICES. Он не восприимчив к изменениям структуры базы данных, пока в таблице OFFICES существуют столбцы с указанными именами.

```
SELECT OFFICE, CITY, REGION, MGR, TARGET, SALES
  FROM OFFICES;
```

Повторяющиеся строки (DISTINCT)

Если в списке возвращаемых столбцов запроса на выборку указать первичный ключ таблицы, то каждая строка результатов запроса будет уникальной (из-за того, что значения первичного ключа во всех строках разные). Если же первичный ключ не включается в результат запроса, в последнем могут содержаться повторяющиеся строки. Предположим, например, что был выполнен следующий запрос.

Список идентификаторов всех менеджеров офисов.

```
SELECT MGR
  FROM OFFICES;

MGR
---
108
106
104
105
108
```

Таблица результатов запроса содержит пять строк (по одной для каждого офиса), однако две из них совпадают. Почему? Потому что Ларри Фитч (Larry Fitch) является менеджером двух офисов: в Лос-Анджелесе и в Денвере. Поэтому его идентификатор (108) содержится в двух строках таблицы OFFICES. Это не совсем те результаты, которых вы ожидали. Если в фирме работают четыре менеджера, то, вероятно, вы ожидали, что результаты запроса будут содержать четыре строки.

Повторяющиеся строки из таблицы результатов запроса можно удалить, если в инструкции SELECT перед списком выбора указать ключевое слово DISTINCT. Ниже приведен вариант предыдущего запроса, возвращающий те результаты, которые вы ожидали.

Список идентификаторов всех менеджеров офисов.

```
SELECT DISTINCT MGR
  FROM OFFICES;

MGR
---
106
104
105
108
```

Этот запрос выполняется следующим образом. Вначале генерируются все строки результатов (пять строк), а затем удаляются те из них, которые в точности совпадают с другими. Ключевое слово DISTINCT можно указывать независимо от содержимого списка возвращаемых столбцов инструкции SELECT (с некоторыми ограничениями для итоговых запросов, описанными в главе 8).

Если ключевое слово DISTINCT не указано, повторяющиеся строки не удаляются. Можно также использовать ключевое слово ALL, явно указывая, что повторяющиеся строки следует оставить, однако делать это не обязательно — ключевое слово ALL используется по умолчанию.

Отбор строк (WHERE)

SQL-запросы, извлекающие из таблицы все строки, полезны при просмотре базы данных и создании отчетов, однако редко применяются для чего-нибудь еще. Обычно требуется выбрать из таблицы и включить в результаты запроса только несколько строк. Чтобы указать, какие строки нужно отобрать, используется предложение WHERE. Ниже показано несколько запросов, в которых встречается это предложение.

Офисы, в которых фактические объемы продаж превысили плановые.

```
SELECT CITY, SALES, TARGET
  FROM OFFICES
 WHERE SALES > TARGET;

CITY                 SALES        TARGET
-----------     -----------   -----------
New York        $692,637.00   $575,000.00
Atlanta         $367,911.00   $350,000.00
Los Angeles     $835,915.00   $725,000.00
```

Имя, объемы фактических и плановых продаж служащего с идентификатором 105.

```
SELECT NAME, SALES, QUOTA
  FROM SALESREPS
 WHERE EMPL_NUM = 105;

NAME                SALES        QUOTA
----------    -----------  -----------
Bill Adams    $367,911.00  $350,000.00
```

Список всех служащих, менеджером которых является Боб Смит (идентификатор 104).

```
SELECT NAME, SALES
  FROM SALESREPS
 WHERE MANAGER = 104;

NAME                 SALES
-----------     -----------
Bill Adams      $367,911.00
Dan Roberts     $305,673.00
Paul Cruz       $286,775.00
```

Предложение WHERE состоит из ключевого слова WHERE, за которым следует условие отбора, определяющее, какие именно строки требуется извлечь. В предыдущем запросе, например, условием отбора являлось выражение MANAGER =104. На рис. 6.4 изображено, как работает предложение WHERE. Концептуально все строки в таблице SALESREPS просматриваются одна за другой, и к каждой из них применяется условие отбора. Если в условии отбора встречается имя столбца (как, например, имя MANAGER в предыдущем примере), то используется значение этого столбца из текущей строки. Для каждой из строк условие отбора может иметь одно из трех перечисленных ниже значений.

- Если условие отбора имеет значение TRUE, строка будет включена в результаты запроса. Например, значение столбца MANAGER в строке для Билла Адамса (Bill Adams) соответствует условию отбора, поэтому строка для него включается в результаты запроса.
- Если условие отбора имеет значение FALSE, то строка исключается из результатов запроса. Например, значение столбца MANAGER в строке для Сью Смит (Sue Smith) не соответствует условию отбора, поэтому строка для нее исключается из результатов запроса.
- Если условие отбора имеет значение NULL, то строка исключается из результатов запроса. Например, значение столбца MANAGER в строке для Сэма Кларка (Sam Clark) равно NULL, поэтому строка для него исключается из результатов запроса.

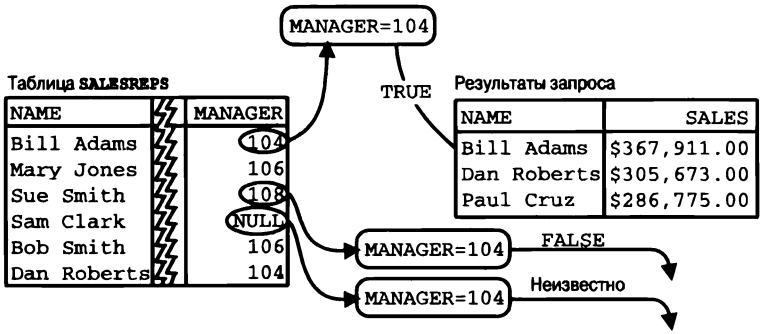


Рис. 6.4. Отбор строк с помощью предложения WHERE

Рис. 6.5 иллюстрирует другой способ рассмотрения роли, выполняемой условием отбора в предложении WHERE. Можно сказать, что условие отбора служит фильтром для строк таблицы. Строки, удовлетворяющие условию отбора, проходят через фильтр и становятся частью результатов запроса. Строки, не удовлетворяющие условию отбора, отфильтровываются и исключаются из результатов запроса.

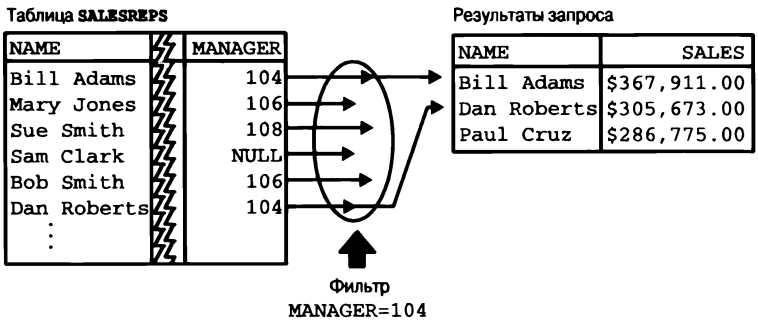


Рис. 6.5. Предложение WHERE в роли фильтра

Условия отбора

В SQL используется множество условий отбора, позволяющих эффективно и естественно создавать различные типы запросов. Ниже рассматриваются пять основных условий отбора (в стандарте ANSI/ISO они называются *предикатами*).

- *Сравнение.* Значение одного выражения сравнивается со значением другого выражения. Например, такое условие отбора используется для выбора всех офисов, находящихся в восточном регионе, или всех служащих, фактические объемы продаж которых превышают плановые.
- *Проверка на принадлежность диапазону.* Проверяется, попадает ли указанное значение в определенный диапазон значений. Например, такое условие отбора используется для нахождения служащих, фактические объемы продаж которых превышают $100000, но меньше $500000.
- *Проверка наличия во множестве.* Проверяется, совпадает ли значение выражения с одним из значений из заданного множества. Например, такое условие отбора используется для выбора офисов, расположенных в Нью-Йорке, Чикаго или Лос-Анджелесе.
- *Проверка на соответствие шаблону.* Проверяется, соответствует ли строковое значение, содержащееся в столбце, определенному шаблону. Например, такое условие отбора используется для выбора клиентов, имена которых начинаются с буквы "E".
- *Проверка на равенство значению NULL.* Проверяется, содержится ли в столбце значение NULL. Например, такое условие отбора используется для нахождения всех служащих, которым еще не был назначен менеджер.

Сравнение (=, <>, <, <=, >, >=)

Наиболее распространенным условием отбора в SQL является *сравнение*. При сравнении SQL вычисляет и сравнивает значения двух SQL-выражений для каждой строки данных. Выражения могут быть как очень простыми, например содержать одно имя столбца или константу, так и более сложными, например содержать арифметические операции. В SQL имеется шесть различных способов сравнения двух выражений, показанных на рис. 6.6.

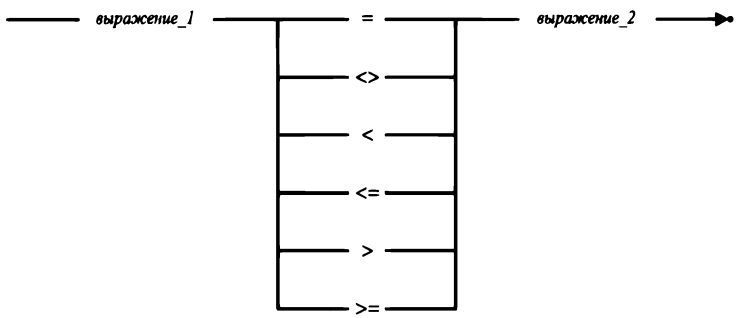


Рис. 6.6. Синтаксическая диаграмма сравнения

Ниже приведены типичные примеры сравнения.

Найти имена всех служащих, принятых на работу до 2006 года.

```
SELECT NAME
  FROM SALESREPS
 WHERE HIRE_DATE < '2006-01-01';

NAME
-----------
Sue Smith
Bob Smith
Dan Roberts
Paul Cruz
```

Заметим, что не все SQL-продукты обрабатывают даты одинаково, поскольку разные производители были вынуждены поддерживать даты еще до того, как был создан стандарт SQL. Формат YYYY-MM-DD, показанный в предыдущем примере, работает в большинстве продуктов, но кое-где его следует изменить. В Oracle, например, вам надо либо заменить формат даты на принятый в Oracle по умолчанию ('01-JAN-88'), либо изменить формат по умолчанию для вашей сессии при помощи следующей команды.

```
ALTER SESSION SET NLS_DATE_FORMAT='YYYY-MM-DD';
```

Вывести список офисов, фактические объемы продаж в которых составили менее 80 процентов от плановых.

```
SELECT CITY, SALES, TARGET
  FROM OFFICES
 WHERE SALES < (.8 * TARGET);

CITY          SALES        TARGET
------  -----------   -----------
Denver  $186,042.00   $300,000.00
```

Вывести список офисов, менеджером которых не является служащий с идентификатором 108.

```
SELECT CITY, MGR
  FROM OFFICES
 WHERE MGR <> 108;

CITY        MGR
--------    ---
New York    106
Chicago     104
Atlanta     105
```

Как показано на рис. 6.6, в соответствии со спецификацией ANSI/ISO проверка на неравенство записывается как A <> B. В ряде реализаций SQL используются альтернативные системы записи, как, например, A != B (поддерживается в SQL Server, DB2, Oracle и MySQL). Иногда такая форма записи является одной из допустимых, а иногда — единственной.

Когда СУБД сравнивает значения двух выражений, могут быть получены три результата:

- если сравнение истинно, то результат проверки имеет значение TRUE;
- если сравнение ложно, то результат проверки имеет значение FALSE;
- если хотя бы одно из двух выражений имеет значение NULL, то результатом сравнения будет NULL.

Выборка одной строки

Чаще всего используется сравнение, в котором определяется, равно ли значение столбца некоторой константе. Если этот столбец представляет собой первичный ключ, то запрос возвращает всего одну строку, как в следующем примере.

Узнать имя и лимит кредита клиента с идентификатором 2107.

```
SELECT COMPANY, CREDIT_LIMIT
  FROM CUSTOMERS
 WHERE CUST_NUM = 2107;

COMPANY              CREDIT_LIMIT
-----------------    ------------
Ace International       $35,000.00
```

Этот тип запросов лежит в основе выборки из баз данных на основе форм веб-страниц. Пользователь вводит в форму идентификатор клиента, и программа использует его при создании и выполнении запроса. После этого она отображает извлеченные данные в форме. Обратите внимание на то, что инструкции SQL, предназначенные для выбора конкретного клиента по идентификатору, как в предыдущем примере, и для выбора всех клиентов, удовлетворяющих определенным параметрам (например, с лимитом кредита более $25000), имеют абсолютно одинаковый формат.

Значения NULL

Использование значений NULL в запросах может привести к "очевидным" предположениям, которые истинны только на первый взгляд, но на самом деле таковыми не являются. Например, можно предположить, что каждая строка из таблицы SALESREPS будет содержаться в результатах только одного из двух следующих запросов.

Вывести список служащих, превысивших плановый объем продаж.

```
SELECT NAME
  FROM SALESREPS
 WHERE SALES > QUOTA;

NAME
-----------
Bill Adams
Mary Jones
Sue Smith
Sam Clark
Dan Roberts
```

```
Larry Fitch
Paul Cruz
```

Вывести список служащих, не выполнивших план.

```
SELECT NAME
  FROM SALESREPS
 WHERE SALES < QUOTA;

NAME
-------------
Bob Smith
Nancy Angelli
```

Однако результаты этих запросов состоят из семи и двух строк соответственно, что дает в сумме девять строк, в то время как в таблице находится десять строк. Строка для Тома Снайдера (Tom Snyder) содержит значение NULL в столбце QUOTA, поскольку ему еще не был назначен плановый объем продаж. Эта строка не вошла ни в один запрос.

Как показывает приведенный пример, при определении условия отбора необходимо помнить об обработке значений NULL. В трехзначной логике, принятой в SQL, условие отбора может иметь значения TRUE, FALSE или NULL. А в результаты запроса попадают только те строки, для которых условие отбора равно TRUE. Мы еще встретимся с NULL позже в этой главе.

Проверка на принадлежность диапазону (BETWEEN)

Следующей формой условия отбора является проверка на принадлежность диапазону значений (оператор BETWEEN...AND), схематически изображенная на рис. 6.7. При этом проверяется, находится ли элемент данных между двумя заданными значениями. В условие отбора входят три выражения. Первое выражение определяет проверяемое значение; второе и третье выражения определяют нижнюю и верхнюю границы проверяемого диапазона. Типы данных трех выражений должны быть сравнимыми.

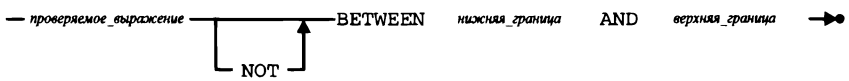


Рис. 6.7. Синтаксическая диаграмма проверки на принадлежность диапазону (BETWEEN)

Следующий пример иллюстрирует типичную процедуру проверки на принадлежность диапазону.

Найти все заказы, сделанные в последнем квартале 2007 года.

```
SELECT ORDER_NUM, ORDER_DATE, MFR, PRODUCT, AMOUNT
  FROM ORDERS
 WHERE ORDER_DATE BETWEEN '2007-10-01' AND '2007-12-31';

ORDER_NUM  ORDER_DATE  MFR  PRODUCT       AMOUNT
---------  ----------  ---  -------  -----------
   112961  2007-12-17  REI  2A44L     $31,500.00
   112968  2007-10-12  ACI  41004      $3,978.00
```

```
   112963  2007-12-17  ACI  41004      $3,276.00
   112983  2007-12-27  ACI  41004        $702.00
   112979  2007-10-12  ACI  4100Z     $15,000.00
   112992  2007-11-01  ACI  41002        $760.00
   112975  2007-10-12  REI  2A44G      $2,100.00
   112987  2007-12-31  ACI  4100Y     $27,500.00
```

При проверке на принадлежность диапазону верхняя и нижняя границы считаются частью диапазона, поэтому в результаты запроса вошли заказы, сделанные 1 октября и 31 декабря. Далее приведен другой пример проверки на принадлежность диапазону.

Найти заказы, стоимости которых попадают в различные диапазоны.

```
SELECT ORDER_NUM, AMOUNT
  FROM ORDERS
 WHERE AMOUNT BETWEEN 20000.00 AND 29999.99;

ORDER_NUM            AMOUNT
---------        ----------
   113036        $22,500.00
   112987        $27,500.00
   113042        $22,500.00

SELECT ORDER_NUM, AMOUNT
  FROM ORDERS
 WHERE AMOUNT BETWEEN 30000.00 AND 39999.99;

ORDER_NUM            AMOUNT
---------        ----------
   112961        $31,500.00
   113069        $31,350.00

SELECT ORDER_NUM, AMOUNT
  FROM ORDERS
 WHERE AMOUNT BETWEEN 40000.00 AND 49999.99;

ORDER_NUM            AMOUNT
---------        ----------
   113045        $45,000.00
```

Инвертированная версия проверки на принадлежность диапазону (NOT BETWEEN) позволяет выбрать значения, которые лежат за пределами диапазона, как в следующем примере.

Вывести список служащих, фактические объемы продаж которых не попадают в диапазон от 80 до 120 процентов плана.

```
SELECT NAME, SALES, QUOTA
  FROM SALESREPS
 WHERE SALES NOT BETWEEN (.8 * QUOTA) AND (1.2 * QUOTA);

NAME                   SALES              QUOTA
-------------     -----------        -----------
Mary Jones        $392,725.00        $300,000.00
Sue Smith         $474,050.00        $350,000.00
```

```
Bob Smith        $142,594.00        $200,000.00
Nancy Angelli    $186,042.00        $300,000.00
```

Проверяемое выражение, задаваемое в операторе BETWEEN, может быть любым допустимым выражением SQL, однако на практике оно обычно представляет собой имя столбца.

В стандарте ANSI/ISO определены относительно сложные правила обработки значений NULL в проверке BETWEEN.

- Если проверяемое выражение имеет значение NULL либо оба выражения, определяющие диапазон, равны NULL, то проверка BETWEEN возвращает NULL.
- Если выражение, определяющее нижнюю границу диапазона, имеет значение NULL, то проверка BETWEEN возвращает FALSE, когда проверяемое значение больше верхней границы диапазона, и NULL — в противном случае.
- Если выражение, определяющее верхнюю границу диапазона, имеет значение NULL, то проверка BETWEEN возвращает FALSE, когда проверяемое значение меньше нижней границы диапазона, и NULL — в противном случае.

Однако прежде чем полагаться на эти правила, неплохо было бы поэкспериментировать со своей СУБД.

Необходимо отметить, что проверка на принадлежность диапазону не расширяет возможности SQL, поскольку ее можно выразить в виде двух сравнений. Проверка

```
A BETWEEN B AND C
```

полностью эквивалентна сравнению

```
(A >= B) AND (A <= C)
```

Тем не менее проверка BETWEEN является более простым способом выразить условие отбора в терминах диапазона значений.

Проверка наличия во множестве (IN)

Еще одним распространенным условием отбора является проверка на наличие во множестве (IN), схематически изображенная на рис. 6.8. В этом случае выполняется проверка, соответствует ли значение какому-либо элементу заданного списка. Ниже приведен ряд запросов с использованием проверки наличия во множестве.

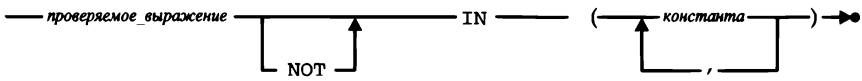


Рис. 6.8. Синтаксическая диаграмма проверки наличия во множестве (IN)

Вывести список служащих, которые работают в Нью-Йорке, Атланте или Денвере.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS
 WHERE REP_OFFICE IN (11, 13, 22);
```

```
NAME                 QUOTA              SALES
-------------    -----------        -----------
Bill Adams       $350,000.00        $367,911.00
Mary Jones       $300,000.00        $392,725.00
Sam Clark        $275,000.00        $299,912.00
Nancy Angelli    $300,000.00        $186,042.00
```

Найти все заказы, сделанные в пятницы в январе 2008 года.

```
SELECT ORDER_NUM, ORDER_DATE, AMOUNT
  FROM ORDERS
 WHERE ORDER_DATE IN ('2008-01-04', '2008-01-11',
                      '2008-01-18', '2008-01-25');

 ORDER_NUM  ORDER_DATE     AMOUNT
---------- ----------- ----------
    113012  2008-01-11  $3,745.00
    113003  2008-02-25  $5,625.00
```

Найти все заказы, полученные четырьмя конкретными служащими.

```
SELECT ORDER_NUM, REP, AMOUNT
  FROM ORDERS
 WHERE REP IN (107, 109, 101, 103);

 ORDER_NUM  REP      AMOUNT
---------- ---- -----------
    112968  101   $3,978.00
    113058  109   $1,480.00
    112997  107     $652.00
    113062  107   $2,430.00
    113069  107  $31,350.00
    112975  103   $2,100.00
    113055  101     $150.00
    113003  109   $5,625.00
    113057  103     $600.00
    113042  101  $22,500.00
```

С помощью проверки `NOT IN` можно проверить, что элемент данных *не* является членом заданного множества. Проверяемое выражение в операторе `IN` может быть любым допустимым SQL-выражением, однако обычно оно представляет собой короткое имя столбца, как в предыдущих примерах. Если результатом проверяемого выражения является значение `NULL`, то проверка `IN` также возвращает `NULL`. Все элементы в списке заданных значений должны иметь один и тот же тип данных, который должен быть сравним с типом данных проверяемого выражения.

Как и проверка `BETWEEN`, проверка `IN` не добавляет в возможности SQL ничего нового, поскольку условие

```
X IN (A, B, C)
```

полностью эквивалентно условию

```
(X = A) OR (X = B) OR (X = C)
```

Однако проверка `IN` предлагает гораздо более эффективный способ выражения условия отбора, особенно если множество содержит большое число элементов.

В стандарте ANSI/ISO не определено максимальное количество элементов множества, и в большинстве СУБД явный верхний предел не задан. По соображениям переносимости, лучше избегать множеств, содержащих один элемент.

```
CITY IN ('New York')
```

Их следует заменять простым сравнением:

```
CITY = 'New York'
```

Проверка на соответствие шаблону (LIKE)

Для выборки строк, в которых содержимое некоторого текстового столбца совпадает с заданным текстом, можно использовать простое сравнение. Например, следующий запрос извлекает строку из таблицы CUSTOMERS по имени.

Показать лимит кредита для Smithson Corp.

```
SELECT COMPANY, CREDIT_LIMIT
  FROM CUSTOMERS
 WHERE COMPANY = 'Smithson Corp.';
```

Однако очень легко можно забыть, какое именно название носит интересующая нас компания: "Smith", "Smithson" или "Smithsonian". Проверка на соответствие шаблону позволяет выбрать из базы данных строки на основе частичного соответствия имени клиента.

Проверка на соответствие шаблону (оператор LIKE), схематически изображенная на рис. 6.9, позволяет определить, соответствует ли значение данных в столбце некоторому *шаблону*. Шаблон представляет собой строку, в которую может входить один или несколько *подстановочных символов*. Эти символы интерпретируются особым образом.

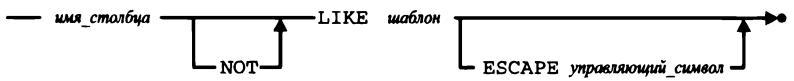


Рис. 6.9. Синтаксическая диаграмма проверки на соответствие шаблону (LIKE)

Подстановочные знаки

Подстановочный знак % совпадает с любой последовательностью из нуля или более символов. Ниже приведена измененная версия предыдущего запроса, в которой используется шаблон, содержащий знак процента.

```
SELECT COMPANY, CREDIT_LIMIT
  FROM CUSTOMERS
 WHERE COMPANY LIKE 'Smith% Corp.';
```

Оператор LIKE указывает SQL, что необходимо сравнивать содержимое столбца NAME с шаблоном "Smith% Corp.". Этому шаблону соответствуют все перечисленные ниже имена.

```
Smith Corp.        Smithson Corp.
Smithsen Corp.     Smithsonian Corp.
```

А вот эти имена данному шаблону не соответствуют.

```
SmithCorp        Smithson Inc.
```

Подстановочный знак _ (символ подчеркивания) совпадает с любым отдельным символом. Например, если вы уверены, что название компании либо "Smithson", либо "Smithsen", то можете воспользоваться следующим запросом.

```
SELECT COMPANY, CREDIT_LIMIT
  FROM CUSTOMERS
 WHERE COMPANY LIKE 'Smiths_n Corp.';
```

В таком случае шаблону будет соответствовать любое из представленных ниже имен.

```
Smithson Corp.      Smithsen Corp.      Smithsun Corp.
```

А вот ни одно из следующих ему соответствовать не будет.

```
Smithsoon Corp.        Smithsn Corp.
```

Подстановочные знаки можно помещать в любое место строки шаблона, и в одной строке может содержаться несколько подстановочных знаков. Следующий запрос допускает как написание "Smithson" и "Smithsen", так и любое другое окончание названия компании, включая "Corp.", "Inc." или какое-то другое.

```
SELECT COMPANY, CREDIT_LIMIT
  FROM CUSTOMERS
 WHERE COMPANY LIKE 'Smiths_n %';
```

С помощью формы NOT LIKE можно находить строки, которые *не* соответствуют шаблону. Проверку LIKE можно применять только к столбцам, имеющим строковый тип данных. Если в столбце содержится значение NULL, то результатом проверки LIKE будет NULL.

Вероятно, вы уже встречались с проверкой на соответствие шаблону в операционных системах, имеющих интерфейс командной строки (таких, как Unix). Обычно в этих системах звездочка (*) используется для тех же целей, что и символ процента (%) в SQL, а вопросительный знак (?) соответствует символу подчеркивания (_) в SQL, но в целом возможности работы с шаблонами строк в них такие же.

Управляющие символы*

При проверке строк на соответствие шаблону может оказаться, что подстановочные знаки входят в строку символов в качестве литералов. Например, нельзя проверить, содержится ли знак процента в строке, просто включив его в шаблон, поскольку SQL будет считать этот знак подстановочным. Как правило, это не вызывает серьезных проблем, поскольку подстановочные знаки довольно редко встречаются в именах, названиях товаров и других текстовых данных, которые обычно хранятся в базе данных.

В стандарте ANSI/ISO определен способ проверки наличия в строке литералов, использующихся в качестве подстановочных знаков. Для этого применяются *управляющие символы*. Когда в шаблоне встречается такой символ, то символ, следующий непосредственно за ним, считается не подстановочным знаком, а литера-

лом. Непосредственно за управляющим символом может следовать либо один из двух подстановочных символов, либо сам управляющий символ, поскольку он также приобретает в шаблоне особое значение.

Символ пропуска определяется в виде строки, состоящей из одного символа, и предложения ESCAPE (рис. 6.9). Ниже приведен пример использования знака доллара ($) в качестве управляющего символа.

Найти товары, коды которых начинаются с четырех букв "A%BC".

```
SELECT ORDER_NUM, PRODUCT
  FROM ORDERS
 WHERE PRODUCT LIKE 'A$%BC%' ESCAPE '$';
```

Первый символ процента в шаблоне, следующий за управляющим символом, считается литералом, второй — подстановочным символом.

Управляющие символы — распространенная практика в приложениях проверки на соответствие шаблону; именно поэтому они были включены и в стандарт ANSI/ISO. Однако они не входили в ранние реализации SQL и поэтому не очень распространены. Для обеспечения переносимости приложений следует избегать использования предложения ESCAPE.

Проверка на равенство NULL (IS NULL)

Значения NULL обеспечивают возможность трехзначной логики в условиях отбора. Для любой заданной строки результат применения условия отбора может быть TRUE, FALSE или NULL (в случае, когда в одном из столбцов содержится значение NULL). Иногда необходимо явно проверять значения столбцов на равенство NULL и непосредственно обрабатывать их. Для этого в SQL имеется специальная проверка IS NULL, синтаксическая диаграмма которой изображена на рис. 6.10.

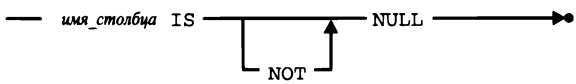


Рис. 6.10. Синтаксическая диаграмма проверки на равенство NULL (IS NULL)

В следующем запросе проверка на равенство NULL используется для нахождения в учебной базе данных служащего, который еще не был закреплен за офисом.

Найти служащего, который еще не закреплен за офисом.

```
SELECT NAME
  FROM SALESREPS
 WHERE REP_OFFICE IS NULL;

NAME
----------
Tom Snyder
```

Инвертированная форма проверки на равенство NULL (IS NOT NULL) позволяет отыскать строки, которые не содержат значений NULL.

Вывести список служащих, которые уже закреплены за офисами.

```
SELECT NAME
  FROM SALESREPS
 WHERE REP_OFFICE IS NOT NULL;

NAME
-------------
Bill Adams
Mary Jones
Sue Smith
Sam Clark
Bob Smith
Dan Roberts
Larry Fitch
Paul Cruz
Nancy Angelli
```

В отличие от условий отбора, описанных выше, проверка на равенство NULL не может возвратить значение NULL в качестве результата. Она всегда возвращает TRUE или FALSE.

Может показаться странным, что нельзя проверить значение на равенство NULL с помощью операции сравнения, например:

```
SELECT NAME
  FROM SALESREPS
 WHERE REP_OFFICE = NULL;
```

Ключевое слово NULL здесь нельзя использовать, поскольку на самом деле это не настоящее значение; это просто свидетельство того, что значение неизвестно. Даже если бы сравнение

```
REP_OFFICE = NULL
```

было возможно, правила обработки значений NULL в сравнениях привели бы к тому, что оно вело бы себя не так, как ожидается. Если бы СУБД обнаружила строку, в которой столбец REP_OFFICE содержит значение NULL, выполнилась бы следующая проверка.

```
NULL = NULL
```

Что будет результатом этого сравнения: TRUE или FALSE? Так как значения по обе стороны знака равенства неизвестны, то, в соответствии с правилами логики SQL, условие отбора должно вернуть значение NULL. Поскольку условие отбора возвращает результат, отличный от TRUE, строка исключается из таблицы результатов запроса — это противоположно тому, к чему вы стремились! Из-за правил обработки значений NULL в SQL необходимо использовать проверку IS NULL.

Составные условия отбора (AND, OR и NOT)

Простые условия отбора, описанные в предыдущих разделах, после применения к некоторой строке возвращают значения TRUE, FALSE или NULL. С помощью правил логики эти простые условия можно объединять в более сложные, как изо-

бражено на рис. 6.11. Обратите внимание на то, что условия отбора, объединяемые с помощью операторов AND, OR и NOT, сами могут быть составными.

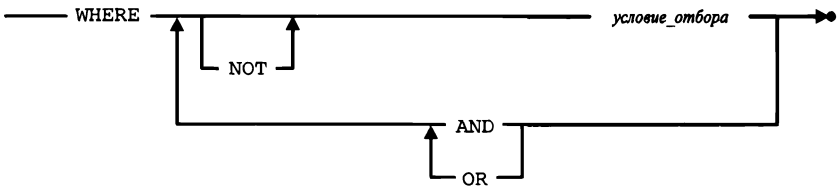


Рис. 6.11. Синтаксическая диаграмма предложения WHERE

Оператор OR используется для объединения двух условий отбора, из которых или одно, или другое (или оба) должно быть истинным.

Найти служащих, у которых фактический объем продаж меньше планового или меньше $300 000.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS
 WHERE SALES < QUOTA
    OR SALES < 300000.00;

NAME                  QUOTA              SALES
-------------    -----------        -----------
Sam Clark        $275,000.00        $299,912.00
Bob Smith        $200,000.00        $142,594.00
Tom Snyder              NULL         $75,985.00
Paul Cruz        $275,000.00        $286,775.00
Nancy Angelli    $300,000.00        $186,042.00
```

Для объединения двух условий отбора, оба из которых должны быть истинными, следует использовать оператор AND.

Найти служащих, у которых фактический объем продаж меньше планового и меньше $300 000.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS
 WHERE SALES < QUOTA
   AND SALES < 300000.00;

NAME                  QUOTA              SALES
-------------    -----------        -----------
Bob Smith        $200,000.00        $142,594.00
Nancy Angelli    $300,000.00        $186,042.00
```

И наконец, можно использовать оператор NOT, чтобы выбрать строки, для которых условие отбора ложно.

Найти служащих, у которых фактический объем продаж меньше планового, но не меньше $150 000.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS
 WHERE SALES < QUOTA
```

```
   AND NOT SALES < 150000.00;

NAME                 QUOTA          SALES
-------------    -----------    -----------
Nancy Angelli    $300,000.00    $186,042.00
```

С помощью логических операторов AND, OR, NOT и круглых скобок можно создавать очень сложные условия отбора, как в следующем примере.

Найти всех служащих, которые: (а) работают в Денвере, Нью-Йорке или Чикаго; или (б) не имеют менеджера и были приняты на работу после июня 2006 года; или (в) у которых продажи превысили плановый объем, но не превысили $600 000.

```
SELECT NAME
  FROM SALESREPS
 WHERE (REP_OFFICE IN (22, 11, 12))
    OR (MANAGER IS NULL AND HIRE_DATE >= '2006-06-01')
    OR (SALES > QUOTA AND NOT SALES > 600000.00);
```

Лично для меня остается загадкой, зачем может понадобиться такой список имен, однако приведенный пример является иллюстрацией довольно сложного запроса.

Как и в случае с простыми условиями отбора, значения NULL влияют на интерпретацию составных условий отбора, вследствие чего результаты последних становятся не столь очевидными. В частности, результатом операции NULL OR TRUE является значение TRUE, а не NULL, как можно было ожидать. Табл. 6.1–6.3 являются таблицами истинности для операторов AND, OR и NOT соответственно в случае тернарной логики (со значениями NULL).

Таблица 6.1. Таблица истинности оператора AND

AND	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Таблица 6.2. Таблица истинности оператора OR

OR	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Таблица 6.3. Таблица истинности оператора NOT

NOT	**TRUE**	**FALSE**	**NULL**
	FALSE	TRUE	NULL

В соответствии со стандартом ANSI/ISO, если с помощью операторов AND, OR и NOT объединяется более двух условий отбора, то оператор NOT имеет наивысший

приоритет, за ним следует AND и только потом OR. Однако чтобы гарантировать переносимость, всегда следует использовать круглые скобки; это позволит устранить все возможные неоднозначности.

В стандарте SQL2 (известном также как SQL-92 и SQL:1992) появилось еще одно логическое условие отбора — проверка IS. На рис. 6.12 изображена синтаксическая диаграмма этой проверки. Оператор IS проверяет значение результата логического выражения.

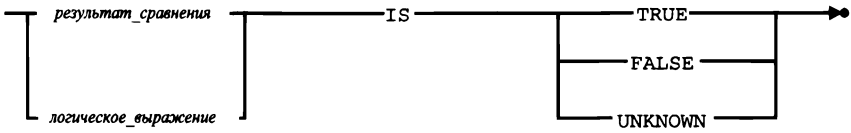


Рис. 6.12. Синтаксическая диаграмма оператора IS

Например, проверку

```
((SALES - QUOTA) > 10000.00) IS UNKNOWN
```

можно использовать, чтобы отыскать строки, в которых нельзя выполнить сравнение из-за того, что либо столбец SALES, либо столбец QUOTA имеет значение NULL. Подобным образом проверка

```
((SALES - QUOTA) > 10000.00) IS FALSE
```

позволяет выбрать строки, в которых значение столбца SALES если и превышает значение столбца QUOTA, то незначительно. Как показывает данный пример, на самом деле проверка IS не привносит в SQL ничего нового, поскольку ее можно легко переписать в следующем виде.

```
NOT ((SALES - QUOTA) > 10000.00)
```

Хотя проверка IS внесена в стандарт SQL с 1992 года, ее поддерживает очень небольшое количество SQL-продуктов. Так что для обеспечения максимальной переносимости следует избегать подобных проверок и записывать выражения только с помощью операторов AND, OR и NOT. Однако избежать проверки IS UNKNOWN удается не всегда.

Сортировка результатов запроса (ORDER BY)

Строки результатов запроса, как и строки таблицы базы данных, не имеют определенного порядка. Но, включив в инструкцию SELECT предложение ORDER BY, можно отсортировать результаты запроса. Это предложение, синтаксическая диаграмма которого изображена на рис. 6.13, содержит список имен или порядковых номеров столбцов, разделенных запятыми. Например, результаты следующего запроса отсортированы по двум столбцам, REGION и CITY.

Показать фактические объемы продаж для каждого офиса, отсортированные в алфавитном порядке по регионам, а в каждом регионе — по городам.

```
SELECT CITY, REGION, SALES
  FROM OFFICES
 ORDER BY REGION, CITY;

CITY          REGION               SALES
-----------   -------        -----------
Atlanta       Eastern        $367,911.00
Chicago       Eastern        $735,042.00
New York      Eastern        $692,637.00
Denver        Western        $186,042.00
Los Angeles   Western        $835,915.00
```

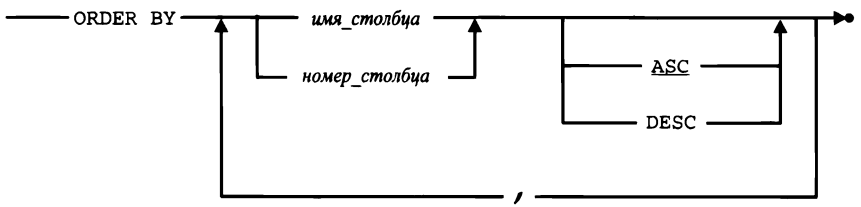


Рис. 6.13. Синтаксическая диаграмма предложения ORDER BY

Первый указанный в предложении ORDER BY столбец (REGION) является *старшим* ключом сортировки; столбцы, следующие за ним (в данном примере CITY), являются *младшими* ключами, что позволяет сортировать строки результатов с одинаковыми старшими ключами. Предложение ORDER BY позволяет выполнять сортировку в возрастающем или убывающем порядке, при этом сортируя результаты запроса по любому элементу списка возвращаемых столбцов.

По умолчанию данные сортируются в порядке возрастания. Чтобы сортировать их по убыванию, следует включить в предложение сортировки ключевое слово DESC, как это сделано в следующем примере.

Вывести список офисов, отсортированный по фактическим объемам продаж в порядке убывания.

```
SELECT CITY, REGION, SALES
  FROM OFFICES
 ORDER BY SALES DESC;

CITY          REGION               SALES
-----------   -------        -----------
Los Angeles   Western        $835,915.00
Chicago       Eastern        $735,042.00
New York      Eastern        $692,637.00
Atlanta       Eastern        $367,911.00
Denver        Western        $186,042.00
```

Как видно из рис. 6.13, чтобы определить порядок сортировки по возрастанию, необходимо использовать ключевое слово ASC, однако из-за того, что этот порядок принят по умолчанию, ключевое слово ASC обычно не указывают.

Если столбец результатов запроса, используемый для сортировки, является вычисляемым, то у него нет имени, которое можно указать в предложении сортировки. В таком случае вместо имени столбца необходимо указать его порядковый номер или повторить выражение в предложении ORDER BY, как в приведенном далее примере. Использование номера столбца — более старый способ, который не рекомендуется к применению, как более подверженный ошибкам (например, при изменении порядка столбцов в предложении SELECT).

Вывести список всех офисов, отсортированный по разности между фактическим и плановым объемами продаж в порядке убывания.

```
SELECT CITY, REGION, (SALES - TARGET)
  FROM OFFICES
 ORDER BY 3 DESC;
```

или

```
SELECT CITY, REGION, (SALES - TARGET)
  FROM OFFICES
 ORDER BY (SALES - TARGET) DESC;

CITY           REGION        (SALES-TARGET)
-----------    -------       --------------
New York       Eastern          $117,637.00
Los Angeles    Western          $110,915.00
Atlanta        Eastern           $17,911.00
Chicago        Eastern          -$64,958.00
Denver         Western         -$113,958.00
```

Полученные результаты запроса отсортированы по третьему столбцу, значения которого представляют собой разности между значениями столбцов SALES и TARGET для каждого офиса. Одновременно используя имена и номера столбцов, а также возрастающий и убывающий порядки сортировки, можно сортировать результаты запроса по достаточно сложному алгоритму, как это сделано в следующем примере.

Вывести список офисов, отсортированный в алфавитном порядке по названиям регионов, а в каждом регионе — по разности между фактическим и плановым объемами продаж в порядке убывания.

```
SELECT CITY, REGION, (SALES-TARGET)
  FROM OFFICES
 ORDER BY REGION ASC, 3 DESC

CITY           REGION        (SALES-TARGET)
-----------    -------       --------------
New York       Eastern          $117,637.00
Atlanta        Eastern           $17,911.00
Chicago        Eastern          -$64,958.00
Los Angeles    Western          $110,915.00
Denver         Western         -$113,958.00
```

Стандарт SQL позволяет управлять порядком сортировки отдельных символов в наборе, что может оказаться очень важным при работе с локализованными набора-

ми символов (например, можно указать порядок сортировки для диакритических символов) или для обеспечения переносимости между системами с таблицами кодировок ASCII и EBCDIC. Однако эта часть стандарта SQL очень сложна, и во многих реализациях SQL либо вообще игнорируются вопросы, связанные с порядком сортировки, либо используются собственные схемы для управления сортировкой.

Правила выполнения однотабличных запросов

Однотабличные запросы в большинстве своем являются простыми, и смысл такого запроса обычно можно легко понять, просто прочитав инструкцию SELECT. Однако по мере возрастания сложности запроса появляется необходимость в более точном "определении" результатов, которые будут возвращены данной инструкцией SELECT. Ниже описана процедура генерации результатов SQL-запроса, включающего в себя предложения, описанные в настоящей главе.

Как видно из описания, результаты запроса, возвращенные инструкцией SELECT, получаются при поочередном применении входящих в инструкцию предложений. Вначале применяется предложение FROM (выбирает таблицу, содержащую требуемые данные), затем — WHERE (по определенному критерию отбирает из таблицы строки), далее — SELECT (создает указанные столбцы результатов запроса и при необходимости удаляет повторяющиеся строки) и, наконец, ORDER BY (сортирует результаты запроса).

Итак, при генерации результатов запроса к одной таблице выполняются следующие действия.

1. Взять таблицу, указанную в предложении FROM.
2. Если имеется предложение WHERE, применить заданное в нем условие отбора к каждой строке таблицы и оставить только те строки, для которых это условие выполняется, т.е. имеет значение TRUE; строки, для которых условие отбора имеет значение FALSE или NULL, — отбросить.
3. Для каждой из оставшихся строк вычислить значение каждого элемента в списке возвращаемых столбцов и создать одну строку таблицы результатов запроса. При каждой ссылке на столбец используется значение столбца для текущей строки.
4. Если указано ключевое слово DISTINCT, удалить из таблицы результатов запроса все повторяющиеся строки.
5. Если имеется предложение ORDER BY, отсортировать результаты запроса.

Строки, сгенерированные описанной процедурой, составляют результаты запроса.

Эти "правила" обработки запросов SQL в следующих трех главах будут неоднократно расширены, чтобы охватить остальные предложения инструкции SELECT.

Объединение результатов нескольких запросов (UNION)*

Иногда появляется необходимость объединения результатов двух или более запросов в одну таблицу. SQL поддерживает такую возможность с помощью операции UNION в инструкции SELECT. Рис. 6.14 иллюстрирует использование операции UNION для выполнения следующего запроса.

Вывести список всех товаров, цена которых превышает $2000 или которых было заказано более чем на $30 000 за один раз.

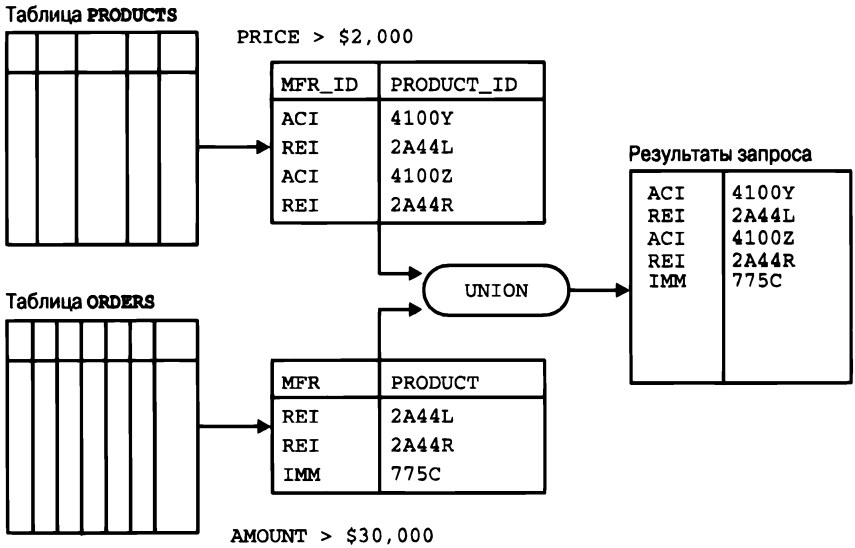


Рис. 6.14. Применение UNION для объединения результатов запроса

Первой части основного запроса удовлетворяет запрос, изображенный в верхней части рисунка.

Вывести список всех товаров, цена которых превышает $2000.

```
SELECT MFR_ID, PRODUCT_ID
  FROM PRODUCTS
 WHERE PRICE > 2000.00;

MFR_ID      PRODUCT_ID
------      ----------
ACI         4100Y
REI         2A44L
ACI         4100Z
REI         2A44R
```

Аналогично вторую часть основного запроса можно выполнить с помощью запроса, изображенного в нижней части рисунка.

Вывести список всех товаров, которых было заказано более чем на $30 000 за один раз.

```
SELECT DISTINCT MFR, PRODUCT
  FROM ORDERS
 WHERE AMOUNT > 30000.00;

MFR     PRODUCT
---     -------
IMM     775C
REI     2A44L
REI     2A44R
```

Как видно из рисунка, операция UNION создает одну таблицу результатов запроса, в которой содержатся строки результатов как верхнего, так и нижнего запросов. В запросе ключевое слово UNION используется следующим образом.

Вывести список всех товаров, цена которых превышает $2000 или которых было заказано более чем на $30 000 за один раз.

```
SELECT MFR_ID, PRODUCT_ID
  FROM PRODUCTS
 WHERE PRICE > 2000.00
 UNION
SELECT DISTINCT MFR, PRODUCT
  FROM ORDERS
 WHERE AMOUNT > 30000.00;

ACI         4100Y
REI         2A44L
ACI         4100Z
REI         2A44R
IMM         775C
```

На таблицы результатов запроса, которые можно объединять с помощью операции UNION, накладываются следующие ограничения:

- эти таблицы должны содержать одинаковое число столбцов;
- тип данных каждого столбца первой таблицы должен совпадать с типом данных соответствующего столбца во второй таблице;
- ни одна из двух таблиц не может быть отсортирована с помощью предложения ORDER BY, однако объединенные результаты запроса можно отсортировать, как описано ниже.

Обратите внимание на то, что имена столбцов в двух запросах, объединенных с помощью операции UNION, не обязательно должны быть одинаковыми. В предыдущем примере в первой таблице результатов запроса имеются столбцы с именами MFR_ID и PRODUCT_ID, в то время как во второй таблице результатов запроса соответствующие им столбцы имеют имена MFR и PRODUCT. Поскольку столбцы в двух таблицах могут иметь различные имена, столбцы результатов запроса, возвращенные операцией UNION, являются безымянными.

Стандарт ANSI/ISO накладывает дополнительные ограничения на инструкции SELECT, участвующие в операции UNION. Он разрешает использовать в списке возвращаемых столбцов только имена столбцов или указатель на все столбцы

(SELECT *) и запрещает использовать выражения. В большинстве коммерческих реализаций SQL это ограничение ослаблено, и в списке возвращаемых столбцов разрешено использовать простые выражения. Однако во многих реализациях SQL в инструкции SELECT не разрешается включать предложения GROUP BY или HAVING, а в некоторых не разрешается использовать в списке возвращаемых столбцов статистические функции (т.е. нельзя использовать суммирующие запросы, описанные в главе 8, "Итоговые запросы"). Более того, некоторые простые реализации SQL не поддерживают саму операцию UNION.

Объединение и повторяющиеся строки*

Поскольку операция UNION объединяет строки из двух результатов запросов, вполне вероятно, что в полученной таблице будут содержаться повторяющиеся строки. Например, в запросе, представленном на рис. 6.14, стоимость товара REI-2A44L составляет $4500.00, поэтому он входит в результаты верхнего запроса. Так как в таблице ORDERS для этого товара имеется заказ на сумму $31500.00, он входит и в результаты нижнего запроса. По умолчанию операция UNION в процессе своего выполнения *удаляет* повторяющиеся строки. Таким образом, в объединенных результатах запроса содержится только *одна* строка для товара REI-2A44L.

Если в таблице результатов операции UNION необходимо сохранить повторяющиеся строки, сразу за ключевым словом UNION следует указать предикат ALL. Приведенный далее запрос вернет таблицу с повторяющимися строками.

Вывести список всех товаров, цена которых превышает $2000 или которых было заказано более чем на $30000 за один раз.

```
SELECT MFR_ID, PRODUCT_ID
  FROM PRODUCTS
 WHERE PRICE > 2000.00
 UNION ALL
SELECT DISTINCT MFR, PRODUCT
  FROM ORDERS
 WHERE AMOUNT > 30000.00;

ACI        4100Y
REI        2A44L
ACI        4100Z
REI        2A44R
IMM        775C
REI        2A44L
REI        2A44R
```

Обратите внимание на то, что обработка по умолчанию повторяющихся строк в операции UNION и инструкции SELECT осуществляется по-разному. Инструкция SELECT по умолчанию оставляет такие строки (SELECT ALL). Чтобы удалить их, необходимо явно указать предикат DISTINCT. Операция UNION по умолчанию удаляет повторяющиеся строки. Чтобы оставить их, следует явно задать предикат ALL.

Специалисты по работе с базами данных критиковали обработку повторяющихся строк в SQL и указывали на эту несогласованность языка как на одну из проблем. Причина такой несогласованности заключается в том, что в SQL в качестве установок по умолчанию выбираются наиболее часто используемые варианты.

- На практике большинство простых инструкций SELECT не возвращает повторяющихся строк, поэтому по умолчанию принято их не удалять.
- На практике большинство операций UNION возвращает повторяющиеся строки, что нежелательно, поэтому по умолчанию такие строки удаляются.

Удаление повторяющихся строк из таблицы результатов запроса занимает много времени, особенно если таблица содержит большое количество строк. Если известно, что операция UNION не возвратит повторяющихся строк, стоит явно указать предикат ALL — тогда запрос будет выполняться быстрее.

Объединение и сортировка*

Предложение ORDER BY нельзя использовать ни в одной из инструкций SELECT, объединенных операцией UNION. Нет смысла выполнять сортировку результатов таких запросов, поскольку пользователь все равно не увидит их в чистом виде. Однако *объединенные* результаты запросов, возвращенные операцией UNION, можно отсортировать с помощью предложения ORDER BY, следующего за второй инструкцией SELECT. Поскольку столбцы таблицы результатов запроса на объединение не имеют имен, в этом предложении следует указывать номера столбцов. Однако многие СУБД, включая Oracle, SQL Server и MySQL, используют имена столбцов из первого запроса SELECT, так что эти имена можно использовать и в предложении ORDER BY.

Ниже показан тот же запрос, что и на рис. 6.14, в котором результаты дополнительно отсортированы по производителю и номеру товара.

Вывести список всех товаров, цена которых превышает $2000 или которых было заказано более чем на $30 000 за один раз; список отсортировать по наименованию производителя и номеру товара.

```
SELECT MFR_ID, PRODUCT_ID
  FROM PRODUCTS
 WHERE PRICE > 2000.00
 UNION
SELECT DISTINCT MFR, PRODUCT
  FROM ORDERS
 WHERE AMOUNT > 30000.00
 ORDER BY 1,2;

ACI        4100Y
ACI        4100Z
IMM        775C
REI        2A44L
REI        2A44R
```

Вложенные объединения*

Операцию UNION можно использовать многократно, чтобы объединить результаты трех или более запросов так, как изображено на рис. 6.15. В результате объединения таблиц B и C получается одна объединенная таблица. Затем с помощью другой операции UNION эта таблица объединяется с таблицей A. Синтаксис запроса, представленного на рисунке, имеет следующий вид.

```
SELECT *
  FROM A
 UNION (SELECT *
          FROM B
         UNION
        SELECT *
          FROM C);

Bill
Mary
George
Fred
Sue
Julia
Harry
```

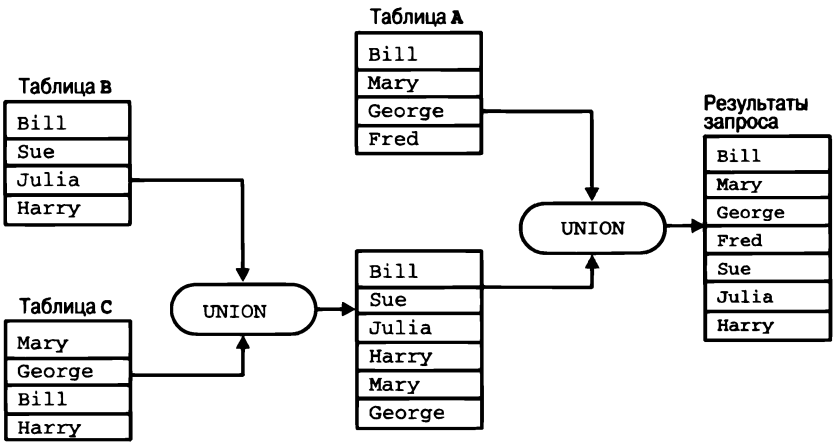


Рис. 6.15. Вложенные операции UNION

Скобки в запросе показывают, какая операция UNION должна выполняться первой. В действительности, если все операции UNION удаляют повторяющиеся строки или все сохраняют их, то порядок выполнения инструкций не имеет значения. Следующие три выражения полностью эквивалентны

```
A UNION (B UNION C)
(A UNION B) UNION C
(A UNION C) UNION B
```

и возвращают семь строк результатов запроса. Аналогично три следующих выражения полностью эквивалентны и возвращают двенадцать строк результатов запроса, поскольку повторяющиеся строки сохраняются.

```
A UNION ALL (B UNION ALL C)
(A UNION ALL B) UNION ALL C
(A UNION ALL C) UNION ALL B
```

Однако если в запросы на объединения входят как операции UNION, так и операции UNION ALL, то порядок этих инструкций имеет значение. Если выражение

```
A UNION ALL B UNION C
```

интерпретировать как

```
A UNION ALL (B UNION C)
```

то оно вернет десять строк (шесть из внутренней инструкции плюс четыре строки из таблицы A). Но если его интерпретировать как

```
(A UNION ALL B) UNION C
```

то оно вернет только четыре строки, поскольку внешняя операция UNION удалит все повторяющиеся строки. По этой причине всегда необходимо использовать круглые скобки, чтобы указать последовательность выполнения в запросах на объединение, содержащих три или более операций UNION.

Резюме

Данная глава является первой из четырех глав, посвященных SQL-запросам. В ней были рассмотрены следующие вопросы.

- Инструкция SELECT используется для формирования SQL-запроса. Каждая инструкция SELECT возвращает таблицу результатов запроса, содержащую один или более столбцов и нуль или более строк.
- Предложение FROM определяет таблицы, в которых содержатся выбираемые данные.
- Предложение SELECT определяет столбцы данных, которые необходимо включить в результаты запроса и которые могут представлять собой столбцы из базы данных или вычисляемые столбцы.
- Предложение WHERE отбирает строки, которые необходимо включить в результаты запроса, путем применения условия отбора к строкам базы данных.
- Условие отбора позволяет выбрать строки путем сравнения значений, проверки значений на принадлежность множеству или диапазону значений, проверки на соответствие шаблону и на равенство значению NULL.
- Простые условия отбора могут объединяться в более сложные условия с помощью операторов AND, OR и NOT.
- Предложение ORDER BY указывает, что результаты запроса должны быть отсортированы по возрастанию или убыванию на основе значений одного или нескольких столбцов.
- Для объединения результатов двух инструкций SELECT в одно множество можно использовать операцию UNION.

ГЛАВА 7

Многотабличные запросы (соединения)

Наиболее полезны запросы, которые выбирают информацию из нескольких таблиц базы данных. Например, приведенные ниже запросы к учебной базе данных извлекают данные из двух, трех или четырех таблиц соответственно.

- Вывести список служащих и офисов, в которых они работают (таблицы SALESREPS и OFFICES).
- Вывести список заказов, сделанных на прошлой неделе, включая следующую информацию: стоимость заказа, имя клиента, сделавшего заказ, и название заказанного товара (таблицы ORDERS, CUSTOMERS и PRODUCTS).
- Показать все заказы, принятые в восточном регионе, в том числе описания товаров и имена служащих, принявших заказы (таблицы ORDERS, SALESREPS, OFFICES и PRODUCTS).

SQL позволяет получить ответы на эти запросы посредством многотабличных запросов, которые *соединяют* данные из нескольких таблиц. В настоящей главе рассматриваются такие запросы и имеющиеся в SQL средства соединения. Мы начнем с возможностей соединения, которые являются частью SQL с самых первых дней. Затем будут рассмотрены дополнительные возможности, впервые появившиеся в стандарте SQL2 и ныне имеющиеся в большинстве распространенных баз данных.

Пример двухтабличного запроса

Чтобы понять, как в SQL реализуются многотабличные запросы, лучше всего начать с рассмотрения простого запроса, который соединяет данные из двух различных таблиц.

Перечислить все заказы, включая номер и стоимость заказа, а также имя и лимит кредита клиента, сделавшего заказ.

Из рис. 7.1 видно, что четыре запрашиваемых элемента данных хранятся в двух различных таблицах.

- В таблице ORDERS содержится номер и стоимость каждого заказа, но в ней отсутствуют имена клиентов и пределы предоставляемых им кредитов.
- В таблице CUSTOMERS содержатся имена клиентов и данные о состоянии их счетов, но в ней нет информации о заказах.

Однако между двумя этими таблицами существует связь. В каждой строке столбца CUST таблицы ORDERS содержится идентификатор клиента, сделавшего заказ, соответствующий значению одной из строк столбца CUST_NUM таблицы CUSTOMERS. Очевидно, чтобы получить требуемые результаты, в инструкции SELECT, с помощью которой осуществляется запрос, необходимо как-то учесть эту связь между таблицами.

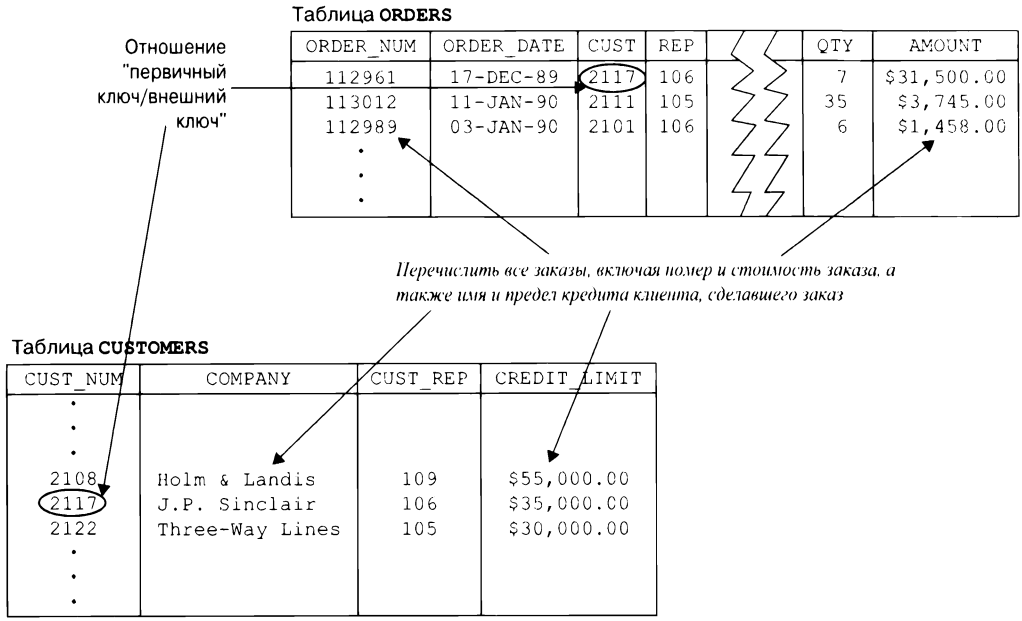


Рис. 7.1. Двухтабличный запрос

Прежде чем рассматривать инструкцию SELECT, выполняющую этот запрос, будет полезно обдумать, как бы вы выполнили этот запрос вручную с помощью карандаша и бумаги. На рис. 7.2 изображены действия, которые, предположительно, вам придется выполнить.

1. Нарисовать таблицу для результатов запроса, содержащую четыре столбца, и записать имена столбцов. Затем перейти к таблице ORDERS и начать с первого заказа.
2. Найти в строке номер первого заказа (112961) и его сумму ($31500.00) и переписать оба значения в первую строку таблицы результатов.

3. В строке первого заказа найти идентификатор клиента, сделавшего заказ (2117). Перейти к таблице CUSTOMERS и в столбце CUST_NUM найти строку с идентификатором клиента 2117.
4. В строке таблицы CUSTOMERS найти имя клиента (J.P. Sinclair) и лимит кредита ($35500.00) и переписать их в таблицу результатов.
5. Вы создали только одну строку результатов запроса! Вернитесь к таблице ORDERS и принимайтесь за следующую строку. Повторяйте процесс со второго заказа и до тех пор, пока не переберете все заказы.

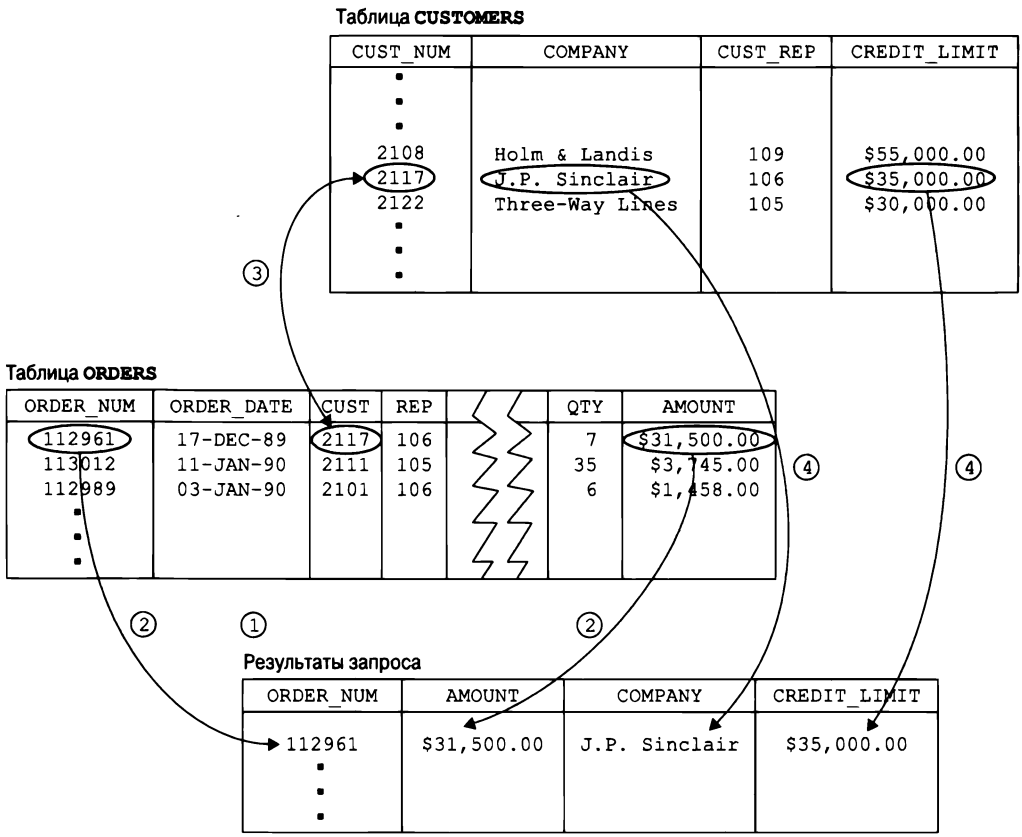


Рис. 7.2. Ручное выполнение многотабличного запроса

Конечно, это не единственный способ получить результаты данного запроса; но как бы вы их ни получали, всегда будет справедливо следующее.

- Каждая строка таблицы результатов запроса формируется из *пары* строк: одна строка находится в таблице ORDERS, а другая — в таблице CUSTOMERS.
- Эта пара строк определяется путем сравнения содержимого *соответствующих друг другу столбцов* этих таблиц.

Простое соединение таблиц

Процесс формирования пар строк путем сравнения содержимого соответствующих столбцов называется *соединением таблиц*. Получаемая в результате таблица, содержащая данные из обеих исходных таблиц, называется *соединением* (join) двух таблиц. (Соединение на основе точного равенства между двумя столбцами более корректно именуется *соединением по равенству* (equi-join). Соединения могут основываться и на других видах сравнения столбцов, как описано далее в этой главе.)

Соединения представляют собой основу многотабличных запросов в SQL. В реляционной базе данных вся информация хранится в виде явных значений данных в столбцах, так что все возможные отношения между таблицами можно сформировать, сопоставляя содержимое связанных столбцов. Таким образом, соединения являются мощным (и, к тому же, *единственным* ввиду отсутствия указателей или иных механизмов связи строк) средством выявления отношений, существующих между данными.

Так как в SQL многотабличные запросы выполняются путем сопоставления столбцов, неудивительно, что инструкция SELECT для многотабличного запроса должна содержать условие отбора, которое определяет взаимосвязь между столбцами. Вот как выглядит инструкция SELECT для запроса, выполненного вручную на рис. 7.2.

Перечислить все заказы, включая номер и стоимость заказа, а также имя и лимит кредита клиента, сделавшего заказ.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, CREDIT_LIMIT
  FROM ORDERS, CUSTOMERS
 WHERE CUST = CUST_NUM;

ORDER_NUM     AMOUNT   COMPANY            CREDIT_LIMIT
--------- ----------   -----------------  ------------
112989     $1,458.00   Jones Mfg.           $65,000.00
112968     $3,978.00   First Corp.          $65,000.00
112963     $3,276.00   Acme Mfg.            $50,000.00
112987    $27,500.00   Acme Mfg.            $50,000.00
112983       $702.00   Acme Mfg.            $50,000.00
113027     $4,104.00   Acme Mfg.            $50,000.00
112993     $1,896.00   Fred Levis Corp.     $65,000.00
113065     $2,130.00   Fred Levis Corp.     $65,000.00
113036    $22,500.00   Ace International    $35,000.00
113034       $632.00   Ace International    $35,000.00
113058     $1,480.00   Holm & Landis        $55,000.00
113055       $150.00   Holm & Landis        $55,000.00
113003     $5,625.00   Holm & Landis        $55,000.00
.
.
.
```

Вспомните, что различные инструменты SQL по-разному форматируют результаты запросов, так что они могут иметь разный внешний вид, в частности, это касается денежных сумм в приведенном примере. Данный запрос очень похож на запросы, рассмотренные в предыдущей главе, однако между ними есть два отличия. Во-первых, предложение FROM содержит две таблицы, а не одну. Во-вторых, в условии отбора

```
CUST = CUST_NUM
```

сравниваются столбцы из двух различных таблиц. Мы будем называть эти столбцы *связанными*, или *согласованными* (matching). Данное условие отбора, как и любое другое, уменьшает число строк в таблице результатов запроса. А поскольку запрос двухтабличный, условие отбора на самом деле уменьшает число *пар* строк в таблице результатов. Фактически в условии отбора заданы те же самые связанные столбцы, которые использовались при выполнении запроса с помощью карандаша и бумаги. Действительно, в этом условии очень хорошо отражена суть сопоставления столбцов, производимого вручную.

Включить в таблицу результатов запроса только те пары строк, у которых идентификатор клиента (CUST) в таблице ORDERS соответствует идентификатору клиента (CUST_NUM) в таблице CUSTOMERS.

Обратите внимание вот на что: в инструкции SELECT ничего не говорится о том, *как* должен выполняться запрос SQL. Там нет никаких упоминаний вроде "начните с заказов" или "начните с клиентов". Вместо этого в запросе сообщается, *что* должны представлять собой результаты запроса, а способ их получения оставлен на усмотрение СУБД.

Запросы с использованием отношения "предок-потомок"

Среди многотабличных запросов наиболее распространены запросы к двум таблицам, связанным с помощью естественного отношения "предок-потомок". Запрос о заказах и клиентах в предыдущем разделе является примером такого запроса. У каждого заказа (потомка) есть соответствующий ему клиент (предок), и каждый клиент (предок) может иметь много своих заказов (потомков). Пары строк, из которых формируются результаты запроса, связаны отношением "предок-потомок".

Из главы 4, "Реляционные базы данных", можно вспомнить, что в SQL-базе данных первичные и внешние ключи создают отношение "предок-потомок". Таблица, содержащая внешний ключ, является потомком, а таблица с первичным ключом — предком. Чтобы использовать в запросе отношение "предок-потомок", необходимо задать условие отбора, в котором первичный ключ сравнивается с внешним ключом. Вот пример такого запроса (рис. 7.3).

Вывести список всех служащих с городами и регионами, в которых они работают.

```
SELECT NAME, CITY, REGION
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE;

NAME            CITY          REGION
-------------   -----------   -------
Mary Jones      New York      Eastern
Sam Clark       New York      Eastern
Bob Smith       Chicago       Eastern
Paul Cruz       Chicago       Eastern
Dan Roberts     Chicago       Eastern
Bill Adams      Atlanta       Eastern
Sue Smith       Los Angeles   Western
Larry Fitch     Los Angeles   Western
Nancy Angelli   Denver        Western
```

Таблица **OFFICES**

OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300,000.00	$186,042.00
11	New York	Eastern	106	$575,000.00	$692,637.00
12	Chicago	Eastern	104	$800,000.00	$735,042.00
13	Atlanta	Eastern	NULL	$350,000.00	$367,911.00
21	Los Angeles	Western	108	$725,000.00	$835,915.00

Таблица **SALESREPS**

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE
105	Bill Adams	37	13	Sales Rep
109	Mary Jones	31	11	Sales Rep
102	Sue Smith	48	21	Sales Rep
106	Sam Clark	52	11	VP Sales
104	Bob Smith	33	12	Sales Mgr
101	Dan Roberts	45	12	Sales Rep
110	Tom Snyder	41	NULL	Sales Rep
108	Larry Fitch	62	21	Sales Mgr
103	Paul Cruz	29	12	Sales Rep
107	Nancy Angelli	49	22	Sales Rep

Результаты запроса

NAME	CITY	REGION

Рис. 7.3. Запрос с использованием отношения "предок-потомок" между таблицами OFFICES **и** SALESREPS

Таблица SALESREPS (потомок) содержит столбец REP_OFFICE, который является внешним ключом для таблицы OFFICES (предок). Здесь отношение "предок-потомок" используется с целью поиска в таблице OFFICE для каждого служащего соответствующей строки, содержащей город и регион, и включения ее в результаты запроса.

Вот еще один запрос к этим же двум таблицам, но здесь роли предка и потомка меняются (рис. 7.4).

Таблица **SALESREPS**

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE
105	Bill Adams	37	13	Sales Rep
109	Mary Jones	31	11	Sales Rep
102	Sue Smith	48	21	Sales Rep
106	Sam Clark	52	11	VP Sales
104	Bob Smith	33	12	Sales Mgr
101	Dan Roberts	45	12	Sales Rep
110	Tom Snyder	41	NULL	Sales Rep
108	Larry Fitch	62	21	Sales Mgr
103	Paul Cruz	29	12	Sales Rep
107	Nancy Angelli	49	22	Sales Rep

Таблица **OFFICES**

OFFICE	CITY	REGION	MGR	TARGET
22	Denver	Western	108	$300,000.00
11	New York	Eastern	106	$575,000.00
12	Chicago	Eastern	104	$800,000.00
13	Atlanta	Eastern	NULL	$350,000.00
21	Los Angeles	Western	108	$725,000.00

Результаты запроса

CITY	NAME	TITLE

Рис. 7.4. Другой запрос с использованием отношения "предок-потомок" между таблицами OFFICES **и** SALESREPS

Вывести список офисов с именами и должностями их руководителей.

```
SELECT CITY, NAME, TITLE
  FROM OFFICES, SALESREPS
 WHERE MGR = EMPL_NUM;

CITY           NAME         TITLE
-----------    -----------  ---------
Chicago        Bob Smith    Sales Mgr
Atlanta        Bill Adams   Sales Rep
New York       Sam Clark    VP Sales
Denver         Larry Fitch  Sales Mgr
Los Angeles    Larry Fitch  Sales Mgr
```

Таблица OFFICES (потомок) содержит столбец MGR, представляющий собой внешний ключ для таблицы SALESREPS (предок). Это отношение используется здесь, чтобы для каждого офиса найти в таблице SALESREPS соответствующую строку, содержащую имя и должность руководителя, и включить ее в результаты запроса.

SQL не требует, чтобы связанные столбцы были включены в результаты многотабличного запроса. На практике они чаще всего и не включаются, как это было в двух предыдущих примерах. Это связано с тем, что первичные и внешние ключи зачастую представляют собой идентификаторы (такие, как идентификатор офиса или идентификатор служащего в приведенных примерах), которые человеку трудно запомнить, тогда как соответствующие названия (города, районы, имена, должности) запомнить гораздо легче. Поэтому вполне естественно, что в предложении WHERE для соединения двух таблиц используются идентификаторы, а в предложении SELECT для создания столбцов результатов запроса — более удобные для восприятия имена.

Еще один способ определения соединений

Простейший способ указать соединяемые таблицы состоит в их перечислении в списке в предложении FROM инструкции SELECT, как показано в предыдущих примерах. Этот метод указания соединяемых таблиц появился в самых ранних реализациях SQL фирмы IBM. Он включен в исходный SQL-стандарт и поддерживается всеми SQL-базами данных.

Последующие версии стандарта существенно расширили возможности соединения и добавили новые варианты предложения FROM. С использованием этих вариантов предыдущие примеры могут быть записаны следующим образом.

Вывести список всех служащих с городами и регионами, в которых они работают.

```
SELECT NAME, CITY, REGION
  FROM SALESREPS JOIN OFFICES
    ON REP_OFFICE = OFFICE;

NAME            CITY          REGION
-------------   -----------   -------
Mary Jones      New York      Eastern
Sam Clark       New York      Eastern
Bob Smith       Chicago       Eastern
Paul Cruz       Chicago       Eastern
Dan Roberts     Chicago       Eastern
Bill Adams      Atlanta       Eastern
```

```
Sue Smith       Los Angeles   Western
Larry Fitch     Los Angeles   Western
Nancy Angelli   Denver        Western
```

Вывести список офисов с именами и должностями их руководителей.

```
SELECT CITY, NAME, TITLE
  FROM OFFICES JOIN SALESREPS
    ON MGR = EMPL_NUM;

CITY          NAME         TITLE
-----------   -----------  ---------
Chicago       Bob Smith    Sales Mgr
Atlanta       Bill Adams   Sales Rep
New York      Sam Clark    VP Sales
Denver        Larry Fitch  Sales Mgr
Los Angeles   Larry Fitch  Sales Mgr
```

Вместо списка с разделяющими элементы запятыми, предложение FROM в приведенных примерах применяет для описания операции соединения ключевое слово JOIN. Связанные столбцы, использующиеся при соединении, указаны в предложении ON, находящемся в конце предложения FROM. В этих простых примерах применение нового синтаксиса дает мало преимуществ по сравнению со старым вариантом инструкции SELECT. Однако, как вы узнаете позже в этой главе, диапазон соединений, выражаемых при помощи нового синтаксиса, весьма широк. Многие ведущие производители СУБД включили поддержку нового синтаксиса JOIN именно по этой причине.

Соединения с условиями отбора строк

В многотабличном запросе можно комбинировать условие отбора, в котором задаются связанные столбцы, с другими условиями отбора, чтобы еще больше сузить результаты запроса. Предположим, что требуется повторить предыдущий запрос, но включить в него только офисы, имеющие превышающие $600000 плановые объемы продаж.

Перечислить офисы, план продаж которых превышает $600 000.

```
SELECT CITY, NAME, TITLE
  FROM OFFICES, SALESREPS
 WHERE MGR = EMPL_NUM
   AND TARGET > 600000.00;

CITY          NAME         TITLE
-----------   -----------  ---------
Chicago       Bob Smith    Sales Mgr
Los Angeles   Larry Fitch  Sales Mgr
```

Вследствие применения дополнительного условия отбора число строк в таблице результатов запроса уменьшилось. Согласно первому условию (MGR=EMPL_NUM), из таблиц OFFICES и SALESREPS отбираются пары строк, которые имеют соответствующее отношение "предок-потомок"; согласно второму условию, производится дальнейший отбор только тех пар строк, где плановый объем продаж превышает $600000. В данном запросе условие соединения и условие отбора находятся в пред-

ложении WHERE. При использовании новейшего синтаксиса условия соединения находятся в предложении ON, а условия отбора — в предложении WHERE, что делает запрос более легким для понимания.

Перечислить офисы, план продаж которых превышает $600 000.

```
SELECT CITY, NAME, TITLE
  FROM OFFICES JOIN SALESREPS
    ON MGR = EMPL_NUM
 WHERE TARGET > 600000.00;

CITY          NAME          TITLE
------------  ------------  ----------
Chicago       Bob Smith     Sales Mgr
Los Angeles   Larry Fitch   Sales Mgr
```

Несколько связанных столбцов

Таблицы ORDERS и PRODUCTS в учебной базе данных связаны парой составных ключей. Столбцы MFR и PRODUCT в таблице ORDERS вместе образуют внешний ключ для таблицы PRODUCTS и связаны с ее столбцами MFR_ID и PRODUCT_ID соответственно. Чтобы соединить таблицы на основе такого отношения "предок-потомок", необходимо задать *обе* пары связанных столбцов, как показано в приведенном далее примере.

Вывести список всех заказов, включая стоимости и описания товаров.

```
SELECT ORDER_NUM, AMOUNT, DESCRIPTION
  FROM ORDERS, PRODUCTS
 WHERE MFR = MFR_ID
   AND PRODUCT = PRODUCT_ID;

ORDER_NUM        AMOUNT     DESCRIPTION
---------      ---------    ---------------
   113027      $4,104.00    Size 2 Widget
   112992        $760.00    Size 2 Widget
   113012      $3,745.00    Size 3 Widget
   112968      $3,978.00    Size 4 Widget
   112963      $3,276.00    Size 4 Widget
   112983        $702.00    Size 4 Widget
   113055        $150.00    Widget Adjuster
   113057        $600.00    Widget Adjuster
      .
      .
      .
```

Условие отбора в данном запросе указывает SQL, что связанными парами строк таблиц ORDERS и PRODUCTS являются те, в которых пары связанных столбцов содержат одни и те же значения. Альтернативная форма запроса указывает связанные столбцы тем же образом.

```
SELECT ORDER_NUM, AMOUNT, DESCRIPTION
  FROM ORDERS JOIN PRODUCTS
    ON MFR = MFR_ID
   AND PRODUCT = PRODUCT_ID;
```

Многостолбчатые соединения обычно встречаются в запросах с составными внешними ключами, как в приведенном примере. SQL не накладывает ограничений на количество столбцов в условиях связи, но обычно соединения отражают реальные взаимоотношения между объектами, представленными в таблицах баз данных, а эти отношения, как правило, включают только один или несколько столбцов таблиц.

Естественные соединения

Зачастую связанные столбцы, используемые при соединении двух таблиц, имеют одно и то же имя в обеих таблицах. В нашей учебной базе, где первичные и внешние ключи имеют несколько отличающиеся имена, что позволяет легко различать их в примерах, это не так. Но на практике создатель базы данных часто использует одно и то же имя для столбца, который содержит идентификатор клиента или номер служащего, во *всех* таблицах, содержащих эти данные.

Предположим, что идентификатор производителя и идентификатор товара имеют имена MFR и PRODUCT в нашей учебной базе данных в *обеих* таблицах — и в ORDERS, и в PRODUCTS. В таком случае большинство естественных соединений между двумя таблицами были бы соединениями по равенству на основе имен столбцов, имеющихся в обеих таблицах. Такое соединение в стандарте SQL так и называется — *естественное соединение*. Синтаксис соединения из SQL-стандарта позволяет легко указать, что вам требуется естественное соединение.

```
SELECT ORDER_NUM, AMOUNT, DESCRIPTION
  FROM ORDERS NATURAL JOIN PRODUCTS;
```

Эта инструкция указывает СУБД на необходимость соединения таблиц ORDERS и PRODUCTS по *всем* столбцам, имеющим в этих таблицах одинаковые имена. В данном примере это столбцы MFR и PRODUCT.

Кроме того, в этой ситуации — применения нового синтаксиса — можно явно указать имена связанных столбцов.

```
SELECT ORDER_NUM, AMOUNT, DESCRIPTION
  FROM ORDERS JOIN PRODUCTS
 USING (MFR, PRODUCT);
```

В предложении USING в скобках перечисляются связанные столбцы (имеющие одинаковые имена в обеих таблицах). Заметим, что предложение USING представляет собой более компактную альтернативу предложению ON. Предыдущий запрос полностью эквивалентен следующему (в предположении одинаковых имен столбцов в обеих таблицах).

```
SELECT ORDER_NUM, AMOUNT, DESCRIPTION
  FROM ORDERS JOIN PRODUCTS
    ON ORDERS.MFR = PRODUCTS.MFR
   AND ORDERS.PRODUCT = PRODUCTS.PRODUCT;
```

Зачастую соединение с помощью предложения USING оказывается предпочтительнее явного указания NATURAL JOIN. Например, если за поддержку таблиц ORDERS и PRODUCTS отвечают два разных администратора (совершенно обычная ситуация для большой базы производственных данных), возможна ситуация, когда они оба случайно выбирают одно и то же имя для вновь добавляемого столбца, хо-

тя эти столбцы не имеют между собой ничего общего. В этой ситуации инструкция с применением NATURAL JOIN попытается использовать новые столбцы для соединения таблиц, что, скорее всего, приведет к ошибке. Предложение USING защищает запрос от неприятностей, связанных с такими изменениями в структуре базы данных. Кроме того, предложение USING позволяет выбрать отдельные столбцы для соединения таблиц, в то время как NATURAL JOIN автоматически использует все столбцы с совпадающими именами. Наконец, если нет столбцов с совпадающими именами, то запрос NATURAL JOIN может вернуть декартово произведение (о нем будет рассказано позже в этой главе) или ошибку — в зависимости от конкретной СУБД; запрос же с применением USING в случае отсутствия столбцов с указанными именами в обеих таблицах всегда вернет ошибку.

Запросы к трем и более таблицам

SQL позволяет соединять данные из трех или более таблиц, используя ту же методику, что и для соединения данных из двух таблиц. Вот простой пример соединения трех таблиц.

Вывести список заказов стоимостью выше $25 000, включающий имя служащего, принявшего заказ, и имя клиента, сделавшего его.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, NAME
  FROM ORDERS, CUSTOMERS, SALESREPS
 WHERE CUST = CUST_NUM
   AND REP = EMPL_NUM
   AND AMOUNT > 25000.00;

ORDER_NUM        AMOUNT  COMPANY          NAME
---------     ----------  ---------------  -----------
   112987     $27,500.00  Acme Mfg.        Bill Adams
   113069     $31,350.00  Chen Associates  Paul Cruz
   113045     $45,000.00  Zetacorp         Larry Fitch
   112961     $31,500.00  J.P. Sinclair    Sam Clark
```

Как видно из рис. 7.5, в этом запросе используются два внешних ключа таблицы ORDERS. Столбец CUST является внешним ключом для таблицы CUSTOMERS; он связывает каждый заказ с клиентом, сделавшим его. Столбец REP является внешним ключом для таблицы SALESREPS, связывая каждый заказ со служащим, принявшим его. Проще говоря, запрос связывает каждый заказ с соответствующим клиентом и служащим.

Альтернативная запись этого запроса более явно указывает каждое соединение и связанные столбцы.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, NAME
  FROM ORDERS JOIN CUSTOMERS ON CUST = CUST_NUM
              JOIN SALESREPS ON REP  = EMPL_NUM
 WHERE AMOUNT > 25000.00;
```

Вот еще один запрос к трем таблицам, в котором используется другая комбинация отношений "предок-потомок".

Вывести список заказов стоимостью выше $25 000, включающий имя клиента, сделавшего заказ, и имя служащего, закрепленного за этим клиентом.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, NAME
  FROM ORDERS, CUSTOMERS, SALESREPS
 WHERE CUST = CUST_NUM
   AND CUST_REP = EMPL_NUM
   AND AMOUNT > 25000.00;

ORDER_NUM          AMOUNT  COMPANY          NAME
---------       ----------  ---------------  -----------
   112987       $27,500.00  Acme Mfg.        Bill Adams
   113069       $31,350.00  Chen Associates  Paul Cruz
   113045       $45,000.00  Zetacorp         Larry Fitch
   112961       $31,500.00  J.P. Sinclair    Sam Clark
```

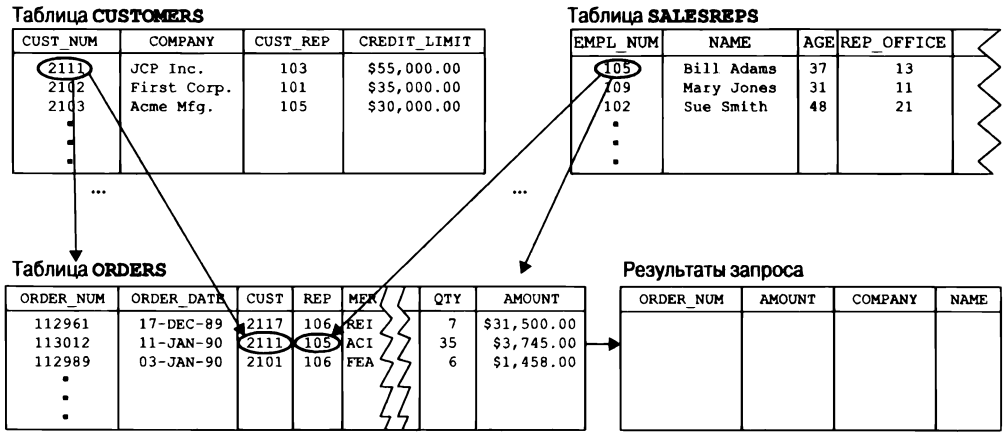


Рис. 7.5. Запрос к трем таблицам

На рис. 7.6 изображены отношения, используемые в приведенном запросе. В первом отношении столбец CUST из таблицы ORDERS используется в качестве внешнего ключа для таблицы CUSTOMERS. Во втором отношении в качестве внешнего ключа для таблицы SALESREPS используется столбец CUST_REP из таблицы CUSTOMERS. Проще говоря, данный запрос связывает каждый заказ с клиентом, а каждого клиента — с закрепленным за ним служащим.

Заметим, что порядок соединений в этих многотабличных запросах значения не имеет. СУБД может соединить таблицу ORDERS с таблицей CUSTOMERS, а затем полученный результат с таблицей SALESREPS. Или же она может сначала соединить таблицу CUSTOMERS с таблицей SALESREPS, а затем результат с таблицей ORDERS. В любом случае окончательный результат будет один и тот же, так что СУБД может выполнять соединения в том порядке, который наиболее эффективен. Однако некоторые сложные соединения, рассматривающиеся позже в этой главе, оказываются чувствительны к порядку отдельных соединений. Одно из преимуществ новейшего синтаксиса соединения в стандарте SQL заключается в том, что в таких случаях он позволяет определить порядок выполнения соединений.

Таблица **SALESREPS**

EMPL_NUM	NAME	AGE	REP_OFFICE
.			
.			
.			
108	Larry Fitch	62	21
103	Paul Cruz	29	12
107	Nancy Angelli	49	22

Таблица **CUSTOMERS**

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2111	JCP Inc.	103	$50,000.00
2102	First Corp.	101	$65,000.00
2103	Acme Mfg.	105	$50,000.00
.			
.			
.			

Таблица **ORDERS**

ORDER_NUM	ORDER_DATE	CUST	REP	QTY	AMOUNT
112961	17-DEC-89	2117	106	7	$31,500.00
113012	11-JAN-90	2111	105	35	$3,745.00
112989	03-JAN-90	2101	106	6	$1,458.00
.					
.					
.					

Результаты запроса

Рис. 7.6. Запрос к трем таблицам с каскадными отношениями "предок-потомок"

В промышленных приложениях нередко встречаются запросы к трем или четырем таблицам, а в больших хранилищах данных интеллектуальные бизнес-запросы запросто включают более десятка таблиц. Даже в рамках маленькой учебной базы данных, состоящей из пяти таблиц, нетрудно создать запрос к четырем таблицам, имеющий реальный смысл.

Вывести список заказов стоимостью выше $25 000, включающий имя клиента, сделавшего заказ, имя закрепленного за ним служащего и офис, в котором работает этот служащий.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, NAME, CITY
  FROM ORDERS, CUSTOMERS, SALESREPS, OFFICES
 WHERE CUST = CUST_NUM
   AND CUST_REP = EMPL_NUM
   AND REP_OFFICE = OFFICE
   AND AMOUNT > 25000.00;

ORDER_NUM      AMOUNT  COMPANY          NAME         CITY
---------  ----------  ---------------  -----------  -----------
   112987  $27,500.00  Acme Mfg.        Bill Adams   Atlanta
   113069  $31,350.00  Chen Associates  Paul Cruz    Chicago
   113045  $45,000.00  Zetacorp         Larry Fitch  Los Angeles
   112961  $31,500.00  J.P. Sinclair    Sam Clark    New York
```

На рис. 7.7 изображены отношения "предок-потомок", используемые в данном запросе. Логически он просто на один шаг расширяет последовательность соеди-

нений из предыдущего запроса, связывая заказ с клиентом, клиента — с закрепленным за ним служащим, а служащего — с его офисом.

Таблица **OFFICES**

OFFICE	CITY	REGION	MGR
22	Denver	Western	108
11	New York	Eastern	106
12	Chicago	Eastern	104
13	Atlanta	Eastern	NULL
21	Los Angeles	Western	108
...			

Таблица **SALESREPS**

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE
•				
•				
•				
108	Larry Fitch	62	21	Sales Mgr
103	Paul Cruz	29	12	Sales Rep
107	Nancy Angelli	49	22	Sales Rep
...				

Таблица **CUSTOMERS**

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2111	JCP Inc.	103	$50,000.00
2102	First Corp.	101	$65,000.00
2103	Acme Mfg.	105	$50,000.00
•			
•			
•			
...			

Таблица **ORDERS**

ORDER_NUM	ORDER_DATE	CUST	AMOUNT
112961	17-DEC-89	2117	$31,500.00
113012	11-JAN-90	2111	$3,745.00
112989	03-JAN-90	2101	$1,458.00
•			
•			
•			

Результаты запроса

Рис. 7.7. Соединение с участием четырех таблиц

Прочие соединения по равенству

Огромное множество многотабличных запросов основано на отношениях "предок-потомок", но в SQL не требуется, чтобы связанные столбцы представляли собой пару "внешний ключ–первичный ключ". Любые два столбца из двух таблиц могут быть связанными, если только они имеют сравнимые типы данных. Вот пример запроса, в котором связанными являются столбцы с датами.

Найти все заказы, полученные в тот день, когда на работу был принят новый служащий.

```
SELECT ORDER_NUM, AMOUNT, ORDER_DATE, NAME
  FROM ORDERS, SALESREPS
 WHERE ORDER_DATE = HIRE_DATE;

 ORDER_NUM      AMOUNT ORDER_DATE  NAME
---------- ----------- ---------- ------------
    112968   $3,978.00 12-OCT-07   Mary Jones
    112979  $15,000.00 12-OCT-07   Mary Jones
    112975   $2,100.00 12-OCT-07   Mary Jones
    112968   $3,978.00 12-OCT-07   Larry Fitch
    112979  $15,000.00 12-OCT-07   Larry Fitch
    112975   $2,100.00 12-OCT-07   Larry Fitch
```

Как показано на рис 7.8, результатами запроса являются пары строк из таблиц ORDERS и SALESREPS, имеющие одинаковые значения в столбцах ORDER_DATE и HIRE_DATE. Эти столбцы не являются ни внешним, ни первичным ключом, да и, вообще, отношение между этими парами строк, надо признать, довольно странное: общее у заказов и служащих только то, что они появились в компании в один день. Тем не менее СУБД с готовностью соединяет таблицы в соответствии с запросом.

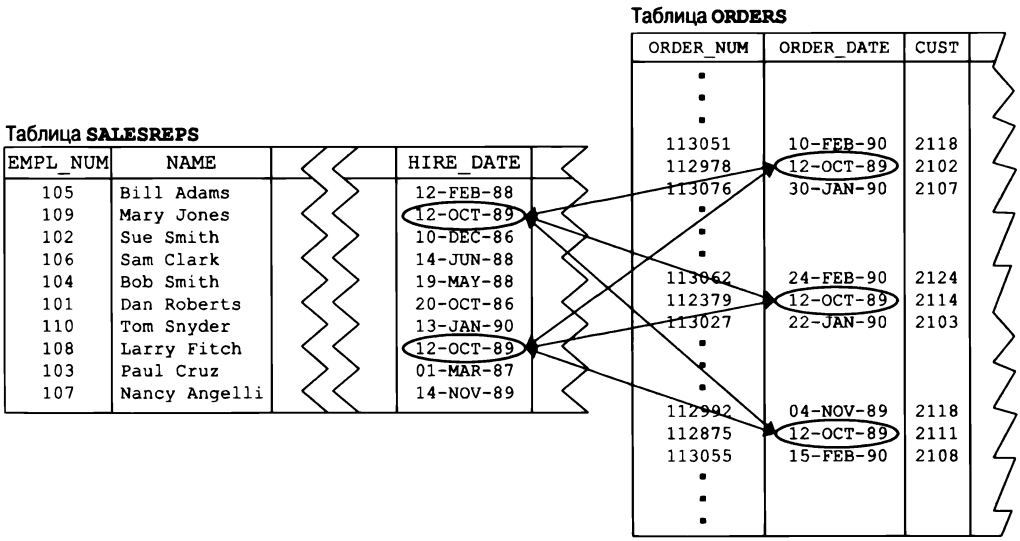


Рис. 7.8. Соединение, в котором не используются первичные и внешние ключи

Связанные столбцы, подобные приведенным в данном примере, создают между двумя таблицами отношение "многие-ко-многим". Может поступить много заказов в день приема на работу какого-нибудь служащего, а также в день получения какого-нибудь заказа на работу может быть принято несколько служащих. В нашем примере 12 октября 2007 года было получено три заказа (112968, 112975 и 112979), и в тот же день на работу было принято двое служащих (Ларри Фитч и Мери Джонс). Три заказа и двое служащих дают шесть строк в таблице результатов запроса.

Отношение "многие-ко-многим" отличается от отношения "один-ко-многим", создаваемого, когда в качестве связанных столбцов используются первичный и внешний ключи. Можно подвести следующие итоги.

- Соединение, созданное путем связи первичного ключа с внешним, всегда создает отношение предок-потомок "один-ко-многим".
- В прочих соединениях также могут существовать отношения "один-ко-многим", если по крайней мере в одной таблице связанный столбец содержит уникальные значения во всех строках.
- В общем случае в соединениях, созданных на основе произвольных связанных столбцов, существуют отношения "многие-ко-многим".

Обратите внимание на то, что отличия трех этих ситуаций не влияют на форму записи инструкции SELECT, выражающей соединение. Соединения всех трех типов записываются одним и тем же способом — включением операции сравнения связанных столбцов в предложение WHERE или ON. Тем не менее полезно рассмотреть соединения с указанной точки зрения, чтобы лучше понимать, как превратить запрос, сформулированный на естественном языке, в правильную инструкцию SELECT.

Соединение по неравенству

Термин "соединение" применяется к любому запросу, который соединяет данные из двух таблиц базы данных путем сравнения значений в двух столбцах этих таблиц. Самыми распространенными являются соединения, созданные на основе равенства связанных столбцов (соединения по равенству). Но SQL позволяет соединять таблицы и с помощью других операций сравнения. В приведенном ниже примере для соединения таблиц используется операция сравнения "больше" (>).

Перечислить все комбинации служащих и офисов, где плановый объем продаж служащего больше, чем план какого-либо офиса, независимо от места работы служащего.

```
SELECT NAME, QUOTA, CITY, TARGET
  FROM SALESREPS, OFFICES
 WHERE QUOTA > TARGET;

NAME                QUOTA  CITY            TARGET
-----------   -----------  ------     -----------
Bill Adams    $350,000.00  Denver     $300,000.00
Sue Smith     $350,000.00  Denver     $300,000.00
Larry Fitch   $350,000.00  Denver     $300,000.00
```

Как и в других запросах к двум таблицам, каждая строка результатов запроса получается из пары строк, в данном случае содержащихся в таблицах SALESREPS и OFFICES. Условие

```
QUOTA > TARGET
```

отбирает те пары строк, в которых значение столбца QUOTA из таблицы SALESREPS превышает значение столбца TARGET из таблицы OFFICES. Обратите внимание:

выбранные из таблиц SALESREPS и OFFICES пары строк связаны *только* этим; в частности, не требуется, чтобы строка таблицы SALESREPS представляла служащего, который работает в офисе, представленном строкой таблицы OFFICES. Следует признать, что данный пример носит несколько искусственный характер и является иллюстрацией того, почему соединения по неравенству практически не встречаются. Тем не менее они могут оказаться полезными в приложениях, предназначенных для поддержки принятия решений, и в других приложениях, исследующих более сложные взаимосвязи в базе данных.

Особенности многотабличных запросов

Многотабличные запросы, рассмотренные до сих пор, не требовали применения специальных синтаксических форм или каких-либо других особенностей языка SQL, помимо тех, что использовались для создания однотабличных запросов. Однако некоторые многотабличные запросы нельзя создать без использования дополнительных возможностей языка SQL.

- Иногда в многотабличных запросах требуется использовать *квалифицированные имена столбцов*, чтобы исключить неоднозначности при ссылках на столбцы.
- В многотабличных запросах *выбор всех столбцов* (SELECT *) имеет особый смысл.
- Для создания многотабличных запросов, связывающих таблицу саму с собой, можно воспользоваться *самосоединениями*.
- В предложении FROM можно воспользоваться *псевдонимами таблиц*, чтобы упростить полные имена столбцов и обеспечить однозначность ссылок на столбцы в самосоединении.

Квалифицированные имена столбцов

В учебной базе данных имеется несколько случаев, когда две таблицы содержат столбцы с одинаковыми именами. Например, столбцы с именем SALES имеются в таблицах OFFICES и SALESREPS. В столбце SALES таблицы OFFICES содержится объем продаж на текущий день для каждого офиса; в аналогичном столбце таблицы SALESREPS содержится текущий объем продаж для каждого служащего. Обычно с этими двумя столбцами затруднений не возникает, поскольку в предложении FROM указано, какая таблица имеется в виду, как в следующих примерах.

Показать названия городов, в которых фактический объем продаж превышает плановый.

```
SELECT CITY, SALES
  FROM OFFICES
 WHERE SALES > TARGET;
```

Показать имена служащих, у которых объемы продаж превышают $350 000.

```
SELECT NAME, SALES
  FROM SALESREPS
 WHERE SALES > 350000.00;
```

Однако рассмотрим запрос, в котором одинаковые имена столбцов представляют проблему.

Показать имя, офис и объем продаж каждого служащего.

```
SELECT NAME, SALES, CITY
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE;

Error: Ambiguous column name "SALES"
```

Хотя в формулировке запроса на естественном языке подразумевается столбец SALES из таблицы SALESREPS, созданный запрос SQL является неоднозначным. Чтобы исключить разночтения, при указании столбцов необходимо использовать их квалифицированные (полные) имена. В главе 5, "Основы SQL", говорилось, что квалифицированное имя столбца содержит имя столбца и имя таблицы, в которой он находится. Так, квалифицированные имена двух столбцов SALES в учебной базе данных будут следующими.

```
OFFICES.SALES и SALESREPS.SALES
```

В инструкции SELECT вместо простых имен столбцов всегда можно использовать полные имена. Таблица, указанная в полном имени столбца, должна, само собой, соответствовать одной из таблиц, указанных в предложении FROM. Вот исправленный вариант предыдущего запроса.

Показать имя, офис и объем продаж каждого служащего.

```
SELECT NAME, SALESREPS.SALES, CITY
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE;

NAME            SALESREPS.SALES  CITY
--------------  ---------------  ------------
Mary Jones          $392,725.00  New York
Sam Clark           $299,912.00  New York
Bob Smith           $142,594.00  Chicago
Paul Cruz           $286,775.00  Chicago
Dan Roberts         $305,673.00  Chicago
Bill Adams          $367,911.00  Atlanta
Sue Smith           $474,050.00  Los Angeles
Larry Fitch         $361,865.00  Los Angeles
Nancy Angelli       $186,042.00  Denver
```

Использование в многотабличных запросах квалифицированных имен столбцов — неплохая мысль. Недостатком такого решения, естественно, является то, что запросы становятся длиннее. В интерактивном режиме можно сначала попробовать выполнить запрос с простыми именами столбцов, чтобы найти все сомнительные столбцы. Если выдается сообщение об ошибке, запрос редактируют и точно указывают столбцы с повторяющимися именами.

Выборка всех столбцов

Как уже говорилось в главе 6, "Простые запросы", инструкция SELECT * может использоваться для выборки всех столбцов таблицы, указанной в предложении FROM. В многотабличном запросе звездочка означает выбор всех столбцов из *всех* таблиц, указанных в предложении FROM. Например, таблица результатов следующего запроса состоит из пятнадцати столбцов (девять столбцов из таблицы SALESREPS, за которыми следуют шесть столбцов из таблицы OFFICES).

Сообщить всю информацию о служащих и офисах, в которых они работают.

```
SELECT *
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE;
```

Очевидно, что инструкция SELECT * становится менее практичной, когда в предложении FROM указаны две, три или большее количество таблиц.

Многие диалекты SQL трактуют звездочку как особый вид универсального имени столбца, которое преобразуется в список всех столбцов. В этих диалектах звездочку можно использовать вместе с именем таблицы вместо списка полных имен столбцов. В следующем запросе имя SALESREPS.* означает список имен всех столбцов таблицы SALESREPS.

Сообщить всю информацию о служащих и офисах, где они работают.

```
SELECT SALESREPS.*, CITY, REGION
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE;
```

Таблица результатов запроса будет состоять из одиннадцати столбцов — девять столбцов таблицы SALESREPS и два столбца таблицы OFFICES, указанных явно. Такой тип "квалифицированных имен всех столбцов" поддерживается многими СУБД. Он введен в стандарте SQL2 и поддерживается всеми основными СУБД.

Самосоединения

Некоторые многотабличные запросы используют отношения, существующие внутри одной таблицы. Предположим, например, что требуется вывести список имен всех служащих и их руководителей. Каждому служащему соответствует одна строка в таблице SALESREPS, а столбец MANAGER содержит идентификатор служащего, являющегося руководителем. Столбцу MANAGER следовало бы быть внешним ключом для таблицы, в которой хранятся данные о руководителях. И фактически он им и является — это внешний ключ для самой таблицы SALESREPS!

Если бы вы попытались создать этот запрос как любой другой запрос к двум таблицам с отношением "первичный ключ–внешний ключ", то он выглядел бы так.

```
SELECT NAME, NAME
  FROM SALESREPS, SALESREPS
 WHERE MANAGER = EMPL_NUM;
```

Эта инструкция SELECT является неправильной из-за двойной ссылки на таблицу SALESREPS в предложении FROM. Вы могли бы попробовать убрать вторую ссылку на таблицу SALESREPS:

```
SELECT NAME, NAME
  FROM SALESREPS
 WHERE MANAGER = EMPL_NUM;
```

Такой запрос будет правильным, но он сделает не то, что вам нужно. Это однотабличный запрос, поэтому СУБД поочередно просматривает все строки таблицы SALESREPS, чтобы найти те, которые удовлетворяют условию отбора.

```
MANAGER = EMPL_NUM
```

Этому условию удовлетворяют строки, в которых оба столбца имеют одинаковые значения, т.е. служащий является своим руководителем. Таких строк нет, поэтому запрос не даст никакого результата — а это совершенно не то, что сказано в постановке задачи.

Чтобы понять, как в SQL решается эта проблема, представим себе, что имеются *две идентичные копии* таблицы SALESREPS. Одна копия называется EMPS и содержит список служащих, а другая называется MGRS и содержит список руководителей (рис. 7.9). Столбец MANAGER таблицы EMPS является внешним ключом для таблицы MGRS, и следующий запрос будет корректно работать.

Таблица **MGRS** (копия **SALESREPS**)

EMPL_NUM	NAME	AGE	REP_OFFICE
105	Bill Adams	37	13
109	Mary Jones	31	11
102	Sue Smith	48	21
106	Sam Clark	52	11
104	Bob Smith	33	12
101	Dan Roberts	45	12
110	Tom Snyder	41	NULL
108	Larry Fitch	62	21
103	Paul Cruz	29	12
107	Nancy Angelli	49	22

Таблица **EMPS** (копия **SALESREPS**)

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER
105	Bill Adams	37	13	Sales Rep	12-JAN-88	104
109	Mary Jones	31	11	Sales Rep	12-OCT-89	106
102	Sue Smith	48	21	Sales Rep	10-DEC-86	108
106	Sam Clark	52	11	VP Sales	14-JUN-88	NULL
104	Bob Smith	33	12	Sales Mgr	19-MAY-87	106
101	Dan Roberts	45	12	Sales Rep	20-OCT-86	104
110	Tom Snyder	41	NULL	Sales Rep	13-JAN-90	101
108	Larry Fitch	62	21	Sales Mgr	12-OCT-89	106
103	Paul Cruz	29	12	Sales Rep	01-MAR-87	104
107	Nancy Angelli	49	22	Sales Rep	14-NOV-88	108

Рис. 7.9. Самосоединение таблицы SALESREPS

Вывести список всех служащих и их руководителей.

```
SELECT EMPS.NAME, MGRS.NAME
  FROM EMPS, MGRS
 WHERE EMPS.MANAGER = MGRS.EMPL_NUM;
```

Так как столбцы в двух таблицах имеют идентичные имена, следует использовать квалифицированные имена столбцов. Во всем остальном этот запрос выглядит как обычный запрос к двум таблицам.

Для соединения таблицы с самой собой в SQL применяется метод "воображаемой копии". Вместо того чтобы на самом деле создавать копию таблицы, СУБД просто позволяет вам сослаться на нее, используя другое имя, *псевдоним таблицы*. Вот тот же запрос, но записанный с использованием псевдонимов EMPS и MGRS для таблицы SALESREPS.

Вывести список всех служащих и их руководителей.

```
SELECT EMPS.NAME, MGRS.NAME
  FROM SALESREPS EMPS, SALESREPS MGRS
 WHERE EMPS.MANAGER = MGRS.EMPL_NUM;

EMPS.NAME        MGRS.NAME
-------------    -----------
Tom Snyder       Dan Roberts
Bill Adams       Bob Smith
Dan Roberts      Bob Smith
Paul Cruz        Bob Smith
Mary Jones       Sam Clark
Bob Smith        Sam Clark
Larry Fitch      Sam Clark
Sue Smith        Larry Fitch
Nancy Angelli    Larry Fitch
```

В предложении FROM для каждой "копии" таблицы SALESREPS назначается псевдоним, который располагается сразу за настоящим именем таблицы. Как видно из примера, если в предложении FROM содержится псевдоним таблицы, то в квалифицированном имени столбца должен использоваться именно псевдоним, а не реальное имя таблицы. Конечно, на самом деле в этом запросе достаточно применить псевдоним только для одной из двух "копий" таблицы. Запрос вполне можно записать и так.

```
SELECT SALESREPS.NAME, MGRS.NAME
  FROM SALESREPS, SALESREPS MGRS
 WHERE SALESREPS.MANAGER = MGRS.EMPL_NUM;
```

Здесь псевдоним MGRS присваивается только одной "копии" таблицы, а для другой используется собственное имя таблицы. Вот еще несколько примеров самосоединения.

Вывести список служащих, планы продаж которых превышают планы их руководителей.

```
SELECT SALESREPS.NAME, SALESREPS.QUOTA, MGRS.QUOTA
  FROM SALESREPS, SALESREPS MGRS
 WHERE SALESREPS.MANAGER = MGRS.EMPL_NUM
   AND SALESREPS.QUOTA > MGRS.QUOTA;
```

```
SALESREPS.NAME      SALESREPS.QUOTA     MGRS.QUOTA
--------------      ---------------     -----------
Bill Adams              $350,000.00     $200,000.00
Dan Roberts             $300,000.00     $200,000.00
Paul Cruz               $275,000.00     $200,000.00
Mary Jones              $300,000.00     $275,000.00
Larry Fitch             $350,000.00     $275,000.00
```

Вывести список служащих, которые работают в различных офисах со своими руководителями, включающий имена и офисы как служащих, так и их руководителей.

```
SELECT EMPS.NAME, EMP_OFFICE.CITY, MGRS.NAME, MGR_OFFICE.CITY
  FROM SALESREPS EMPS, SALESREPS MGRS,
       OFFICES EMP_OFFICE, OFFICES MGR_OFFICE
 WHERE EMPS.REP_OFFICE = EMP_OFFICE.OFFICE
   AND MGRS.REP_OFFICE = MGR_OFFICE.OFFICE
   AND EMPS.MANAGER = MGRS.EMPL_NUM
   AND EMPS.REP_OFFICE <> MGRS.REP_OFFICE;
```

```
EMPS.NAME      EMP_OFFICE.CITY   MGRS.NAME     MGR_OFFICE.CITY
-------------  ---------------   -----------   ---------------
Bob Smith      Chicago           Sam Clark     New York
Bill Adams     Atlanta           Bob Smith     Chicago
Larry Fitch    Los Angeles       Sam Clark     New York
Nancy Angelli  Denver            Larry Fitch   Los Angeles
```

Псевдонимы таблиц

Как уже говорилось в предыдущем разделе, псевдонимы таблиц необходимы в запросах, включающих самосоединения. Однако псевдоним можно использовать в любом запросе — например, если запрос касается таблицы другого пользователя или если имя таблицы очень длинное и употреблять его в квалифицированных именах столбцов утомительно. Приведенный далее запрос ссылается на таблицу BIRTHDAYS, принадлежащую пользователю по имени SAM.

Вывести список имен, плановых объемов продаж и дней рождения служащих.

```
SELECT SALESREPS.NAME, QUOTA, SAM.BIRTHDAYS.BIRTH_DATE
  FROM SALESREPS, BIRTHDAYS
 WHERE SALESREPS.NAME = SAM.BIRTHDAYS.NAME;
```

Если вместо имен двух таблиц использовать псевдонимы S и B, то и вводить, и читать этот запрос будет легче.

Вывести список имен, плановых объемов продаж и дней рождения служащих.

```
SELECT S.NAME, S.QUOTA, B.BIRTH_DATE
  FROM SALESREPS S, SAM.BIRTHDAYS B
 WHERE S.NAME = B.NAME;
```

На рис. 7.10 изображена структура предложения FROM для многотабличной инструкции SELECT, содержащей псевдонимы таблиц. Это предложение выполняет две важные функции.

- В предложении FROM указываются все таблицы, из которых извлекаются данные. Любой столбец, указанный в инструкции SELECT, должен принадлежать одной из таблиц, заданных в предложении FROM. (Имеется исключение для *внешних* ссылок, содержащихся во вложенном запросе. Оно рассматривается в главе 9, "Подзапросы и выражения с запросами".)
- Предложение FROM содержит *тег* таблицы, который используется для указания таблицы в полном имени столбца в инструкции SELECT. Если в предложении FROM указывается псевдоним таблицы, то он становится ее тегом; в противном случае тегом становится имя таблицы в точности в том виде, в каком оно присутствует в предложении FROM.

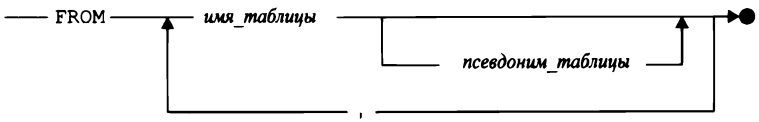


Рис. 7.10. Синтаксическая диаграмма предложения FROM

Единственное требование, предъявляемое к тегам таблиц в предложении FROM, состоит в том, что все они должны отличаться друг от друга. Даже если вы не используете псевдонимы таблиц в ваших SQL-запросах, то все равно встретитесь с ними, если рассмотрите SQL-код, сгенерированный инструментарием наподобие предназначенного для бизнес-анализа или для создания отчетов. Такие инструменты обычно предоставляют графический интерфейс, который позволяет легко выбирать столбцы, таблицы, связанные столбцы, условия отбора и другие элементы запроса, и автоматически генерируют соответствующие инструкции SQL, передаваемые СУБД. Такой инструментарий в предложении FROM сгенерированного SQL-кода почти всегда использует теги таблиц (обычно именуемые как T1, T2, T3 и т.д.), что позволяет легко и однозначно записать оставшуюся часть запроса, независимо от реальных имен таблиц, столбцов и прочих элементов баз данных.

Стандарт SQL допускает необязательную вставку ключевого слова AS между именем и псевдонимом таблицы. В стандарте для того, что мы именовали *псевдонимом таблицы* (table alias), используется термин *связанное имя* (correlation name). Функция и смысл связанного имени те же, что и в точности описанные выше; многие SQL-продукты используют термин *псевдоним* (alias), что лучше описывает суть решаемой псевдонимами таблиц задачи. Стандарт SQL определяет аналогичный метод создания альтернативных имен *столбцов*; в этом случае в стандарте *псевдонимы столбцов* именно так и называются — *псевдонимами* (alias).

Производительность при обработке многотабличных запросов

С увеличением количества таблиц в запросе резко возрастает объем работы, необходимой для выполнения запроса. В самом SQL нет ограничений на число таблиц, соединяемых в одном запросе. Но некоторые маломощные и встраиваемые SQL-продукты ограничивают число таблиц (обычно около восьми таблиц). На практике

высокие затраты на обработку многотабличных запросов во многих приложениях накладывают еще более сильные ограничения на количество таблиц.

В приложениях, предназначенных для оперативной обработки транзакций (OLTP), запрос обычно включает одну или две таблицы. В этих приложениях время ответа является критичной величиной — пользователь, как правило, вводит один или два элемента данных, и ему требуется получить ответ от базы данных в течение одной или двух секунд. Вот некоторые типичные OLTP-запросы для учебной базы данных.

- Пользователь вводит в форму идентификатор клиента, и СУБД выводит на экран лимит кредита, состояние счета и другие данные об этом клиенте (запрос к одной таблице).
- Кассир с помощью специального устройства сканирует с упаковки номер товара, и СУБД выводит на экран наименование и цену товара (запрос к одной таблице).
- Пользователь вводит имя служащего, и программа выдает список текущих заказов, принятых данным служащим (запрос к двум таблицам).

В отличие от OLTP-приложений, в приложениях, предназначенных для поддержки принятия решений, запрос, как правило, обращается ко многим таблицам и использует сложные отношения, существующие в базе данных. В этих приложениях результаты запроса часто нужны для принятия важных решений, поэтому вполне приемлемыми считаются запросы, которые выполняются несколько минут или даже несколько часов. Вот типичные для учебной базы данных запросы, связанные с принятием решений.

- Пользователь вводит название офиса, и программа выдает список двадцати пяти самых больших заказов, принятых служащими этого офиса (запрос к трем таблицам).
- В отчете суммируются продажи каждого служащего по типам товаров и показывается, какой служащий какие товары продал (запрос к трем таблицам).
- Руководитель рассматривает возможность открытия нового офиса в Сиэтле и выполняет запрос для анализа заказов, клиентов, товаров и служащих (запрос к четырем таблицам).

В случае малых таблиц учебной базы данных даже такие запросы (даже на дешевом аппаратном обеспечении) будут требовать всего лишь несколько секунд. Но если таблицы содержат десятки миллионов строк, время выполнения запросов существенно возрастает. Производительность многотабличных соединений может существенно зависеть от структуры индексов и других внутренних структур данных, используемых СУБД для организации хранимых данных. В общем случае запросы с использованием отношений "первичный ключ–внешний ключ" выполняются неплохо, так как обычно СУБД оптимизированы именно для их выполнения.

Внутренняя структура соединения таблиц

Для простых соединений таблиц довольно легко написать правильную инструкцию SELECT на основе запроса, выраженного на обычном языке. И обратно, глядя на инструкцию SELECT, можно легко определить, что она делает. Однако если в соединении участвует много таблиц или используется сложное условие отбора, становится трудно понять, какую функцию выполняет та или иная инструкция SELECT. По этой причине необходимо дать более точное определение понятию "соединение".

Умножение таблиц

Соединение — это частный случай более общей комбинации данных из двух таблиц, известной под названием *декартово произведение* (или просто *произведение*) двух таблиц. Произведение двух таблиц представляет собой таблицу (называемую *таблицей произведения*), состоящую из всех возможных пар строк обеих таблиц. Столбцами таблицы произведения являются все столбцы первой таблицы, за которыми следуют все столбцы второй таблицы. На рис. 7.11. изображен пример произведения двух маленьких таблиц.

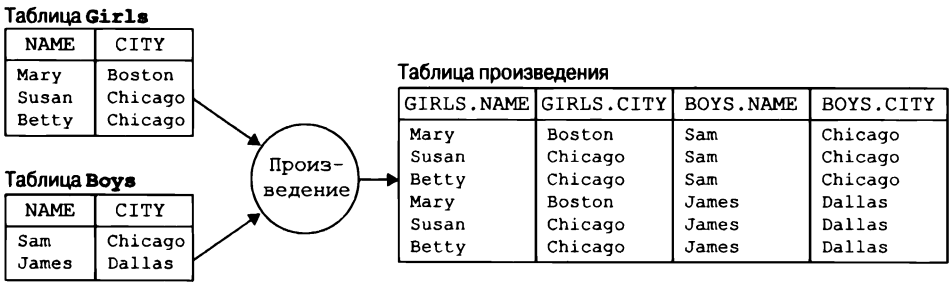


Рис. 7.11. Произведение двух таблиц

Если создать запрос к двум таблицам без предложения WHERE, то таблица результатов запроса окажется произведением двух таблиц. Например, результатом запроса

Показать все возможные комбинации служащих и городов.

```
SELECT NAME, CITY
  FROM SALESREPS, OFFICES;
```

будет произведение таблиц SALESREPS и OFFICES, состоящее из всех возможных комбинаций "служащий-город". Таблица результатов запроса будет иметь 50 строк (5 офисов × 10 служащих = 50 комбинаций). Обратите внимание на то, что для соединения двух упомянутых таблиц используется точно такая же инструкция SELECT, но только с предложением WHERE, содержащим условие сравнения связанных столбцов.

Показать список служащих и городов, в которых они работают.

```
SELECT NAME, CITY
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE;
```

Два приведенных выше запроса указывают на важную связь между соединениями и произведением: соединение двух таблиц является произведением этих таблиц, из которого удалены некоторые строки. Удаляются те строки, которые не удовлетворяют условию, налагаемому на связанные столбцы для данного соединения. Понятие произведения очень важно, так как оно входит в формальное определение правил выполнения многотабличных запросов на выборку, приведенное в следующем разделе.

Правила выполнения многотабличных запросов на выборку

Ниже, после кода запроса, представлены правила выполнения SQL-запроса на выборку, которые ранее рассматривались в главе 6, "Простые запросы", и теперь расширены для случая многотабличных запросов. Эти правила определяют смысл любой многотабличной инструкции SELECT, в точности определяя процедуру, которая всегда позволяет получить корректный набор результатов запроса. Чтобы увидеть эту процедуру в действии, рассмотрим следующий запрос.

Вывести имя компании и все заказы клиента номер 2103.

```
SELECT COMPANY, ORDER_NUM, AMOUNT
  FROM CUSTOMERS JOIN ORDERS
    ON CUST_NUM = CUST
 WHERE CUST_NUM = 2103
 ORDER BY ORDER_NUM;

COMPANY     ORDER_NUM      AMOUNT
---------- ---------- -----------
Acme Mfg.      112963   $3,276.00
Acme Mfg.      112983     $702.00
Acme Mfg.      112987  $27,500.00
Acme Mfg.      113027   $4,104.00
```

Для генерации результатов запроса для инструкции SELECT следует выполнить следующее.

1. Если запрос представляет собой объединение (UNION) инструкций SELECT, для каждой из них выполнить шаги 2–5 для генерации отдельных результатов запросов.
2. Сформировать произведение таблиц, указанных в предложении FROM. Если там указана только одна таблица, то произведением будет она сама.
3. При наличии предложения ON применить указанное в нем условие к каждой строке таблицы произведения, оставляя только те строки, для которых условие истинно, и отбрасывая строки, для которых условие ложно или равно NULL.

4. При наличии предложения WHERE применить указанное в нем условие к каждой строке таблицы произведения, оставляя только те строки, для которых условие истинно, и отбрасывая строки, для которых условие ложно или равно NULL.
5. Для каждой из оставшихся строк вычислить значение каждого элемента в списке возвращаемых столбцов и тем самым создать одну строку таблицы результатов запроса. При любой ссылке на столбец используется значение столбца в текущей строке.
6. Если указан предикат DISTINCT, удалить из таблицы результатов запроса все повторяющиеся строки.
7. Если запрос является объединением инструкций SELECT, объединить результаты выполнения отдельных инструкций в одну таблицу результатов запроса. Удалить из нее повторяющиеся строки, если только в запросе не указан предикат UNION ALL.
8. Если имеется предложение ORDER BY, отсортировать результаты запроса, как указано в нем.

Сгенерированные этой процедурой строки представляют собой результат запроса.

Чтобы увидеть, как работает эта процедура на практике, применим ее к приведенному выше запросу.

1. Предложение FROM генерирует все возможные комбинации строк из таблицы CUSTOMERS (21 строка) и таблицы ORDERS (30 строк), формируя таблицу произведения, состоящую из 630 строк.
2. Условие отбора в предложении ON отбирает только те строки таблицы произведения, у которых одинаковы идентификаторы клиентов (CUST_NUM = = CUST), оставляя из 630 строк только 30 (по одной для каждого заказа).
3. Условие отбора в предложении WHERE отбирает только те строки таблицы произведения, у которых идентификатор клиента равен указанному (CUST_NUM = 2103), оставляя из 30 строк только 4 и удаляя остальные 26.
4. Предложение SELECT отбирает из оставшихся в таблице произведения строк только три столбца (COMPANY, ORDER_NUM и AMOUNT), генерируя четыре строки таблицы результатов запроса.
5. В соответствии с предложением ORDER BY, эти четыре строки сортируются по столбцу ORDER_NUM, и получается окончательный результат.

Очевидно, что ни одна реляционная СУБД не будет выполнять запрос подобным образом, но цель приведенного определения не в том, чтобы описать, как выполняется запрос в СУБД. Оно просто является *определением* того, что означает понятие "многотабличный запрос".

Внешние соединения

Операция соединения в SQL комбинирует информацию из двух таблиц, формируя *пары* связанных строк из этих двух таблиц; в каждую пару входят те строки из различных таблиц, у которых в связанных столбцах содержатся одинаковые значения. Если строка одной из таблиц не имеет пары, то соединение может привести к неожиданным результатам, что иллюстрируют следующие далее запросы.

Вывести список служащих и офисов, где они работают.

```
SELECT NAME, REP_OFFICE
FROM SALESREPS;

NAME           REP_OFFICE
-------------  ----------
Bill Adams             13
Mary Jones             11
Sue Smith              21
Sam Clark              11
Bob Smith              12
Dan Roberts            12
Tom Snyder           NULL
Larry Fitch            21
Paul Cruz              12
Nancy Angelli          22
```

Не забывайте, что не все SQL-инструменты выводят значение NULL так, как показано выше.

Вывести список служащих и городов, где они работают.

```
SELECT NAME, CITY
  FROM SALESREPS JOIN OFFICES
    ON REP_OFFICE = OFFICE;

NAME           CITY
-------------  -----------
Mary Jones     New York
Sam Clark      New York
Bob Smith      Chicago
Paul Cruz      Chicago
Dan Roberts    Chicago
Bill Adams     Atlanta
Sue Smith      Los Angeles
Larry Fitch    Los Angeles
Nancy Angelli  Denver
```

На первый взгляд, эти два запроса должны давать одинаковое количество строк, но результаты первого запроса насчитывают десять строк, а второго — только девять. Почему? Потому что Том Снайдер (Tom Snyder) в настоящий момент еще не получил назначение ни в один офис, и его строка имеет значение NULL в столбце REP_OFFICE (это связанный столбец для данного соединения). Значение NULL не совпадает ни с одним идентификатором офиса в таблице OFFICES, поэтому строка для Тома Снайдера в таблице SALESREPS остается без пары. В ре-

зультате она "исчезает" из соединения, определенного при помощи предложений ON или WHERE. Таким образом, стандартное SQL-соединение потенциально может привести к потере информации, если соединяемые таблицы содержат несвязанные строки.

Опираясь на словесную версию второго запроса, можно предположить, что требуется получить результаты, которые возвращает следующий запрос.

Вывести список служащих и городов, где они работают.

```
SELECT NAME, CITY
  FROM SALESREPS LEFT OUTER JOIN OFFICES
    ON REP_OFFICE = OFFICE;

NAME            CITY
-------------   -----------
Tom Snyder      NULL
Mary Jones      New York
Sam Clark       New York
Bob Smith       Chicago
Paul Cruz       Chicago
Dan Roberts     Chicago
Bill Adams      Atlanta
Sue Smith       Los Angeles
Larry Fitch     Los Angeles
Nancy Angelli   Denver
```

Эти результаты запроса получаются с помощью другого типа операции соединения, именуемого *внешним соединением таблиц* (как указано при помощи дополнительных ключевых слов в предложении FROM). Внешнее соединение является расширением стандартного соединения, описанного ранее в настоящей главе, которое технически называется *внутренним соединением*. В исходном стандарте SQL дано определение только внутреннего соединения; ранние СУБД фирмы IBM также поддерживали только внутреннее соединение. Однако внешнее соединение является понятной, полезной и все более важной частью реляционной модели баз данных и было реализовано во многих SQL-продуктах различных фирм, включая Microsoft, Sybase и Oracle. Внешние соединения включены в стандарт, начиная с SQL2, и в настоящее время поддерживаются во всех основных СУБД, хотя многие малые реализации SQL, такие как предназначенные для встроенных устройств, поддерживают только внутреннее соединение.

Чтобы понять смысл внешнего соединения, будет полезно отвлечься от учебной базы данных и рассмотреть две простые таблицы, изображенные на рис. 7.12. В таблице GIRLS находится список пяти девочек и городов, в которых они живут; в таблице BOYS содержится список пяти мальчиков и их городов. Чтобы найти пары девочек и мальчиков, живущих в одном и том же городе, можно использовать следующий запрос, формирующий внутреннее соединение двух таблиц.

Вывести список девочек и мальчиков, живущих в одних и тех же городах.

```
SELECT *
  FROM GIRLS INNER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;
```

```
GIRLS.NAME     GIRLS.CITY     BOYS.NAME     BOYS.CITY
----------     ----------     ---------     ---------
Mary           Boston         John          Boston
Mary           Boston         Henry         Boston
Susan          Chicago        Sam           Chicago
Betty          Chicago        Sam           Chicago
```

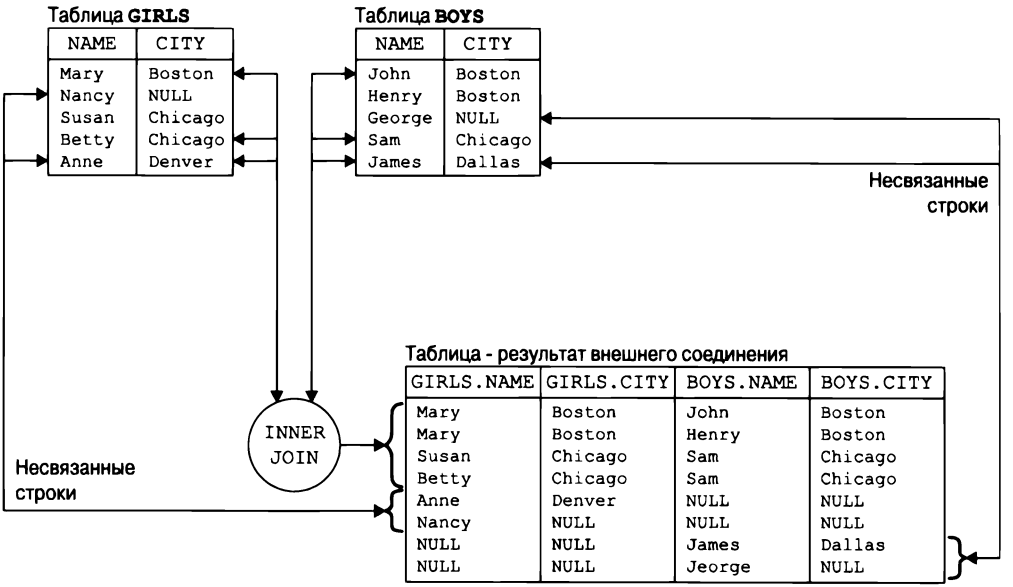


Рис. 7.12. Анатомия внешнего соединения

Этот запрос явно требует внутреннего соединения двух таблиц и дает в результате четыре строки. Внутреннее соединение используется по умолчанию, так что тот же результат будет получен и в том случае, когда используется ключевое слово INNER в предложении FROM. Обратите внимание: две девочки (Anne и Nancy) и два мальчика (James и George) не представлены в таблице результатов запроса. Эти строки не имеют пары в другой таблице и поэтому отсутствуют в таблице результатов внутреннего соединения. Две несвязанные строки (Anne и James) имеют действительные значения в столбцах CITY, но они не совпадают ни с одним городом в противоположной таблице. Две другие несвязанные строки (Nancy и George) имеют в столбцах CITY значение NULL, а по правилам SQL значение NULL не равно *никакому* другому значению (даже другому значению NULL).

Предположим, что вы хотите вывести список пар "девочка/мальчик", живущих в одних и тех же городах, и включить в него девочек и мальчиков без пары. Этот результат дает *полное внешнее соединение* таблиц GIRLS и BOYS. Ниже приведена последовательность построения внешнего соединения, а графически этот процесс показан на рис. 7.12.

1. Начать со внутреннего соединения двух таблиц обычным способом. (Так на рисунке получаются первые четыре строки результирующей таблицы.)

2. Для каждой строки первой таблицы, которая не имеет связи ни с одной строкой второй таблицы, добавить в результаты запроса строку со значениями столбцов из первой таблицы, а вместо значений столбцов второй таблицы использовать значения NULL. (Так получаются пятая и шестая строки результирующей таблицы.)
3. Для каждой строки второй таблицы, которая не имеет связи ни с одной строкой первой таблицы, добавить в результаты запроса строку со значениями столбцов из второй таблицы, а вместо значений столбцов первой таблицы использовать значения NULL. (Так получаются седьмая и восьмая строки результирующей таблицы.)
4. Результирующая таблица является внешним соединением двух таблиц.

Вот инструкция SQL для внешнего соединения таблиц.

Вывести список девочек и мальчиков из одних и тех же городов, включая тех, кто не имеет пары.

```
SELECT *
  FROM GIRLS FULL OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;

GIRLS.NAME    GIRLS.CITY    BOYS.NAME    BOYS.CITY
----------    ----------    ---------    ---------
Mary          Boston        John         Boston
Mary          Boston        Henry        Boston
Susan         Chicago       Sam          Chicago
Betty         Chicago       Sam          Chicago
Anne          Denver        NULL         NULL
Nancy         NULL          NULL         NULL
NULL          NULL          James        Dallas
NULL          NULL          George       NULL
```

Как видно из этого примера, полное внешнее соединение является соединением, "сохраняющим информацию". (Некоторые СУБД, такие как MySQL 5.0, пока не поддерживают полное внешнее соединение.) Каждая строка таблицы BOYS представлена в таблице результатов запроса (некоторые — более одного раза). Аналогично каждая строка таблицы GIRLS представлена в таблице результатов (некоторые, опять же, более одного раза).

Левое и правое внешние соединения

Полное внешнее соединение, полученное в предыдущем запросе, симметрично по отношению к обеим таблицам. Однако существует еще два типа внешних соединений, которые таковыми не являются.

Левое внешнее соединение двух таблиц получается, если выполнить шаги 1 и 2 из предыдущего описания соединения, а шаг 3 пропустить. Таким образом, левое внешнее соединение включает все несвязанные строки первой (левой) таблицы, дополняя их значениями NULL, но не включает несвязанные строки второй (правой) таблицы. Вот как выглядит левое внешнее соединение таблиц GIRLS и BOYS.

Вывести список девочек и мальчиков из одних и тех же городов и девочек, не имеющих пары.

```
SELECT *
  FROM GIRLS LEFT OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;

GIRLS.NAME    GIRLS.CITY    BOYS.NAME    BOYS.CITY
----------    ----------    ---------    ---------
Mary          Boston        John         Boston
Mary          Boston        Henry        Boston
Susan         Chicago       Sam          Chicago
Betty         Chicago       Sam          Chicago
Anne          Denver        NULL         NULL
Nancy         NULL          NULL         NULL
```

Таблица результатов этого запроса содержит шесть строк: все пары "девочка/мальчик" из одних и тех же городов и девочки без пары. Мальчики, не имеющие пары, в этой таблице отсутствуют.

Аналогично *правое внешнее соединение* двух таблиц получается, если выполнить шаги 1 и 3, а шаг 2 пропустить. Таким образом, правое внешнее соединение включает все несвязанные строки второй (правой) таблицы, дополняя их значениями NULL, но не включает несвязанные строки первой (левой) таблицы. Вот правое внешнее соединение таблиц GIRLS и BOYS.

Вывести список девочек и мальчиков из одних и тех же городов и мальчиков, не имеющих пары.

```
SELECT *
  FROM GIRLS RIGHT OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;

GIRLS.NAME    GIRLS.CITY    BOYS.NAME    BOYS.CITY
----------    ----------    ---------    ---------
Mary          Boston        John         Boston
Mary          Boston        Henry        Boston
Susan         Chicago       Sam          Chicago
Betty         Chicago       Sam          Chicago
NULL          NULL          James        Dallas
NULL          NULL          George       NULL
```

Этот запрос также дает таблицу результатов из шести строк: все пары "девочка/мальчик" из одних и тех же городов и мальчики без пары. На этот раз в таблицу не вошли девочки, не имеющие пары.

Как указывалось ранее, только полное внешнее соединение рассматривает обе таблицы одинаково; в случае левого и правого внешних соединений это не так. Зачастую полезно рассматривать одну из таблиц как "старшую", или "ведущую" (все строки которой оказываются в результате запроса), а другую — как "младшую", или "ведомую" (столбцы которой в результате запроса содержат значения NULL). В левом внешнем соединении левая (первая) таблица является старшей, а правая (вторая) — младшей. В случае правого внешнего соединения роли меняются (правая таблица становится старшей, левая — младшей).

На практике левое и правое внешние соединения более полезны, чем полное внешнее соединение, особенно если таблицы связаны через внешний и первичный ключи. В качестве иллюстрации обратимся снова к нашей учебной базе данных. Мы уже приводили пример левого внешнего соединения таблиц SALESREPS и OFFICES. Столбец REP_OFFICE таблицы SALESREPS является внешним ключом таблицы OFFICES; он содержит номера офисов, в которых работают служащие, и допускает наличие значений NULL, если новому служащему еще не был назначен офис. В нашей базе данных такой служащий есть — это Том Снайдер. Любое соединение, которое формируется на основе данных из этих двух таблиц и в которое предполагается включить сведения о Томе Снайдере, *обязано* быть внешним, а таблица SALESREPS должна выступать в роли старшей.

Вывести список служащих и городов, где они работают.

```
SELECT NAME, CITY
  FROM SALESREPS LEFT OUTER JOIN OFFICES
    ON REP_OFFICE = OFFICE;

NAME            CITY
-------------   -----------
Tom Snyder      NULL
Mary Jones      New York
Sam Clark       New York
Bob Smith       Chicago
Paul Cruz       Chicago
Dan Roberts     Chicago
Bill Adams      Atlanta
Sue Smith       Los Angeles
Larry Fitch     Los Angeles
Nancy Angelli   Denver
```

Обратите внимание на то, что в данном случае дочерняя таблица (SALESREPS, содержащая внешний ключ) является старшей, родительская таблица (OFFICES) — младшей. Цель запроса заключается в том, чтобы сохранить в результатах запроса те строки дочерней таблицы, которые содержат значения NULL в столбце внешнего ключа. Не играет роли, используется ли левое соединение (как в примере) или правое с переставленными таблицами.

Вывести список служащих и городов, где они работают.

```
SELECT NAME, CITY
  FROM OFFICES RIGHT OUTER JOIN SALESREPS
    ON OFFICE = REP_OFFICE;

NAME            CITY
-------------   -----------
Tom Snyder      NULL
Mary Jones      New York
Sam Clark       New York
Bob Smith       Chicago
Paul Cruz       Chicago
Dan Roberts     Chicago
Bill Adams      Atlanta
Sue Smith       Los Angeles
```

```
Larry Fitch      Los Angeles
Nancy Angelli    Denver
```

Важно то, что дочерняя таблица является в соединении старшей.

Иногда находят применение запросы, в которых родительская таблица является старшей, а дочерняя — младшей. Предположим, например, что компания открывает новый офис в Далласе, но еще не набрала в него служащих. Если вы генерируете отчет, в котором перечислены все офисы и имена работающих в них служащих, то можете захотеть включить в него строку, представляющую офис в Далласе. Вот внешнее соединение, решающее эту задачу.

Вывести список офисов и служащих, работающих в каждом из них.

```
SELECT CITY, NAME
  FROM OFFICES LEFT OUTER JOIN SALESREPS
    ON OFFICE = REP_OFFICE;

CITY           NAME
-----------    -------------
New York       Mary Jones
New York       Sam Clark
Chicago        Bob Smith
Chicago        Paul Cruz
Chicago        Dan Roberts
Atlanta        Bill Adams
Los Angeles    Sue Smith
Los Angeles    Larry Fitch
Denver         Nancy Angelli
Dallas         NULL
```

В этом случае родительская таблица (OFFICES) является старшей во внешнем соединении, а дочерняя (SALESREPS) — младшей. Цель запроса — гарантировать, что все строки таблицы OFFICES будут представлены в результатах запроса, поэтому данная таблица является старшей. Как видим, по сравнению с предыдущим примером, таблицы SALESREPS и OFFICES поменялись ролями. Строка для Тома Снайдера, которая ранее (когда старшей таблицей была SALESREPS) включалась в результат запроса, теперь в нем отсутствует, так как сейчас таблица SALESREPS является младшей.

Старая запись внешнего соединения *

Понятие внешнего соединения не вошло в изначальный стандарт SQL и не было реализовано в ранних СУБД фирмы IBM, поэтому поставщики тех СУБД, в которых это понятие поддерживалось, изобретали собственные системы записи для выражения внешних соединений. Пользователи этих продуктов разрабатывали программы с применением таких систем записи, привязывая таким образом свои приложения к Oracle или SQL Server. Создатели стандарта SQL2 старались добавить поддержку внешних соединений так, чтобы не "мешать" существующим программам, чтобы они могли сосуществовать с новыми программами, основанными на новом стандарте. Все основные производители в настоящее время полностью или частично поддерживают стандартные внешние соединения и поощряют своих

клиентов пользоваться ими. Тем не менее вам могут встретиться старые программы, в которых применяются старые "фирменные" обозначения для внешних соединений, так что вкратце опишем их здесь.

SQL Server в своих ранних реализациях (во времена Sybase) поддерживал внешние соединения и продолжает поддерживать после приобретения его Microsoft. В SQL Server в предложении WHERE, определяющем условие соединения, к знаку равенства добавляется звездочка (*). Так, полное внешнее соединение таблиц GIRLS и BOYS, в стандартной записи имеющее вид

Вывести список девочек и мальчиков из одних и тех же городов, включая тех, кто не имеет пары.

```
SELECT *
  FROM GIRLS FULL OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;
```

при использовании SQL Server превращается в

```
SELECT *
  FROM GIRLS, BOYS
 WHERE GIRLS.CITY *=* BOYS.CITY;
```

Для указания полного внешнего соединения двух таблиц звездочка помещается и до, и после знака равенства, определяющего соединение. При левом внешнем соединении указывается только ведущая звездочка, что дает запрос

```
SELECT *
  FROM GIRLS, BOYS
 WHERE GIRLS.CITY *= BOYS.CITY;
```

который эквивалентен следующему запросу в стандартном виде.

```
SELECT *
  FROM GIRLS LEFT OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;
```

Аналогично правое внешнее соединение указывается при помощи звездочки после знака равенства:

```
SELECT *
  FROM GIRLS, BOYS
 WHERE GIRLS.CITY =* BOYS.CITY;
```

Звездочка может использоваться в сочетании и с другими операторами сравнения, такими как "больше" или "меньше", для указания внешнего соединения по неравенству. Эта старая запись SQL Server все еще поддерживается в текущих версиях продукта при установке соответствующего уровня совместимости, но начиная с SQL Server 2005 такую запись рекомендуется больше не применять. Еще одно место, где можно встретить такую звездочку, — это хранимые процедуры в языке Transact-SQL в SQL Server.

Oracle также издавна поддерживает внешние соединения, но его синтаксис отличается от синтаксиса SQL Server. Здесь внешнее соединение указывается в предложении WHERE при помощи знака "плюс" в скобках, следующего за столбцом, *таблица которого является младшей* в соединении. Так, левое внешнее соединение таблиц GIRLS и BOYS получается в Oracle при помощи запроса следующего вида:

```
SELECT *
  FROM GIRLS, BOYS
 WHERE GIRLS.CITY = BOYS.CITY (+);
```

что эквивалентно стандартному запросу.

```
SELECT *
  FROM GIRLS LEFT OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;
```

Обратите внимание на то, что знак "плюс" ставится с *противоположной* стороны знака сравнения по отношению к звездочке в записи SQL Server. Аналогично правое внешнее соединение указывается с другой стороны от знака равенства.

```
SELECT *
  FROM GIRLS, BOYS
 WHERE GIRLS.CITY (+) = BOYS.CITY;
```

Oracle не поддерживает собственного вида полного внешнего соединения, но, как отмечалось ранее, это не умаляет практической пользы внешних соединений Oracle, и вы можете найти старую запись в существующих программах для работы с Oracle.

И запись SQL Server, и запись Oracle имеют существенные ограничения по сравнению со стандартной записью — например, когда при помощи внешнего соединения комбинируются три или большее количество таблиц, порядок соединения которых влияет на результаты запроса. Результат

```
(TBL1 OUTER JOIN TBL2) OUTER JOIN TBL3
```

в общем случае отличается от результата

```
TBL1 OUTER JOIN (TBL2 OUTER JOIN TBL3)
```

При использовании синтаксиса SQL Server или Oracle невозможно указать порядок вычисления внешних соединений. В связи с этим, результаты, получающиеся при внешнем соединении трех или большего числа таблиц, зависят от конкретной реализации СУБД. По этой и другим причинам новые программы следует всегда писать с применением стандартной записи внешнего соединения. Неплохо также при исправлениях старых программ заодно приводить внешние соединения к стандартному виду.

Соединения и стандарт SQL

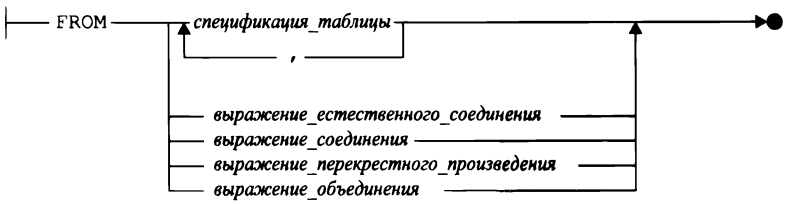
В стандарте SQL2 существенно расширена поддержка соединений с помощью новой, расширенной, формы записи предложения FROM, которая позволяет выразить даже самые сложные из соединений. На момент написания книги все основные SQL-продукты поддерживали все (или почти все) расширенные возможности соединений SQL2. Поддержка соединений обеспечена за счет существенного усложнения того, что раньше было одной из простейших частей SQL. Фактически расширенная поддержка соединений является частью гораздо большего расширения возможностей запросов в SQL2 и последующих версиях стандарта SQL, где имеется еще больше возможностей (и сложностей). Прочие расширенные возможности включают операции с множествами (соединение, пересечение и разность таблиц) и гораздо более богатые выражения в запросах для работы со стро-

ками и таблицами, а также обеспечивают возможность их использования во вложенных запросах. Эти возможности описаны в следующей главе, после рассмотрения базовых подзапросов.

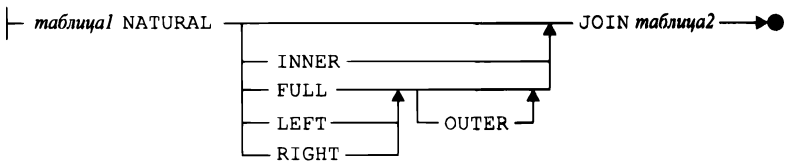
Внутренние соединения в стандарте SQL

На рис. 7.13 в упрощенном виде изображена синтаксическая диаграмма предложения FROM в стандарте SQL. Изучить все ее варианты легче всего, рассматривая по очереди каждый тип соединения, начиная с внутреннего и переходя к разным видам внешних соединений. Так, стандартное внутреннее соединение таблиц GIRLS и BOYS на первоначальной версии языка SQL можно записать так.

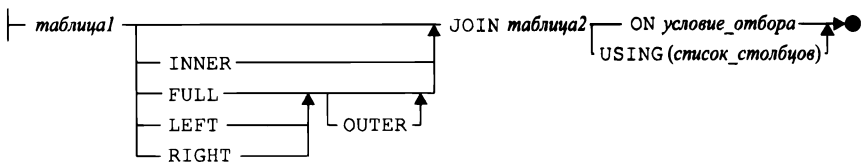
```
SELECT *
  FROM GIRLS, BOYS
 WHERE GIRLS.CITY = BOYS.CITY;
```



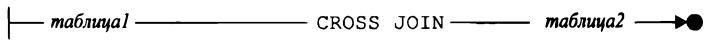
Выражение естественного соединения



Выражение соединения



Выражение перекрестного произведения



Выражение объединения

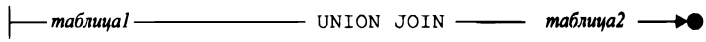


Рис. 7.13. Расширенное предложение FROM в стандарте SQL

В последней версии стандарта это по-прежнему допустимая инструкция. Создатели стандарта не могли отказаться от подобного синтаксиса, так как он применяется в миллионах многотабличных запросов, написанных с начала 1990-х годов. Но современный стандарт SQL предоставляет альтернативные способы записи внутреннего соединения, с которыми мы уже встречались в рассматриваемых примерах.

```
SELECT *
  FROM GIRLS INNER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;

SELECT *
  FROM GIRLS INNER JOIN BOYS
 USING (CITY);

SELECT *
  FROM GIRLS NATURAL INNER JOIN BOYS;
```

Ключевое слово INNER необязательно, так как соединение является внутренним по умолчанию. Спецификация NATURAL JOIN может использоваться в том случае, когда все идентично именованные в двух таблицах столбцы являются связанными; в противном случае для указания конкретных связанных столбцов следует применять предложение USING. В нашем случае связанными столбцами являются NAME и CITY, а поскольку никто из мальчиков не имеет такого же имени, как у какой-то из девочек, соединение NATURAL JOIN не возвращает ни одной строки. Если связанные столбцы имеют в разных таблицах разные имена или если требуется соединение по неравенству, для указания связанных столбцов следует воспользоваться полным предложением ON или WHERE. Предложения ON и WHERE поддерживаются существенно шире, чем NATURAL и USING.

Внешние соединения в стандарте SQL*

Мы уже видели поддержку внешних соединений в расширенном стандарте SQL — полное, левое и правое внешние соединения.

```
SELECT *
  FROM GIRLS FULL OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;

SELECT *
  FROM GIRLS LEFT OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;

SELECT *
  FROM GIRLS RIGHT OUTER JOIN BOYS
    ON GIRLS.CITY = BOYS.CITY;
```

Применение ключевого слова OUTER необязательно; СУБД в состоянии понять по наличию ключевых слов FULL, LEFT или RIGHT, что следует выполнить внешнее соединение. Результаты выполнения приведенных выше примеров бу-

дут различны: FULL OUTER JOIN вернет все строки из обеих таблиц; LEFT OUTER JOIN — все строки из левой таблицы (GIRLS), плюс связанные строки из правой таблицы (BOYS); а RIGHT OUTER JOIN вернет все строки из правой таблицы (BOYS), плюс связанные строки из левой таблицы (GIRLS). Как и в случае внутреннего соединения, естественное соединение может быть указано при помощи ключевого слова NATURAL, которое устраняет необходимость явного указания имен связанных столбцов. Аналогично связанные столбцы могут быть указаны в предложении USING.

Перекрестные соединения в стандарте SQL*

Поддержка расширенных соединений включает два других метода комбинации данных из двух таблиц. *Перекрестное соединение* (cross join) представляет собой другое название декартова произведения двух таблиц, описанного ранее в этой главе. Вот запрос, который генерирует полное произведение таблиц GIRLS и BOYS.

```
SELECT * FROM GIRLS CROSS JOIN BOYS;
```

По определению декартово произведение (иногда называемое *перекрестным произведением* (cross product), отсюда и название CROSS JOIN) содержит все возможные пары строк из двух таблиц. Оно является результатом "умножения" двух таблиц, превращая таблицы трех девочек и двух мальчиков в таблицу шести пар "девочка/мальчик" (3×2=6). Перекрестным соединениям не сопутствуют никакие "связанные столбцы" или "условия отбора", поэтому предложения ON и USING в них не допускаются. Следует отметить, что операция перекрестного соединения не добавляет ничего нового к возможностям SQL. Те же результаты можно получить с помощью внутреннего соединения, если не задать в нем условия отбора. Поэтому предыдущий запрос можно переписать так.

```
SELECT *
  FROM GIRLS, BOYS;
```

Ключевые слова CROSS JOIN в предложении FROM просто в явном виде указывают на то, что создается декартово произведение. Для большинства баз данных такая операция вряд ли будет представлять практический интерес. Она полезна лишь тогда, когда на основе полученной таблицы строятся более сложные выражения, например итоговые запросы (рассматриваются в следующей главе), или в последующей обработке применяются операции над множествами. На момент написания книги DB2 не поддерживал синтаксис перекрестного соединения, но тот же результат легко получить при помощи старого синтаксиса SQL.

Расширенное соединение (union join) заключает в себе некоторые особенности как операции UNION (которая рассматривалась в предыдущей главе), так и операции JOIN. Однако операция UNION JOIN объявлена как не рекомендуемая в стандарте SQL:1999 и полностью удалена в стандарте SQL:2003. Так что если вы пользуетесь СУБД, которая соответствует новейшему стандарту, вполне вероятно, что синтаксис UNION JOIN в ней не поддерживается. Он действительно не поддерживается текущими версиями Oracle, SQL Server, MySQL и DB2.

Вспомним, что операция UNION соединяет строки двух таблиц, у которых должно быть одинаковое число столбцов, причем типы соответствующих столбцов обеих таблиц должны быть совместимыми. Простейший запрос на соединение

```
SELECT *
  FROM GIRLS
 UNION ALL
SELECT *
  FROM BOYS;
```

будучи примененным к таблице GIRLS, состоящей из трех строк, и таблице BOYS, состоящей из двух строк, возвратит таблицу из пяти строк. Каждая из них в точности соответствует одной из строк таблицы GIRLS либо BOYS. В таблице результатов запроса будет два столбца, NAME и CITY, как и в обеих исходных таблицах.

Расширенное соединение этих же таблиц записывается так.

```
SELECT *
  FROM GIRLS
 UNION JOIN BOYS
```

Таблица результатов такого запроса снова будет содержать пять строк, каждая из которых образована ровно одной строкой из таблиц GIRLS либо BOYS. Но, в отличие от предыдущего случая, столбцов в полученной таблице будет четыре, а не два, — все столбцы из первой таблицы *плюс* все столбцы из второй таблицы. Здесь можно найти определенное сходство с внешним соединением. В каждой строке таблицы результатов запроса, построенной на основе строки из таблицы GIRLS, значения в столбцах, взятых из таблицы GIRLS, будут оставлены без изменений; все остальные столбцы (из таблицы BOYS) заполняются значениями NULL. Точно так же в каждой строке из таблицы BOYS значения в столбцах этой таблицы остаются неизменными, а столбцы из таблицы GIRLS заполняются значениями NULL.

Еще один способ взглянуть на результаты расширенного соединения — это сравнить их с полным внешним соединением таблиц GIRLS и BOYS. Результаты расширенного соединения включают строки данных из таблицы GIRLS с добавленными значениями NULL *и* строки данных из таблицы BOYS, расширенные значениями NULL, но *не* включают никакие строки, сгенерированные связанными столбцами. Возвращаясь к определению внешнего соединения, можно сказать, что на рис. 7.14 расширенное соединение получается при игнорировании шага 1 и выполнении шагов 2 и 3.

В заключение полезно будет рассмотреть различия наборов результатов, получаемых при выполнении различных типов соединений (рис. 7.14). Из рисунка видно, что при соединении двух таблиц — TBL1 с числом строк m и TBL2 с числом строк n — происходит следующее.

- *Перекрестное соединение* содержит $m \times n$ строк всех возможных пар строк обеих таблиц.

- `TBL1 INNER JOIN TBL2` содержит некоторое число строк *r*, меньшее *m*×*n*. Внутреннее соединение является строгим подмножеством перекрестного соединения. Оно образуется путем удаления из таблицы перекрестного соединения тех строк, которые не удовлетворяют условию внутреннего соединения.
- *Левое внешнее соединение* содержит все строки внутреннего соединения плюс расширенные значениями `NULL` строки таблицы `TBL1`, не удовлетворяющие условию соединения.

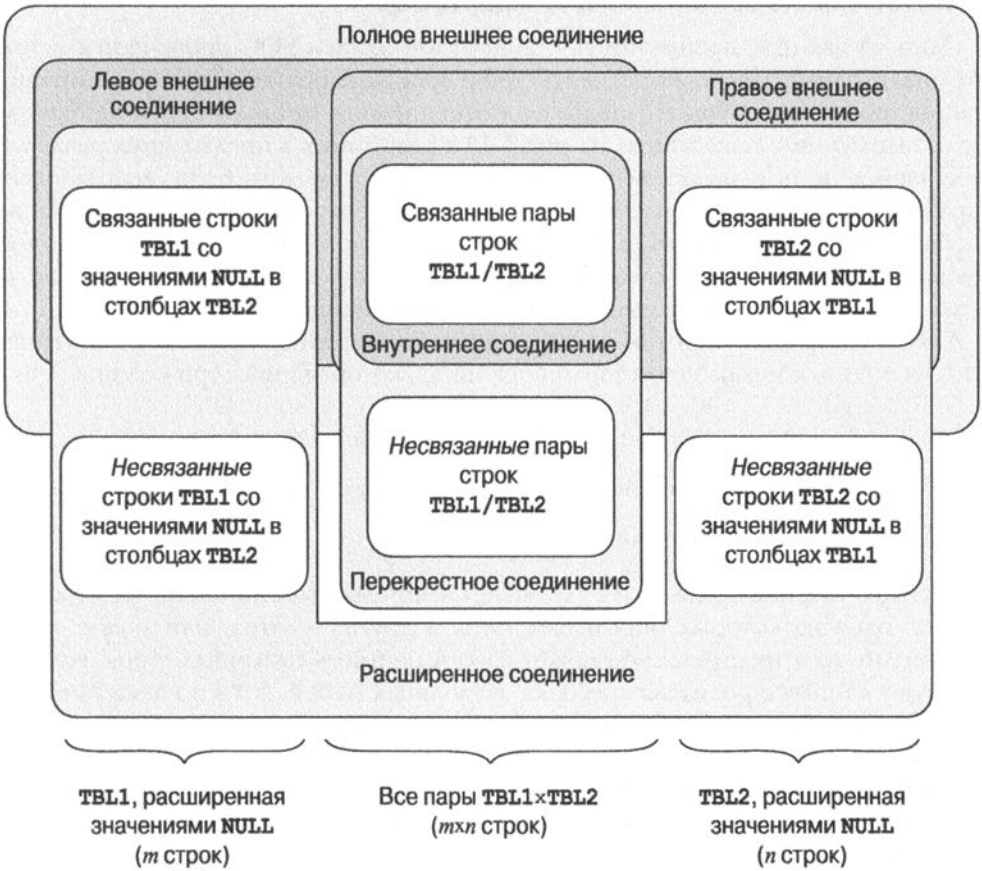


Рис. 7.14. Различные типы соединений

- *Правое внешнее соединение* содержит все строки внутреннего соединения плюс расширенные значениями `NULL` строки таблицы `TBL2`, не удовлетворяющие условию соединения.
- *Полное внешнее соединение* содержит все строки внутреннего соединения плюс расширенные значениями `NULL` строки таблицы `TBL1`, не удовлетворяющие условию соединения, плюс расширенные значениями `NULL` строки таблицы `TBL2`, не удовлетворяющие условию соединения. Грубо

говоря, результат запроса есть левое внешнее соединение плюс правое внешнее соединение.

- *Расширенное соединение* содержит расширенные значениями NULL строки таблицы TBL1, не удовлетворяющие условию соединения, плюс расширенные значениями NULL строки таблицы TBL2, не удовлетворяющие условию соединения. Грубо говоря, результат запроса есть полное внешнее соединение минус внутреннее соединение.

Многотабличные соединения в стандарте SQL

Одно из важных преимуществ стандартной записи SQL заключается в том, что она позволяет ясно определить трех- или четырехтабличные соединения. Для построения этих более сложных соединений можно заключить в скобки любое из выражений, показанных на рис. 7.13 и описанных в предыдущих разделах. Получаемое в результате выражение соединения может быть использовано в другом выражении соединения, как если бы это была простая таблица. Так же как SQL позволяет комбинировать математические операции (+, -, * и /) со скобками и строить более сложные выражения, стандарт SQL позволяет тем же путем создавать более сложные выражения соединений.

Для иллюстрации многотабличных соединений предположим, что к таблицам GIRLS и BOYS добавлена новая таблица PARENTS, которая имеет три столбца.

CHILD	Соответствует столбцу NAME в таблицах GIRLS и BOYS
TYPE	Принимает значение FATHER (отец) или MOTHER (мать)
PNAME	Имя родителя

Строка в таблице GIRLS или BOYS может иметь две связанные строки в таблице PARENT, одна из которых определяет мать, а другая — отца, или может иметь только одну из этих строк либо может совсем не иметь связанных строк, если отсутствуют данные о родителях ребенка. В таблицах GIRLS, BOYS и PARENTS в совокупности содержится достаточно богатый набор данных, чтобы обеспечить несколько примеров многотабличных запросов.

Предположим, например, что вы хотите составить список всех девочек и их матерей, а также мальчиков, которые живут в одном и том же городе. Вот запрос, создающий такой список.

```
SELECT GIRLS.NAME, PNAME, BOYS.NAME
  FROM ((GIRLS JOIN PARENTS
    ON PARENTS.CHILD = NAME)
  JOIN BOYS
    ON (GIRLS.CITY = BOYS.CITY))
 WHERE TYPE = 'MOTHER';
```

Так как оба соединения являются внутренними, то девочки, у которых нет живущих в том же городе мальчиков или у которых в базе данных не указаны матери, в таблицу результатов запроса не попадут. Это может быть желаемым результатом, а может и нет. Включить в результаты запроса девочек, у которых отсутст-

вует информация о матерях, можно путем замены соединения между таблицами GIRLS и PARENTS на левое внешнее соединение.

```
SELECT GIRLS.NAME, PNAME, BOYS.NAME
  FROM ((GIRLS LEFT JOIN PARENTS
    ON PARENTS.CHILD = NAME)
  JOIN BOYS
    ON (GIRLS.CITY = BOYS.CITY))
 WHERE (TYPE = 'MOTHER') OR (TYPE IS NULL);
```

В результатах этого запроса по-прежнему отсутствуют девочки, в одном городе с которыми не живет ни одного мальчика. Если их тоже нужно включить в запрос, следует сделать левым внешним и второе соединение.

```
SELECT GIRLS.NAME, PNAME, BOYS.NAME
  FROM ((GIRLS LEFT JOIN PARENTS
    ON PARENTS.CHILD = NAME)
  LEFT JOIN BOYS
    ON (GIRLS.CITY = BOYS.CITY))
 WHERE (TYPE = 'MOTHER') OR (TYPE IS NULL);
```

Обратите внимание на то, что расширение значениями NULL строк таблицы GIRLS при внешнем соединении с матерями приводит к дополнительному усложнению в предложении WHERE. Если у какой-либо девочки отсутствует информация о матери, то не только столбец PNAME, но и столбец TYPE будет иметь значение NULL. Поэтому простая проверка

```
WHERE (TYPE = 'MOTHER')
```

для таких строк вернет значение "неизвестно", и они не будут включены в результаты запроса. Но ведь это противоречит смыслу самого запроса! Для решения данной проблемы в предложение WHERE добавлена вторая проверка на равенство столбца TYPE значению NULL.

В качестве завершающего примера предположим, что вы опять, как в предыдущих случаях, хотите найти пары "девочка/мальчик" из одного и того же города, но на этот раз хотите еще и включить в таблицу результатов имя отца мальчика и имя матери девочки. Такой запрос требует четырехтабличного соединения (таблицы BOYS, GIRLS и две копии таблицы PARENTS: одна для соединения с таблицей BOYS с целью получения имен отцов, а вторая для соединения с таблицей GIRLS с целью получения имен матерей). И вновь то, что в соединениях данного примера могут присутствовать несвязанные строки, означает существование нескольких возможных "правильных" ответов на запрос. Предположим, как и ранее, что вам нужно включить в результаты запроса все пары "девочка/мальчик" из одних и тех городов, даже те, в которых мальчик или девочка не имеют связанных строк с таблицей PARENTS. В этом запросе придется использовать два внешних соединения (между таблицами BOYS и PARENTS, а также таблицами GIRLS и PARENTS) и одно внутреннее (между таблицами BOYS и GIRLS). Вот как выглядит запрос, который выдаст интересующую нас информацию.

```
SELECT GIRLS.NAME, MOTHERS.PNAME, BOYS.NAME, FATHERS.PNAME
  FROM GIRLS LEFT JOIN PARENTS AS MOTHERS
    ON ((MOTHERS.CHILD = GIRLS.NAME)
          AND (MOTHERS.TYPE = 'MOTHER'))
  JOIN BOYS ON (GIRLS.CITY = BOYS.CITY)
       LEFT JOIN PARENTS AS FATHERS
    ON ((FATHERS.CHILD = BOYS.NAME)
          AND (FATHERS.TYPE = 'FATHER'));
```

В этом запросе проблема проверки столбца TYPE в предложении WHERE решена по-другому: сама проверка перемещена в предложение ON обеих операций соединения. В этом случае столбец TYPE будет проверяться на этапе построения каждого соединения, когда строки, расширенные значениями NULL, еще не добавлены в таблицу результатов запроса. Поскольку таблица PARENTS встречается в предложениях FROM дважды в разных ролях, необходимо назначить ей два псевдонима, чтобы в предложении SELECT можно было указать правильные столбцы.

Как видно из примера, даже запрос с тремя соединениями, в соответствии с расширенным синтаксисом стандарта SQL, может иметь весьма сложный вид. Синтаксис у разных реализаций SQL может отличаться. Например, Oracle не допускает ключевое слово AS между именем таблицы и псевдонимом в предложении JOIN. Однако, несмотря на сложность, запрос *точно* определяет то, что должна выполнить СУБД. Нет никакой неоднозначности в отношении порядка соединения таблиц или в том, какие соединения являются внешними, а какие — внутренними. В общем, новые возможности стоят дополнительных сложностей расширенного предложения FROM обновленного стандарта SQL.

Хотя примеры запросов в этом разделе не включали предложений WHERE и ORDER BY, они могут свободно использоваться с расширенной поддержкой соединений. Связь между ними простая и остается такой, как было описано ранее в данной главе. Сначала выполняются операции, указанные в предложении FROM, включая все соединения. Условия соединения, заданные в предложениях USING или ON, выступают как часть конкретного соединения, по отношению к которому они определены. Когда выполнение операций в предложении FROM заканчивается, к таблице результатов применяются условия отбора, заданные в предложении WHERE. Таким образом, в предложении ON задаются условия отбора, применяемые к отдельным соединениям; в предложении WHERE задается условие отбора, которое применяется к результирующей таблице этих соединений.

В табл. 7.1 резюмируется синтаксис соединений SQL (как старая, так и новая, стандартная, версия) с применением примеров из данной главы.

Таблица 7.1. Синтаксис соединений SQL

Тип	Старый синтаксис	Стандартный синтаксис	Описание
Внутренние соединения[1]			
Простое соединение по равенству	SELECT NAME, CITY FROM SALESREPS, OFFICES WHERE REP_OFFICE = OFFICE;	SELECT NAME, CITY FROM SALESREPS JOIN OFFICES ON REP_OFFICE = OFFICE;	Образует пары строк с одинаковым содержимым связанных столбцов
Явное соединение по равенству	–	SELECT NAME, CITY FROM SALESREPS INNER JOIN OFFICES ON REP_OFFICE = OFFICE;	Вариант синтаксиса с применением ключевых слов INNER JOIN вместо простого JOIN
Запрос "предок-потомок"	SELECT CITY, NAME, TITLE FROM OFFICES, SALESREPS WHERE MGR = EMPL_NUM;	SELECT CITY, NAME, TITLE FROM OFFICES JOIN SALESREPS ON MGR = EMPL_NUM;	Соединение по равенству, в котором проверяется соответствие основного ключа одной таблицы внешнему ключу другой
Условие отбора строк	SELECT CITY, NAME, TITLE FROM OFFICES, SALESREPS WHERE MGR = EMPL_NUM AND TARGET > 600000.00;	SELECT CITY, NAME, TITLE FROM OFFICES JOIN SALESREPS ON MGR = EMPL_NUM WHERE TARGET > 600000.00;	Нежелательные строки отфильтровываются из результата запроса с помощью предиката WHERE
Несколько связанных столбцов	SELECT ORDER_NUM, AMOUNT, DESCRIPTION FROM ORDERS, PRODUCTS WHERE MFR = MFR_ID AND PRODUCT = PRODUCT_ID;	SELECT ORDER_NUM, AMOUNT, DESCRIPTION FROM ORDERS JOIN PRODUCTS ON MFR = MFR_ID AND PRODUCT = PRODUCT_ID;	Многостолбчатые первичный и внешний ключи требуют соответствия нескольких столбцов в предикате соединения
Трехтабличное соединение	SELECT ORDER_NUM, AMOUNT, COMPANY, NAME FROM ORDERS, CUSTOMERS, SALESREPS WHERE CUST = CUST_NUM AND REP = EMPL_NUM AND AMOUNT > 25000.00;	SELECT ORDER_NUM, AMOUNT, COMPANY, NAME FROM ORDERS JOIN CUSTOMERS ON CUST = CUST_NUM JOIN SALESREPS ON REP = EMPL_NUM WHERE AMOUNT > 25000.00;	Более двух таблиц соединяются путем добавления дополнительных предложений JOIN
Соединение по неравенству	SELECT NAME, QUOTA, TARGET FROM SALESREPS, OFFICES WHERE QUOTA > TARGET;	SELECT NAME, QUOTA, TARGET FROM SALESREPS JOIN OFFICES ON QUOTA > TARGET;	Оператор сравнения в предикате соединения отличен от равенства (=)
Естественное соединение	SELECT ORDER_NUM, AMOUNT, DESCRIPTION FROM ORDERS, PRODUCTS WHERE MFR = MFR_ID AND PRODUCT = PRODUCT_ID;	SELECT ORDER_NUM, AMOUNT, DESCRIPTION FROM ORDERS NATURAL JOIN PRODUCTS;[4]	Соединение по равенству, когда связанные столбцы имеют одни и те же имена в соединяемых таблицах
Соединение с предложением USING	–	SELECT ORDER_NUM, AMOUNT, DESCRIPTION FROM ORDERS JOIN PRODUCTS USING (MFR, PRODUCT);[5]	Соединение по равенству с явным указанием имен связанных столбцов с одним и тем же именем в соединяемых таблицах

Тип	Старый синтаксис	Стандартный синтаксис	Описание
Самосоединение	SELECT EMPS.NAME, MGRS.NAME FROM SALESREPS EMPS, SALESREPS MGRS WHERE EMPS.MANAGER = MGRS.EMPL_NUM;	SELECT EMPS.NAME, MGRS.NAME FROM SALESREPS EMPS JOIN SALESREPS MGRS ON EMPS.MANAGER = MGRS.EMPL_NUM;	Соединение по равенству таблицы с самой собой, когда каждая строка связана с другой в той же самой таблице
Внешние соединения[2,3]			
Полное внешнее соединение	SELECT * FROM GIRLS, BOYS WHERE GIRLS.CITY = BOYS.CITY(+) UNION SELECT * FROM GIRLS, BOYS WHERE GIRLS.CITY(+) = BOYS.CITY;	SELECT * FROM GIRLS FULL OUTER JOIN BOYS ON GIRLS.CITY = BOYS.CITY;	Добавляет к результатам запроса расширенную значениями NULL строку для каждой несвязанной строки каждой из соединяемых таблиц
Естественное полное внешнее соединение	–	SELECT * FROM GIRLS NATURAL FULL OUTER JOIN BOYS;	Полное внешнее соединение, основанное на всех связанных столбцах с одинаковыми именами в соединяемых таблицах
Полное внешнее соединение с предложением USING	–	SELECT * FROM GIRLS FULL OUTER JOIN BOYS USING (CITY);	Полное внешнее соединение на основе явно указанных имен столбцов с одинаковыми именами в соединяемых таблицах
Полное внешнее соединение с подразумеваемым ключевым словом OUTER	–	SELECT * FROM GIRLS FULL JOIN BOYS USING (CITY);	Многие реализации SQL позволяют опустить ключевое слово OUTER, которое подразумевается ключевым словом FULL
Левое внешнее соединение	SELECT * FROM GIRLS, BOYS WHERE GIRLS.CITY = BOYS.CITY(+);	SELECT * FROM GIRLS LEFT OUTER JOIN BOYS ON GIRLS.CITY = BOYS.CITY;	Для каждой несвязанной строки из первой (левой) таблицы добавляет к результатам запроса строку, расширенную значениями NULL
Внешние соединения[4,5]			
Левое внешнее соединение с предложением USING	–	SELECT * FROM GIRLS LEFT OUTER JOIN BOYS USING (CITY);	Левое внешнее соединение на основе явно указанных имен столбцов с одинаковыми именами в соединяемых таблицах

Тип	Старый синтаксис	Стандартный синтаксис	Описание
Правое внешнее соединение	SELECT * FROM GIRLS, BOYS WHERE GIRLS.CITY(+) = BOYS.CITY;	SELECT * FROM GIRLS RIGHT OUTER JOIN BOYS ON GIRLS.CITY = BOYS.CITY;	Для каждой несвязанной строки из второй (правой) таблицы добавляет к результатам запроса строку, расширенную значениями NULL
Правое внешнее соединение с предложением USING	–	SELECT * FROM GIRLS RIGHT OUTER JOIN BOYS USING (CITY);	Правое внешнее соединение на основе явно указанных имен столбцов с одинаковыми именами в соединяемых таблицах
Внешние соединения			
Перекрестное соединение	SELECT * FROM GIRLS, BOYS;	SELECT * FROM GIRLS CROSS JOIN BOYS;	Явный запрос декартова произведения таблиц, которое представляет собой таблицу, состоящую из всех возможных пар строк из обеих таблиц
Расширенное соединение (объединение)	SELECT * FROM GIRLS UNION ALL SELECT * FROM BOYS;	SELECT * FROM GIRLS UNION ALL SELECT * FROM BOYS;	Технически соединением не является; инструкции SELECT обрабатываются независимо, а результирующие множества соединяются оператором UNION

[1] Внутренние соединения потенциально могут приводить к потере информации при наличии в соединяемых таблицах несвязанных строк.

[2] Внешние соединения не теряют информацию, поскольку добавляют к результату запроса несвязанные строки.

[3] В примерах старого синтаксиса использован синтаксис Oracle.

[4] Этот запрос приведен только для иллюстрации, так как в таблицах ORDERS и PRODUCTS учебной базы данных нет столбцов с одинаковыми именами. При попытке выполнения такого или подобного естественного соединения таблиц, в которых нет столбцов с одинаковыми именами, многие СУБД вернут декартово произведение.

[5] Этот запрос приведен только для иллюстрации и не будет работать в учебной базе данных, так как в таблице PRODUCTS нет столбцов MFR и PRODUCT.

Резюме

В настоящей главе рассмотрены запросы SQL, соединяющие данные из двух и более таблиц.

- Имена таблиц, из которых берутся данные, указываются в многотабличном запросе (*соединении*) в предложении FROM.
- Каждая строка таблицы результатов запроса соединяет по одной строке из каждой исходной таблицы и является *единственной* строкой, содержащей комбинацию данных строк.
- В многотабличных запросах чаще всего используются отношения "предок-потомок", создаваемые первичными и внешними ключами.
- В общем случае соединения можно создавать путем сравнения *любых* пар столбцов из двух соединяемых таблиц, используя условие равенства или любое другое условие сравнения.
- Соединение можно представить как произведение двух таблиц, из которого удалена часть строк.
- Таблицу можно соединять саму с собой; при таком самосоединении необходимо использовать псевдонимы таблицы.
- Внешнее соединение является расширением стандартного (внутреннего) соединения, добавляющим в таблицу результатов запроса несвязанные строки одной или обеих исходных таблиц, при этом ячейки, соответствующие столбцам другой таблицы, заполняются значениями NULL.
- Стандарт SQL обеспечивает полную поддержку внутренних и внешних соединений и позволяет соединять результаты соединений с другими многотабличными операциями, такими как объединения запросов, пересечения и разности.

ГЛАВА 8

Итоговые запросы

Многие запросы к базе данных не требуют того уровня детализации, который обеспечивают SQL-запросы, рассмотренные в двух предыдущих главах. Например, во всех перечисленных ниже запросах требуется узнать всего одно или несколько значений, которые подытоживают информацию, содержащуюся в базе данных.

- Какова общая сумма плановых объемов продаж для всех служащих?
- Каковы наибольший и наименьший плановые объемы продаж?
- Сколько служащих перевыполнили свой план?
- Какова средняя стоимость заказа?
- Какова средняя стоимость заказа в каждом офисе?
- Сколько служащих закреплено за каждым офисом?

В SQL запросы такого типа можно создавать с помощью агрегирующих функций и предложений `GROUP BY` и `HAVING` инструкции `SELECT`, описанных в данной главе.

Агрегирующие функции

SQL позволяет получить итоговые данные с использованием набора *статистических*, или *агрегирующих*, *функций* (column functions). Такая функция принимает в качестве аргумента какой-либо столбец данных целиком и возвращает единственное итоговое значение для столбца. Например, функция `AVG()` принимает в качестве аргумента столбец чисел и вычисляет их среднее значение. Ниже приведен запрос, в котором функция `AVG()` используется для вычисления среднего значения в двух столбцах таблицы `SALESREPS`.

Каковы средний плановый и средний фактический объемы продаж у продавцов?

```
SELECT AVG(QUOTA), AVG(SALES)
  FROM SALESREPS;

AVG(QUOTA)      AVG(SALES)
-----------     -----------
$300,000.00     $289,353.20
```

На рис. 8.1 изображена схема выполнения такого запроса. Первая функция принимает в качестве аргументов все значения, содержащиеся в столбце QUOTA, и вычисляет их среднее значение; вторая функция подсчитывает среднее значение столбца SALES. Результатом запроса является одна строка, представляющая итоги по информации, содержащейся в таблице SALESREPS.

Таблица **SALESREPS**

EMPL_NUM	NAME	MANAGER	QUOTA	SALES
105	Bill Adams	104	$350,000.00	$367,911.00
109	Mary Jones	106	$300,000.00	$392,725.00
102	Sue Smith	108	$350,000.00	$474,050.00
106	Sam Clark	NULL	$275,000.00	$299,912.00
104	Bob Smith	106	$200,000.00	$142,594.00
101	Dan Roberts	104	$300,000.00	$305,673.00
110	Tom Snyder	101	NULL	$75,985.00
108	Larry Fitch	106	$350,000.00	$361,865.00
103	Paul Cruz	104	$275,000.00	$286,775.00
107	Nancy Angelli	108	$300,000.00	$186,042.00

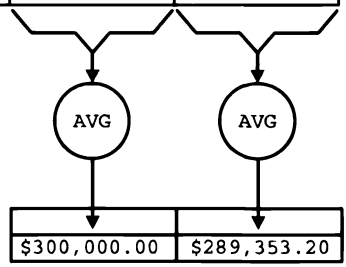


Рис. 8.1. Выполнение итогового запроса

В стандарте SQL определен ряд таких функций; помимо них, многие производители СУБД добавляют собственные функции в своих реализациях SQL. Шесть наиболее распространенных функций, показанных на рис. 8.2, позволяют получать различные виды итоговой информации.

- Функция SUM() вычисляет сумму всех значений столбца.
- Функция AVG() вычисляет среднее всех значений столбца.
- Функция MIN() находит наименьшее среди всех значений столбца.
- Функция MAX() находит наибольшее среди всех значений столбца.
- Функция COUNT() подсчитывает количество значений, содержащихся в столбце.
- Функция COUNT(*) подсчитывает количество строк в таблице результатов запроса (фактически это альтернативная форма функции COUNT()).

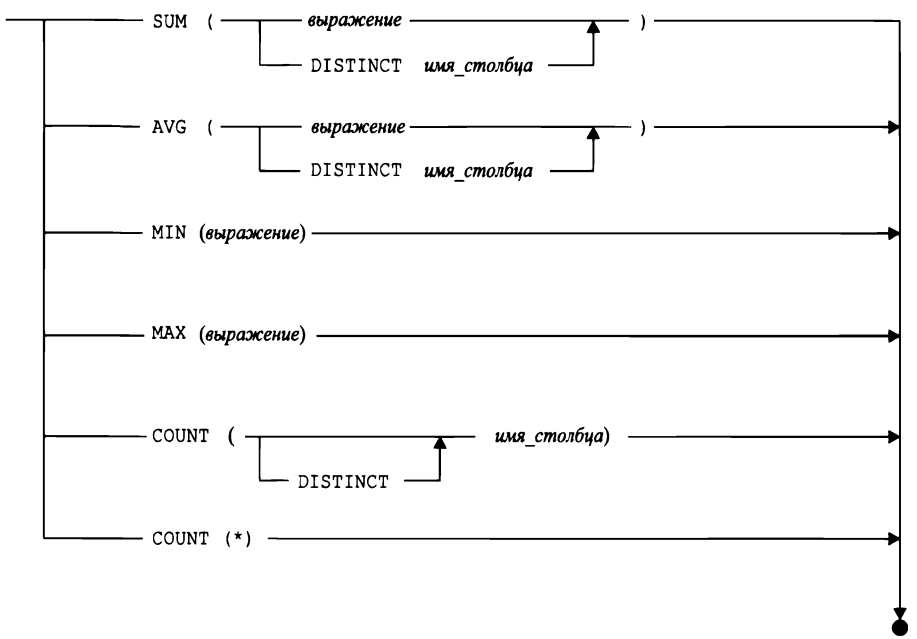


Рис. 8.2. Синтаксическая диаграмма статистических функций

Аргументом статистической функции может быть как простое имя столбца, показанное в предыдущем примере, так и выражение, как в следующем запросе.

Каков средний процент выполнения плана в компании?

```
SELECT AVG(100 * (SALES/QUOTA))
  FROM SALESREPS;

AVG(100 * (SALES/QUOTA))
------------------------
                  102.60
```

При выполнении этого запроса СУБД создает временный столбец, содержащий значения `100*(SALES/QUOTA)` для каждой строки таблицы `SALESREPS`, а затем вычисляет среднее значение временного столбца.

Вычисление суммы значений столбца

Статистическая функция `SUM()` вычисляет сумму всех значений столбца. При этом столбец должен иметь числовой тип данных (содержать целые числа, десятичные числа, числа с плавающей точкой или денежные величины). Результат, возвращаемый этой функцией, имеет тот же тип данных, что и столбец, однако точность результата может быть выше. Например, если применить функцию `SUM()` к столбцу, содержащему 16-разрядные целые числа, она может вернуть в качестве результата 32-разрядное целое число.

Ниже приведен ряд примеров, в которых используется функция `SUM()`.

Каковы общий плановый и общий фактический объемы продаж?

```
SELECT SUM(QUOTA), SUM(SALES)
  FROM SALESREPS;

   SUM(QUOTA)         SUM(SALES)
-------------      -------------
$2,700,000.00      $2,893,532.00
```

Какова сумма всех заказов, принятых Биллом Адамсом?

```
SELECT SUM(AMOUNT)
  FROM ORDERS, SALESREPS
 WHERE NAME = 'Bill Adams'
   AND REP = EMPL_NUM;

  SUM(AMOUNT)
------------
  $39,327.00
```

Вычисление среднего значений столбца

Статистическая функция AVG() вычисляет среднее всех значений столбца. Как и в случае с функцией SUM(), данные, содержащиеся в столбце, должны иметь числовой тип. Поскольку функция AVG() вначале суммирует все значения, содержащиеся в столбце, а затем делит сумму на количество этих значений, возвращаемый ею результат может иметь тип данных, не совпадающий с типом данных столбца. Например, если применить функцию AVG() к столбцу целых чисел, результат будет либо десятичным числом, либо числом с плавающей точкой, в зависимости от используемой СУБД.

Ниже приведено несколько примеров использования функции AVG().

Вычислить среднюю цену товаров от производителя ACI.

```
SELECT AVG(PRICE)
  FROM PRODUCTS
 WHERE MFR_ID = 'ACI';

AVG(PRICE)
----------
   $804.29
```

Вычислить среднюю стоимость заказов, сделанных компанией Acme Mfg. (клиент 2103).

```
SELECT AVG(AMOUNT)
  FROM ORDERS
 WHERE CUST = 2103;

AVG(AMOUNT)
-----------
  $8,895.50
```

Вычисление предельных значений

Статистические функции MIN() и MAX() позволяют найти соответственно наименьшее и наибольшее значения в столбце. При этом столбец может содержать числа, строки либо значения даты/времени. Результат, возвращаемый этими функциями, имеет точно тот же тип данных, что и сам столбец.

Ниже приведен ряд примеров, иллюстрирующих использование упомянутых функций.

Каковы наибольший и наименьший плановые объемы продаж?

```
SELECT MIN(QUOTA), MAX(QUOTA)
  FROM SALESREPS;

  MIN(QUOTA)    MAX(QUOTA)
------------  -----------
 $200,000.00  $350,000.00
```

Когда был сделан самый первый из всех содержащихся в базе данных заказов?

```
SELECT MIN(ORDER_DATE)
  FROM ORDERS;

MIN(ORDER_DATE)
---------------
2007-01-04
```

Каков наибольший процент выполнения плана среди всех служащих?

```
SELECT MAX(100 * (SALES/QUOTA))
  FROM SALESREPS;

MAX(100 * (SALES/QUOTA))
------------------------
                  135.44
```

В случае применения функций MIN() и MAX() к числовым данным числа сравниваются по арифметическим правилам (среди двух отрицательных чисел меньше то, у которого модуль больше; нуль меньше любого положительного числа и больше любого отрицательного). Сравнение дат происходит последовательно (более ранние значения дат считаются меньшими, чем более поздние). Сравнение интервалов времени выполняется на основании их продолжительности (более короткие интервалы времени считаются меньшими, чем более длинные).

В случае применения функций MIN() и MAX() к строковым данным результат сравнения двух строк зависит от используемой таблицы кодировки. На персональных компьютерах и мини-компьютерах, где используется таблица кодировки ASCII, установлен порядок сортировки, при котором цифры идут перед буквами, а все прописные буквы — перед строчными. На мэйнфреймах компании IBM, где используется таблица кодировки EBCDIC, строчные символы расположены перед прописными, а цифры следуют за буквами. Ниже приведено сравнение последовательностей сортировки, принятых в таблицах кодировки ASCII и EBCDIC, на примере строк, упорядоченных по возрастанию.

ASCII	EBCDIC
1234ABC	acme mfg.
5678ABC	zeta corp.
ACME MFG.	Acme Mfg.
Acme Mfg.	ACME MFG.
ZETA CORP.	Zeta Corp.
Zeta Corp.	ZETA CORP.
acme mfg.	1234ABC
zeta corp.	5678ABC

Отличия в порядке сортировки приводят к тому, что один и тот же запрос, содержащий предложение ORDER BY, в различных системах может привести к разным результатам.

Хранение в таблицах символов национальных алфавитов (например, кириллицы) может вызвать дополнительные проблемы. В некоторых СУБД для каждого языка используется свой алгоритм сортировки. В других СУБД такие символы сортируются в соответствии с кодом символа. Для решения этой проблемы в стандарт SQL включены поддержка национальных наборов символов, пользовательских наборов символов и альтернативных последовательностей сортировки. К сожалению, поддержка этих возможностей сильно отличается от производителя к производителю. Если в вашем приложении используются символы национальных алфавитов, необходимо поэкспериментировать с вашей конкретной СУБД, чтобы выяснить, как она их обрабатывает.

Подсчет количества данных

Статистическая функция COUNT() подсчитывает количество значений в столбце; тип данных столбца при этом роли не играет. Функция COUNT() всегда возвращает целое число, независимо от типа данных столбца. Ниже приведен ряд запросов, в которых используется эта функция.

Сколько клиентов у нашей компании?

```
SELECT COUNT(CUST_NUM)
  FROM CUSTOMERS;

 COUNT(CUST_NUM)
----------------
              21
```

Сколько служащих перевыполнили план?

```
SELECT COUNT(NAME)
  FROM SALESREPS
 WHERE SALES > QUOTA$

COUNT(NAME)
-----------
          7
```

Сколько имеется заказов стоимостью более $25000?

```
SELECT COUNT(AMOUNT)
  FROM ORDERS
 WHERE AMOUNT > 25000.00;

COUNT(AMOUNT)
-------------
            4
```

Обратите внимание на то, что функция COUNT() с переданным именем столбца не учитывает значения NULL в этом столбце; это делает функция COUNT(*), которая подсчитывает все строки независимо от их значений. Если же не рассматривать значения NULL, то функция COUNT() игнорирует значения данных в столбце и просто подсчитывает их количество. Таким образом, не имеет значения, какой именно столбец передан функции COUNT() в качестве аргумента. Так, последний пример может быть переписан следующим образом.

```
SELECT COUNT(ORDER_NUM)
  FROM ORDERS
 WHERE AMOUNT > 25000.00;

 COUNT(ORDER_NUM)
-----------------
                4
```

Мысленно трудно представить запрос вроде "подсчитать, сколько стоимостей заказов" или "подсчитать, сколько номеров заказов"; гораздо проще представить запрос "подсчитать, сколько заказов". Поэтому в SQL была введена специальная статистическая функция COUNT(*), которая подсчитывает строки, а не значения данных. Вот предыдущий запрос, переписанный с использованием этой функции.

```
SELECT COUNT(*)
  FROM ORDERS
 WHERE AMOUNT > 25000.00;

COUNT(*)
--------
       4
```

Если применять COUNT(*) в качестве функции подсчета строк, то запрос становится более удобочитаемым. На практике для подсчета строк всегда используется функция COUNT(*), а не COUNT().

Статистические функции в списке возвращаемых столбцов

Назначение простого запроса, в котором участвуют статистические функции, понять достаточно легко. Однако если в список возвращаемых столбцов входит несколько таких функций или аргументом функции является сложное выражение, понять запрос становится значительно сложнее. Ниже приведены шаги выполнения SQL-запросов, в очередной раз расширенные с учетом применения статистических функций. Как и ранее, эти правила являются точным определением того,

что означает запрос, а не описанием того, как СУБД на самом деле получает результаты запроса.

Для генерации результатов запроса SELECT следует выполнить следующие шаги.

1. Если запрос представляет собой объединение (UNION) инструкций SELECT, для каждой из них выполнить шаги 2–5 для генерации отдельных результатов запросов.
2. Сформировать произведение таблиц, указанных в предложении FROM. Если там указана только одна таблица, то произведением будет она сама.
3. При наличии предложения WHERE применить указанное в нем условие к каждой строке таблицы произведения, оставляя только те строки, для которых условие истинно, и отбрасывая строки, для которых условие ложно или равно NULL.
4. Для каждой из оставшихся строк вычислить значение каждого элемента в списке возвращаемых столбцов и тем самым создать одну строку таблицы результатов запроса. При любой ссылке на столбец используется значение столбца в текущей строке. В качестве аргумента статистической функции используется весь набор строк.
5. Если указан предикат DISTINCT, удалить из таблицы результатов запроса все повторяющиеся строки.
6. Если запрос является объединением инструкций SELECT, объединить результаты выполнения отдельных инструкций в одну таблицу результатов запроса. Удалить из нее повторяющиеся строки, если только в запросе не указан предикат UNION ALL.
7. Если имеется предложение ORDER BY, отсортировать результаты запроса, как указано в нем.

Строки, генерируемые этой процедурой, составляют результаты запроса.

Один из лучших способов понять, как выполняются итоговые запросы со статистическими функциями, — представить запрос разбитым на два этапа. Сначала подумайте, как работал бы запрос *без* статистических функций, возвращая несколько строк детального результата запроса. Затем представьте, как СУБД применяет статистические функции к детальным результатам запроса, возвращая одну итоговую строку. Например, рассмотрим следующий сложный запрос.

Найти среднюю стоимость заказов, общую стоимость заказов, среднюю стоимость заказов в процентах от лимита кредита клиентов, а также среднюю стоимость заказов в процентах от плановых объемов продаж служащих.

```
SELECT AVG(AMOUNT), SUM(AMOUNT),
       (100 * AVG(AMOUNT/CREDIT_LIMIT)),
       (100 * AVG(AMOUNT/QUOTA))
  FROM ORDERS, CUSTOMERS, SALESREPS
 WHERE CUST = CUST_NUM
   AND REP = EMPL_NUM;
```

```
AVG(AMOUNT) SUM(AMOUNT) (100*AVG(AMOUNT/CREDIT_LIMIT))
----------- ----------- ------------------------------
  $8,256.37 $247,691.00                          24.44

 (100*AVG(AMOUNT/QUOTA))
 -----------------------
                    2.51
```

Размер строки оказался слишком большим из-за очень длинных заголовков столбцов. Если вы используете SQL-клиент, вывод которого ограничен 80 или менее символами, то каждая строка окажется разбита на несколько, и результат будет не столь удобочитаемым, как показано здесь. Позже вы узнаете, как использовать псевдонимы строк для устранения длинных заголовков, генерируемых СУБД при наличии статистических функций.

Без статистических функций запрос выглядел бы следующим образом.

```
SELECT AMOUNT, AMOUNT, AMOUNT/CREDIT_LIMIT, AMOUNT/QUOTA
  FROM ORDERS, CUSTOMERS, SALESREPS
 WHERE CUST = CUST_NUM
   AND REP = EMPL_NUM;
```

Этот запрос возвращал бы одну строку результатов запроса для каждого заказа. Статистические функции используют столбцы таблицы результатов запроса для получения однострочной таблицы с итоговыми результатами.

В инструкции SQL статистическая функция может использоваться везде, где можно указать имя столбца. Например, она может входить в выражение, в котором суммируются или вычитаются значения двух статистических функций. Однако в некоторых реализациях SQL, в частности, основанных на стандарте SQL1, аргумент статистической функции не может содержать другую такую функцию, поскольку получаемое в таком случае выражение не имеет смысла. Иногда это правило формулируют таким образом: "статистические функции не должны быть вложенными".

Кроме того, в списке возвращаемых столбцов нельзя одновременно использовать статистические функции и обычные имена столбцов (за исключением запросов с группировкой и подзапросов), поскольку в этом также нет смысла. Например, рассмотрим запрос.

```
SELECT NAME, SUM(SALES)
  FROM SALESREPS;
```

Первый элемент списка возвращаемых столбцов просит СУБД создать таблицу, которая будет состоять из десяти строк и содержать обычные результаты запроса — по одной строке для каждого служащего. Второй элемент списка возвращаемых столбцов просит СУБД получить одно результирующее значение, представляющее собой сумму значений столбца SALES. Эти элементы инструкции SELECT противоречат друг другу, что приводит к ошибке. По этой причине либо все ссылки на столбцы в списке возвращаемых столбцов должны являться аргументами статистических функций (и тогда запрос возвращает итоговые результаты), либо в списке возвращаемых столбцов не должно быть ни одной статистической функции (и тогда запрос возвращает обычные результаты), за исключением некоторых случаев, описанных ниже в данной главе.

Статистические функции и значения NULL

Функции SUM(), AVG(), MIN(), MAX() и COUNT() в качестве аргумента принимают столбец значений и возвращают в качестве результата одно значение. А что происходит, когда в столбце встречается одно или несколько значений NULL? В стандарте ANSI/ISO сказано, что значения NULL статистическими функциями *игнорируются*.

Следующий запрос показывает, что статистическая функция COUNT() игнорирует все значения NULL, содержащиеся в столбце.

```
SELECT COUNT(*), COUNT(SALES), COUNT(QUOTA)
  FROM SALESREPS;

COUNT(*)    COUNT(SALES)      COUNT(QUOTA)
--------    ------------      ------------
      10              10                 9
```

В таблице SALESREPS содержится десять строк, поэтому функция COUNT(*) возвращает число 10. В столбце SALES содержится десять значений, причем ни одно из них не равно NULL, поэтому функция COUNT(SALES) также возвращает число 10. А вот в столбце QUOTA содержится одно значение NULL — для служащего, принятого совсем недавно. Функция COUNT(QUOTA) игнорирует это значение и возвращает число 9. Именно из-за таких расхождений вместо функции COUNT() для подсчета строк почти всегда используется функция COUNT(*). Исключение составляют случаи, когда необходимо не учитывать строки, содержащие значения NULL в определенном столбце.

Игнорирование значений NULL не оказывает влияния на результаты, возвращаемые статистическими функциями MIN() и MAX(). Однако оно может привести к проблемам при использовании функций SUM() и AVG(), что иллюстрирует следующий запрос.

```
SELECT SUM(SALES), SUM(QUOTA),
       (SUM(SALES) - SUM(QUOTA)),
        SUM(SALES-QUOTA)
  FROM SALESREPS

   SUM(SALES)      SUM(QUOTA)  (SUM(SALES) - SUM(QUOTA))
-------------  -------------  -------------------------
$2,893,532.00  $2,700,000.00                193,532.00

  SUM(SALES-QUOTA)
  ----------------
        $117,547.00
```

Можно ожидать, что выражения

```
(SUM(SALES) - SUM(QUOTA))
```

и

```
SUM(SALES-QUOTA)
```

вернут одинаковые результаты, однако пример показывает, что так не происходит. И снова причиной является строка со значением NULL в столбце QUOTA. Выражение

```
SUM(SALES)
```

вычисляет сумму продаж для всех десяти служащих, а выражение

```
SUM(QUOTA)
```

вычисляет сумму только девяти значений и не учитывает значение NULL. Выражение

```
SUM(SALES) - SUM(QUOTA)
```

вычисляет разницу между ними. В то же время выражение

```
SUM(SALES-QUOTA)
```

принимает в качестве аргументов только девять значений, которые не равны NULL. В строке, где значение планового объема продаж равно NULL, операция вычитания возвращает значение NULL, которое функция SUM() игнорирует. Таким образом, из результатов этого выражения исключаются фактические продажи служащего, для которого еще не установлен план, хотя они вошли в результаты предыдущего выражения.

Какой же ответ является "правильным"? Оба! Первое выражение вычисляет именно то, что и означает, т.е. "сумма по SALES минус сумма по QUOTA". И второе выражение также вычисляет именно то, что оно означает, т.е. "сумма (SALES - QUOTA)". Однако при наличии значений NULL результаты выражений отличаются.

В стандарте ANSI/ISO определены следующие точные правила обработки значений NULL в статистических функциях.

- Если какие-либо из значений, содержащихся в столбце, равны NULL, при вычислении результата функции они игнорируются.
- Если все значения в столбце равны NULL, то функции SUM(), AVG(), MIN() и MAX() возвращают значение NULL; функция COUNT() возвращает нуль.
- Если в столбце нет значений (т.е. столбец пустой), то функции SUM(), AVG(), MIN() и MAX() возвращают значение NULL; функция COUNT() возвращает нуль.
- Функция COUNT(*) подсчитывает количество строк и не зависит от наличия или отсутствия в столбце значений NULL. Если строк в таблице нет, эта функция возвращает нуль.

Хотя стандарт предельно точен в данном вопросе, коммерческие SQL-продукты могут выдавать результаты, отличающиеся от стандарта, особенно если все значения, содержащиеся в столбце, равны NULL или таблица пуста. Прежде чем полагаться на правила, определенные в стандарте, следует протестировать свою СУБД.

Удаление повторяющихся строк (DISTINCT)

Из главы 6, "Простые запросы", вы должны помнить, что ключевое слово DISTINCT указывается в начале списка возвращаемых столбцов и служит для удаления повторяющихся строк из таблицы результатов запроса. С помощью этого предиката можно также указать, что перед применением статистической функции к столбцу из него следует удалить все повторяющиеся значения. Для этого необходимо включить предикат DISTINCT перед аргументом статистической функции сразу же после открывающей круглой скобки.

Ниже приведены два запроса, которые иллюстрируют удаление повторяющихся значений перед применением статистических функций.

Сколько различных должностей существует в нашей компании?

```
SELECT COUNT(DISTINCT TITLE)
  FROM SALESREPS;

COUNT(DISTINCT TITLE)
---------------------
                    3
```

В скольких офисах есть служащие, превысившие плановые объемы продаж?

```
SELECT COUNT(DISTINCT REP_OFFICE)
  FROM SALESREPS
 WHERE SALES > QUOTA;

COUNT(DISTINCT REP_OFFICE)
--------------------------
                         4
```

Ключевое слово DISTINCT в одном запросе можно употребить только один раз. Если оно применяется вместе с аргументом одной из статистических функций, его нельзя использовать ни с каким другим аргументом. Если оно указано перед списком возвращаемых столбцов, его нельзя употреблять ни в одной статистической функции. Единственным исключением из этого правила является случай, когда DISTINCT используется второй раз внутри подзапроса. О подзапросах рассказывается в главе 9, "Подзапросы и выражения с запросами".

Запросы с группировкой (GROUP BY)

Результаты итоговых запросов, о которых до сих пор шла речь в настоящей главе, напоминают итоговую информацию, находящуюся обычно в конце отчета. Эти запросы "сжимают" подробные данные, содержащиеся в отчете, в одну строку итоговых результатов. Но, как известно, в отчетах иногда используются и промежуточные итоги. Точно так же бывает необходимо получать и промежуточные итоги результатов запроса. Эту возможность предоставляет предложение GROUP BY инструкции SELECT.

Назначение предложения GROUP BY проще всего понять на примере. Рассмотрим два запроса.

Какова средняя стоимость заказа?

```
SELECT AVG(AMOUNT)
  FROM ORDERS;

 AVG(AMOUNT)
------------
   $8,256.37
```

Какова средняя стоимость заказа для каждого служащего?

```
SELECT REP, AVG(AMOUNT)
  FROM ORDERS
 GROUP BY REP;

 REP      AVG(AMOUNT)
----      -----------
 101        $8,876.00
 102        $5,694.00
 103        $1,350.00
 105        $7,865.40
 106       $16,479.00
 107       $11,477.33
 108        $8,376.14
 109        $3,552.50
 110       $11,566.00
```

Первый запрос представляет собой простой итоговый запрос, аналогичный рассмотренным ранее примерам. Второй запрос возвращает несколько итоговых строк — по одной строке для каждой группы. На рис. 8.3 изображена схема выполнения второго запроса. Концептуально запрос выполняется следующим образом.

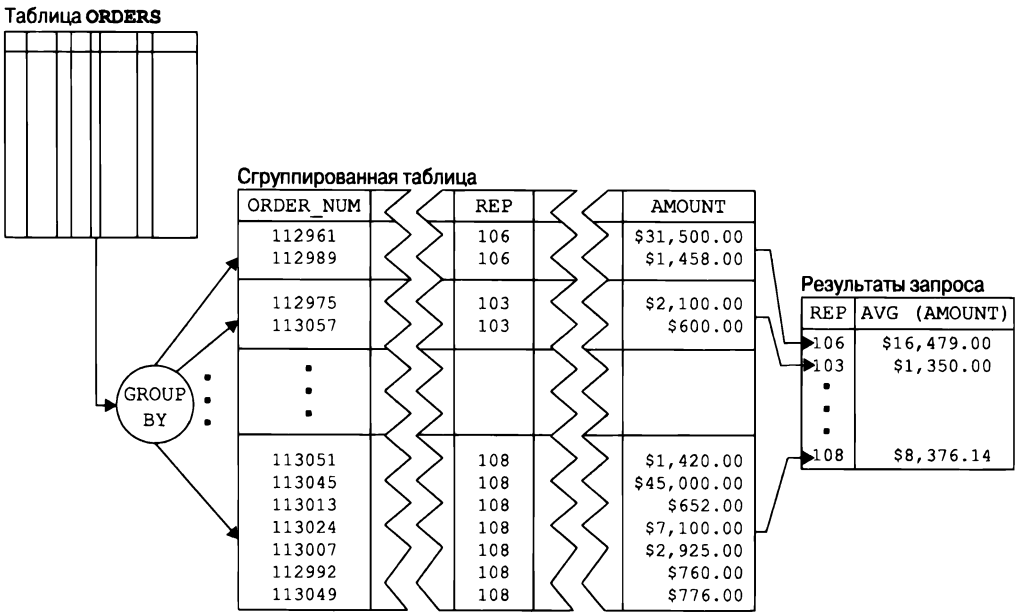


Рис. 8.3. Выполнение запроса с группировкой

1. Заказы делятся на группы, по одной группе для каждого служащего. В каждой группе все заказы имеют одно и то же значение в столбце REP.
2. Для каждой группы вычисляется среднее значение столбца AMOUNT по всем строкам, входящим в группу, и генерируется одна итоговая строка результатов. Эта строка содержит значение столбца REP для группы и среднюю стоимость заказа для данной группы.

Запрос, включающий в себя предложение GROUP BY, называется *запросом с группировкой*, поскольку он объединяет строки исходных таблиц в группы и для каждой группы строк генерирует одну строку таблицы результатов запроса. Столбцы, указанные в предложении GROUP BY, называются *столбцами группировки*, поскольку именно они определяют, по какому признаку строки делятся на группы. Ниже приведен ряд дополнительных примеров запросов с группировкой.

Каков диапазон плановых объемов продаж для каждого офиса?

```
SELECT REP_OFFICE, MIN(QUOTA), MAX(QUOTA)
  FROM SALESREPS
 GROUP BY REP_OFFICE;

REP_OFFICE      MIN(QUOTA)      MAX(QUOTA)
----------     -----------     -----------
      NULL            NULL            NULL
        11     $275,000.00     $300,000.00
        12     $200,000.00     $300,000.00
        13     $350,000.00     $350,000.00
        21     $350,000.00     $350,000.00
        22     $300,000.00     $300,000.00
```

Сколько служащих работает в каждом офисе?

```
SELECT REP_OFFICE, COUNT(*)
  FROM SALESREPS
 GROUP BY REP_OFFICE;

REP_OFFICE     COUNT(*)
----------     --------
      NULL            1
        11            2
        12            3
        13            1
        21            2
        22            1
```

Сколько клиентов обслуживает каждый служащий?

```
SELECT COUNT(DISTINCT CUST_NUM), 'customers for salesrep',
       CUST_REP
  FROM CUSTOMERS
 GROUP BY CUST_REP;

COUNT(DISTINCT CUST_NUM)   CUSTOMERS FOR SALESREP   CUST_REP
------------------------   ----------------------   --------
                       3   customers for salesrep        101
                       4   customers for salesrep        102
                       3   customers for salesrep        103
```

```
1   customers for salesrep        104
2   customers for salesrep        105
2   customers for salesrep        106
.
.
.
```

Между статистическими функциями SQL и предложением GROUP BY существует внутренняя связь. Статистическая функция берет столбец значений и возвращает одно значение. Предложение GROUP BY указывает, что следует разделить результаты запроса на группы, применить статистическую функцию по отдельности к каждой группе и получить для каждой группы одну строку результатов. Ниже показаны шаги выполнения SQL-запроса, расширенные с учетом операции группировки.

1. Если запрос представляет собой объединение (UNION) инструкций SELECT, для каждой из них выполнить шаги 2–7 для генерации отдельных результатов запросов.
2. Сформировать произведение таблиц, указанных в предложении FROM. Если там указана только одна таблица, то произведением будет она сама.
3. При наличии предложения WHERE применить указанное в нем условие к каждой строке таблицы произведения, оставляя только те строки, для которых условие истинно, и отбрасывая строки, для которых условие ложно или равно NULL.
4. Если имеется предложение GROUP BY, разделить строки, оставшиеся в таблице произведения, на группы таким образом, чтобы в каждой группе строки имели одинаковые значения во всех столбцах группировки.
5. Если имеется предложение HAVING, применить указанное в нем условие к каждой строке группы, оставляя только те группы, для которых условие истинно, и отбрасывая группы, для которых условие ложно или равно NULL.
6. Для каждой из оставшихся строк (или для каждой группы строк) вычислить значение каждого элемента в списке возвращаемых столбцов и создать одну строку таблицы результатов запроса. При простой ссылке на столбец берется значение столбца в текущей строке (или группе строк); при использовании статистической функции в качестве ее аргумента при наличии предложения GROUP BY используются значения столбца из всех строк, входящих в группу; в противном случае используется все множество строк.
7. Если указан предикат DISTINCT, удалить из таблицы результатов запроса все повторяющиеся строки.
8. Если запрос является объединением инструкций SELECT, объединить результаты выполнения отдельных инструкций в одну таблицу результатов запроса. Удалить из нее повторяющиеся строки, если только в запросе не указан предикат UNION ALL.
9. Если имеется предложение ORDER BY, отсортировать результаты запроса, как указано в нем.

Строки, генерируемые этой процедурой, составляют результаты запроса.

Несколько столбцов группировки

SQL позволяет группировать результаты запроса на основе двух или более столбцов. Предположим, например, что вам требуется сгруппировать заказы по служащим и клиентам. Эту задачу выполняет приведенный ниже запрос.

Подсчитать общую сумму заказов по каждому клиенту для каждого служащего.

```
SELECT REP, CUST, SUM(AMOUNT)
  FROM ORDERS
 GROUP BY REP, CUST;

REP      CUST      SUM(AMOUNT)
----     ----      ----------
 101     2102       $3,978.00
 101     2108         $150.00
 101     2113      $22,500.00
 102     2106       $4,026.00
 102     2114      $15,000.00
 102     2120       $3,750.00
 103     2111       $2,700.00
 105     2103      $35,582.00
 105     2111       $3,745.00
    .
    .
    .
```

Даже при группировке по нескольким столбцам старые версии SQL обеспечивают только один уровень группировки. Приведенный запрос генерирует одну итоговую строку для каждой пары "служащий/клиент". Чтобы получить несколько уровней промежуточных итогов в более современном SQL, можно воспользоваться операторами WITH ROLLUP и WITH CUBE в комбинации с оператором GROUP BY. Оператор WITH ROLLUP заставляет операцию группировки выводить промежуточные итоги для каждого уровня группировки, работая слева направо по всему списку столбцов группировки. WITH CUBE идет дальше и показывает итоговые результаты для каждой возможной комбинации столбцов группировки. WITH CUBE и WITH ROLLUP, кроме того, подводят общий итог по всему результирующему множеству, но эти значения не всегда выводятся в конце результирующего множества. Так, в случае Oracle, например, итоговые значения выводятся первыми. Опознать строки промежуточных итогов можно по наличию значений NULL в столбцах, не участвующих в группировке. Аналогично, в случае итогового значения по всему множеству, NULL будет во всех столбцах группировки.

Подсчитать общую сумму заказов по каждому клиенту для каждого служащего с промежуточными итогами для каждого служащего.

```
SELECT REP, CUST, SUM(AMOUNT)
  FROM ORDERS
 GROUP BY REP, CUST WITH ROLLUP;

       REP       CUST SUM(AMOUNT)
---------- ---------- -----------
       101       2102        3978
       101       2108         150
       101       2113       22500
       101                  26628
```

```
       102        2106         4026
       102        2114        15000
       102        2120         3750
       102                    22776
       103        2111         2700
       103                     2700
       105        2103        35582
       105        2111         3745
       105                    39327
 .
 .
 .
       110        2107        23132
       110                    23132
                             247691
```

Подсчитать общую сумму заказов по каждому клиенту для каждого служащего с промежуточными итогами для каждого служащего и каждого клиента.

```
SELECT REP, CUST, SUM(AMOUNT)
  FROM ORDERS
 GROUP BY REP, CUST WITH CUBE;

       REP        CUST SUM(AMOUNT)
---------- ---------- -----------
       101                    26628
       101        2102         3978
       101        2108          150
       101        2113        22500
       102                    22776
       102        2106         4026
       102        2114        15000
       102        2120         3750
       103                     2700
       103        2111         2700
       105                    39327
       105        2103        35582
       105        2111         3745
 .
 .
 .
                             247691
                  2101         1458
                  2102         3978
                  2103        35582
                  2106         4026
                  2107        23132
                  2108         7255
                  2109        31350
                  2111         6445
                  2112        47925
                  2113        22500
                  2114        22100
                  2117        31500
                  2118         3608
                  2120         3750
                  2124         3082
```

Синтаксис WITH ROLLUP и WITH CUBE немного варьируется от одной СУБД к другой, так что обратитесь к документации от вашего производителя. В приведенных примерах показан стандартный синтаксис SQL, который поддерживается SQL Server, DB2 Universal Database (UDB) и MySQL (однако в версии MySQL 5.1 WITH CUBE еще не поддерживался). Oracle требует ключевые слова GROUP BY ROLLUP или GROUP BY CUBE, за которыми следует список столбцов группировки, который должен быть заключен в скобки, например GROUP BY CUBE (REP, CUST). DB2 UDB поддерживает как стандартный синтаксис, так и вариант Oracle.

Если ваша реализация SQL не поддерживает ROLLUP или CUBE, то лучшее, что вы можете сделать — это отсортировать данные так, чтобы строки в результатах запроса находились в должном порядке. Во многих реализациях SQL предложение GROUP BY в качестве побочного действия автоматически сортирует данные, но эту сортировку можно перекрыть предложением ORDER BY.

Подсчитать общую сумму заказов по каждому клиенту для каждого служащего; отсортировать результаты запроса по клиентам, а для каждого клиента — по служащим.

```
SELECT CUST, REP, SUM(AMOUNT)
  FROM ORDERS
 GROUP BY CUST, REP
 ORDER BY CUST, REP;

 CUST     REP     SUM(AMOUNT)
 ----     ---     ----------
 2101     106      $1,458.00
 2102     101      $3,978.00
 2103     105     $35,582.00
 2106     102      $4,026.00
 2107     110     $23,132.00
 2108     101        $150.00
 2108     109      $7,105.00
 2109     107     $31,350.00
 2111     103      $2,700.00
 2111     105      $3,745.00
 .
 .
 .
```

Ограничения на запросы с группировкой

На запросы, в которых используется группировка, накладываются строгие ограничения. Столбцы группировки должны представлять собой реальные столбцы таблиц, перечисленных в предложении FROM. Однако некоторые реализации SQL допускают выражения со столбцами и даже позволяют выполнять группировку по выражениям путем повторения выражения в предложении GROUP BY.

Кроме того, существуют ограничения на элементы списка возвращаемых столбцов. Все элементы этого списка должны иметь одно значение для каждой группы строк. Это означает, что таким столбцом в запросе с группировкой может быть:

- столбец (при условии, что это один из столбцов группировки);
- константа;

- статистическая функция, возвращающая одно значение для всех строк группы;
- столбец группировки, который по определению имеет одно и то же значение во всех строках группы;
- выражение, включающее комбинацию перечисленных элементов.

На практике в список возвращаемых столбцов запроса с группировкой всегда входят столбец группировки *и* статистическая функция. Если последняя не указана, значит, запрос можно более просто выразить с помощью инструкции SELECT DISTINCT, без использования GROUP BY. И наоборот, если не включить в результаты запроса столбец группировки, вы не сможете определить, к какой группе относится каждая строка результатов!

Еще одно ограничение запросов с группировкой обусловлено тем, что в SQL игнорируется информация о первичных и внешних ключах при анализе корректности запроса с группировкой. Рассмотрим следующий запрос.

Подсчитать общую сумму заказов каждого служащего.

```
SELECT EMPL_NUM, NAME, SUM(AMOUNT)
  FROM ORDERS, SALESREPS
 WHERE REP = EMPL_NUM
 GROUP BY EMPL_NUM

Error: "NAME" not a GROUP BY expression
```

Исходя из природы данных, запрос имеет смысл, поскольку группировка по идентификатору служащего — фактически то же самое, что и группировка по имени служащего. Говоря более точно, столбец группировки EMPL_NUM является первичным ключом таблицы SALESREPS, поэтому столбец NAME должен иметь одно значение для каждой группы. Тем не менее SQL выдает сообщение об ошибке, поскольку столбец NAME не указан явно в качестве столбца группировки (само сообщение об ошибке может меняться от одной СУБД к другой). Чтобы решить эту проблему, необходимо просто включить столбец NAME в предложение GROUP BY в качестве второго (избыточного) столбца группировки.

Подсчитать общее количество заказов для каждого служащего.

```
SELECT EMPL_NUM, NAME, SUM(AMOUNT)
  FROM ORDERS, SALESREPS
 WHERE REP = EMPL_NUM
 GROUP BY EMPL_NUM, NAME;

EMPL_NUM  NAME          SUM(AMOUNT)
--------  ------------- -----------
     101  Dan Roberts    $26,628.00
     102  Sue Smith      $22,776.00
     103  Paul Cruz       $2,700.00
     105  Bill Adams     $39,327.00
     106  Sam Clark      $32,958.00
     107  Nancy Angelli  $34,432.00
     108  Larry Fitch    $58,633.00
     109  Mary Jones      $7,105.00
     110  Tom Snyder     $23,132.00
```

Конечно, если идентификатор служащего в результатах запроса не требуется, его можно просто исключить из списка возвращаемых столбцов.

Подсчитать общее количество заказов для каждого служащего.

```
SELECT NAME, SUM(AMOUNT)
  FROM ORDERS, SALESREPS
 WHERE REP = EMPL_NUM
 GROUP BY NAME;

NAME           SUM(AMOUNT)
------------- -----------
Bill Adams      $39,327.00
Dan Roberts     $26,628.00
Larry Fitch     $58,633.00
Mary Jones       $7,105.00
Nancy Angelli   $34,432.00
Paul Cruz        $2,700.00
Sam Clark       $32,958.00
Sue Smith       $22,776.00
Tom Snyder      $23,132.00
```

Значения NULL в столбцах группировки

Когда в столбце группировки содержится значение NULL, возникают дополнительные трудности. Если значение столбца неизвестно, к какой группе его следует отнести? В предложении WHERE при сравнении двух значений NULL результат имеет значение NULL (а не TRUE), т.е. два значения NULL *не* считаются одинаковыми. Если такое соглашение применить в предложении GROUP BY, это приведет к тому, что каждая строка со значением NULL в столбце группировки будет помещена в отдельную группу, состоящую из одной этой строки.

На практике это правило очень неудобно. Поэтому в стандарте ANSI/ISO определено, что два значения NULL в предложении GROUP BY равны. Если две строки имеют значение NULL в одинаковых столбцах группировки и идентичные значения во всех остальных столбцах группировки, они помещаются в одну группу. Табл. 8.1 иллюстрирует принцип обработки значений NULL предложением GROUP BY в соответствии со стандартом ANSI/ISO, как показано в следующем запросе.

```
SELECT HAIR, EYES, COUNT(*)
  FROM PEOPLE
 GROUP BY HAIR, EYES;

HAIR      EYES     COUNT(*)
------    -----    --------
Brown     Blue            1
NULL      Blue            2
NULL      NULL            2
Brown     NULL            4
Brown     Brown           1
Blonde    Blue            2
```

Таблица 8.1. Таблица PEOPLE

NAME	HAIR	EYES
Cincly	Brown	Blue
Louise	NULL	Blue
Harry	NULL	Blue
Samantha	NULL	NULL
Joanne	NULL	NULL
George	Brown	NULL
Mary	Brown	NULL
Paula	Brown	NULL
Kevin	Brown	NULL
Joel	Brown	Brown
Susan	Blonde	Blue
Marie	Blonde	Blue

Хотя такой принцип обработки значений NULL и определен в стандарте SQL, он реализован не во всех диалектах SQL. Прежде чем рассчитывать на такое поведение вашей СУБД, ее следует протестировать.

Условия отбора групп (HAVING)

Точно так же, как предложение WHERE используется для отбора отдельных строк, участвующих в запросе, предложение HAVING можно применить для отбора групп строк. Его формат соответствует формату предложения WHERE. Предложение HAVING состоит из ключевого слова HAVING, за которым следует условие отбора. Таким образом, данное предложение определяет условие отбора групп строк.

Следующий пример прекрасно иллюстрирует роль предложения HAVING.

Какова средняя стоимость заказа для каждого служащего из числа тех, у которых общая стоимость заказов превышает $30 000?

```
SELECT REP, AVG(AMOUNT)
  FROM ORDERS
 GROUP BY REP
HAVING SUM(AMOUNT) > 30000.00;

REP      AVG(AMOUNT)
---      -----------
105        $7,865.40
106       $16,479.00
107       $11,477.33
108        $8,376.14
```

На рис. 8.4 изображена схема выполнения этого запроса. Вначале предложение GROUP BY разделяет заказы на группы по служащим. После этого предложение

HAVING исключает все группы, в которых общая стоимость заказа не превышает $30000. И наконец, предложение SELECT вычисляет среднюю стоимость заказа для каждой из оставшихся групп и генерирует таблицу результатов запроса.

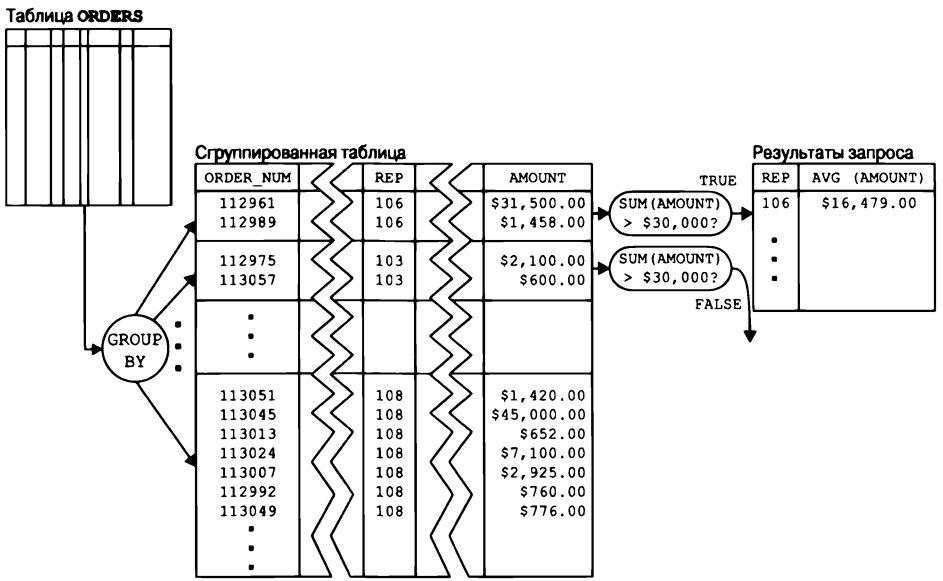


Рис. 8.4. Условие отбора групп в действии

В предложении HAVING указываются точно такие же условия отбора, как и в предложении WHERE; описаны в главах 6, "Простые запросы", и 9, "Подзапросы и выражения с запросами". Ниже приведен еще один пример использования условия для отбора групп.

Для каждого офиса, в котором работают два или более человек, вычислить общий плановый и фактический объемы продаж для всех служащих.

```
SELECT CITY, SUM(QUOTA), SUM(SALESREPS.SALES)
  FROM OFFICES, SALESREPS
 WHERE OFFICE = REP_OFFICE
 GROUP BY CITY
HAVING COUNT(*) >= 2;

CITY          SUM(QUOTA)     SUM(SALESREPS.SALES)
-----------  -----------    ---------------------
Chicago      $775,000.00              $735,042.00
Los Angeles  $700,000.00              $835,915.00
New York     $575,000.00              $692,637.00
```

Ниже показаны шаги выполнения SQL-запроса, расширенные с учетом условия отбора групп.

1. Если запрос представляет собой объединение (UNION) инструкций SELECT, для каждой из них выполнить шаги 2–7 для генерации отдельных результатов запросов.

2. Сформировать произведение таблиц, указанных в предложении FROM. Если там указана только одна таблица, то произведением будет она сама.
3. При наличии предложения WHERE применить указанное в нем условие к каждой строке таблицы произведения, оставляя только те строки, для которых условие истинно, и отбрасывая строки, для которых условие ложно или равно NULL.
4. При наличии предложения GROUP BY разделить строки, оставшиеся в таблице произведения, на группы таким образом, чтобы в каждой группе строки имели одинаковые значения во всех столбцах группировки.
5. Если имеется предложение HAVING, применить указанное в нем условие к каждой строке группы, оставляя только те группы, для которых условие истинно, и отбрасывая группы, для которых условие ложно или равно NULL.
6. Для каждой из оставшихся строк (или для каждой группы строк) вычислить значение каждого элемента в списке возвращаемых столбцов и создать одну строку таблицы результатов запроса. При простой ссылке на столбец берется значение столбца в текущей строке (или группе строк); при использовании статистической функции в качестве ее аргумента при наличии предложения GROUP BY используются значения столбца из всех строк, входящих в группу; в противном случае используется все множество строк.
7. Если указан предикат DISTINCT, удалить из таблицы результатов запроса все повторяющиеся строки.
8. Если запрос является объединением инструкций SELECT, объединить результаты выполнения отдельных инструкций в одну таблицу результатов запроса. Удалить из нее повторяющиеся строки, если только в запросе не указан предикат UNION ALL.
9. Если имеется предложение ORDER BY, отсортировать результаты запроса, как указано в нем.

Строки, генерируемые этой процедурой, составляют результаты запроса.

В соответствии с описанной процедурой, СУБД выполняет приведенный выше запрос таким образом.

1. Объединяет таблицы OFFICES и SALESREPS, чтобы определить город, в котором работает каждый служащий.
2. Группирует строки объединенной таблицы по офисам.
3. Исключает группы, содержащие менее двух строк, — это те группы, которые не удовлетворяют критерию предложения HAVING.
4. Вычисляет общие плановые и фактические объемы продаж для каждой группы.

Вот еще один пример, в котором используются все предложения инструкции SELECT.

Показать цену, количество на складе и общее количество заказанных единиц для каждого наименования товара, если для него общее количество заказанных единиц превышает 75 процентов от количества товара на складе.

```
SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
  FROM PRODUCTS, ORDERS
 WHERE MFR = MFR_ID
   AND PRODUCT = PRODUCT_ID
 GROUP BY MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND
HAVING SUM(QTY) > (.75 * QTY_ON_HAND)
 ORDER BY QTY_ON_HAND DESC;

DESCRIPTION          PRICE   QTY_ON_HAND     SUM(QTY)
--------------- ---------   -----------     --------
Reducer            $355.00            38           32
Widget Adjuster     $25.00            37           30
Motor Mount        $243.00            15           16
Right Hinge      $4,500.00            12           15
500-lb Brace     $1,425.00             5           22
```

Чтобы осуществить этот запрос, СУБД выполняет следующие действия.

1. Объединяет таблицы ORDERS и PRODUCTS, чтобы получить описание, цену и количество единиц на складе для каждого заказанного товара.
2. Группирует строки объединенной таблицы по идентификаторам производителя и товара.
3. Исключает группы, в которых количество заказанных единиц составляет менее 75 процентов от количества на складе.
4. Вычисляет общее количество заказанных единиц для каждой группы.
5. Генерирует одну итоговую строку запроса для каждой группы.
6. Сортирует результаты запроса таким образом, чтобы товары, которых на складе больше, были представлены первыми.

Как было сказано ранее, столбцы DESCRIPTION, PRICE и QTY_ON_HAND должны быть указаны в качестве столбцов группировки, поскольку они перечислены в списке возвращаемых столбцов. Однако на деле они не участвуют в процессе группировки, поскольку столбцы MFR_ID и PRODUCT_ID полностью определяют строку таблицы PRODUCTS, и три оставшихся столбца автоматически имеют в группе одно значение.

Ограничения на условия отбора групп

Предложение HAVING используется для того, чтобы включать и исключать группы строк из результатов запроса, поэтому используемое в нем условие отбора должно применяться не к отдельным строкам, а к группе в целом. Это значит, что элементом условия отбора может быть:

- константа;
- статистическая функция, возвращающая одно значение для всех строк, входящих в группу;

- столбец группировки, который по определению имеет одно и то же значение во всех строках группы;
- выражение, включающее в себя перечисленные выше элементы.

На практике условие отбора предложения HAVING всегда должно включать в себя как минимум одну статистическую функцию. Если это не так, значит, условие отбора можно переместить в предложение WHERE и применить к отдельным строкам. Простейший способ выяснить, должно ли условие отбора находиться в предложении WHERE или HAVING, — вспомнить, как применяются эти предложения:

- предложение WHERE применяется к *отдельным строкам*, поэтому выражения, содержащиеся в нем, должны быть вычислимы для отдельных строк;
- предложение HAVING применяется к *группам строк*, поэтому выражения, содержащиеся в нем, должны быть вычислимы для групп строк.

Значения NULL и условия отбора групп

Как и условие отбора в предложении WHERE, условие отбора в предложении HAVING может дать один из трех результатов.

- Если условие отбора имеет значение TRUE, группа строк остается и для нее генерируется одна строка результатов запроса.
- Если условие отбора имеет значение FALSE, группа строк исключается и строка для нее в результатах запроса не генерируется.
- Если условие отбора имеет значение NULL, группа строк исключается и строка для нее в результатах запроса не генерируется.

Правила обработки значений NULL в условиях отбора для предложения HAVING в точности те же, что и для предложения WHERE, описаны в главе 6, "Простые запросы".

Предложение HAVING без GROUP BY

Предложение HAVING почти всегда используется в сочетании с предложением GROUP BY, однако синтаксис инструкции SELECT не требует этого. Если предложение HAVING используется без предложения GROUP BY, СУБД рассматривает полные результаты запроса как одну группу. Другими словами, статистические функции, содержащиеся в предложении HAVING, для определения того, будет ли группа включена в результаты запроса или нет, применяются к одной и только одной группе, и эта группа состоит из всех строк результата запроса. На практике предложение HAVING очень редко используется без соответствующего предложения GROUP BY.

Резюме

В настоящей главе были рассмотрены итоговые запросы.

- В итоговых запросах используются статистические функции, которые позволяют на основании данных из столбца получить одно итоговое значение.
- Статистические функции могут вычислять среднее значение, сумму, минимальное и максимальное значение в столбце, подсчитать количество значений в столбце или количество строк в результатах запроса.
- Итоговый запрос без предложения GROUP BY генерирует одну строку результатов запроса на основании всех строк таблицы или произведения таблиц.
- Итоговый запрос с предложением GROUP BY генерирует несколько строк результатов запроса, каждая из которых является итоговой для определенной группы.
- Предложение HAVING является аналогом предложения WHERE для групп и позволяет по определенному критерию отбирать группы строк, составляющие результаты запроса.

ГЛАВА 9

Подзапросы и выражения с запросами

Подчиненные запросы, или подзапросы SQL, позволяют использовать один запрос как часть другого. Возможность применения одного запроса внутри другого и была причиной появления слова "структурированный" в названии "язык структурированных запросов". Понятие подзапроса не так широко известно, как понятие соединения, но оно играет важную роль в SQL.

- Инструкция SQL с подзапросом зачастую является самым естественным способом выражения запроса, так как она лучше всего соответствует словесному описанию запроса.
- Подзапросы облегчают написание инструкции `SELECT`, поскольку они позволяют разбивать запрос на части (на запрос и подзапросы), а затем объединять эти части.
- Существуют запросы, которые нельзя сформулировать на SQL, не прибегая к помощи подзапросов.

В первых разделах данной главы рассматриваются подзапросы и их использование в предложениях `WHERE` и `HAVING` инструкций SQL. В остальной части главы описываются расширенные возможности составления запросов, добавленные в стандарт SQL, существенно увеличивающие мощь языка и позволяющие выполнять даже самые сложные операции над базами данных.

Применение подзапросов

Подзапросом называется запрос внутри другого запроса SQL. Результаты подзапроса используются СУБД для определения результатов запроса более высокого уровня, содержащего данный подзапрос. В простейшем случае подзапрос нахо-

дится в предложении WHERE или HAVING другой SQL-инструкции. Подзапросы обеспечивают эффективный естественный способ обработки запросов, которые выражаются через результаты других запросов. Вот пример такого запроса.

Вывести список офисов, в которых плановый объем продаж превышает сумму плановых объемов продаж всех служащих.

В данном запросе требуется получить список офисов из таблицы OFFICES, для которых значение столбца TARGET удовлетворяет некоторому условию. Логично предположить, что инструкция SELECT, выражающая данный запрос, должна выглядеть примерно так, как показано ниже.

```
SELECT CITY    FROM OFFICES    WHERE TARGET > ???
```

Здесь величина ??? равна сумме плановых объемов продаж всех служащих, работающих в данном офисе. Как можно указать ее в данном запросе? Из главы 8, "Итоговые запросы", вам уже известно, что сумму плановых объемов продаж для отдельного офиса (скажем, офиса с идентификатором 21) можно получить с помощью следующего запроса.

```
SELECT SUM(QUOTA)
  FROM SALESREPS
 WHERE REP_OFFICE = 21;
```

Однако ввести такой запрос, записать результаты, а потом ввести предыдущий запрос с корректным значением — не самый эффективный способ работы. Так как вставить результаты этого запроса в предыдущий запрос вместо вопросительных знаков? По-видимому, было бы разумно вначале написать первый запрос, а затем заменить ??? вторым запросом.

```
SELECT CITY
  FROM OFFICES
 WHERE TARGET > (SELECT SUM(QUOTA)
                   FROM SALESREPS
                  WHERE REP_OFFICE = OFFICE)
```

Фактически это корректный SQL-запрос. *Внутренний* запрос (*подзапрос*) вычисляет для каждого офиса сумму плановых объемов продаж всех служащих, работающих в данном офисе. *Главный* (*внешний*) запрос сравнивает план продаж офиса с полученной суммой и, в зависимости от результата сравнения, либо добавляет данный офис в таблицу результатов запроса, либо нет. Вместе главный и подчиненный запросы выражают исходный запрос и извлекают из базы данных требуемую информацию.

Подчиненные SQL-запросы обычно выступают в качестве части предложения WHERE или HAVING. В предложении WHERE они помогают отбирать из таблицы результатов запроса отдельные строки, а в предложении HAVING — группы строк.

Что такое подзапрос

На рис. 9.1 изображена структура подзапроса SQL. Подзапрос всегда заключается в круглые скобки, сохраняя при этом знакомую структуру инструкции SELECT, содержащей предложение FROM и необязательные предложения WHERE,

GROUP BY и HAVING. Структура этих предложений в подзапросе идентична их структуре в инструкции SELECT; в подзапросе эти предложения выполняют свои обычные функции. Однако между подзапросом и инструкцией SELECT имеется ряд отличий.

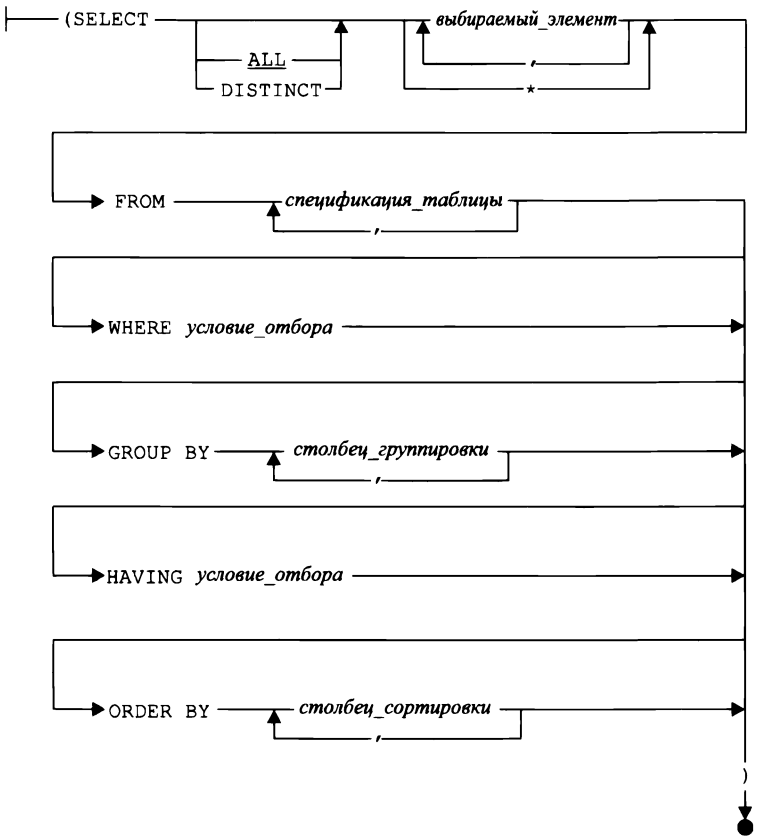


Рис. 9.1. Синтаксическая диаграмма подзапроса

- В большинстве случаев результаты подзапроса всегда состоят из одного столбца. Это означает, что в предложении SELECT подзапроса почти всегда указывается только один элемент списка выбора.
- Хотя в подзапросе может находиться предложение ORDER BY, на самом деле оно используется крайне редко. Результаты подзапроса используются только внутри главного запроса и для пользователя остаются невидимыми, поэтому нет смысла их сортировать. Более того, сортировка большого количества данных может отрицательно сказаться на производительности.
- Имена столбцов в подзапросе могут ссылаться на столбцы таблиц главного запроса. Эти внешние ссылки подробно рассматриваются ниже в данной главе.

- В большинстве реализаций SQL подзапрос не может быть объединением (UNION) нескольких различных инструкций SELECT; допускается использование только одной инструкции SELECT. (Стандарт SQL ослабляет это ограничение, позволяя, как будет показано ниже, создавать гораздо более мощные запросы.)

Подзапросы в предложении WHERE

Чаще всего подзапросы указываются в предложении WHERE инструкции SQL. Когда подзапрос содержится в данном предложении, он участвует в процессе отбора строк. В простейшем случае подзапрос является частью условия отбора и возвращает значение, используемое при проверке истинности или ложности условия. Рассмотрим пример.

Вывести список служащих, чей плановый объем продаж составляет менее 10% от планового объема продаж всей компании.

```
SELECT NAME
  FROM SALESREPS
 WHERE QUOTA < (.1 * (SELECT SUM(TARGET)
                        FROM OFFICES));

NAME
---------
Bob Smith
```

В данном случае подзапрос вычисляет сумму плановых объемов продаж всех офисов, которая затем умножается на 0.1 (10%). Полученное значение используется в условии отбора при сканировании таблицы SALESREPS на предмет поиска нужных строк. В данном простом случае подзапрос возвращает для каждой строки таблицы SALESREPS одно и то же значение. Конечно, запрос можно записать и по-другому, так чтобы умножение выполнялось в подзапросе.

```
SELECT NAME
  FROM SALESREPS
 WHERE QUOTA < (SELECT (SUM(TARGET) * .1) FROM OFFICES);
```

В этом случае более удобным оказалось воспользоваться подзапросом, но это не обязательно. Можно было бы просто выполнить подзапрос как отдельный запрос и получить сумму плановых объемов продаж ($275000 в нашей учебной базе данных), а затем применить полученное значение в предложении WHERE главного запроса.

```
SELECT (SUM(TARGET) * .1)
  FROM OFFICES;

 (SUM(TARGET) * .1)
------------------
            275000


SELECT NAME
  FROM SALESREPS
 WHERE QUOTA < 275000;
```

Однако обычно подзапросы не столь просты. Рассмотрим еще раз запрос из предыдущего раздела.

Вывести список офисов, в которых плановый объем продаж превышает сумму плановых объемов продаж всех служащих.

```
SELECT CITY
  FROM OFFICES
 WHERE TARGET > (SELECT SUM(QUOTA)
                   FROM SALESREPS
                  WHERE REP_OFFICE = OFFICE);

CITY
-----------
Chicago
Los Angeles
```

В этом (более типичном) случае результаты подзапроса нельзя вычислить один раз, так как он возвращает *различные* результаты для каждого конкретного офиса. На рис. 9.2 изображена схема выполнения этого запроса. Главный запрос извлекает данные из таблицы OFFICES, а его предложение WHERE отбирает офисы, которые будут включены в таблицу результатов запроса. Условие, заданное в этом предложении, поочередно применяется ко всем строкам таблицы OFFICES. Предложение WHERE сравнивает значение текущей строки в столбце TARGET со значением, которое возвращается подзапросом для служащих "текущего" офиса. Результатом подзапроса является одно число, и предложение WHERE сравнивает его со значением столбца TARGET, выбирая или отбрасывая текущий офис на основании результата сравнения. Как видно из рисунка, выполнение подзапроса повторяется для каждой строки, проверяемой предложением WHERE главного запроса.

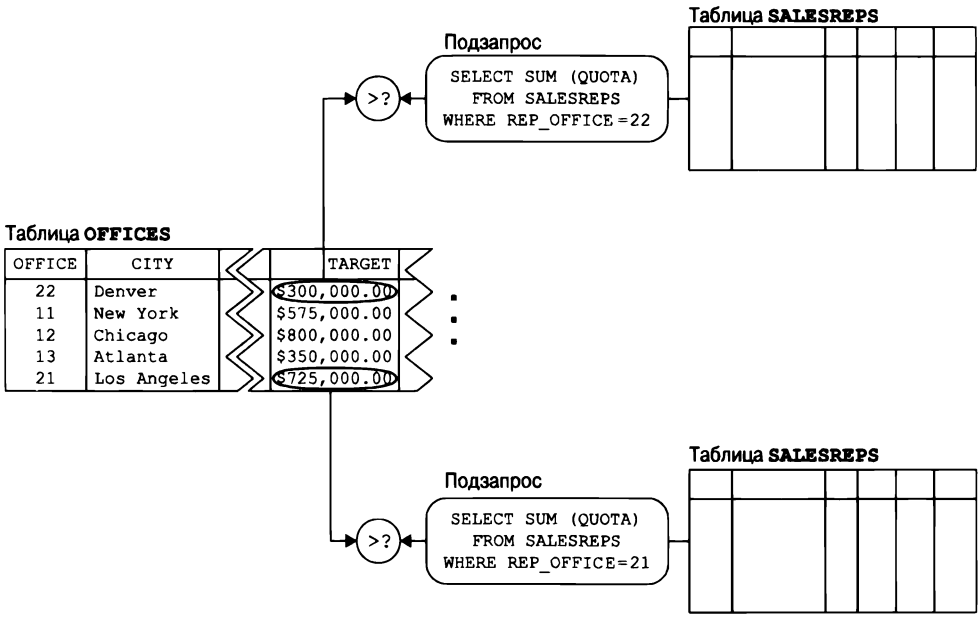


Рис. 9.2. Выполнение подзапроса в предложении WHERE

Внешние ссылки

Часто в теле подзапроса требуется сослаться на значение столбца в текущей строке главного запроса. Рассмотрим еще раз запрос из предыдущих разделов.

Вывести список офисов, в которых плановый объем продаж офиса превышает сумму плановых объемов продаж всех служащих.

```
SELECT CITY
  FROM OFFICES
 WHERE TARGET > (SELECT SUM(QUOTA)
                   FROM SALESREPS
                  WHERE REP_OFFICE = OFFICE);
```

Роль подзапроса в приведенной инструкции SELECT заключается в вычислении суммы плановых объемов продаж для служащих, работающих в конкретном офисе, а именно — в том, который в данный момент проверяется предложением WHERE главного запроса. Подзапрос выполняет вычисления путем сканирования таблицы SALESREPS. Но обратите внимание: столбец OFFICE в предложении WHERE подзапроса является столбцом не таблицы SALESREPS, а таблицы OFFICES, которая входит в главный запрос. Во время последовательной проверки строк таблицы OFFICES значение столбца OFFICE в текущей строке этой таблицы используется для выполнения подзапроса.

Столбец OFFICE в подзапросе является примером *внешней ссылки*. Внешняя ссылка представляет собой имя столбца, не входящего ни в одну из таблиц, перечисленных в предложении FROM подзапроса, и принадлежащего таблице, указанной в предложении FROM главного запроса. Как показывает предыдущий пример, значение в столбце внешней ссылки берется из строки, проверяемой в настоящий момент главным запросом.

Условия отбора в подзапросе

Подзапрос всегда является частью условия отбора в предложении WHERE или HAVING. В главе 6, "Простые запросы", были рассмотрены простые условия отбора, которые могут использоваться в этих предложениях. Кроме того, в SQL используются следующие *условия отбора в подзапросе*.

- *Сравнение с результатом подзапроса*. Значение выражения сравнивается с одним значением, которое возвращается подзапросом. Эта проверка напоминает простое сравнение.
- *Проверка на принадлежность результатам подзапроса*. Значение выражения проверяется на равенство одному из множества значений, которые возвращаются подзапросом. Эта проверка напоминает простую проверку на членство в множестве.
- *Проверка на существование*. Проверяется наличие строк в таблице результатов подзапроса.
- *Многократное сравнение*. Значение выражения сравнивается с каждым из множеств значений, которые возвращаются подзапросом.

Сравнение с результатом подзапроса (=, <>, <, <=, >, >=)

Как видно из рис. 9.3, сравнение с результатом подзапроса является модифицированной формой простого сравнения. Значение выражения сравнивается со значением, которое возвращается подзапросом, и если условие сравнения выполняется, то проверка дает результат TRUE. Эта проверка используется для сравнения значения из проверяемой строки с *одним* значением, полученным от подзапроса, как показано в следующем примере.

Вывести список служащих, у которых плановый объем продаж не меньше планового объема продаж офиса в Атланте.

```
SELECT NAME
  FROM SALESREPS
 WHERE QUOTA >= (SELECT TARGET
                   FROM OFFICES
                  WHERE CITY = 'Atlanta');

NAME
-----------
Bill Adams
Sue Smith
Larry Fitch
```

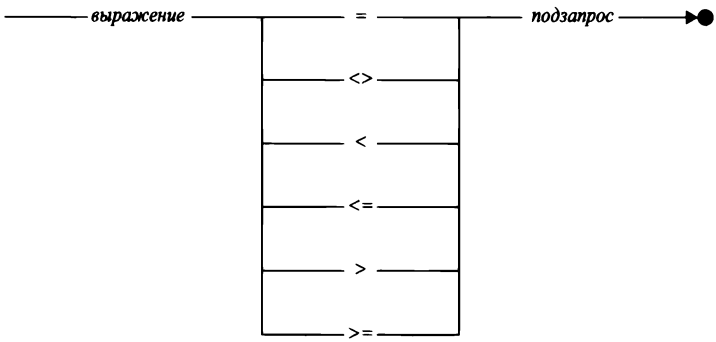


Рис. 9.3. Синтаксическая диаграмма сравнения с результатом подзапроса

В этом примере подзапрос получает плановый объем продаж для офиса в Атланте. Затем полученное значение используется для отбора тех служащих, у которых плановый объем продаж превышает план этого офиса.

В операции сравнения с результатом подзапроса можно использовать те же шесть операторов сравнения (=, <>, <, <=, >, >=), что и при простом сравнении. Подзапрос, участвующий в операции сравнения, должен возвращать в качестве результата *единичное значение*, т.е. только одну строку, содержащую единственный столбец. Если результатом подзапроса являются несколько строк или несколько столбцов, то сравнение не имеет смысла, и SQL выдаст сообщение об ошибке. Если в результате выполнения подзапроса не будет получено ни одной строки или будет получено значение NULL, то операция сравнения возвращает NULL.

Вот еще несколько примеров сравнения с результатом подзапроса.

Вывести список клиентов, которых обслуживает Билл Адамс.

```
SELECT COMPANY
  FROM CUSTOMERS
 WHERE CUST_REP = (SELECT EMPL_NUM
                     FROM SALESREPS
                    WHERE NAME = 'Bill Adams');

COMPANY
---------------
Acme Mfg.
Three-Way Lines
```

Вывести список имеющихся в наличии товаров от компании ACI, количество которых превышает количество товара ACI-41004.

```
SELECT DESCRIPTION, QTY_ON_HAND
  FROM PRODUCTS
 WHERE MFR_ID = 'ACI'
   AND QTY_ON_HAND > (SELECT QTY_ON_HAND
                        FROM PRODUCTS
                       WHERE MFR_ID = 'ACI'
                         AND PRODUCT_ID = '41004');

DESCRIPTION     QTY_ON_HAND
-------------   -----------
Size 3 Widget           207
Size 1 Widget           277
Size 2 Widget           167
```

Сравнение с результатом подзапроса, соответствующее исходному стандарту SQL1 и поддерживаемое всеми ведущими СУБД, допускает наличие подзапроса только в правой части сравнения. Так, неравенство

```
A < (подзапрос)
```

разрешается, а неравенство

```
(подзапрос) > A
```

недопустимо. Это не ограничивает возможности операции сравнения, поскольку знак любого неравенства всегда можно "перевернуть" так, чтобы подзапрос оказался с правой стороны. Однако это говорит о том, что иногда требуется "переворачивать" логику словесного запроса так, чтобы он формально соответствовал разрешенной инструкции SQL.

Последующие версии стандарта устраняют подобное ограничение и позволяют подзапросу находиться с любой стороны от оператора сравнения. Более того, стандарт идет еще дальше и допускает выполнять сравнение с целой строкой значений, а не только с единичным значением. Эти и другие возможности составления сложных запросов описываются ниже в настоящей главе. Однако следует учитывать, что подобные запросы не поддерживаются одинаково всеми текущими версиями ведущих СУБД. С точки зрения переносимости, лучше оставаться в рамках описанных ранее ограничений, накладываемых стандартом SQL1.

Проверка на принадлежность результатам подзапроса (IN)

Как показано на рис. 9.4, проверка на принадлежность результатам подзапроса (предикат IN) является видоизмененной формой простой проверки членства в множестве. При этом одно значение сравнивается со столбцом данных, которые возвращаются подзапросом, и если это значение равно одному из элементов столбца, проверка дает результат TRUE. Данная проверка используется, когда необходимо сравнить значение из проверяемой строки с *множеством* значений, отобранных подзапросом. Вот простой пример.

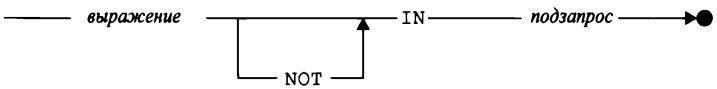


Рис. 9.4. Синтаксическая диаграмма проверки на принадлежность результатам подзапроса

Вывести список служащих тех офисов, где фактический объем продаж превышает плановый.

```
SELECT NAME
  FROM SALESREPS
 WHERE REP_OFFICE IN (SELECT OFFICE
                        FROM OFFICES
                       WHERE SALES > TARGET);

NAME
-----------
Mary Jones
Sam Clark
Bill Adams
Sue Smith
Larry Fitch
```

Подзапрос возвращает список идентификаторов офисов, где фактический объем продаж превышает плановый (в учебной базе данных есть три таких офиса — с номерами 11, 13 и 21). Главный запрос затем проверяет каждую строку таблицы SALESREPS, чтобы определить, работает ли соответствующий служащий в одном из отобранных офисов. Ниже приведено еще пару подобных примеров.

Вывести список служащих, не работающих в офисах, которыми руководит Ларри Фитч (служащий с идентификатором 108).

```
SELECT NAME
  FROM SALESREPS
 WHERE REP_OFFICE NOT IN (SELECT OFFICE
                            FROM OFFICES
                           WHERE MGR = 108);

NAME
-----------
Bill Adams
Mary Jones
Sam Clark
```

```
Bob Smith
Dan Roberts
Paul Cruz
```

Вывести список всех клиентов, заказавших изделия компании ACI (производитель ACI, идентификаторы товаров начинаются с 4100) в период между январем и июнем 2008 года.

```
SELECT COMPANY
  FROM CUSTOMERS
 WHERE CUST_NUM IN
         (SELECT DISTINCT CUST
            FROM ORDERS
           WHERE MFR = 'ACI'
             AND PRODUCT LIKE '4100%'
             AND ORDER_DATE BETWEEN '2008-01-01'
                                AND '2008-06-30');

COMPANY
------------------
Acme Mfg.
Ace International
Holm & Landis
JCP Inc.
```

Обратите внимание на то, что использование ключевого слова DISTINCT в подзапросе не является строго необходимым. Если один и тот же клиент окажется в результатах запроса многократно, главный запрос даст тот же результат. Возникает вопрос о том, что хуже — снижение производительности из-за удаления дубликатов в подзапросе или из-за обработки лишних строк в результатах подзапроса при выполнении проверок WHERE в главном запросе. Обычно большие по размеру результаты промежуточного запроса оказываются эффективнее сортировки, требующейся для удаления дублей. Встречаются и другие проблемы. Например, в Oracle применение GROUP BY обычно эффективнее, чем DISTINCT. Как видите, написание максимально эффективного запроса требует детального знания о том, как конкретная СУБД работает с инструкциями SQL.

Во всех приведенных примерах подзапрос возвращает в качестве результата столбец данных, а предложение WHERE главного запроса проверяет, равно ли значение из строки таблицы главного запроса одному из значений в полученном столбце. Таким образом, проверка IN с подзапросом выполняется аналогично простой проверке IN, за исключением того, что множество значений задается подзапросом, а не указывается явно в инструкции SELECT.

Проверка существования (EXISTS)

Как показано на рис. 9.5, в результате проверки существования (предикат EXISTS) можно выяснить, содержится ли в таблице результатов подзапроса хотя бы одна строка. Способа выполнить те же действия путем простой проверки сравнением не существует. Проверка на существование допустима только в подзапросах.

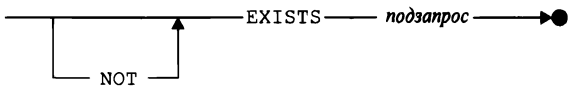


Рис. 9.5. Синтаксическая диаграмма проверки существования

Вот пример запроса, который можно легко сформулировать, используя проверку на существование.

Вывести список товаров, на которые получен заказ стоимостью не менее $25000.

Этот запрос можно перефразировать следующим образом.

Вывести список товаров, для которых в таблице ORDERS существует по крайней мере один заказ, который а) является заказом на данный товар; б) имеет стоимость не менее $25 000.

Инструкция SELECT, используемая для получения требуемого списка товаров, приведена ниже.

```
SELECT DISTINCT DESCRIPTION
  FROM PRODUCTS
 WHERE EXISTS (SELECT ORDER_NUM
                 FROM ORDERS
                WHERE PRODUCT = PRODUCT_ID
                  AND MFR = MFR_ID
                  AND AMOUNT >= 25000.00);

DESCRIPTION
--------------
500-1b Brace
Left Hinge
Right Hinge
Widget Remover
```

Концептуально главный запрос последовательно перебирает все строки таблицы PRODUCTS, и для каждого товара выполняет подзапрос. Результатом подзапроса является столбец данных, содержащий номера всех заказов "текущего" товара на сумму не менее $25000. Если такие заказы есть (т.е. столбец не пустой), то проверка EXISTS возвращает TRUE. Если подзапрос не дает ни одной строки заказов, проверка EXISTS возвращает значение FALSE. Эта проверка не может возвращать значение NULL.

Можно изменить логику проверки EXISTS и использовать форму NOT EXISTS. Тогда в случае, если подзапрос не создает ни одной строки результата, проверка возвращает TRUE, в противном случае — FALSE.

Обратите внимание на то, что предикат EXISTS в действительности вовсе *не использует* результаты подзапроса. Проверяется только наличие результатов. По этой причине в SQL смягчается правило, согласно которому "подзапрос должен возвращать один столбец данных", и в подзапросе проверки EXISTS допускается использование формы SELECT *. Поэтому предыдущий запрос можно переписать следующим образом.

Вывести список товаров, на которые получен заказ стоимостью не менее $25 000.

```
SELECT DESCRIPTION
  FROM PRODUCTS
 WHERE EXISTS (SELECT *
                 FROM ORDERS
                WHERE PRODUCT = PRODUCT_ID
                  AND MFR = MFR_ID
                  AND AMOUNT >= 25000.00);
```

На практике при использовании подзапроса в проверке EXISTS обычно применяется именно эта форма — SELECT *.

Вот некоторые дополнительные примеры запросов, в которых используется проверка EXISTS.

Вывести список клиентов, закрепленных за Сью Смит (Sue Smith), которые не разместили заказы на сумму свыше $3000.

```
SELECT COMPANY
  FROM CUSTOMERS
 WHERE CUST_REP = (SELECT EMPL_NUM
                     FROM SALESREPS
                    WHERE NAME = 'Sue Smith')
   AND NOT EXISTS (SELECT *
                     FROM ORDERS
                    WHERE CUST = CUST_NUM
                      AND AMOUNT > 3000.00);

COMPANY
----------------
Carter & Sons
Fred Lewis Corp.
```

Вывести список офисов, где имеется служащий, чей план превышает 55 процентов от плана офиса.

```
SELECT CITY
  FROM OFFICES
 WHERE EXISTS (SELECT *
                 FROM SALESREPS
                WHERE REP_OFFICE = OFFICE
                  AND QUOTA > (.55 * TARGET));
CITY
-------
Denver
Atlanta
```

Отметим, что во всех приведенных примерах подзапрос содержит внешнюю ссылку на столбец таблицы из главного запроса. На практике в подзапросе проверки EXISTS всегда имеется внешняя ссылка, связывающая подзапрос со строкой, проверяемой в настоящий момент главным запросом.

Многократное сравнение (предикаты ANY и ALL)*

При проверке IN выясняется, не равно ли некоторое значение одному из значений, содержащихся в столбце результатов подзапроса. В SQL имеются также две разновидности *многократного сравнения* — ANY и ALL, расширяющие предыдущую проверку до уровня других операторов сравнения, таких как больше (>) или меньше (<). Как показано на рис. 9.6, в обеих проверках некоторое значение сравнивается со столбцом данных, отобранных подзапросом.

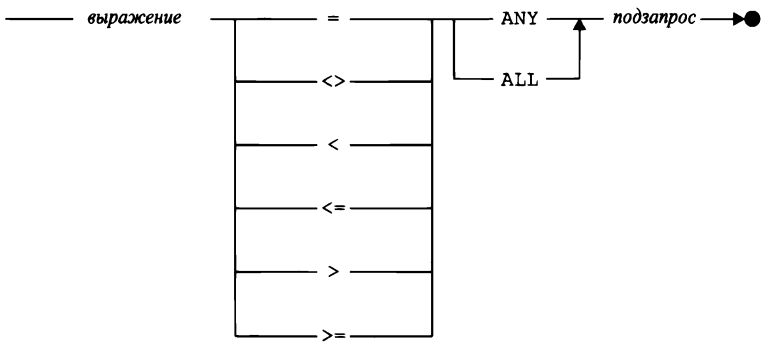


Рис. 9.6. Синтаксическая диаграмма многократного сравнения

Предикат ANY*

Ключевое слово ANY используется совместно с одним из шести операторов сравнения SQL (=, <>, <, <=, >, >=) и сравнивает проверяемое значение со столбцом данных, отобранных подзапросом. Проверяемое значение поочередно сравнивается с *каждым* элементом, содержащимся в столбце. Если *некоторое* из этих сравнений дает результат TRUE, то проверка ANY возвращает значение TRUE.

Вот пример запроса с предикатом ANY.

Вывести список служащих, принявших заказ на сумму, превышающую 10% плана.

```
SELECT NAME
  FROM SALESREPS
 WHERE (.1 * QUOTA) < ANY (SELECT AMOUNT
                             FROM ORDERS
                            WHERE REP = EMPL_NUM);

NAME
-----------
Sam Clark
Larry Fitch
Nancy Angelli
```

Концептуально главный запрос одну за одной проверяет все строки таблицы SALESREPS. Подзапрос находит все заказы, принятые текущим служащим, и возвращает столбец, содержащий стоимости этих заказов. Предложение WHERE главного запроса вычисляет десять процентов от плана текущего служащего и использует это число в качестве проверяемого значения, сравнивая его со стоимостью каждого заказа, отобранного подзапросом. Если есть *хотя бы один* заказ, стои-

мость которого превышает вычисленное проверяемое значение, то проверка < ANY возвращает значение TRUE, а имя служащего заносится в таблицу результатов запроса. Если таких заказов нет, имя служащего в таблицу результатов запроса не попадает. В соответствии со стандартом ANSI/ISO, вместо предиката ANY можно использовать предикат SOME. Обычно можно употреблять любой из них, но некоторые СУБД не поддерживают предикат SOME.

Иногда проверка ANY может оказаться трудной для понимания, поскольку включает в себя не одно сравнение, а несколько. Если прочитать условие сравнения немного по-другому, это поможет понять его смысл. Например, проверку

```
WHERE X < ANY (SELECT Y ...)
```

следует читать не как

```
где X меньше, чем любой выбранный Y...
```

а так:

```
где X меньше некоторого из Y...
```

Тогда предыдущий запрос можно перефразировать следующим образом:

Вывести список служащих, у которых для некоторого заказа, принятого ими, его стоимость превышает десять процентов от планового объема продаж служащего.

Если подзапрос в проверке ANY не создает ни одной строки или если результаты содержат значения NULL, то результат проверки может быть разным в различных СУБД. В стандарте ANSI/ISO для языка SQL содержатся подробные правила, определяющие результаты проверки ANY, когда проверяемое значение сравнивается со столбцом результатов подзапроса.

- Если подзапрос возвращает результат в виде пустого столбца, то проверка ANY возвращает значение FALSE (в результате выполнения подзапроса не получено ни одного значения, для которого выполнялось бы условие сравнения).
- Если операция сравнения дает TRUE *хотя бы для одного* значения в столбце, то проверка ANY возвращает значение TRUE (имеется некоторое значение, полученное подзапросом, для которого условие сравнения выполняется).
- Если операция сравнения дает FALSE для всех значений в столбце, то проверка ANY возвращает значение FALSE (можно утверждать, что ни для одного значения, возвращенного подзапросом, условие сравнения не выполняется).
- Если операция сравнения не дает TRUE ни для одного значения в столбце, но в нем имеется одно или несколько значений NULL, то проверка ANY возвращает результат NULL. В этой ситуации невозможно с определенностью сказать, существует ли полученное подзапросом значение, для которого выполняется условие сравнения; может быть, существует, а может, и нет — все зависит от "настоящих" (но пока неизвестных) значений неизвестных данных.

На практике проверка ANY иногда может приводить к ошибкам, которые трудно выявить, особенно когда применяется оператор сравнения "не равно" (<>). Вот пример, иллюстрирующий данную проблему.

Вывести имена и возраст всех служащих, которые не руководят офисами.

Заманчиво было бы выразить запрос таким образом.

```
SELECT NAME, AGE
  FROM SALESREPS
 WHERE EMPL_NUM <> ANY (SELECT MGR
                          FROM OFFICES);
```

Подзапрос

```
SELECT MGR
  FROM OFFICES;
```

в качестве результатов возвращает идентификаторы служащих, являющихся руководителями офисов, поэтому *кажется*, что запрос имеет следующий смысл.

Найти всех служащих, не являющихся руководителями офисов.

Но это *не так*! На самом деле приведенный запрос означает следующее.

Найти всех служащих, которые для некоторого офиса не являются его руководителями.

Конечно, для любого служащего можно найти *некоторый* офис, руководителем которого он не является. В таблицу результатов запроса войдут *все* служащие, поэтому запрос не дает ответа на поставленный вопрос! А правильным является следующий запрос.

```
SELECT NAME, AGE
  FROM SALESREPS
 WHERE NOT (EMPL_NUM = ANY (SELECT MGR
                             FROM OFFICES));

NAME           AGE
-------------  ---
Mary Jones      31
Sue Smith       48
Dan Roberts     45
Tom Snyder      41
Paul Cruz       29
Nancy Angelli   49
```

Запрос с предикатом ANY всегда можно преобразовать в запрос с предикатом EXISTS, перенося операцию сравнения внутрь условия отбора подзапроса. Это хорошая мысль, поскольку в этом случае исключаются ошибки, подобные только что рассмотренной. Вот альтернативная форма запроса с проверкой EXISTS.

```
SELECT NAME, AGE
  FROM SALESREPS
 WHERE NOT EXISTS (SELECT *
                     FROM OFFICES
                    WHERE EMPL_NUM = MGR);
```

```
NAME            AGE
-------------   ---
Mary Jones       31
Sue Smith        48
Dan Roberts      45
Tom Snyder       41
Paul Cruz        29
Nancy Angelli    49
```

Предикат ALL*

В проверке ALL, как и в проверке ANY, используется один из шести операторов (=, <>, <, <=, >, >=) для сравнения проверяемого значения со столбцом данных, отобранных подзапросом. Проверяемое значение поочередно сравнивается с каждым элементом, содержащимся в столбце. Если все сравнения дают результат TRUE, то проверка ALL возвращает значение TRUE.

Вот пример запроса с предикатом ALL.

Вывести список офисов с плановыми объемами продаж, у всех служащих которых фактический объем продаж превышает 50 процентов от плана офиса.

```
SELECT CITY, TARGET
  FROM OFFICES
 WHERE (.50 * TARGET) < ALL (SELECT SALES
                               FROM SALESREPS
                              WHERE REP_OFFICE = OFFICE);

CITY             TARGET
--------     -----------
Denver       $300,000.00
New York     $575,000.00
Atlanta      $350,000.00
```

Концептуально главный запрос поочередно проверяет каждую строку таблицы OFFICES. Подзапрос находит всех служащих, работающих в текущем офисе, и возвращает столбец с фактическими объемами продаж для каждого служащего. Предложение WHERE главного запроса вычисляет 50 процентов от плана продаж офиса и сравнивает это значение со всеми объемами продаж, получаемыми в результате выполнения подзапроса. Если все объемы продаж превышают проверяемое значение, то проверка < ALL возвращает значение TRUE и данный офис включается в таблицу результатов запроса. Если нет, то офис не попадает в таблицу результатов.

Проверка ALL, подобно проверке ANY, может оказаться сложной для понимания, поскольку включает в себя не одно сравнение, а несколько. И снова, если читать условие сравнения немного иначе, это помогает понять его смысл. Например, проверку

```
WHERE X < ALL (SELECT Y...)
```

следует читать не как

```
где X меньше, чем все выбранные Y...
```

а как

```
где для всех Y X меньше Y...
```

Тогда предыдущий запрос можно представить в таком виде.

Вывести список офисов, где для всех служащих 50 процентов плана офиса меньше, чем фактический объем продаж каждого служащего.

Если подзапрос в проверке ALL не возвращает ни одной строки или если результаты запроса содержат значения NULL, то в различных СУБД проверка ALL может выполняться по-разному. В стандарте ANSI/ISO SQL содержатся подробные правила, определяющие результаты проверки ALL, когда проверяемое значение сравнивается со столбцом результатов подзапроса.

- Если подзапрос возвращает результат в виде пустого столбца, то проверка ALL возвращает значение TRUE. Считается, что условие сравнения выполняется, даже если результаты подзапроса отсутствуют.
- Если операция сравнения дает результат TRUE для каждого значения в столбце, то проверка ALL возвращает значение TRUE. Условие сравнения выполняется для каждого значения, возвращенного подзапросом.
- Если операция сравнения дает результат FALSE для некоторого значения в столбце, то проверка ALL возвращает значение FALSE. В этом случае можно утверждать, что условие сравнения выполняется не для каждого значения, возвращенного подзапросом.
- Если операция сравнения не дает результат FALSE ни для одного значения в столбце, но для одного или нескольких значений дает результат NULL, то проверка ALL возвращает значение NULL. В этой ситуации нельзя с определенностью сказать, для всех ли значений, возвращенных подзапросом, справедливо условие сравнения; может быть, для всех, а может, и нет — все зависит от "настоящих" (но пока неизвестных) значений неизвестных данных.

Ошибки, которые могут случиться, если проверка ANY содержит оператор сравнения "не равно" (<>), могут возникнуть и при проверке ALL. Проверку ALL, так же как и проверку ANY, всегда можно преобразовать в эквивалентную проверку на существование (EXISTS), перенеся операцию сравнения в подзапрос.

Подзапросы и соединения

При чтении данной главы вы, возможно, заметили, что многие запросы, записанные с применением подзапросов, можно представить в виде многотабличных запросов. Такое случается довольно часто, и SQL позволяет записать запрос не одним способом, что иллюстрирует следующий пример.

Вывести имена и возраст служащих, работающих в офисах западного региона.

```
SELECT NAME, AGE
  FROM SALESREPS
 WHERE REP_OFFICE IN (SELECT OFFICE
                        FROM OFFICES
                       WHERE REGION = 'Western');
```

```
NAME             AGE
-------------    ---
Sue Smith         48
Larry Fitch       62
Nancy Angelli     49
```

Эта форма запроса очень близка к его словесной формулировке. Подзапрос возвращает список офисов западного региона, а главный запрос находит служащих, работающих в этих офисах. Вот альтернативная форма данного запроса, использующая соединение двух таблиц.

Вывести имена и возраст служащих, работающих в офисах западного региона.

```
SELECT NAME, AGE
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE
   AND REGION = 'Western';

NAME            AGE
-------------  ----
Sue Smith        48
Larry Fitch      62
Nancy Angelli    49
```

Данная форма запроса связывает таблицы SALESREPS и OFFICES, чтобы найти регион, в котором работает каждый служащий, а затем исключает тех служащих, которые не работают в западном регионе. Еще один способ решения поставленной задачи — запись запроса с применением оператора EXISTS.

Вывести имена и возраст служащих, работающих в офисах западного региона.

```
SELECT NAME, AGE
  FROM SALESREPS
 WHERE EXISTS (SELECT *
                 FROM OFFICES
                WHERE REGION = 'Western'
                  AND REP_OFFICE = OFFICE);

NAME             AGE
--------------  ----
Sue Smith         48
Larry Fitch       62
Nancy Angelli     49
```

Каждый из трех приведенных запросов корректно находит интересующих нас служащих, и ни один из них не является ни единственно правильным, ни ошибочным. Для большинства людей первый вариант (с использованием подзапроса) покажется более естественным, так как в словесном запросе не требуется никакой информации об офисах, и необходимость соединения таблиц SALESREPS и OFFICES для ответа на запрос кажется немного странной. Конечно, если запрос изменить так, чтобы в нем запрашивалась информация из таблицы OFFICES, то вариант с подзапросом больше не годится и необходимо использовать запрос к двум таблицам.

Вывести имена и возраст служащих, работающих в западном регионе, а также города, в которых они работают.

И наоборот, имеется много запросов с подзапросами, которые *нельзя* выразить в виде эквивалентного соединения. Вот простой пример.

Вывести имена и возраст служащих, для которых плановый объем продаж выше среднего.

```
SELECT NAME, AGE
  FROM SALESREPS
 WHERE QUOTA > (SELECT AVG(QUOTA)
                  FROM SALESREPS);

NAME          AGE
-----------   ---
Bill Adams     37
Sue Smith      48
Larry Fitch    62
```

В данном случае внутренний запрос является итоговым, а внешний — нет, поэтому из этих двух запросов нельзя создать соединение.

Вложенные подзапросы

Все рассмотренные до сих пор запросы были двухуровневыми и состояли из главного запроса и подзапроса. Точно так же как внутри главного запроса может находиться подзапрос, так и внутри подзапроса может находиться еще один подзапрос, называемый в таком случае вложенным. Вот пример трехуровневого запроса, в котором имеется главный запрос, подзапрос и "подподзапрос".

Вывести список клиентов, закрепленных за служащими, работающими в офисах восточного региона.

```
SELECT COMPANY
  FROM CUSTOMERS
 WHERE CUST_REP IN (SELECT EMPL_NUM
                      FROM SALESREPS
                     WHERE REP_OFFICE IN (SELECT OFFICE
                                            FROM OFFICES
                                           WHERE REGION = 'Eastern'));

COMPANY
---------------
First Corp.
Smithson Corp.
AAA Investments
JCP Inc.
Chen Associates
QMA Assoc.
Ian & Schmidt
Acme Mfg.
    .
    .
    .
```

В этом примере самый внутренний подзапрос

```
SELECT OFFICE
  FROM OFFICES
 WHERE REGION = 'Eastern';
```

возвращает столбец данных, содержащий идентификаторы офисов восточного региона. Следующий подзапрос

```
SELECT EMPL_NUM
  FROM SALESREPS
 WHERE REP_OFFICE IN (подзапрос);
```

возвращает столбец данных, содержащий идентификаторы служащих, работающих в одном из выбранных офисов. И наконец, внешний запрос

```
SELECT COMPANY
  FROM CUSTOMERS
 WHERE CUST_REP IN (подзапрос);
```

находит клиентов, закрепленных за выбранными служащими.

По такой же методике можно создавать запросы с четырьмя и более уровнями вложенности. Стандарт ANSI/ISO не определяет максимальное число уровней вложенности, но на практике с ростом их числа очень быстро увеличивается время выполнения запроса. Когда запрос имеет более двух уровней вложенности, он становится трудным для чтения и понимания. Во многих реализациях SQL количество уровней вложенности запросов ограничено относительно небольшим числом.

Коррелированные подзапросы*

Концептуально SQL выполняет подзапрос многократно — по одному разу для каждой строки главного запроса. Однако во многих случаях подзапрос возвращает *одни и те же* результаты для всех строк или для группы строк. Рассмотрим пример.

Вывести список офисов, в которых фактический объем продаж ниже усредненного планового объема продаж для всех офисов.

```
SELECT CITY
  FROM OFFICES
 WHERE SALES < (SELECT AVG(TARGET)
                  FROM OFFICES);

CITY
-------
Denver
Atlanta
```

В этом запросе было бы неразумно выполнять подзапрос пять раз (один раз для каждого офиса). Усредненный план не изменяется; он абсолютно не зависит от проверяемого в настоящий момент офиса. Следовательно, подзапрос можно выполнить только один раз и, получив усредненный план ($550 000), привести главный запрос к такому виду.

```
SELECT CITY
  FROM OFFICES
 WHERE SALES < 550000.00;
```

Коммерческие реализации SQL автоматически обнаруживают такие ситуации и используют сокращенное вычисление всякий раз, когда есть такая возможность, чтобы уменьшить объем работы, необходимой для выполнения запроса. Но когда подзапрос содержит внешнюю ссылку, данное упрощение применять нельзя, как показано в следующем примере.

Вывести список офисов, в которых плановые объемы продаж превышают суммарные планы служащих, работающих в них.

```
SELECT CITY
  FROM OFFICES
 WHERE TARGET > (SELECT SUM(QUOTA)
                   FROM SALESREPS
                  WHERE REP_OFFICE = OFFICE);

CITY
-----------
Chicago
Los Angeles
```

В разных строках таблицы OFFICES, проверяемой предложением WHERE главного запроса, столбец OFFICE (который является внешней ссылкой в подзапросе) имеет различные значения. Поэтому подзапрос должен выполняться пять раз — по одному разу для каждой строки таблицы OFFICES. Подзапрос, содержащий внешнюю ссылку, называется *коррелированным*, так как его результаты оказываются коррелированными с каждой строкой таблицы в главном запросе. По той же причине внешняя ссылка называется иногда *коррелированной*.

Подзапрос может содержать внешние ссылки на таблицу в предложении FROM *любого* запроса, который содержит данный подзапрос, независимо от его уровня вложенности. Например, имя столбца в подзапросе четвертого уровня может относиться как к одной из таблиц, указанных в предложении FROM главного запроса, так и к таблице в любом подзапросе второго или третьего уровня, содержащем данный подзапрос четвертого уровня. Независимо от уровня вложенности, внешняя ссылка всегда принимает значение столбца в текущей строке проверяемой таблицы.

Так как подзапрос может содержать внешние ссылки, вероятность появления неоднозначных ссылок на имена столбцов в подзапросе еще выше, чем в главном запросе. Если в подзапросе присутствует неполное имя столбца, SQL должен определить, относится ли оно к таблице предложения FROM самого подзапроса или к предложению FROM запроса, содержащего подзапрос. Чтобы минимизировать возможность путаницы, в SQL всегда предполагается, что ссылка на столбец в подзапросе относится к ближайшему возможному предложению FROM. Для иллюстрации приведем пример, где одна и та же таблица используется и в запросе, и в подзапросе.

Вывести список руководителей старше 40 лет, у которых есть служащие, опережающие план.

```
SELECT NAME
  FROM SALESREPS
 WHERE AGE > 40
   AND EMPL_NUM IN (SELECT MANAGER
                      FROM SALESREPS
                     WHERE SALES > QUOTA);

NAME
-----------
Sam Clark
Larry Fitch
```

Столбцы MANAGER, QUOTA и SALES в подзапросе являются ссылками на таблицу SALESREPS в предложении FROM самого подзапроса; SQL *не* интерпретирует их как внешние ссылки, и подзапрос не является коррелированным. Как уже говорилось ранее, СУБД в данном случае может сначала выполнить подзапрос: найти служащих, опережающих план, и составить список, содержащий идентификаторы их руководителей. Затем СУБД может приступить к выполнению главного запроса и отобрать имена руководителей из полученного списка.

Если вы хотите создать внешнюю ссылку в подзапросе, аналогичном рассмотренному в предыдущем примере, то для этого должны использовать псевдонимы таблицы. Как это делается, показано на примере запроса, в который, по сравнению с предыдущим, добавлено еще одно условие.

Вывести список руководителей старше 40 лет, у которых есть служащие, опережающие план и работающие со своим руководителем в разных офисах.

```
SELECT NAME
  FROM SALESREPS MGRS
 WHERE AGE > 40
   AND MGRS.EMPL_NUM IN (SELECT MANAGER
                           FROM SALESREPS EMPS
                          WHERE EMPS.SALES > EMPS.QUOTA
                            AND EMPS.REP_OFFICE
                             <> MGRS.REP_OFFICE);

NAME
-----------
Sam Clark
Larry Fitch
```

Теперь копия таблицы SALESREPS в главном запросе имеет псевдоним MGRS, а копия в подзапросе — псевдоним EMPS. Подзапрос содержит одно дополнительное условие отбора, требующее, чтобы идентификатор офиса служащего не был равен идентификатору офиса руководителя. Полное имя столбца MGRS.REP_OFFICE в подзапросе — это внешняя ссылка, и данный подзапрос является коррелированным.

Подзапросы в предложении HAVING*

Хотя подзапросы чаще всего находятся в предложении WHERE, их можно использовать и в предложении HAVING главного запроса. Когда подзапрос содержится в предложении HAVING, он участвует в отборе групп строк. Рассмотрим следующий запрос с подзапросом.

Вывести список служащих, у которых средняя стоимость заказов на товары, изготовленные компанией ACI, выше, чем общая средняя стоимость заказов.

```
SELECT NAME, AVG(AMOUNT)
  FROM SALESREPS, ORDERS
 WHERE EMPL_NUM = REP
   AND MFR = 'ACI'
 GROUP BY NAME
HAVING AVG(AMOUNT) > (SELECT AVG(AMOUNT)
                        FROM ORDERS);

NAME         AVG(AMOUNT)
----------   -----------
Sue Smith     $15,000.00
Tom Snyder    $22,500.00
```

На рис. 9.7 показана схема выполнения этого запроса. Подзапрос вычисляет среднюю стоимость по всем заказам. Это простой подзапрос, не содержащий внешних ссылок, поэтому искомая средняя стоимость вычисляется один раз, а затем многократно используется в предложении HAVING. Главный запрос просматривает строки таблицы ORDERS, отыскивая все заказы на товары компании ACI, и группирует их по именам служащих. Затем предложение HAVING сравнивает среднюю стоимость по каждой группе заказов со средней стоимостью по всем заказам, вычисленной ранее. Если средняя стоимость по группе больше, чем общая средняя стоимость, то данная группа строк сохраняется, если нет, то группа строк удаляется. Наконец, предложение SELECT создает для каждой группы итоговую строку, содержащую имя служащего и среднюю стоимость принятых им заказов.

В предложении HAVING можно также использовать и коррелированный подзапрос. Однако поскольку подзапрос выполняется один раз для каждой *группы* строк, все внешние ссылки в коррелированном подзапросе должны иметь одно и то же значение для каждой группы строк. На практике это означает, что внешняя ссылка должна либо быть ссылкой на столбец группировки внешнего запроса, либо находиться внутри статистической функции. В последнем случае значение статистической функции для проверяемой группы строк вычисляется в ходе выполнения подзапроса.

Если слегка изменить предыдущий запрос, то подзапрос в предложении HAVING станет коррелированным.

Вывести список служащих, у каждого из которых средняя стоимость заказов на товары, изготовленные компанией ACI, не меньше, чем средняя стоимость всех заказов этого служащего.

```
SELECT NAME, AVG(AMOUNT)
  FROM SALESREPS, ORDERS
```

```
 WHERE EMPL_NUM = REP
   AND MFR = 'ACI'
 GROUP BY NAME, EMPL_NUM
HAVING AVG(AMOUNT) >= (SELECT AVG(AMOUNT)
                         FROM ORDERS
                        WHERE REP = EMPL_NUM);

NAME          AVG(AMOUNT)
-----------   -----------
Bill Adams      $7,865.40
Sue Smith      $15,000.00
Tom Snyder     $22,500.00
```

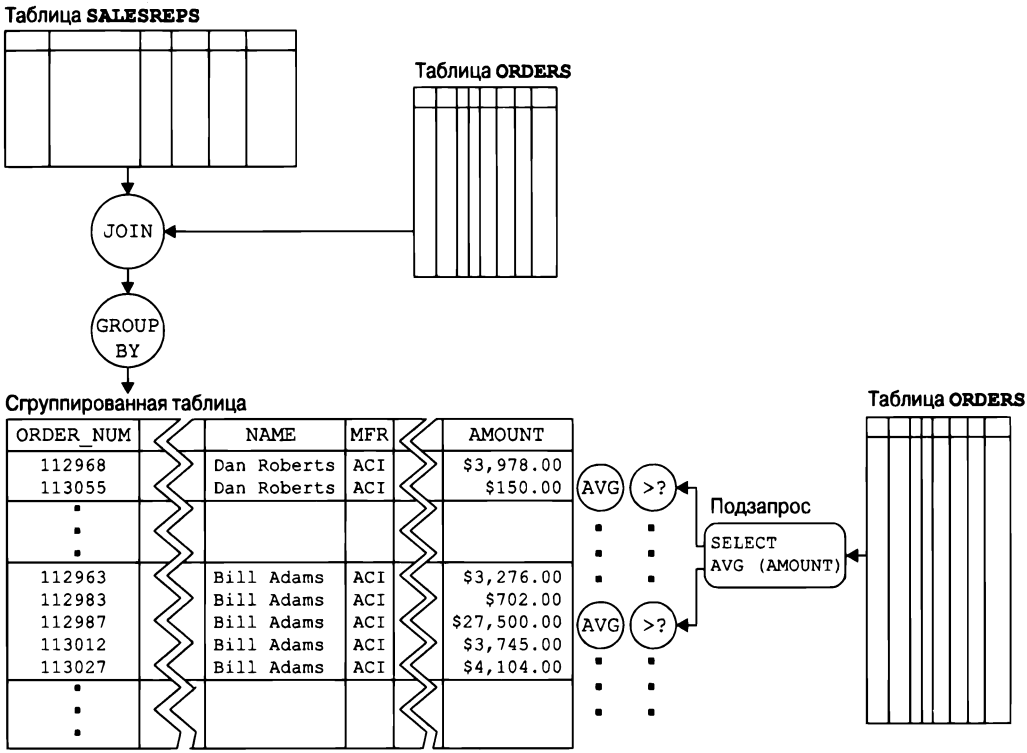


Рис. 9.7. Выполнение подзапроса в предложении HAVING

В этом примере подзапрос должен вычислять среднюю стоимость заказов служащего, чья группа строк проверяется в настоящий момент предложением HAVING. Подзапрос считывает заказы, принятые этим служащим, с помощью внешней ссылки EMPL_NUM. Эта внешняя ссылка корректна, поскольку столбец EMPL_NUM имеет одинаковые значения во всех строках группы, созданной главным запросом.

Резюме по подзапросам

В настоящей главе были рассмотрены подзапросы, позволяющие использовать результаты одного запроса для определения другого.

- Подзапрос является "запросом внутри запроса". Такие запросы содержатся в одном из условий отбора в предложении WHERE либо HAVING.
- Когда подзапрос содержится в предложении WHERE, его результаты используются для отбора отдельных строк, предоставляющих данные в результаты запроса.
- Когда подзапрос содержится в предложении HAVING, его результаты используются для отбора групп строк, предоставляющих данные в результаты запроса.
- Подзапросы могут быть вложены в другие подзапросы.
- При сравнении с результатом подзапроса проверяемое значение сравнивается посредством одного из операторов сравнения с единственным значением, которое возвращается подзапросом.
- При проверке на принадлежность результатам подзапроса (IN) значение выражения проверяется на равенство одному из множества значений, которые возвращаются подзапросом.
- Проверка на существование (EXISTS) позволяет выяснить, возвращает ли подзапрос какие-нибудь значения.
- При многократном сравнении (ANY и ALL) значение выражения сравнивается посредством одного из операторов сравнения со всеми значениями, отобранными подзапросом, чтобы выяснить, выполняется ли условие сравнения для некоторых либо для всех значений.
- В подзапросе можно использовать внешние ссылки на таблицы любого запроса, внутри которого он находится; такая ссылка связывает подзапрос с текущей строкой данного запроса.

Ниже приведена окончательная версия правил обработки запроса SQL, в которую включены подзапросы. Она представляет собой полное определение результатов запроса SELECT.

1. Если запрос представляет собой объединение (UNION) инструкций SELECT, для каждой из них выполнить шаги 2–7 для генерации отдельных результатов запросов.
2. Сформировать произведение таблиц, указанных в предложении FROM. Если там указана только одна таблица, то произведением будет она сама.
3. При наличии предложения WHERE применить указанное в нем условие к каждой строке таблицы произведения, оставляя только те строки, для которых условие истинно, и отбрасывая строки, для которых условие ложно или равно NULL. Если предложение WHERE содержит подзапрос, он выполняется для каждой проверяемой строки.

4. При наличии предложения GROUP BY разделить строки, оставшиеся в таблице произведения, на группы таким образом, чтобы в каждой группе строки имели одинаковые значения во всех столбцах группировки.
5. Если имеется предложение HAVING, применить указанное в нем условие к каждой строке группы, оставляя только те группы, для которых условие истинно, и отбрасывая группы, для которых условие ложно или равно NULL. Если предложение HAVING содержит подзапрос, он выполняется для каждой проверяемой группы строк.
6. Для каждой из оставшихся строк (или каждой группы строк) вычислить значение каждого элемента в списке возвращаемых столбцов и создать одну строку таблицы результатов запроса. При простой ссылке на столбец берется значение столбца в текущей строке (или группе строк); при использовании статистической функции в качестве ее аргумента при наличии предложения GROUP BY используются значения столбца из всех строк, входящих в группу; в противном случае используется все множество строк.
7. Если указан предикат DISTINCT, удалить из таблицы результатов запроса все повторяющиеся строки.
8. Если запрос является объединением инструкций SELECT, объединить результаты выполнения отдельных инструкций в единую таблицу результатов запроса. Удалить из нее повторяющиеся строки, если только в запросе не указан предикат UNION ALL.
9. Если имеется предложение ORDER BY, отсортировать результаты запроса, как в нем указано.

Сгенерированные этой процедурой строки составляют результаты запроса.

Сложные запросы*

Рассматриваемые до этого момента запросы SQL представляют собой базовые возможности языка SQL, поддерживаемые большинством реализаций SQL. Комбинирование этих возможностей — выбор столбцов в предложении SELECT, отбор строк по критерию, заданному в предложении WHERE, соединение таблиц, указанных в предложении FROM, вычисление итоговых данных с помощью предложений GROUP BY и HAVING и использование подзапросов — предоставляет пользователю мощный и эффективный механизм получения и анализа данных. Однако профессионалы сталкиваются с рядом серьезных ограничений, затрудняющих или даже делающих невозможным извлечение из базы данных некоторой информации.

- *Отсутствие средств принятия решений.* Предположим, что вы хотите сгенерировать отчет на основе информации из нашей учебной базы данных, состоящий из двух столбцов: в первом выводятся названия всех офисов компании, а во втором для каждого офиса выводится либо его плановый объем продаж на год, либо объем продаж на текущий момент — в зависимости от того, какая сумма больше. С помощью стандартных

средств SQL сделать это трудно. Еще один пример. У вас есть база данных с квартальными объемами продаж каждого офиса (четыре столбца данных для каждого офиса) и вы хотите написать программу, которая выводит перечень офисов и объемов их продаж за указанный пользователем квартал. Для решения этой задачи стандартные SQL-запросы тоже подходят плохо. Вам придется включить в программу четыре отдельных SQL-запроса (по одному для каждого квартала), и, в зависимости от выбора пользователя, программа должна направлять СУБД один из этих запросов. Конечно, это не так уж сложно сделать, по крайней мере для столь простого примера, но, в целом, такого рода ограничения ведут к усложнению прикладных программ за счет переноса в них логики, которую естественнее было бы включить в запросы.

- *Ограниченное использование подзапросов.* На подзапросы стандарт SQL1 накладывает ряд ограничений. Простейший пример — правило, согласно которому в предложении WHERE подзапрос может стоять только справа от оператора сравнения. Запрос "перечислить офисы, для которых сумма плановых объемов продаж служащих превышает план самого офиса" наиболее естественно было бы выразить так.

```
SELECT OFFICE
  FROM OFFICES
 WHERE (SELECT SUM(QUOTA)
          FROM SALESREPS
         WHERE REP_OFFICE = OFFICE) < TARGET;
```

Однако, с точки зрения стандарта SQL1, эта инструкция неверна, хотя и поддерживается последующими версиями стандарта. Тем не менее большинство людей легче воспримут запрос, если поменять сравниваемые элементы местами.

```
SELECT OFFICE
  FROM OFFICES
 WHERE TARGET > (SELECT SUM(QUOTA)
                  FROM SALESREPS
                 WHERE REP_OFFICE = OFFICE);
```

В этом простом примере нетрудно развернуть логику, но это ограничение в лучшем случае неудобно и, например, не позволяет сравнивать результаты двух подзапросов.

- *Ограниченное использование выражений со строками таблиц.* Предположим, что вы хотите вывести список производителей, идентификаторов и цен нескольких взаимозаменяемых товаров. Концептуально это совокупность товаров, идентификационные данные которых (идентификатор производителя, идентификатор товара) соответствуют одному из элементов заданного набора пар значений, и было бы естественно написать запрос с проверкой на принадлежность множеству.

```
SELECT MFR_ID, PRODUCT_ID, PRICE
  FROM PRODUCTS
 WHERE (MFR_ID, PRODUCT_ID) IN (('ACI',41003),
                                ('BIC',41089), ...);
```

Однако стандарт SQL1 не допускает выполнения такой проверки вхождения во множество. Вместо этого вам придется составить длинную цепочку отдельных сравнений, соединенных операторами AND и OR.

- *Ограниченное использование табличных выражений.* SQL позволяет создать представление BIGORDERS с перечнем "больших" заказов:

```
SELECT *
  FROM ORDERS
 WHERE AMOUNT > 1000;
```

а затем использовать это представление в качестве таблицы в предложении FROM запроса, определяющего, какие товары и в каких количествах входили в эти заказы.

```
SELECT MFR, PRODUCT, SUM(QTY)
  FROM BIGORDERS
 GROUP BY MFR, PRODUCT;
```

Концептуально SQL должен позволять включать определение представления прямо в запрос, например, так.

```
SELECT MFR, PRODUCT, SUM(QTY)
  FROM (SELECT * FROM ORDERS WHERE AMOUNT > 1000) A
 GROUP BY MFR, PRODUCT;
```

Увы, но стандарт SQL1 не допускает непосредственного использования подзапросов в предложении FROM. При этом совершенно очевидно, что СУБД все равно должна уметь определить смысл такого запроса, поскольку должна выполнять, по сути, те же действия при интерпретации определения представления BIGORDERS .

Как показывают эти примеры, стандарт SQL1 и СУБД, реализующие стандарт до этого уровня, существенно ограничивают использование в запросах выражений, включающих отдельные элементы данных, их множества, строки и таблицы. Последующие версии стандарта SQL включают ряд дополнительных возможностей, направленных на устранение указанных ограничений и повышение обобщенности языка SQL. Дух этих изменений можно выразить следующим образом: пользователь должен иметь возможность написать запрос так, что если написанное выражение имеет смысл с точки зрения логики, то оно должно быть корректно и с точки зрения SQL. Поскольку, по сравнению со стандартом SQL1, новые возможности значительно расширили язык SQL, большинство из них требует полной совместимости реализации SQL с последней версией стандарта SQL.

Выражения со скалярными значениями

Простейшие из расширенных возможностей стандарта SQL обеспечивают большую гибкость операций и вычислений, включающих в свой состав отдельные значения данных, называемые в стандарте SQL *скалярами* или *скалярными значениями*. Имеется три основных источника скалярных значений, используемых в инструкциях SQL:

- значение отдельного столбца отдельной строки таблицы;
- литерал, такой как 125.7 или "АБВ";
- значение, введенное пользователем в прикладной программе.

В запросе

```
SELECT NAME, EMPL_NUM, HIRE_DATE, (QUOTA * .9)
  FROM SALESREPS
 WHERE (REP_OFFICE = 13) OR TITLE = 'VP Sales'
```

имена столбцов NAME, EMPL_NUM, HIRE_DATE и QUOTA приводят к генерации отдельных значений для каждой строки результатов запроса, как и имена столбцов REP_OFFICE и TITLE в предложении WHERE. Кроме того, отдельные значения генерируются литералами 0.9 и 13 и строкой "VP Sales". Если эта инструкция будет встроена в прикладную программу (см. главу 17, "Встроенный SQL"), значение номера офиса может храниться, например, в переменной office_num, и с ее участием тот же запрос может выглядеть так.

```
SELECT NAME, EMPL_NUM, HIRE_DATE, (QUOTA * .9)
  FROM SALESREPS
 WHERE (REP_OFFICE = :office_num) OR TITLE = 'VP Sales';
```

Как показывает этот и многие другие рассмотренные ранее примеры, отдельные значения могут объединяться в простые выражения наподобие QUOTA * .9. К этим простейшим выражениям стандарт SQL добавил оператор CAST для явного преобразования типов данных, оператор CASE — для организации ветвлений, оператор COALESCE — для условного создания значений, отличных от NULL, и оператор NULLIF — для условного создания значений NULL.

Выражение CAST

Стандарт SQL предусматривает очень строгие правила сочетания различных типов данных в выражениях. Он требует от СУБД автоматического преобразования простейших типов данных, таких как 2- и 4-байтовые целые. Однако если вы попытаетесь сравнить числовое и символьное значения, то, в соответствии со стандартом, СУБД должна сообщить об ошибке, причем даже в том случае, если строка на самом деле содержит число. Однако вы можете явно попросить СУБД выполнить преобразование типов с помощью выражения CAST (синтаксис которого показан на рис. 9.8).

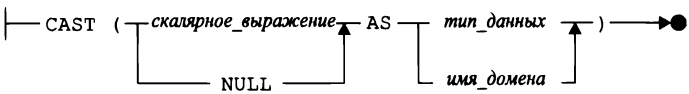


Рис. 9.8. Синтаксическая диаграмма выражения CAST

При интерактивном вводе SQL-инструкций выражение CAST редко оказывается нужным и важным. Однако оно может стать критичным при использовании SQL из языка программирования, типы данных которого не соответствуют типам данных SQL. Например, в приведенном ниже запросе оператор CAST в предложении SELECT преобразовывает значения полей REP_OFFICE (целое число) и HIRE_DATE (дата) в символьные строки, которые будут возвращены в качестве результата запроса.

```
SELECT NAME, CAST (REP_OFFICE AS CHAR),
             CAST (HIRE_DATE AS CHAR)
  FROM SALESREPS;
```

Поддержка CAST различна в разных реализациях SQL. Например, Oracle требует, чтобы типы данных CHAR и VARCHAR, используемые в выражении CAST, включали указание длины; MySQL и DB2 UDB в выражениях CAST не поддерживают тип данных DATE.

В общем случае, оператор CAST может находиться в SQL-инструкциях везде, где допускаются выражения, возвращающие скалярные значения. В приведенном ниже примере он используется в предложении WHERE для преобразования номера клиента из символьной формы в целое число, чтобы его можно было сравнивать со значениями столбца из базы данных.

```
SELECT PRODUCT, QTY, AMOUNT
  FROM ORDERS
 WHERE CUST = CAST ('2107' AS INTEGER);
```

Вместо типа данных в выражении с оператором CAST в SQL2 разрешается указывать *домен*. Домен — это набор допустимых значений, который можно определить в базе данных. О доменах подробно рассказывается в главе 11, "Целостность данных", поскольку они играют важную роль в обеспечении целостности данных. Заметим, что с помощью оператора CAST можно также сгенерировать значение NULL требуемого типа.

Вот для чего чаще всего применяется оператор CAST.

- Для преобразования данных из таблицы, в которой столбец определен с неверным типом данных. Например, столбец может быть определен как строковый, но вы знаете, что на самом деле он содержит числа (т.е. строки цифр) или даты (т.е. строки, которые могут быть интерпретированы как календарные даты).
- Для приведения возвращенных запросом данных к типу, поддерживаемому языком, на котором написано клиентское приложение. Например, большинство языков программирования не поддерживает специальных типов данных для даты и времени, и для обработки таких значений программой их нужно преобразовывать в символьные строки.
- Для устранения отличий типов данных в двух разных таблицах. Например, если в таблице заказов дата заказа хранится в виде значений типа DATE, а в таблице наличия товара на складе дата хранится в виде символьной строки, для их сравнения нужно либо преобразовать дату заказа в строку, либо строку из складской таблицы привести к типу DATE. Подобным же образом, если вы хотите объединить данные из двух таблиц с помощью операции UNION, объединяемые столбцы должны быть одинакового типа. Поэтому для выполнения такого объединения может понадобиться привести столбцы одной таблицы к типу данных столбцов другой.

Выражение CASE

Оператор CASE обеспечивает ограниченные возможности принятия решения в SQL-выражениях. Его базовая структура, представленная на рис. 9.9, подобна структуре конструкции IF...THEN...ELSE, имеющейся во многих языках программирования. Когда СУБД встречает выражение CASE, она вычисляет первое условие, и если оно истинно, выполняется первое результирующее выражение. Если же первое условие ложно, проверяется второе условие, и если оно истинно, выполняется второе результирующее выражение, и т.д.

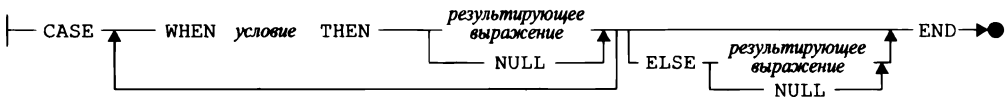


Рис. 9.9. Синтаксическая диаграмма выражения CASE

Вот простейший пример с оператором CASE. Предположим, что мы хотим разделить клиентов на три категории в соответствии с суммами их кредитного лимита. У клиентов категории A лимит кредита превышает $60000, у клиентов категории B — $30000, а остальные клиенты относятся к категории C. Без применения выражения CASE нам пришлось бы извлечь из базы данных имена и кредитные лимиты клиентов и рассчитывать на то, что приложение сумеет проверить эти лимиты и верно назначить клиентам категории. Выражение CASE позволяет возложить всю эту работу на СУБД.

```
SELECT COMPANY,
       CASE WHEN CREDIT_LIMIT > 60000 THEN 'A'
            WHEN CREDIT_LIMIT > 30000 THEN 'B'
            ELSE 'C'
       END AS CREDIT_RATING
  FROM CUSTOMERS;
```

Для каждой строки результатов запроса СУБД вычисляет выражение CASE. Сначала она сравнивает лимит кредита со значением $60000. Если результатом оказывается TRUE, она возвращает во втором столбце значение A. В противном случае лимит кредита сравнивается со значением $30000. Если результатом оказывается TRUE, второй столбец получает значение B, иначе — C.

Это очень простой пример выражения CASE. В этом примере в качестве возможных результатов оператора CASE были указаны простые литералы, но в общем случае это могут быть любые выражения SQL. Кроме того, совершенно не обязательно, чтобы все проверки, выполняемые в предложениях WHEN, были подобными друг другу, как в приведенном примере. Само выражение CASE может находиться и в других предложениях запроса. Рассмотрим пример его использования в предложении WHERE. Предположим, что вам нужны общие объемы продаж служащих по офисам. Если служащему еще не назначен офис, его сумма должна быть включена в итог по офису его руководителя. Вот запрос, который формирует такие данные.

```
SELECT CITY, SUM(SALESREPS.SALES)
  FROM OFFICES, SALESREPS
 WHERE OFFICE = CASE WHEN (REP_OFFICE IS NOT NULL)
```

```
                        THEN REP_OFFICE
                    ELSE (SELECT REP_OFFICE
                            FROM SALESREPS MGRS
                           WHERE MGRS.EMPL_NUM = MANAGER)
                END
 GROUP BY CITY;
```

Стандарт SQL предоставляет сокращенную версию выражения CASE, более удобную для тех распространенных случаев, когда нужно сравнить проверяемое значение некоторого типа с последовательностью значений данных (чаще всего литеральных). Синтаксис этой версии представлен на рис. 9.10. Вместо повторения условия вида

проверяемое_значение = значение

в каждом предложении WHEN, можно указать проверяемое значение только один раз. Предположим, вам нужен список всех офисов компании с именами их руководителей и названиями городов и штатов, в которых они расположены. В нашей базе данных названия штатов отсутствуют, так что запрос должен сам сгенерировать эту информацию. Вот как это делается при помощи выражения CASE в списке SELECT.

```
SELECT NAME, CITY, CASE OFFICE WHEN 11 THEN 'New York'
                               WHEN 12 THEN 'Illinois'
                               WHEN 13 THEN 'Georgia'
                               WHEN 21 THEN 'California'
                               WHEN 22 THEN 'Colorado'
                        END AS STATE
  FROM OFFICES, SALESREPS
 WHERE MGR = EMPL_NUM;
```

Рис. 9.10. Альтернативный синтаксис выражения CASE

Выражение COALESCE

Одно из наиболее частых применений оператора CASE — для работы со значениями NULL. Например, когда вы запрашиваете из базы данных набор строк, который хотите представить пользователю, значения NULL в нем обычно лучше заменять некоторыми литералами, соответствующими их реальному смыслу, вроде слова "отсутствует". Пусть, к примеру, пользователю нужен отчет о служащих и их плановых объемах продаж. Если служащему еще не назначен план, в этом столбце отчета нужно вывести его фактический объем продаж на текущий период. Если же по какой-то причине значением столбца SALES также оказывается NULL, тогда во втором столбце отчета нужно вывести нуль. Для выполнения этой задачи требуется конструкция IF...THEN...ELSE, которую прекрасно эмулирует оператор CASE.

```
SELECT NAME, CASE WHEN (QUOTA IS NOT NULL) THEN QUOTA
                  WHEN (SALES IS NOT NULL) THEN SALES
                  ELSE 0.00
                  END AS ADJUSTED_QUOTA
  FROM SALESREPS;
```

Поскольку значения NULL очень часто обрабатываются подобным образом, в стандарт SQL для выполнения этих действий включено выражение COALESCE (рис. 9.11) — нечто вроде выражения CASE специального вида. Правила его обработки очень просты. СУБД вычисляет первое выражение в списке. Если его значение не равно NULL, оно становится результатом всего оператора COALESCE. В противном случае СУБД переходит ко второму выражению и проверяет, равно ли оно NULL. Если нет, оно возвращается в качестве результата, иначе СУБД переходит к третьему выражению и т.д. Вот как приведенный выше пример можно переписать с использованием оператора COALESCE:

```
SELECT NAME, COALESCE (QUOTA, SALES, 0.00)
  FROM SALESREPS;
```

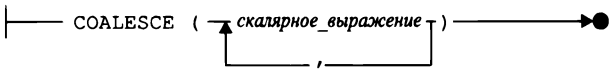


Рис. 9.11. Синтаксическая диаграмма выражения COALESCE

Как видите, второй запрос намного проще и понятнее первого, а их действия абсолютно идентичны. Так что оператор COALESCE упрощает написание определенного типа запросов, хотя с функциональной точки зрения не привносит в SQL никаких новых возможностей.

Выражение NULLIF

Если в одних случаях значения NULL нужно удалить из набора данных, то в других эти значения нужно, наоборот, создать. Во многих приложениях, связанных с обработкой данных (особенно разработанных до того, как реляционные СУБД стали столь популярны), отсутствующие данные представляются не значениями NULL, а специальными кодами, которые не могут интерпретироваться как данные. Предположим, например, что в нашей учебной базе данных, когда служащий не имеет начальника, в столбце MANAGER стоит не NULL, как это бывает обычно, а нуль (0). В некоторых случаях требуется выявлять такие коды и менять их на NULL. Это можно сделать с помощью выражения NULLIF (рис. 9.12). Когда СУБД встречает такое выражение, она анализирует первое выражение (обычно это имя столбца) и сравнивает его значение со значением второго выражения (обычно кодом, представляющим отсутствующие данные). Если они равны, результатом оператора NULLIF становится NULL, иначе — значение первого выражения.

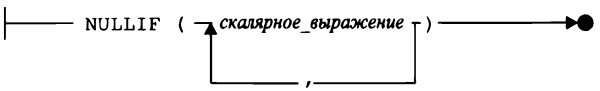


Рис. 9.12. Синтаксическая диаграмма выражения NULLIF

Вот пример запроса, в котором нуль означает отсутствие номера офиса.

```
SELECT CITY, SUM(SALESREPS.SALES)
  FROM OFFICES, SALESREPS
 WHERE OFFICE = NULLIF (REP_OFFICE, 0)
 GROUP BY CITY;
```

Конечно, операторам CASE, COALESCE и NULLIF далеко до настоящих логических конструкций, имеющихся в классических языках программирования, но для выражения логики запросов их возможностей вполне достаточно. Одним из их важнейших преимуществ является то, что они позволяют возложить на СУБД большую часть обработки данных, чем это было возможно до их появления, разгрузив тем самым прикладные программы и освободив от лишней работы пользователей, которые вводят запросы вручную.

Выражения со строками таблиц

Хотя столбцы и содержащиеся в них скалярные значения представляют собой атомарные строительные блоки реляционной базы данных, одной из самых важных функций реляционной модели является объединение столбцов в строки, представляющие объекты реального мира, такие как отдельные офисы, клиенты или заказы. Стандарт SQL1 и большинство коммерческих СУБД предоставляют весьма ограниченные возможности для манипулирования строками и группами строк. SQL1 разрешает только добавление строки в таблицу, а также выборку, обновление или удаление групп строк (с помощью инструкций INSERT, SELECT, UPDATE и DELETE).

Более поздние версии стандарта SQL продвинулись в этом направлении гораздо дальше, позволяя использовать в SQL-выражениях строки практически так же, как и скалярные данные. В стандарте предусмотрен специальный синтаксис для формирования строк данных. Он допускает существование подзапросов, возвращающих строки. И кроме того, он определяет смысл операторов сравнения и других действий при работе со строками.

Конструктор строки

Стандарт SQL содержит специальную конструкцию, предназначенную для определения строк данных, — *конструктор строкового значения* (рис. 9.13). В наиболее распространенном случае это просто разделенный запятыми список литералов или скалярных выражений. Вот, например, конструктор строки данных, структура которой соответствует структуре таблицы OFFICES.

```
(23, 'San Diego', 'Western', NULL, DEFAULT, 0.00)
```

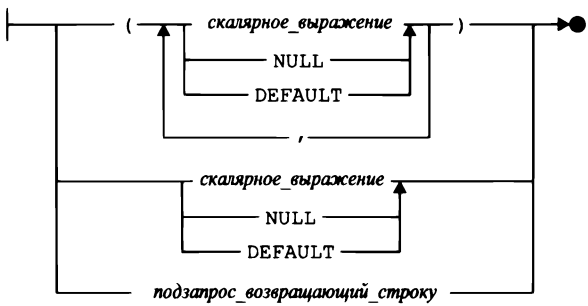


Рис. 9.13. Синтаксическая диаграмма конструктора строкового значения

Результатом этого выражения является одна строка данных с шестью столбцами. Ключевое слово NULL указывает, что четвертый столбец результирующей строки должен содержать значение NULL. Ключевое слово DEFAULT в пятой позиции указывает на то, что пятый столбец должен содержать значение, присваиваемое этому столбцу по умолчанию. Это ключевое слово может встречаться в конструкторе только в определенных ситуациях — например, когда конструктор включен в инструкцию INSERT и генерируемая им строка добавляется в таблицу.

Если конструктор строки находится в предложении WHERE, в нем могут использоваться имена отдельных столбцов — в качестве отдельных элементов данных или в составе выражений. В качестве примера рассмотрим следующий запрос.

Перечислить номера заказов на изделие ACI 41002 с указанием количества единиц товара и суммы сделки.

```
SELECT ORDER_NUM, QTY, AMOUNT
  FROM ORDERS
 WHERE (MFR, PRODUCT) = ('ACI', '41002');
```

При выполнении этого запроса предложение WHERE по очереди применяется к каждой строке таблицы ORDERS. Первый конструктор в этом предложении (слева от знака равенства) генерирует строку из двух столбцов, содержащую идентификаторы производителя и товара текущего обрабатываемого заказа. Второй конструктор (справа от знака равенства) генерирует строку из двух столбцов, содержащую литерально заданные идентификатор производителя ACI и идентификатор товара 41002. Далее оператор = сравнивает две *строки* (обратите внимание — строки, а не скалярные значения!). Стандартом SQL определено, что оператор сравнения строк по очереди сравнивает пары значений для каждого столбца. Результатом сравнения является значение TRUE тогда и только тогда, когда все попарные сравнения вернули TRUE. Конечно, тот же запрос можно написать и без конструкторов.

```
SELECT ORDER_NUM, QTY, AMOUNT
  FROM ORDERS
 WHERE (MFR = 'ACI') AND (PRODUCT = '41002');
```

Для такого простого примера обе формы запроса одинаково понятны. Но в случае более сложных запросов роль конструкторов строк очень важна, особенно когда в сравнении участвует результат подзапроса.

Подзапросы, возвращающие строки

Как уже говорилось ранее в этой главе, стандарт SQL1 предусматривает возможность создания сложных запросов к базе данных с использованием механизма подзапросов. Подзапрос — это обычный запрос на выборку (т.е. инструкция SELECT), но стандарт SQL1 требует, чтобы он возвращал скалярное значение, т.е. только *одно* значение данных. Это значение участвует в одном из выражений в главной SQL-инструкции, содержащей подзапрос. Такая методика использования подзапросов поддерживается всеми ведущими современными СУБД масштаба предприятия.

Последующие версии стандарта SQL значительно расширяют возможности подзапросов: теперь они могут генерировать не только одиночные значения, но и *строки*, которые могут участвовать в выражениях и сравниваться с другими строками. Предположим, нам требуется узнать номера и даты заказов самого дорогого товара. Логично начать с написания запроса, который находит идентификатор этого товара и его производителя.

Найти идентификатор самого дорогого товара и идентификатор его производителя.

```
SELECT MFR_ID, PRODUCT_ID
  FROM PRODUCTS
 WHERE PRICE = (SELECT MAX(PRICE)
                  FROM PRODUCTS);
```

Если предположить, что в базе данных есть лишь один самый дорогой товар, то этот запрос сгенерирует одну строку данных, состоящую из двух столбцов. Воспользовавшись механизмом подзапросов SQL, мы встраиваем наш запрос в качестве подчиненного в запрос, извлекающий из базы данных информацию о заказах на найденный товар.

Вывести номера и даты заказов самого дорогого товара.

```
SELECT ORDER_NUM, ORDER_DATE
  FROM ORDERS
 WHERE (MFR, PRODUCT) = (SELECT MFR_ID, PRODUCT_ID
                           FROM PRODUCTS
                          WHERE PRICE = (SELECT MAX(PRICE)
                                           FROM PRODUCTS));
```

Предложение WHERE на самом верхнем уровне содержит оператор сравнения строк. Слева от него стоит конструктор строк, в котором указаны имена двух столбцов. Каждый раз, когда проверяется это предложение, конструктор генерирует строку, состоящую из идентификатора производителя и идентификатора товара из текущей строки таблицы ORDERS. Справа от знака равенства стоит подзапрос, который находит идентификационные данные самого дорогого товара. Результатом этого запроса также является строка, состоящая из двух столбцов с теми же типами данных, что и в строке слева от знака равенства.

Тот же запрос можно сформулировать и без помощи подзапроса, возвращающего строку, но результат получится более громоздким.

```
SELECT ORDER_NUM, ORDER_DATE
  FROM ORDERS
 WHERE (MFR IN (SELECT MFR_ID
                  FROM PRODUCTS
                 WHERE PRICE = (SELECT MAX(PRICE)
                                  FROM PRODUCTS)))
   AND (PRODUCT IN (SELECT PRODUCT_ID
                      FROM PRODUCTS
                     WHERE PRICE = (SELECT MAX(PRICE)
                                      FROM PRODUCTS)));
```

Вместо сравнения строк в предложении WHERE нам пришлось выполнить два отдельных скалярных сравнения: одно — для идентификаторов производителей, а другое — для идентификаторов товаров. Из-за разбиения сравнений подзапрос

для поиска максимальной цены приходится выполнять дважды. Кроме того, запрос с обработкой строк формулируется гораздо естественнее и понятнее.

Сравнение строк

Чаще всего выражения со строками используются в предложениях WHERE и HAVING в операции проверки на равенство, как в нескольких последних примерах. Построенная строка (зачастую содержащая значения столбцов из строки, претендующей на включение в результаты запроса) сравнивается с другой строкой (обычно с результатом подзапроса или набором литеральных значений), и если эти строки совпадают, строка-кандидат включается в таблицу результатов запроса. Стандартом SQL предусмотрены и другие возможности сравнения строк — на неравенство и принадлежность определенному диапазону. При сравнении строк на неравенство в SQL используются те же правила, что и при обычной сортировке строк. Сначала сравниваются первые столбцы двух строк, в случае их неравенства они используются для определения порядка строк. Если же они равны, сравниваются вторые столбцы двух строк и т.д. Далее приведены результаты сравнения нескольких строк, составленных из трех столбцов, взятых из таблицы ORDERS.

```
('ACI', '41002', 54) < ('REI', '2A44R',  5)
                          — сортировка по первому столбцу
('ACI', '41002', 54) < ('ACI', '41003', 35)
                          — сортировка по второму столбцу
('ACI', '41002', 10) < ('ACI', '41002', 54)
                          — сортировка по третьему столбцу
```

Табличные выражения

Стандарт SQL существенно расширил возможности языка не только в отношении работы со скалярными значениями и строками, но и в плане работы с таблицами. Им определен механизм динамического создания таблиц прямо в инструкциях SQL. Согласно стандарту, подзапросы могут возвращать целые таблицы, а главный запрос может анализировать эти таблицы и сверять с ними собственные данные. Кроме того, подзапросы могут использоваться и в других местах инструкций SQL — например, в предложении FROM инструкции SELECT в качестве одной или нескольких таблиц-источников. Наконец, стандарт SQL предлагает расширенные возможности комбинирования таблиц, включая операции объединения (UNION), пересечения (INTERSECTION) и вычитания (EXCEPT).

Конструктор таблицы

Стандарт SQL позволяет указать значения ячеек таблицы непосредственно в SQL-инструкции при помощи выражения специального типа — *конструктора таблицы* (рис. 9.14). В своей простейшей форме этот конструктор представляет собой разделенный запятыми список конструкторов строк, каждый из которых, в свою очередь, состоит из разделенных запятыми литералов — значений отдельных столбцов. Например, инструкция SQL INSERT использует табличный конструктор в качестве источника вставляемых в таблицу данных. Если в SQL1 инструкция INSERT (см. главу 10, "Внесение изменений в базу данных") могла добавить в таблицу только одну строку, то стандарт SQL2 (и последующие)

позволяет добавлять в таблицу сразу несколько строк, как в приведенном ниже запросе. Следует отметить, что не все реализации SQL поддерживают этот синтаксис.

Добавить в таблицу OFFICES данные о трех новых офисах.

```
INSERT INTO OFFICES (OFFICE, CITY, REGION, MGR, SALES)
     VALUES (23, 'San Diego', 'Western', 108, 0.00),
            (24, 'Seattle', 'Western', 104, 0.00),
            (14, 'Boston', 'Eastern', NULL, 0.00);
```

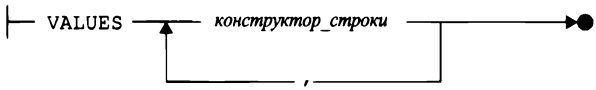


Рис. 9.14. Синтаксическая диаграмма конструктора таблицы

Заметим, что отдельные строки в табличном конструкторе не обязательно должны содержать только литеральные значения. Источником данных может быть и запрос, возвращающий скалярное значение либо строку. Хотя для нашей учебной базы данных следующий пример не имеет особого смысла, а использованный в нем синтаксис поддерживается не всеми СУБД, примененная в нем инструкция INSERT вполне корректна.

Добавить в таблицу OFFICES данные о трех новых офисах.

```
INSERT INTO OFFICES (OFFICE, CITY, REGION, MGR, SALES)
     VALUES (23, 'San Diego', 'Western', 108, 0.00),
            (24, 'Seattle', 'Western',
                       (SELECT MANAGER
                          FROM SALESREPS
                         WHERE EMPL_NUM = 105), 0.00),
            (SELECT 14, 'Boston', REGION, MGR, 0.00
               FROM OFFICES
              WHERE OFFICE = 12);
```

Как и в предыдущем примере, предложение VALUES в инструкции INSERT генерирует таблицу из трех строк и пяти столбцов, которые должны быть добавлены в таблицу OFFICES. Первая строка задана как набор литеральных значений. Во второй строке значение четвертого столбца генерируется скалярным подзапросом, который извлекает из таблицы SALESREPS идентификатор менеджера служащего с идентификатором 105. Третья строка целиком генерируется строковым подзапросом. Три столбца, указанных в его предложении SELECT, представляют собой константные значения, но значения третьего и четвертого столбцов извлекаются из таблицы OFFICES — это регион и руководитель Нью-йоркского офиса (идентификатор офиса 12).

Табличные подзапросы

Стандарт SQL поддерживает подзапросы, возвращающие не только скалярные значения и строки, но и *табличные подзапросы*, возвращающие целые таблицы. (Заметим, что SQL Server не поддерживает данный синтаксис, в отличие от таких СУБД, как Oracle, MySQL и DB2 UDB.) Табличные подзапросы играют очень важ-

ную роль в предложениях WHERE и HAVING, где возвращаемые ими значения могут использоваться в выражениях со специальными операциями сравнения. Предположим, например, что нам нужны описания и цены всех товаров, для которых имеются заказы на сумму, большую $20000. Пожалуй, наиболее естественно сформулировать этот запрос с использованием табличного подзапроса.

Вывести описания и цены всех товаров, для которых имеются отдельные заказы на сумму, большую $20000.

```
SELECT DESCRIPTION, PRICE
  FROM PRODUCTS
 WHERE (MFR_ID, PRODUCT_ID) IN (SELECT MFR, PRODUCT
                                  FROM ORDERS
                                 WHERE AMOUNT > 20000.00);
```

Запрос соответствует простому описанию на естественном языке: выбрать описания и цены товаров, идентификационные данные которых (как и в предыдущем примере, это идентификаторы товара и его производителя) соответствуют одному товару из некоторого набора. Этот набор — точнее, таблица из двух столбцов — генерируется подзапросом, а вхождение в набор идентификаторов очередного анализируемого товара проверяется в предложении WHERE с помощью специального предиката IN.

Конечно, запрос для нашего примера можно выразить и иначе. Вспоминая главу 7, "Многотабличные запросы (соединения)", вы, вероятно, можете предложить альтернативный вариант — соединение таблиц PRODUCTS и ORDERS на основе составного условия.

Вывести описания и цены всех товаров, для которых имеются отдельные заказы на сумму, большую $20000.

```
SELECT DISTINCT DESCRIPTION, PRICE
  FROM PRODUCTS, ORDERS
 WHERE (MFR_ID = MFR)
   AND (PRODUCT_ID = PRODUCT)
   AND (AMOUNT > 20000.00);
```

Обратите внимание на необходимость добавить ключевое слово DISTINCT, поскольку имеется два заказа более чем на $20000 на товар "Right Hinge". Приведенный запрос равнозначен предыдущему, но он куда меньше соответствует естественному описанию, так что большинству людей его понять труднее. Поэтому для более сложных запросов возможность использовать подзапросы, возвращающие таблицы, очень важна — она позволяет выражать запросы проще и понятнее.

Спецификация запроса SQL

То, что мы с вами в трех последних главах называли инструкцией SELECT, запросом на выборку или просто запросом, стандарт SQL формально называет *спецификацией запроса*. Чтобы понять выражения с таблицами, описанные в следующем разделе, полезно ознакомиться с этим формальным определением. Синтаксис спецификации запроса показан на рис. 9.15. Ее компоненты должны быть вам уже хорошо знакомы из предыдущих глав.

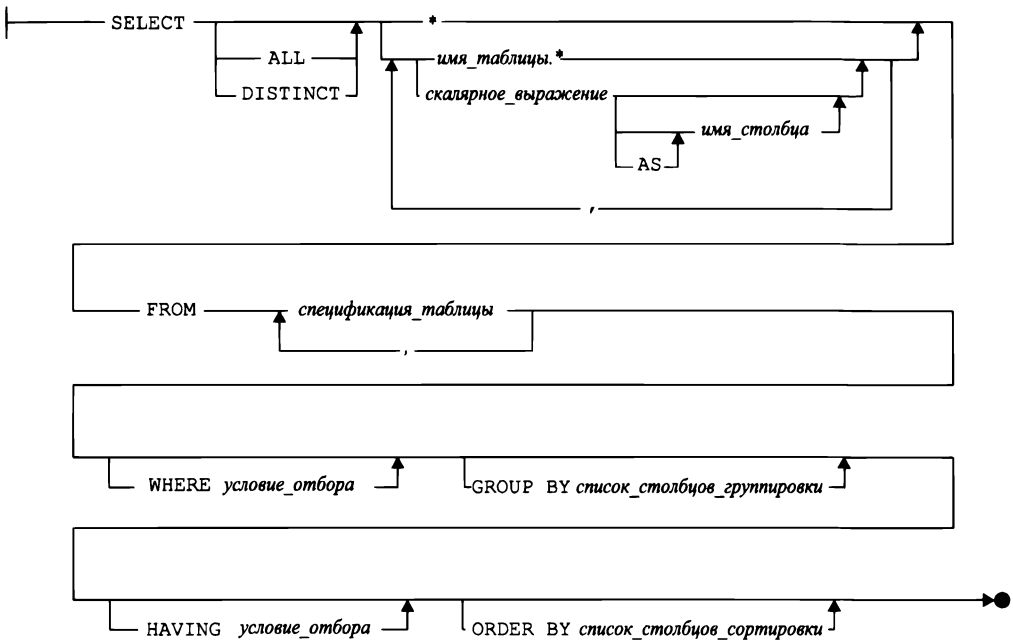


Рис. 9.15. Спецификация запроса SQL: формальное определение

- В *списке возвращаемых столбцов* предложения SELECT определены столбцы результатов запроса. Каждый столбец определяется выражением, которое сообщает СУБД, как вычислять его значения. Столбцам могут (но не обязательно должны) быть присвоены псевдонимы, задаваемые с помощью ключевого слова AS.
- Ключевые слова ALL и DISTINCT определяют, будут ли из результирующего набора строк удаляться повторяющиеся строки.
- Предложение FROM определяет таблицы, на основе которых будут формироваться результаты запроса.
- Предложение WHERE указывает СУБД, как ей определять, какие из строк следует включать в запрос.
- Предложения GROUP BY и HAVING управляют процессом группировки отобранных строк и отбором групп для включения в окончательные результаты запроса.
- Предложение ORDER BY определяет желаемую последовательность строк в результатах запроса.

Спецификация запроса является базовым блоком для построения запросов в стандарте SQL. Концептуально она описывает процесс составления строк и столбцов результирующего набора строк на основе данных из таблиц, указанных в предложении FROM. "Значением" спецификации запроса является *таблица* данных. В простейшем случае запрос SQL состоит из одной простой спецификации запроса. Более сложный запрос может представлять собой несколько специ-

фикаций, вложенных одна в другую по принципу подзапросов. Наконец, с помощью табличных операций спецификации могут объединяться в выражения запросов, о которых рассказывается в следующем разделе.

Выражения запросов

Стандарт SQL определяет *выражение запроса* как универсальный способ определения таблицы результатов запроса. В выражении запроса могут использоваться следующие базовые строительные блоки.

- Спецификация запроса, описанная в предыдущем разделе (`SELECT... FROM...`). Ее значением является таблица результатов запроса.
- Конструктор таблицы, о котором рассказывалось выше (`VALUES...`). Его значением является таблица, построенная из заданных в конструкторе значений.
- Явная ссылка на таблицу (`TABLE` *имя_таблицы*). Ее значением является содержимое указанной таблицы.

SQL позволяет комбинировать результаты этих конструкций с помощью следующих операций.

- **`JOIN`**. SQL поддерживает перекрестные соединения (декартовые произведения), естественные, внутренние, а также все типы внешних соединений, описанные в главе 7, "Многотабличные запросы (соединения)". Операция `JOIN` из двух входных таблиц создает выходную таблицу результатов запроса в соответствии со спецификацией соединения.
- **`UNION`**. Операция SQL `UNION` выполняет сложение двух совместимых таблиц (т.е. таблиц с одинаковым количеством столбцов и одинаковыми типами данных соответствующих столбцов). Операция `UNION` из двух входных таблиц создает выходную объединенную таблицу результатов запроса.
- **`EXCEPT`**. Операция SQL `EXCEPT` получает две входные таблицы и генерирует одну, содержащую все строки, которые имеются в первой таблице, но отсутствуют во второй. Концептуально операция `EXCEPT` представляет собой вычитание таблиц. Строки второй таблицы удаляются из первой, а то, что остается, становится результатом операции.
- **`INTERSECT`**. Операция SQL `INTERSECT` получает две входные таблицы и генерирует одну выходную, содержащую все строки, которые имеются в обеих входных таблицах.

Операции UNION, INTERSECT и EXCEPT

Операции `UNION`, `INTERSECT` и `EXCEPT` представляют три базовые операции над множествами, которые из двух входных создают одну выходную таблицу. Почти все производители СУБД поддерживают операцию `UNION`, но поддержка `INTERSECT` и `EXCEPT` у разных производителей реализована по-разному. Например, Oracle вместо ключевого слова `EXCEPT` использует ключевое слово `MINUS`. Все три операции требуют, чтобы входные таблицы были *совместимыми* — имели

одинаковое количество столбцов, а соответствующие столбцы имели одинаковые типы данных. Вот несколько простых примеров SQL запросов, в которых используются эти операции.

Вывести список товаров, для которых имеются заказы на сумму более $30 000, а также тех товаров, которых на складе имеется на сумму более $30 000.

```
(SELECT MFR, PRODUCT
   FROM ORDERS
  WHERE AMOUNT > 30000.00)
UNION
(SELECT MFR_ID, PRODUCT_ID
   FROM PRODUCTS
  WHERE (PRICE * QTY_ON_HAND) > 30000);
```

Вывести список товаров, для которых имеются заказы на сумму более $30 000 и которых при этом имеется на складе на сумму более $30 000.

```
(SELECT MFR, PRODUCT
   FROM ORDERS
  WHERE AMOUNT > 30000.00)
INTERSECT
(SELECT MFR_ID, PRODUCT_ID
   FROM PRODUCTS
  WHERE (PRICE * QTY_ON_HAND) > 30000);
```

Вывести список товаров, для которых имеются заказы на сумму более $30 000, за исключением тех, которые стоят менее $100.

```
(SELECT MFR, PRODUCT
   FROM ORDERS
  WHERE AMOUNT > 30000.00)
EXCEPT
(SELECT MFR_ID, PRODUCT_ID
   FROM PRODUCTS
  WHERE PRICE < 100.00);
```

По умолчанию операции UNION, INTERSECT и EXCEPT удаляют из результирующей таблицы повторяющиеся строки. Как правило, это хорошо; в частности, в приведенных выше примерах именно такой результат и нужен, но в некоторых случаях может потребоваться оставить все повторяющиеся строки. Для этого используются специальные формы трех операций: UNION ALL, INTERSECT ALL и EXCEPT ALL.

Обратите внимание на то, что в каждом из приведенных примеров создается таблица с двумя столбцами. Ее строки извлекаются из двух разных таблиц: ORDERS и PRODUCTS. Однако столбцы этих таблиц имеют одинаковые типы данных, так что к ним могут быть применены операции сложения, вычитания и пересечения. В нашей учебной базе данных соответствующие столбцы имеют разные имена (например, идентификатор производителя в таблице ORDERS называется MFR, а в таблице PRODUCTS — MFR_ID).

Запросы в предложении FROM

Составной запрос в SQL — это гораздо более мощное и гибкое средство создания и комбинирования таблиц результатов запроса, нежели обычный механизм подзапросов и операция UNION, которые предлагались стандартом SQL1. Чтобы сделать составные запросы еще более универсальными, стандарт SQL разрешает использовать их практически везде, где в запросах SQL1 допускались ссылки на таблицы. В частности, выражение запроса может указываться вместо имени таблицы в предложении FROM. Вот простейший пример запроса, в котором используется эта возможность.

Вывести имена и общие суммы заказов для всех клиентов, чей лимит кредита превышает $50 000.

```
SELECT COMPANY, TOT_ORDERS
  FROM CUSTOMERS, (SELECT CUST, SUM(AMOUNT) AS TOT_ORDERS
                     FROM ORDERS
                    GROUP BY CUST)
 WHERE (CREDIT_LIMIT > 50000.00)
   AND (CUST_NUM = CUST);
```

Если первая спецификация таблицы в предложении FROM — это, как обычно, имя таблицы базы данных, то вторая спецификация — вовсе не имя таблицы, а полноценный запрос. В действительности, последний может быть намного сложнее и включать собственные операции UNION или JOIN. Когда запрос включается в предложение FROM, как в данном примере, концептуально СУБД *сначала* выполняет этот запрос и создает временную таблицу его результатов (в нашем случае эта временная таблица состоит из двух столбцов, содержащих номера пользователей и общие суммы их заказов). Затем СУБД выполняет главный запрос, используя в нем созданную таблицу в качестве источника данных точно так же, как она использовала бы обычную таблицу, хранящуюся в базе данных. В нашем примере содержимым временной таблицы является информация, извлеченная из таблицы ORDERS, и к этой информации присоединяется таблица CUSTOMERS для получения названий компаний.

Тот же запрос можно сформулировать и другими способами. Например, можно написать одноуровневый запрос с операцией JOIN, соединяющей таблицы CUSTOMERS и ORDERS, и группировкой результатов. Операция соединения таблиц может быть выполнена явно с применением оператора JOIN, с последующей группировкой в запросе верхнего уровня. Как показывает этот пример, одним из преимуществ составных запросов SQL является то, что у пользователя обычно есть несколько способов получения одного и того же результата.

В основе возможностей SQL в данной области лежит стремление к тому, чтобы язык SQL позволял пользователю выразить свой запрос совершенно естественно, не ограничивая его жесткими рамками раз и навсегда определенных конструкций. СУБД должна уметь проанализировать запрос пользователя, разделить его на базовые блоки и определить наиболее эффективный способ его выполнения. Этот внутренний план выполнения запроса может сильно отличаться от той последовательности действий, которая соответствует форме запроса. Но именно в этом и вся прелесть: пользователю (или программисту) нужно только сообщить СУБД, что именно он хочет получить, а как это сделать, причем сделать наиболее эффективно, — это уже ее забота.

Резюме по SQL-запросам

На этом мы заканчиваем изучение SQL-запросов и инструкции SELECT, начатое в главе 6, "Простые запросы". Как показали четыре последние главы, предложения инструкции SELECT предоставляют программисту мощный и в то же время гибкий набор средств для получения информации из базы данных. Каждое предложение при выборке данных играет свою особую роль.

- В предложении FROM указываются исходные таблицы, из которых данные считываются в таблицу результатов запроса. Каждое имя столбца в инструкции SELECT должно однозначно определять столбец одной из этих таблиц или представлять собой внешнюю ссылку на столбец исходной таблицы внешнего запроса.
- Предложение WHERE, если оно имеется, выбирает из исходных таблиц отдельные комбинации строк, которые включаются в таблицу результатов запроса. Подзапросы в предложении WHERE выполняются для каждой отдельной строки.
- Предложение GROUP BY, если оно имеется, группирует отдельные строки, отобранные предложением WHERE, в группы строк.
- Предложение HAVING, если оно имеется, отбирает группы строк, которые включаются в таблицу результатов запроса. Подзапросы в предложении HAVING выполняются для каждой группы строк.
- Предложение SELECT определяет, какие именно данные будут представлены столбцами окончательных результатов запроса.
- Предикат DISTINCT, если он имеется, исключает из таблицы результатов запроса повторяющиеся строки.
- Операция UNION, если она имеется, объединяет результаты, полученные отдельными инструкциями SELECT, в одно множество результатов запроса.
- Предложение ORDER BY, если оно имеется, сортирует окончательную таблицу результатов запроса по одному или нескольким столбцам.
- Возможности выражений запросов SQL добавляют к стандарту SQL1 выражения со строками и таблицами, а также операции INTERSECT и EXCEPT. Фундаментальный поток обработки запросов при этом остается неизменным, но существенно возрастает возможность использования подзапросов в запросах.

III ЧАСТЬ

Обновление данных

Глава 10
Внесение изменений в базу данных

Глава 11
Целостность данных

Глава 12
Обработка транзакций

SQL позволяет не только извлекать данные, находящиеся в базе данных, но и изменять их. Следующие три главы посвящены вопросам, связанным с внесением изменений в базу данных. В главе 10, "Внесение изменений в базу данных", рассматриваются инструкции SQL, с помощью которых можно добавлять и удалять данные, а также обновлять данные, уже находящиеся в базе. В главе 11, "Целостность данных", рассказывается о том, как SQL поддерживает целостность данных при их изменении. В главе 12, "Обработка транзакций", описываются средства обработки транзакций в SQL, позволяющие нескольким пользователям одновременно изменять содержимое одной базы данных.

10
ГЛАВА

Внесение изменений в базу данных

SQL представляет собой полноценный язык, предназначенный для работы с данными и используемый не только для извлечения информации из базы данных путем запросов на выборку, но и для изменения содержащейся в ней информации с помощью запросов на добавление, удаление и обновление. По сравнению со сложностью инструкции SELECT, с помощью которой выполняются запросы на выборку, инструкции SQL, изменяющие содержимое базы данных, кажутся исключительно простыми. Однако при изменении содержимого базы данных к СУБД предъявляется ряд дополнительных требований. При внесении изменений СУБД должна сохранять целостность данных, допускать внесение в базу данных только допустимых значений, а также обеспечивать непротиворечивость базы данных даже в случае системной ошибки. Помимо этого, СУБД должна обеспечивать возможность одновременного изменения базы данных несколькими пользователями таким образом, чтобы они не мешали друг другу.

В данной главе рассматриваются три инструкции SQL, используемые для изменения содержимого базы данных.

- **INSERT** Добавляет новые строки в таблицу
- **DELETE** Удаляет строки из таблицы
- **UPDATE** Изменяет существующие данные в таблице

В главе 11, "Целостность данных", описываются средства SQL, поддерживающие целостность данных, а в главе 12, "Обработка транзакций", — средства SQL, обеспечивающие одновременную работу нескольких пользователей.

Добавление новых данных

Добавление новой строки в реляционную базу данных происходит тогда, когда во внешнем мире появляется новый объект, представляемый этой строкой. На примере нашей учебной базы данных это выглядит следующим образом.

- Если вы принимаете на работу нового служащего, в таблицу SALESREPS необходимо добавить новую строку с данными о нем.
- Если служащий заключает договор с новым клиентом, в таблицу CUSTOMERS должна быть добавлена новая строка, представляющая этого клиента.
- Если клиент делает заказ, в таблицу ORDERS требуется добавить новую строку, содержащую информацию об этом заказе.

Во всех случаях новая строка добавляется для того, чтобы база данных оставалась точной моделью реального мира. Наименьшей единицей информации, которую можно добавить в реляционную базу данных, является строка. В общем случае, в реляционной СУБД существует три способа добавления новых строк в базу данных.

- **Однострочная инструкция INSERT**. *Однострочная* инструкция INSERT позволяет добавить в таблицу одну новую строку. Она широко используется в повседневных приложениях, например в программах ввода данных.
- **Многострочная инструкция INSERT**. *Многострочная* инструкция INSERT обеспечивает извлечение строк из одной части базы данных и добавление их в другую таблицу. Например, она обычно используется в конце месяца или года, когда "старые" строки таблицы пересылаются в неактивную таблицу для хранения.
- **Пакетная загрузка**. Утилита *пакетной загрузки* служит для добавления в таблицу данных из внешнего файла. Эта утилита обычно используется для первоначальной загрузки базы данных либо для загрузки данных из другой компьютерной системы или данных, собранных из различных источников.

Однострочная инструкция INSERT

Однострочная инструкция INSERT, синтаксическая диаграмма которой представлена на рис. 10.1, добавляет в таблицу новую строку. В предложении INTO указывается таблица-получатель (*целевая* таблица), в которую добавляется новая строка, а в предложении VALUES содержатся значения данных для новой строки. Список столбцов определяет, какие значения в какой столбец новой строки заносятся.

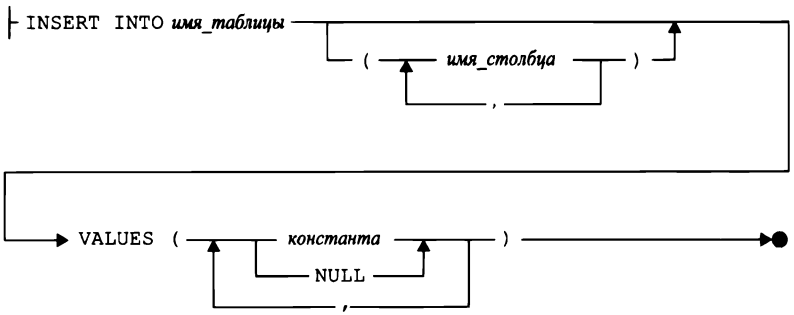


Рис. 10.1. Синтаксическая диаграмма однострочной инструкции INSERT

Предположим, вы только что приняли на работу нового служащего Генри Якобсена (Henry Jacobsen) со следующими данными.

Имя	Henry Jacobsen
Возраст	36
Идентификатор	111
Должность	Sales Mgr
Офис	Атланта (идентификатор офиса 13)
Дата приема	25 июля 2008 года
Личный план продаж	Еще не установлен
Объем продаж на текущую дату	$0.00

Ниже приведена инструкция INSERT, которая добавляет информацию об этом служащем в нашу учебную базу данных.

Добавить информацию о новом служащем.

```
INSERT INTO SALESREPS (NAME, AGE, EMPL_NUM, SALES, TITLE,
                       HIRE_DATE, REP_OFFICE)
     VALUES ('Henry Jacobsen', 36, 111, 0.00, 'Sales Mgr',
             '2008-07-25', 13);

1 row inserted.
```

На рис. 10.2 представлена графическая схема выполнения инструкции INSERT. Вначале инструкция создает новую строку, структура которой повторяет структуру столбцов таблицы, а затем заполняет ее значениями из предложения VALUES и добавляет эту строку в таблицу. Строки в таблице не упорядочены, поэтому нет никаких указаний о том, где вставлять строку: в начало, в конец таблицы или где-то между строк. Эта строка будет входить в результаты последующих запросов на выборку таблицы SALESREPS, но в таблице результатов запроса она может находиться в любом месте.

Инструкция SQL

```
INSERT INTO SALESREPS (NAME, AGE, EMPL_NUM, SALES, QUOTA, TITLE, ...)
VALUES ('Henry Jacobsen', 36, 111, 0.00, NULL, 'Sales Mgr', NULL, '2008-07-25', 13);
```

Новая строка

111	Henry Jacobson	36	13	Sales Mgr	2008-07-25	NULL	NULL	0.00

Таблица **SALESREPS**

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	2006-02-12	104	$350,000.00	$367,911.00
109	Mary Jones	31	11	Sales Rep	2007-10-12	106	$300,000.00	$392.725.00
102	Sue Smith	48	21	Sales Rep	2004-12-10	108	$350,000.00	$474,050.00
106	Sam Clark	52	11	VP Sales	2006-06-14	NULL	$275,000.00	$299,912.00
104	Bob Smith	33	12	Sales Mgr	2005-05-19	106	$200,000.00	$142,594.00
101	Dan Roberts	45	12	Sales Rep	2004-10-20	104	$300,000.00	$305,673.00
110	Tom Snyder	41	NULL	Sales Rep	2008-01-13	101	NULL	$75,985.00
108	Larry Fitch	62	21	Sales Mgr	2007-10-12	106	$350,000,00	$361,865.00
103	Paul Cruz	29	12	Sales Rep	2005-03-01	104	$275,000.00	$286,775.00
107	Nancy Angelli	49	22	Sales Rep	2006-11-14	108	$300,000.00	$186,042.00

Рис. 10.2. Добавление в таблицу одной строки

Предположим, что служащий Якобсен получает свой первый заказ от компании InterCorp, нового клиента, которому присвоен идентификатор 2126. Это заказ на 20 изделий ACI-41004 общей стоимостью $2340, и ему присваивается номер 113069. Вот как выглядят инструкции INSERT, добавляющие в базу данных информацию о новом клиенте и заказе.

Добавить информацию о новом клиенте и заказе для служащего Якобсена.

```
INSERT INTO CUSTOMERS (COMPANY, CUST_NUM,
                       CREDIT_LIMIT, CUST_REP)
     VALUES ('InterCorp', 2126, 15000.00, 111);

1 row inserted.

INSERT INTO ORDERS (AMOUNT, MFR, PRODUCT, QTY,
                    ORDER_DATE, ORDER_NUM, CUST, REP)
     VALUES (2340.00, 'ACI', '41004', 20, CURRENT_DATE,
             113069, 2126, 111);

1 row inserted.
```

Как показывает приведенный пример, если в таблице много столбцов, то инструкция INSERT может оказаться довольно длинной, однако ее структура по-прежнему останется очень простой. Во второй инструкции INSERT в предложении VALUES используется системная константа CURRENT DATE, которая обеспечивает ввод текущей даты в качестве даты получения заказа. Эта системная константа определена в стандарте SQL и поддерживается многими ведущими СУБД, включая Oracle и MySQL. В других СУБД, таких как SQL Server и DB2UDB, для получения текущих даты и времени используются другие системные константы или встроенные функции.

Инструкцию INSERT можно использовать в интерактивном режиме для добавления строк в таблицы, которые изменяются очень редко, например в таблицу OFFICES. Однако на практике данные о новом клиенте, заказе или служащем почти всегда добавляются в базу данных с помощью программ ввода данных, использующих специальные формы. После окончания ввода данных в форму приложение добавляет новую строку данных с помощью программного SQL. Независимо от того, какой SQL используется (интерактивный или программный), инструкция INSERT имеет один и тот же вид.

Как правило, в инструкции INSERT указывается неквалифицированное имя, определяющее вашу собственную таблицу. Чтобы вставить данные в таблицу, принадлежащую другому пользователю, необходимо указать ее полное имя. И, конечно же, необходимо иметь права на ввод данных в эту таблицу, иначе выполнение инструкции INSERT закончится неудачей. Схема безопасности SQL и права пользователя рассматриваются в главе 15, "SQL и безопасность".

Список столбцов в инструкции INSERT предназначен для установления соответствия между значениями данных, содержащимися в предложении VALUES, и столбцами, для которых эти данные предназначены. Списки значений и столбцов должны содержать одинаковое число элементов, а тип данных каждого значения должен быть совместимым с типом соответствующего столбца, иначе произойдет ошибка. Стандарт ANSI/ISO требует использования в списке столбцов неквалифицированных имен, но во многих СУБД допускается применение квалифицированных имен. Очевидно, что в именах столбцов в любом случае не может быть неоднозначности, потому что все они должны быть ссылками на столбцы целевой таблицы.

Вставка значений NULL

При добавлении в таблицу новой строки данных всем столбцам, имена которых отсутствуют в списке столбцов инструкции INSERT, автоматически присваивается значение NULL. В инструкции INSERT, с помощью которой в таблицу SALESREPS была добавлена информация о служащем Якобсене, были опущены столбцы QUOTA и MANAGER.

```
INSERT INTO SALESREPS (NAME, AGE, EMPL_NUM, SALES, TITLE,
                       HIRE_DATE, REP_OFFICE)
     VALUES ('Henry Jacobsen', 36, 111, 0.00, 'Sales Mgr',
             '2008-07-25', 13);
```

Поэтому новая строка в столбцах QUOTA и MANAGER содержит значение NULL, как показано на рис. 10.2. Значение NULL можно присвоить и явно, включив эти столбцы в список столбцов, а в списке значений задав для них ключевое слово NULL. Применение следующей инструкции INSERT приведет к тому же результату, что и ранее.

```
INSERT INTO SALESREPS (NAME, AGE, EMPL_NUM, SALES, QUOTA,
                       TITLE, MANAGER, HIRE_DATE, REP_OFFICE)
     VALUES ('Henry Jacobsen', 36, 111, 0.00, NULL,
             'Sales Mgr', NULL, '2008-07-25', 13);
```

Вставка всех столбцов

Для удобства в SQL разрешается не включать список столбцов в инструкцию INSERT. Если список столбцов опущен, он генерируется автоматически и в нем слева направо перечисляются все столбцы таблицы. При выполнении инструкции SELECT * генерируется такой же список столбцов. Пользуясь этой сокращенной формой записи, предыдущую инструкцию INSERT можно переписать следующим образом.

```
INSERT INTO SALESREPS
     VALUES (111, 'Henry Jacobsen', 36, 13, 'Sales Mgr',
             '2008-07-25', NULL, NULL, 0.00);
```

Очевидно, что если список столбцов опущен, то в списке значений *необходимо* явно указывать значения NULL (что и подтверждается в приведенном примере). Кроме того, последовательность значений данных должна в точности соответствовать порядку столбцов в таблице.

В интерактивном режиме удобно не включать в инструкцию INSERT список столбцов, так как это уменьшает длину инструкции. В случае программного SQL список столбцов должен быть задан всегда, поскольку такую программу легче читать и понимать. Кроме того, структура таблицы зачастую изменяется со временем — в нее добавляются новые столбцы или убираются старые, более не используемые. Программа с инструкцией INSERT без явного указания столбцов может вполне корректно работать месяцы и годы, а потом неожиданно сообщать об ошибке — при изменении количества столбцов или типов данных администратором базы данных.

Многострочная инструкция INSERT

Многострочная инструкция INSERT, синтаксическая диаграмма которой изображена на рис. 10.3, добавляет в целевую таблицу несколько строк (более одной). В этой разновидности инструкции INSERT значения для новых строк явно не задаются. Источником новых строк служит запрос на выборку, содержащийся внутри инструкции INSERT.

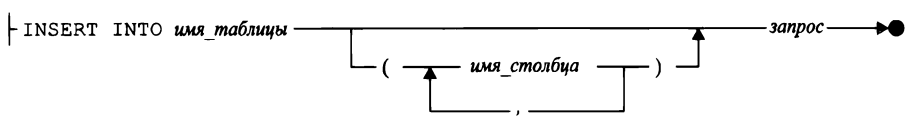


Рис. 10.3. Синтаксическая диаграмма многострочной инструкции INSERT

Процедура добавления строк со значениями, взятыми из той же базы данных, может показаться странной, но иногда она необходима. Предположим, нам требуется скопировать номера, даты и стоимости всех заказов, сделанных до 1 января 2008 года, из таблицы ORDERS в другую таблицу с именем OLDORDERS. Многострочная инструкция INSERT позволяет скопировать данные компактно и быстро.

Скопировать старые заказы в таблицу OLDORDERS.

```
INSERT INTO OLDORDERS (ORDER_NUM, ORDER_DATE, AMOUNT)
     SELECT ORDER_NUM, ORDER_DATE, AMOUNT
```

```
    FROM ORDERS
   WHERE ORDER_DATE < '2008-01-01';

9 rows inserted.
```

Хотя многострочная инструкция INSERT выглядит сложнее однострочной, в действительности она является очень простой. В ней, как и в однострочной инструкции INSERT, задаются таблица и столбцы, в которые заносятся новые элементы данных. Оставшаяся часть инструкции представляет собой запрос, извлекающий данные из таблицы ORDERS. На рис. 10.4 изображена схема выполнения инструкции INSERT, приведенной в примере. Вначале выполняется запрос к таблице ORDERS, а затем таблица результатов этого запроса построчно добавляется в таблицу OLDORDERS.

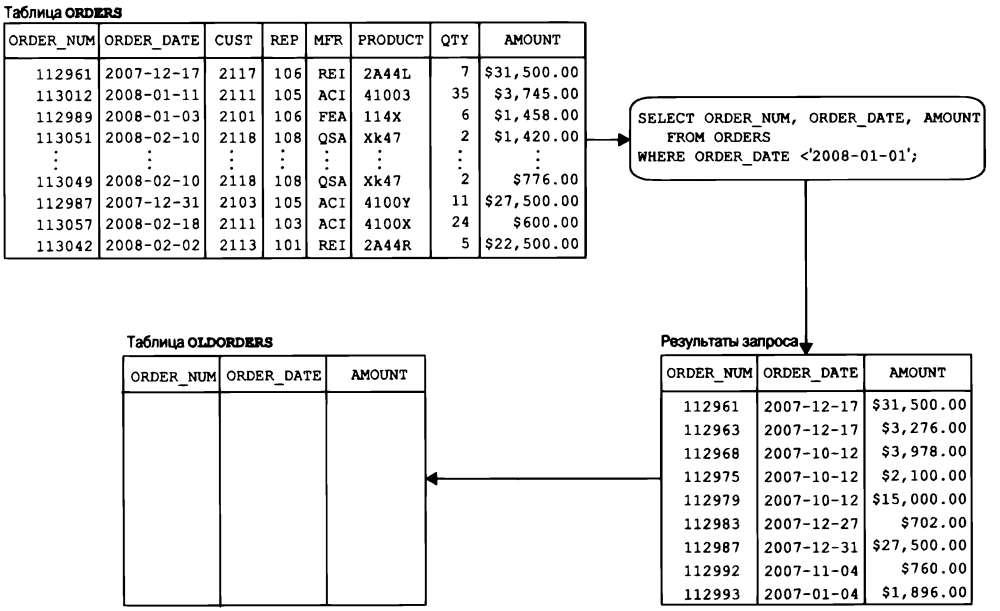


Рис. 10.4. Вставка нескольких строк

Вот еще одна ситуация, когда можно использовать многострочную инструкцию INSERT. Предположим, требуется проанализировать, что именно приобретают клиенты, и для этого необходимо просмотреть информацию о клиентах и служащих, имеющих большие заказы — стоимостью свыше $15000. Запросы, которые необходимо для этого выполнить, объединяют информацию из таблиц CUSTOMERS, SALESREPS и ORDERS. В маленькой учебной базе данных эти трехтабличные запросы будут выполняться довольно быстро, но в реальной корпоративной базе данных, содержащей тысячи строк информации, выполнение таких запросов заняло бы длительное время.

Вместо выполнения нескольких длинных трехтабличных запросов лучше создать для требуемых данных новую таблицу с именем BIGORDERS, определяемую следующим образом.

Столбец	Информация
AMOUNT	Стоимость заказа (из таблицы ORDERS)
COMPANY	Имя клиента (из таблицы CUSTOMERS)
NAME	Имя служащего (из таблицы SALESREPS)
PERF	Перевыполнение/недовыполнение плана (вычисляется по таблице SALESREPS)
MFR	Идентификатор производителя (из таблицы ORDERS)
PRODUCT	Идентификатор товара (из таблицы ORDERS)
QTY	Заказанное количество (из таблицы ORDERS)

После создания таблицы BIGORDERS ее можно заполнить данными с помощью следующей инструкции INSERT.

Загрузить в таблицу BIGORDERS данные для анализа.

```
INSERT INTO BIGORDERS (AMOUNT, COMPANY, NAME, PERF,
                       PRODUCT, MFR, QTY)
     SELECT AMOUNT, COMPANY, NAME, (SALES - QUOTA),
            PRODUCT, MFR, QTY
       FROM ORDERS, CUSTOMERS, SALESREPS
      WHERE CUST = CUST_NUM
        AND REP = EMPL_NUM
        AND AMOUNT > 15000.00;

6 rows inserted.
```

В больших базах данных выполнение такой инструкции INSERT займет некоторое время, поскольку она содержит запрос к трем таблицам. После того как выполнение инструкции завершится, в таблице BIGORDERS будет содержаться копия данных из других таблиц. Кроме того, таблица BIGORDERS не будет автоматически изменяться при добавлении в базу данных новых заказов, поэтому данные в ней могут быстро устареть. Эти факторы выглядят как недостаток. Однако последующие запросы на выборку к таблице BIGORDERS будут представлять собой запросы к одной таблице.

Следует также отметить, что каждый из этих запросов будет выполняться намного быстрее, чем при использовании трехтабличного соединения. Следовательно, копирование данных можно назвать хорошим методом проведения анализа, особенно если исходные таблицы являются большими. В данном случае таблица BIGORDERS, скорее всего, будет использоваться в качестве временной для выполнения анализа. Она будет создана и наполнена данными, которые представляют собой снимок состояния заказов в некоторый момент времени, после чего будет запущена программа анализа, после выполнения которой таблица будет опустошена или удалена.

На запрос, содержащийся внутри многострочной инструкции INSERT, стандарт SQL1 накладывает несколько логических ограничений.

- В запрос нельзя включать предложение ORDER BY. Не имеет смысла сортировать таблицу результатов запроса, поскольку она добавляется в таблицу, которая, как и все остальные, не упорядочена.
- Таблица результатов запроса должна содержать то же количество столбцов, что и в списке столбцов в инструкции INSERT (или во всей целевой таблице полностью, если список столбцов опущен), а типы данных соответствующих столбцов таблицы результатов запроса и целевой таблицы должны быть совместимыми.

Программы пакетной загрузки

Часто возникает необходимость загрузить в базу данных информацию из другого компьютера или из файла, в который она была собрана из различных источников. Для загрузки данных в таблицу можно написать программу с циклом, в котором из файла считывается одна запись, а затем с помощью однострочной инструкции INSERT эта запись добавляется в таблицу. Однако накладные расходы, связанные с циклическим выполнением однострочной инструкции INSERT, могут оказаться очень высокими. Если принять, что в типичном случае ввод одной строки занимает полсекунды, то для интерактивного режима это, по-видимому, допустимое быстродействие. Но если необходимо загрузить 50000 строк данных, то такое быстродействие неприемлемо. Загрузка данных в этом случае заняла бы свыше шести часов.

По этой причине в большинстве коммерческих СУБД имеются средства пакетной загрузки, которые с высокой скоростью загружают данные из файла в таблицу. В стандарте SQL этот тип загрузки не упоминается, и обычно он осуществляется автономными служебными утилитами без участия SQL. Утилиты различных поставщиков СУБД немного отличаются набором функций, команд и свойств.

Когда SQL используется в прикладной программе, для эффективной вставки большого количества данных в базу данных зачастую применяется другая методика. Стандартная программная инструкция INSERT вставляет одну строку данных, как и интерактивная однострочная инструкция INSERT из предыдущего примера. Но многие коммерческие СУБД допускают передачу данных из двух и более строк (зачастую до сотен строк) единой пакетной инструкции INSERT. Все такие данные должны быть новыми строками одной целевой таблицы инструкции INSERT, указанной в предложении INTO.

Выполнение пакетной инструкции INSERT для 100 строк данных эквивалентно, с точки зрения состояния базы данных, выполнению 100 отдельных однострочных инструкций INSERT. Однако обычно такая пакетная инструкция существенно эффективнее, так как включает только один вызов СУБД. При загрузке тысяч строк величина выигрыша во времени может оказаться более чем на порядок.

Удаление существующих данных

Удалять ту или иную строку из реляционной базы данных приходится тогда, когда объект, представляемый этой строкой, исчезает из внешнего мира. Скажем, в случае учебной базы возможны следующие варианты.

- Если клиент отменяет заказ, необходимо удалить соответствующую строку из таблицы ORDERS.
- Если служащий увольняется из компании, должна быть удалена соответствующая строка из таблицы SALESREPS.
- Если ликвидируется офис, необходимо удалить соответствующую строку из таблицы OFFICES; в случае, когда служащие этого офиса увольняются, их строки в таблице SALESREPS также должны быть удалены; если служащие переводятся в другой офис, то соответствующие значения в столбце OFFICE необходимо обновить.

Во всех приведенных примерах новая строка удаляется для того, чтобы база данных оставалась точной моделью реального мира. Наименьшей единицей информации, которую можно удалить из реляционной базы данных, является одна строка.

Инструкция DELETE

Инструкция DELETE, синтаксическая диаграмма которой изображена на рис. 10.5, удаляет выбранные записи из одной таблицы. В предложении FROM указывается таблица, содержащая строки, которые требуется удалить. В предложении WHERE указывается критерий отбора строк, которые должны быть удалены.

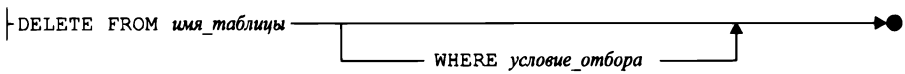


Рис. 10.5. Синтаксическая диаграмма инструкции DELETE

Предположим, что недавно принятый на работу Генри Якобсен решил уволиться из компании. Вот инструкция DELETE, удаляющая относящуюся к служащему строку из таблицы SALESREPS.

Удалить информацию о Генри Якобсене из базы данных.

```
DELETE FROM SALESREPS
 WHERE NAME = 'Henry Jacobsen';

1 row deleted.
```

В приведенном выше примере в предложении WHERE определена одна строка таблицы SALESREPS, которая будет удалена из этой таблицы. Предложение WHERE имеет знакомый вид — это то же самое предложение, которое необходимо включить в инструкцию SELECT для *выборки* из таблицы этой же строки. Условия отбора, которые можно задать в предложении WHERE инструкции DELETE, полностью

совпадают с условиями отбора, доступными в одноименном предложении инструкции SELECT (главы 6–9).

Вспомним, что в предложении WHERE инструкции SELECT условия отбора могут определять как одну строку, так и набор строк — в зависимости от конкретного условия. То же самое справедливо и для предложения WHERE инструкции DELETE. Предположим, например, что клиент служащего Якобсена, компания InterCorp (идентификатор клиента 2126), отменил все свои заказы. Вот инструкция DELETE, удаляющая указанные заказы из таблицы ORDERS.

Удалить все заказы компании InterCorp (идентификатор клиента 2126).

```
DELETE FROM ORDERS
 WHERE CUST = 2126;

2 rows deleted.
```

В данном примере предложение WHERE выбирает несколько строк таблицы ORDERS, которые затем удаляются из нее. Все строки таблицы ORDERS последовательно проверяются на соответствие условию отбора. Строки, в которых условие отбора имеет значение TRUE, удаляются, а строки, в которых условие отбора имеет значение FALSE или NULL, сохраняются. Поскольку инструкция DELETE этого типа осуществляет в таблице поиск удаляемых строк, ее иногда называют *поисковой*, в отличие от инструкции DELETE другого типа, всегда удаляющей одну строку и называемой *позиционной*. Позиционная инструкция DELETE применяется только в программном SQL и описана в главе 17, "Встроенный SQL".

Ниже приведен ряд дополнительных примеров поисковых инструкций DELETE.

Удалить все заказы, сделанные до 15 ноября 2007 года.

```
DELETE FROM ORDERS
 WHERE ORDER_DATE < '2007-11-15';

5 rows deleted.
```

Удалить данные о всех клиентах, обслуживаемых Биллом Адамсом, Мери Джонс и Дэном Робертсом (идентификаторы служащих 105, 109 и 101).

```
DELETE FROM CUSTOMERS
 WHERE CUST_REP IN (105, 109, 101);

7 rows deleted.
```

Удалить данные о всех служащих, принятых на работу до июля 2006 года и еще не имеющих личного плана.

```
DELETE FROM SALESREPS
 WHERE HIRE_DATE < '2006-07-01'
   AND QUOTA IS NULL;

0 rows deleted.
```

Удаление всех строк

Хотя предложение WHERE в инструкции DELETE является необязательным, оно присутствует почти всегда. Если же оно отсутствует, то удаляются все строки целевой таблицы.

Удалить все заказы.

```
DELETE FROM ORDERS;

30 rows deleted.
```

Хотя в результате выполнения приведенной инструкции DELETE таблица ORDERS становится пустой, из базы данных она не удаляется. Определение таблицы ORDERS и ее столбцов остается в базе данных. Таблица по-прежнему существует, и в нее по-прежнему можно добавлять новые строки с помощью инструкции INSERT. Чтобы удалить из базы данных определение таблицы, необходимо использовать инструкцию DROP TABLE (рассматривается в главе 13, "Создание базы данных").

Такая инструкция DELETE несет в себе потенциальную угрозу удаления необходимых строк, поэтому всегда следует задавать условие отбора и обращать внимание на то, отбирает ли оно действительно ненужные строки. Желательно вначале проверить предложение WHERE в интерактивном режиме в составе инструкции SELECT и отобразить выбранные строки на экране. Убедившись, что это именно те строки, которые требуется удалить, можно использовать предложение WHERE в инструкции DELETE.

Инструкция DELETE с подзапросом*

Инструкции DELETE с простыми условиями отбора, рассмотренные в предыдущих примерах, отбирают строки для удаления исключительно на основании содержимого этих строк. Но иногда отбор строк необходимо производить, опираясь на данные из других таблиц. Предположим, вы хотите удалить все заказы, принятые служащей Сью Смит. Не зная ее идентификатора, вы не сможете найти ее заказы, пользуясь одной только таблицей ORDERS. Чтобы найти эти заказы, можно было бы воспользоваться запросом к двум таблицам.

Найти заказы, принятые Сью Смит.

```
SELECT ORDER_NUM, AMOUNT
  FROM ORDERS, SALESREPS
 WHERE REP = EMPL_NUM
   AND NAME = 'Sue Smith';

ORDER_NUM        AMOUNT
---------    ----------
   112979    $15,000.00
   113065     $2,130.00
   112993     $1,896.00
   113048     $3,750.00
```

Однако в инструкции DELETE запрещено использовать соединение таблиц. Инструкция DELETE с параллельным удалением из двух таблиц является неправильной.

```
DELETE FROM ORDERS, SALESREPS
 WHERE REP = EMPL_NUM
   AND NAME = 'Sue Smith';

Error: More than one table specified in FROM clause
```

Чтобы выполнить поставленную задачу, необходимо использовать условие отбора с *подзапросом*. Ниже показана правильная инструкция DELETE, выполняющая поставленную задачу.

Удалить все заказы, принятые Сью Смит.

```
DELETE FROM ORDERS
 WHERE REP = (SELECT EMPL_NUM
                FROM SALESREPS
               WHERE NAME = 'Sue Smith');

4 rows deleted.
```

Подзапрос находит идентификатор Сью Смит, а затем предложение WHERE отбирает заказы с данным идентификатором. Как видно из этого примера, подзапросы в инструкции DELETE играют важную роль, поскольку они позволяют удалять строки, основываясь на информации, содержащейся в других таблицах. Вот еще два примера инструкции DELETE, в которых используются условия отбора с подзапросом.

Удалить данные о всех клиентах, обслуживаемых служащими, у которых фактический объем продаж меньше 80 процентов их плана.

```
DELETE FROM CUSTOMERS
 WHERE CUST_REP IN (SELECT EMPL_NUM
                      FROM SALESREPS
                     WHERE SALES < (.8 * QUOTA));

2 rows deleted.
```

Удалить данные о всех служащих, у которых сумма текущих заказов меньше двух процентов их личного плана.

```
DELETE FROM SALESREPS
 WHERE (.02 * QUOTA) > (SELECT SUM(AMOUNT)
                          FROM ORDERS
                         WHERE REP = EMPL_NUM);

1 row deleted.
```

Подзапросы в предложении WHERE могут иметь несколько уровней вложенности. Они могут также содержать внешние ссылки на целевую таблицу инструкции DELETE. В этом случае предложение FROM инструкции DELETE играет такую же роль, как и предложение FROM инструкции SELECT. Вот пример запроса на удаление, в котором требуется использовать подзапрос, содержащий внешнюю ссылку.

Удалить данные о всех клиентах, которые не делали заказов с 10 ноября 2007 года.

```
DELETE FROM CUSTOMERS
 WHERE NOT EXISTS (SELECT *
```

```
                FROM ORDERS
               WHERE CUST = CUST_NUM
                 AND ORDER_DATE > '2007-11-10')

16 rows deleted.
```

Приведенная выше инструкция DELETE выполняется таким образом: каждая строка таблицы CUSTOMERS по очереди проверяется на соответствие условию отбора. Для каждого клиента подзапрос отбирает все заказы, размещенные этим клиентом после указанной даты. Ссылка на столбец CUST_NUM в подзапросе является внешней ссылкой на идентификатор клиента той строки таблицы CUSTOMERS, которая проверяется инструкцией DELETE в настоящий момент. Подзапрос в данном примере является коррелированным (такие запросы рассматривались в главе 9, "Подзапросы и выражения с запросами").

В подзапросе инструкции DELETE внешние ссылки встречаются часто, поскольку они реализуют соединение таблицы (или таблиц) подзапроса и целевой таблицы инструкции DELETE. Стандарт SQL указывает, что инструкция DELETE должна рассматривать такой подзапрос как применяемый ко всей целевой таблице, до удаления из нее каких бы то ни было строк. Это привносит дополнительные накладные расходы, связанные с необходимостью более аккуратной обработки подзапросов и удаления строк, но зато поведение данной инструкции оказывается точно определено стандартом.

Обновление существующих данных

Обновлять информацию, хранимую в базе данных, требуется тогда, когда соответствующие изменения происходят во внешнем мире. В случае учебной базы данных возможны следующие варианты.

- Если клиент изменяет количество заказанного товара, в соответствующей строке таблицы ORDERS должен быть обновлен столбец QTY.
- Если руководитель переходит из одного офиса в другой, столбец MGR таблицы OFFICES и столбец REP_OFFICE таблицы SALESREPS необходимо обновить, чтобы отобразить новое назначение.
- Если личные планы продаж в нью-йоркском офисе увеличиваются на пять процентов, значения столбца QUOTA в соответствующих строках таблицы SALESREPS должны быть обновлены.

Во всех приведенных примерах значения данных обновляются для того, чтобы база данных оставалась точной моделью реального мира. Наименьшей единицей информации, которую можно обновить в реляционной базе данных, является значение одного столбца в одной строке.

Инструкция UPDATE

Инструкция UPDATE, синтаксическая диаграмма которой изображена на рис. 10.6, обновляет значения одного или нескольких столбцов в выбранных стро-

ках одной таблицы. В инструкции указывается целевая таблица, которая должна быть модифицирована, при этом пользователь должен иметь права на обновление как таблицы, так и каждого конкретного столбца. Предложение WHERE отбирает строки таблицы, подлежащие обновлению. В предложении SET указывается, какие столбцы должны быть обновлены, и для них задаются новые значения.

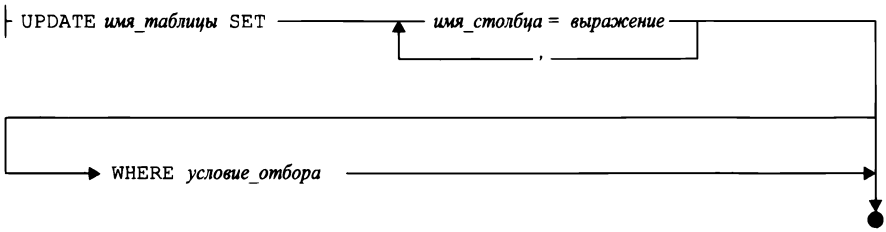


Рис. 10.6. Синтаксическая диаграмма инструкции UPDATE

Вот простая инструкция UPDATE, которая изменяет лимит кредита клиента и закрепляет последнего за новым служащим.

Увеличить предельный кредит для компании Acme Manufacturing до $60 000 и закрепить ее за Мери Джонс (идентификатор 109).

```
UPDATE CUSTOMERS
   SET CREDIT_LIMIT = 60000.00, CUST_REP = 109
 WHERE COMPANY = 'Acme Mfg.';

1 row updated.
```

В этом примере в предложении WHERE определена одна строка таблицы CUSTOMERS, а предложение SET присваивает новые значения двум столбцам этой строки. Условия отбора, которые могут быть заданы в предложении WHERE инструкции UPDATE, в точности соответствуют условиям отбора, доступным в инструкциях SELECT и DELETE.

Как и инструкция DELETE, инструкция UPDATE может одновременно обновить несколько строк, соответствующих условию отбора.

Перевести всех служащих из чикагского офиса (идентификатор 12) в нью-йоркский офис (идентификатор 11) и понизить их личные планы на десять процентов.

```
UPDATE SALESREPS
   SET REP_OFFICE = 11, QUOTA = .9 * QUOTA
 WHERE REP_OFFICE = 12;

3 rows updated.
```

В данном примере предложение WHERE отбирает несколько строк таблицы SALESREPS, и в них обновляются столбцы OFFICE и QUOTA. Инструкция UPDATE выполняется таким образом: все строки таблицы SALESREPS по очереди проверяются на соответствие условию отбора. Строки, для которых условие отбора выполняется (результат проверки имеет значение TRUE), обновляются, а строки, для которых условие не выполняется (результат проверки имеет значение FALSE или NULL), не обновляются. Поскольку инструкция UPDATE данного типа производит

в таблице поиск строк, она иногда называется *поисковой*. Инструкция UPDATE другого типа, всегда обновляющая одну строку, называется *позиционной*. Позиционная инструкция UPDATE применяется только в программном SQL и рассматривается в главе 17, "Встроенный SQL".

Вот несколько дополнительных примеров применения поисковых инструкций UPDATE.

Перевести всех клиентов, обслуживаемых служащими с идентификаторами 105, 106 и 107, к служащему с идентификатором 102.

```
UPDATE CUSTOMERS
   SET CUST_REP = 102
 WHERE CUST_REP IN (105, 106, 107);

5 rows updated.
```

Установить личный план продаж в $100 000 всем служащим, не имеющим в настоящий момент плана.

```
UPDATE SALESREPS
   SET QUOTA = 100000.00
 WHERE QUOTA IS NULL;

1 row updated.
```

Предложение SET в инструкции UPDATE представляет собой список операций присваивания, отделяемых друг от друга запятыми. В каждой операции идентифицируется целевой столбец, который должен обновляться, и определяется новое значение для этого столбца. Каждый целевой столбец должен встречаться в списке только один раз; не должно быть двух операций присваивания для одного и того же целевого столбца. Согласно стандарту ANSI/ISO, для целевых столбцов необходимо использовать простые имена, но некоторые СУБД допускают использование полных имен столбцов. Так как эти имена являются ссылками на столбцы целевой таблицы, то в любом случае неоднозначность невозможна.

Выражение в операции присваивания может быть любым допустимым SQL-выражением, результирующее значение которого имеет тип данных, соответствующий целевому столбцу. Необходимо, чтобы значение выражения вычислялось на основе значений строки, которая в данный момент обновляется в целевой таблице. В большинстве реализаций СУБД оно не может включать в себя какие-либо статистические функции или подзапросы.

Если выражение в операции присваивания содержит ссылку на один из столбцов целевой таблицы, то для вычисления выражения используется значение этого столбца в текущей строке, которое было *перед* обновлением. То же самое справедливо для ссылок на столбцы в предложении WHERE. В качестве примера рассмотрим следующую (несколько надуманную) инструкцию UPDATE:

```
UPDATE OFFICES
   SET QUOTA = 400000.00, SALES = QUOTA
 WHERE QUOTA < 400000.00;
```

До обновления личный план (QUOTA) Билла Адамса составлял $350000, а объем продаж (SALES) — $367911. После обновления объем продаж (SALES) в его строке стал равен $350000, а не $400000. Таким образом, порядок операций присваивания в предложении SET не играет роли; он может быть любым.

Обновление всех строк

Предложение WHERE в инструкции UPDATE является необязательным. Если оно опущено, то обновляются *все* строки целевой таблицы, например, как в следующем запросе.

Увеличить все личные планы на пять процентов.

```
UPDATE SALESREPS
   SET QUOTA = 1.05 * QUOTA;

10 rows updated.
```

В отличие от инструкции DELETE, в которой предложение WHERE практически никогда не опускается, инструкция UPDATE и без предложения WHERE выполняет полезную функцию. В основном, она применяется для обновления всей таблицы, что и было продемонстрировано в предыдущем примере.

Инструкция UPDATE с подзапросом*

В инструкции UPDATE, так же как и в инструкции DELETE, подзапросы могут играть важную роль, поскольку они дают возможность отбирать строки для обновления, опираясь на информацию из других таблиц. Ниже приведены примеры инструкций UPDATE, в которых используются подзапросы.

Увеличить на $5000 лимит кредита для тех клиентов, которые сделали заказ на сумму более $25000.

```
UPDATE CUSTOMERS
   SET CREDIT_LIMIT = CREDIT_LIMIT + 5000.00
 WHERE CUST_NUM IN (SELECT DISTINCT CUST
                      FROM ORDERS
                     WHERE AMOUNT > 25000.00);

4 rows updated.
```

Переназначить клиентов, обслуживаемых служащими, чей объем продаж меньше 80 процентов их личного плана, служащему с идентификатором 105.

```
UPDATE CUSTOMERS
   SET CUST_REP = 105
 WHERE CUST_REP IN (SELECT EMPL_NUM
                      FROM SALESREPS
                     WHERE SALES < (.8 * QUOTA));

2 rows updated.
```

Всех служащих, обслуживающих более трех клиентов, подчинить непосредственно Сэму Кларку (идентификатор 106).

```
UPDATE SALESREPS
   SET MANAGER = 106
 WHERE 3 < (SELECT COUNT(*)
              FROM CUSTOMERS
             WHERE CUST_REP = EMPL_NUM);

1 row updated.
```

Подзапросы в предложении WHERE инструкции UPDATE, так же как и в инструкции DELETE, могут иметь любой уровень вложенности и содержать внешние ссылки на целевую таблицу инструкции UPDATE. Имя столбца EMPL_NUM в подзапросе предыдущего примера является такой внешней ссылкой; она относится к столбцу EMPL_NUM той строки таблицы SALESREPS, которая проверяется в настоящий момент инструкцией UPDATE. Подзапрос в этом примере является коррелированным (такие запросы рассматривались в главе 9, "Подзапросы и выражения с запросами").

Внешние ссылки часто встречаются в подзапросах инструкции UPDATE, поскольку они реализуют соединение таблицы (или таблиц) подзапроса и целевой таблицы инструкции UPDATE. Стандарт SQL указывает, что ссылка на целевую таблицу в подзапросе вычисляется с использованием данных, имевшихся в целевой таблице до обновления.

Резюме

В настоящей главе рассматривались инструкции SQL, которые используются для изменения содержимого базы данных.

- Однострочная инструкция INSERT добавляет в таблицу одну строку данных. Значения новой строки задаются в инструкции в виде констант.
- Многострочная инструкция INSERT добавляет в таблицу нуль или более строк данных. Значения новых строк берутся из запроса, являющегося частью инструкции INSERT.
- Инструкция DELETE удаляет из таблицы нуль или более строк данных. Удаляемые строки задаются с помощью условия отбора.
- Инструкция UPDATE обновляет значения одного или более столбцов в нуле или более строках таблицы. Обновляемые строки задаются с помощью условия отбора. Обновляемые столбцы и выражения, определяющие новые значения, задаются в инструкции UPDATE.

В отличие от инструкции SELECT, которая может работать с несколькими таблицами, инструкции INSERT, DELETE и UPDATE обращаются только к одной таблице. Условие отбора в инструкциях DELETE и UPDATE имеет тот же вид, что и в инструкции SELECT.

ГЛАВА 11

Целостность данных

Термин *целостность данных* относится к правильности и полноте информации, содержащейся в базе данных. При изменении содержимого базы данных с помощью инструкций INSERT, DELETE или UPDATE может произойти нарушение целостности содержащихся в ней данных.

- В базу могут быть внесены неправильные данные, скажем, заказ, в котором указан несуществующий товар.
- Имеющимся данным, в результате изменения, могут быть присвоены некорректные значения, например служащий может быть назначен в несуществующий офис.
- Изменения, внесенные в базу данных, могут быть утеряны из-за системной ошибки или сбоя в электропитании либо они могут быть внесены лишь частично; например, заказ на товар может быть добавлен без учета изменения количества товара, имеющегося на складе.

Одной из важнейших задач реляционной СУБД является поддержка целостности данных на максимально возможном уровне. В настоящей главе рассматриваются особенности языка SQL, помогающие СУБД выполнять эту задачу.

Условия целостности данных

Для сохранения непротиворечивости и правильности хранимой информации в реляционных СУБД устанавливается одно или несколько *условий (ограничений) целостности данных*. Эти условия определяют, какие значения могут быть записаны в базу данных в результате добавления или обновления данных. Как правило, в реляционных базах данных используются следующие условия целостности данных.

- **Обязательное наличие данных**. Некоторые столбцы в базе данных должны содержать значения в каждой строке; в таких столбцах не могут содержаться значения NULL или не содержаться никакие значения. На-

пример, в учебной базе данных для каждого заказа должен существовать соответствующий клиент, сделавший этот заказ. Поэтому столбец CUST в таблице ORDERS является обязательным. Можно указать СУБД, что запись значения NULL в такие столбцы недопустима.

- **Условие на значение**. У каждого столбца в базе данных есть свой *домен*, т.е. набор значений, которые допускается хранить в данном столбце. В учебной базе данных заказы нумеруются начиная с числа 100001, поэтому доменом столбца ORDER_NUM являются положительные целые числа, большие 100000. Аналогично идентификаторы служащих в столбце EMPL_NUM должны находиться в диапазоне от 101 до 999. Можно указать СУБД, что запись значений, не входящих в определенный диапазон, в такие столбцы недопустима.
- **Логическая целостность данных**. Первичный ключ таблицы должен в каждой строке иметь уникальное значение, отличное от значений во всех остальных строках. Например, каждая строка таблицы PRODUCTS имеет уникальную комбинацию значений в столбцах MFR_ID и PRODUCT_ID, которая однозначно идентифицирует товар, представляемый данной строкой. Повторяющиеся значения в этих столбцах недопустимы, поскольку тогда база данных не сможет отличить один товар от другого. Можно потребовать от СУБД обеспечения логической целостности данных.
- **Ссылочная целостность**. В реляционной базе данных каждая строка дочерней таблицы связана с помощью внешнего ключа со строкой родительской таблицы, содержащей первичный ключ, значение которого равно значению внешнего ключа. В учебной базе данных значение столбца REP_OFFICE таблицы SALESREPS связывает служащего с офисом, в котором он работает. Столбец REP_OFFICE *обязан* содержать значение из столбца OFFICE таблицы OFFICES; в противном случае служащий будет закреплен за несуществующим офисом. Можно указать СУБД, чтобы она обеспечивала соответствующее ограничение на значения внешнего ключа.
- **Другие соотношения между данными**. Моделируемая базой данных ситуация реального мира зачастую накладывает собственные ограничения на корректность данных, которые могут храниться в базе данных. Например, в нашей учебной базе данных вице-президент по продажам может захотеть, чтобы план продаж каждого офиса не превышал суммарных планов продаж сотрудников этого офиса. СУБД можно предупредить о том, что при внесении изменений в планы продаж офиса и служащих должно выполняться указанное выше ограничение.
- **Бизнес-правила**. Обновление информации в базе данных может быть ограничено бизнес-правилами, которым подчиняются сделки, представляемые подобными обновлениями. Например, компания, использующая учебную базу данных, может установить бизнес-правило, запрещающее принимать заказы на товар в количествах, превышающих количество товара на складе. Можно указать СУБД, что следует проверять каждую но-

вую строку, добавляемую в таблицу ORDERS, и убеждаться, что значение в столбце QTY не нарушает установленное бизнес-правило.

- **Непротиворечивость**. Многие реальные деловые операции вызывают в базе данных несколько изменений одновременно. Например, операция "прием заказа" может включать в себя добавление строки в таблицу ORDERS, увеличение значения столбца SALES в таблице SALESREPS для служащего, принявшего заказ, и увеличение значения столбца SALES в таблице OFFICES для офиса, за которым закреплен этот служащий. Одна инструкция INSERT и две инструкции UPDATE — *все* они должны быть выполнены для того, чтобы база данных осталась в правильном, непротиворечивом, состоянии. Можно указать СУБД, что следует обеспечивать непротиворечивость изменяемых данных.

В стандарте ANSI/ISO определены наиболее простые условия целостности данных. Например, условие обязательности данных поддерживается стандартом ANSI/ISO и одинаковым образом реализовано почти во всех коммерческих СУБД. Более сложные условия, в частности, бизнес-правила, не упоминаются в стандарте, и среди методов их реализации в различных СУБД наблюдается большое разнообразие. В настоящей главе рассматриваются средства SQL, поддерживающие первые пять описанных условий целостности данных. Механизм обработки транзакций в SQL, который обеспечивает выполнение условия согласованности данных, будет рассмотрен в главе 12, "Обработка транзакций".

Обязательность данных

Это, наиболее простое, условие целостности данных требует, чтобы некоторые столбцы не содержали значений NULL. Стандарт ANSI/ISO и большинство коммерческих СУБД поддерживают выполнение подобного условия, позволяя пользователю при создании таблицы объявить, что некоторые столбцы не могут содержать значений NULL. Само условие задается как часть инструкции CREATE TABLE в виде ограничения NOT NULL.

Если на столбец наложено ограничение NOT NULL, то для выполнения этого условия СУБД обеспечивает следующее.

- Ни в одной инструкции INSERT, добавляющей в таблицу строку или строки, нельзя указывать значение NULL для этого столбца; попытка добавить строку, содержащую (явно или неявно) значение NULL для такого столбца, вызовет ошибку.
- Ни в одной инструкции UPDATE, обновляющей столбец, нельзя присваивать столбцу значение NULL; попытка обновить такой столбец, присвоив ему значение NULL, вызовет ошибку.

Недостатком условия обязательного наличия данных является то, что его необходимо задавать при создании таблицы. Как правило, для таблицы, созданной ранее, отменить это условие уже нельзя. Но это не очень большой недостаток, по-

скольку обычно уже при создании таблицы бывает ясно, какой столбец может содержать значения NULL, а какой — нет.

Невозможность наложения ограничения NOT NULL на уже существующую таблицу зачастую является следствием способа реализации значений NULL внутри СУБД. Обычно СУБД резервирует в каждой хранимой строке по одному дополнительному байту на каждый столбец, в котором значения NULL допустимы. Дополнительный байт служит "индикатором" значения NULL, и если в столбце содержится NULL, то этому байту присваивается заранее установленное значение. Если же для столбца определено ограничение NOT NULL, то байт индикатора отсутствует, что позволяет экономить дисковую память. Чтобы динамически отменить это ограничение, требуется на лету реконфигурировать хранимые на диске записи, что в больших базах данных непрактично и весьма накладно.

Условия на значения

Наиболее рудиментарная поддержка ограничений корректных значений, которые могут находиться в столбцах, — это применение типов данных, описанных в стандарте SQL. При создании таблицы каждому столбцу назначается определенный тип данных, и СУБД следит за тем, чтобы в столбец вводились данные только этого типа. Например, для столбца EMPL_NUM таблицы SALESREPS определен тип данных INTEGER, и СУБД выдаст ошибку, если инструкция INSERT или UPDATE попытается ввести в столбец строку символов или десятичное число.

Однако в первом стандарте (SQL1) и многих ранних коммерческих СУБД не имелось способа ограничить данные в столбце конкретными значениями. СУБД без колебаний добавит в таблицу SALESREPS строку с идентификатором служащего 12345, хотя в учебной базе данных идентификаторы служащих должны состоять только из трех цифр. Аналогично дата "25 декабря" без возражений будет принята в качестве даты приема на работу, хотя на Рождество в компании был выходной день.

До принятия ограничений на значения и доменов в стандарте SQL2 некоторые коммерческие СУБД реализовывали расширенные возможности по проверке допустимости вводимых данных. Для обратной совместимости многие из них до сих пор поддерживаются производителями СУБД. Так, в СУБД DB2 за каждой таблицей может быть закреплена соответствующая *процедура проверки данных* — программа, написанная пользователем и проверяющая допустимость значений данных. Эта процедура в DB2 вызывается всякий раз, когда инструкция SQL пытается изменить или добавить строку таблицы. В процедуру проверки передаются "предлагаемые" значения столбцов этой строки, а сама процедура проверяет данные и посредством возвращаемого значения сообщает, являются ли они приемлемыми. Процедура проверки представляет собой обычную программу (написанную, например, на ассемблере S/370 или на PL/I), поэтому она может выполнять любые проверки значений данных, включая проверку на принадлежность диапазону или на внутреннюю непротиворечивость записи в таблице. В то же время процедура проверки *не имеет* доступа к базе данных, поэтому ее нельзя использовать для проверки уникальности значений или отношений "внешний ключ–первичный ключ".

SQL Server также предоставляет возможность проверки допустимости данных, позволяя пользователю создавать *правила*, определяющие, какие данные можно вводить в указанный столбец. SQL Server применяет созданное правило всякий раз, когда инструкция INSERT или UPDATE обращается к таблице, содержащей этот столбец. В отличие от процедуры проверки данных в СУБД DB2, правила в SQL Server создаются с использованием диалекта Transact-SQL. Например, ниже приведена инструкция Transact-SQL, устанавливающая правило для столбца QUOTA таблицы SALESREPS.

```
CREATE RULE QUOTA_LIMIT
    AS @VALUE BETWEEN 0.00 AND 500000.00;
```

Это правило запрещает вставлять в столбец QUOTA отрицательные числа, а также числа, большие 500000. Как видно из примера, тому или иному правилу в SQL Server можно присвоить имя (в данном случае это QUOTA_LIMIT). Однако, как и процедуры проверки данных в DB2, правила в SQL Server не имеют доступа к столбцам или другим объектам базы данных.

Начиная с SQL2 стандарт SQL обеспечивает расширенную поддержку проверки данных на правильность, предоставляя два дополнительных средства — ограничения на значения столбца и домены. Оба этих средства позволяют разработчику базы данных указать СУБД, какие данные в столбце считать правильными, а какие — нет. Ограничение на значения столбца служит для проверки правильности значений отдельного столбца. С помощью домена можно задать условие проверки один раз, а затем многократно применять его для столбцов, область значений которых одинакова.

Ограничения на значения столбца

В SQL *ограничение на значения столбца* аналогично условию отбора в предложении WHERE, которое возвращает значение TRUE или FALSE. Если для столбца задано ограничение, то при каждом добавлении новой строки или обновлении старой СУБД автоматически проверяет, выполняется ли ограничение для значения в этом столбце. Если оно не выполняется, то инструкция INSERT или UPDATE завершается ошибкой. Ограничение на значение столбца задается при определении столбца в инструкции CREATE TABLE, рассматриваемой в главе 13, "Создание базы данных".

Рассмотрим фрагмент инструкции CREATE TABLE, создающей таблицу SALESREPS с тремя дополнительными ограничениями.

```
CREATE TABLE SALESREPS
   (EMPL_NUM INTEGER NOT NULL
             CHECK (EMPL_NUM BETWEEN 101 AND 199),
         AGE INTEGER
             CHECK (AGE >= 21),
          .
          .
          .
       QUOTA DECIMAL(9,2)
             CHECK (QUOTA >= 0.0),
          .
          .
          .
```

Первое ограничение (для столбца EMPL_NUM) требует, чтобы идентификаторы служащих представляли собой трехзначные числа в диапазоне от 101 до 199. Второе ограничение (для столбца AGE) запрещает нанимать на работу людей младше 21 года. Третье ограничение (для столбца QUOTA) не позволяет назначать служащему плановый объем продаж меньше $0.00.

Эти три ограничения являются очень простыми, но наглядно демонстрируют возможности стандарта SQL. Вообще говоря, после ключевого слова CHECK в круглых скобках может стоять любое условие отбора, имеющее смысл в контексте определения данного столбца. В условии отбора могут сравниваться предполагаемые значения двух различных столбцов или даже может выполняться сравнение вносимых данных с другими данными из базы. Более подробно эта возможность описывается ниже в этой главе.

Домены

Домен в SQL обобщает понятие ограничения на значения столбца и позволяет применять одно и то же ограничение для различных столбцов в базе данных. Домен представляет собой множество допустимых значений. Хотя домены определены в стандарте SQL начиная с SQL2, их поддержка в текущих реализациях SQL достаточно редка. На момент написания книги ни в одной из СУБД DB2, Oracle, SQL Server и MySQL не было поддержки доменов, хотя в некоторых СУБД и имеются некоторые достаточно схожие расширения SQL, такие как инструкция CREATE TYPE в Oracle.

Домен создается с помощью инструкции CREATE DOMAIN, которая описана в главе 13, "Создание базы данных". Как и в случае определения ограничения на значение столбца, для указания диапазона допустимых значений используется условие отбора. Вот пример инструкции, создающей домен VALID_EMPLOYEE_ID, который включает все множество допустимых значений идентификаторов служащих.

```
CREATE DOMAIN VALID_EMPLOYEE_ID INTEGER
 CHECK (VALUE BETWEEN 101 AND 199);
```

После создания домена его можно применять в качестве типа данных при определении столбцов таблиц базы данных. Теперь инструкцию CREATE TABLE для таблицы SALESREPS можно записать так.

```
CREATE TABLE SALESREPS
   (EMPL_NUM VALID_EMPLOYEE_ID,
         AGE INTEGER
             CHECK (AGE >= 21),
          .
          .
          .
       QUOTA DECIMAL(9,2)
             CHECK (QUOTA >= 0.0),
          .
          .
          .
```

Преимущество домена заключается в том, что если, например, какой-нибудь столбец другой таблицы также содержит идентификаторы служащих, то при определении этого столбца можно еще раз воспользоваться этим доменом.

```
CREATE TABLE OFFICES
   (OFFICE INTEGER NOT NULL,
      CITY VARCHAR(15) NOT NULL,
    REGION VARCHAR(10) NOT NULL,
       MGR VALID_EMPLOYEE_ID,
    TARGET DECIMAL(9,2),
     SALES DECIMAL(9,2) NOT NULL,
        .
        .
        .
```

Еще одним очевидным преимуществом доменов является то, что все определения "допустимых данных" (как допустимые идентификаторы служащих в показанном примере) хранятся в одном месте базы данных. Если определение домена впоследствии потребуется изменить (к примеру, компания разрастется и возникнет необходимость расширить диапазон идентификаторов до 299), то намного проще сделать это один раз, чем менять определения столбцов во всей базе данных. В больших базах данных уровня предприятия могут иметься определения сотен доменов, и преимущества доменов при внесении изменений окажутся существенны.

Целостность таблицы

Каждая строка таблицы должна иметь уникальное значение первичного ключа, иначе база данных потеряет свою целостность и перестанет быть адекватной моделью внешнего мира. Например, если бы две строки таблицы SALESREPS имели в столбце EMPL_NUM значение 106, то невозможно было бы сказать, какая из них относится к реальному субъекту — Биллу Адамсу, имеющему идентификатор 106. По этой причине требование, чтобы первичные ключи имели уникальные значения, называется условием *целостности таблицы*.

В ранних коммерческих СУБД первичные ключи отсутствовали, но сейчас они повсеместно распространены. В DB2 первичные ключи появились в 1988 году, а в стандарт ANSI/ISO SQL они были добавлены в виде промежуточного изменения, внесенного перед появлением полного стандарта SQL2. Первичные ключи указываются как часть инструкции CREATE TABLE или ALTER TABLE (см. главу 13, "Создание базы данных"). В определениях всех таблиц учебной базы данных в приложении А, "Учебная база данных", указаны первичные ключи, следующие синтаксису ANSI/ISO.

При указании первичного ключа в определении таблицы СУБД автоматически проверяет уникальность его значений при выполнении каждой инструкции INSERT или UPDATE. Попытка добавить строку с уже имеющимся значением первичного ключа или обновить строку таким образом, что первичный ключ потеряет свою уникальность, завершится выдачей сообщения об ошибке.

Прочие условия уникальности столбцов

Иногда требуется, чтобы столбец, не являющийся первичным ключом таблицы, все же содержал уникальные значения во всех строках. Предположим, например, что требуется ограничить данные в таблице SALESREPS таким образом, чтобы не было двух служащих с одинаковыми именами. Достичь этой цели можно, наложив условие *уникальности* на столбец NAME. СУБД обеспечивает это условие точно так же, как обеспечивает уникальность первичного ключа. Любая попытка добавить или обновить строку, нарушающая условие уникальности, завершится неуспешно.

И ограничение уникальности, и первичный ключ предотвращают появление одинаковых значений во множестве столбцов таблицы, но между ними имеются фундаментальные отличия.

- Таблица может иметь только один первичный ключ, в то время как ограничений уникальности может быть наложено несколько.
- Столбцы, указанные в первичном ключе, должны быть определены как NOT NULL, в то время как столбцы, включенные в условие уникальности, могут быть определены и как NULL, и как NOT NULL.

В соответствии со стандартом ANSI/ISO, условие уникальности столбцов или комбинаций столбцов определяется в инструкции CREATE TABLE или ALTER TABLE. Однако в DB2 условия уникальности были реализованы задолго до того, как они вошли в стандарт ANSI/ISO, и в этой СУБД они были реализованы как часть инструкции CREATE INDEX. Это одна из административных инструкций реляционной базы данных, работающая с физическим хранением базы данных на диске. Обычно пользователю не приходится беспокоиться об этих инструкциях — это дело администратора базы данных.

Многие коммерческие СУБД первоначально придерживались соглашений DB2, а не стандарта ANSI/ISO, и требовали использования инструкции CREATE INDEX. Однако со временем разработчики DB2 перенесли условия уникальности в инструкции CREATE TABLE и ALTER TABLE. Большинство прочих производителей коммерческих СУБД пошли по тому же пути и в настоящее время поддерживают синтаксис условия уникальности ANSI/ISO.

Уникальность и значения NULL

Значения NULL создают проблемы в столбце первичного ключа таблицы или в столбце, для которого задано условие уникальности. Предположим, что вы пытаетесь добавить в таблицу строку с первичным ключом, имеющим значение NULL (или, если первичный ключ является составным, частично имеющим значение NULL). Из-за значения NULL СУБД не может однозначно решить, является ли первичный ключ дубликатом уже имеющегося в таблице ключа. Может оказаться верным и то и другое, в зависимости от "настоящего" значения отсутствующих данных. По этой причине стандарт SQL требует, чтобы любой столбец, являющийся частью первичного ключа, был объявлен с ограничением NOT NULL.

Стандарт SQL не накладывает такое ограничение на столбцы в условии уникальности, хотя некоторые реализации SQL, такие как DB2, и поступают таким

образом. Однако имеются значительные расхождения в том, как разные реализации SQL обеспечивают выполнение условия уникальности в случае столбцов, которые содержат значения NULL, в частности, когда условие уникальности охватывает несколько столбцов, допускающих значения NULL. Для иллюстрации этих различий рассмотрим следующую таблицу, которая может использоваться для распределения студентов по руководителям.

```
CREATE TABLE ADVISOR_ASSIGNMENTS
    (STUDENT_NAME VARCHAR(25),
     ADVISOR_NAME VARCHAR(25),
     UNIQUE (STUDENT_NAME, ADVISOR_NAME));
```

Ниже приведена таблица, полученная в результате вставки данных в таблицу ADVISOR_ASSIGNMENTS при помощи нескольких инструкций INSERT. В первом столбце находятся значения столбца STUDENT_NAME, во втором — столбца ADVISOR_NAME, а в остальных трех столбцах показаны результаты для текущих версий Oracle, SQL Server и MySQL соответственно (СУБД DB2 не включена, поскольку она не допускает распространения условия уникальности на столбцы, в которых могут быть значения NULL). Интересно, что никакие две СУБД не дают одинаковые результаты во всех строках.

Номер строки	STUDENT_NAME	ADVISOR_NAME	Oracle	SQL Server	MySQL
1	NULL	NULL	Допустимая строка	Допустимая строка	Допустимая строка
2	NULL	NULL	Допустимая строка	Копия строки 1	Допустимая строка
3	Bill	NULL	Допустимая строка	Допустимая строка	Допустимая строка
4	Bill	NULL	Копия строки 3	Копия строки 3	Допустимая строка
5	Sue	Harrison	Допустимая строка	Допустимая строка	Допустимая строка
6	Sue	Harrison	Копия строки 5	Копия строки 5	Копия строки 5

Ссылочная целостность

В главе 4, "Реляционные базы данных", уже были рассмотрены первичные и внешние ключи, а также отношения "предок-потомок" между таблицами, создаваемые этими ключами. На рис. 11.1 изображены таблицы SALESREPS и OFFICE, а также связь между ними, реализованная через первичный и внешний ключи. Столбец OFFICE является первичным ключом таблицы OFFICES и уникальным образом идентифицирует каждую строку в этой таблице. Столбец REP_OFFICE таблицы SALESREPS представляет собой внешний ключ для таблицы OFFICES. Он идентифицирует офис, за которым закреплен каждый служащий.

Таблица **OFFICES**

OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300,000.00	$186,042.00
11	New York	Eastern	106	$575,000.00	$692,637.00
12	Chicago	Eastern	104	$800,000.00	$735,042.00
13	Atlanta	Eastern	NULL	$350,000.00	$367,911.00
21	Los Angeles	Western	108	$725,000.00	$835,915.00

Первичный ключ

Связь

Таблица **SALESREPS**

Внешний ключ

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE
105	Bill Adams	37	13	Sales Rep
109	Mary Jones	31	11	Sales Rep
102	Sue Smith	48	21	Sales Rep
106	Sam Clark	52	11	VP Sales
104	Bob Smith	33	12	Sales Mgr
101	Dan Roberts	45	12	Sales Rep
110	Tom Snyder	41	NULL	Sales Rep
108	Larry Fitch	62	21	Sales Mgr
103	Paul Cruz	29	12	Sales Rep
107	Nancy Angelli	49	22	Sales Rep

Рис. 11.1. Отношение "внешний ключ-первичный ключ"

Столбцы REP_OFFICE и OFFICE создают между строками таблиц OFFICES и SALESREPS отношение "предок-потомок". Для каждой строки таблицы OFFICES (предок) существует нуль или более строк таблицы SALESREPS (потомки) с таким же идентификатором офиса. Для каждой строки таблицы SALESREPS (потомок) существует ровно одна строка таблицы OFFICES (предок) с таким же идентификатором офиса.

Предположим, что вы пытаетесь вставить в таблицу SALESREPS новую строку, содержащую недопустимый идентификатор офиса, как в следующем примере:

```
INSERT INTO SALESREPS (EMPL_NUM, NAME, REP_OFFICE, AGE,
                       HIRE_DATE, SALES)
VALUES (115, 'George Smith', 31, 37, '2008-04-01', 0.00);
```

На первый взгляд, это корректная инструкция INSERT. Фактически, в некоторых реализациях SQL такая строка будет успешно добавлена в таблицу. В базе данных появится информация о том, что Джордж Смит (George Smith) работает в офисе номер 31, хотя такого офиса в таблице OFFICES нет. Очевидно, что новая строка нарушает отношение "предок-потомок", существующее между таблицами OFFICES и SALESREPS. Скорее всего, идентификатор офиса 31 в инструкции INSERT является ошибочным — вероятно, пользователь намеревался ввести идентификатор 11, 21 или 13.

Кажется достаточно очевидным, что допустимое значение для столбца REP_OFFICE должно быть равно одному из значений, содержащихся в столбце OFFICE. Это правило известно как ограничение *ссылочной целостности*. Оно обеспечивает целостность отношений "предок-потомок", создаваемых внешними и первичными ключами.

Ссылочная целостность является ключевым элементом реляционной модели с момента ее представления доктором Коддом. Однако условия ссылочной целостности отсутствовали как в экспериментальной СУБД IBM System/R, так и в первых версиях DB2 и SQL/DS. Компания IBM добавила поддержку ссылочной целостности в DB2 в 1989 году, а в стандарт SQL она была добавлена уже после выхода первой редакции стандарта SQL1. В настоящее время большинство производителей СУБД поддерживают ссылочную целостность в своих программных продуктах.

Проблемы, связанные со ссылочной целостностью

Существует четыре типа изменений базы данных, которые могут нарушить ссылочную целостность отношений "предок-потомок". Рассмотрим каждую из этих четырех ситуаций на примере таблиц OFFICES и SALESREPS, представленных на рис. 11.1.

- **Добавление новой дочерней строки**. Когда происходит добавление новой строки в дочернюю таблицу (SALESREPS), значение ее внешнего ключа (REP_OFFICE) *должно* быть равно одному из значений первичного ключа (OFFICE) в родительской таблице (OFFICES). Если значение внешнего ключа не равно ни одному из значений первичного ключа, то добавление такой строки повредит базу данных, поскольку в ней появится потомок без предка ("*сирота*"). Обратите внимание на то, что добавление строки в родительскую таблицу проблем никогда не вызывает — она просто становится предком без потомков.
- **Обновление внешнего ключа в дочерней строке**. Это та же проблема, что и в предыдущей ситуации, но выраженная в иной форме. Если внешний ключ (REP_OFFICE) обновляется инструкцией UPDATE, то его новое значение должно быть равно одному из значений первичного ключа (OFFICE) в родительской таблице (OFFICES). В противном случае обновленная строка окажется "сиротой".
- **Удаление родительской строки**. Если из родительской таблицы (OFFICES) будет удалена строка, у которой есть хотя бы один потомок (в таблице SALESREPS), то дочерние строки станут "сиротами". Значения внешних ключей (REP_OFFICE) в этих строках больше не будут равны ни одному из значений первичного ключа (OFFICE) родительской таблицы. Обратите внимание на то, что удаление строки из дочерней таблицы никогда не вызывает проблем — просто предок этой строки после удаления будет иметь на одного потомка меньше.
- **Обновление первичного ключа в родительской строке**. Это иная форма проблемы, рассмотренной в предыдущем пункте. Если происходит изменение первичного ключа (OFFICE) некоторой строки в родительской таблице (OFFICES), все существующие потомки этой строки становятся "сиротами", поскольку их внешние ключи больше не равны ни одному первичному ключу.

Средства поддержки ссылочной целостности в стандарте ANSI/ISO позволяют обрабатывать каждую из четырех описанных ситуаций. Первая проблема (добавление строки в дочернюю таблицу) решается путем проверки значений в столбцах внешнего ключа перед выполнением инструкции INSERT. Если они не равны ни одному из значений первичного ключа, то инструкция INSERT отбрасывается и выдается сообщение об ошибке. По отношению к рис. 11.1 это означает, что для добавления в таблицу SALESREPS нового служащего необходимо, чтобы в таблице OFFICES уже был офис, в который назначается данный служащий. Как видите, в учебной базе данных это ограничение имеет смысл.

Вторая проблема (обновление дочерней таблицы) решается аналогично, путем проверки нового значения внешнего ключа. Если нет ни одного равного ему значения первичного ключа, инструкция UPDATE отбрасывается с выдачей сообщения об ошибке. По отношению к рис. 11.1 это означает, что для перевода служащего в другой офис необходимо, чтобы этот офис уже присутствовал в таблице OFFICES. В учебной базе данных это ограничение также имеет смысл.

Третья проблема (удаление родительской строки) является более сложной. Предположим, например, что вы закрыли офис в Лос-Анджелесе и хотите удалить соответствующую строку из таблицы OFFICES. Что при этом должно произойти с двумя дочерними строками в таблице SALESREPS, которые представляют служащих, закрепленных за офисом в Лос-Анджелесе? В зависимости от ситуации, можно сделать следующее:

- не удалять из базы данных офис до тех пор, пока служащие не будут переведены в другой офис;
- автоматически удалить двух служащих из таблицы SALESREPS;
- в столбце REP_OFFICE установить для этих двух служащих значение NULL, показывая тем самым, что идентификатор их офиса неизвестен;
- в столбце REP_OFFICE для этих двух служащих установить по умолчанию некоторое значение, например идентификатор главного офиса в Нью-Йорке, указывая тем самым, что служащие автоматически переводятся в этот офис.

Аналогичные сложности существуют и в четвертой ситуации (обновление первичного ключа в родительской таблице). Допустим, по каким-либо причинам требуется изменить идентификатор офиса в Лос-Анджелесе с 21 на 23. Подобно предыдущему примеру, возникает вопрос о том, как поступить с двумя дочерними строками в таблице SALESREPS, представляющими служащих лос-анжелесского офиса. И снова проблему можно решить четырьмя способами:

- не изменять идентификатор офиса до тех пор, пока служащие не будут переведены в другой офис; в таком случае в таблицу OFFICES следует вначале добавить строку с новым идентификатором офиса в Лос-Анджелесе, затем обновить таблицу SALESREPS и, наконец, удалить строку со старым идентификатором лос-анжелесского офиса;

- автоматически обновить идентификатор офиса этих двух служащих в таблице SALESREPS для того, чтобы их строки были по-прежнему связаны с лос-анжелесской строкой в таблице OFFICES через ее новый идентификатор офиса;
- в столбце REP_OFFICE установить для этих двух служащих значение NULL, показывая тем самым, что идентификатор их офиса неизвестен;
- в столбце REP_OFFICE установить по умолчанию для этих двух служащих некоторое значение, например идентификатор главного офиса в Нью-Йорке, указывая тем самым, что служащие автоматически переводятся в этот офис.

В этом конкретном примере некоторые способы решения могут показаться более логичными, чем другие, но относительно легко можно придумать примеры, в которых любая из четырех возможностей окажется наиболее "правильным" вариантом (если необходимо, чтобы база данных служила точной моделью реальных ситуаций). В исходном стандарте SQL1 для примеров, приведенных выше, предусмотрена только первая возможность — запрет на изменение значения используемого первичного ключа и на удаление строки с используемым первичным ключом. Однако в DB2 существуют *правила удаления*, благодаря которым становятся возможными и другие варианты. Начиная со стандарта SQL2 эти правила были преобразованы в *правила удаления и обновления*, охватывающие как удаление родительских строк, так и обновление первичных ключей.

Правила удаления и обновления*

Для каждого отношения "предок-потомок" в базе данных, создаваемого внешним ключом, стандарт SQL позволяет указать связанные с ним правило удаления и правило обновления. Правило удаления определяет те действия, которые СУБД выполняет, когда пользователь пытается удалить строку из родительской таблицы. Можно задать одно из четырех возможных правил удаления.

- **RESTRICT** — запрещает удаление строки из родительской таблицы, если строка имеет потомков. Инструкция DELETE, пытающаяся удалить такую строку, отвергается, и выдается сообщение об ошибке. Таким образом, из родительской таблицы можно удалять только строки, не имеющие потомков. Для таблиц, представленных на рис. 11.1, это правило можно сформулировать так: "Нельзя удалить офис, если в нем кто-то работает".
- **CASCADE** — определяет, что при удалении родительской строки все дочерние строки *также* автоматически удаляются из дочерней таблицы. Для таблиц на рис. 11.1 это правило можно сформулировать так: "При удалении офиса его служащие автоматически увольняются".
- **SET NULL** — определяет, что при удалении родительской строки внешним ключам во всех ее дочерних строках автоматически присваивается значение NULL. Таким образом, удаление строки из родительской таблицы вызывает установку значений NULL в некоторых столбцах дочерней таблицы. Для таблиц, представленных на рис. 11.1, это правило можно

сформулировать так: "При расформировании офиса служащие остаются в резерве, т.е. их офис неизвестен".

- **SET DEFAULT** — определяет, что при удалении родительской строки внешним ключам во всех ее дочерних строках присваивается определенное значение, по умолчанию установленное для данного столбца. Таким образом, удаление строки из родительской таблицы вызывает присваивание значений по умолчанию некоторым столбцам дочерней таблицы. Для таблиц, приведенных на рис. 11.1, это правило можно сформулировать так: "При удалении офиса его служащие автоматически переводятся в офис по умолчанию, заданный в определении таблицы SALESREPS".

Стандарт SQL на самом деле называет правило RESTRICT правилом NO ACTION. Подобное название несколько запутывает. Оно означает, что "если вы попытаетесь удалить родительскую строку, у которой имеются дочерние строки, то СУБД не предпримет никаких действий с этой строкой". Однако СУБД на самом деле сгенерирует код ошибки. Имя RESTRICT, которое для этого ограничения используют DB2 и некоторые другие СУБД, интуитивно представляется более корректно описывающим происходящее. Последние версии DB2 поддерживают оба правила — как RESTRICT, так и NO ACTION. Разница между ними заключается в сроках выполнения. Правило RESTRICT применяется до всех прочих ограничений; правило NO ACTION выполняется после всех прочих ссылочных ограничений. Практически при любых обстоятельствах эти два правила работают одинаково.

Как вы можете догадаться, поддержка правил удаления различна в разных реализациях SQL. В табл. 11.1 показано, какие правила поддерживаются текущими версиями популярных СУБД.

Таблица 11.1. Поддержка правил удаления в ведущих СУБД

Правило	Oracle	DB2	SQL Server	MySQL
RESTRICT (NO ACTION)	Да, по умолчанию (правило нельзя указать явно)	Да	Да	Да
CASCADE	Да	Да	Да	Да
SET NULL	Да	Да	Да	Да
SET DEFAULT	Нет	Нет	Да	Да

Применительно к таблице ORDERS из учебной базы данных, вот как выглядят три определения ограничений на внешние ключи, использующие разные правила удаления.

```
CREATE TABLE ORDERS
       (.
        .
        .
       FOREIGN KEY PLACEDBY (CUST)
        REFERENCES CUSTOMERS(CUST_NUM)
         ON DELETE CASCADE,
       FOREIGN KEY TAKENBY (REP)
        REFERENCES SALESREPS(EMPL_NUM)
```

```
  ON DELETE SET NULL,
FOREIGN KEY ISFOR (MFR, PRODUCT)
 REFERENCES PRODUCTS(MFR_ID, PRODUCT_ID)
  ON DELETE RESTRICT);
```

Аналогично тому, как правила удаления определяют действия СУБД при попытке удалить строку из родительской таблицы, так и правила обновления определяют действия СУБД, когда пользователь пытается обновить значение первичного ключа в таблице-предке. Вновь имеется четыре возможности, аналогичные правилам удаления.

- **RESTRICT** — запрещает обновление первичного ключа в строке родительской таблицы, если у строки есть потомки. Инструкция UPDATE, пытающаяся изменить значение первичного ключа в такой строке, отбрасывается, и выдается сообщение об ошибке. Таким образом, в родительской таблице можно обновлять первичные ключи только в строках, не имеющих потомков. Для таблиц на рис. 11.1 это правило можно сформулировать так: "Нельзя изменить офис, в котором работают служащие".
- **CASCADE** — указывает, что при изменении значения первичного ключа в родительской строке соответствующее значение внешнего ключа в дочерней таблице *также* автоматически изменяется во всех строках-потомках таким образом, чтобы соответствовать новому значению первичного ключа. Для таблиц, представленных на рис. 11.1, это правило можно сформулировать так: "Изменение офиса вызывает автоматическое изменение его идентификатора для всех служащих данного офиса".
- **SET NULL** — указывает, что при обновлении значения первичного ключа в родительской строке внешним ключам во всех ее дочерних строках автоматически присваивается значение NULL. Таким образом, изменение первичного ключа в родительской таблице вызывает установку значений NULL в некоторых столбцах дочерней таблицы. Для таблиц на рис. 11.1 это правило можно сформулировать так: "При изменении офиса его служащие переводятся неизвестно куда".
- **SET DEFAULT** — указывает, что при обновлении значения первичного ключа в родительской строке внешним ключам во всех ее дочерних строках присваивается значение по умолчанию, установленное для данного столбца. Таким образом, изменение первичного ключа в родительской таблице вызывает выполнение установки значения по умолчанию в некоторых столбцах дочерней таблицы. Для таблиц на рис. 11.1 это правило можно сформулировать так: "При изменении идентификатора офиса все его служащие переводятся в офис по умолчанию, указанный в определении таблицы SALESREPS".

Правило RESTRICT называется так в DB2 и некоторых других реализациях SQL; в стандарте SQL оно называется NO ACTION.

Поддержка правил обновления, как и правил удаления, различна в разных реализациях SQL. В табл. 11.2 показано, какие правила поддерживаются текущими версиями популярных СУБД.

Таблица 11.2. Поддержка правил обновления в ведущих СУБД

Правило	Oracle	DB2	SQL Server	MySQL
RESTRICT (NO ACTION)	Да, по умолчанию (правило нельзя указать явно)	Да	Да	Да
CASCADE	Нет	Нет	Да	Да
SET NULL	Нет	Нет	Да	Да
SET DEFAULT	Нет	Нет	Да	Да

Вы можете указать два разных правила в качестве правила удаления и правила обновления для отношения "предок-потомок", хотя в большинстве случаев эти правила совпадают. Если вы не укажете правило, по умолчанию будет применяться правило RESTRICT, как обладающее наименьшими возможностями случайно уничтожить или модифицировать данные. Каждое из правил подходит для своей ситуации. Если, например, мы хотим назначить для ограничения HASMGR, связывающего таблицы SALESREPS и OFFICES, правило удаления SET NULL, а правило обновления — CASCADE, то можем воспользоваться для этого приведенной далее инструкцией ALTER TABLE вместо показанной в приложении А, "Учебная база данных".

```
ALTER TABLE OFFICES
  ADD CONSTRAINT HASMGR
         FOREIGN KEY (MGR) REFERENCES SALESREPS(EMPL_NUM)
              ON UPDATE CASCADE
              ON DELETE SET NULL;
```

Какое именно правило следует применить, обычно определяется реальными ситуациями внешнего мира, которые моделируются базой данных. В учебной базе данных таблица ORDERS содержит три отношения "внешний ключ–первичный ключ" (рис. 11.2). Каждый заказ связывается с

- заказанным товаром;
- клиентом, сделавшим заказ;
- служащим, принявшим заказ.

Для каждого из этих отношений подходит свое правило.

- Для отношения между заказом и заказанным товаром следует, по-видимому, применять правило RESTRICT (как для удаления, так и для обновления). Нельзя удалить информацию о товаре или изменить его идентификатор, если в настоящий момент на этот товар имеются заказы.
- Для отношения между заказом и клиентом, сделавшим его, следует, по-видимому, использовать правило CASCADE (как при удалении, так и при обновлении). Если клиент прекращает сотрудничество с компанией, то из базы данных необходимо удалить его строку. В этом случае вместе с данными о клиенте следует также удалить и все его текущие заказы. Аналогично изменение идентификатора клиента следует автоматически распространить на все его заказы.

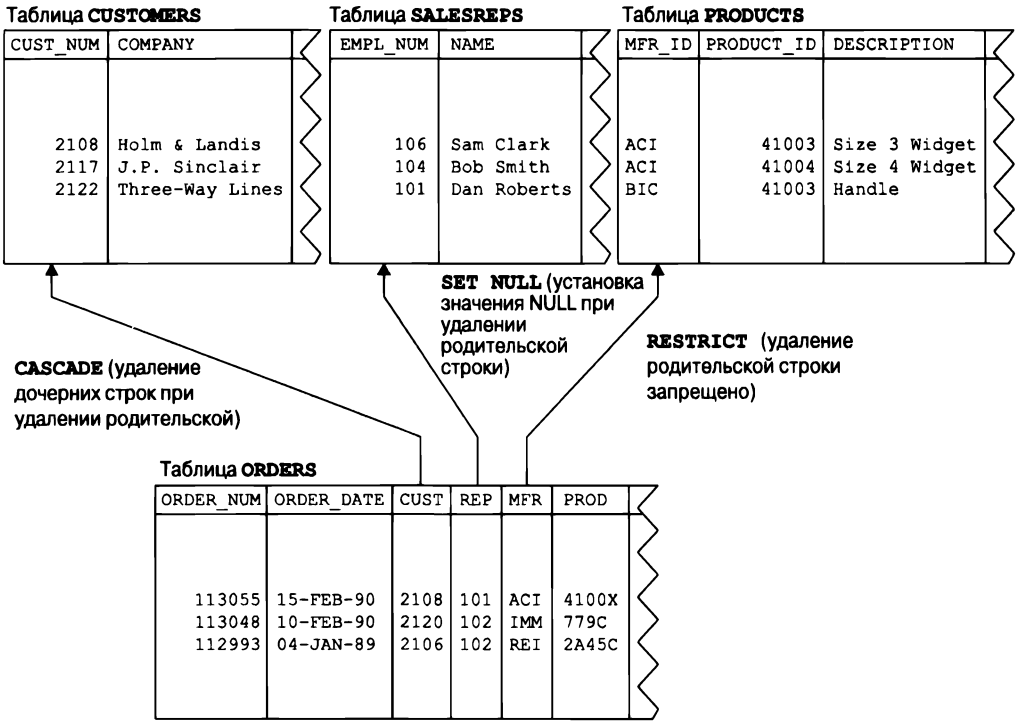


Рис. 11.2. Правила удаления в действии

- Для отношения между заказом и служащим, принявшим его, следует, по-видимому, применять правило SET NULL. Если служащий покидает компанию, то все заказы, принятые им, останутся без служащего (будут заказами "неизвестного служащего") до тех пор, пока их не закрепят за другим служащим. Для этого отношения можно также использовать правило SET DEFAULT, чтобы автоматически закреплять такие заказы за вице-президентом по сбыту. Что касается правила обновления, то для этого отношения необходимо, вероятно, применять правило CASCADE, чтобы автоматически распространять изменение идентификатора служащего на таблицу ORDERS.

Каскадные удаления и обновления*

Правило RESTRICT является одноуровневым, в том смысле, что в отношении "предок-потомок" оно затрагивает только родительскую таблицу. Правило CASCADE, напротив, может быть многоуровневым, что хорошо видно из рис. 11.3.

Предположим, что отношения между таблицами OFFICES/SALESREPS и SALESREPS/ORDERS, изображенные на данном рисунке, подчиняются правилу CASCADE. Что произойдет при удалении из таблицы OFFICES строки для лос-анжелесского офиса? В соответствии с правилом CASCADE для отношения таблиц OFFICES/SALESREPS, СУБД автоматически удалит из таблицы SALESREPS все строки, относящиеся к офису в Лос-Анджелесе (идентификатор офиса 21). Но

удаление в таблице SALESREPS строки для Сью Смит (Sue Smith) приводит в действие правило CASCADE для отношения таблиц SALESREPS/ORDERS. Согласно этому правилу, СУБД автоматически удалит из таблицы ORDERS все строки, относящиеся к Сью Смит (идентификатор служащего 102). Таким образом, удаление офиса вызывает каскадное удаление соответствующих записей о служащих, что, в свою очередь, вызывает каскадное удаление заказов.

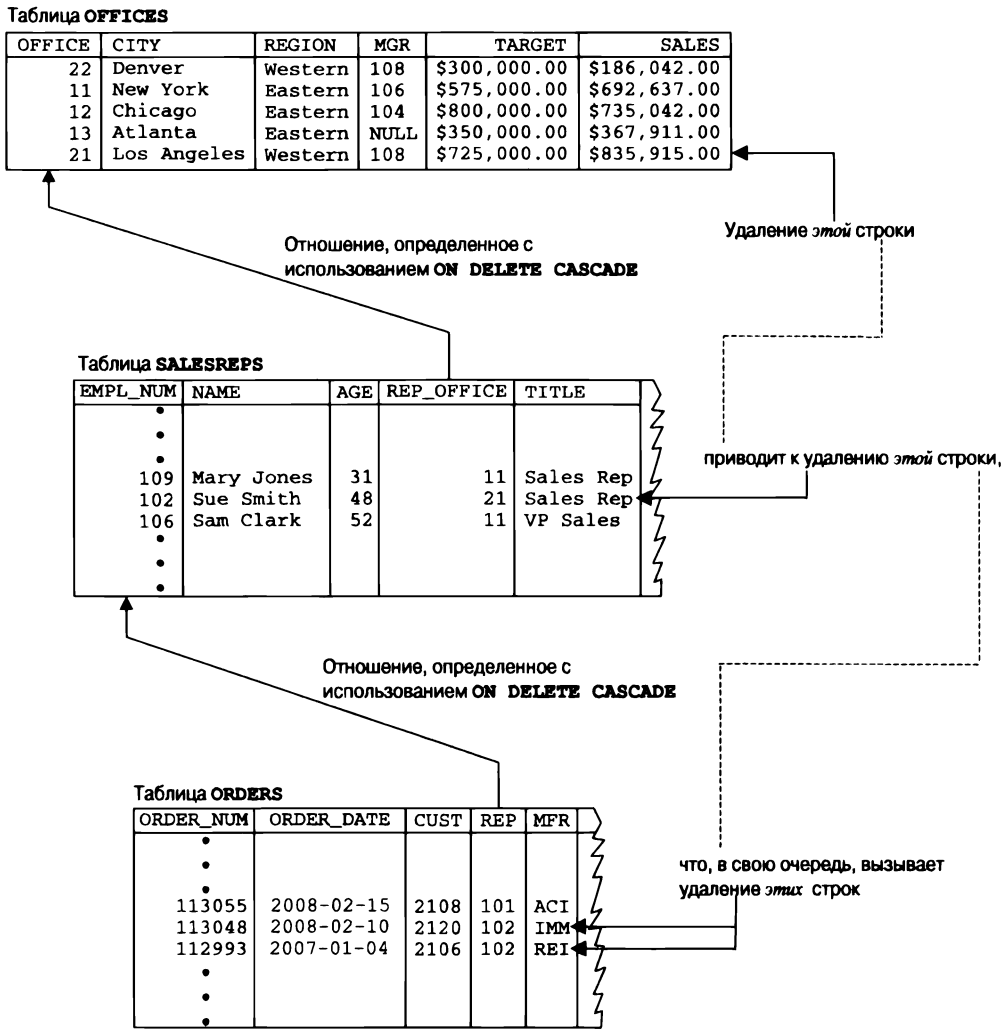


Рис. 11.3. Два уровня правил CASCADE

Как показывает данный пример, правило CASCADE следует применять с осторожностью, поскольку некорректное его использование может вызвать лавинообразное автоматическое удаление данных. Правила каскадного обновления могут привести к подобным многоуровневым обновлениям, если внешний ключ в таблице-потомке одновременно является и ее первичным ключом. На практике такая

ситуация встречается не часто, поэтому каскадное обновление обычно не имеет таких далеко идущих последствий, как каскадное удаление.

Правила удаления и обновления SET NULL и SET DEFAULT являются двухуровневыми; их влияние заканчивается на дочерней таблице. На рис. 11.4 снова изображены таблицы OFFICES, SALESREPS и ORDERS, но на этот раз отношение между таблицами OFFICES/SALESREPS подчиняется правилу удаления SET NULL. Теперь при удалении лос-анжелесского офиса СУБД установит в столбце REP_OFFICE таблицы SALESREPS значение NULL в тех строках, где был идентификатор офиса 21. Однако строки остаются в таблице SALESREPS, и влияние операции удаления распространяется только на дочернюю таблицу.

Таблица **OFFICES**

OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300,000.00	$186,042.00
11	New York	Eastern	106	$575,000.00	$692,637.00
12	Chicago	Eastern	104	$800,000.00	$735,042.00
13	Atlanta	Eastern	NULL	$350,000.00	$367,911.00
21	Los Angeles	Western	108	$725,000.00	$835,915.00

Удаление *этой* строки

Отношение, определенное с использованием **ON DELETE SET NULL**

Таблица **SALESREPS**

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE
•				
•				
•				
109	Mary Jones	31	11	Sales Rep
102	Sue Smith	48	21	Sales Rep
106	Sam Clark	52	11	VP Sales
•				
•				
•				

приводит к присваиванию значения **NULL** столбцу **REP_OFFICE** в *этой* строке,

Отношение, определенное с использованием **ON DELETE CASCADE**

Таблица **ORDERS**

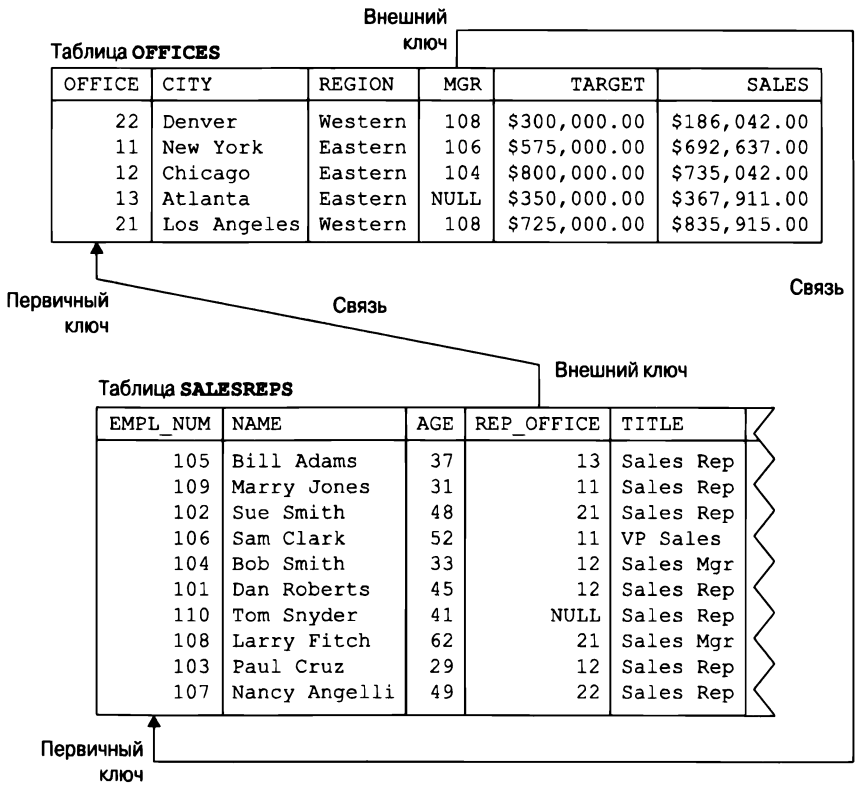
ORDER_NUM	ORDER_DATE	CUST	REP	MFR
•				
•				
•				
113055	2008-02-15	2108	101	ACI
113048	2008-02-10	2120	102	IMM
112993	2007-01-04	2106	102	REI
•				
•				
•				

что не оказывает *никакого влияния* на *эти* строки

Рис. 11.4. Комбинация правил удаления

Ссылочные циклы*

В учебной базе данных таблица SALESREPS содержит столбец REP_OFFICE — внешний ключ для таблицы OFFICES. Таблица OFFICES, в свою очередь, содержит столбец MGR — внешний ключ для таблицы SALESREPS. Как видно из рис. 11.5, эти два отношения образуют *ссылочный цикл*. Любая строка таблицы SALESREPS имеет ссылку на строку таблицы OFFICES, которая имеет ссылку на строку таблицы SALESREPS, и т.д. Этот цикл включает в себя две таблицы, но можно легко создать циклы из трех и более таблиц.



OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300,000.00	$186,042.00
11	New York	Eastern	106	$575,000.00	$692,637.00
12	Chicago	Eastern	104	$800,000.00	$735,042.00
13	Atlanta	Eastern	NULL	$350,000.00	$367,911.00
21	Los Angeles	Western	108	$725,000.00	$835,915.00

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE
105	Bill Adams	37	13	Sales Rep
109	Marry Jones	31	11	Sales Rep
102	Sue Smith	48	21	Sales Rep
106	Sam Clark	52	11	VP Sales
104	Bob Smith	33	12	Sales Mgr
101	Dan Roberts	45	12	Sales Rep
110	Tom Snyder	41	NULL	Sales Rep
108	Larry Fitch	62	21	Sales Mgr
103	Paul Cruz	29	12	Sales Rep
107	Nancy Angelli	49	22	Sales Rep

Рис. 11.5. Ссылочный цикл

Ссылочные циклы представляют особую проблему для ссылочной целостности независимо от количества таблиц в них. Предположим, что в двух таблицах, изображенных на рис. 11.5, для первичных и внешних ключей не допускаются значения NULL. (На самом деле в учебной базе данных такая ситуация допустима; причины этого сейчас станут вам ясны.) Рассмотрим следующий запрос на добавление и соответствующие ему инструкции INSERT.

Вы только что приняли на работу нового служащего Бена Адамса (идентификатор служащего 115), который назначен руководителем нового офиса в Детройте (идентификатор офиса 14).

```
INSERT INTO SALESREPS (EMPL_NUM, NAME, REP_OFFICE,
                       HIRE_DATE, SALES)
     VALUES (115, 'Ben Adams', 14, '2008-04-01', 0.00);

INSERT INTO OFFICES (OFFICE, CITY, REGION, MGR, TARGET, SALES)
     VALUES (14, 'Detroit', 'Eastern', 115, 0.00, 0.00);
```

К сожалению, выполнение первой инструкции (для Бена Адамса) будет неуспешным. Почему? Потому что в новой строке есть ссылка на идентификатор офиса 14, которого в базе данных еще нет! Очевидно, что изменение порядка инструкции INSERT ни к чему не приведет.

```
INSERT INTO OFFICES (OFFICE, CITY, REGION, MGR, TARGET, SALES)
     VALUES (14, 'Detroit', 'Eastern', 115, 0.00, 0.00);

INSERT INTO SALESREPS (EMPL_NUM, NAME, REP_OFFICE,
                       HIRE_DATE, SALES)
     VALUES (115, 'Ben Adams', 14, '2008-04-01', 0.00);
```

Выполнение первой инструкции INSERT (на этот раз для офиса в Детройте) также закончится неудачей, поскольку в новой строке есть ссылка на идентификатор служащего 115 (руководитель офиса), а Бен Адамс пока еще отсутствует в базе данных! Для предотвращения подобной взаимоблокировки по крайней мере один из внешних ключей ссылочного цикла *должен* допускать значения NULL. Учебная база данных спроектирована таким образом, что в столбце MGR значения NULL не допускаются, а в столбце REP_OFFICE — допускаются. Так что ввод двух строк можно выполнить с помощью двух инструкций INSERT и одной инструкции UPDATE.

```
INSERT INTO SALESREPS (EMPL_NUM, NAME, REP_OFFICE,
                       HIRE_DATE, SALES)
     VALUES (115, 'Ben Adams', NULL, '2008-04-01', 0.00);

INSERT INTO OFFICES (OFFICE, CITY, REGION, MGR, TARGET, SALES)
     VALUES (14, 'Detroit', 'Eastern', 115, 0.00, 0.00);

UPDATE SALESREPS
   SET REP_OFFICE = 14
 WHERE EMPL_NUM = 115;
```

Как видно из данного примера, в некоторых ситуациях было бы удобно, чтобы условия ссылочной целостности не проверялись до тех пор, пока не будет выполнен ряд взаимосвязанных обновлений. Определенные возможности такой отложенной проверки имеются в стандарте SQL, начиная с SQL2, о чем будет рассказано позже в данной главе.

Наличие ссылочного цикла накладывает ограничения на выбор правил удаления и обновления для отношений, образующих цикл. Рассмотрим ссылочный цикл из трех таблиц, изображенный на рис. 11.6. Таблица PETS содержит имена трех домашних животных и трех мальчиков, которые нравятся этим животным. Таблица GIRLS содержит имена трех девочек и трех домашних животных, которые нравятся этим

девочкам. Таблица BOYS содержит имена четырех мальчиков и имена девочек, которые нравятся мальчикам. Для всех трех отношений данного цикла задано правило удаления RESTRICT. А теперь обратите внимание на то, что строка мальчика Джорджа (George) — это *единственная* строка в трех таблицах, которую можно удалить. Любая другая строка является предком в каком-нибудь отношении и потому защищена правилом RESTRICT от удаления. Из-за возможности подобной аномалии не следует задавать данное правило для всех отношений ссылочного цикла.

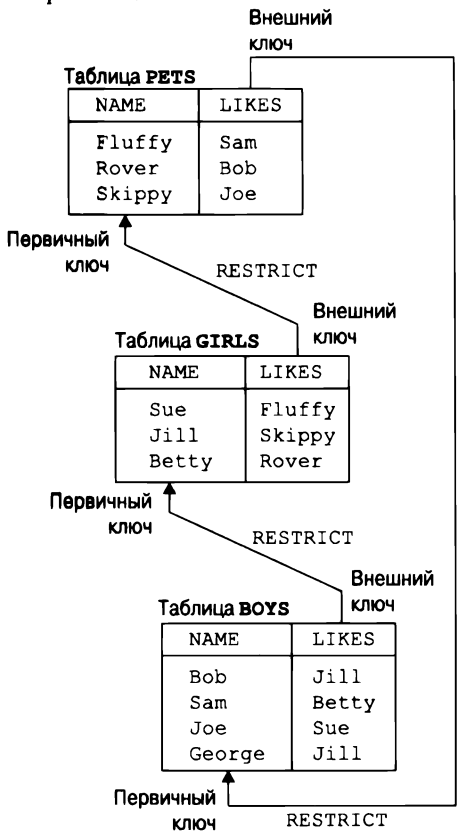


Рис. 11.6. Цикл с правилами RESTRICT

Как показано на рис. 11.7, правило удаления CASCADE создает не меньшую проблему. На данном рисунке изображены те же таблицы, что и на рис. 11.6, но правило удаления RESTRICT заменено правилом CASCADE. Предположим, что вы хотите удалить из таблицы BOYS строку Боба. В соответствии с правилами удаления, СУБД удалит строку Ровера (животного, которому нравится Боб) из таблицы PETS, затем удалит строку Бетти (которой нравится Ровер) из таблицы GIRLS, потом удалит строку Сэма (который обожает Бетти) и так далее до тех пор, пока не будут удалены все строки из трех таблиц! Для таких маленьких таблиц подобная операция может оказаться полезной, но для промышленных баз данных с тысячами строк невозможно проследить все каскадные удаления и сохранить целостность базы данных. По этой причине в DB2 существует правило, которое запреща-

ет ссылочные циклы из двух или более таблиц, в которых все правила удаления имеют тип CASCADE. В таком цикле хотя бы одно отношение *должно* иметь правило удаления RESTRICT или SET NULL, чтобы прервать цепочку каскадных удалений.

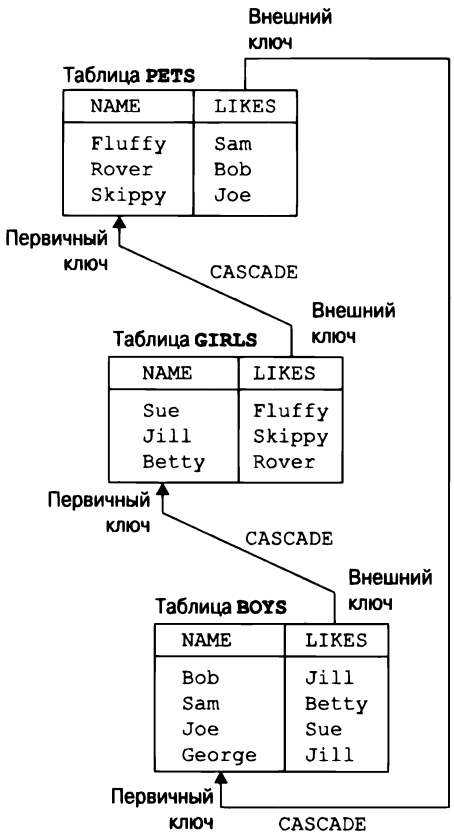


Рис. 11.7. Некорректный цикл с правилами CASCADE

Внешние ключи и значения NULL*

В отличие от первичных ключей, внешние ключи в реляционной базе данных могут содержать значения NULL. В учебной базе данных внешний ключ REP_OFFICE в таблице SALESREPS может принимать значения NULL. И действительно, данный столбец содержит значение NULL в строке Тома Снайдера, поскольку Том еще не назначен в какой-либо офис. Но значение NULL во внешнем ключе вызывает интересный вопрос об условии ссылочной целостности отношения "первичный ключ–внешний ключ". Равно ли значение NULL одному из значений первичного ключа или нет? Ответ на этот вопрос следующий: "Может быть. Все зависит от настоящего значения отсутствующих или неизвестных данных".

Стандарт SQL автоматически полагает, что внешний ключ, имеющий значение NULL, удовлетворяет условию ссылочной целостности. Другими словами, сомнение трактуется в пользу строки, и ей разрешается быть частью таблицы-потомка,

хотя значение ее внешнего ключа не равно ни одному значению первичного ключа в таблице-предке. Интересно отметить, что условие ссылочной целостности считается выполненным, если *любая часть* внешнего ключа имеет значение NULL. Для составных внешних ключей это может привести к непредвиденным последствиям, как, например, в случае составного внешнего ключа, связывающего таблицу ORDERS с таблицей PRODUCTS.

Предположим на минутку, что в столбце PRODUCT таблицы ORDERS разрешены значения NULL и что для отношения таблиц PRODUCTS/ORDERS установлено правило удаления SET NULL (на самом деле учебная база данных спроектирована иначе по причинам, которые станут понятны из данного примера). Благодаря первому условию в таблицу ORDERS можно успешно добавить заказ на товар с идентификатором производителя (MFR), имеющим значение "ABC", и идентификатором товара (PRODUCT), имеющим значение NULL, поскольку столбец PRODUCT может содержать значения NULL. Следуя стандарту ANSI/ISO, СУБД полагает, что данная строка соответствует условию ссылочной целостности для таблиц ORDERS и PRODUCTS, хотя в таблице PRODUCTS нет ни одного товара с идентификатором изготовителя "ABC".

Правило удаления SET NULL может вызвать аналогичный эффект. Удаление строки из таблицы PRODUCTS должно привести к тому, что внешним ключам во всех строках-потомках таблицы ORDERS будет присвоено значение NULL. Но на самом деле значение NULL будет записано только в те столбцы внешних ключей, которые могут принимать такие значения. Если бы в таблице PRODUCTS имелась одна строка для предприятия-изготовителя "DEF", то удаление этой строки привело бы к тому, что в столбце PRODUCT таблицы ORDERS для всех ее строк-потомков были установлены значения NULL, но в столбце MFR сохранились бы значения "DEF". В результате данные строки имели бы в столбце MFR значение, которое не соответствовало бы ни одной строке таблицы PRODUCTS.

Во избежание подобной ситуации следует очень осторожно обходиться со значениями NULL в составных внешних ключах. В приложении, в котором осуществляется ввод данных в таблицы, содержащие внешний ключ, или выполняется обновление данных в таких таблицах, по отношению к столбцам внешнего ключа должен применяться принцип "все значения NULL или ни одного значения NULL". Внешние ключи, частично допускающие значения NULL, а частично нет, могут легко привести к возникновению проблем.

В стандарте SQL эта проблема решается путем предоставления администратору базы данных более широких возможностей управления значениями NULL во внешних ключах для обеспечения целостности данных. С помощью инструкции CREATE TABLE можно установить один из двух режимов, обеспечивающих целостность данных (однако их поддержка в текущих реализациях SQL встречается весьма редко).

- **MATCH FULL**. В режиме MATCH FULL (полное соответствие) требуется, чтобы внешние ключи таблицы-потомка были полностью равны первичному ключу таблицы-предка. В этом режиме ни одна часть внешнего ключа не может содержать значение NULL, поэтому в правилах удаления и обновления не затрагивается вопрос обработки этих значений.

- **MATCH PARTIAL**. В режиме MATCH PARTIAL (частичное соответствие) допускается, чтобы часть внешнего ключа имела значение NULL, при условии что остальная часть внешнего ключа равна соответствующей части какого-либо первичного ключа в таблице-предке. В данном режиме обработка значений NULL производится так, как было описано выше в правилах удаления и обновления.

Расширенные возможности ограничений

Ограничения на первичные и внешние ключи, условия уникальности столбцов и ограничения на отсутствующие значения (NULL) представляют собой достаточно специализированные способы обеспечения целостности данных. Начиная с SQL2, стандарт SQL идет гораздо дальше, предлагая более универсальные способы задания условий целостности данных и контроля над их выполнением. Полная схема включает четыре категории ограничений.

- **Ограничения столбцов** указываются как часть определения столбца при создании таблицы или при ее последующем изменении. Концептуально они ограничивают его допустимые значения. Ограничения столбцов располагаются в определениях отдельных столбцов в инструкциях CREATE TABLE или ALTER TABLE.
- **Домены** — это особая форма ограничений столбца. С их помощью можно создавать нечто наподобие новых типов данных, используемых в конкретной базе данных. По своей сути домен — это один из предопределенных типов данных, на который наложены некоторые дополнительные ограничения, являющиеся частью определения домена. После того как домен именован и определен, его имя может использоваться вместо типа данных в определениях новых столбцов. Столбцы наследуют ограничения домена. Сами домены создаются вне определений таблиц и столбцов с помощью инструкции CREATE DOMAIN. Как уже упоминалось, поддерживают эту инструкцию только некоторые из реализаций SQL.
- **Ограничения таблиц** задаются как часть определения таблицы при ее создании (в инструкции CREATE TABLE). Концептуально они ограничивают значения, которые могут присутствовать в строках таблицы. Обычно ограничения таблицы задаются в виде отдельной группы предложений после определений столбцов, но стандарт SQL позволяет перемежать ими определения столбцов.
- **Утверждения** являются наиболее общим типом ограничений SQL. Как и домены, они создаются вне определений таблиц и столбцов. Концептуально утверждение определяет взаимосвязь между значениями столбцов из разных таблиц одной базы данных. К сожалению, как и домены, утверждения поддерживаются только небольшим количеством реализаций SQL.

Каждый из четырех типов ограничений имеет собственное концептуальное назначение и является отдельной частью синтаксиса SQL. Однако различия между некоторыми из них довольно условны. Любое ограничение столбца может быть

задано как ограничение таблицы. Аналогично ограничение таблицы может быть задано как утверждение. На практике, пожалуй, лучше всего определять каждое ограничение на том уровне, которому оно наиболее естественно соответствует в ситуации реального мира, моделируемой базой данных. Ограничения, которые относятся ко всей ситуации (бизнес-процессы, взаимосвязи между клиентами и продуктами и т.п.), должны определяться как утверждения. Ограничения, относящиеся к конкретному типу объектов реального мира (клиентам или заказам), лучше определять на уровне таблиц и столбцов, соответствующих этому типу объектов. Когда одно и то же ограничение относится к нескольким различным столбцам, содержащим данные об одном и том же типе объектов, такое ограничение лучше всего задать как домен.

Утверждения

Примеры первых трех типов ограничений приводились в начале этой главы. Ограничения четвертого типа, *утверждения*, задаются с помощью инструкции SQL `CREATE ASSERTION`. Вот пример утверждения, которое может быть полезно в нашей учебной базе данных.

Гарантировать, что плановый объем продаж офиса не превысит сумму плановых объемов продаж его служащих.

```
CREATE ASSERTION target_valid
 CHECK ((OFFICES.TARGET <= SUM(SALESREPS.QUOTA)) AND
        (SALESREPS.REP_OFFICE = OFFICES.OFFICE));
```

Поскольку утверждение является одним из объектов базы данных (таким же, как таблицы и столбцы), оно должно обладать именем (в данном случае `target_valid`). Это имя используется в сообщениях об ошибках, генерируемых СУБД в случае нарушения утверждения. Утверждение, вызвавшее ошибку, может быть очевидным для маленькой демонстрационной базы данных, но в больших базах данных бывают определены десятки и сотни утверждений, поэтому важно знать, какое именно из них нарушено.

Вот еще один пример утверждения для нашей учебной базы данных.

Гарантировать, что общий объем заказов клиента не превысит предел его кредита.

```
CREATE ASSERTION credit_orders
 CHECK (CUSTOMERS.CREDIT_LIMIT >=
               SELECT SUM(ORDERS.AMOUNT)
                 FROM ORDERS
                WHERE ORDERS.CUST = CUSTOMER.CUST_NUM);
```

Как показывают эти примеры, утверждение SQL определяется заключенным в скобки условием, следующим за ключевым словом `CHECK`. При каждой попытке изменения содержимого базы данных с помощью инструкции `INSERT`, `DELETE` или `UPDATE` проверяется выполнение этого условия (с предлагаемыми для внесения данными). Если оно возвращает `TRUE`, модификация допускается. В противном случае СУБД не выполняет предложенные изменения, а вместо этого возвращает код ошибки и указывает, какое именно утверждение было нарушено.

Теоретически утверждения должны сильно замедлять работу СУБД, поскольку они проверяются для каждой инструкции, модифицирующей данные. Однако на практике после создания утверждения СУБД анализирует его и определяет, к каким таблицам и столбцам оно относится. После этого утверждение проверяется только при изменении указанных в нем таблиц и столбцов. Тем не менее утверждения нужно создавать очень осторожно, чтобы в базе данных не выполнялось проверок больше, чем это действительно необходимо для поддержания целостности данных, и эффект от их применения не нивелировался высокими накладными расходами.

Типы ограничений SQL

По функциональному назначению все ограничения, предусматриваемые стандартом SQL, можно разделить на несколько типов.

- Ограничение `NOT NULL` может действовать только на уровне столбца. Оно запрещает присваивать ячейкам столбца значение `NULL`.
- Ограничение `PRIMARY KEY` может действовать на уровне столбца или таблицы. Если первичный ключ состоит из одного столбца, данное ограничение удобнее определить для этого столбца. Если же ключ содержит несколько столбцов, ограничение должно быть определено для всей таблицы.
- Ограничение `UNIQUE` может действовать на уровне столбца или таблицы. Если уникальными должны быть значения одного столбца, данное ограничение лучше всего определить для этого столбца. Если же уникальной должна быть комбинация значений нескольких столбцов, тогда следует определить ограничение на уровне таблицы.
- Ссылочное ограничение `FOREIGN KEY` может действовать на уровне столбца или таблицы. Если внешний ключ состоит из одного столбца, данное ограничение удобнее задать для этого конкретного столбца. Если же ключ состоит из нескольких столбцов, лучше определить ограничение на уровне таблицы. Когда таблица связана через внешние ключи с большим количеством других таблиц, удобнее собрать *все* ограничения внешних ключей в одном месте в определении таблицы, а не разбрасывать их определения по отдельным столбцам.
- Ограничение `CHECK` может действовать на уровне столбца или таблицы. Кроме того, это *единственный* вид ограничения, который может быть частью определения домена или утверждения. Ограничение этого типа задается как условие отбора, подобное тем, что используются в предложении `WHERE` в запросах к базе данных. Данные удовлетворяют ограничению в том случае, если выполняется указанное в нем условие.

Каждому ограничению (независимо от его типа) может быть присвоено *имя*, однозначно идентифицирующее его среди других ограничений, определенных в этой же базе данных. Делать это не обязательно, особенно в простой базе данных, где связи ограничений с конкретными полями, таблицами и доменами достаточно очевидны и накладки маловероятны. Однако в сложной базе данных, где с каждым

столбцом или таблицей может быть связано несколько ограничений, полезно идентифицировать каждое ограничение по имени (особенно когда начинают происходить ошибки). Кроме того, используемые СУБД имена по умолчанию часто только запутывают пользователя и не несут смысловой нагрузки, так что всегда лучше назначать ограничениям свои собственные имена. Обратите внимание на то, что ограничения CHECK в утверждениях *обязаны* иметь имена; эти имена становятся именами утверждений, содержащих ограничения.

Отложенная проверка ограничений

В своей простейшей форме ограничения, определенные в базе данных, проверяются при каждой попытке изменения ее содержимого — т.е. при выполнении каждой инструкции INSERT, UPDATE или DELETE. Для СУБД, совместимых со стандартом SQL только на среднем или начальном уровне, это единственный допустимый режим работы. При полном же соответствии стандарту SQL имеется дополнительная возможность *отложенной проверки* ограничений.

Когда проверка ограничения откладывается, оно не проверяется для каждой отдельной SQL-инструкции. Вопрос принятия изменений остается нерешенным до завершения транзакции. (Об обработке транзакций и соответствующих SQL-инструкциях подробно рассказывается в главе 12, "Обработка транзакций".) Когда поступает инструкция COMMIT, сообщающая об окончании транзакции, СУБД проверяет все отложенные ограничения. Если все они удовлетворяются, транзакция завершается успешно и все изменения, внесенные в ходе транзакции, сохраняются в базе данных. Если же хотя бы одно из ограничений оказывается нарушенным, происходит *откат* транзакции, т.е. ни одно из внесенных в ходе транзакции изменений не принимается и база данных возвращается к тому состоянию, в котором она находилась на момент начала транзакции.

Возможность осуществлять отложенную проверку ограничений может быть очень важна в тех случаях, когда несколько изменений должны быть выполнены как одна операция, переводящая базу данных из одного согласованного состояния в другое. Предположим, например, что в нашей учебной базе данных определено такое утверждение.

Гарантировать, что плановый объем продаж офиса будет всегда в точности равен сумме плановых объемов продаж его служащих:

```
CREATE ASSERTION quota_totals
 CHECK ((OFFICES.TARGET = SUM(SALESREPS.QUOTA))
   AND (SALESREPS.REP_OFFICE = OFFICES.OFFICE));
```

Если не откладывать проверку до завершения транзакции, данное ограничение не позволит добавить в базу данных информацию о новом служащем. Почему? Да потому что для того, чтобы плановые объемы продаж офиса и всех его служащих оставались равными, нужно *одновременно* добавить запись для нового служащего с некоторым плановым объемом продаж (с помощью инструкции INSERT) и увеличить на ту же сумму плановый объем продаж офиса (с помощью инструкции UPDATE). Если вы попытаетесь сначала выполнить инструкцию INSERT для табли-

цы SALESREPS, таблица OFFICES еще не будет обновлена, и утверждение окажется невыполненным, поэтому инструкция INSERT будет отменена.

Аналогично, если вы попытаетесь сначала выполнить инструкцию UPDATE для таблицы OFFICES, таблица SALESREPS еще не будет обновлена, и утверждение снова окажется невыполненным. Единственное решение этой проблемы — отложить проверку утверждения до тех пор, пока не завершатся *обе* инструкции, и уже после этого убедиться, что вместе эти операции перевели базу данных в допустимое состояние.

Предлагаемый стандартом SQL механизм отложенной проверки ограничений обеспечивает такую возможность (и не только ее). Каждое создаваемое в базе данных ограничение (любого типа) может быть определено с атрибутом DEFERRABLE или NOT DEFERRABLE.

- Проверка ограничения, заданного как DEFERRABLE, может быть отложена до конца транзакции. Именно так должно быть определено утверждение, описанное в предыдущем примере. В нем для обновления плановых объемов продаж или добавления информации о новых служащих нужно обязательно отложить проверку ограничения.
- Проверка ограничения, определенного как NOT DEFERRABLE, не может быть отложена. Обычно к этой категории относятся ограничения первичного ключа и условия уникальности, а также многие ограничения на значения столбцов. Соблюдение условий целостности данных, налагаемых такими ограничениями, обычно не зависит от других выполняемых операций, и эти условия не могут быть нарушены даже временно. Они могут и должны проверяться после выполнения *каждой* SQL-инструкции, которая пытается модифицировать базу данных.

Поскольку ограничения типа NOT DEFERRABLE обеспечивают более жесткий контроль целостности данных, этот режим используется по умолчанию. Если же вы хотите создать ограничение, которое может быть отложено, включите в его объявление ключевое слово DEFERRABLE. Обратите также внимание на одну важную особенность этих двух атрибутов: они определяют только то, *может ли* быть отложена проверка данного ограничения. Будет ли она на самом деле отложена для очередной операции, определяет СУБД. Для ограничения может быть определено *начальное состояние*.

- Ограничение, определенное как INITIALLY IMMEDIATE, начинает свою жизнь как "безотлагательное", т.е. оно проверяется немедленно после выполнения каждой SQL-инструкции.
- Ограничение, определенное как INITIALLY DEFERRED, начинает свою жизнь как "отложенное", т.е. оно проверяется только по окончании транзакции. Конечно, этот атрибут не может использоваться совместно с атрибутом NOT DEFERRABLE.

Начальное состояние ограничения вступает в силу сразу после его создания. Кроме того, ограничение переводится в это состояние перед началом каждой

транзакции. Поскольку атрибут INITIALLY IMMEDIATE обеспечивает наиболее жесткий контроль целостности данных, он используется по умолчанию. Если же вы хотите, чтобы перед началом каждой транзакции ограничение автоматически переводилось в отложенное состояние, вам нужно включить в его определение атрибут INITIALLY DEFERRED.

Стандарт SQL допускает еще один механизм управления отложенностью ограничений. Вы можете динамически изменять способ обработки ограничений с помощью инструкции SET CONSTRAINTS. Предположим, например, что наша база данных содержит ограничение.

```
CREATE ASSERTION quota_totals
 CHECK ((OFFICES.TARGET = SUM(SALESREPS.QUOTA))
   AND (SALESREPS.REP_OFFICE = OFFICES.OFFICE))
       DEFERRABLE INITIALLY IMMEDIATE;
```

Это объявление говорит о том, что данное ограничение можно откладывать, но в обычном режиме оно должно проверяться после каждой операции над базой данных. Однако для особой транзакции, добавляющей в базу данных запись для нового сотрудника, можно временно отложить проверку ограничения. Делается это так.

```
SET CONSTRAINTS quota_totals DEFERRED;

INSERT INTO SALESREPS (EMPL_NUM, NAME, REP_OFFICE,
                       HIRE_DATE, QUOTA, SALES)
     VALUES (:num, :name, :office_num, :date, :amount, 0);

UPDATE OFFICES SET TARGET = TARGET + :amount
 WHERE (OFFICE = :office_num);

COMMIT;
```

После поступления инструкции COMMIT, завершающей транзакцию, ограничение quota_totals переводится в режим IMMEDIATE, поскольку оно определено как INITIALLY IMMEDIATE. Если вы захотите, чтобы ограничение quota_totals было проверено до окончания транзакции, например в случае, если между инструкциями UPDATE и COMMIT должны выполняться еще какие-то действия, не влияющие на данное ограничение, можете вручную перевести ограничение в режим IMMEDIATE следующей инструкцией.

```
SET CONSTRAINTS quota_totals IMMEDIATE;
```

Инструкция SET CONSTRAINTS допускает установку одного и того же режима сразу для нескольких ограничений — их нужно просто перечислить через запятую.

```
SET CONSTRAINTS quota_totals, rep_totals IMMEDIATE;
```

И наконец, можно одной инструкцией установить режим обработки всех ограничений базы данных.

```
SET CONSTRAINTS ALL DEFERRED;
```

Определенные стандартом SQL средства отложенной проверки ограничений значительно расширяют возможности обеспечения целостности баз данных.

Принципы управления отложенной проверкой ограничений, как и многое другое, стандарт SQL частично перенял у существующих СУБД, другая же их часть была реализована в некоторых СУБД уже после публикации стандарта. Например, СУБД DB2 фирмы IBM поддерживает возможность отложенной проверки ограничений, причем в соответствии с синтаксисом SQL. Однако ее инструкция `SET CONSTRAINTS` отличается от стандарта. Она оперирует отдельными таблицами базы данных, включая и отключая режим отложенной проверки ограничений, связанных с содержимым этих таблиц.

Бизнес-правила

В реальной жизни вопрос целостности данных часто бывает связан с порядками и правилами, установленными в конкретных организациях. Например, в компании, представленной учебной базой данных, могут быть установлены такие правила:

- клиентам не разрешается размещать заказы на сумму, превышающую предел их кредита;
- если клиенту назначается лимит кредита, превышающий $50000, то об этом должен быть уведомлен вице-президент по продажам;
- заказы хранятся в бухгалтерских книгах в течение шести месяцев; затем они аннулируются.

Кроме того, часто существуют различные бухгалтерские правила, которые необходимо соблюдать для сохранения целостности сумм, счетов и других величин, хранимых в базе данных. Для учебной базы данных вполне разумными будут, по-видимому, следующие правила:

- каждый раз, когда принимается новый заказ, значения в столбцах `SALES` для служащего, принявшего заказ, и для офиса, в котором этот служащий работает, должны быть увеличены на сумму заказа; удаление заказа или изменение его суммы также должно сопровождаться изменением столбцов `SALES`;
- каждый раз, когда принимается новый заказ, значение в столбце `QTY_ON_HAND` для заказываемого товара должно быть уменьшено на заказываемое количество; удаление заказа, изменение заказанного количества или изменение заказанного товара должно сопровождаться соответствующим изменением столбца `QTY_ON_HAND`.

В стандарте SQL определено, что эти правила выходят за рамки языка SQL. СУБД отвечает за хранение, организацию и обеспечение целостности данных, а за реализацию бизнес-правил отвечает прикладная программа, осуществляющая доступ к базе данных.

Перенос всей нагрузки по обеспечению бизнес-правил на прикладную программу имеет ряд недостатков.

- **Дублирование**. Если шесть различных программ осуществляют различные изменения в таблице ORDERS, то каждая из них должна содержать подпрограммы, обеспечивающие соблюдение бизнес-правил, касающихся этих изменений.
- **Недостаточная согласованность**. Если несколько программ, написанных разными программистами, осуществляют изменения в некоторой таблице, то поддержка бизнес-правил будет, вероятно, выполняться этими программами немного по-разному.
- **Трудность сопровождения**. Если бизнес-правила изменяются, то программисты должны найти все программы, в которых реализована поддержка этих правил, определить в них соответствующий код и корректно его модифицировать.
- **Сложность**. Часто приходится иметь дело с большим числом бизнес-правил. Даже программа для маленькой учебной базы данных, выполняющая обновление заказов, должна учитывать лимиты кредитов, изменять суммы продаж для служащих и офисов, а также имеющееся количество товара. В результате программа, осуществляющая простые обновления таблиц, может очень быстро стать довольно сложной.

Требование стандарта SQL, касающееся того, чтобы поддержку бизнес-правил осуществляли прикладные программы, не является чем-то уникальным. Так было с первых дней появления программ на языке COBOL и файловых СУБД. Однако много лет существовало постоянное стремление переложить ответственность за целостность данных на саму базу данных. В 1986 году в СУБД Sybase было введено понятие *триггер*, что явилось шагом по включению бизнес-правил в реляционную базу данных. Понятие триггера оказалось очень популярным, и в начале 90-х годов поддержку механизма триггеров начали осуществлять многие ведущие реляционные СУБД. Триггеры и обеспечение с их помощью выполнения бизнес-правил в особенности полезны в базах данных масштаба предприятия. Когда ежегодно десятки прикладных программистов создают и изменяют десятки прикладных программ, возможность централизованного определения бизнес-правил и управления ими становится особенно ценной.

Что такое триггер

Понятие *триггер* является относительно простым. С любым событием, вызывающим изменение содержимого таблицы, пользователь может связать сопутствующее действие (триггер), которое СУБД должна выполнять при каждом возникновении события. Тремя такими событиями, запускающими триггеры, являются попытки изменить содержимое таблицы инструкциями INSERT, DELETE и UPDATE. Действие, вызываемое событием, задается как последовательность инструкций SQL.

Чтобы понять, как работает триггер, рассмотрим конкретный пример. Когда в таблицу ORDERS добавляется новый заказ, в базу данных необходимо внести еще два изменения:

- значение в столбце SALES для служащего, принявшего заказ, должно быть увеличено на стоимость заказа;
- имеющееся количество заказываемого товара в столбце QTY_ON_HAND должно быть уменьшено на заказанное количество единиц товара.

Следующая инструкция Transact-SQL определяет триггер SQL Server по имени NEWORDER, который вызывает автоматическое выполнение описанных выше изменений.

```
CREATE TRIGGER NEWORDER
    ON ORDERS
   FOR INSERT
    AS UPDATE SALESREPS
          SET SALES = SALES + INSERTED.AMOUNT
         FROM SALESREPS, INSERTED
        WHERE SALESREPS.EMPL_NUM = INSERTED.REP
       UPDATE PRODUCTS
          SET QTY_ON_HAND = QTY_ON_HAND - INSERTED.QTY
         FROM PRODUCTS, INSERTED
        WHERE PRODUCTS.MFR_ID = INSERTED.MFR
          AND PRODUCTS.PRODUCT_ID = INSERTED.PRODUCT;
```

В первой части определения триггера указывается, что он вызывается всякий раз, когда к таблице ORDERS обращается инструкция INSERT. В оставшейся части определения (после ключевого слова AS) описывается действие, выполняемое триггером. В данном случае это действие представляет собой последовательность двух инструкций UPDATE: одна для таблицы SALESREPS, а другая для таблицы PRODUCTS. Ссылка на добавляемую строку делается с помощью имени псевдотаблицы INSERTED внутри инструкций UPDATE. Как видно из этого примера, в SQL Server с целью поддержки триггеров язык SQL был существенно расширен. К другим расширениям, не показанным здесь, относятся проверки IF/THEN/ELSE, циклы, вызовы процедур и даже инструкции PRINT, выводящие пользовательские сообщения.

Триггеры добавлены в версию SQL:1999 стандарта ANSI/ISO SQL, после того как наиболее популярные СУБД стали их поддерживать. Как и в случае со многими другими средствами SQL, рост популярности которых предшествовал их стандартизации, существуют значительные отличия в плане поддержки триггеров в различных СУБД. Некоторые из этих отличий чисто синтаксические, другие отражают особенности реализации триггеров непосредственно в самой СУБД.

Сказанное можно достаточно наглядно продемонстрировать на примере СУБД DB2. Вот определение рассмотренного выше триггера NEWORDER, записанное в синтаксисе DB2.

```
CREATE TRIGGER NEWORDER
         AFTER INSERT ON ORDERS
   REFERENCING NEW AS NEW_ORD
      FOR EACH ROW MODE DB2SQL
         BEGIN ATOMIC
        UPDATE SALESREPS
           SET SALES = SALES + NEW_ORD.AMOUNT
         WHERE SALESREPS.EMPL_NUM = NEW_ORD.REP;
        UPDATE PRODUCTS
           SET QTY_ON_HAND = QTY_ON_HAND - NEW_ORD.QTY
```

```
WHERE PRODUCTS.MFR_ID = NEW_ORD.MFR
          AND PRODUCTS.PRODUCT_ID = NEW_ORD.PRODUCT;
          END
```

Первая часть определения триггера включает те же элементы, которые используются и в SQL Server, только в несколько иной последовательности. Вторая строка явно указывает на то, что триггер вызывается *после* (AFTER) добавления заказа в базу данных. DB2 позволяет также выполнять триггер непосредственно *перед* выполнением связанной с ним операции. В данном случае это смысла не имеет, поскольку триггер связан с инструкцией INSERT, но в случае инструкций UPDATE и DELETE это вполне может потребоваться.

Предложение REFERENCING в DB2 задает *псевдоним таблицы* (NEW_ORD), который будет использоваться для ссылки на добавляемую строку в оставшейся части определения триггера. Функционально этот псевдоним аналогичен ключевому слову INSERTED в SQL Server. Ключевое слово NEW указано потому, что строка добавляется. В случае инструкции DELETE потребовалось бы указать слово OLD. Для инструкции UPDATE DB2 позволяет обращаться к обоим значениям — NEW (ссылается на таблицу после обновления) и OLD (ссылается на таблицу перед обновлением).

Предложения BEGIN ATOMIC и END по сути служат скобками вокруг SQL-инструкций, определяющих действие триггера. Две поисковые инструкции UPDATE в теле определения триггера являются простыми модификациями своих аналогов из SQL Server. Они следуют стандартному синтаксису SQL для поисковых инструкций UPDATE с использованием псевдонима таблицы, указанного в предложении REFERENCING для определения конкретной обновляемой строки таблицы SALESREPS и таблицы PRODUCTS. Обращение к вставляемой строке выполняется с помощью имени псевдотаблицы в инструкции UPDATE.

Триггеры и ссылочная целостность

Триггеры обеспечивают альтернативный способ реализации ограничений ссылочной целостности с использованием первичных и внешних ключей. Фактически сторонники триггеров указывают, что механизм триггеров более гибок, чем строгая ссылочная целостность, обеспечиваемая стандартом ANSI/ISO. Однако оппоненты триггеров апеллируют к тому, что поведение триггеров сильно отличается в разных СУБД, в частности в способе отката транзакций или в том, как работает блокировка в процессе выполнения триггера. Например, вот как выглядит триггер SQL Server, который обеспечивает ссылочную целостность отношения таблиц OFFICES/SALESREPS и выводит сообщение при неудачной попытке обновления.

```
CREATE TRIGGER REP_UPDATE
    ON SALESREPS
   FOR INSERT, UPDATE
    AS IF ((SELECT COUNT(*)
              FROM OFFICES, INSERTED
             WHERE OFFICES.OFFICE = INSERTED.REP_OFFICE) = 0)
       BEGIN
          PRINT "Указан неверный идентификатор офиса."
          ROLLBACK TRANSACTION
       END;
```

Триггеры можно использовать и для обеспечения расширенных форм ссылочной целостности. Первоначально в DB2 с помощью правила CASCADE обеспечивались каскадные удаления, но не поддерживались каскадные обновления при изменении значения первичного ключа. Однако триггеров это ограничение не касается. Следующий триггер в SQL Server распространяет любое обновление столбца OFFICE в таблице OFFICES на столбец REP_OFFICE таблицы SALESREPS.

```
CREATE TRIGGER CHANGE_REP_OFFICE
    ON OFFICES
   FOR UPDATE
    AS IF UPDATE (OFFICE)
       BEGIN
          UPDATE SALESREPS
             SET SALESREPS.REP_OFFICE = INSERTED.OFFICE
            FROM SALESREPS, INSERTED, DELETED
           WHERE SALESREPS.REP_OFFICE = DELETED.OFFICE
       END;
```

Как и в предыдущих примерах с SQL Server, ссылки DELETED.OFFICE и INSERTED.OFFICE в триггере относятся к значениям столбца OFFICE соответственно до и после выполнения инструкции UPDATE. Для корректного выполнения поиска и модификации данных определение триггера должно быть способно различать значения до и после обновления.

Преимущества и недостатки триггеров

В течение нескольких последних лет механизмы триггеров во многих коммерческих СУБД были существенно расширены. Во многих коммерческих реализациях различия между триггерами и хранимыми процедурами (описанными в главе 20, "Хранимые процедуры SQL") весьма размыты, так что действия, выполняемые при единственном изменении базы данных, могут быть определены сотнями строк программы хранимой процедуры. Таким образом, роль триггеров вышла за рамки обеспечения целостности данных и превратилась в подраздел программирования в базах данных.

Триггеры выходят за рамки данной книги, однако даже из приведенных простых примеров видно, насколько это мощный механизм. Основным преимуществом триггеров является то, что создаваемые с их помощью бизнес-правила можно хранить в базе данных и применять при каждой операции добавления или обновления. Это позволяет существенно уменьшить сложность прикладной программы, работающей с базой данных. Однако у триггеров имеются и некоторые недостатки, включая следующие.

- **Сложность базы данных**. Когда бизнес-правила становятся частью базы данных, она существенно усложняется. Пользователи, которые рассчитывали создавать на основе этой базы данных маленькие специализированные приложения, могут столкнуться с тем, что программная логика триггеров делает эту задачу гораздо более сложной.
- **Скрытые правила**. Когда бизнес-правила скрыты в базе данных, программы, осуществляющие казалось бы обычные обновления данных, мо-

гут в действительности привести к выполнению огромного объема вычислений. У программиста исчезает возможность управлять всеми процессами, происходящими в базе данных, поскольку программные запросы могут вызывать выполнение различных других скрытых действий.

- **Скрытое влияние на производительность**. Когда в базе данных имеются триггеры, последовательность выполнения SQL-инструкций становится неочевидной для программиста. В частности, даже самая простая инструкция теоретически может привести к сканированию огромной таблицы, что отнимает очень много времени. В результате для программиста может быть совершенно непонятно, почему введенная им инструкция выполняется так долго.

Триггеры и стандарты SQL

С момента своего появления триггеры были одной из самых разрекламированных характеристик Sybase SQL Server, и с тех пор они нашли свое применение во многих коммерческих СУБД. И хотя при составлении стандарта SQL2 была возможность унифицировать их реализацию, комитет по стандартизации предложил вместо них ввести ограничение CHECK, оставляя триггеры следующей версии стандарта (SQL:1999, известная также как SQL3). Как показывают приведенные ранее примеры, подобное ограничение может успешно использоваться для контроля данных, добавляемых в таблицу или модифицируемых в ней. Однако, в отличие от триггеров, оно не может вызвать выполнение в базе данных независимых действий, таких как добавление строки или изменение данных в другой таблице.

Некоторые эксперты доказывают, что триггеры представляют собой загрязнение функции базы данных управления данными и что функции, выполняемые триггерами, принадлежат к четвертому поколению языков (4GL) и сторонним инструментам баз данных, но не самой СУБД. Пока продолжаются дебаты, СУБД продолжают экспериментировать с новыми возможностями триггеров, выходящими за рамки баз данных. Эти расширенные возможности позволяют модификациям данных в базе данных автоматически выполнять такие действия, как отправка электронной почты, уведомление пользователя или запуск некоторой программы. Это делает триггеры еще более полезными, а споры еще более жаркими. В любом случае нет сомнений, что в последние годы триггеры становятся все более и более важной частью SQL в приложениях уровня предприятия.

Резюме

В SQL имеется ряд средств, помогающих сохранять целостность данных, содержащихся в реляционной базе данных.

- При создании таблицы можно определить столбцы, которые обязательно должны содержать данные, и СУБД будет запрещать ввод в них значений NULL.

- В стандартном SQL проверка данных на правильность ограничена проверкой их типа, однако во многих СУБД имеются и другие средства проверки.
- Условие целостности таблицы гарантирует, что первичный ключ однозначно идентифицирует каждую сущность, представленную строкой в базе данных.
- Условие ссылочной целостности гарантирует, что связи между сущностями в базе данных сохраняются при изменении данных (добавлении, обновлении или удалении).
- Стандарт SQL и новые реализации обеспечивают дополнительную поддержку ссылочной целостности, включая правила удаления и обновления, указывающие СУБД, как следует обрабатывать удаления и обновления строк, на которые ссылаются другие строки.
- Выполнение бизнес-правил можно обеспечить с помощью механизма триггеров, популяризованного Sybase и SQL Server и позже включенного в стандарт SQL:1999. Триггеры позволяют СУБД предпринимать сложные действия в ответ на такие события, как попытка выполнить инструкции `INSERT`, `DELETE` и `UPDATE`. По сравнению с триггерами, ограничения на значения столбцов обеспечивают менее эффективный способ включения бизнес-правил в определение базы данных.

12
ГЛАВА

Обработка транзакций

Обычно изменения в базе данных обусловлены событиями, происходящими в моделируемом внешнем мире, такими, например, как прием нового заказа от клиента. При этом подобное событие приводит не к одному, а к *четырем* изменениям в учебной базе данных:

- добавление нового заказа в таблицу ORDERS;
- обновление фактического объема продаж для служащего, принявшего заказ;
- обновление фактического объема продаж для офиса, в котором работает данный служащий;
- обновление количества товара, имеющегося в наличии.

Чтобы не нарушить целостность базы данных, четыре указанных изменения следует выполнить как единое целое. Если из-за системного сбоя или другой ошибки получится, что одна часть изменений была внесена, а другая — нет, то это нарушит целостность хранимых данных и при последующих вычислениях результаты окажутся неверными. Точно так же, если другие пользователи будут выполнять вычисления итоговых значений или некоторых отношений во время внесения указанных изменений, они получат некорректные результаты. Поэтому изменения базы данных, которые вызваны одним событием, необходимо вносить по принципу "либо все, либо ничего". SQL обеспечивает такое поведение посредством возможностей обработки транзакций, которые и рассматриваются в данной главе.

Что такое транзакция

Транзакция — это несколько последовательных инструкций SQL, которые вместе образуют логическую единицу работы. Инструкции, входящие в транзакцию, обычно тесно связаны между собой и выполняют взаимосвязанные действия. Каждая инструкция решает часть общей задачи, но для того, чтобы задачу можно было считать

решенной, требуется выполнить *все* эти инструкции. Группировка инструкций в единую транзакцию указывает СУБД, что вся их последовательность должна выполняться так, чтобы пройти так называемый ACID-тест. ACID — аббревиатура, обычно используемая для обозначения четырех характеристик транзакции.

- ***A*tomic (Атомарность)**. "Девиз" транзакции — или все, или ничего. Либо успешно выполняются все операции транзакции, либо не выполняется ни одна из них. Если выполнены лишь некоторые инструкции, то транзакция оказывается неуспешной, и в результате будет выполнен откат выполненных инструкций. Только когда все инструкции выполнены корректно, транзакция может рассматриваться как завершенная, а ее результаты фиксируются в базе данных.
- ***C*onsistent (Целостность)**. Транзакция должна переводить базу данных из одного согласованного состояния в другое. База данных должна быть в согласованном состоянии по окончании каждой транзакции, а это означает, что должны выполняться все правила и ограничения. Ни один пользователь не должен иметь доступ к данным, несогласованным из-за незавершенности транзакции.
- ***I*solated (Изолированность)**. Каждая транзакция должна выполняться сама по себе, без взаимодействия с другими транзакциями. Для этого ни одна транзакция не должна работать с изменениями, вносимыми другой транзакцией, пока та не будет завершена.
- ***D*urable (Постоянство)**. По завершении транзакции все внесенные ею изменения должны быть сохранены. Данные должны быть в согласованном состоянии, даже если по окончании транзакции произойдет аппаратный или программный сбой. В объектно-ориентированном программировании для этого свойства используется термин *персистентность* (persistence).

Ниже приведен ряд типичных примеров транзакций для учебной базы данных, а также описаны инструкции, входящие в эти транзакции.

- **Прием заказа**. Для приема заказа от клиента программа ввода заказов должна: (а) выполнить запрос к таблице `PRODUCTS` и проверить наличие товара на складе; (б) добавить заказ в таблицу `ORDERS`; (в) обновить таблицу `PRODUCTS`, вычтя заказанное количество товара из количества товара, имеющегося в наличии; (г) обновить таблицу `SALESREPS`, добавив стоимость заказа к объему продаж служащего, принявшего заказ; и (д) обновить таблицу `OFFICES`, добавив стоимость заказа к объему продаж офиса, в котором работает данный служащий.
- **Отмена заказа**. Чтобы отменить заказ, принятый от клиента ранее, программа должна: (а) удалить заказ из таблицы `ORDERS`; (б) обновить таблицу `PRODUCTS`, откорректировав количество товара, имеющегося в наличии; (в) обновить таблицу `SALESREPS`, вычтя стоимость заказа из объема продаж служащего; и (г) обновить таблицу `OFFICES`, вычтя стоимость заказа из объема продаж офиса.

- **Перевод клиента**. При переводе клиента от одного служащего к другому программа должна (а) соответствующим образом обновить таблицу CUSTOMERS; (б) обновить таблицу ORDERS, изменив имя служащего, ответственного за заказы данного клиента; (в) обновить таблицу SALESREPS, уменьшив план для служащего, теряющего клиента; и (г) обновить таблицу SALESREPS, увеличив план служащего, приобретающего клиента.

Во всех указанных случаях для обработки одной логической транзакции требуется выполнить последовательность из четырех или пяти действий, каждое из которых представлено отдельной инструкцией SQL.

Понятие транзакции является критичным для программ, изменяющих содержимое базы данных, так как позволяет обеспечить ее целостность. Транзакции в реляционной СУБД подчиняются следующему правилу.

Инструкции, входящие в транзакцию, выполняются атомарно, как единое неделимое целое. Либо все инструкции будут выполнены успешно, либо ни одна из них не должна быть выполнена.

Как следует из рис. 12.1, СУБД должна следовать этому правилу, даже если во время выполнения транзакции произойдет ошибка в программе или аппаратный сбой. В любом случае СУБД должна гарантировать, что при восстановлении базы данных после сбоя частичное выполнение транзакции в ней отражено не будет.

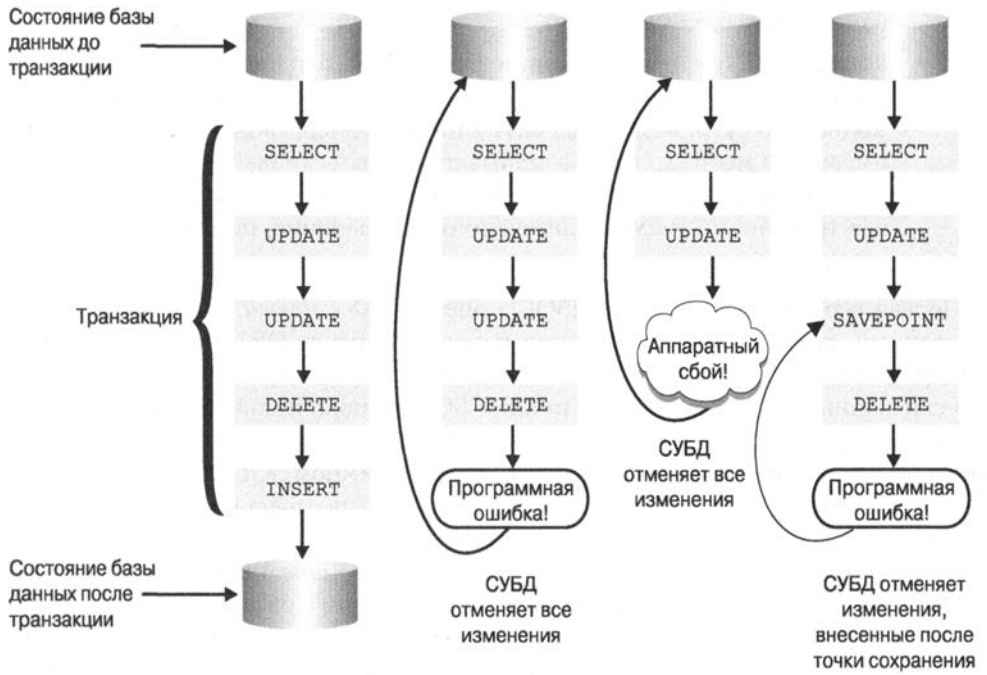


Рис. 12.1. Концепция транзакции в SQL

Модель транзакции ANSI/ISO SQL

В стандарте ANSI/ISO определена *модель транзакций* SQL, а также семь инструкций для поддержки работы с транзакциями.

- **START TRANSACTION**. Устанавливает свойства новой транзакции и запускает транзакцию.
- **SET TRANSACTION**. Устанавливает свойства очередной выполняемой транзакции. Не влияет на текущую выполняемую транзакцию.
- **SET CONSTRAINTS**. Устанавливает режим ограничений в текущей транзакции. Режим ограничений управляет тем, применяется ли ограничение немедленно или откладывается до более позднего момента. Инструкция SET CONSTRAINTS представлена в главе 11, "Целостность данных".
- **SAVEPOINT**. Создает точку сохранения в пределах транзакции. Точка сохранения представляет собой место в последовательности событий транзакции, которое может выступать в качестве промежуточной точки восстановления. Откат текущей транзакции может быть выполнен не к началу транзакции, а к точке сохранения.
- **RELEASE SAVEPOINT**. Освобождает точку сохранения и все ресурсы, которые она могла захватить.
- **COMMIT**. Завершает успешную транзакцию и сохраняет все внесенные изменения в базе данных.
- **ROLLBACK**. При использовании без точки сохранения прекращает неудачную транзакцию и выполняет откат всех изменений к началу транзакции, по сути, возвращая базу данных к ее согласованному состоянию, имевшему место до начала транзакции (как если бы транзакция никогда не выполнялась). При использовании с точкой сохранения выполняет откат транзакции к именованной точке сохранения, но допускает продолжение выполнения транзакции.

Первая версия стандарта SQL (SQL1) определяла *неявный* режим транзакции, основанный на поддержке транзакций в ранних версиях DB2. В неявном режиме поддерживались только инструкции COMMIT и ROLLBACK. Транзакция SQL автоматически начиналась с первой инструкции SQL, выполняемой пользователем или программой, и завершалась выполнением COMMIT или ROLLBACK. Завершение одной транзакции неявно начинало новую. В программном SQL успешное завершение программы обрабатывалось, как если бы была выполнена инструкция COMMIT, а аварийное завершение программы приравнивалось к ROLLBACK. Многие коммерческие продукты, в частности DB2 и Oracle, все еще по умолчанию входят в режим неявной транзакции при первом подключении к базе данных.

Версии SQL2 и SQL:1999 стандарта ANSI/ISO SQL добавили новые инструкции, приведенные в предыдущем списке. Они поддерживаются более современными коммерческими продуктами, однако имеются и исключения. Так, Sybase ASE и SQL SERVER вместо START TRANSACTION поддерживают инструкцию BEGIN TRANSACTION и SAVE TRANSACTION — вместо SAVEPOINT.

Инструкции START TRANSACTION и SET TRANSACTION

Синтаксические диаграммы инструкций START TRANSACTION и SET TRANSACTION показаны на рис. 12.2. Фундаментальное различие между ними заключается в том, что START TRANSACTION начинает новую транзакцию с определенным набором свойств, в то время как инструкция SET TRANSACTION устанавливает свойство *следующей* транзакции — она не может использоваться в существующей транзакции и, таким образом, не в состоянии влиять на выполняющуюся транзакцию. Еще одно отличие заключается в том, что инструкция SET TRANSACTION может применяться с необязательным ключевым словом LOCAL, которое используется для установки свойств при выполнении распределенной на несколько серверов транзакции на локальном сервере.

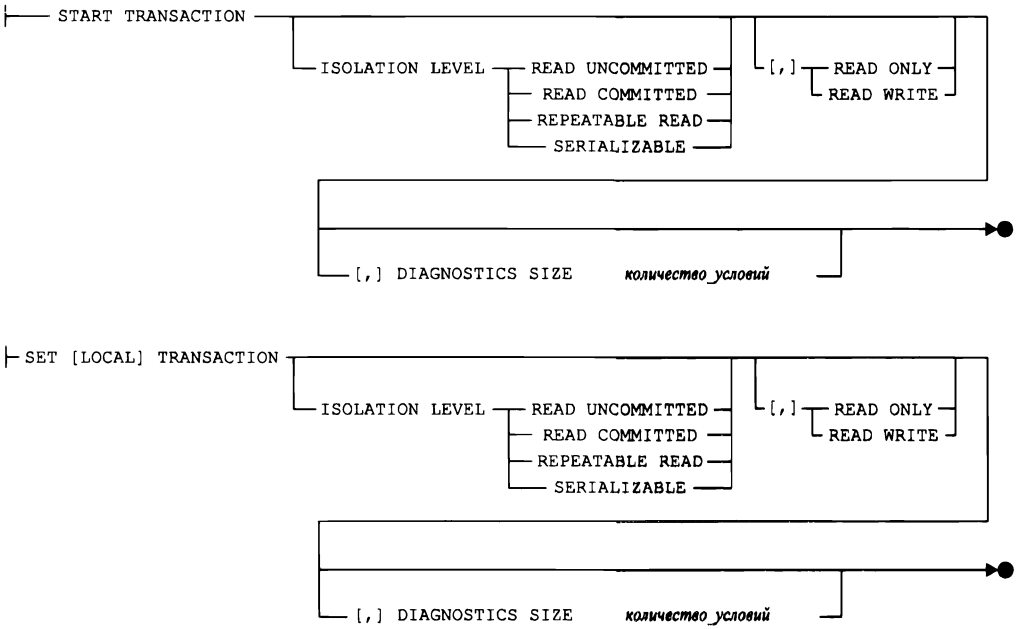


Рис. 12.2. Инструкции START TRANSACTION и SET TRANSACTION

Эти инструкции могут устанавливать следующие три свойства транзакций (в любом порядке).

- **Уровень изоляции**. Определяет, насколько транзакция изолирована от действия других транзакций. Соответствующие параметры (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ и SERIALIZABLE) детально рассматриваются ниже в данной главе. Если уровень изоляции не указан, по умолчанию используется SERIALIZABLE.
- **Уровень доступа**. Определяет, может ли транзакция содержать инструкции, которые модифицируют базу данных (READ WRITE), или не может содержать таковые (READ ONLY). Уровень по умолчанию зависит от

выбранного уровня изоляции, но если таковой не указан, то по умолчанию используется уровень доступа READ WRITE.

- **Размер диагностики**. Определяет размер области диагностики, используемой для условий, которые могут генерироваться при выполнении инструкций SQL. *Условие* представляет собой предупреждение, исключение или иной тип сообщения, генерируемого при выполнении инструкции SQL. Например, если размер диагностики равен 10, то для выполняемой инструкции может сохраняться до 10 условий. Заметим, что во время написания этого материала это свойство не поддерживалось ни SQL Server, ни Sybase ASE, ни Oracle, ни MySQL, ни DB2, ни большинством прочих текущих реализаций SQL.

Вот пример инструкции START TRANSACTION, которая устанавливает уровень изоляции READ UNCOMMITTED, уровень доступа READ ONLY и размер диагностики, равный 5.

```
START TRANSACTION
      ISOLATION LEVEL READ UNCOMMITTED,
      READ ONLY,
      DIAGNOSTICS SIZE 5;
```

Инструкции SAVEPOINT и RELEASE SAVEPOINT

Как указывалось ранее, инструкция SAVEPOINT устанавливает в транзакции точку, к которой может быть выполнен откат транзакции при помощи последующей инструкции ROLLBACK. Ее синтаксис весьма прост; она имеет единственный параметр, который представляет собой уникальное имя точки сохранения в пределах транзакции.

```
SAVEPOINT имя_точки_сохранения;
```

Очевидное преимущество применения точек сохранения заключается в возможности отката части транзакции в случае небольших и потенциально восстановимых ошибок. Примером может служить крайняя справа транзакция на рис. 12.1. В большинстве реализаций транзакции могут иметь любое необходимое количество точек сохранения, лишь бы у каждой из них было свое уникальное в пределах транзакции имя. Например, приложение для ввода заказов может создавать точку сохранения после каждой введенной строки заказа. Если добавление новой строки из заказа приводит к превышению лимита кредита клиента, приложение может выполнить откат к точке сохранения, установленной непосредственно перед этой строкой. Откат должен отменить строку, которая вызвала проблемы, вывести соответствующее сообщение оператору, который вводит заказ, и позволить продолжить работу — например, вводить менее дорогой товар, который не вызовет таких проблем.

Недостатком точек сохранения является потенциальное использование большого количества ресурсов (дисковой и/или оперативной памяти). Хотя завершение транзакции автоматически освобождает все точки сохранения, иногда имеет смысл явно освобождать те точки сохранения, которые больше не нужны. Это можно сделать при помощи инструкции RELEASE SAVEPOINT, синтаксис которой также очень прост.

```
RELEASE SAVEPOINT имя_точки_сохранения;
```

Инструкции COMMIT и ROLLBACK

SQL поддерживает две инструкции SQL, явно завершающие транзакции. Эти инструкции показаны на рис. 12.3. Инструкции могут сопровождаться следующими параметрами.

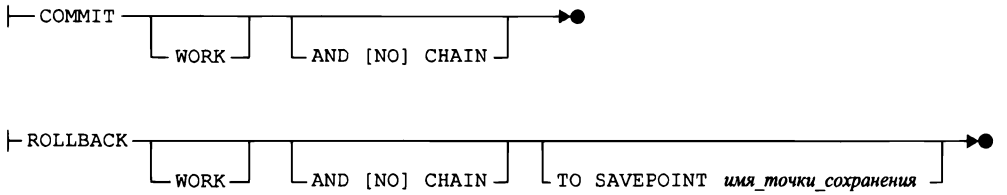


Рис. 12.3. Синтаксические диаграммы инструкций COMMIT **и** ROLLBACK

- **WORK**. Это ключевое слово не выполняет никаких действий и включено в стандарт исключительно для совместимости с некоторыми реализациями SQL, требующими его наличия.
- **AND [NO] CHAIN**. Определяет, должна ли новая транзакция автоматически начинаться с теми же свойствами, что и только что завершившаяся. На момент написания этого материала данный параметр не поддерживался ни одной из СУБД Oracle, SQL Server, DB2 UDB или MySQL.
- **TO SAVEPOINT**. Этот параметр применим только в инструкции ROLLBACK. Параметр указывает, что откат должен быть выполнен только до определенной точки сохранения, созданной ранее в транзакции, а не к началу транзакции.

Инструкции COMMIT и ROLLBACK представляют собой выполнимые инструкции SQL, как и SELECT, INSERT, UPDATE или DELETE. Ниже приведен пример успешной обновляющей транзакции, которая изменяет объем и сумму заказа и обновляет итоговые значения для товара, продавца и офиса, связанных с этим заказом. Такие изменения обычно обрабатываются специальной программой на основе форм ввода информации, которая для выполнения приведенных инструкций применяет программный SQL.

Изменить объем заказа 113051 с 4 до 10 единиц, что повышает его сумму с $1458 до $3550. Заказ на товар QSA-XK47 был принят Ларри Фитчем (персональный номер 108), который работает в Лос-Анджелесе (офис номер 21).

```
UPDATE ORDERS
   SET QTY = 10, AMOUNT = 3550.00
 WHERE ORDER_NUM = 113051;

UPDATE SALESREPS
   SET SALES = SALES - 1458.00 + 3550.00
 WHERE EMPL_NUM = 108;

UPDATE OFFICES
```

```
   SET SALES = SALES - 1458.00 + 3550.00
 WHERE OFFICE = 21;

UPDATE PRODUCTS
   SET QTY_ON_HAND = QTY_ON_HAND + 4 - 10
 WHERE MFR_ID = 'QSA'
   AND PRODUCT_ID = 'XK47';
```

. . . последнее подтверждение введенной информации оператором . . .

```
COMMIT WORK;
```

Вот та же транзакция, но в этот раз предполагается, что оператор сделал ошибку при вводе номера товара. Для коррекции ошибки транзакция выполняет откат, так что можно повторно внести корректную информацию.

Изменить объем заказа 113051 с 4 до 10 единиц, что повышает его сумму с $1458 до $3550. Заказ на товар QSA-XK47 был принят Ларри Фитчем (персональный номер 108), который работает в Лос-Анджелесе (офис номер 21).

```
UPDATE ORDERS
   SET QTY = 10, AMOUNT = 3550.00
 WHERE ORDER_NUM = 113051;

UPDATE SALESREPS
   SET SALES = SALES - 1458.00 + 3550.00
 WHERE EMPL_NUM = 108;

UPDATE OFFICES
   SET SALES = SALES - 1458.00 + 3550.00
 WHERE OFFICE = 21;

UPDATE PRODUCTS
   SET QTY_ON_HAND = QTY_ON_HAND + 4 - 10
 WHERE MFR_ID = 'QAS'
   AND PRODUCT_ID = 'XK47';
```

. . . ой! производитель — "QSA", а не "QAS" . . .

```
ROLLBACK WORK;
```

На рис. 12.4 показаны типичные транзакции, иллюстрирующие разные ситуации, возникающие при работе базы данных.

Вспомним, что стандарт ANSI/ISO SQL, в первую очередь, фокусируется на *программном* SQL, предназначенном для использования прикладными программами. В программном SQL транзакции играют очень важную роль, поскольку даже простая прикладная программа зачастую требует для решения стоящей перед ней задачи выполнить последовательность из нескольких инструкций SQL. Поскольку клиенты могут изменять свои намерения, да и от других ситуаций, таких как нехватка товара на складе, тоже никто не застрахован, прикладная программа должна быть способна к частичному выполнению транзакции с последующим выбором — отказ от сделанного или продолжение работы. Эту возможность и обеспечивают инструкции COMMIT и ROLLBACK.

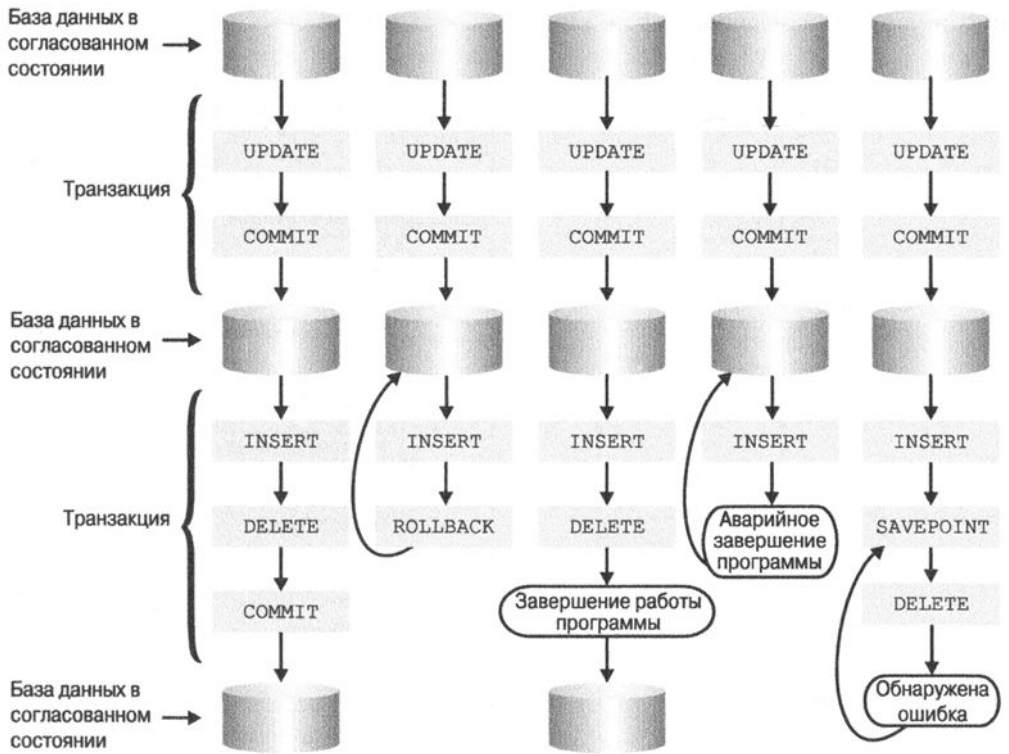


Рис. 12.4. Транзакции — принятие и откат

Инструкции COMMIT и ROLLBACK могут также применяться и в интерактивном SQL, но на практике они редко встречаются в данном контексте. Интерактивный SQL обычно используется для запросов к базе данных; обновление информации — существенно реже встречающаяся задача, а уж обновления, которые включают несколько инструкций, практически не встречаются при работе интерактивного SQL. В результате, транзакции в интерактивном SQL, по сути, не используются. На практике многие интерактивные SQL-продукты по умолчанию работают в режиме автосохранения, когда инструкция COMMIT автоматически выполняется после каждой введенной пользователем инструкции SQL. Это, по сути, делает каждую инструкцию SQL своей собственной транзакцией.

Транзакции: что за сценой*

Реализация в СУБД принципа "все или ничего" по отношению к инструкциям транзакции кажется новичку в SQL почти чудом. Каким образом СУБД может отменить изменения, внесенные в базу данных, в особенности если во время выполнения транзакции происходит системная ошибка? В различных СУБД для этого используются различные методы, но почти все они, как правило, основаны на применении *журнала транзакций* (рис. 12.5). Несмотря на широкое применение термина *журнал* (log), некоторые СУБД применяют более сложные механизмы для хранения

информации об изменениях в базе данных, необходимой для восстановления. Oracle, например, хранит эту информацию в специальных сегментах базы данных.

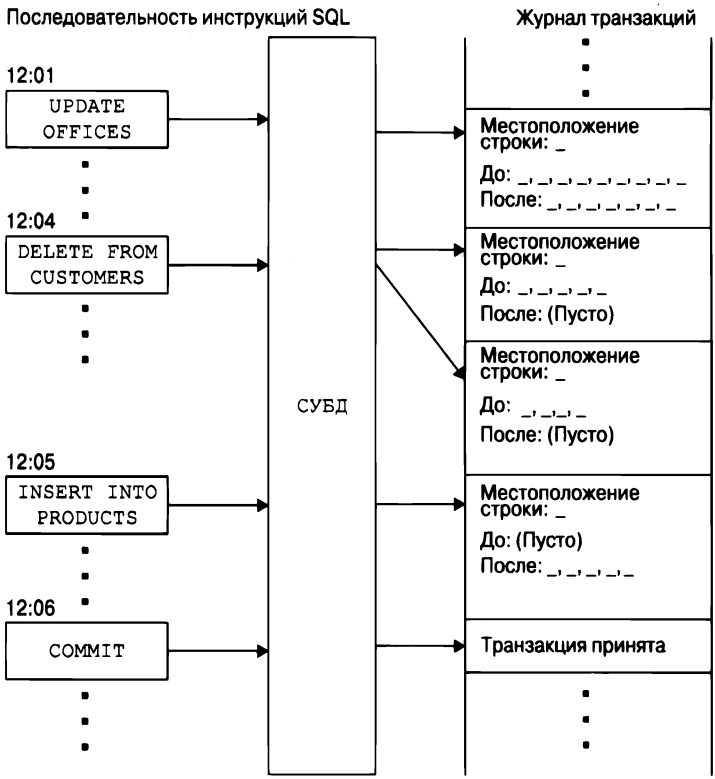


Рис. 12.5. Журнал транзакций

Вот как можно упрощенно, концептуально, описать работу журнала транзакций. Когда пользователь выполняет запрос на изменение базы данных, СУБД автоматически вносит в журнал транзакций одну запись для каждой строки, измененной в процессе выполнения запроса. Эта запись содержит две копии строки. Одна копия представляет собой строку *до* изменения, а другая — *после* изменения. СУБД изменяет физическую строку на диске только после того, как в журнале будет сделана соответствующая запись. Затем, если пользователь выполняет инструкцию COMMIT, в журнале отмечается конец транзакции. Если же пользователь выполняет инструкцию ROLLBACK, СУБД обращается к журналу и извлекает из него исходные копии строк, измененных во время транзакции. Используя эти копии, СУБД возвращает строки в прежнее состояние и таким образом отменяет изменения, внесенные в базу данных в ходе транзакции.

В более старых СУБД в случае системного сбоя администратор базы данных восстанавливал ее с помощью специальной утилиты восстановления, поставляемой вместе с СУБД. В более новых базах данных операция восстановления обычно запускается автоматически при первой же возможности. Утилита восстановления просматривает журнал транзакций и отыскивает транзакции, которые не были за-

вершены к моменту сбоя. Затем утилита отменяет все незавершенные транзакции; таким образом, в базе данных будут отражены только завершенные транзакции. Транзакции, которые выполнялись в момент системного сбоя, также отменяются.

Очевидно, что в результате использования журнала транзакций увеличивается время, необходимое для внесения изменений в базу данных. На практике, чтобы минимизировать эти затраты, в коммерческих СУБД применяются гораздо более интеллектуальные схемы ведения журнала транзакций, чем описанная здесь. Кроме того, с целью уменьшения времени доступа журнал транзакций обычно хранится на отдельном высокоскоростном жестком диске, а не там, где хранится сама база данных. Некоторые СУБД для персональных компьютеров даже позволяют пользователю отключать журнал транзакций для увеличения производительности СУБД, например, при загрузке больших объемов данных. Но если для небольших приложений это может быть приемлемо, то в промышленных базах данных ведение журнала транзакций является строго обязательным, неотделимым от других операций базы данных.

Транзакции и работа в многопользовательском режиме

Если с базой данных одновременно работают двое или более пользователей, СУБД должна не только осуществлять восстановление базы данных после отмены транзакции или системного сбоя, но и гарантировать, что пользователи не будут мешать друг другу. В идеальном случае каждый пользователь должен работать с базой данных так, как если бы он имел к ней монопольный доступ, и не должен беспокоиться о действиях других пользователей. Средства обработки транзакций в SQL позволяют реляционным СУБД изолировать пользователей друг от друга именно таким образом.

Лучший способ понять, как выполняются параллельные транзакции, — это рассмотреть проблемы, которые могут возникнуть, если транзакции не будут корректно обработаны. Такие проблемы можно разбить на четыре основные категории, рассматриваемые в четырех последующих разделах.

Проблема пропавшего обновления

На рис. 12.6 изображена схема работы простой прикладной программы, с помощью которой двое служащих принимают заказы от клиентов. Перед вводом заказа программа по таблице PRODUCTS проверяет, имеется ли требуемый товар в наличии. На данном рисунке Джо начинает вводить заказ от своего клиента на 100 изделий ACI-41004. В это же время Мери начинает вводить заказ от своего клиента на 125 тех же изделий ACI-41004. Обе программы для ввода заказов выполняют запрос к таблице PRODUCTS, и каждая из них выясняет, что на складе имеется 139 требуемых изделий — более чем достаточно для выполнения заказа. Джо просит клиента подтвердить заказ; затем его копия программы обновляет таблицу PRODUCTS, показывая, что в наличии осталось (139–100)=39 изделий ACI-41004, и добавляет в таблицу ORDERS новый заказ на 100 изделий. Через несколько секунд Мери просит своего клиента подтвердить заказ. Ее копия программы обновляет

таблицу PRODUCTS, показывая, что на складе осталось (139–125)=14 изделий ACI-41004, и добавляет в таблицу ORDERS новый заказ на 125 изделий.

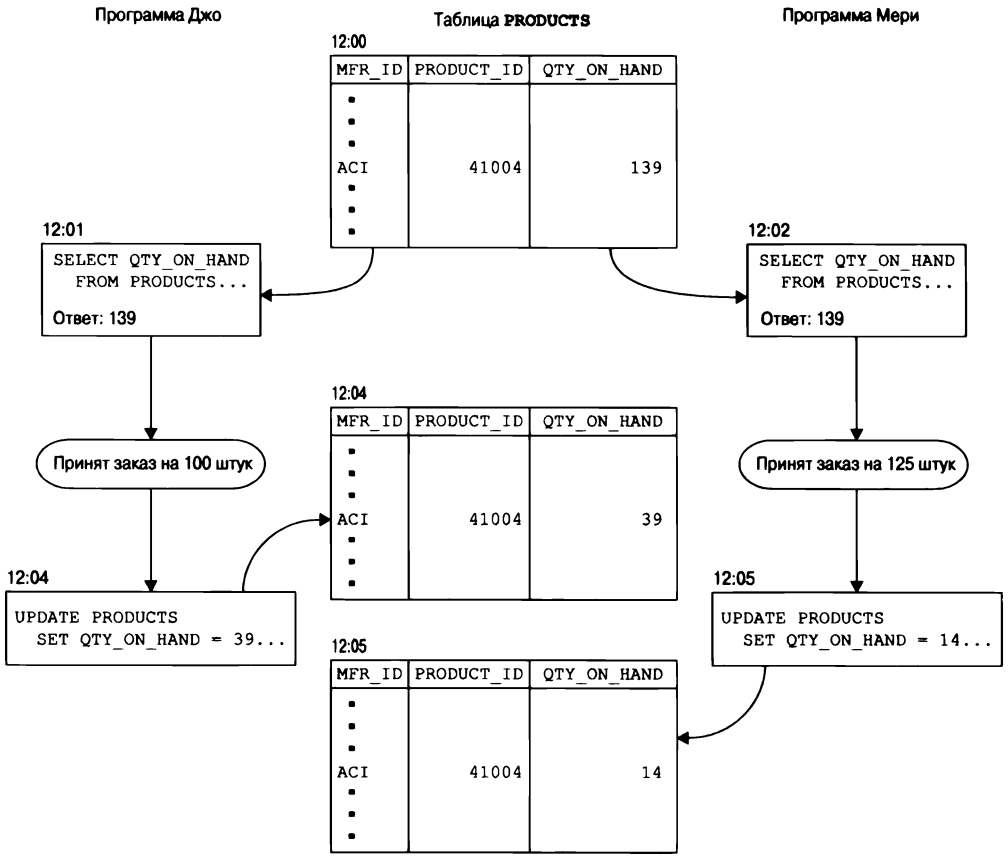


Рис. 12.6. Проблема пропавшего обновления

Очевидно, что обработка этих двух заказов привела к возникновению противоречия в базе данных. Первое из двух обновлений таблицы PRODUCTS пропало! Заказы от обоих клиентов приняты, но на складе нет достаточного количества изделий для удовлетворения обоих заказов. Более того, таблица PRODUCTS показывает, что в наличии осталось еще 14 изделий! Данный пример показывает, что проблема пропавшего обновления может возникнуть всякий раз, когда две программы извлекают из базы одни и те же данные, используют их для каких-либо расчетов, а затем пытаются обновить эти данные.

Проблема промежуточных данных

На рис. 12.7 изображена схема работы той же программы для обработки заказов, что и на рис. 12.6. Джо снова начинает принимать от клиента заказ на 100 изделий ACI-41004. На этот раз его копия программы запрашивает таблицу PRODUCTS, выясняет, что в наличии имеется 139 изделий, и обновляет таблицу PRODUCTS, показывая,

что после принятия заказа в наличии осталось 39 изделий. После этого Джо начинает обсуждать с клиентом относительные достоинства изделий ACI-41004 и ACI-41005. Тем временем клиент Мери пытается заказать 125 изделий ACI-41004. Ее копия программы запрашивает таблицу PRODUCTS, выясняет, что в наличии имеется только 39 изделий, и отказывается принять заказ. Программа генерирует также сообщение для менеджера по снабжению о том, что необходимо закупить изделия ACI-41004, которые пользуются большим спросом. А первый клиент после беседы с Джо решает, что изделия ACI-41004 ему вовсе не нужны, и программа выполняет инструкцию ROLLBACK для отмены транзакции.

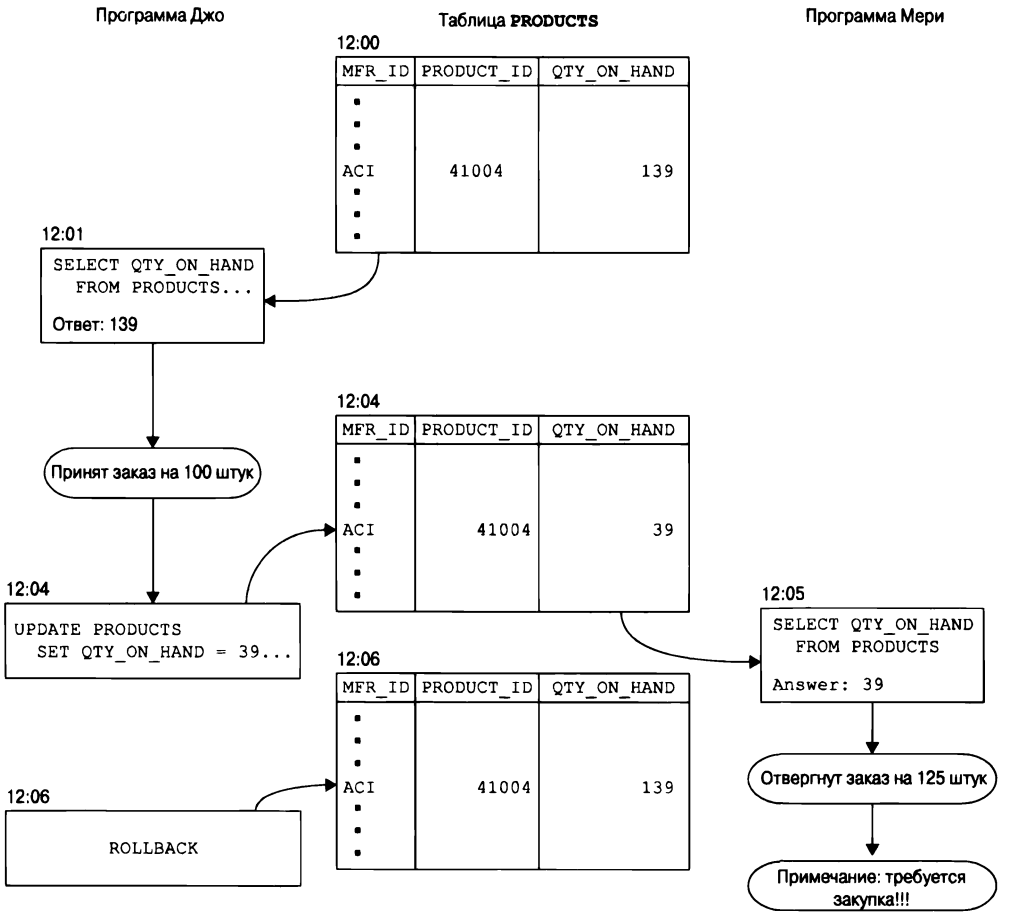


Рис. 12.7. Проблема промежуточных данных

Поскольку программа Мери имела доступ к промежуточным данным программы Джо, ее клиенту было отказано в приеме заказа, а менеджер по снабжению закажет дополнительное количество изделий ACI-41004, хотя их на складе и так 139 штук. Ситуация могла бы быть еще хуже, если бы клиент Мери решил заказать 39 изделий. В этом случае программа Мери занесла бы в таблицу PRODUCTS число нуль, показывая, что изделия ACI-41004 на складе отсутствуют.

Но после операции ROLLBACK, отменяющей транзакцию Джо, СУБД восстановила бы в таблице число 139, хотя 39 изделий из них уже предназначены клиенту Мери. В данном примере проблема заключается в том, что программа Мери имела доступ к промежуточным результатам работы программы Джо и, опираясь на них, занесла в базу данных ошибочную информацию. В стандарте SQL эта проблема обозначена как *P1*, или проблема *промежуточного чтения* (dirty read). Выражаясь языком стандарта, данные, которые извлекала программа Мери, являлись промежуточными, так как еще не были подтверждены программой Джо.

Проблема несогласованных данных

На рис. 12.8 изображена еще одна схема работы программы для обработки заказов. Джо снова начинает принимать от своего клиента заказ на 100 изделий ACI-41004. Вскоре после этого Мери начинает со своим клиентом разговор об этих же изделиях, и ее программа выясняет их количество на складе. Затем клиент начинает расспрашивать Мери об изделиях ACI-41005, и программа Мери выполняет запрос о наличии этих изделий. Тем временем клиент Джо решает заказать изделия ACI-41004, поэтому его программа обновляет соответствующую строку и выполняет инструкцию COMMIT, завершая транзакцию по приему заказа. После некоторых размышлений клиент Мери решает заказать изделия ACI-41004, которые Мери предлагала ему вначале. Ее программа вновь запрашивает информацию об изделиях ACI-41004. Но новый запрос показывает, что в наличии имеется только 39 изделий, вместо 139, показанных предыдущим запросом несколько секунд тому назад.

В данном примере, в отличие от двух предыдущих, состояние базы данных правильно отражает реальную ситуацию. В наличии осталось только 39 изделий ACI-41004, поскольку клиент заказал у Джо 100 штук. Из-за того что Мери увидела промежуточные данные программы Джо, ничего особенного не произошло — заказ от Джо был успешно принят. Однако, с точки зрения программы Мери, база данных не была целостной в течение выполняемой ею транзакции. В начале транзакции некоторая строка содержала одни данные, а позднее в той же самой транзакции она содержала другие данные, поскольку "внешние события" нарушили целостное восприятие базы данных этой программой. Такая несогласованность может привести к различным проблемам даже в том случае, если программа Мери не будет обновлять базу данных, опираясь на результаты первого запроса.

Например, если ее программа накапливает итоговые суммы или собирает статистические данные, то нет гарантии, что она отражает информацию правильно. В этом примере проблема заключается в том, что программа Мери имела доступ к результатам обновления, выполненного программой Джо для строки, которую программа Мери уже извлекала ранее. В стандарте SQL эта проблема обозначена как *P2*, или проблема *нестабильных результатов чтения*. Программа Мери не смогла дважды выполнить один и тот же запрос и при этом получить одинаковые результаты.

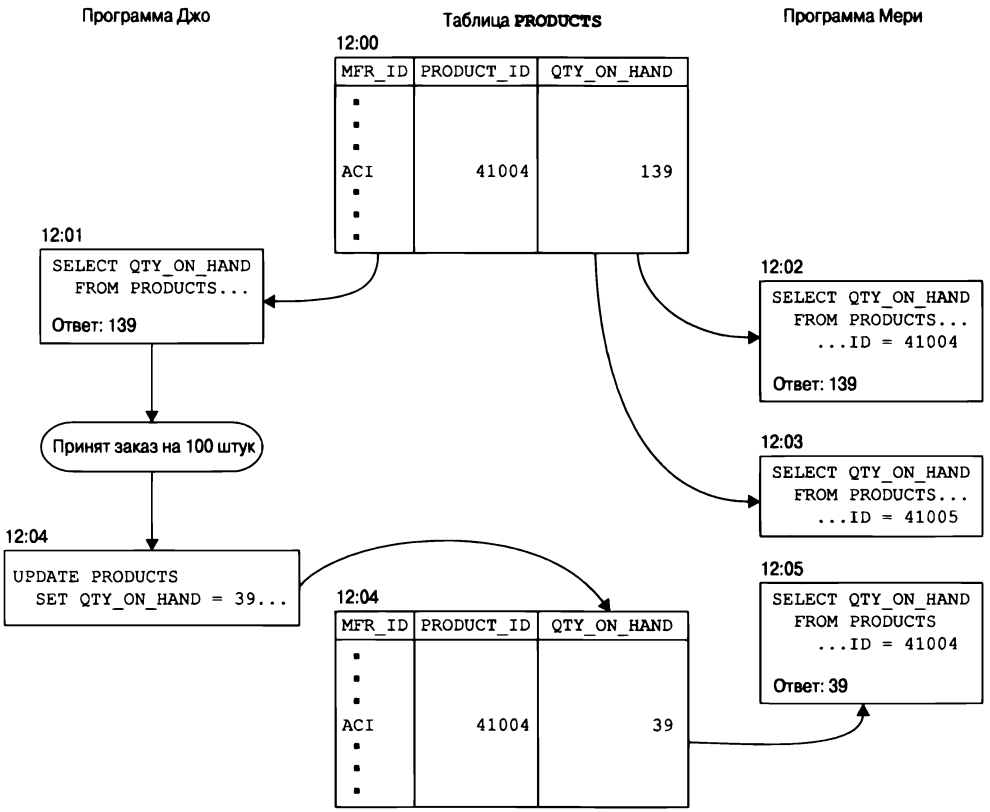


Рис. 12.8. Проблема несогласованных данных

Проблема строк-призраков

На рис. 12.9 еще раз изображена схема работы программы для обработки заказов. На этот раз менеджер по продажам запустил программу генерации отчетов, которая просматривает таблицу ORDERS и печатает список заказов от клиентов Билла Адамса, подсчитывая их итоговую сумму. Тем временем Биллу звонит клиент и размещает дополнительный заказ на $5000. Заказ добавляется в базу данных, и транзакция завершается. Вскоре после этого программа менеджера по продажам снова просматривает таблицу ORDERS, выполняя тот же запрос, что и прежде. На этот раз в таблице имеется дополнительный заказ, а итоговая сумма заказов на $5000 больше, чем в результате первого запроса.

Здесь, как и в предыдущем примере, проблема заключается в несогласованности данных. Состояние базы данных соответствует ситуации в реальной жизни, и целостность данных не нарушена, но один и тот же запрос, выполненный дважды в течение одной транзакции, возвращает два различных результата. В предыдущем примере запрос извлекал одну строку, и противоречивость данных была вызвана выполнением инструкции UPDATE. Выполнение инструкции DELETE могло бы вызвать ту же проблему.

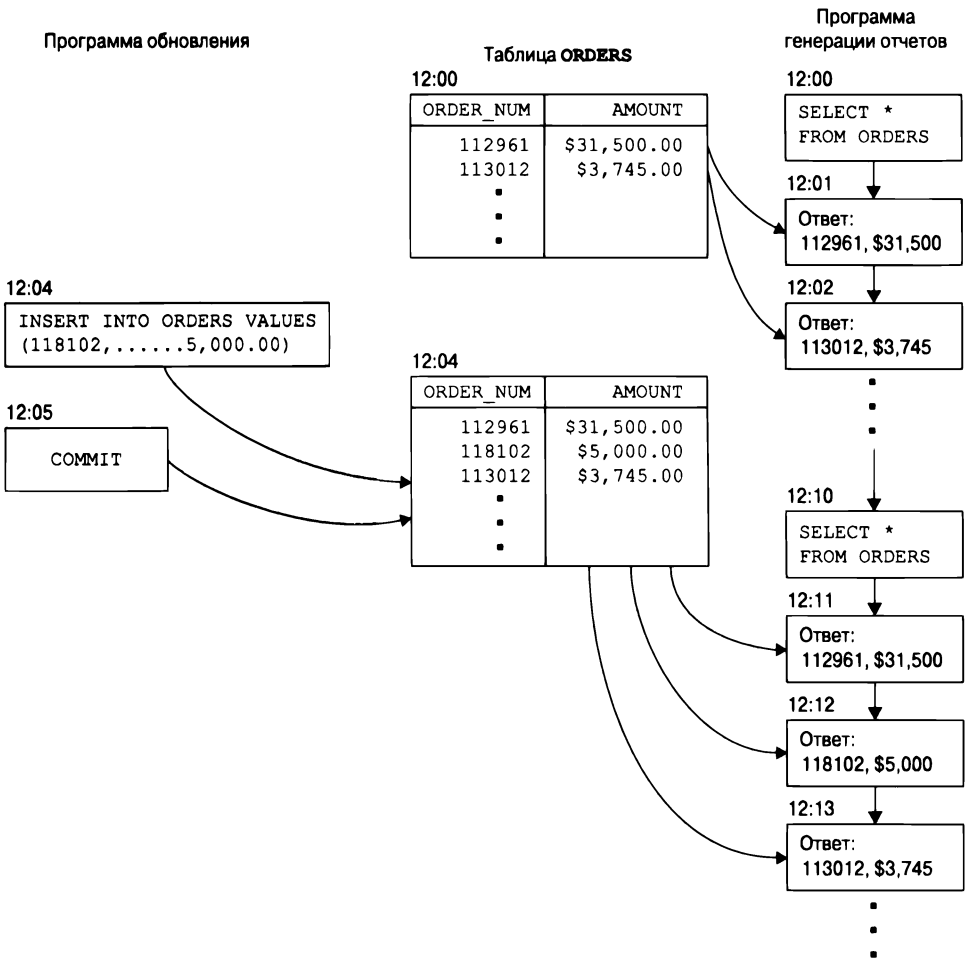


Рис. 12.9. Проблема строк-призраков

В примере, представленном на рис. 12.9, проблема возникла из-за принятия результатов выполнения инструкции INSERT. В первом запросе дополнительной строки не было, и после выполнения второго запроса сложилось такое впечатление, что она появилась из ниоткуда, как призрак. Проблема строк-призраков, как и проблема несогласованных данных, может привести к противоречивым и неправильным расчетам. В стандарте SQL эта проблема обозначена как *P3*, или просто проблема *призрака*.

Параллельные транзакции

Как видно из приведенных выше примеров, при обновлении базы данных в многопользовательском режиме существует возможность нарушения ее целостности. Чтобы исключить такую возможность, в SQL используется механизм транзакций. Помимо того что реляционная СУБД обязана выполнять транзакции по принципу "либо все, либо ничего", она имеет и другое обязательство по отношению к транзакциям.

В ходе выполнения транзакции пользователь видит полностью непротиворечивую базу данных. Он не должен видеть промежуточные результаты транзакций других пользователей, и даже принятие таких транзакций не должно отражаться на данных, которые пользователь видит в течение транзакции.

Таким образом, в реляционной базе данных транзакции играют ключевую роль как при восстановлении после сбоев, так и при параллельной работе нескольких пользователей. Приведенное выше правило можно сформулировать иначе, пользуясь концепцией параллельного выполнения транзакций.

Когда две транзакции, A *и* B, *выполняются параллельно, СУБД гарантирует, что результаты их выполнения будут точно такими же, как и в случае, если либо (а) вначале выполняется транзакция* A, *а затем транзакция* B; *либо (б) вначале выполняется транзакция* B, *а затем транзакция* A.

Данная концепция называется *сериализацией* транзакций. На практике это означает, что каждый пользователь может работать с базой данных так, как если бы не было других пользователей, работающих параллельно. На практике в больших промышленных базах данных возможно параллельное выполнение сотен транзакций. Концепция сериализации может быть непосредственно расширена для такой ситуации. Сериализация гарантирует, что если выполняется некоторое количество *N* параллельных транзакций, то СУБД должна гарантировать, что их результаты будут такими же, как если бы они выполнялись в некоторой очередности последовательно, одна за другой. Данная концепция не указывает, *какая* именно из последовательностей транзакций должна быть использована, она только утверждает, что конечные результаты должны соответствовать *некоторой* последовательности.

Однако тот факт, что SQL изолирует пользователя от действий других пользователей, не означает, что о них можно забыть. Совсем наоборот. Поскольку другим пользователям также требуется обновлять базу данных параллельно с вами, ваши транзакции должны быть как можно более простыми и короткими, чтобы максимизировать количество параллельно выполняемой работы.

Предположим, что вы запускаете программу, последовательно выполняющую три больших запроса на выборку. Так как программа не обновляет базу данных, может показаться, что нет необходимости думать о транзакциях и не требуется использовать инструкцию `COMMIT`. Однако на самом деле программа должна выполнять эту инструкцию после каждого запроса. Почему? Вспомним, что транзакция начинается автоматически вместе с первой SQL-инструкцией в программе. Без инструкции `COMMIT` транзакция будет продолжаться до окончания программы. Кроме того, SQL гарантирует, что данные, извлекаемые в течение транзакции, будут непротиворечивыми и не будут зависеть от транзакций других пользователей. Это означает, что если ваша программа извлекла из базы данных определенную строку, то ни один пользователь, кроме вас, не сможет изменить эту строку до окончания вашей транзакции. Так происходит из-за того, что позднее в этой же транзакции вы можете снова извлечь ту же строку, а СУБД должна гарантировать, что в этой строке будут содержаться те же данные, что и при первой выборке. Поэтому по мере того, как ваша программа будет последовательно выполнять три запроса, другие пользователи не смогут изменять все большее количество данных.

Мораль этого примера проста: при написании программ для промышленных реляционных баз данных всегда необходимо думать о транзакциях. Транзакции должны быть как можно короче. "Используйте инструкцию COMMIT как можно раньше и как можно чаще", — это хороший совет для всех, кто работает с программным SQL. Кроме того, помните, что имеется множество способов обработки транзакций различными СУБД, так что всегда внимательно просматривайте документацию на используемый вами продукт.

Практическая реализация строгой модели многопользовательских транзакций может привести к существенному замедлению работы с базами данных, насчитывающими сотни и тысячи пользователей. Кроме того, логика работы многих приложений вовсе не требует полной изоляции пользователей, которая подразумевается описанной моделью. Например, разработчик приложения может точно знать, что программа генерации отчетов никогда не запрашивает повторно одну и ту же строку таблицы в течение одной транзакции. В этом случае проблема несогласованных данных просто не может возникнуть. Или, допустим, разработчик может знать, что приложение обращается к некоторым таблицам базы данных только для чтения. Если бы иметь возможность сообщить подобного рода сведения СУБД, это позволило бы устранить многие проблемы, связанные с производительностью при обработке транзакций.

В исходном стандарте SQL1 вопросы производительности не рассматривались, поэтому в большинстве СУБД были реализованы собственные схемы повышения производительности транзакций. В стандарте SQL2 описана новая инструкция SET TRANSACTION, а в стандарте SQL:1999 введена новая инструкция START TRANSACTION (обе они показаны на рис. 12.2), назначение которых заключается в том, чтобы указать СУБД, какую степень изоляции следует обеспечить при обработке транзакций. Применять эти инструкции нет необходимости, если для вашего приложения вопросы обработки транзакций не играют роли или сами транзакции достаточно просты. Для полного понимания особенности работы указанных инструкций полезно разобраться с блокировкой и другими методами, применяемыми в коммерческих СУБД для реализации многопользовательских транзакций SQL. В оставшейся части главы подробно рассматриваются тонкости механизма блокировки и способы повышения производительности транзакций, реализованные в различных СУБД.

Блокировка*

Практически во всех ведущих СУБД для обработки параллельных транзакций применяется довольно сложный механизм блокировки. Однако принципы блокировки транзакций довольно просты. На рис. 12.10 изображена схема простой блокировки, устраняющей конфликты между двумя параллельными транзакциями.

Когда транзакция A обращается к базе данных, СУБД автоматически блокирует все части базы данных, в которых транзакция осуществляет выборку или изменение. Транзакция B выполняется параллельно, и СУБД также блокирует те части базы данных, к которым она обращается. Если транзакция B обращается к той части базы данных, которая заблокирована транзакцией A, то СУБД приостанавливает выполнение транзакции B, заставляя ее ждать до тех пор, пока данные не будут разблокированы. СУБД снимает блокировку, вызванную транзакцией A, только после того, как в этой транзакции встретится инструкция COMMIT или ROLLBACK.

Затем СУБД позволяет продолжить выполнение транзакции B. Теперь транзакция B блокирует эту же часть базы данных, защищая ее от других транзакций.

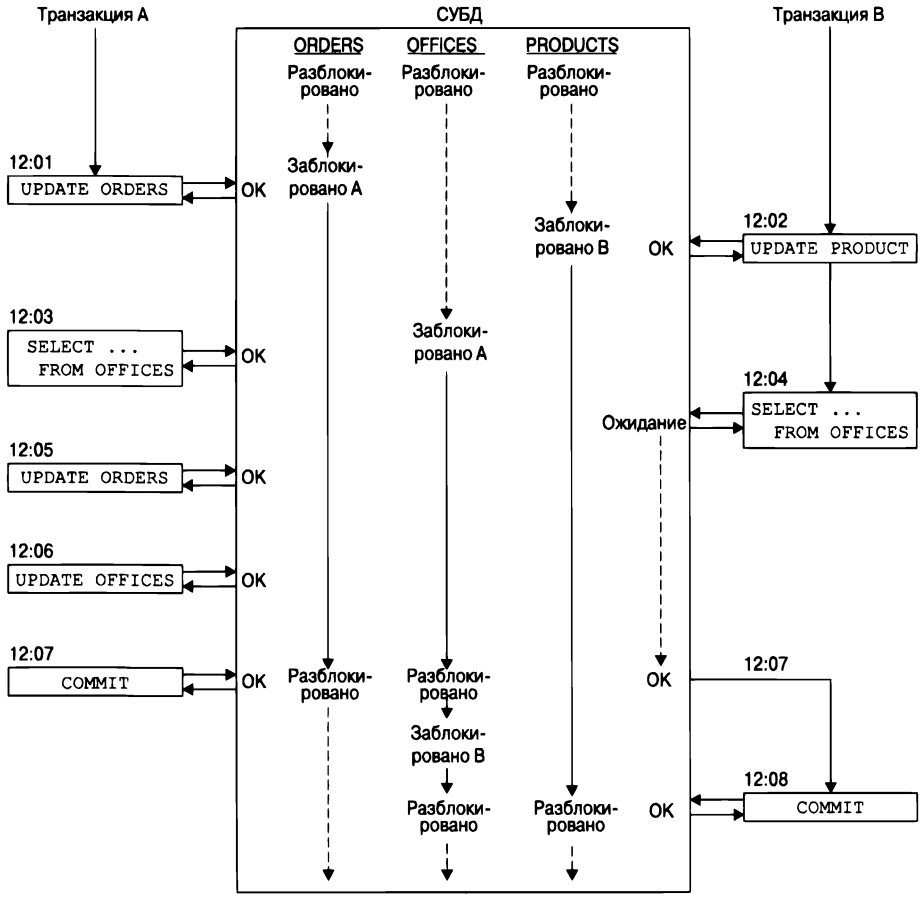


Рис. 12.10. Блокировка при параллельном выполнении двух транзакций

Как видно из рисунка, при блокировке транзакция временно получает монопольный доступ к некоторой части базы данных, причем другим транзакциям запрещается изменять заблокированные данные. Таким образом, блокировка решает все проблемы, возникающие при параллельном выполнении транзакций. Она не допускает разрушения базы данных в результате потери обновлений и использования промежуточных или несогласованных данных. Однако блокировка создает новую проблему: период времени, в течение которого транзакция ожидает освобождения части базы данных, заблокированной другой транзакцией, может быть достаточно большим.

Уровни блокировки

В базе данных блокировка может быть реализована на различных уровнях. Самой простой формой блокировки является блокировка всей базы данных. Этот вид блокировки легко реализовать, но при этом в каждый момент времени можно бу-

дет выполнять только одну транзакцию. Если транзакция затрачивает некоторое время "на размышления" (например, время обсуждения заказа с клиентом), то доступ всех остальных пользователей к базе данных будет при этом заблокирован, что приводит к слишком низкой производительности. Однако блокировка на уровне базы данных может оказаться подходящей для некоторых типов транзакций, например, тех, которые изменяют структуру базы данных, или для сложных запросов, которые должны последовательно сканировать много больших таблиц. В этих случаях может оказаться, что быстро и без накладных расходов выполнить единственную блокировку, после чего без помех и задержек выполнить операцию над базой данных, а затем разблокировать базу данных, будет быстрее, чем индивидуально блокировать десятки таблиц.

Более прогрессивной формой блокировки является блокировка *на уровне таблиц*. В этом случае СУБД блокирует только те таблицы, к которым обращается транзакция. Остальные транзакции в это время могут обращаться к другим таблицам. Этот вид блокировки предпочтительнее, чем блокировка базы данных, поскольку обеспечивает возможность параллельной работы транзакций. Однако в таких приложениях, как программы для ввода заказов, в которых несколько пользователей совместно используют одни и те же таблицы, данный вид блокировки все еще приводит к слишком низкой производительности.

Во многих СУБД реализована блокировка *на уровне страниц*. В этом случае СУБД блокирует при обращении к ним отдельные блоки данных (страницы) на диске. Остальным транзакциям запрещается доступ к заблокированным страницам, но они могут обращаться к другим страницам данных (и блокировать их для себя). Обычно используются страницы размером 2, 4 и 16 Кбайт. Поскольку большая таблица состоит из сотен или даже тысяч страниц, две транзакции, обращающиеся к двум различным строкам таблицы, как правило, обращаются к различным страницам; в результате обе транзакции выполняются параллельно.

За последние несколько лет в большинстве ведущих коммерческих СУБД была реализована блокировка *на уровне строк*. Она допускает параллельное выполнение транзакций, которые обращаются к двум различным строкам таблицы, даже если эти строки содержатся на одной странице. Хотя такая возможность кажется несущественной, в случае таблиц, содержащих небольшое число строк (например, таблица `OFFICES` из учебной базы данных), она может играть важную роль.

Блокировка на уровне строк обеспечивает большую степень параллельного выполнения транзакций. К сожалению, осуществить блокировку частей переменной длины (другими словами, строк) гораздо сложнее, чем блокировку страниц фиксированного размера, так что повышение степени параллельности работы обеспечивается за счет усложнения логики блокирования и роста накладных расходов. В действительности для некоторых транзакций или приложений накладные расходы на блокировку на уровне строк могут перекрывать выигрыш от повышения степени параллельности работы с базой данных.

Некоторые СУБД решают эту задачу путем автоматического переноса блокировок отдельных строк в случае, когда их количество достигает некоторого предела, на уровень страниц или таблиц. Меньший уровень блокировок не всегда дает выигрыш; какая схема является наилучшей, зависит от конкретных транзакций и содержащихся в них операций SQL.

Теоретически можно пойти еще дальше и осуществлять блокировку отдельных ячеек таблиц. В теории такая блокировка должна обеспечить еще большую степень параллелизма, чем блокировка строк, поскольку она разрешает параллельный доступ двух различных транзакций к одной и той же строке, если они обращаются к различным наборам столбцов. Однако затраты, связанные с блокировкой ячеек, значительно превышают потенциальную отдачу. Ни в одной коммерческой реляционной СУБД не используется блокировка такого типа. В действительности блокировка в базах данных является предметом изучения, и схемы блокировки, реально используемые в коммерческих продуктах, существенно сложнее описанных здесь. Наиболее простые из этих коммерческих схем будут рассмотрены немного позже в этой главе.

Блокировка с обеспечением совместного доступа и исключающая блокировка

В большинстве коммерческих СУБД для повышения степени параллельности доступа нескольких пользователей к одной базе данных используются блокировки различных типов. Наиболее широко распространены два из них.

- **Блокировка с обеспечением совместного доступа**, или блокировка без монополизации (shared lock). Когда транзакция извлекает информацию из базы данных, СУБД применяет блокировку без монополизации. При этом другие транзакции, выполняемые параллельно, могут извлекать те же данные.
- **Монопольная**, или исключающая блокировка (exclusive lock). Когда транзакция обновляет информацию в базе данных, СУБД применяет исключающую блокировку. Если транзакция монопольно заблокировала какие-либо данные, другие транзакции не могут обращаться к ним ни для выборки, ни для записи.

В таблице на рис. 12.11 изображены допустимые комбинации блокировок для двух параллельно выполняемых транзакций. Следует отметить, что транзакция может применять для данных монопольную блокировку только в том случае, если ни одна другая транзакция не блокирует эти данные. Если транзакция пытается осуществить блокировку, не разрешенную правилами, представленными на рис. 12.11, ее выполнение приостанавливается до тех пор, пока другие транзакции не разблокируют необходимые ей данные.

		Транзакция В		
		Без блокировки	Блокировка без монополизации	Монопольная блокировка
Транзакция А	Без блокировки	Да	Да	Да
	Блокировка без монополизации	Да	Да	Нет
	Монопольная блокировка	Да	Нет	Нет

Рис. 12.11. Правила применения блокировок

На рис. 12.12 изображены те же транзакции, что и на рис. 12.10, но в них на этот раз используются описанные выше монопольная и немонопольная блокировки. Если сравнить два этих рисунка, то можно увидеть, что новый механизм блокировки повышает степень параллельности доступа к базе данных. В таких сложных СУБД, как DB2, имеется еще большее количество типов блокировок, и на различных уровнях базы данных применяются различные механизмы блокировки. Несмотря на повышенную сложность, цель блокировки при этом остается той же: предотвращать конфликты между транзакциями, обеспечивая при этом максимальную параллельность доступа к базе данных и минимальные затраты на реализацию механизма блокировки.

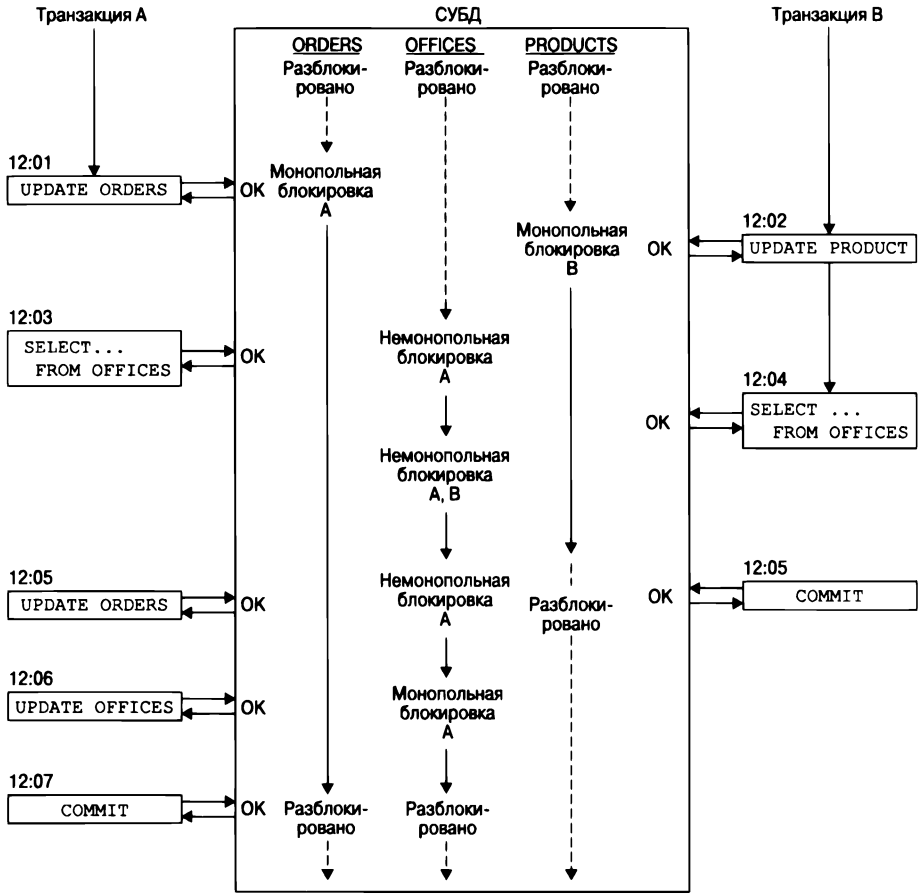


Рис. 12.12. Использование монопольной и немонопольной блокировок

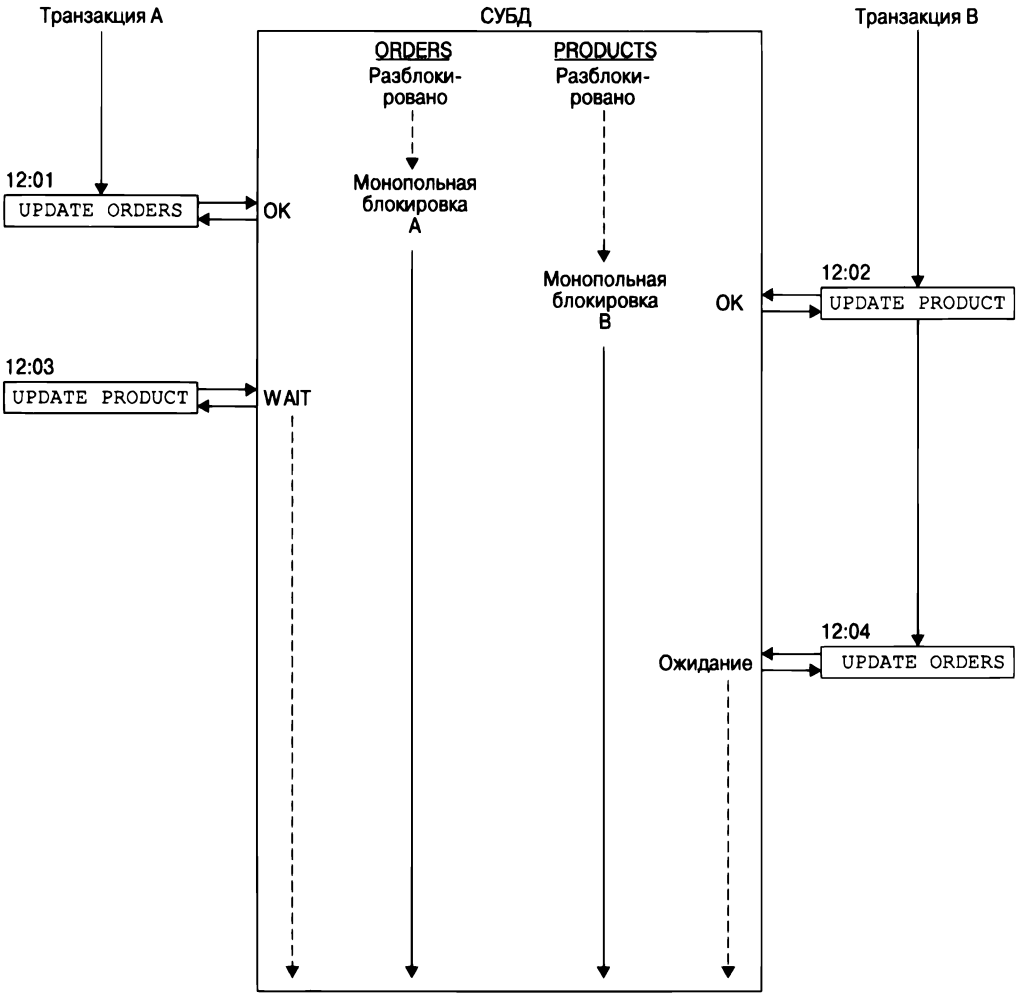


Рис. 12.13. Взаимоблокировка транзакций

Как правило, для устранения взаимоблокировок СУБД периодически (например, каждые пять секунд) проверяет блокировки, удерживаемые различными транзакциями. Если СУБД обнаруживает взаимоблокировку, то произвольно выбирает одну транзакцию в качестве "проигравшей" и отменяет ее. Это освобождает блокировки, удерживаемые проигравшей транзакцией, позволяя "победителю" продолжить работу. Проигравшей программе посылается код ошибки, показывающий, что она проиграла в ситуации взаимоблокировки и ее текущая транзакция была отменена.

Такая схема устранения взаимоблокировок означает, что любая инструкция SQL потенциально может вернуть код ошибки "проигрыша во взаимоблокировке", хотя сама по себе инструкция будет правильной. Транзакция, пытающаяся выполнить эту инструкцию, отменяется не по вине последней, а из-за параллельной работы с базой данных многих пользователей. Такая схема может показаться

несправедливой, но на практике она лучше двух возможных альтернатив: "зависания" или нарушения целостности данных. Если ошибка "проигрыша во взаимоблокировке" происходит в интерактивном режиме, пользователь может просто повторно ввести инструкции (одну или несколько). В программном SQL прикладная программа сама должна обработать код подобной ошибки. Обычно в таких случаях программа либо выдает предупреждающее сообщение пользователю, либо пытается выполнить транзакцию повторно.

Вероятность возникновения взаимоблокировок можно резко уменьшить, если тщательно планировать обновления базы данных. Все программы, которые в своих транзакциях обновляют несколько таблиц, по возможности всегда должны обновлять таблицы в одном и том же порядке. Это приводит к плавному перемещению блокировок по таблицам и сводит к минимуму возможность возникновения взаимоблокировок. Кроме того, для дальнейшего уменьшения количества возникающих взаимоблокировок можно использовать более сложные средства блокировки, описанные в следующем разделе.

Усовершенствованные методы блокировки*

Во многих коммерческих СУБД предлагаются усовершенствованные средства блокировки, которые гораздо лучше стандартных. К таким средствам относятся следующие.

- **Явная блокировка**. Программа может явно заблокировать целую таблицу или другую часть базы данных, если планирует многократно обращаться к ней.
- **Уровни изоляции**. Можно проинформировать СУБД о том, что некоторая программа не будет повторно извлекать данные во время транзакции, позволяя тем самым СУБД снять блокировку еще до окончания транзакции.
- **Параметры блокировки**. Администратор базы данных может вручную установить размер блокируемого участка базы данных и другие параметры блокировки.

По своей природе все эти средства являются не стандартными, а специфичными для каждой конкретной СУБД. Однако некоторые из них, особенно используемые в DB2, были реализованы в нескольких коммерческих СУБД и приобрели статус общепринятых (чтобы не сказать стандартных) средств. Например, уровни изоляции, появившиеся в DB2, были официально закреплены в стандарте SQL2.

Явная блокировка*

Если программа многократно обращается к таблице, затраты, связанные с блокировкой большого количества маленьких участков таблицы, могут оказаться значительными. Например, программа пакетного обновления, которая обращается к каждой строке таблицы, в процессе работы будет блокировать таблицу по частям. Для более быстрого выполнения транзакции такого типа программе следует явно заблокировать всю таблицу, выполнить все необходимые обновления, а затем разблокировать ее. Блокировка всей таблицы имеет три преимущества:

- она устраняет накладные расходы, связанные с блокировкой строк (страниц);
- она исключает возможность того, что другая транзакция заблокирует часть таблицы, вынуждая программу пакетного обновления находиться в состоянии ожидания;
- она исключает возможность того, что другая транзакция заблокирует часть таблицы и вызовет взаимоблокировку с программой пакетного обновления, вынуждая ее начать выполнение повторно.

Конечно, у блокировки таблицы есть тот недостаток, что все остальные транзакции, которые попытаются обратиться к таблице, должны ждать окончания ее обновления. Но поскольку транзакция, выполняющая пакетное обновление, обычно работает гораздо быстрее остальных, общая производительность СУБД при явной блокировке таблиц может увеличиться.

В базах данных компании IBM для явной блокировки целой таблицы применяется инструкция LOCK TABLE, синтаксическая диаграмма которой изображена на рис. 12.14. С помощью этой инструкции можно установить один из двух режимов блокировки.

- **EXCLUSIVE**. В этом режиме осуществляется монопольная блокировка всей таблицы. Ни одна другая транзакция при этом не имеет доступа ни к одной части таблицы. Данный режим следует использовать, если транзакция выполняет пакетное обновление всей таблицы.
- **SHARE**. В режиме SHARE осуществляется немонопольная блокировка таблицы. Другие транзакции могут извлекать данные из таблицы (т.е. они тоже могут осуществлять немонопольную блокировку), но не могут обновлять ее. Для того чтобы обновить часть таблицы, транзакция, выполнившая инструкцию LOCK TABLE, должна монопольно заблокировать требуемую часть таблицы. Этот режим блокировки следует использовать также в том случае, если требуется получить "мгновенный снимок" таблицы в какой-то определенный момент времени.

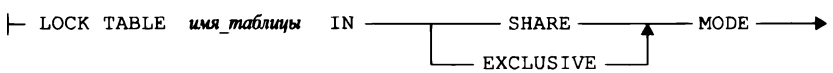


Рис. 12.14. Синтаксическая диаграмма инструкции LOCK TABLE

Oracle поддерживает также инструкцию LOCK TABLE в стиле DB2, а MySQL обеспечивает аналогичные возможности с помощью инструкций LOCK TABLES и UNLOCK TABLES. В некоторых других СУБД, таких как SQL Server, явная блокировка не применяется вообще, а вместо этого усовершенствована методика применения неявной блокировки.

Уровни изоляции*

В соответствии со строгим определением транзакции, в SQL ни одна из других транзакций, выполняемых параллельно, не может влиять на данные, к которым ваша транзакция обращается во время своего выполнения. Если в течение тран-

закции ваша программа выполняет запрос на выборку информации из базы данных, после этого производит какие-нибудь другие действия, а затем выполняет тот же запрос на выборку во второй раз, механизм транзакций SQL гарантирует, что данные, возвращенные обоими запросами, будут идентичными (если только *ваша* транзакция не изменила эти данные). Возможность повторной выборки строки во время транзакции означает наиболее высокую степень изоляции вашей программы от других программ и пользователей. Степень изоляции одной транзакции от других транзакций называется *уровнем изоляции*.

Абсолютная изоляция транзакции от других транзакций требует больших затрат на блокировку. По мере того как программа извлекает строки из таблицы результатов запроса, СУБД должна блокировать эти строки в базе данных (в режиме немонопольной блокировки), чтобы предотвратить их изменение параллельными транзакциями. Установленные таким образом блокировки необходимо удерживать до конца транзакции на случай, если программа повторит свой запрос. Во многих случаях СУБД может значительно сократить затраты на блокировку, если она будет заранее знать, каким образом программа будет обращаться к базе данных во время транзакции.

Для этого в основных СУБД для мэйнфреймов компании IBM добавлена поддержка задаваемого пользователем уровня изоляции, который позволяет найти некоторый компромисс между степенью изоляции и эффективностью работы. В спецификации стандарта SQL это понятие формализовано и расширено и включает четыре уровня, показанные в табл. 12.1. Уровни изоляции напрямую связаны с основными проблемами многопользовательского обновления данных, которые были рассмотрены ранее в этой главе. По мере уменьшения уровня изоляции (в таблице в направлении сверху вниз) сокращается число проблем, от которых СУБД защищает пользователя.

Таблица 12.1. Уровни изоляции и многопользовательские обновления

Уровень изоляции	Пропавшие обновления	Промежуточные данные	Несогласованные данные	Строки-призраки
SERIALIZABLE	Предотвращается	Предотвращается	Предотвращается	Предотвращается
REPEATABLE READ	Предотвращается	Предотвращается	Предотвращается	Возможно
READ COMMITTED	Предотвращается	Предотвращается	Возможно	Возможно
READ UNCOMMITTED	Предотвращается	Возможно	Возможно	Возможно

Наиболее высоким уровнем изоляции является уровень SERIALIZABLE. На этом уровне СУБД гарантирует, что результаты параллельного выполнения транзакций будут точно такими же, как если бы эти транзакции выполнялись последовательно. Этот уровень изоляции устанавливается по умолчанию, поскольку он соответствует главному принципу, в соответствии с которым должна работать база данных. Если программе во время транзакции требуется дважды выполнить один и тот же запрос к нескольким строкам и при этом необходимо гарантировать, что результаты будут идентичными независимо от параллельно выполняемых в базе данных транзакций, то следует установить уровень изоляции SERIALIZABLE.

Уровень изоляции REPEATABLE READ является вторым по степени изоляции после уровня SERIALIZABLE. На этом уровне транзакция не имеет доступа к промежуточным или окончательным результатам других транзакций, выполняющих обновления данных, поэтому такие проблемы, как пропавшее обновление, промежуточные или несогласованные данные, возникнуть не могут. Однако во время своей транзакции вы можете увидеть строку, добавленную в базу данных другой транзакцией. Поэтому один и тот же запрос к нескольким строкам, выполненный дважды в течение одной транзакции, может возвратить различные таблицы результатов (проблема строк-призраков). Если программе не требуется повторять многострочный запрос в течение одной транзакции, то для повышения производительности СУБД можно установить уровень изоляции REPEATABLE READ, не рискуя при этом нарушить целостность данных. Этот уровень изоляции поддерживается в СУБД, используемых на мэйнфреймах компании IBM.

Уровень READ COMMITTED является третьим по степени изоляции. В этом режиме транзакция не имеет доступа к промежуточным результатам других транзакций, поэтому проблемы пропавшего обновления и промежуточных данных возникнуть не могут. Однако окончательные результаты других, параллельно выполняемых, транзакций могут быть доступны вашей транзакции. Программа могла бы, например, дважды в течение транзакции выполнить однострочную инструкцию SELECT и обнаружить, что некоторая строка была изменена другим пользователем. Если программе не требуется повторно извлекать одну и ту же строку в течение транзакции и она не накапливает итоги и не выполняет других вычислений, для которых необходимы непротиворечивые данные, можно безо всяких опасений применять уровень изоляции READ COMMITTED. Обратите внимание на следующее: если программа попытается обновить строку, которую уже обновил другой пользователь, то транзакция будет автоматически отменена во избежание возникновения проблемы пропавшего обновления.

Уровень READ UNCOMMITTED является наиболее низким уровнем изоляции в стандарте SQL. В этом режиме на выполнение транзакции могут повлиять как окончательные, так и промежуточные результаты других транзакций, поэтому могут возникнуть проблемы как строк-призраков, так и промежуточных и несогласованных данных. Однако СУБД по-прежнему предотвращает проблему пропавшего обновления. В общем случае уровень READ UNCOMMITTED подходит только для некоторых приложений со специальными запросами, где пользователь может позволить, чтобы результаты запроса содержали "грязные" данные. Если для вас важно, чтобы результаты запроса представляли только ту информацию, которая является окончательной на текущий момент для базы данных, то не следует использовать этот режим в своих программах.

Вспомните, что в стандарте SQL описаны инструкции SET TRANSACTION и START TRANSACTION, представленные на рис. 12.2, которые могут использоваться для установки уровня изоляции транзакций. Эти инструкции позволяют также указать тип транзакции — READ ONLY (данная транзакция выполняет только запросы к базе данных) или READ WRITE (может как запрашивать, так и обновлять информацию в базе данных). СУБД может использовать эту информацию (наряду со сведениями об уровне изоляции) для оптимизации работы базы данных. По

умолчанию устанавливается уровень изоляции SERIALIZABLE. Если задан уровень READ UNCOMMITTED, то предполагается, что транзакция имеет тип READ ONLY, поэтому указывать тип READ WRITE нельзя. В противном случае по умолчанию считается, что транзакция имеет атрибут READ WRITE. Эти установки по умолчанию обеспечивают максимальную безопасность транзакций, пусть даже ценой некоторой потери производительности. Это оберегает неопытных программистов от нечаянного столкновения с рассмотренными ранее проблемами обработки данных в многопользовательской среде.

Заметим, что инструкции SET TRANSACTION и START TRANSACTION, определенные в стандарте SQL, являются исполняемыми инструкциями SQL. Возможно, а иногда даже желательно, чтобы одна транзакция в программе выполнялась в одном режиме, а следующая — в другом. В то же время нельзя изменить уровень изоляции и атрибут чтения/записи по ходу самой транзакции. В стандарте требуется, чтобы инструкция SET TRANSACTION стояла первой в транзакции. Это означает, что она должна выполняться сразу же вслед за инструкциями COMMIT и ROLLBACK либо быть первой в программе, пока другие инструкции еще не успели повлиять на содержимое или структуру базы данных.

Как уже упоминалось ранее, во многих коммерческих СУБД были реализованы собственные схемы блокировки и повышения производительности транзакций, затрагивающие ядро самой СУБД и определяющие логику ее работы, еще задолго до появления этих возможностей в стандарте SQL. Поэтому неудивительно, что поддержка стандарта SQL в этой части внедрялась медленнее, чем в других частях. Например, в СУБД для мэйнфреймов компании IBM (DB2 и SQL/DS) исторически существовало лишь два уровня изоляции — REPEATABLE READ или READ COMMITTED (CURSOR STABILITY в терминологии компании IBM). В реализациях IBM выбор осуществляется в процессе разработки программы, на шаге BIND, описанном в главе 17, "Встроенный SQL". Хотя, строго говоря, режимы не являются частью языка SQL, выбор режима оказывает сильное влияние на то, каким образом выполняется программа и как используются извлеченные ею данные.

В СУБД Ingres предоставляются функциональные возможности, аналогичные уровням изоляции в СУБД компании IBM, но реализованные в другом виде. С помощью инструкции SET LOCKMODE прикладная программа говорит Ingres о том, какой тип блокировки следует использовать при обработке запроса к базе данных. Можно установить следующие режимы.

- **Нет блокировки** — аналогичен уровню CURSOR STABILITY от IBM.
- **Немонопольная блокировка** — аналогичен уровню REPEATABLE READ от IBM.
- **Монопольная блокировка** — обеспечивает монопольный доступ к таблице в течение всего запроса (аналогично инструкции LOCK TABLE).

По умолчанию в СУБД Ingres устанавливается режим немонопольной блокировки, а в СУБД компании IBM — уровень REPEATABLE READ (аналогичный ему). В отличие от СУБД компании IBM, где уровни изоляции должны быть выбраны во время компиляции программы, в СУБД Ingres режимы блокировки устанавливаются с помощью выполняемой инструкции SQL; их можно выбирать в процессе выполнения программы и даже изменять от запроса к запросу.

Параметры блокировки*

В таких развитых СУБД, как DB2, SQL/DS, Oracle, Sybase ASF или SQL Server, применяются механизмы блокировки, которые гораздо сложнее описанных здесь. Администратор базы данных может повысить производительность этих СУБД, устанавливая параметры блокировки вручную. С другой стороны, сегодня производители основных СУБД обычно препятствуют ручному вмешательству в механизм блокировок, поскольку он стал настолько сложен, что попытки улучшить его, скорее, приведут к обратному эффекту. В старых же системах ручная настройка может дать положительный эффект. К типичным настраиваемым параметрам относятся следующие.

- **Размер заблокированного участка**. В некоторых СУБД можно выбрать, что будет блокироваться — таблица, страница, строка или другие участки данных. Для различных прикладных программ подходящими будут различные размеры заблокированных участков.
- **Число блокировок**. Обычно СУБД позволяет каждой транзакции иметь ограниченное число блокировок. Предел устанавливает администратор базы данных, увеличивая его для сложных транзакций и уменьшая для простых.
- **Наращивание блокировок**. Часто СУБД автоматически наращивает блокировки, заменяя множество маленьких блокировок одной большой (например, заменяя несколько страничных блокировок блокировкой таблицы). Администратор базы данных также может управлять процессом наращивания.
- **Тайм-аут блокировки**. Если транзакция заблокирована другой транзакцией, она может довольно долго ожидать, пока вторая транзакция снимет свои блокировки. Поэтому в некоторых СУБД реализованы *тайм-ауты* блокировки. По истечении интервала тайм-аута SQL-инструкция завершается неуспешно и возвращает соответствующий код ошибки SQL. Значение интервала тайм-аута обычно задается администратором базы данных.

Управление версиями*

В реляционных СУБД для поддержки параллельного многопользовательского выполнения транзакций наиболее широко применяются методы блокировки, описанные в предыдущих разделах. Иногда блокировку называют пессимистическим подходом к параллельности, поскольку, блокируя части базы данных, СУБД неявно предполагает, что параллельные транзакции, вероятно, будут влиять друг на друга. В последние годы возрастает популярность иного подхода — *управления версиями* (versioning), который уже реализован в некоторых СУБД. Иногда его называют оптимистическим подходом к параллельности, потому что в этом случае СУБД неявно предполагает, что параллельные транзакции не влияют друг на друга.

В случае блокировочной (пессимистической) архитектуры СУБД внутренне поддерживает одну и только одну копию данных для каждой строки базы данных. При доступе к базе данных нескольких пользователей схема блокировки распределяет доступ к строке между пользователями (вернее, между их параллельно выполняемыми транзакциями). При использовании же архитектуры управления версиями (оптимистической), СУБД создает две или большее количество копий данных строки при попытках пользователя ее обновить. Одна копия содержит старые данные (до обновления), вторая — новые (после обновления). СУБД внутренне отслеживает, какую версию строки должна видеть та или иная транзакция, в зависимости от ее уровня изоляции.

Управление версиями в действии*

На рис. 12.15 показана простая архитектура управления версиями в действии. Транзакция A начинает работу, считывая строку таблицы PRODUCTS, и находит 139 доступных единиц изделия ACI-41004. Затем начинается транзакция B, которая обновляет эту же строку, уменьшая количество доступных единиц до 39. В ответ СУБД внутренне создает новую копию строки. С этого момента, если транзакция B перечитывает содержимое строки, то оно берется из этой новой копии, которая отражает обновленное транзакцией B значение (39 единиц). Затем идет транзакция C, которая вновь пытается прочесть ту же самую строку. Поскольку обновление транзакцией B пока что еще не принято, СУБД передает транзакции C данные из старой копии строки, в которой указано 139 доступных единиц. То же самое происходит и несколькими секундами позже с транзакцией D; она также видит 139 доступных единиц. Теперь транзакция B выполняет инструкцию COMMIT, делая обновление строки постоянным. Некоторое время спустя транзакция E пытается читать строку. Поскольку обновление, выполненное транзакцией B, уже принято, СУБД передаст транзакции E данные из новой копии, в которой указаны 39 доступных единиц. Наконец, транзакции C, D и E завершают работу с базой данных операцией COMMIT.

Действия, показанные на рис. 12.15, отвечают требованиям сериализации операций СУБД. Последовательность транзакций A-C-D-B-E даст тот же результат, что и показанные на рисунке (кстати, последовательность A-D-C-B-E также приведет к этому результату). Более того, реализация системы управления версиями обеспечивает выполнение операций, не заставляя ни одну из транзакций ожидать — в отличие от типичной реализации блокировки, показанной на рис. 12.16.

На рис. 12.16 первой вновь выполняется транзакция A, находя 139 доступных единиц изделия ACI-41004. Внутренне СУБД немонопольно блокирует строку. Затем транзакция B пытается обновить строку, снижая указанную величину до 39. Если транзакция A работает на строгом уровне изоляции (таком, как REPEATABLE READ), транзакция B в этот момент приостанавливается, будучи не в силах установить монопольную блокировку. Если же транзакция A работает на менее строгом уровне изоляции, СУБД позволяет транзакции B продолжать работу, передавая ей в монопольное владение строку и выполняя обновление данных. Внутренняя строка в базе данных (вспомните, что в данной архитектуре блоки-

ровки имеется только одна копия строки) теперь содержит величину 39 для количества доступных единиц. Когда начинается транзакция C, она должна ожидать, пока транзакция B снимет блокировку — если только транзакция C не работает на очень низком (READ UNCOMMITTED) уровне изоляции. То же самое верно и в отношении транзакции D. Транзакции C и D могут продолжать работу только после того, как будут приняты изменения, внесенные транзакцией B.

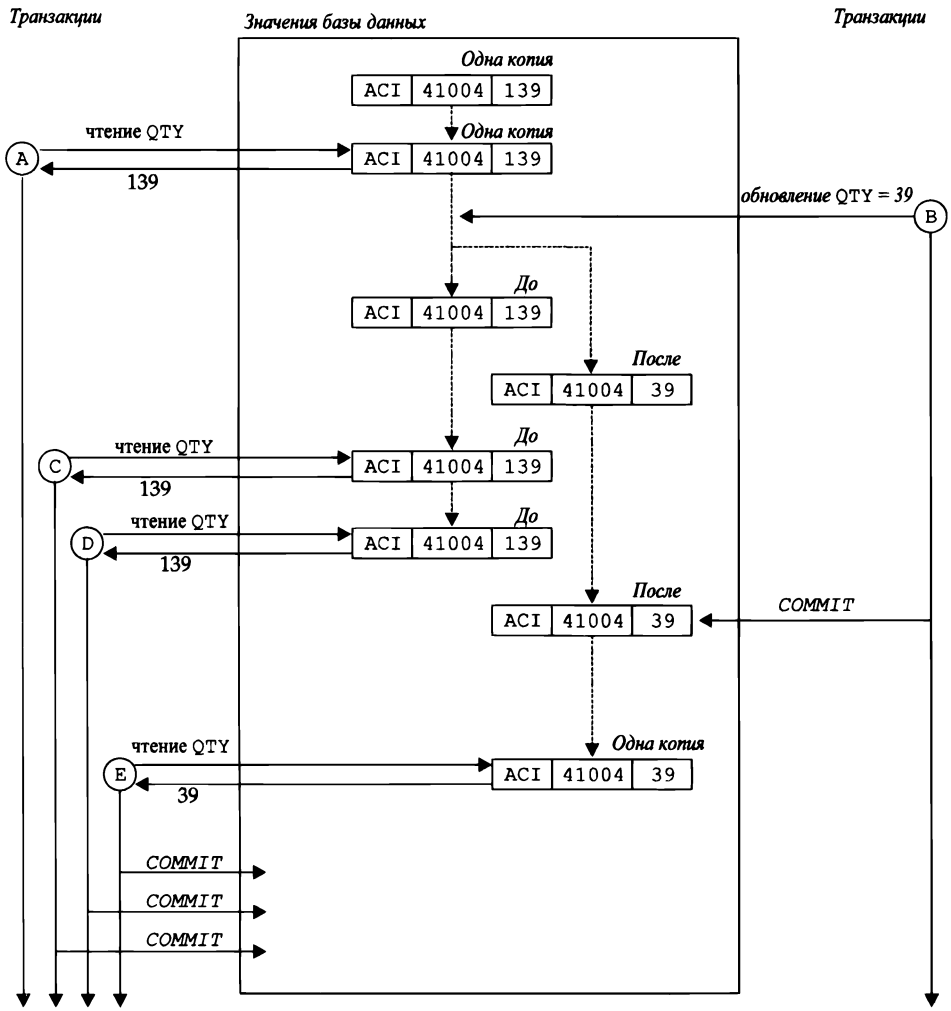


Рис. 12.15. Параллельные транзакции в архитектуре управления версиями

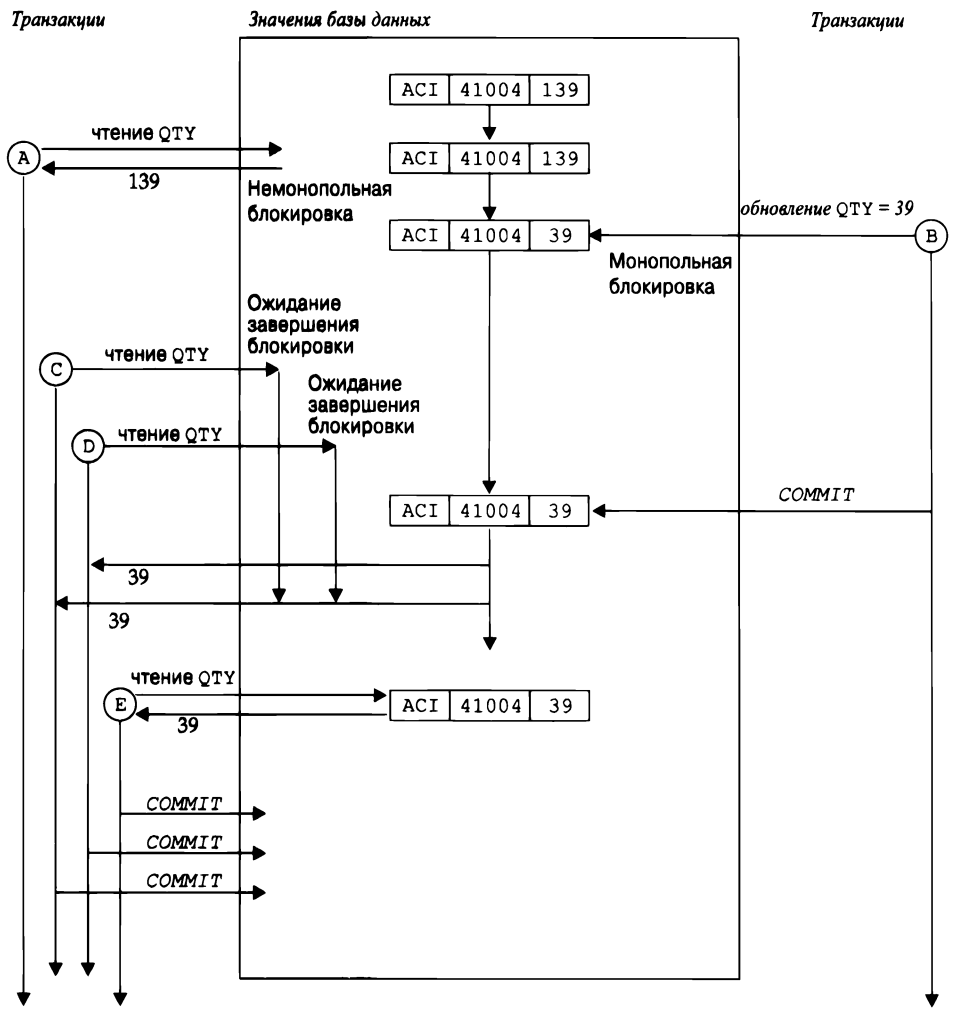


Рис. 12.16. Параллельные транзакции в архитектуре с блокировками

Сравнивая операции на рис. 12.15 и 12.16, стоит обратить внимание на два отличия. Первое, более фундаментальное, заключается в том, что подход с управлением версиями на рис. 12.15 обеспечивает большую степень параллельности транзакций. Подход с применением блокировок на рис. 12.16 в большинстве случаев приводит к тому, что ряд транзакций вынужден находиться в состоянии ожидания, пока другие транзакции не завершатся и не снимут свои блокировки. Второе, более тонкое, отличие состоит в том, что на двух рисунках отличается порядок выполнения транзакций. На рис. 12.15 конечный результат получается из последовательности транзакций A-C-D-B-E. На рис. 12.16 результат создает последовательность A-B-C-D-E. Заметим, что ни одна, ни другая последовательности не являются корректными или некорректными; принцип сериализации гласит лишь, что конечный результат соответствует *некоторой* последовательности транзакций.

Пример на рис. 12.15 включает только одну обновляющую транзакцию, так что требуются только две копии обновленной строки (до и после обновления). Архитектура с управлением версиями легко расширяется для поддержки большего количества параллельных обновлений. Для каждой попытки обновления строки СУБД может создавать другую новую строку, отражающую обновление. При таком подходе становится более сложной задача отслеживания того, какая версия строки должна быть видна той или иной транзакции. На практике такое решение о видимости строки транзакции зависит не только от последовательности операций над базой данных, но и от уровней изоляции, запрошенных каждой из транзакций.

Управление версиями не устраняет полностью возможность взаимоблокировок в базе данных. Две транзакции на рис. 12.13 с их чередующимися попытками обновить две разные таблицы в разном порядке вызывают проблемы даже при применении схемы управления версиями. Однако при загрузке базы данных большим количеством операций чтения и обновления схема управления версиями может существенно снизить степень заблокированности и уменьшить тайм-ауты блокировок или взаимоблокировок.

Преимущества и недостатки управления версиями*

Преимущества архитектуры управления версиями заключаются в том, что, при соответствующих условиях, она может значительно увеличить количество параллельно выполняемых транзакций. Параллельное выполнение транзакций становится все более и более важным в больших СУБД, в особенности поддерживающих веб-сайты, которые одновременно могут посещать тысячи или десятки тысяч пользователей. Система с управлением версиями становится более полезна также при увеличении количества процессоров типичного сервера СУБД. Серверы с 16 и более процессорами становятся достаточно распространенным явлением, а большие серверы СУБД могут поддерживать 64 и более процессоров в симметричной многопроцессорной конфигурации (SMP). Такие серверы в состоянии параллельно выполнять много приложений, обращающихся к базе данных, распределяя нагрузку среди процессоров.

Недостатком архитектуры управления версиями являются внутренние накладные расходы СУБД. Одним очевидным расходом является повышение требований к оперативной и дисковой памяти для хранения большого количества копий обновляемых строк. На практике более существенные накладные расходы получаются из-за повышенной активности системы управления памятью, которая должна постоянно выделять память для каждой временной копии строки (потенциально — тысячи раз в секунду), а затем освобождать ее для повторного использования, когда старые копии строки становятся ненужными. Еще одним дополнительным расходом является отслеживание, какие копии строк должны быть видимы тем или иным транзакциям.

Неявно архитектура управления версиями основана на предположении, что большинство параллельных транзакций не влияет друг на друга. Если это действительно так (т.е. если параллельно выполняемые транзакции в основном считывают и обновляют разные строки или если основная часть транзакций считывает данные, а не обновляет их), то накладные расходы, добавляемые схемой управления версия-

ми, будут невелики по сравнению с выигрышем. Если же данное предположение неверно (т.е. если параллельно выполняемые транзакции считывают и обновляют в основном одни и те же строки), то накладные расходы становятся велики и сводят на нет получаемые путем повышения степени параллельности преимущества.

Резюме

В настоящей главе был описан механизм транзакций, поддерживаемый в SQL.

- В реляционной СУБД транзакция представляет собой логическую единицу работы. Транзакция состоит из последовательности инструкций SQL, которые СУБД выполняет как одно целое.
- Инструкции `SET TRANSACTION` и `START TRANSACTION` могут использоваться для установки уровня изоляции и уровня доступа транзакций.
- Инструкция `SAVEPOINT` создает промежуточную точку восстановления внутри транзакции.
- Инструкция `RELEASE SAVEPOINT` удаляет точку сохранения и освобождает захваченные ею ресурсы.
- Инструкция `COMMIT` сообщает об успешном завершении транзакции и вносит все изменения в базу данных, делая их постоянными.
- Инструкция `ROLLBACK` предлагает СУБД отменить транзакцию и все изменения, уже внесенные в базу данных данной транзакцией.
- Транзакции играют ключевую роль при восстановлении базы данных после системного сбоя. В восстановленной базе данных остаются результаты выполнения только тех транзакций, которые были завершены на момент сбоя.
- Транзакции играют ключевую роль при параллельном доступе к данным в многопользовательской базе данных. Пользователю или программе гарантируется, что на их транзакцию не повлияют транзакции других пользователей.
- Иногда конфликт с другой параллельной транзакцией может привести к отмене транзакции не по ее вине. Приложение должно быть готово к решению этой проблемы в случае ее возникновения.
- Одной из наиболее сложных областей использования и настройки большой базы данных является управление транзакциями и их влияние на производительность СУБД. Здесь основные коммерческие СУБД проявляют большое разнообразие, по-разному решая эти вопросы.
- Многие СУБД для обработки параллельных транзакций применяют методику блокировки. Изменение параметров блокировок и инструкции явной блокировки обеспечивают возможность тонкой настройки обработки транзакций и повышения производительности базы данных.
- Альтернативой блокировкам служит поддерживаемый рядом СУБД метод управления версиями.

IV ЧАСТЬ

Структура базы данных

Глава 13
Создание базы данных

Глава 14
Представления

Глава 15
SQL и безопасность

Глава 16
Системный каталог

Одной из важных задач SQL является определение структуры и организации базы данных. В четырех последующих главах описываются средства SQL, выполняющие эту задачу. В главе 13, "Создание базы данных", рассказывается, как следует создавать базу данных и ее таблицы. В главе 14, "Представления", рассматриваются представления, важная возможность SQL, позволяющая предоставлять различным пользователям по-разному организованные данные. Вопросы безопасности и защиты хранимых данных, имеющиеся в SQL, представлены в главе 15, "SQL и безопасность". Наконец, в главе 16, "Системный каталог", рассматривается системный каталог — совокупность системных таблиц, описывающих структуру базы данных.

13
ГЛАВА

Создание базы данных

Большинству пользователей не приходится самостоятельно создавать базы данных. Как правило, они используют программный или интерактивный SQL для доступа к базе данных, созданной кем-то другим. Например, в типичной корпоративной базе данных ее администратор может выдать пользователю разрешение на выборку и, возможно, на изменение хранимых данных. Однако администратор не позволит пользователю создавать новые базы данных или изменять структуру существующих таблиц.

Тем не менее, по мере приобретения опыта работы с SQL, вы, вероятно, захотите создавать собственные таблицы для хранения своих данных, например результатов технических испытаний или плана-прогноза объема продаж. При работе в многопользовательской среде может возникнуть потребность в создании нескольких таблиц или даже целой базы данных для совместного использования с другими сотрудниками. Если вы работаете с базой данных, расположенной на персональном компьютере, то вам наверняка потребуется создавать собственные таблицы и базы данных для поддержки своих прикладных программ.

В настоящей главе рассматриваются средства языка SQL, позволяющие создавать таблицы и базы данных и определять их структуру.

Язык определения данных

Инструкции `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `COMMIT` и `ROLLBACK`, рассмотренные в предыдущих частях книги, предназначены для обработки данных. В совокупности эти инструкции называются *языком обработки данных* или DML (Data Manipulation Language). Инструкции DML могут модифицировать информацию, хранимую в базе данных, но не могут изменять ее структуру. Например, ни одна из этих инструкций не позволяет создавать и удалять таблицы или столбцы.

Для изменения структуры базы данных предназначен другой набор инструкций SQL, так называемый *язык определения данных* или DDL (Data Definition Language). С помощью инструкций DDL можно выполнить следующее:

- определить структуру новой таблицы и создать ее;
- удалить таблицу, которая больше не нужна;
- изменить определение существующей таблицы;
- определить виртуальную таблицу (или представление) данных;
- обеспечить безопасность базы данных;
- создать индекс для ускорения доступа к таблице;
- управлять физическим размещением данных.

В большинстве случаев инструкции DDL обеспечивают высокий уровень доступа к данным и позволяют пользователю не вникать в детали хранения информации в базе данных на физическом уровне. Они оперируют абстрактными объектами базы данных, такими как таблицы и столбцы. Однако DDL не может не затрагивать вопросов, связанных с физической памятью. Естественно, что инструкции и предложения DDL, управляющие физической памятью, могут быть разными в различных СУБД.

Ядро языка определения данных образуют три команды:

- **CREATE** (создать), позволяющая определить и создать объект базы данных;
- **DROP** (удалить), служащая для удаления существующего объекта базы данных;
- **ALTER** (изменить), посредством которой можно изменить определение объекта базы данных.

Все основные реляционные СУБД позволяют использовать три указанные команды DDL во время работы. Таким образом, структура реляционной базы данных является динамической. Например, СУБД может создавать, удалять или изменять таблицы, одновременно с этим обеспечивая доступ пользователям к базе данных. Это — одно из главных преимуществ реляционных баз данных по сравнению с более ранними системами, в которых изменять структуру базы данных можно было только после прекращения работы СУБД. Это означает, что с течением времени реляционная база данных может расти и изменяться. Ее промышленная эксплуатация может продолжаться в то время, когда в базу данных добавляются все новые таблицы и приложения.

Хотя DDL и DML являются двумя отдельными частями SQL, в большинстве реляционных СУБД такое разделение существует лишь на абстрактном уровне. Обычно инструкции DDL и DML в СУБД абсолютно равноправны, и их можно произвольно чередовать как в интерактивном, так и в программном SQL. Если программе или пользователю требуется таблица для временного хранения результатов, они могут создать эту таблицу, заполнить ее, проделать с данными необходимую работу и затем удалить таблицу. Это большое преимущество по сравнению с более ранними моделями представления данных, когда структура базы данных жестко фиксировалась при ее создании.

Хотя практически все коммерческие СУБД поддерживают DDL как неотъемлемую часть языка SQL, исходный стандарт SQL1 этого не требовал. В SQL1 между инструкциями DDL и DML имеется четкое разделение и допускается реализовать DML как надстройку над нереляционным ядром базы данных. В последующих

версиях стандарта SQL все еще имеется разграничение между различными типами инструкций SQL (инструкции DDL называются инструкциями SQL-схемы, а инструкции DML — инструкциями SQL-данных и инструкциями SQL-транзакций), но в то же время сам стандарт приведен в соответствие реальным принципам реализации современных СУБД: в нем требуется, чтобы инструкции DDL можно было выполнять как в интерактивном режиме, так и в приложениях.

В стандарте SQL определены только те части DDL, которые относительно независимы от структур физического хранения, особенностей операционных систем и других специфических особенностей СУБД. На практике все СУБД включают множество дополнений к DDL, связанных со спецификой реализации каждой конкретной СУБД. Ниже в настоящей главе описываются различия между стандартом ANSI/ISO и реализацией DDL в популярных реляционных СУБД.

Создание базы данных

В СУБД, установленных на мэйнфреймах или в крупных корпоративных сетях, за создание новых баз данных отвечает только администратор. В СУБД, установленных на серверах более низкого уровня, отдельным пользователям может быть разрешено создавать собственные базы данных, но обычно в таких СУБД базы данных создаются централизованно, а пользователи затем работают с ними. Если вы работаете с базой данных на персональном компьютере, то, скорее всего, являетесь как ее администратором, так и пользователем, и вам придется создавать базу самостоятельно.

В стандарте SQL1 содержится спецификация языка SQL, используемого для описания структуры базы данных, но не указывается способ создания базы данных, поскольку в различных СУБД применялись разные подходы к этому вопросу. Эти различия остались и сегодня. Методы создания баз данных, применяемые в ведущих реляционных СУБД, отчетливо иллюстрируют эти различия.

- В СУБД DB2 структура базы данных определена по умолчанию. База данных ассоциируется с выполняемой копией серверного обеспечения DB2, и пользователь получает доступ к базе данных, подключаясь к серверу DB2. Таким образом, база данных создается в процессе инсталляции DB2 на конкретную компьютерную систему.
- В СУБД Oracle база данных создается в процессе установки программного обеспечения, как и в DB2. Однако изредка администратор Oracle самостоятельно создает базу данных при помощи команды `CREATE DATABASE` или графического приложения из поставки Oracle, устанавливая определенные параметры базы данных, наилучшим образом соответствующие ее ожидаемому использованию. Как правило, каждая копия программного обеспечения СУБД Oracle управляет единственной базой данных, указанной в конфигурационном файле; пользовательские таблицы располагаются в схемах в этой базе данных. Заметим, что конкретный сервер или мэйнфрейм могут управлять несколькими базами данных, но в этом случае каждая из них требует работы собственной копии программного обеспечения СУБД.

- В СУБД Microsoft SQL Server и Sybase имеется инструкция CREATE DATABASE, которая является частью языка определения данных. Сопутствующая ей инструкция DROP DATABASE удаляет существующие базы данных. Эти инструкции можно использовать как в интерактивном, так и в программном SQL. Имена создаваемых баз данных отслеживаются в специальной "главной" базе данных, которая связана с конкретной инсталляцией СУБД. Хотя архитектура этих СУБД отличается от DB2 и Oracle, базы данных Sybase и SQL Server концептуально аналогичны схемам Oracle или DB2. Имена баз данных в пределах инсталляции SQL Server должны быть уникальны. С помощью параметров инструкции CREATE DATABASE можно задать физическое устройство, на котором размещается база данных.
- СУБД Informix Universal Server (ныне — продукт IBM) поддерживает инструкции CREATE DATABASE и DROP DATABASE. Инструкция CREATE DATABASE позволяет создать базу данных в указанной *области*, представляющей собой именованный участок дисковой памяти, находящийся под управлением СУБД. Кроме того, можно указать тип подключения к новой базе данных, а также сделать выбор между производительностью базы данных и целостностью ее данных при системных сбоях.
- MySQL также поддерживает инструкции SQL CREATE DATABASE и DROP DATABASE. Инструкция CREATE DATABASE управляет параметрами хранения базы данных и механизма СУБД. Каждая база данных хранится в своем подкаталоге корневого каталога инсталляции MySQL.

В стандарте SQL умышленно не дано определение термина *база данных*, поскольку в различных СУБД этот термин часто трактуется по-разному. Вместо него стандарт употребляет термин *каталог*, означающий именованную коллекцию системных таблиц, описывающих структуру базы данных (которая и называется базой данных в ведущих СУБД). (Дополнительная информация о структуре базы данных в SQL приведена позже в данной главе.) В стандарте не описано, как должен создаваться и уничтожаться каталог, и специально указано, что подобные вопросы зависят от реализации. Подчеркивается также, что вопросы о количестве каталогов в системе и о том, могут ли те или иные инструкции SQL обращаться к данным из различных каталогов, тоже зависят от реализации СУБД. На практике, как было показано выше, в большинстве СУБД для создания и удаления баз данных используется пара инструкций CREATE DATABASE/DROP DATABASE.

Определения таблиц

В реляционной базе данных наиболее важным элементом ее структуры является таблица. В многопользовательской базе данных основные таблицы, как правило, создает администратор, а пользователи просто обращаются к ним в процессе работы. Тем не менее при работе с такой базой данных часто оказывается удобным создавать собственные таблицы и хранить в них собственные данные, а также данные, извлеченные из других таблиц. Эти таблицы могут быть временными и существовать только в течение одного интерактивного SQL-сеанса, но могут сохраняться и в про-

должение нескольких недель или даже месяцев. В базе данных на персональном компьютере структуры таблиц являются еще более динамичными. Поскольку вы являетесь и пользователем, и администратором базы данных, можете создавать и удалять таблицы по мере необходимости, не думая о других пользователях.

Создание таблицы (CREATE TABLE)

Инструкция CREATE TABLE, синтаксическая диаграмма которой изображена на рис. 13.1, определяет новую таблицу и подготавливает ее к приему данных. Различные предложения инструкции задают элементы определения таблицы. Синтаксическая диаграмма инструкции кажется довольно громоздкой, поскольку требуется указать много элементов и параметров для них. Кроме того, некоторые параметры в одних СУБД присутствуют, а в других нет. На практике же создать таблицу относительно несложно.

После выполнения инструкции CREATE TABLE вы становитесь владельцем новой таблицы, которой присваивается указанное в инструкции имя. Если вы обладаете соответствующими привилегиями, можете создавать таблицы для других пользователей, используя квалифицированные имена с указанием владельца. Имя таблицы должно быть идентификатором, допустимым в SQL, и не должно конфликтовать с именами существующих таблиц. Таблица создается пустой, но СУБД подготавливает ее к приему данных, которые записываются с помощью инструкции INSERT.

Определения столбцов

Столбцы новой таблицы задаются в инструкции CREATE TABLE. Определения столбцов представляют собой заключенный в скобки список, элементы которого отделены друг от друга запятыми. Порядок следования определений столбцов в списке определяет расположение столбцов в таблице. В инструкции CREATE TABLE, поддерживаемой основными СУБД, каждое определение столбца содержит следующую информацию.

- **Имя столбца** используется для обращения к столбцу в инструкциях SQL. Каждый столбец в таблице должен иметь уникальное имя, но в разных таблицах имена столбцов могут совпадать.
- **Тип данных столбца** указывает, данные какого вида хранятся в столбце. Типы данных были рассмотрены в главе 5, "Основы SQL". В стандарте SQL указаны и домены (описаны в главе 11, "Целостность данных"), но они поддерживаются только некоторыми из современных СУБД. Для некоторых типов, например VARCHAR и DECIMAL, требуется дополнительная информация, такая как длина или число десятичных разрядов. Эта дополнительная информация заключается в скобки за ключевым словом, определяющим тип данных.
- **Обязательность данных** определяет, допускаются ли в данном столбце значения NULL.
- **Значение по умолчанию**. Это необязательное значение *по умолчанию*, которое заносится в столбец в том случае, если в инструкции INSERT для таблицы не указано значение для данного столбца.

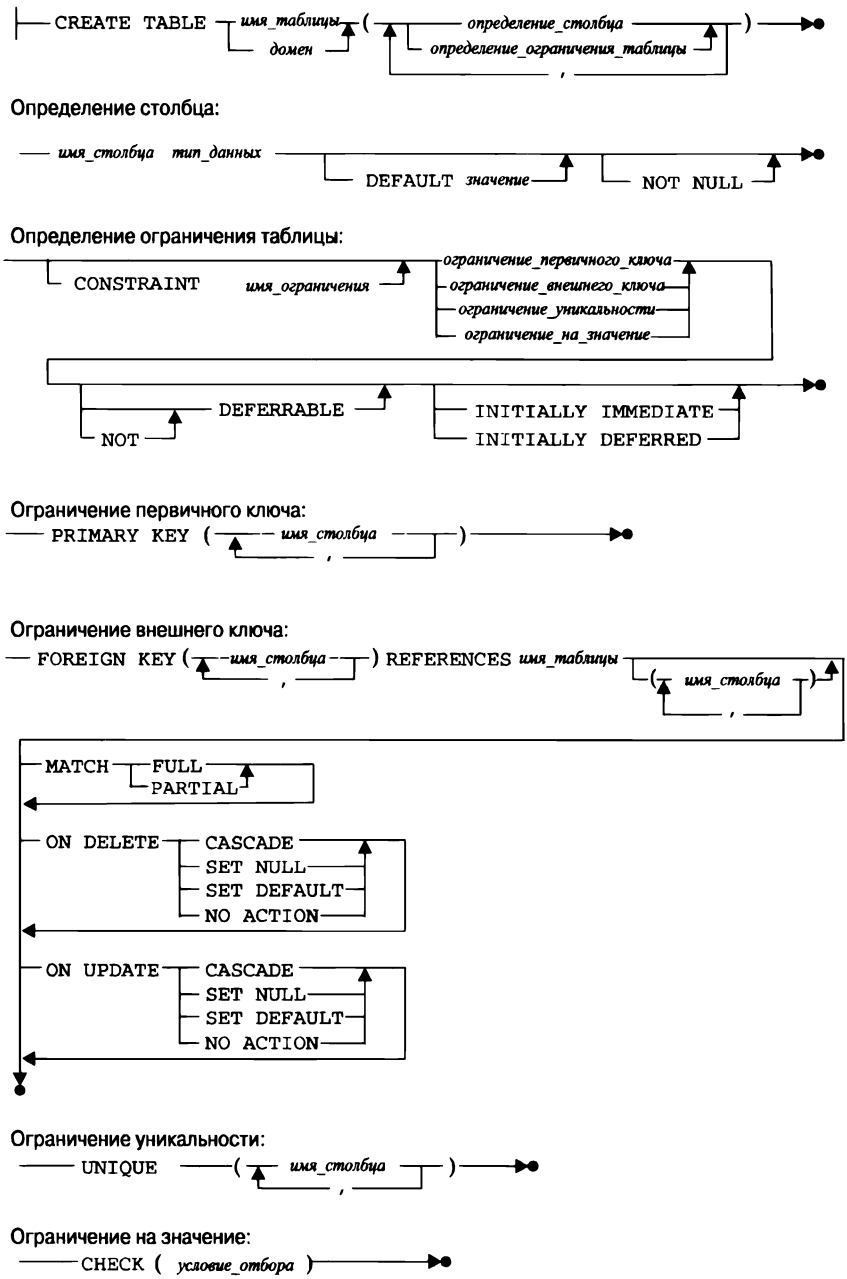


Рис. 13.1. Синтаксическая диаграмма инструкции CREATE TABLE

Стандарт SQL позволяет указать в определении столбца несколько дополнительных элементов, с помощью которых можно установить, что столбец должен содержать уникальные значения либо являться первичным или внешним ключом, а также ограничить допустимые значения данных в столбце. Эти элементы, по сути, пред-

ставляют собой варианты других предложений инструкции CREATE TABLE, предназначенные для одного столбца, и описываются в последующих разделах как составляющие этой инструкции.

Ниже приведено несколько простых инструкций CREATE TABLE для таблиц из учебной базы данных.

Определение таблицы OFFICES и ее столбцов.

```
CREATE TABLE OFFICES
     (OFFICE INTEGER      NOT NULL,
        CITY VARCHAR(15)  NOT NULL,
      REGION VARCHAR(10)  NOT NULL,
         MGR INTEGER,
      TARGET DECIMAL(9,2),
       SALES DECIMAL(9,2) NOT NULL);
```

Определение таблицы ORDERS и ее столбцов.

```
CREATE TABLE ORDERS
  (ORDER_NUM INTEGER      NOT NULL,
  ORDER_DATE DATE         NOT NULL,
        CUST INTEGER      NOT NULL,
         REP INTEGER,
         MFR CHAR(3)      NOT NULL,
     PRODUCT CHAR(5)      NOT NULL,
         QTY INTEGER      NOT NULL,
      AMOUNT DECIMAL(9,2) NOT NULL);
```

В различных СУБД инструкции CREATE TABLE для одной и той же таблицы могут немного отличаться, поскольку каждая СУБД поддерживает собственный набор типов данных и использует собственные ключевые слова для их идентификации в определениях столбцов. Кроме того, в стандарте SQL в определении столбца разрешается вместо базового типа данных указывать *домен* (домены описывались в главе 11, "Целостность данных"). Домен представляет собой именованное описание множества допустимых значений столбца, хранящееся в базе данных. В основе определения домена лежит один из базовых типов данных, на значения которого наложены дополнительные ограничения, проверяемые при каждом внесении данных в соответствующий столбец. Если, например, в базе данных, совместимой со стандартом SQL, создан следующий домен:

```
CREATE DOMAIN VALID_OFFICE_ID INTEGER
 CHECK (VALUE BETWEEN 11 AND 99);
```

то определение таблицы OFFICES можно записать так.

Определение таблицы OFFICES и ее столбцов.

```
CREATE TABLE OFFICES
     (OFFICE VALID_OFFICE_ID NOT NULL,
        CITY VARCHAR(15)     NOT NULL,
      REGION VARCHAR(10)     NOT NULL,
         MGR INTEGER,
      TARGET DECIMAL(9,2),
       SALES DECIMAL(9,2)    NOT NULL);
```

Теперь СУБД будет автоматически проверять каждую добавляемую в эту таблицу строку и выяснять, находится ли указанный в строке номер офиса в определенном диапазоне значений. Домены особенно эффективны, когда в базе данных одни и те же ограничения применимы сразу к нескольким столбцам. В нашей учебной базе данных номера офисов появляются в таблицах OFFICES и SALESREPS, поэтому домен VALID_OFFICE_ID можно применить к соответствующим столбцам обеих таблиц. В промышленных базах данных могут существовать десятки и сотни столбцов, данные которых относятся к одному и тому же домену.

Значения по умолчанию и отсутствующие значения

В определении каждого столбца указывается, могут ли в нем отсутствовать данные, т.е. допускается ли хранение в нем значений NULL. В большинстве СУБД и в стандарте SQL по умолчанию считается, что значения NULL допускаются. Если же столбец обязательно должен содержать данные в каждой строке, в его определение необходимо включить ограничение NOT NULL. В СУБД Sybase и SQL Server, наоборот, предполагается, что значения NULL недопустимы, если иное явно не объявлено в определении столбца.

Как стандарт SQL, так и многие реляционные СУБД поддерживают задание для столбцов значений по умолчанию, которые указываются в определении столбца. Вот пример инструкции CREATE TABLE для таблицы OFFICES, задающей значения по умолчанию.

Определение таблицы OFFICES со значениями по умолчанию (стандарт ANSI/ISO).

```
CREATE TABLE OFFICES
     (OFFICE INTEGER      NOT NULL,
        CITY VARCHAR(15)  NOT NULL,
      REGION VARCHAR(10)  NOT NULL DEFAULT 'Eastern',
         MGR INTEGER              DEFAULT 106,
      TARGET DECIMAL(9,2)         DEFAULT NULL,
       SALES DECIMAL(9,2) NOT NULL DEFAULT 0.00);
```

При таком определении таблицы в процессе ввода в нее сведений о новом офисе необходимо задать только идентификатор офиса и город, в котором он расположен. По умолчанию устанавливаются: район — Eastern, менеджер офиса — Сэм Кларк (идентификатор служащего 106), текущий объем продаж — нуль, план продаж — NULL. Отметим, что в определении столбца TARGET можно не указывать описание DEFAULT NULL; по умолчанию он все равно будет принимать значения NULL.

Определения первичного и внешнего ключей

Кроме определений столбцов таблицы, в инструкции CREATE TABLE указывается информация о первичном ключе таблицы и ее связях с другими таблицами базы данных. Эта информация содержится в предложениях PRIMARY KEY и FOREIGN KEY. Вначале они поддерживались в IBM SQL, а затем были внесены в стандарт ANSI/ISO. Большинство основных реализаций SQL поддерживает эти предложения.

В предложении PRIMARY KEY задается столбец или столбцы, которые образуют первичный ключ таблицы. Как говорилось в главе 4, "Реляционные базы данных", этот столбец (или комбинация столбцов) служит в качестве уникального иденти-

фикатора строк таблицы. СУБД автоматически следит за тем, чтобы первичный ключ каждой строки таблицы имел уникальное значение. Кроме того, в определениях столбцов первичного ключа должно быть указано, что они не могут содержать значения NULL (имеют ограничение NOT NULL).

В предложении FOREIGN KEY задается внешний ключ таблицы и определяется связь, которую он создает для нее с другой (родительской) таблицей. В этом предложении указывается следующее:

- столбец или столбцы создаваемой таблицы, которые образуют внешний ключ;
- таблица, связь с которой создает внешний ключ. Это родительская таблица; определяемая таблица в данном отношении является дочерней;
- необязательный список имен столбцов родительской таблицы, которые соответствуют столбцам внешнего ключа определяемой таблицы. Если имена столбцов опущены, в родительской таблице обязаны быть столбцы с именами, идентичными именам столбцов во внешнем ключе;
- необязательное имя для этого отношения; оно не используется в инструкциях SQL, но может появляться в сообщениях об ошибках и потребуется в дальнейшем, если будет необходимо удалить внешний ключ;
- как СУБД должна трактовать значения NULL в одном или нескольких столбцах внешнего ключа при связывании его со строками таблицы-предка;
- необязательное правило удаления для данного отношения (CASCADE, SET NULL, SET DEFAULT или NO ACTION, как описывалось в главе 11, "Целостность данных"), которое определяет действие, предпринимаемое при удалении строки родительской таблицы;
- необязательное правило обновления для данного отношения (эти правила описаны в главе 11, "Целостность данных"), которое определяет действие, предпринимаемое при обновлении первичного ключа в строке родительской таблицы;
- необязательное условие на значения, которое ограничивает данные в таблице так, чтобы они отвечали определенному критерию отбора.

Ниже приводится расширенная инструкция CREATE TABLE для таблицы ORDERS, в которую входит определение первичного ключа и трех внешних ключей таблицы.

Определение таблицы ORDERS с первичным и внешними ключами.

```
CREATE TABLE ORDERS
  (ORDER_NUM INTEGER       NOT NULL,
  ORDER_DATE DATE          NOT NULL,
        CUST INTEGER       NOT NULL,
         REP INTEGER,
         MFR CHAR(3)       NOT NULL,
     PRODUCT CHAR(5)       NOT NULL,
         QTY INTEGER       NOT NULL,
      AMOUNT DECIMAL(9,2)  NOT NULL,
```

```
PRIMARY KEY (ORDER_NUM),
 CONSTRAINT PLACEDBY
FOREIGN KEY (CUST)
 REFERENCES CUSTOMERS
  ON DELETE CASCADE,
 CONSTRAINT TAKENBY
FOREIGN KEY (REP)
 REFERENCES SALESREPS
  ON DELETE SET NULL,
 CONSTRAINT ISFOR
FOREIGN KEY (MFR, PRODUCT)
 REFERENCES PRODUCTS
  ON DELETE RESTRICT);
```

На рис. 13.2 изображены три созданные этой инструкцией связи с присвоенными им именами. В общем случае связи желательно давать имя, поскольку оно помогает лучше понять, какая именно связь создана внешним ключом. Например, каждый заказ делается клиентом, идентификатор которого находится в столбце CUST таблицы ORDERS. Связь с этим столбцом получила имя PLACEDBY (кем сделан).

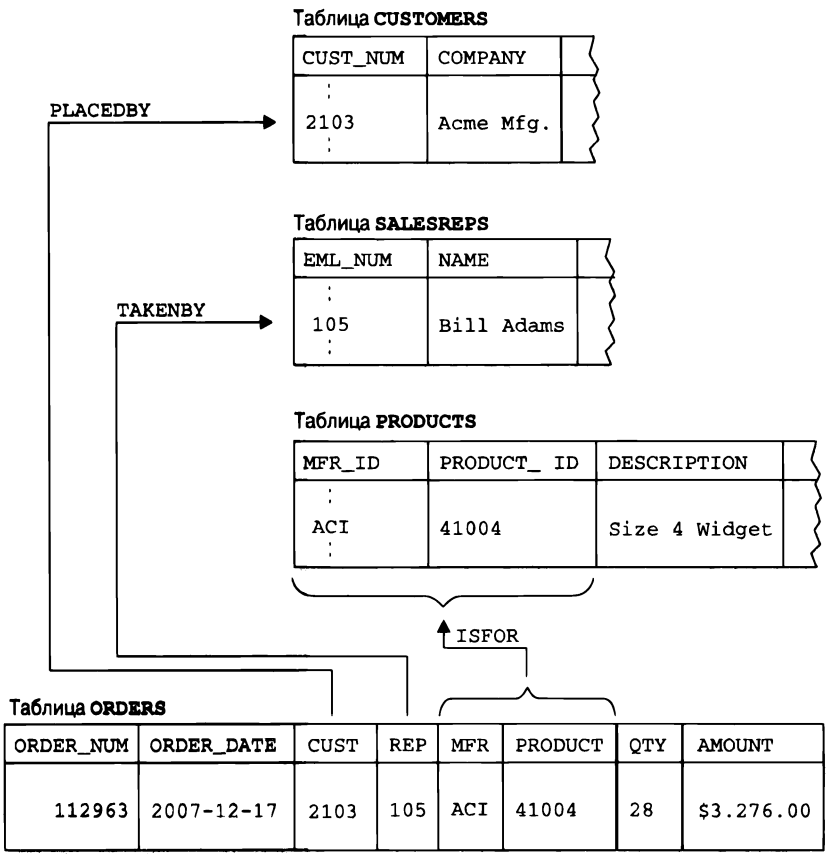


Рис. 13.2. Имена связей в инструкции CREATE TABLE

Когда СУБД выполняет инструкцию CREATE TABLE, она сравнивает определение каждого внешнего ключа с определениями связанных таблиц. СУБД проверяет, соответствуют ли друг другу внешний ключ и первичный ключ в связанных таблицах как по числу столбцов, так и по типу данных. Для того чтобы такая проверка была возможна, связанная таблица уже должна быть определена.

Заметим, что предложение FOREIGN KEY также определяет правила удаления и обновления для создаваемого им отношения "предок-потомок". Правила удаления и обновления, а также действия, которые они могут запускать, рассматриваются в главе 11, "Целостность данных". Если правила явно не указаны, СУБД использует правила по умолчанию (NO ACTION).

Если две или более таблиц образуют ссылочный цикл (как таблицы OFFICES и SALESREPS), то для первой создаваемой таблицы невозможно определить внешний ключ, так как связанная с нею таблица еще не существует. СУБД откажется выполнять инструкцию CREATE TABLE, выдав сообщение о том, что в определении таблицы присутствует ссылка на несуществующую таблицу. В этом случае необходимо создать таблицу без определения внешнего ключа и добавить данное определение позже с помощью инструкции ALTER TABLE. (В стандарте SQL и нескольких ведущих СУБД предлагается иной подход к решению данной проблемы — инструкция CREATE SCHEMA, с помощью которой одновременно создается все множество таблиц. Эта инструкция, а также объекты базы данных, входящие в состав схемы в SQL, описываются позже в данной главе.)

Условия уникальности

Стандарт SQL определяет, что условия уникальности также задаются в инструкции CREATE TABLE, с применением предложения UNIQUE, показанного на рис. 13.1. Вот как выглядит инструкция CREATE TABLE для таблицы OFFICES с включенным в нее условием уникальности для столбца CITY.

Определение таблицы OFFICES с условием уникальности.

```
CREATE TABLE OFFICES
     (OFFICE INTEGER      NOT NULL,
        CITY VARCHAR(15)  NOT NULL,
      REGION VARCHAR(10)  NOT NULL,
         MGR INTEGER,
      TARGET DECIMAL(9,2),
       SALES DECIMAL(9,2) NOT NULL,
 PRIMARY KEY (OFFICE),
  CONSTRAINT HASMGR
 FOREIGN KEY (MGR)
  REFERENCES SALESREPS
   ON DELETE SET NULL,
      UNIQUE (CITY));
```

Если первичный или внешний ключ включает в себя только один столбец либо если условие уникальности или условие на значение касаются одного столбца, стандарт ANSI/ISO разрешает использовать сокращенную форму ограничения, при которой оно просто добавляется в конец определения столбца, как показано в следующем примере.

Определение таблицы OFFICES с условием уникальности (синтаксис ANSI/ISO).

```
CREATE TABLE OFFICES
     (OFFICE INTEGER      NOT NULL  PRIMARY KEY,
       CITY VARCHAR(15)  NOT NULL  UNIQUE,
     REGION VARCHAR(10)  NOT NULL,
        MGR INTEGER      REFERENCES SALESREPS,
     TARGET DECIMAL(9,2),
      SALES DECIMAL(9,2) NOT NULL);
```

Такой синтаксис поддерживается в ряде ведущих СУБД, в частности, в SQL Server, Informix, Sybase и DB2.

Ограничения на значения столбцов

Еще одна возможность обеспечения целостности данных в SQL, ограничение CHECK (которое уже рассматривалось в главе 11, "Целостность данных"), также задается в инструкции CREATE TABLE. Оно содержит условие на значение (идентичное условию отбора в запросе на выборку), проверяемое всякий раз при попытке модификации содержимого таблицы (с помощью инструкций INSERT, UPDATE или DELETE). Если после модификации условие остается истинным, такое изменение допускается; в противном случае СУБД отвергает изменения и выдает сообщение об ошибке. Ниже приведена инструкция CREATE TABLE, создающая таблицу OFFICES с простым ограничением на значение столбца TARGET, позволяющим гарантировать, что плановый объем продаж офиса всегда будет больше $0.00.

Определение таблицы OFFICES с ограничением на значение столбца TARGET.

```
CREATE TABLE OFFICES
     (OFFICE INTEGER      NOT NULL,
       CITY VARCHAR(15)  NOT NULL,
     REGION VARCHAR(10)  NOT NULL,
        MGR INTEGER,
     TARGET DECIMAL(9,2),
      SALES DECIMAL(9,2) NOT NULL,
 PRIMARY KEY (OFFICE),
  CONSTRAINT HASMGR
 FOREIGN KEY (MGR)
  REFERENCES SALESREPS
   ON DELETE SET NULL,
       CHECK (TARGET >= 0.00));
```

Дополнительно можно задать необязательное имя ограничения, которое будет отображаться в сообщениях об ошибках, выдаваемых СУБД при нарушении данного ограничения. Вот пример более сложного ограничения для таблицы SALESREPS, требующего соблюдения правила "служащим, принятым на работу после 1 января 2006 года, нельзя назначать план больше чем $300000". Этому ограничению назначено имя QUOTA_CAP.

```
CREATE TABLE SALESREPS
   (EMPL_NUM INTEGER      NOT NULL,
        NAME VARCHAR (15) NOT NULL,
          .
          .
          .
```

```
  CONSTRAINT WORKSIN
 FOREIGN KEY (REP_OFFICE)
  REFERENCES OFFICES
   ON DELETE SET NULL
  CONSTRAINT QUOTA_CAP CHECK ((HIRE_DATE < "2006-01-01") OR
                              (QUOTA <= 300000)));
```

Подобный синтаксис поддерживается многими ведущими СУБД.

Определение физического хранения*

Обычно в инструкцию CREATE TABLE входит одно или несколько необязательных предложений, в которых для создаваемой таблицы определяются характеристики физического хранения данных. Этими предложениями пользуется, в основном, администратор, для того чтобы оптимизировать работу промышленной базы данных. В силу своей природы эти предложения являются весьма специфическими, зависят от конкретной СУБД и для большинства пользователей не представляют какого-либо практического интереса. Различные структуры физической памяти, применяемые в разных СУБД, характеризуют уровень сложности СУБД и то, для каких целей она предназначена.

Большинство баз данных на персональных компьютерах имеет очень простую организацию физической памяти. СУБД данного класса хранят всю базу данных в едином файле или используют отдельный файл для каждой таблицы и могут требовать, чтобы вся таблица или база данных размещались на едином физическом диске.

В многопользовательских базах данных физическая память, как правило, организована более сложным образом, что обеспечивает повышение производительности базы данных. Например, в СУБД Ingres администратор может задать несколько именованных *областей памяти*, в которых будет храниться информация базы данных. Эти области памяти могут быть распределены по нескольким дисковым томам для параллельного выполнения операций ввода-вывода. В инструкции CREATE TABLE при необходимости можно указать одну или несколько областей памяти для хранения таблицы:

```
CREATE TABLE OFFICES (определение_таблицы)
  WITH LOCATION = (ОБЛАСТЬ1, ОБЛАСТЬ2, ОБЛАСТЬ3);
```

Задавая группу областей памяти, содержимое таблицы можно распределить по нескольким дисковым томам, чтобы обеспечить параллельный доступ к таблице.

Аналогичный подход используется и в Sybase ASE, где администратор может задать одно или несколько *логических устройств базы данных*, предназначенных для хранения данных. Соответствие между логическим устройством и реальным жестким диском устанавливается с помощью специальной утилиты, а не посредством SQL. Инструкция CREATE DATABASE позволяет затем указать, на каких логических устройствах должна размещаться база данных.

```
CREATE DATABASE имя
    ON УСТРОЙСТВО1, УСТРОЙСТВО2, УСТРОЙСТВО3;
```

В пределах логического устройства администратор может определить логические *сегменты*, пользуясь одной из системных хранимых процедур. Наконец, в ин-

струкции CREATE TABLE можно указать сегмент, в котором будет располагаться создаваемая таблица.

```
CREATE TABLE OFFICES (определение_таблицы)
    ON SEGMENT SEG1A;
```

В DB2 также применяется весьма сложная схема управления физической памятью, основанная на концепции *табличных пространств* и *узловых групп*. Табличное пространство представляет собой хранилище на логическом уровне, а узловая группа связана непосредственно с физической памятью. Когда в DB2 создается таблица, ее можно закрепить за каким-нибудь определенным табличным пространством.

```
CREATE TABLE OFFICES (определение_таблицы)
    IN ADMINDB.табличное_пространство
```

В отличие от СУБД Sybase, в DB2 основная задача по управлению хранением данных возлагается на язык SQL и его инструкции CREATE TABLESPACE и CREATE NODEGROUP. Как следствие, порядок записи имен файлов и каталогов в этих инструкциях зависит от операционной системы, в которой работает DB2. Имеются и другие инструкции, позволяющие задать буфер промежуточного хранения, скорость обмена данными и другие характеристики, связанные с физическим хранением данных. DB2 использует всю эту информацию в алгоритмах оптимизации производительности.

SQL Server поддерживает создание файловых групп, которые представляют собой логический эквивалент табличных пространств. Каждая файловая группа имеет один или несколько связанных с нею физических файлов. При создании базы данных с ней связывается файловая группа по умолчанию, и таблицы, создаваемые в этой базе данных, хранятся в этой группе, если только иное не указано явно в инструкции CREATE TABLE наподобие следующей.

```
CREATE TABLE OFFICES (определение_таблицы)
    ON имя_файловой_группы;
```

Oracle поддерживает создание табличных пространств, в некоторой степени аналогичных DB2. С табличным пространством связаны один или несколько физических файлов. Табличное пространство по умолчанию может быть назначено глобально или для каждой пользовательской схемы. При создании таблицы можно указать табличное пространство и различные атрибуты физического хранения данных, такие как изначально выделенное пространство, шаг увеличения при необходимости большего пространства или процент увеличения каждого очередного выделения памяти.

```
CREATE TABLE OFFICES (определение_таблицы)
  TABLESPACE имя_табличного_пространства
     STORAGE (INITIAL     начальный_размер
              NEXT        размер_увеличения
              PCTINCREASE процент_роста);
```

Удаление таблицы (DROP TABLE)

С течением времени структура базы данных изменяется. Для представления новых объектов создаются новые таблицы, а некоторые старые таблицы становятся ненужными. Эти ненужные таблицы можно удалить из базы данных инструкцией DROP TABLE, показанной на рис. 13.3.

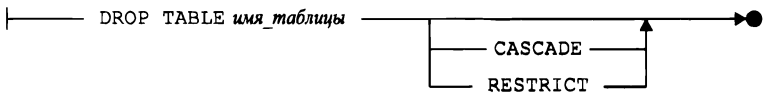


Рис. 13.3. Синтаксическая диаграмма инструкции DROP TABLE

Инструкция содержит имя удаляемой таблицы. Обычно пользователь удаляет одну из своих собственных таблиц и указывает в инструкции неквалифицированное имя таблицы. Имея соответствующее разрешение, можно удалить и таблицу другого пользователя, но в этом случае необходимо указать полное имя таблицы. Вот несколько примеров инструкции DROP TABLE.

Таблица CUSTOMERS была заменена двумя новыми таблицами — CUST_INFO и ACCOUNT_INFO — и больше не нужна.

```
DROP TABLE CUSTOMERS;
```

Сэм дает вам разрешение удалить его таблицу BIRTHDAYS.

```
DROP TABLE SAM.BIRTHDAYS;
```

Когда инструкция DROP TABLE удаляет из базы данных таблицу, ее определение и все содержимое теряются. Восстановить данные невозможно, и, чтобы повторно создать определение таблицы, следует использовать новую инструкцию CREATE TABLE. (Однако в настоящее время Oracle позволяет восстанавливать удаленные таблицы из Корзины, и другие производители могут последовать этому примеру.) Так как выполнение инструкции DROP TABLE может привести к серьезным последствиям, пользоваться ею следует крайне осторожно!

Стандарт SQL требует, чтобы инструкция DROP TABLE включала в себя либо параметр CASCADE, либо RESTRICT, которые определяют, как влияет удаление таблицы на другие объекты базы данных (например, представления, рассматриваемые в главе 14, "Представления"), зависящие от этой таблицы. Если задан параметр RESTRICT и в базе данных имеются объекты, которые содержат ссылку на удаляемую таблицу, то выполнение инструкции DROP TABLE закончится неуспешно. В большинстве коммерческих СУБД допускается применение инструкции DROP TABLE без каких-либо параметров, однако имеются и исключения.

- MySQL допускает в целях совместимости применение RESTRICT и CASCADE, но по крайней мере до версии 5.0 они не оказывали никакого действия.
- Oracle по умолчанию использует режим RESTRICT (который не может быть указан явно) и требует применения ключевых слов CASCADE CONSTRAINTS вместо CASCADE.
- SQL Server и DB2 не поддерживают RESTRICT и CASCADE.

Изменение определения таблицы (ALTER TABLE)

В процессе работы с таблицей у пользователя часто возникает необходимость добавить в таблицу некоторую информацию. Например, в учебной базе данных может потребоваться выполнить следующее:

- добавить в каждую строку таблицы CUSTOMERS имя и номер телефона служащего компании-клиента, через которого поддерживается контакт, если необходимо использовать эту таблицу для связи с клиентами;
- добавить в таблицу PRODUCTS столбец с указанием минимального количества на складе, чтобы база данных могла автоматически предупреждать о том, что запас какого-либо товара стал меньше допустимого предела;
- сделать столбец REGION в таблице OFFICES внешним ключом для вновь созданной таблицы REGIONS, первичным ключом которой является название региона;
- удалить определение внешнего ключа для столбца CUST таблицы ORDERS, связывающего ее с таблицей CUSTOMERS, и заменить его определениями двух внешних ключей, связывающих столбец CUST с двумя вновь созданными таблицами CUST_INFO и ACCOUNT_INFO.

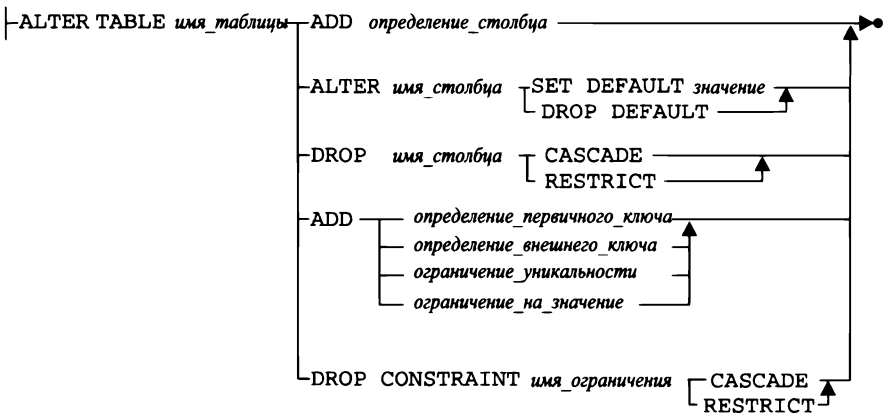


Рис. 13.4. Синтаксическая диаграмма инструкции ALTER TABLE

Эти и другие изменения можно осуществить с помощью инструкции ALTER TABLE, синтаксическая диаграмма которой изображена на рис. 13.4. Данная инструкция, как и DROP TABLE, обычно применяется пользователем по отношению к своим собственным таблицам. При наличии соответствующих прав и используя полное имя таблицы, можно изменять таблицы других пользователей. Как видно из рисунка, с помощью инструкции ALTER TABLE можно сделать следующее:

- добавить в таблицу определение столбца;
- удалить столбец из таблицы;
- изменить значение по умолчанию для какого-либо столбца;

- добавить или удалить первичный ключ таблицы;
- добавить или удалить внешний ключ таблицы;
- добавить или удалить условие уникальности;
- добавить или удалить условие на значение.

Предложения на рис. 13.4 изображены в соответствии со стандартом SQL. Во многих СУБД некоторые из них не используются либо используются специфические для конкретной СУБД предложения, которые изменяют другие, не представленные здесь характеристики таблицы. Стандарт SQL требует, чтобы инструкция ALTER TABLE применялась для единичного изменения таблицы. Например, для добавления столбца и определения нового внешнего ключа требуются две различные инструкции. В некоторых СУБД это ограничение ослаблено и допускается присутствие нескольких предложений в одной инструкции ALTER TABLE.

Добавление столбца

Чаще всего инструкция ALTER TABLE применяется для добавления столбца в существующую таблицу. Предложение с определением столбца в инструкции ALTER TABLE имеет точно такой же вид, как и в инструкции CREATE TABLE, и выполняет ту же функцию. Новое определение добавляется в конец определений столбцов таблицы, и в последующих запросах новый столбец будет крайним справа. СУБД обычно предполагает, что новый столбец во всех существующих строках содержит значения NULL. Если столбец объявлен как NOT NULL и со значением по умолчанию, то СУБД считает, что все его значения — значения по умолчанию.

Обратите внимание на то, что нельзя объявлять столбец просто как NOT NULL, поскольку СУБД подставляла бы в существующие строки значения NULL, нарушая тем самым заданное условие. (В действительности, когда вы добавляете новый столбец, СУБД может заносить во все существующие строки нового столбца значения NULL или значения по умолчанию. Некоторые СУБД обнаруживают тот факт, что строка слишком коротка для нового определения таблицы, только при выборке этой строки пользователем и расширяют ее значениями NULL (или значениями по умолчанию) непосредственно перед выводом на экран или передачей в программу пользователя.)

Ниже даны примеры инструкций ALTER TABLE, добавляющих новые столбцы.

Добавить контактный телефон и имя служащего компании-клиента в таблицу CUSTOMERS.

```
ALTER TABLE CUSTOMERS
  ADD CONTACT_NAME VARCHAR(30);

ALTER TABLE CUSTOMERS
  ADD CONTACT_PHONE CHAR(10);
```

Добавить в таблицу PRODUCTS столбец минимального количества товара на складе.

```
ALTER TABLE PRODUCTS
  ADD MIN_QTY INTEGER NOT NULL DEFAULT 0;
```

В первом примере новые столбцы будут иметь значения NULL для существующих клиентов. Во втором примере столбец MIN_QTY для существующих товаров будет содержать нули (0), что вполне уместно.

Когда инструкция ALTER TABLE впервые появилась в реализациях SQL, единственными структурными элементами таблиц были определения столбцов, поэтому было понятно, что означает предложение ADD этой инструкции. Со временем таблицы стали включать определения первичных и внешних ключей и прочих ограничений, а в предложении ADD стал указываться тип добавляемого ограничения. В целях унификации стандарт SQL позволяет добавлять после ключевых слов ADD слово COLUMN для явного указания на то, что создается столбец. С учетом этого, предыдущий пример можно записать так.

Добавить в таблицу PRODUCTS столбец минимального количества товара на складе.

```
ALTER TABLE PRODUCTS
  ADD COLUMN MIN_QTY INTEGER NOT NULL DEFAULT 0;
```

Удаление столбца

С помощью инструкции ALTER TABLE можно удалить из существующей таблицы один или несколько столбцов, если в них больше нет необходимости. Вот пример удаления столбца HIRE_DATE из таблицы SALESREPS.

Удалить столбец из таблицы SALESREPS.

```
ALTER TABLE SALESREPS
 DROP HIRE_DATE;
```

Стандарт SQL требует, чтобы одна инструкция ALTER TABLE использовалась для удаления только одного столбца, но в ряде ведущих СУБД такое ограничение снято.

Следует учитывать, что операция удаления столбца вызывает те же проблемы целостности данных, которые были описаны в главе 11, "Целостность данных", на примере обновления таблиц. Например, при удалении столбца, являющегося первичным ключом в каком-либо отношении, связанные с ним внешние ключи становится недействительными. Похожая проблема возникает, когда удаляется столбец, участвующий в проверке ограничения на значение другого столбца. Те же проблемы возникают и в представлениях, основанных на удаляемом столбце.

Описанные проблемы в стандарте SQL решены так же, как и в случае инструкций DELETE и UPDATE, — с помощью *правила удаления* (которые в стандарте именуются *поведением при удалении*). Можно выбрать одно из двух правил.

- **RESTRICT**. Если с удаляемым столбцом связан какой-либо объект в базе данных (внешний ключ, ограничение и т.п.), инструкция ALTER TABLE завершится выдачей сообщения об ошибке и столбец не будет удален.
- **CASCADE**. Любой объект базы данных (внешний ключ, ограничение и т.п.), связанный с удаляемым столбцом, *также* будет удален.

Правило CASCADE может вызвать лавину изменений, поэтому применять его следует с осторожностью. Лучше указывать правило RESTRICT, а связанные внешние ключи или ограничения обрабатывать с помощью дополнительных инструкций типа ALTER или DROP.

Изменение первичных и внешних ключей

Еще одним распространенным случаем применения инструкции ALTER TABLE является изменение или добавление определения первичных и внешних ключей таблицы. Так как поддержка первичных и внешних ключей включена во многие новые реляционные СУБД, данная форма инструкции ALTER TABLE является особенно полезной. С ее помощью можно информировать СУБД о межтабличных связях, уже существующих в базе данных, но не определенных явно.

Используя инструкцию ALTER TABLE, определения первичного и внешних ключей, в отличие от определений столбцов, можно как добавлять в таблицу, *так и удалять* из нее. Предложения, добавляющие определения первичного и внешнего ключей, являются точно такими же, как в инструкции CREATE TABLE, и выполняют те же функции. Предложения, удаляющие первичный или внешний ключи, являются довольно простыми, как видно из приведенных ниже примеров. Заметим, что удалить внешний ключ можно только тогда, когда создаваемая им связь имеет имя. Если имя присвоено не было, то задать эту связь в инструкции ALTER TABLE невозможно. В этом случае для удаления внешнего ключа необходимо удалить таблицу и воссоздать ее в новом формате.

Перед вами пример, когда определение внешнего ключа добавляется в существующую таблицу.

Сделать столбец REGION таблицы OFFICES внешним ключом для вновь созданной таблицы REGIONS, первичным ключом которой является название региона.

```
   ALTER TABLE OFFICES
ADD CONSTRAINT INREGION
   FOREIGN KEY (REGION)
    REFERENCES REGIONS;
```

Далее приведен пример инструкции ALTER TABLE, модифицирующей первичный ключ. Обратите внимание на то, что внешний ключ, соответствующий исходному первичному ключу, должен быть удален, так как он больше не является внешним ключом для изменяемой таблицы.

Изменить первичный ключ таблицы OFFICES.

```
ALTER TABLE SALESREPS
       DROP CONSTRAINT WORKSIN;

ALTER TABLE OFFICES
       DROP PRIMARY KEY;
```

Определения ограничений

Основу базы данных составляют таблицы, и в самых первых коммерческих реляционных СУБД они были единственным структурным элементом базы данных. С появлением первичных и внешних ключей вначале в DB2, а затем и в стандарте SQL понятие структуры базы данных было расширено и включило в себя *отношения* между таблицами. Позже, в процессе эволюции стандарта SQL и коммерческих СУБД, в структуру базы данных вошел новый элемент — ограничения, на-

кладываемые на данные, которые могут быть внесены в базу данных. Типы ограничений и решаемые с их помощью проблемы целостности данных рассматривались в главе 11, "Целостность данных".

Четыре типа ограничений (ограничения на первичный и внешний ключи, условие уникальности и условие на значение) тесно связаны с конкретной таблицей. Они задаются как часть инструкции CREATE TABLE и могут быть модифицированы или удалены с помощью инструкции ALTER TABLE. Два других типа ограничений — утверждения и домены — создаются как отдельные объекты в базе данных, не зависящие от определения какой бы то ни было таблицы.

Утверждения

Утверждение (assertion) накладывает ограничение на содержимое всей базы данных. Как и условие на значение, утверждение задается в виде условия отбора, однако может ограничивать содержимое нескольких таблиц. По этой причине утверждение задается как часть определения всей базы данных с помощью инструкции SQL CREATE ASSERTION. Предположим, вы хотите наложить на содержимое базы данных следующее ограничение: сумма заказов любого клиента не должна превышать предел его кредита. Это ограничение можно реализовать посредством такой инструкции.

```
CREATE ASSERTION CREDLIMIT
        CHECK ((CUSTOMERS.CUST_NUM = ORDERS.CUST) AND
               (SUM (AMOUNT) <= CREDIT_LIMIT));
```

Если это утверждение (названное CREDLIMIT) является частью определения базы данных, то СУБД проверяет его справедливость всякий раз, когда какая-либо инструкция SQL пытается модифицировать таблицу CUSTOMERS или ORDERS. Если впоследствии вы решите, что утверждение больше не нужно, то сможете удалить его с помощью инструкции DROP ASSERTION.

```
DROP ASSERTION CREDLIMIT;
```

В SQL нет инструкции ALTER ASSERTION. Если вы хотите изменить определение утверждения, вам придется удалить его и создать заново с помощью инструкции CREATE ASSERTION.

Хотя спецификация утверждений входит в стандарт SQL с 1992 года, ее поддерживают немногие реализации SQL. Фактически на момент написания данного материала она не поддерживалась ни одной из СУБД Oracle, DB2 UDB, SQL Server или MySQL.

Домены

Стандарт SQL реализует концепцию домена как часть определения базы данных. Как говорилось в главе 11, "Целостность данных", домен представляет собой именованную совокупность значений данных, применяемую в качестве дополнительного типа данных в определениях базы данных. Домен создается посредством инструкции CREATE DOMAIN. После создания домена внутри определения таблицы с ним можно обращаться как с обычным типом данных. Ниже показана инст-

рукция CREATE DOMAIN, создающая домен VALID_EMPL_IDS, который определяет диапазон допустимых идентификаторов служащих в учебной базе данных. Идентификатор служащего представляет собой целое число в диапазоне от 101 до 199.

```
CREATE DOMAIN VALID_EMPL_IDS INTEGER
 CHECK (VALUE BETWEEN 101 AND 199);
```

Если домен больше не нужен, его можно удалить с помощью инструкции DROP DOMAIN.

```
DROP DOMAIN VALID_EMPL_IDS CASCADE;
```

или

```
DROP DOMAIN VALID_EMPL_IDS RESTRICT;
```

Правила удаления CASCADE и RESTRICT здесь имеют тот же смысл, что и при удалении столбцов таблицы. Если указано правило CASCADE, то любой столбец, определенный с использованием этого домена, также будет удален. Если указано правило RESTRICT, то попытка удалить домен, с которым связан хотя бы один столбец, завершится неуспешно. В этом случае вам придется сначала удалить или изменить определения всех связанных столбцов. Это дополнительная мера защиты от случайного удаления столбцов (и, что более важно, содержащихся в них данных).

Как и утверждения, домены входят в стандарт SQL с 1992 года, но имеется лишь несколько коммерческих продуктов, поддерживающих их. В SQL Server и Oracle имеются инструкции CREATE TYPE, которые в определенной степени схожи с инструкциями создания доменов, но в DB2 UDB, MySQL и других ведущих СУБД такая поддержка отсутствует.

Псевдонимы, или синонимы (CREATE/DROP ALIAS)

Промышленные базы данных часто организованы подобно примеру базы данных, изображенному на рис. 13.5, где все основные таблицы собраны вместе и принадлежат администратору. Администратор базы данных дает другим пользователям права на доступ к таблицам, руководствуясь правилами обеспечения безопасности, описанными в главе 15, "SQL и безопасность". Вспомните, однако, что для обращения к таблицам других пользователей необходимо использовать полностью квалифицированные имена таблиц. На практике это означает, что в *каждом* запросе к главным таблицам на рис. 13.5 следует указывать полные имена таблиц, в результате чего запросы становятся длинными, а их ввод — утомительным.

Вывести имя, объем продаж, идентификатор офиса и объем продаж офиса каждого служащего.

```
SELECT NAME, OP_ADMIN.SALESREPS.SALES, OFFICE,
             OP_ADMIN.OFFICES.SALES
  FROM OP_ADMIN.SALESREPS, OP_ADMIN.OFFICES;
```

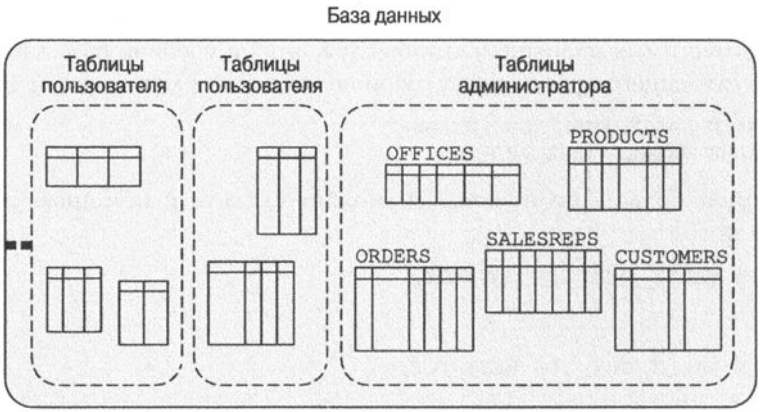


Рис. 13.5. Типичная организация промышленной базы данных

Для решения этой проблемы во многих СУБД вводится понятие *псевдонима* или *синонима*. Псевдоним — это назначаемое пользователем имя, которое заменяет имя некоторой таблицы. В DB2 псевдоним создается с помощью инструкции CREATE ALIAS. (В более ранних версиях DB2 существовала инструкция CREATE SYNONYM, а в Oracle или SQL Server она имеется до сих пор, хотя ее действие аналогично действию инструкции CREATE ALIAS.) Если бы вы были, например, пользователем по имени Джордж, то могли бы создать пару приведенных далее псевдонимов.

Создать псевдонимы для двух таблиц, принадлежащих другому пользователю.

```
CREATE ALIAS REPS
   FOR OP_ADMIN.SALESREPS;

CREATE ALIAS OFFICES
   FOR OP_ADMIN.OFFICES;
```

После создания псевдонима его можно использовать в запросах SQL как обычное имя таблицы. Тогда предыдущий запрос принимает следующий вид.

```
SELECT NAME, REPS.SALES, OFFICE, OFFICES.SALES
  FROM REPS, OFFICES;
```

Применение псевдонимов не изменяет смысл запроса, и вам по-прежнему необходимо иметь права для доступа к таблицам других пользователей. Тем не менее псевдонимы упрощают инструкции SQL, и последние приобретают такой вид, как если бы вы обращались к своим собственным таблицам. Если позднее вы решите, что больше не нуждаетесь в псевдонимах, то можете их удалить посредством инструкции DROP ALIAS.

Удалить ранее созданные псевдонимы.

```
DROP ALIAS REPS;
DROP ALIAS OFFICES;
```

Псевдонимы используются в DB2, Oracle, SQL Server и Informix. Однако в стандарте ANSI/ISO их нет.

Индексы (CREATE/DROP INDEX)

Одним из структурных элементов физической памяти, присутствующих в большинстве реляционных СУБД, является *индекс*. Индекс — это средство, обеспечивающее быстрый доступ к строкам таблицы на основе значений одного или нескольких столбцов. На рис. 13.6 изображена таблица PRODUCTS и два созданных для нее индекса. Один из индексов обеспечивает доступ к таблице на основе столбца DESCRIPTION. Другой обеспечивает доступ на основе первичного ключа таблицы, представляющего собой комбинацию столбцов MFR_ID и PRODUCT_ID.

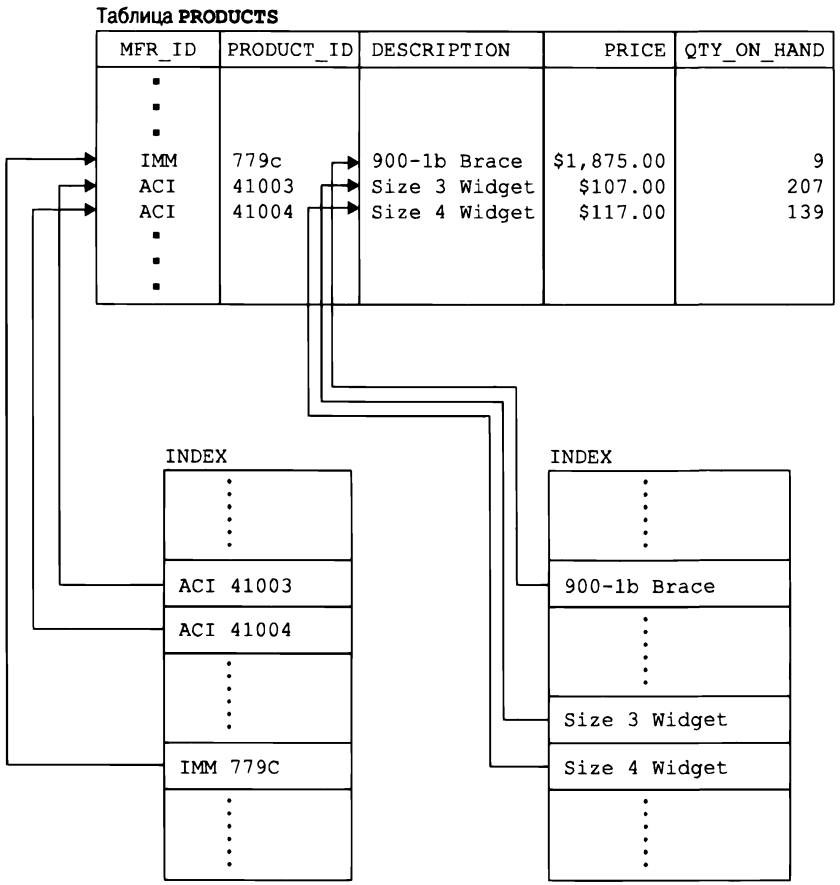


Рис. 13.6. Два индекса таблицы PRODUCTS

СУБД пользуется индексом так же, как вы пользуетесь предметным указателем книги. В индексе хранятся значения данных и указатели на строки, где эти данные встречаются. Данные в индексе располагаются в убывающем или возрастающем порядке, чтобы СУБД могла быстро найти требуемое значение. Затем по указателю СУБД может быстро найти строку, содержащую искомое значение.

Наличие или отсутствие индекса совершенно незаметно для пользователя, обращающегося к таблице. Рассмотрим, например, такую инструкцию SELECT.

Найти количество на складе и цену изделия "Size 4 Widget".

```
SELECT QTY_ON_HAND, PRICE
  FROM PRODUCTS
 WHERE DESCRIPTION = 'Size 4 Widget';
```

В инструкции ничего не говорится о том, имеется ли индекс для столбца DESCRIPTION или нет, и СУБД выполнит запрос в любом случае.

Если бы индекса для столбца DESCRIPTION не существовало, то СУБД была бы вынуждена выполнять запрос путем последовательного сканирования таблицы PRODUCTS, строка за строкой, просматривая в каждой строке столбец DESCRIPTION. Для получения гарантии того, что она нашла все строки, удовлетворяющие условию отбора, СУБД должна просмотреть *каждую* строку таблицы. Просмотр больших таблиц, содержащих миллионы строк, может занять минуты и даже часы.

Если для столбца DESCRIPTION имеется индекс, СУБД находит требуемые данные с гораздо меньшими усилиями. Она просматривает индекс, чтобы найти требуемое значение (изделие "Size 4 Widget"), а затем с помощью указателя находит требуемую строку (строки) таблицы. Поиск в индексе осуществляется достаточно быстро, так как индекс отсортирован и его строки очень короткие. Переход от индекса к строке (строкам) также происходит довольно быстро, поскольку в индексе содержится информация о том, где именно на диске располагается эта строка (строки).

Как видно из этого примера, преимущество индекса в том, что он в огромной степени ускоряет выполнение инструкций SQL с условиями отбора, имеющими ссылки на индексный столбец (столбцы). К недостаткам индекса относится то, что, во-первых, он занимает на диске дополнительное место и, во-вторых, индекс необходимо обновлять каждый раз, когда в таблицу добавляется строка или обновляется индексный столбец таблицы. Это требует дополнительных затрат на выполнение инструкций INSERT и UPDATE, которые обращаются к данной таблице.

В общем-то, полезно создавать индекс лишь для тех столбцов, которые часто используются в условиях отбора. Индексы удобны также в тех случаях, когда инструкции SELECT обращаются к таблице гораздо чаще, чем инструкции INSERT и UPDATE. Большинство СУБД *всегда* создает индекс для первичного ключа таблицы, так как ожидает, что доступ к таблице чаще всего будет осуществляться через первичный ключ. Кроме того, индекс первичного ключа помогает СУБД быстро найти дублирующиеся строки при вставке новых строк в таблицу.

Большинство СУБД также автоматически создает индекс для всех столбцов (или комбинаций столбцов), указанных в ограничении уникальности. Как и в случае первичного ключа, СУБД должна при каждой вставке новой строки или обновлении существующей проверять значение такого столбца — нет ли уже такого значения в таблице. Без индекса для такого столбца (столбцов) при проверке СУБД пришлось бы выполнять последовательное сканирование всех строк таблицы. При наличии индекса СУБД может просто воспользоваться индексом для поиска строки (если таковая существует) с интересующим нас значением, что гораздо быстрее последовательного поиска.

В учебной базе данных было бы уместно создать дополнительные индексы на основе следующих столбцов.

- COMPANY таблицы CUSTOMERS, если данные из таблицы часто извлекаются по названию компании.
- NAME таблицы SALESREPS, если данные о служащих часто извлекаются по имени служащего.
- REP таблицы ORDERS, если данные о заказах часто извлекаются с использованием имени служащего, принявшего их.
- CUST таблицы ORDERS, если данные о заказах часто извлекаются с использованием имени клиента, сделавшего их.
- MFR и PRODUCT таблицы ORDERS, если данные о заказах часто извлекаются по названию заказанного товара.

В стандарте SQL ничего не говорится об индексах и о том, как их создавать. Они относятся к "деталям реализации", выходящим за рамки ядра языка SQL. Тем не менее индексы весьма важны для обеспечения требуемой производительности любой серьезной базы данных уровня предприятия.

На практике в большинстве популярных СУБД (включая Oracle, Microsoft SQL Server, MySQL, Informix, Sybase и DB2) для создания индекса используется та или иная форма инструкции CREATE INDEX (рис. 13.7). В инструкции индексу назначается имя и указывается таблица, для которой он создается. Задается также индексируемый столбец (столбцы) и порядок его сортировки (по возрастанию или убыванию). Представленная на рис. 13.7 версия инструкции CREATE INDEX для СУБД DB2 — одна из наиболее простых. В ней можно использовать ключевое слово UNIQUE для указания того, что индексный столбец (столбцы) должен содержать уникальные значения в каждой строке таблицы.

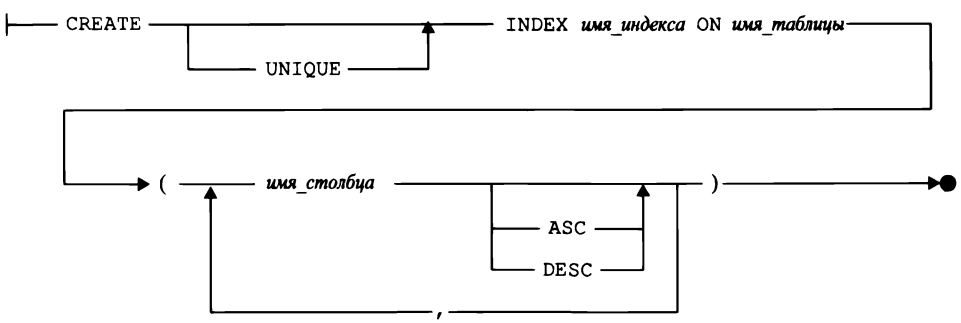


Рис. 13.7. Синтаксическая диаграмма базовой инструкции CREATE INDEX

Ниже дан пример инструкции CREATE INDEX, которая создает индекс для таблицы ORDERS на основе столбцов MFR и PRODUCT и содержит требование уникальности для комбинаций этих столбцов.

Создать индекс для таблицы OFFICES.

```
CREATE UNIQUE INDEX OFC_MGR_IDX
    ON OFFICES (MGR);
```

Создать индекс для таблицы ORDERS.

```
CREATE UNIQUE INDEX ORD_PROD_IDX
    ON ORDERS (MFR, PRODUCT);
```

В большинстве СУБД инструкция CREATE INDEX содержит дополнительные, специфические для каждой СУБД предложения, в которых задаются местоположение индекса на диске и рабочие параметры, такие как размер страниц индекса; сколько свободного пространства (в процентах) должен резервировать индекс для дополнительных строк; тип создаваемого индекса; должна ли выполняться кластеризация (т.е. должны ли записи на диске физически располагаться в том же порядке, что и в самом индексе) и т.д.

Некоторые СУБД поддерживают два или более различных типов индексов, оптимизированных для разных видов обращений к базе данных. Например, индекс *B-tree* использует древовидную структуру записей и блоков (групп записей) индекса для организации значений содержащихся в нем данных в возрастающем или убывающем порядке. Этот тип индекса (используемый по умолчанию почти во всех СУБД) обеспечивает эффективный поиск конкретного значения (или диапазона значений), требующийся при выполнении сравнения или проверке на принадлежность диапазону (BETWEEN).

Другой тип индекса, *хеш*-индекс, использует метод рандомизации для размещения всех возможных значений данных в небольшом количестве блоков индекса. Например, при наличии 10 миллионов возможных значений данных может оказаться разумным индекс с 500 хеш-блоками. Поскольку заданное значение всегда попадает в один и тот же блок, СУБД может выполнять поиск, находя соответствующий значению блок и выполняя поиск в нем. При наличии 500 блоков количество сканируемых при поиске элементов уменьшается в среднем в 500 раз. Это делает хеш-индексы очень быстрым средством при поиске точного значения. Однако распределение значений по блокам не сохраняет порядок данных, так что хеш-индекс нельзя использовать для работы с неравенствами или диапазонами.

Есть и иные типы индексов, подходящие для решения тех или иных задач базы данных, в том числе следующие.

- **T-tree** представляет собой вариацию индекса B-tree, оптимизированную для баз данных, работающих в оперативной памяти.
- **Битовая карта** полезна при относительно малом количестве возможных значений данных.
- **Индексная таблица** представляет собой относительно новую методику, когда в индексе хранится вся таблица целиком. Эта технология хорошо работает с таблицами с малым количеством столбцов, не входящих в первичный ключ, такими как таблицы поиска кодов, в которых обычно имеется только код (например, номер отдела) и описание (например, название этого отдела).

Если СУБД поддерживает различные типы индексов, инструкция CREATE INDEX не только определяет и создает индекс, но и указывает его тип.

Если вы создали для таблицы индекс, а позднее решили, что он не нужен, то можете удалить его из базы данных посредством инструкции DROP INDEX. Ниже приводится инструкция, которая удаляет индекс, созданный в предыдущем примере.

Удалить созданный ранее индекс.

```
DROP INDEX ORD_PROD_IDX;
```

Управление другими объектами базы данных

В SQL команды CREATE, DROP и ALTER образуют краеугольный камень языка определения данных (DDL). Инструкции с этими командами используются во всех реляционных СУБД для управления таблицами, индексами и представлениями (последние рассматриваются в главе 14, "Представления"). В большинстве популярных СУБД эти команды служат также для образования дополнительных инструкций DDL, которые создают, удаляют и модифицируют другие объекты базы данных, имеющиеся в конкретной СУБД.

Например, Sybase была первой СУБД, в которой появились триггеры и хранимые процедуры, трактуемые как объекты реляционной базы данных наряду с таблицами, утверждениями, индексами и прочими структурами. Специально для работы с этими объектами в диалект языка SQL в СУБД Sybase были внедрены дополнительные инструкции CREATE TRIGGER и CREATE PROCEDURE, а также соответствующие инструкции для удаления этих объектов. По мере роста популярности концепций триггеров и хранимых процедур, аналогичные инструкции стали появляться и в других СУБД.

Обычно такие инструкции подчиняются общим соглашениям — *а*) они используют ключевые слова CREATE/ALTER/DROP, *б*) следующее за ними слово указывает тип объекта, к которому они относятся, *в*) третье слово представляет собой имя объекта, которое должно отвечать соглашениям SQL об именовании. Все остальное в таких инструкциях у разных СУБД может быть различным (и нестандартным). Несмотря на это, такая общность придает ощущение унифицированности различных диалектов SQL. Как минимум, она говорит вам, в каком месте документации искать описание новой возможности. Если вы встретитесь с новой реляционной СУБД и обнаружите, что в ней имеется такой объект, как BLOB, то вполне вероятно, что в этой СУБД имеются также инструкции CREATE BLOB, DROP BLOB и ALTER BLOB. В табл. 13.1 показано, как используются команды CREATE, DROP и ALTER в расширенных диалектах DDL популярных СУБД. Стандарт SQL принял это соглашение для создания, удаления и изменения всех объектов в базе данных.

Таблица 13.1. Инструкции DDL в популярных СУБД

Инструкции DDL	Управляемый объект
В большинстве СУБД	
CREATE/DROP/ALTER TABLE	Таблица
CREATE/DROP/ALTER VIEW	Представление
CREATE/DROP/ALTER INDEX	Индекс

Продолжение табл. 13.1

Инструкции DDL	Управляемый объект
В стандарте ANSI/ISO	
CREATE/DROP ASSERTION	Утверждение
CREATE/DROP CHARACTER SET	Расширенный набор символов
CREATE/DROP COLLATION	Порядок сортировки набора символов
CREATE/DROP/ALTER DOMAIN	Домен
CREATE/DROP SCHEMA	Схема базы данных
CREATE/DROP TRANSLATION	Преобразование наборов символов
В DB2	
CREATE/DROP ALIAS	Псевдоним таблицы или представления
CREATE/DROP/ALTER BUFFERPOOL	Коллекция буферов ввода-вывода
CREATE/DROP DISTINCT TYPE	Пользовательский тип данных (DISTINCT)
CREATE/DROP FUNCTION	Пользовательская функция
CREATE/DROP/ALTER NODEGROUP	Группа разделов или узлов базы данных
DROP PACKAGE	Программный модуль доступа
CREATE/DROP PROCEDURE	Пользовательская хранимая процедура
CREATE/DROP SCHEMA	Схема базы данных
CREATE/DROP/ALTER TABLESPACE	Табличное пространство
CREATE/DROP TRIGGER	Триггер
В Informix	
CREATE/DROP CAST	Правило преобразования типов данных
CREATE/DROP DATABASE	Именованная база данных Informix
CREATE/DROP DISTINCT TYPE	Пользовательский тип данных (DISTINCT)
CREATE/DROP FUNCTION	Пользовательская функция
CREATE/DROP OPAQUE TYPE	Пользовательский тип данных (категория OPAQUE)
CREATE/DROP OPCLASS	Пользовательский метод доступа к дисковому хранилищу
CREATE/DROP PROCEDURE	Пользовательская хранимая процедура
CREATE/DROP ROLE	Пользовательская роль в базе данных
CREATE/DROP ROUTINE	Пользовательская хранимая процедура
CREATE/DROP ROW TYPE	Именованный тип строки (объектное расширение)
CREATE SCHEMA	Схема базы данных
CREATE/DROP SYNONYM	Псевдоним таблицы или представления
CREATE/DROP TRIGGER	Триггер
В Microsoft SQL Server	
CREATE/DROP/ALTER DATABASE	База данных
CREATE/DROP DEFAULT	Значение столбца по умолчанию (не рекомендовано начиная с SQL Server 2005)
CREATE/DROP/ALTER FULLTEXT CATALOG	Каталог текстового поиска
CREATE/DROP/ALTER FULLTEXT INDEX	Индекс текстового поиска
CREATE/DROP/ALTER FUNCTION	Функция

Продолжение табл. 13.1

Инструкции DDL	Управляемый объект
CREATE/DROP/ALTER LOGIN	Регистрационное имя
CREATE/DROP/ALTER PROCEDURE	Хранимая процедура
CREATE/DROP/ALTER ROLE	Роль
CREATE/DROP RULE	Правило соблюдения целостности столбца
CREATE SCHEMA	Схема базы данных
CREATE/DROP SYNONYM	Синоним (псевдоним)
CREATE/DROP/ALTER TRIGGER	Хранимый триггер
CREATE/DROP TYPE	Тип
CREATE/DROP/ALTER USER	Учетная запись
CREATE/DROP/ALTER XML	Схема XML
В Oracle	
CREATE/DROP CLUSTER	Табличный кластер
CREATE/DROP/ALTER DATABASE	Именованная база данных Oracle
CREATE/DROP DATABASE LINK	Сетевое подключение к удаленной базе данных
CREATE/DROP DIRECTORY	Каталог для хранения больших объектов
CREATE/DROP/ALTER FUNCTION	Пользовательская функция
CREATE/DROP LIBRARY	Библиотека внешних функций, вызываемых посредством PL/SQL
CREATE/DROP/ALTER MATERIALIZED VIEW	Представление, физически хранящее результаты запроса
CREATE/DROP/ALTER PACKAGE	Группа процедур PL/SQL, доступных для коллективного использования
CREATE/DROP PACKAGE BODY	Содержимое пакета
CREATE/DROP/ALTER PROCEDURE	Пользовательская хранимая процедура
CREATE/DROP/ALTER PROFILE	Набор ограничений на использование ресурсов базы данных
CREATE/DROP/ALTER ROLE	Роль пользователя в базе данных
CREATE/DROP/ALTER ROLLBACK SEGMENT	Область дисковой памяти, используемая для восстановления базы данных при отмене транзакции
CREATE SCHEMA	Схема базы данных
CREATE/DROP/ALTER SEQUENCE	Пользовательская последовательность значений
CREATE/DROP/ALTER SNAPSHOT	Таблица результатов запроса, доступная только для чтения
CREATE/DROP SYNONYM	Псевдоним таблицы или представления
CREATE/DROP/ALTER TABLESPACE	Табличная область
CREATE/DROP/ALTER TRIGGER	Триггер
CREATE/DROP TYPE	Пользовательский абстрактный тип данных
CREATE/DROP TYPE BODY	Методы абстрактного типа данных
CREATE/DROP/ALTER USER	Идентификатор пользователя Oracle
В Sybase	
CREATE/DROP/ALTER DATABASE	База данных

Окончание табл. 13.1

Инструкции DDL	Управляемый объект
CREATE/DROP DEFAULT	Значение столбца по умолчанию
CREATE EXISTING TABLE	Локальная копия таблицы, к которой осуществляется удаленный доступ
CREATE/DROP PROCEDURE	Хранимая процедура
CREATE/DROP/ALTER ROLE	Роль пользователя в базе данных
CREATE/DROP RULE	Правило соблюдения целостности столбца
CREATE SCHEMA	Схема базы данных
CREATE/DROP TRIGGER	Триггер
В MySQL	
CREATE/ALTER/DROP DATABASE	База данных
CREATE/ALTER/DROP FUNCTION	Функция MySQL
CREATE/ALTER/DROP PROCEDURE	Хранимая процедура MySQL
CREATE/DROP TRIGGER	Хранимый триггер
CREATE/ALTER/DROP SCHEMA	Схема базы данных
CREATE/DROP USER	Учетная запись пользователя MySQL

Структура базы данных

Стандарт SQL1 определяет простую структуру содержимого базы данных, показанную на рис. 13.8. У каждого пользователя есть принадлежащая ему совокупность таблиц. Такая структура используется практически во всех основных СУБД, хотя в некоторых из них (например, в узкоспециализированных или настольных) отсутствует понятие принадлежности таблиц тем или иным пользователям. В таких СУБД все таблицы базы данных входят в один большой набор таблиц.

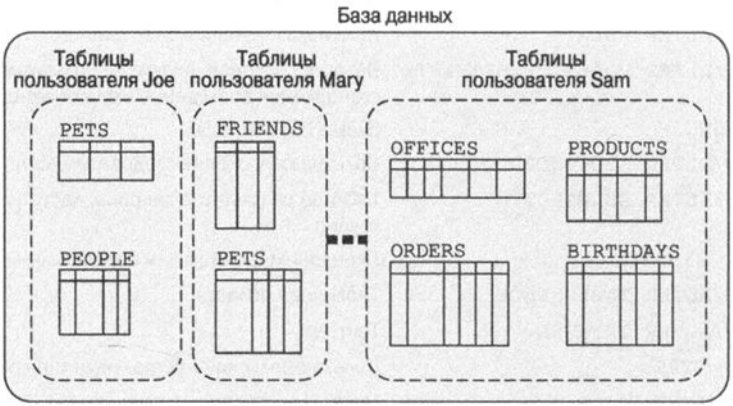


Рис. 13.8. Организация базы данных согласно стандарту SQL1

Несмотря на то что в различных СУБД используется одинаковая логическая структура отдельной базы данных, наблюдается большое разнообразие в том, как организована и структурирована вся совокупность баз данных, управляемая той

или иной СУБД в конкретной операционной системе. В одних СУБД все данные хранятся в одной общей базе данных, в других — в нескольких базах данных и каждой из них присваивается имя. В третьих СУБД информация хранится в нескольких базах данных, организованных в виде системы каталогов.

Это разнообразие не оказывает какого-либо влияния на структуру языка SQL, используемого для доступа к информации, хранимой в базе данных. Однако оно влияет на способ организации данных; например, данные по обработке заказов и бухгалтерские данные можно хранить в одной базе данных или в разных. Оно влияет также на способ первоначального обращения к базе данных. Если, к примеру, имеется несколько баз данных, необходимо сообщить СУБД, с какой из них вы хотите работать. Чтобы проиллюстрировать, как в различных СУБД решаются все эти вопросы, предположим, что учебная база данных расширена и, в дополнение к данным для программы обработки заказов, содержит данные для программ начисления зарплаты и бухгалтерского учета.

Архитектура с одной базой данных

На рис. 13.9 представлена архитектура, при которой СУБД поддерживает единую базу данных в системе. Базы данных для мэйнфреймов и мини-компьютеров (например, версия DB2 для мэйнфреймов) тяготеют к использованию именно этой архитектуры. (Заметим, что версия DB2 для мэйнфреймов кардинально отличается от DB2 UDB, которая работает под управлением Linux, Unix и Windows.) Данные по бухгалтерскому учету, зарплате и заказам хранятся в таблицах внутри одной базы данных. Главные таблицы каждой программы собраны вместе и принадлежат одному пользователю, который, вероятно, является ответственным за эту программу на данном компьютере.

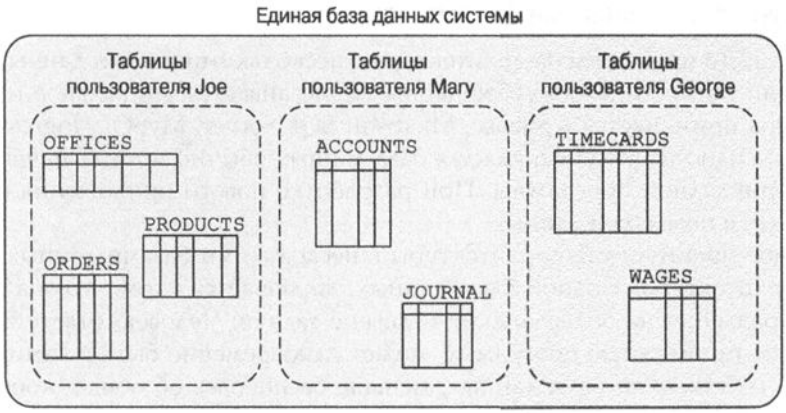


Рис. 13.9. Архитектура с одной базой данных

Преимущество такой архитектуры заключается в том, что таблицы из разных приложений могут легко обращаться друг к другу. Например, таблица `TIMECARDS` из программы начисления зарплаты может содержать внешний ключ к таблице `OFFICES`, и другие программы могут использовать это отношение для вычисления

комиссионных от сделок. Имея соответствующие права, пользователи могут выполнять запросы, объединяющие данные из разных приложений.

Недостатком такой архитектуры является то, что со временем, по мере добавления новых прикладных программ, база данных приобретает огромные размеры. В базе данных DB2 для мэйнфреймов не редкость базы данных, имеющие несколько сотен таблиц. Проблемы, связанные с управлением базой данных таких размеров, очевидны — резервное копирование, восстановление данных, анализ производительности и так далее требуют, как правило, постоянного внимания администратора базы данных.

При архитектуре с одной базой данных доступ к ней осуществляется очень просто, так как база данных одна и не требуется делать какой-либо выбор. В этой архитектуре база данных обычно связана с единственной работающей копией программного обеспечения СУБД, так что пользователь просто подключается к базе данных. DB2 для мэйнфреймов часто запускает две отдельные базы данных — для работы и для тестирования. Однако все производственные данные при этом хранятся только в одной базе данных.

Oracle применяет архитектуру с одной базой данных, при использовании которой каждая база данных связана с единственной копией программного обеспечения СУБД. Обычно базы данных Oracle организуются в соответствии с их назначением или основными функциями, так что на одном сервере может работать несколько баз данных. Для подключения Oracle предоставляет команду `CONNECT`, которая требует комбинации пользовательской учетной записи и идентификатора базы данных. Имеется несколько методов идентификации и локализации базы данных в процессе подключения, и в большинстве случаев пользователю просто не надо знать, где именно в сети располагается интересующая его база данных.

Архитектура с несколькими базами данных

На рис. 13.10 представлена архитектура с несколькими базами данных, при использовании которой каждой базе данных присваивается уникальное имя. Такая архитектура применяется в Sybase, Microsoft SQL Server, MySQL, Ingres и других СУБД. Как видно из рисунка, каждая база данных обычно предназначена для отдельной прикладной программы. При разработке нового приложения создается, как правило, и новая база данных.

Основное преимущество архитектуры с несколькими базами данных, по сравнению с архитектурой с одной базой данных, заключается в том, что она разделяет задачу управления на более мелкие и легкие задачи. Человек, ответственный за конкретную прикладную программу, может одновременно быть и администратором своей собственной базы данных, меньше беспокоясь об общей координации баз данных. Если у него появится необходимость добавить новое приложение, он сможет сделать это в своей базе данных, не нарушая работу других баз данных. Кроме того, пользователям и программистам легче запомнить общую структуру своей собственной базы данных.

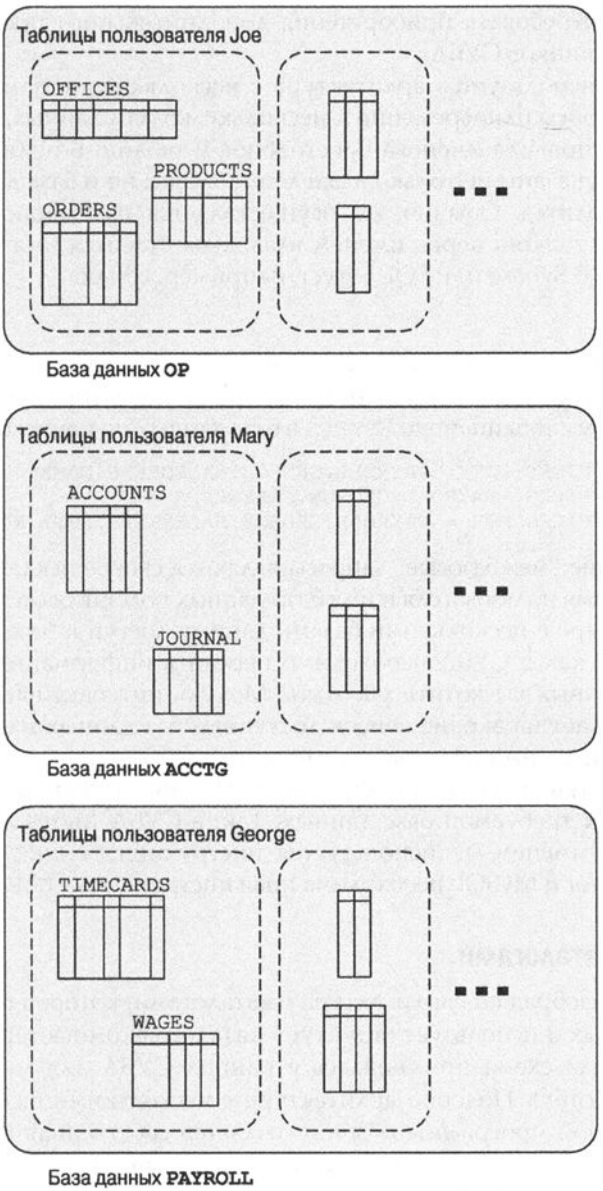


Рис. 13.10. Архитектура с несколькими базами данных

Главным недостатком архитектуры с несколькими базами данных является то, что индивидуальные базы данных могут стать островами информации, не связанными друг с другом. Как правило, таблица одной базы данных не может содержать внешний ключ для таблицы другой базы данных. Часто СУБД не могут выполнять запросы, выходящие за границы базы данных, не позволяя тем самым связывать данные из двух приложений. Возможность выполнения запросов одновременно к нескольким базам данных может быть сопряжена со значительными

затратами или потребовать приобретения дополнительного программного обеспечения от поставщиков СУБД.

Если в СУБД используется архитектура с несколькими базами данных и поддерживаются запросы одновременно к нескольким базам данных, то в ней должны быть расширены правила именования столбцов и таблиц. В полном имени таблицы должны быть указаны не только владелец таблицы, но и база данных, в которой эта таблица находится. Обычно это осуществляется путем расширения записи имени таблицы с точкой: перед именем владельца ставится имя базы данных, отделяемое точкой. В Sybase или SQL Server, например, ссылка

```
OP.JOE.OFFICES
```

относится к таблице OFFICES, принадлежащей пользователю JOE в базе данных с именем OP (используется для обработки заказов), а приведенный далее запрос объединяет эту таблицу с таблицей SALESREPS в базе данных по начислению зарплаты.

```
SELECT OP.JOE.OFFICES.CITY, PAYROLL.GEORGE.SALESREPS.NAME
  FROM OP.JOE.OFFICES, PAYROLL.GEORGE.SALESREPS
 WHERE OP.JOE.OFFICES.MGR = PAYROLL.GEORGE.SALESREPS.EMPL_NUM;
```

К счастью, такие "межбазовые" запросы являются скорее исключением, чем правилом, и обычно имя пользователя и имя базы данных подставляются по умолчанию.

При архитектуре с несколькими базами данных доступ к базе данных немного усложняется, так как в СУБД необходимо передать информацию о том, с какой именно базой данных вы хотите работать. Модуль интерактивных запросов, как правило, отображает на экране список доступных баз данных или просит вас ввести имя базы данных, имя пользователя и пароль. Для обеспечения программного доступа СУБД обычно расширяет встроенный SQL инструкцией, которая подключает программу к требуемой базе данных. Так, в СУБД Ingres для подключения к базе данных с именем OP используется инструкция CONNECT 'OP'.В Sybase, Microsoft SQL Server и MySQL необходима иная инструкция — USE 'OP'.

Архитектура с каталогами

На рис. 13.11 изображена архитектура с каталогами, которая поддерживает несколько баз данных и использует структуру каталогов компьютера для их организации. Такого рода схема применялась в ранних СУБД для мини-компьютеров (Rdb/VMS и Informix). Подобно архитектуре с несколькими базами данных, для каждой прикладной программы обычно создается собственная база данных. Как видно из рисунка, каждая база данных имеет имя, причем две различные базы данных в разных каталогах могут иметь одно и то же имя.

Главное преимущество архитектуры с каталогами — ее гибкость. Такая архитектура особенно полезна для инженерных и конструкторских программ. Обычно с ними работает много опытных пользователей, которым может потребоваться несколько баз данных для структурированного хранения своей информации. Недостатки у архитектуры с каталогами те же самые, что и у архитектуры с несколькими базами. Кроме того, СУБД не имеет, как правило, информации обо всех созданных базах данных, которые могут быть распределены по всей структуре каталогов. Нет и главной базы данных, отслеживающей все остальные базы данных, что затрудняет централизованное администрирование.

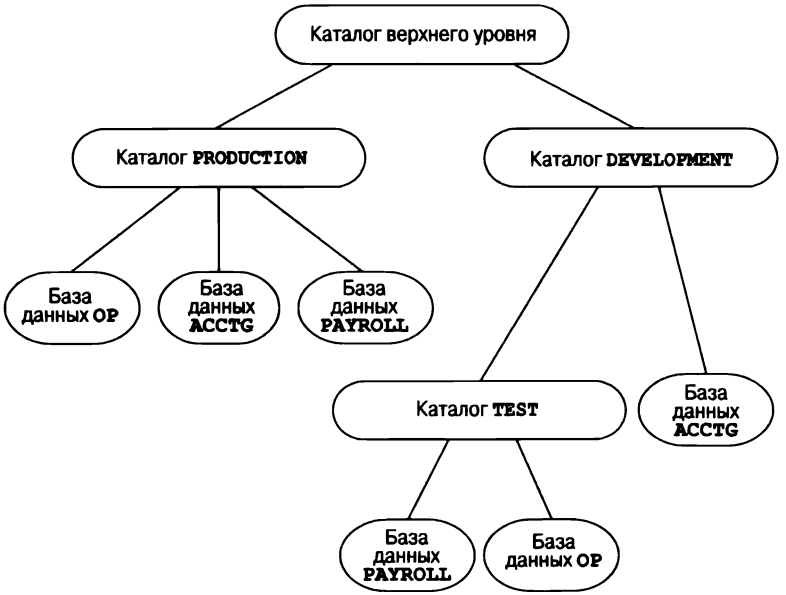


Рис. 13.11. Архитектура с каталогами

Архитектура с каталогами еще больше усложняет доступ к базе данных, поскольку для этого необходимо указать не только имя базы данных, но и ее местоположение в иерархии каталогов. Для доступа к базе данных Rdb/VMS применяется инструкция DECLARE DATABASE диалекта VAX SQL. Например, ниже приведена инструкция DECLARE DATABASE для подключения к базе данных с именем OP, которая находится в каталоге VAX/VMS с именем SYS$ROOT:[DEVELOPMENT.TEST].

```
DECLARE DATABASE
        FILENAME 'SYS$ROOT:[DEVELOPMENT.TEST]OP';
```

Если база данных находится в текущем каталоге пользователя (что бывает достаточно часто), то инструкция упрощается.

```
DECLARE DATABASE
        FILENAME 'OP';
```

Некоторые СУБД с архитектурой с каталогами, даже если и не могут выполнить запросы, выходящие за пределы одной базы данных, то во всяком случае обеспечивают доступ к нескольким базам данных одновременно. Для различения большого количества баз данных применяется самая обычная методика "сверхполного" имени таблицы. Так как две базы данных в разных каталогах могут иметь одинаковые имена, для устранения неоднозначности приходится также вводить *псевдоним базы данных*. Ниже приведены две инструкции VAX SQL, открывающие в СУБД Rdb/VMS две разные базы данных с одинаковыми именами.

```
DECLARE DATABASE OP1
        FILENAME 'SYS$ROOT:[PRODUCTION\]OP';
```

```
DECLARE DATABASE OP2
        FILENAME 'SYS$ROOT:[DEVELOPMENT.TEST]OP';
```

Они присваивают двум базам данных псевдонимы OP1 и OP2, которые используются в последующих инструкциях VAX SQL для указания полного имени таблицы.

Как видно из сказанного выше, существует огромное разнообразие в том, как в различных СУБД организованы базы данных и каким образом к ним осуществляется доступ. Эта область языка SQL — одна из самых нестандартных и часто является самой первой, с которой сталкивается пользователь, впервые приступающий к работе с базой данных. Такая несогласованность между различными СУБД делает невозможным простой перенос программ с одной СУБД на другую; правда, процесс изменения программ является не сложным, а, скорее, монотонным.

Базы данных на нескольких серверах

С развитием серверов баз данных и локальных сетей понятие расположения базы данных естественным образом расширяется до понятия физического сервера базы данных. На практике большинство СУБД используют архитектуру с несколькими базами данных на физическом сервере. На наивысшем уровне база данных связана с именованным сервером в сети. В пределах сервера может быть несколько именованных баз данных. Отображение имен серверов на их физические местоположения обрабатывается программным обеспечением сети, а отображение имен баз данных на физические файлы или файловые системы на серверах — программным обеспечением СУБД.

Структура базы данных и стандарт ANSI/ISO

В стандарте ANSI/ISO SQL1 две части языка SQL — язык обработки данных (DML) и язык определения данных (DDL) — были четко отделены друг от друга. Они рассматриваются как два отдельных, относительно независимых языка. Стандарт не требует, чтобы СУБД воспринимала инструкции DDL в процессе своей работы. Одно из преимуществ такого разделения DML и DDL состояло в том, что стандарт допускал наличие статической структуры базы данных, наподобие старых иерархических и сетевых СУБД, как показано на рис. 13.12.

Структура базы данных, задаваемая стандартом SQL1, довольно проста. Наборы таблиц определяются в *схеме базы данных*, связанной с конкретным пользователем. У небольшой базы данных, изображенной на рис. 13.12, имеется две схемы. Одна из них связана с пользователем по имени Джо (по распространенной терминологии, она *принадлежит* Джо), а вторая принадлежит Мери. Схема Джо содержит две таблицы с именами PEOPLE и PLACES. Схема Мери содержит две другие таблицы — THINGS и PLACES. Хотя в базе данных есть две таблицы с именем PLACES, они различимы, поскольку у них разные владельцы.

Начиная с SQL2 значительно расширен смысл определения базы данных и ее схемы. Как уже упоминалось, стандарт SQL требует, чтобы инструкции DDL мог-

ли выполняться как в интерактивном режиме, так и из программного SQL. Тогда структуру базы данных можно будет менять и после ее создания. Кроме того, стандарт SQL2 значительно расширил и концепцию пользователей базы данных (в стандарте им назначен термин *идентификаторы авторизации*). На рис. 13.13 представлена высокоуровневая структура базы данных согласно текущей версии стандарта SQL.

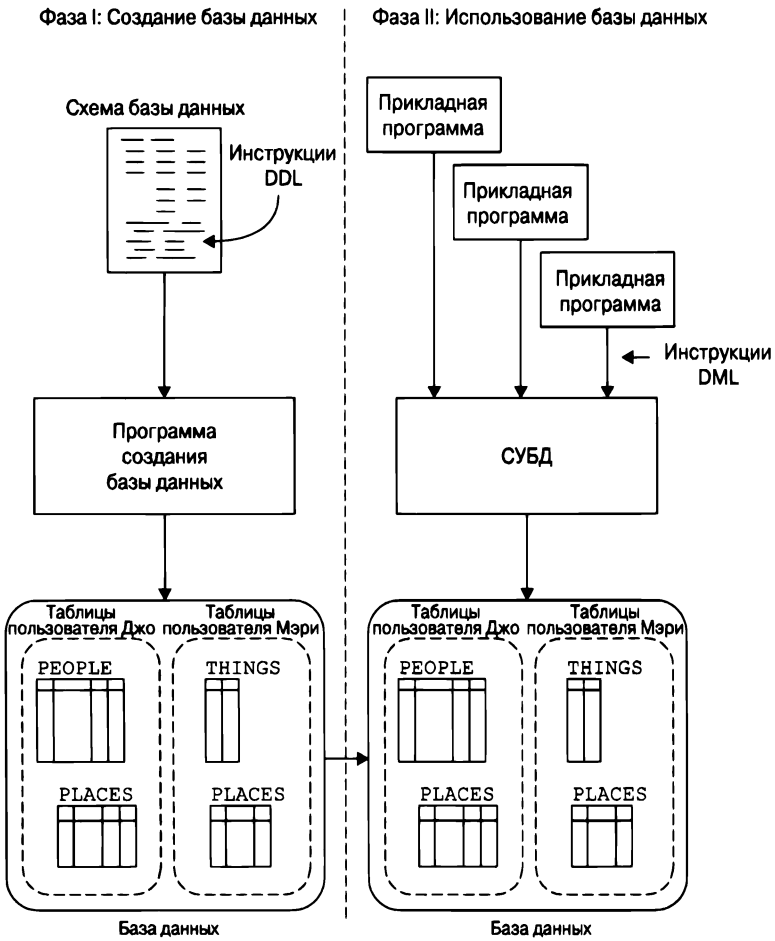


Рис. 13.12. СУБД со статическим DDL

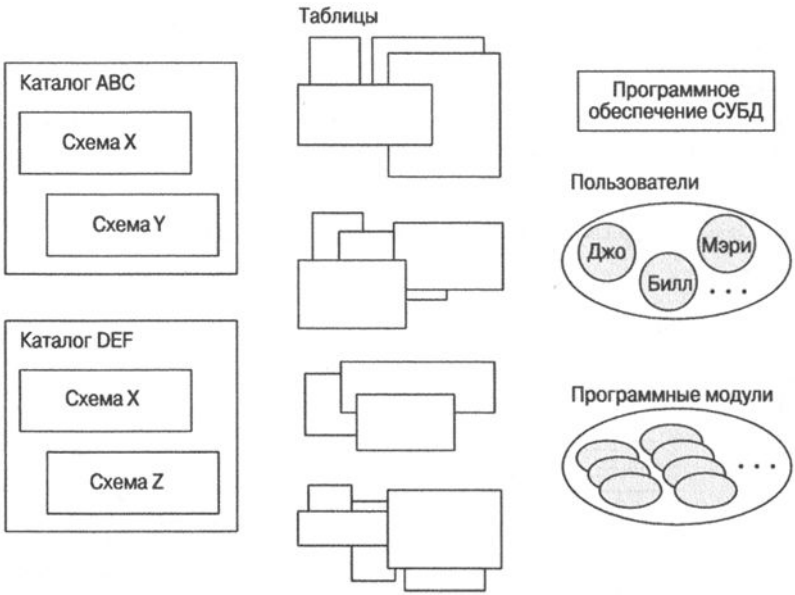


Рис. 13.13. Структура базы данных в стандарте SQL

Структура базы данных верхнего уровня в стандарте SQL именуется *среда SQL*. Это концептуальная совокупность сущностей базы данных, связанных с реализацией СУБД, отвечающей стандарту SQL. Стандарт не описывает способ создания среды SQL: он зависит от конкретной СУБД. В стандарте перечислены следующие ее компоненты.

- **Программное обеспечение СУБД**, соответствующее стандарту SQL.
- **Именованные пользователи** (названные в стандарте идентификаторами авторизации), имеющие привилегии для выполнения определенных операций над данными и структурой базы данных.
- **Программные модули**, применяемые для доступа к базе данных. Стандарт SQL описывает выполнение инструкций SQL в терминах языка модулей, который большинством коммерческих СУБД не используется. Независимо от того, как реально создаются SQL-программы, стандарт SQL гласит, что концептуально программный код для доступа к базе данных входит в состав SQL-среды.
- **Каталоги**, описывающие структуру базы данных. Схемы баз данных стандарта SQL1 содержатся в этих каталогах.
- **Данные базы данных**, управляемые программным обеспечением СУБД и доступные пользователям через прикладные программы. Структура данных описана в каталогах. Хотя концептуально стандарт описывает данные как располагающиеся вне структуры каталога, распространенным является представление о данных как о содержимом таблиц, входящих в схему, в свою очередь входящую в каталог.

Каталоги

В среде SQL структура базы данных определяется одним или несколькими именованными *каталогами*. Слово "каталог" в данном контексте употребляется в том же смысле, в каком оно исторически использовалось в системах на базе мэйнфреймов, — для описания наборов объектов (как правило, файлов). В мини-компьютерах и персональных компьютерах эта концепция приближенно аналогична каталогу. В случае стандартной базы данных SQL каталог представляет собой коллекцию именованных схем. Каталог содержит также системные таблицы (часто некорректно именуемые системным каталогом), которые описывают структуру базы данных. Таким образом, каталог представляет собой самодокументируемую сущность в базе данных. Эта характеристика каталогов (поддерживаемая во всех основных реализациях SQL) детально описывается в главе 16, "Системный каталог".

Стандарт SQL описывает роль каталога и указывает, что среда SQL может содержать один или более (на самом деле их может быть нуль или более) каталогов, каждый из которых должен иметь уникальное имя. В стандарте сказано, что механизм создания и уничтожения каталогов определяется конкретной реализацией СУБД. Стандарт также гласит, что конкретной реализацией определяется и возможность СУБД выполнять обращение к данным между каталогами. В частности, это относится к тому, может ли одна инструкция SQL получить доступ к данным из нескольких каталогов, может ли одна SQL-транзакция работать с несколькими каталогами и может ли пользователь в рамках своего сеанса с СУБД пересечь границы каталога.

В стандарте сказано, что когда пользователь или программа впервые устанавливает контакт со средой SQL, один из ее каталогов определяется как используемый данным сеансом по умолчанию. (Способ выбора этого каталога опять же определяется реализацией.) В течение сеанса стандартный каталог может быть изменен с помощью инструкции `SET CATALOG`.

Схемы

Схема — это ключевой высокоуровневый контейнер объектов в структуре базы данных SQL. Схема представляет собой именованную сущность в базе данных и включает определения следующих объектов.

- **Таблицы**. Вместе со связанными с ними структурами (столбцами, первичными и внешними ключами, ограничениями и т.п.), таблицы остаются базовыми строительными блоками базы данных в схеме.
- **Представления**. Представления — это виртуальные таблицы, создаваемые на основе реальных таблиц, определенных в схеме (о представлениях рассказывается в главе 14, "Представления").
- **Домены**. Домены работают как расширенные типы данных для определения столбцов в таблицах схемы, как описано в главе 11, "Целостность данных".

- **Утверждения**. Эти ограничения целостности базы данных ограничивают возможные сочетания данных из разных таблиц схемы, как было описано ранее в данной главе.
- **Привилегии**. Привилегии базы данных управляют возможностями, предоставляемыми пользователям для доступа к данным и их обновлению, а также для модификации структуры базы данных. Схема безопасности SQL, создаваемая этими привилегиями, описана в главе 14, "Представления".
- **Наборы символов**. Базы данных поддерживают работу с разными языками и управляют представлением нелатинских символов в этих языках (например, диакритические или кириллические символы или двухбайтовое представление иероглифов многих азиатских языков) посредством наборов символов, определяемых схемой.
- **Порядки сортировки**. Они работают рука об руку с наборами символов и определяют правила сортировки текста для различных наборов символов.
- **Правила конвертирования текста**. Эти правила управляют преобразованием текста из одного набора символов в другой и сравнением текстовых данных, использующих различные наборы символов.

Схема создается с помощью инструкции CREATE SCHEMA, синтаксическая диаграмма которой изображена на рис. 13.14. Вот как выглядит определение простой двухтабличной схемы для пользователя Джо, показанной на рис. 13.12.

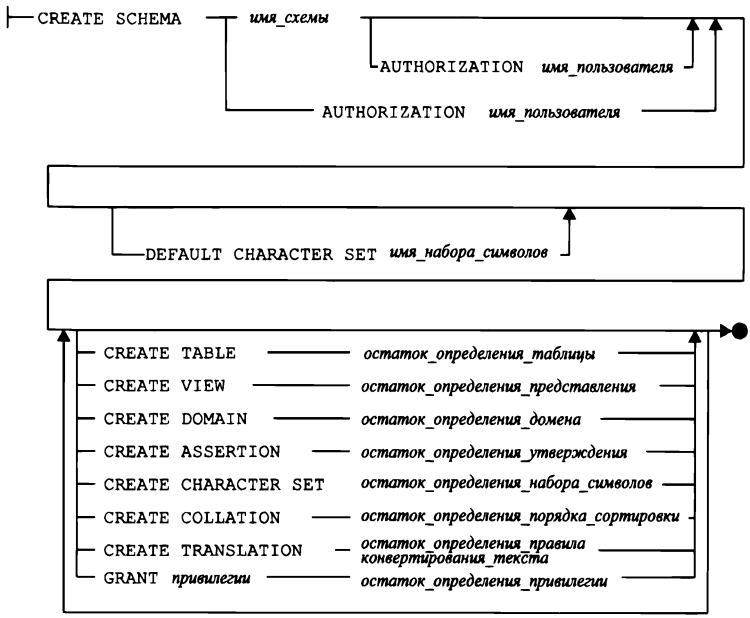


Рис. 13.14. Синтаксическая диаграмма инструкции CREATE SCHEMA

```
CREATE SCHEMA JSCHEMA AUTHORIZATION JOE
 CREATE TABLE PEOPLE
        (NAME VARCHAR(30),
          AGE INTEGER)
 CREATE TABLE PLACES
        (CITY VARCHAR(30),
        STATE VARCHAR(30))
        GRANT ALL PRIVILEGES
           ON PEOPLE
           TO PUBLIC
        GRANT SELECT
           ON PLACES
           TO MARY;
```

Эта схема определяет две таблицы и предоставляет права доступа к ним для других пользователей. Она не содержит определений никаких дополнительных структур, таких как представления или утверждения. Обратите внимание на то, что инструкции CREATE TABLE, входящие в инструкцию CREATE SCHEMA, — это полноценные SQL-инструкции создания таблиц, описанные ранее в этой главе. Если ввести их в интерактивном режиме, то, согласно стандарту, СУБД создаст указанные таблицы в *схеме по умолчанию* текущего сеанса.

Следует также отметить, что структура схемы связана с конкретным пользователем, но не зависит от него. Один пользователь может быть владельцем нескольких различных схем. Для сохранения совместимости со стандартом SQL1 текущий стандарт SQL позволяет создать схему со следующими параметрами:

- **имя схемы и имя пользователя** (как в последнем примере);
- **только имя схемы**; в этом случае пользователь, выполнивший инструкцию CREATE SCHEMA, автоматически становится ее владельцем;
- **только имя пользователя**. В этом случае именем схемы становится имя пользователя. Это соответствует стандарту SQL1 и практике многих коммерческих СУБД, не допускающих создания нескольких схем для одного пользователя.

Схема, которая вам больше не нужна, может быть удалена с помощью инструкции DROP SCHEMA (рис. 13.15). В этой инструкции необходимо указать одно из правил удаления, о которых мы уже говорили при обсуждении удаления столбцов, — либо CASCADE, либо RESTRICT. Если указать правило CASCADE, тогда все структуры в определении схемы (таблицы, представления, утверждения и т.п.) будут автоматически удалены. Если же выбрать правило RESTRICT, то инструкция не будет выполнена при наличии в схеме хотя бы одной из этих структур. Таким образом, правило RESTRICT требует, чтобы перед удалением схемы из нее были удалены все вложенные структуры; это защищает вас от случайного удаления нужных данных или определений. Стандарт SQL не предлагает специальной инструкции ALTER SCHEMA. Для изменения схемы используются обычные инструкции изменения входящих в нее объектов, как, например, ALTER TABLE.

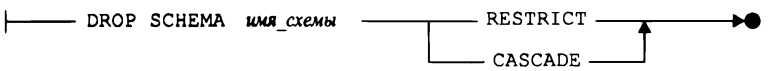


Рис. 13.15. Синтаксическая диаграмма инструкции DROP SCHEMA

Когда пользователь (или программа) обращается к базе данных, одна из ее схем идентифицируется как *схема по умолчанию*. Именно к ней неявно обращается любая инструкция DDL, выполняемая для создания, удаления или изменения элементов схемы. Можно сказать, что именем этой схемы неявно квалифицируются имена всех таблиц, указываемых в SQL-инструкциях. Однако, как уже упоминалось в главе 5, "Основы SQL", можно использовать квалифицированные имена таблиц, чтобы работать и с другими схемами базы данных, если только у вас есть соответствующие привилегии. Согласно стандарту SQL, имя таблицы квалифицируется именем схемы. Так, если набор таблиц нашей учебной базы данных был создан как часть схемы SALES, то квалифицированное имя таблицы OFFICES — SALES.OFFICES.

Если схема была создана с указанием только имени пользователя, то имя таблицы строится так, как описано в главе 5, "Основы SQL". Имя пользователя становится именем схемы и указывается перед именем таблицы через точку.

У инструкции CREATE SCHEMA есть еще одно неочевидное преимущество. Как вы помните из описания инструкции CREATE TABLE, создавать ссылочные циклы (две или более таблиц, ссылающихся друг на друга, используя отношения первичного и внешнего ключа) достаточно сложно. Сначала создается первая таблица без определения внешнего ключа, затем, после того как будет создана другая таблица (или таблицы), определение внешнего ключа добавляется посредством инструкции ALTER TABLE. В инструкции CREATE SCHEMA эта проблема решена, так как до тех пор, пока не будут созданы *все* таблицы, СУБД не проверяет определяемые схемой условия ссылочной целостности. На практике для создания нового набора связанных между собой таблиц обычно используется инструкция CREATE SCHEMA. Для последующего добавления, удаления и модификации отдельных таблиц применяются инструкции CREATE/DROP/ALTER TABLE.

Во многие известные СУБД уже включены некоторые формы инструкции CREATE SCHEMA, хотя между СУБД разных производителей имеются существенные различия. Например, в Oracle инструкция CREATE SCHEMA позволяет создавать только таблицы, представления и привилегии (но не другие структуры SQL) и, кроме того, требует, чтобы имя схемы совпадало с именем пользователя. Аналогично работает эта инструкция и в Informix Universal Server, только набор структур, которые могут входить в определение схемы, дополнен индексами, триггерами и синонимами (псевдонимами). Подобные возможности предоставляет и Sybase. Во всех трех СУБД предлагаемые возможности соответствуют начальному уровню совместимости со стандартом SQL.

Резюме

В данной главе рассматривался язык определения данных (DDL), с помощью которого формируется структура базы данных.

- Инструкция CREATE TABLE создает таблицу, полностью определяя ее столбцы, первичный ключ и внешние ключи. Она также может применяться для определения первичного ключа, внешнего ключа и ограничений CHECK.
- Инструкция DROP TABLE удаляет из базы данных ранее созданную таблицу.
- Инструкцию ALTER TABLE можно использовать для добавления, удаления или изменения столбца в существующей таблице.
- Инструкции CREATE INDEX и DROP INDEX соответственно создают и удаляют индексы, которые ускоряют выполнение запросов к базе данных, но увеличивают затраты на ее обновление.
- В большинстве СУБД имеются дополнительные инструкции вида CREATE, DROP и ALTER, которые выполняют соответствующие действия над различными объектами, входящими в состав той или иной СУБД.
- Стандарт SQL определяет понятие схемы базы данных, содержащей набор таблиц и управляемой инструкциями CREATE SCHEMA и DROP SCHEMA.
- В различных СУБД используются самые разные подходы к организации одной или нескольких управляемых ими баз данных. Эти различия влияют на структуру баз данных и на доступ к ним.

14
ГЛАВА

Представления

Таблицы определяют структуру базы данных и организацию информации в ней. Однако SQL с помощью представлений позволяет взглянуть на данные под другим углом. *Представлением* называется SQL-запрос на выборку, которому присвоили имя и который затем сохранили в базе данных. Представление позволяет пользователю увидеть результаты сохраненного запроса, а SQL обеспечивает доступ к этим результатам таким образом, как если бы они были реальной таблицей базы данных.

Представления используются по нескольким причинам:

- они позволяют сделать так, что разные пользователи базы данных будут видеть ее по-разному;
- с их помощью можно ограничить доступ к данным, разрешая пользователям видеть только некоторые из строк и столбцов таблицы;
- они упрощают доступ к базе данных, показывая каждому пользователю структуру хранимых данных в наиболее подходящем для него виде.

В настоящей главе рассказывается о том, как создавать представления и применять их для упрощения работы с базой данных и повышения степени ее безопасности.

Что такое представление

Как видно из рис. 14.1, представление является виртуальной таблицей, содержимое которой определяется запросом. Для пользователя базы данных представление выглядит как реальная таблица, состоящая из строк и столбцов. Однако, в отличие от таблицы, представление как совокупность значений в базе данных реально не существует. Строки и столбцы данных, которые пользователь видит с помощью представления, являются результатами запроса, лежащего в его основе. SQL создает иллюзию представления, присваивая ему имя, как таблице, и сохраняя его определение в базе данных.

Таблица **SALESREPS**

EMPL_NUM	NAME	AGE	QUOTA	SALES
105	Bill Adams	37	$350,000.00	$367,911.00
109	Mary Jones	31	$300,000.00	$392,725.00
102	Sue Smith	48	$350,000.00	$474,050.00
106	Sam Clark	52	$275,000.00	$299,912.00
104	Bob Smith	33	$200,000.00	$142,594.00
101	Dan Roberts	45	$300,000.00	$305,673.00
110	Tom Snyder	41	NULL	$75,985.00
108	Larry Fitch	62	$350,000.00	$361,865.00

Представление **REPDATA**

NAME	CITY	REGION	QUOTA	SALES
Mary Jones	New York	Eastern	$300,000.00	$392,725.00
Sam Clark	New York	Eastern	$275,000.00	$299,912.00
Bob Smith	Chicago	Eastern	$200,000.00	$142,594.00
Paul Cruz	Chicago	Eastern	$275,000.00	$286,775.00
Dan Roberts	Chicago	Eastern	$300,000.00	$305,673.00
Bill Adams	Atlanta	Eastern	$350,000.00	$367,911.00
Sue Smith	Los Angeles	Western	$350,000.00	$474,050.00
Larry Fitch	Los Angeles	Western	$350,000.00	$361,865.00
Nancy Angelli	Denver	Western	$300,000.00	$186,042.00

Таблица **OFFICES**

OFFICE	CITY	REGION	MGR
22	Denver	Western	108
11	New York	Eastern	106
12	Chicago	Eastern	104
13	Atlanta	Eastern	NULL
21	Los Angeles	Western	108

Рис. 14.1. Типичное представление с двумя исходными таблицами

На рис. 14.1 изображено типичное представление. Оно получило имя REPDATA и определено в виде запроса к двум таблицам.

```
SELECT NAME, CITY, REGION, QUOTA, SALESREPS.SALES
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE;
```

Данные для этого представления берутся из таблиц SALESREPS и OFFICES. Эти таблицы называются *исходными таблицами* представления, поскольку служат для него источниками данных. Каждая строка представления содержит информацию о служащем, дополненную информацией о городе и регионе, в которых он работает. Как показывает рисунок, представление выглядит как таблица, а его содержимое в точности соответствует результатам, которые вы получили бы, если бы действительно выполнили запрос.

После определения представления к нему можно обращаться с помощью инструкции SELECT, как к обычной таблице.

Вывести список служащих, опережающих план, включая имя, город и регион.

```
SELECT NAME, CITY, REGION
  FROM REPDATA
 WHERE SALES > QUOTA;

NAME          CITY         REGION
-----------   -----------  -------
Mary Jones    New York     Eastern
Sam Clark     New York     Eastern
Dan Roberts   Chicago      Eastern
Paul Cruz     Chicago      Eastern
Bill Adams    Atlanta      Eastern
Sue Smith     Los Angeles  Western
Larry Fitch   Los Angeles  Western
```

Имя представления, REPDATA, указывается в предложении FROM как имя обычной таблицы, а ссылка на столбцы представления в инструкции SELECT осуществляется точно так же, как на столбцы таблицы. К некоторым представлениям можно также применять инструкции INSERT, DELETE и UPDATE для изменения данных. Таким образом, представление можно использовать в инструкциях SQL так, *как будто* оно является обычной таблицей. Однако перед тем как приступать к обновлению представления, крайне важно разобраться с его влиянием на лежащие в его основе таблицы.

Как СУБД работает с представлениями

Когда СУБД встречает в инструкции SQL ссылку на представление, она отыскивает его определение, сохраненное в базе данных. Затем СУБД преобразует пользовательский запрос, ссылающийся на представление, в *эквивалентный* запрос к исходным таблицам представления и выполняет его. Таким образом, СУБД создает иллюзию существования представления в виде отдельной таблицы и в то же время сохраняет целостность исходных таблиц.

Если определение представления простое, то СУБД формирует каждую строку представления на лету, извлекая данные из исходных таблиц. Если же определение сложное, СУБД приходится *материализовывать* представление. Это означает, что СУБД выполняет запрос, определяющий представление, и сохраняет его результаты во временной таблице. Из нее СУБД берет данные для формирования результатов пользовательского запроса, а когда временная таблица становится ненужной, удаляет ее. Но независимо от того, как именно СУБД выполняет инструкцию, являющуюся определением представления, для пользователя результат будет одним и тем же. Ссылаться на представление в инструкции SQL можно так же, как если бы оно было реальной таблицей базы данных.

Преимущества представлений

Использование представлений в базах данных различных типов может оказаться полезным в самых разнообразных ситуациях. В базах данных на персональных компьютерах представления применяются для удобства и позволяют упрощать запросы к базе данных. В промышленных базах данных представления играют главную роль в создании собственной структуры базы данных для каждого пользователя и обеспечении ее безопасности. Основные преимущества представлений перечислены ниже.

- **Безопасность**. Каждому пользователю можно разрешить доступ к небольшому числу представлений, содержащих только ту информацию, которую ему позволено знать. Таким образом можно осуществить ограничение доступа пользователей к хранимой информации.
- **Простота запросов**. С помощью представления можно извлечь данные из нескольких таблиц и представить их как одну таблицу, превращая тем самым запрос ко многим таблицам в однотабличный запрос к представлению.
- **Структурная простота**. С помощью представлений для каждого пользователя можно создать собственную структуру базы данных, определив ее как множество доступных пользователю виртуальных таблиц.
- **Защита от изменений**. Представление может возвращать непротиворечивый и неизменный образ структуры базы данных, даже если исходные таблицы разделяются, реструктуризуются или переименовываются. Заметим, однако, что определение представления должно быть обновлено, когда переименовываются лежащие в его основе таблицы или столбцы.
- **Целостность данных**. Если доступ к данным или ввод данных осуществляется с помощью представления, СУБД может автоматически проверять, выполняются ли определенные условия целостности.

Недостатки представлений

Наряду с перечисленными выше преимуществами, представления обладают и тремя существенными недостатками.

- **Производительность**. Представление создает лишь видимость существования соответствующей таблицы, и СУБД приходится преобразовывать запрос к представлению в запрос к исходным таблицам. Если представление отображает многотабличный запрос, то простой запрос к представлению становится сложным объединением и на его выполнение может потребоваться много времени. Однако это связано не с тем, что запрос обращается к представлению, — любой плохо построенный вопрос может вызвать проблемы с производительностью. Дело в том, что сложность запроса скрывается в представлении, так что пользователи не представляют, какой объем работы может вызвать даже кажущийся простым запрос.

- **Управляемость**. Представления, как и все прочие объекты баз данных, должны быть управляемы. Если разработчики и пользователи баз данных смогут бесконтрольно создавать представления, то работа администратора базы данных станет существенно сложнее. Это в особенности справедливо в том случае, когда создаются представления, в основе которых лежат другие представления, которые, в свою очередь, могут быть основаны на других представлениях. Чем больше уровней между базовыми таблицами и представлениями, тем сложнее решать проблемы с представлениями, которые могут возникнуть в такой системе.
- **Ограничения на обновление**. Когда пользователь пытается обновить строки представления, СУБД должна преобразовать запрос в запрос на обновление строк исходных таблиц. Это возможно для простых представлений; более сложные представления обновлять нельзя, они доступны только для выборки.

Указанные недостатки означают, что не стоит без разбора применять представления, в особенности многоуровневые, и использовать их вместо исходных таблиц. В каждом конкретном случае необходимо учитывать перечисленные выше преимущества и недостатки представлений.

Создание представлений (CREATE VIEW)

Для создания представлений используется инструкция CREATE VIEW, синтаксическая диаграмма которой изображена на рис. 14.2. В ней указываются имя представления и запрос, лежащий в его основе. Для успешного создания представления необходимо иметь права на доступ ко всем таблицам, входящим в запрос. В некоторых СУБД (таких, как Oracle) у вас должны быть права на создание представлений.

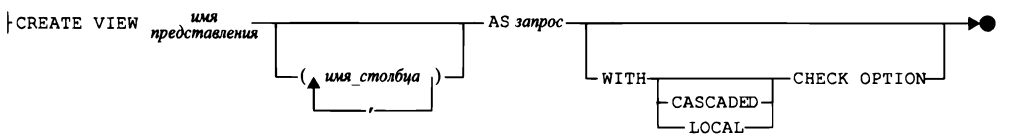


Рис. 14.2. Синтаксическая диаграмма инструкции CREATE VIEW

При необходимости в инструкции CREATE VIEW можно указать имя для каждого столбца создаваемого представления. Если указывается список имен столбцов, то он должен содержать столько элементов, сколько столбцов содержится в запросе. Обратите внимание на то, что задаются только имена столбцов; тип данных, длина и другие характеристики берутся из определения столбца в исходной таблице. Если список имен столбцов в инструкции CREATE VIEW отсутствует, каждый столбец представления получает имя соответствующего столбца запроса. Список имен столбцов обязательно должен быть указан, если в запросе получаются два столбца с одинаковыми именами, а в некоторых СУБД — при наличии в запросе вычисляемых столбцов. Хотя некоторые СУБД автоматически присваивают имена

вычисляемым столбцам, эти имена в действительности не слишком полезны, так что лучше иметь привычку всегда назначать собственные имена вычислимым столбцам.

Хотя все представления создаются одинаковым образом, на практике представления различных типов, как правило, используются для разных целей. В нескольких последующих разделах рассматриваются типы представлений и приводятся примеры инструкций CREATE VIEW.

Горизонтальные представления

Представления широко применяются для ограничения доступа пользователей к строкам таблиц, чтобы пользователи могли видеть не все строки, а только некоторые из них. Например, в учебной базе данных можно позволить менеджеру по продажам видеть в таблице SALESREPS только строки служащих, работающих в его регионе. Для этого можно использовать два приведенных далее представления.

Представление, показывающее информацию о служащих восточного региона.

```
CREATE VIEW EASTREPS AS
     SELECT *
       FROM SALESREPS
      WHERE REP_OFFICE IN (11, 12, 13);
```

Представление, показывающее информацию о служащих западного региона.

```
CREATE VIEW WESTREPS AS
     SELECT *
       FROM SALESREPS
      WHERE REP_OFFICE IN (21, 22);
```

Теперь каждому менеджеру по продажам можно разрешить доступ либо к представлению EASTREPS, либо к WESTREPS и одновременно запретить доступ к другому представлению, а также к таблице SALESREPS. Таким образом, менеджер по продажам получает собственное представление таблицы SALESREPS, в котором отражены только данные о служащих соответствующего региона.

Такие представления, как EASTREPS или WESTREPS, называются *горизонтальными представлениями*. Как видно из рис. 14.3, горизонтальное представление разрезает исходную таблицу по горизонтали. В него входят все столбцы исходной таблицы и только часть ее строк. Горизонтальные представления удобно применять, когда исходная таблица содержит данные, которые относятся к различным организациям или пользователям. Они предоставляют каждому пользователю личную таблицу, содержащую только те строки, которые ему необходимы.

Вот еще несколько примеров горизонтальных представлений.

Представление, содержащее офисы только восточного региона.

```
CREATE VIEW EASTOFFICES AS
     SELECT *
       FROM OFFICES
      WHERE REGION = 'Eastern';
```

Представление для Сью Смит (идентификатор служащего 102), показывающее заказы, сделанные только ее клиентами.

```
CREATE VIEW SUEORDERS AS
     SELECT *
       FROM ORDERS
      WHERE CUST IN (SELECT CUST_NUM
                       FROM CUSTOMERS
                      WHERE CUST_REP = 102);
```

Представление, показывающее только тех клиентов, которые в настоящий момент сделали заказы на сумму более $30 000.

```
CREATE VIEW BIGCUSTOMERS AS
     SELECT *
       FROM CUSTOMERS
      WHERE 30000.00 < (SELECT SUM(AMOUNT)
                          FROM ORDERS
                         WHERE CUST = CUST_NUM);
```

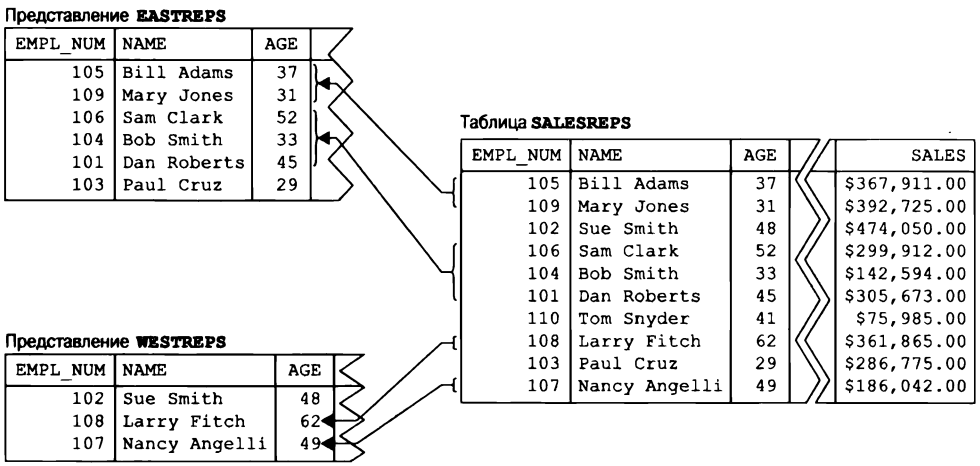


Рис. 14.3. Горизонтальные представления таблицы SALESREPS

В каждом из приведенных примеров источником данных для представления является одна исходная таблица. Представление задается с помощью инструкции SELECT * и потому содержит те же столбцы, что и исходная таблица. Предложение WHERE выбирает, какие строки исходной таблицы войдут в представление.

Вертикальные представления

Еще одним распространенным применением представлений является ограничение доступа к столбцам таблицы. Например, отделу, обрабатывающему заказы к учебной базе данных, для выполнения своих функций может потребоваться следующая информация: имя, идентификатор служащего и офис, в котором он работает. Но отделу вовсе не обязательно знать плановый и фактический объемы продаж того или иного служащего. Такой избирательный образ таблицы SALESREPS можно получить с помощью приведенного ниже представления.

Представление, показывающее избранную информацию о служащих.

```
CREATE VIEW REPINFO AS
     SELECT EMPL_NUM, NAME, REP_OFFICE
       FROM SALESREPS;
```

Разрешив отделу обработки заказов доступ к этому представлению и одновременно запретив доступ к самой таблице SALESREPS, можно ограничить доступ к конфиденциальной информации, каковой являются фактический и плановый объемы продаж.

Такие представления, как REPINFO, называются *вертикальными представлениями*. Как показано на рис. 14.4, вертикальное представление разрезает исходную таблицу по вертикали. Вертикальные представления часто применяются там, где данные используются различными пользователями или группами пользователей. Они предоставляют каждому пользователю личную виртуальную таблицу, содержащую только те столбцы, которые ему необходимы.

Представление REPINFO

EMPL_NUM	NAME	REP_OFFICE
105	Bill Adams	13
109	Mary Jones	11
102	Sue Smith	21
106	Sam Clark	11
104	Bob Smith	12
101	Dan Roberts	12
110	Tom Snyder	NULL
108	Larry Fitch	21
103	Paul Cruz	12
107	Nancy Angelli	22

Таблица SALESREPS

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	2006-02-12	104	$350,000.00	$367,911.00
109	Mary Jones	31	11	Sales Rep	2007-10-12	106	$300,000.00	$392,725.00
102	Sue Smith	48	21	Sales Rep	2004-12-10	108	$350,000.00	$474,050.00
106	Sam Clark	52	11	VP Sales	2006-06-14	NULL	$275,000.00	$299,912.00
104	Bob Smith	33	12	Sales Mgr	2005-05-19	106	$200,000.00	$142,594.00
101	Dan Roberts	45	12	Sales Rep	2004-10-20	104	$300,000.00	$305,673.00
110	Tom Snyder	41	NULL	Sales Rep	2008-01-13	101	NULL	$75,985.00
108	Larry Fitch	62	21	Sales Mgr	2007-10-12	106	$350,000,00	$361,865.00
103	Paul Cruz	29	12	Sales Rep	2005-03-01	104	$275,000.00	$286,775.00
107	Nancy Angelli	49	22	Sales Rep	2006-11-14	108	$300,000.00	$186,042.00

Рис. 14.4. Вертикальное представление таблицы SALESREPS

Вот еще несколько примеров вертикальных представлений.

Для отдела обработки заказов создать представление таблицы OFFICES, которое включает в себя идентификатор офиса, город и регион.

```
CREATE VIEW OFFICEINFO AS
     SELECT OFFICE, CITY, REGION
       FROM OFFICES;
```

Представление таблицы CUSTOMERS, включающее только имена клиентов и имена закрепленных за ними служащих.

```
CREATE VIEW CUSTINFO AS
     SELECT COMPANY, CUST_REP
       FROM CUSTOMERS;
```

В каждом из приведенных примеров источником данных для представления является одна исходная таблица. Список имен избранных столбцов в инструкции CREATE VIEW определяет, какие столбцы исходной таблицы войдут в представление. Поскольку это вертикальные представления, в них входят все строки исходных таблиц, и предложение WHERE в определении представления не требуется.

Смешанные представления

При создании представлений SQL не разделяет их на горизонтальные и вертикальные. В SQL просто отсутствуют понятия горизонтального и вертикального представлений. Они лишь помогают вам понять, каким образом из исходной таблицы формируется представление. Использование представлений, разделяющих исходную таблицу как в горизонтальном, так и вертикальном направлении, — вполне распространенное явление.

Представление, включающее идентификатор клиента, имя компании и лимит кредита для всех клиентов Билла Адамса (идентификатор служащего 105).

```
CREATE VIEW BILLCUST AS
     SELECT CUST_NUM, COMPANY, CREDIT_LIMIT
       FROM CUSTOMERS
      WHERE CUST_REP = 105;
```

Данные, полученные при помощи этого представления, представляют собой подмножество строк и столбцов таблицы CUSTOMERS. В этом представлении будут видны только те столбцы, которые явно указаны в предложении SELECT, и только те строки, которые удовлетворяют условию отбора в предложении WHERE.

Сгруппированные представления

Запрос, определяющий представление, может содержать предложение GROUP BY. Представление такого типа называется *сгруппированным представлением*, поскольку данные в нем являются результатом запроса с группировкой. Сгруппированные представления выполняют ту же функцию, что и запросы с группировкой, — в них родственные строки данных объединяются в группы и для каждой группы в таблице результатов запроса создается одна строка, содержащая итоговые данные по этой груп-

пе. С помощью сгруппированного представления запрос с группировкой превращается в виртуальную таблицу, к которой в дальнейшем можно обращаться с запросами.

Вот пример сгруппированного представления.

Представление, включающее суммарные данные о заказах по каждому служащему.

```
CREATE VIEW ORD_BY_REP (WHO, HOW_MANY, TOTAL,
                        LOW, HIGH, AVERAGE)
    AS
    SELECT REP, COUNT(*), SUM(AMOUNT), MIN(AMOUNT),
           MAX(AMOUNT), AVG(AMOUNT)
      FROM ORDERS
     GROUP BY REP;
```

Как видно из этого примера, в определении сгруппированного представления всегда содержится список имен столбцов. В этом списке присваиваются имена тем столбцам сгруппированного представления, которые являются производными от статистических функций, таких как SUM() и MIN(). В нем также может быть задано измененное имя столбца группировки. В данном примере столбец REP таблицы ORDERS становится столбцом WHO представления ORD_BY_REP.

После создания сгруппированного представления его можно использовать для упрощения запросов. Например, результатом приведенного ниже запроса является отчет, содержащий суммы заказов по каждому служащему.

Отобразить имя, число заказов, общую стоимость заказов и среднюю стоимость заказа по каждому служащему.

```
SELECT NAME, HOW_MANY, TOTAL, AVERAGE
  FROM SALESREPS, ORD_BY_REP
 WHERE WHO = EMPL_NUM
 ORDER BY TOTAL DESC;

NAME            HOW_MANY       TOTAL     AVERAGE
--------------- --------- ----------- -----------
Larry Fitch            7  $58,633.00   $8,376.14
Bill Adams             5  $39,327.00   $7,865.40
Nancy Angelli          3  $34,432.00  $11,477.33
Sam Clark              2  $32,958.00  $16,479.00
Dan Roberts            3  $26,628.00   $8,876.00
Tom Snyder             2  $23,132.00  $11,566.00
Sue Smith              4  $22,776.00   $5,694.00
Mary Jones             2   $7,105.00   $3,552.50
Paul Cruz              2   $2,700.00   $1,350.00
```

В отличие от горизонтальных и вертикальных представлений, каждой строке сгруппированного представления не соответствует какая-то одна строка исходной таблицы. Сгруппированное представление не является просто фильтром исходной таблицы, скрывающим некоторые строки и столбцы. Оно отображает исходную таблицу в виде резюме, поэтому поддержка такой виртуальной таблицы требует от СУБД значительного объема вычислений.

К сгруппированным представлениям можно обращаться с запросами точно так же, как и к более простым представлениям. Однако сгруппированные запросы нельзя обновлять. Причина очевидна: что значит, например, "обновить среднюю

величину заказа для служащего с идентификатором 105"? Так как каждая строка сгруппированного представления соответствует *группе* строк исходной таблицы, а столбцы, как правило, содержат вычисляемые данные, невозможно преобразовать запрос на обновление сгруппированного представления в запрос на обновление строк исходной таблицы. Таким образом, сгруппированные представления функционируют как представления, доступные в режиме только для чтения, к которым можно обращаться с запросами на выборку, но не на обновление.

Сгруппированные представления могут использоваться в запросах, которые группируют строки, выполняя тем самым двойное группирование строк, которое невозможно в обычных запросах. Рассмотрим следующий пример.

Для каждого офиса вывести на экран диапазон средних значений заказов для всех служащих, работающих в офисе.

```
SELECT REP_OFFICE, MIN(AVERAGE), MAX(AVERAGE)
  FROM SALESREPS, ORD_BY_REP
 WHERE EMPL_NUM = WHO
   AND REP_OFFICE IS NOT NULL
 GROUP BY REP_OFFICE;
```

Этот запрос корректно работает в большинстве современных реализаций SQL. Это двухтабличный запрос, который группирует строки представления ORD_BY_REP на основе офиса служащего. Вспомним, однако, что представление ORD_BY_REP уже сгруппировало строки. Если попытаться получить тот же результат без использования представления (помещая запрос, содержащийся в представлении ORD_BY_REP, в новый запрос), мы получим что-то наподобие следующего.

```
SELECT REP_OFFICE, MIN(AVG(AMOUNT)), MAX(AVG(AMOUNT))
  FROM SALESREPS, ORDERS
 WHERE EMPL_NUM = REP
 GROUP BY REP
 GROUP BY REP_OFFICE;
```

Такой запрос является недопустимым, поскольку содержит два предложения GROUP BY. Кроме того, в некоторых старых реализациях SQL не поддерживаются вложенные статистические функции, такие как MIN(AVG(AMOUNT)). Но, к счастью, все современные реализации SQL в настоящее время поддерживают их.

Соединенные представления

Часто представления используют для упрощения многотабличных запросов. Задавая в определении представления двух- или трехтабличный запрос, можно создать *соединенное представление* — виртуальную таблицу, данные в которую извлекаются из двух или трех различных таблиц. После создания такого представления к нему можно обращаться с помощью однотабличного запроса; в противном случае пришлось бы применять двух- или трехтабличное соединение. Предположим, например, что Сэм Кларк (Sam Clark), вице-президент по продажам, часто обращается с запросами к таблице ORDERS учебной базы данных. Однако Сэм не любит работать с идентификаторами служащих и клиентов. Он хотел бы восполь-

зоваться такой версией таблицы ORDERS, где вместо идентификаторов стояли бы имена. Вот представление, отвечающее требованиям Сэма.

Представление таблицы ORDERS с именами вместо идентификаторов.

```
CREATE VIEW ORDER_INFO (ORDER_NUM, COMPANY,
                        REP_NAME, AMOUNT) AS
     SELECT ORDER_NUM, COMPANY, NAME, AMOUNT
       FROM ORDERS, CUSTOMERS, SALESREPS
      WHERE CUST = CUST_NUM
        AND REP = EMPL_NUM;
```

Данное представление является объединением трех таблиц. Как и в случае сгруппированного представления, для создания такой виртуальной таблицы требуется значительный объем работы. Источником каждой строки представления является *комбинация* трех строк: по одной строке из таблиц ORDERS, CUSTOMERS и SALESREPS.

Несмотря на свое относительно сложное определение, это представление может принести реальную пользу. Вот запрос к представлению, дающий сводку заказов, сгруппированную по служащим и компаниям.

Для каждого служащего отобразить на экране общую стоимость заказов по каждой компании.

```
SELECT REP_NAME, COMPANY, SUM(AMOUNT)
  FROM ORDER_INFO
 GROUP BY REP_NAME, COMPANY;

REP_NAME     COMPANY           SUM(AMOUNT)
------------ ---------------- ------------
Bill Adams   Acme Mfg.          $35,582.00
Bill Adams   JCP Inc.            $3,745.00
Dan Roberts  First Corp.         $3,978.00
Dan Roberts  Holm & Landis         $150.00
Dan Roberts  Ian & Schmidt      $22,500.00
Larry Fitch  Midwest Systems     $3,608.00
Larry Fitch  Orion Corp.         $7,100.00
Larry Fitch  Zetacorp           $47,925.00
    .
    .
    .
```

Обратите внимание: этот запрос является однотабличной инструкцией SELECT, которая гораздо проще, чем эквивалентная трехтабличная инструкция SELECT для исходных таблиц.

```
SELECT NAME, COMPANY, SUM(AMOUNT)
  FROM SALESREPS, ORDERS, CUSTOMERS
 WHERE REP = EMPL_NUM
   AND CUST = CUST_NUM
 GROUP BY NAME, COMPANY;
```

Аналогично с помощью следующего запроса к данному представлению можно легко получить список крупных заказов, показывая, кто их сделал и кто принял.

Вывести список крупных заказов, упорядоченных по стоимости.

```
SELECT COMPANY, AMOUNT, REP_NAME
  FROM ORDER_INFO
 WHERE AMOUNT > 20000.00
 ORDER BY AMOUNT DESC;

COMPANY                  AMOUNT REP_NAME
------------------ ----------- --------------
Zetacorp            $45,000.00 Larry Fitch
J.P. Sinclair       $31,500.00 Sam Clark
Chen Associates     $31,350.00 Nancy Angelli
Acme Mfg.           $27,500.00 Bill Adams
Ace International   $22,500.00 Tom Snyder
Ian & Schmidt       $22,500.00 Dan Roberts
```

Значительно легче сформировать запрос к такому представлению, чем создавать эквивалентное объединение трех таблиц. Конечно, чтобы получить результаты однотабличного запроса к представлению, СУБД должна выполнить тот же объем работы, что и при генерации результатов эквивалентного трехтабличного запроса. В действительности, объем работы над запросом даже немного больше, чем над представлением. Но главное то, что пользователю гораздо легче создавать и понимать однотабличные запросы к представлению.

Обновление представлений

Что значит вставить, удалить или обновить строку представления? Для некоторых типов представлений эти операции можно очевидным образом преобразовать в эквивалентные операции по отношению к исходным таблицам представления. Например, вернемся к представлению EASTREPS, рассмотренному ранее в настоящей главе.

Представление, показывающее информацию о служащих только восточного региона.

```
CREATE VIEW EASTREPS AS
     SELECT *
       FROM SALESREPS
      WHERE REP_OFFICE IN (11, 12, 13);
```

Это простое горизонтальное представление, основанное на одной исходной таблице. Как видно из рис. 14.5, добавление строки в данное представление имеет смысл; оно означает, что новая строка должна быть вставлена в таблицу SALESREPS, лежащую в основе представления. Аналогично имеет смысл удаление строки из представления EASTREPS — это удаление соответствующей строки из таблицы SALESREPS. Наконец, обновление строки представления EASTREPS также имеет смысл: оно будет обновлением соответствующей строки таблицы SALESREPS. Во всех случаях требуемое действие можно выполнить по отношению к соответствующей строке исходной таблицы, тем самым сохраняя целостность исходной таблицы и представления.

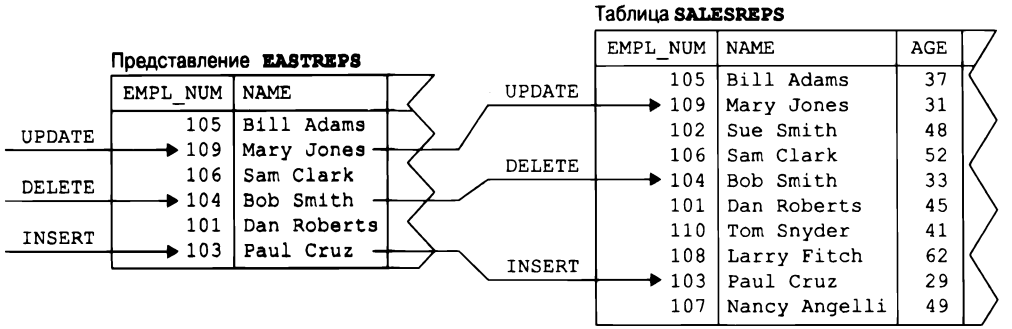


Рис. 14.5. Обновление данных с использованием представления

Теперь обратимся к сгруппированному представлению ORD_BY_REP, также рассматривавшемуся ранее в настоящей главе.

Представление, в которое входят суммарные данные о заказах по каждому служащему.

```
CREATE VIEW ORD_BY_REP (WHO, HOW_MANY, TOTAL,
                        LOW, HIGH, AVERAGE) AS
     SELECT REP, COUNT(*), SUM(AMOUNT), MIN(AMOUNT),
            MAX(AMOUNT), AVG(AMOUNT)
       FROM ORDERS
      GROUP BY REP;
```

Между строками этого представления и исходной таблицы нет взаимооднозначного соответствия, поэтому добавление, удаление или обновление строк представления не имеет смысла. Представление ORD_BY_REP обновлять невозможно, его можно только просматривать.

Представления EASTREPS и ORD_BY_REP являются двумя противоположностями в смысле сложности их определений. Имеются более сложные представления, чем EASTREPS, обновление которых возможно, и есть представления, менее сложные, чем ORD_BY_REP, обновление которых, однако, невозможно. Обновление представлений — непростой вопрос, который является предметом исследований в области реляционных баз данных уже много лет.

Обновление представлений и стандарт ANSI/ISO

В исходном стандарте SQL1 четко указано, какие представления базы данных обновимы в соответствии со стандартом (*обновимость* в данном контексте означает вставку, модификацию или удаление). Согласно стандарту, представление можно обновлять в том случае, если определяющий его запрос соответствует всем перечисленным ниже ограничениям.

- Должен отсутствовать предикат DISTINCT, т.е. повторяющиеся строки не должны исключаться из таблицы результатов запроса.
- В предложении FROM должна быть задана только одна обновляемая таблица, т.е. у представления должна быть одна исходная таблица, а пользователь должен иметь соответствующие права доступа к ней. Если исход-

ная таблица сама является представлением, то оно также должно удовлетворять этим условиям.

- Каждое имя в списке возвращаемых столбцов должно быть ссылкой на простой столбец; в этом списке не должны содержаться выражения, вычисляемые столбцы или статистические функции.
- Предложение WHERE не должно содержать подчиненный запрос; в нем могут присутствовать только простые построчные условия отбора.
- В запросе не должны содержаться предложения GROUP BY и HAVING.

Базовая концепция, скрывающаяся за этими ограничениями, проще для запоминания, чем сами правила. Чтобы представление было обновимо, СУБД должна быть способна для каждой строки представления найти соответствующую строку в исходной таблице, а для каждого обновляемого столбца представления — соответствующий столбец в исходной таблице.

Если представление соответствует этим требованиям, то над ним и над исходной таблицей можно выполнять имеющие смысл операции вставки, удаления и обновления. Однако если представление пропускает столбец исходной таблицы с ограничением NOT NULL без спецификации DEFAULT, то нельзя выполнить вставку новой строки в таблицу посредством данного представления. Причина в том, что в этом случае нет способа указать значение данных для пропущенного столбца.

Обновление представлений в коммерческих СУБД

Правила, установленные в стандарте SQL1, являются очень жесткими. Существует множество представлений, которые теоретически можно обновлять, хотя они соответствуют не всем этим правилам. Кроме того, существуют представления, над которыми можно производить не все операции обновления, а также представления, в которых можно обновлять не все столбцы. В большинстве коммерческих реализаций SQL правила обновления представлений значительно менее строгие, чем в стандарте SQL. Рассмотрим такой пример.

Представление, включающее плановый и фактический объемы продаж, а также разницу между ними для каждого служащего.

```
CREATE VIEW SALESPERF (EMPL_NUM, SALES, QUOTA, DIFF) AS
     SELECT EMPL_NUM, SALES, QUOTA, (SALES - QUOTA)
       FROM SALESREPS;
```

Стандарт SQL запрещает обновление этого представления, поскольку четвертый столбец у него вычисляемый. Однако обратите внимание: каждая строка представления основана только на одной строке исходной таблицы (SALESREPS). По этой причине в DB2 (и ряде других коммерческих СУБД) разрешается выполнять инструкцию DELETE по отношению к этому представлению. Кроме того, в DB2 разрешаются операции обновления столбцов EMPL_NUM, SALES и QUOTA, поскольку они берутся непосредственно из исходной таблицы. Не допускается только обновление столбца DIFF. В DB2 не разрешается выполнять инструкцию

INSERT по отношению к этому представлению, так как добавление какого-либо значения в столбец DIFF не имеет смысла.

Правила обновления представлений неодинаковы в различных СУБД, и обычно они довольно детальны. Некоторые представления, например, созданные на основе сгруппированных запросов, не обновляются ни в одной СУБД, так как эта операция просто не имеет смысла. Некоторые типы представлений в одних СУБД обновлять можно, в других можно только частично, а в третьих вообще нельзя. Это нашло свое отражение в последующих версиях стандарта SQL: в них расширен круг обновляемых представлений и допускается значительное разнообразие правил обновлений для различных СУБД. Чтобы узнать правила обновления представлений в какой-нибудь конкретной СУБД, лучше всего обратиться к руководству пользователя или поэкспериментировать с различными типами представлений.

Контроль над обновлением представлений (CHECK OPTION)

Если представление создается посредством запроса с предложением WHERE, то в представлении будут видны только строки, удовлетворяющие условию отбора. Остальные строки могут присутствовать в исходной таблице, но быть невидимы в представлении. Например, представление EASTREPS, которое уже рассматривалось ранее в настоящей главе, содержит только строки таблицы SALESREPS с определенными значениями в столбце REP_OFFICE.

Представление, показывающее информацию о служащих только восточного региона.

```
CREATE VIEW EASTREPS AS
     SELECT *
       FROM SALESREPS
      WHERE REP_OFFICE IN (11, 12, 13);
```

Это представление является обновляемым в большинстве коммерческих СУБД. В него можно добавить информацию о новом служащем посредством инструкции INSERT.

```
INSERT INTO EASTREPS (EMPL_NUM, NAME, REP_OFFICE,
                      AGE, HIRE_DATE, SALES)
     VALUES (113, 'Jake Kimball', 11, 43,
             '2009-01-01', 0.00)$
```

СУБД добавит новую строку в исходную таблицу SALESREPS; эта строка будет видна в представлении EASTREPS. Но давайте посмотрим, что получится, если добавить строку для нового служащего с помощью следующей инструкции INSERT.

```
INSERT INTO EASTREPS (EMPL_NUM, NAME, REP_OFFICE,
                      AGE, HIRE_DATE, SALES)
     VALUES (114, 'Fred Roberts', 21, 47,
             '2009-01-01', 0.00);
```

Это вполне корректная инструкция SQL, и СУБД произведет требуемое добавление в таблицу SALESREPS. Однако новая строка не удовлетворяет условию отбора для данного представления. Значение 21 в столбце REP_OFFICE этой строки означает офис в Лос-Анджелесе, относящийся к западному региону. В результате, если сразу после инструкции INSERT выполнить запрос

```
SELECT EMPL_NUM, NAME, REP_OFFICE
  FROM EASTREPS;

EMPL_NUM    NAME         REP_OFFICE
--------    -----------  ----------
     105    Bill Adams           13
     109    Mary Jones           11
     106    Sam Clark            11
     104    Bob Smith            12
     101    Dan Roberts          12
     103    Paul Cruz            12
```

то добавленная строка в нем будет отсутствовать. То же самое произойдет, если изменить идентификатор офиса у одного из служащих, включенных в представление. Данная инструкция UPDATE

```
UPDATE EASTREPS
   SET REP_OFFICE = 21
 WHERE EMPL_NUM = 104;
```

обновляет один из столбцов в строке Боба Смита (Bob Smith), что приводит к ее немедленному исчезновению из представления. Конечно, обе исчезнувшие строки будут видны в запросе к исходной таблице.

```
SELECT EMPL_NUM, NAME, REP_OFFICE
  FROM SALESREPS;

EMPL_NUM    NAME           REP_OFFICE
--------    -------------  ----------
     105    Bill Adams             13
     109    Mary Jones             11
     102    Sue Smith              21
     106    Sam Clark              11
     104    Bob Smith              21
     101    Dan Roberts            12
     110    Tom Snyder           NULL
     108    Larry Fitch            21
     103    Paul Cruz              12
     107    Nancy Angelli          22
     114    Fred Roberts           21
```

Тот факт, что в результате выполнения инструкции INSERT или UPDATE из представления исчезают строки, в лучшем случае вызывает замешательство. Вам, вероятно, захочется, чтобы СУБД обнаруживала такие инструкции и предотвращала их выполнение. SQL позволяет организовать этот вид контроля целостности представления путем создания представлений с *проверкой*. Проверка задается в инструкции CREATE VIEW с помощью предложения WITH CHECK OPTION.

```
DROP VIEW EASTREPS;

CREATE VIEW EASTREPS AS
     SELECT *
       FROM SALESREPS
      WHERE REP_OFFICE IN (11, 12, 13)
       WITH CHECK OPTION;
```

Обратите внимание на то, что уже существующее представление должно быть удалено и создано заново для того, чтобы добавить в него проверку. Удаление представлений будет рассмотрено далее.

Когда для представления установлен режим контроля, SQL автоматически проверяет каждую операцию INSERT или UPDATE, выполняемую над представлением, чтобы удостовериться, что полученные в результате строки удовлетворяют условиям отбора в определении представления. Если добавляемая или обновляемая строка не удовлетворяет этим условиям, то выполнение инструкции INSERT или UPDATE завершается ошибкой; другими словами, операция не выполняется.

Стандарт SQL позволяет также указывать параметр проверки CASCADE или LOCAL. Этот параметр применим только к тем представлениям, в основе которых лежит не таблица базы данных, а одно или несколько других представлений. В основе исходного представления может лежать другое представление и т.д. Для каждого из представлений в такой цепочке может быть задан (или не задан) режим проверки. Если представление верхнего уровня создано с предложением WITH CASCADE CHECK OPTION, то любая попытка обновить такое представление вынудит СУБД просмотреть всю цепочку представлений нижнего уровня и проверить те из них, для которых задан режим проверки. Если же представление верхнего уровня создано с предложением WITH LOCAL CHECK OPTION, то СУБД ограничится проверкой только этого представления. Считается, что параметр CASCADE установлен по умолчанию; его можно не указывать.

Из сказанного выше должно быть понятно, что введение режима проверки приводит к значительным затратам на выполнение инструкций INSERT и UPDATE по отношению к многоуровневым представлениям. Однако он играет важную роль в обеспечении целостности базы данных. Если обновляемое представление создается с целью повысить безопасность базы данных, то следует всегда задавать режим проверки. Тогда пользователь не сможет путем модификации представления воздействовать на данные, доступ к которым ему запрещен.

Удаление представления (DROP VIEW)

Вспомним, что в стандарте SQL1 язык определения данных (DDL) рассматривается как средство статического задания структуры базы данных, включающей таблицы и представления. По этой причине в стандарте SQL1 не предусмотрена возможность удаления представления, когда в нем больше нет необходимости. Однако во всех основных СУБД такая возможность существует. Поскольку представления подобны таблицам и не могут иметь совпадающие с ними имена, во многих СУБД для удаления представлений используется инструкция DROP TABLE. В других СУБД этой же цели служит отдельная инструкция DROP VIEW.

В стандарте SQL2 было формально закреплено использование инструкции DROP VIEW для удаления представлений. В нем также детализированы правила удаления представлений, на основе которых были созданы другие представления. Предположим, например, что с помощью инструкций CREATE VIEW были созданы следующие два представления таблицы SALESREPS.

```
CREATE VIEW EASTREPS AS
     SELECT *
       FROM SALESREPS
      WHERE REP_OFFICE IN (11, 12, 13);

CREATE VIEW NYREPS AS
     SELECT *
       FROM EASTREPS
      WHERE REP_OFFICE = 11;
```

Для иллюстрации правил удаления представление NYREPS создано на основе представления EASTREPS, хотя его с таким же успехом можно было бы создать на основе исходной таблицы. Согласно стандарту SQL, следующая инструкция DROP VIEW удалит из базы данных *оба* представления.

```
DROP VIEW EASTREPS CASCADE;
```

Параметр CASCADE означает, что СУБД должна удалить не только указанное в инструкции представление, но и все представления, созданные на его основе. В противоположность этому, инструкция DROP VIEW

```
DROP VIEW EASTREPS RESTRICT;
```

не выполнится и вернет ошибку, так как параметр RESTRICT означает, что СУБД должна удалить представление только в том случае, если нет других представлений, созданных на его основе. Это служит дополнительной защитой от случайных побочных эффектов при применении инструкции DROP VIEW. Стандарт SQL требует, чтобы в инструкции DROP VIEW обязательно присутствовал или параметр RESTRICT, или CASCADE, но в большинстве коммерческих СУБД используется инструкция DROP VIEW без каких-либо явно заданных параметров. Это сделано для поддержания обратной совместимости с теми продуктами, которые были выпущены до публикации стандарта SQL2. Поведение инструкции DROP VIEW в этом случае зависит от конкретной СУБД.

Материализованные представления*

Концептуально представление — это *виртуальная* таблица базы данных. Строки и столбцы представления физически в базе данных не хранятся: они выводятся из действительных данных лежащих в основе представления исходных таблиц. Если определение представления относительно простое (например, если представление — это простое подмножество строк/столбцов единственной таблицы или простое соединение, основанное на отношении внешнего ключа), то СУБД достаточно просто транслировать операции базы данных над представлением в операции над лежащими в его основе таблицами. В этой ситуации СУБД выполняет трансляцию на лету, операция за операцией, как при обработке запросов или обновлений базы данных. В общем случае операции, обновляющие базу данных через представление (операции INSERT, UPDATE или DELETE), всегда выполняются таким способом — путем трансляции операции в одну или несколько операций над исходными таблицами.

Если определение представления более сложное, СУБД может потребоваться материализовать представление для того, чтобы выполнить запросы к нему. СУБД реально выполняет запрос, определяющий представление, и сохраняет его результаты во временной таблице базы данных. Затем СУБД выполняет затребованный запрос ко временной таблице для получения интересующих результатов. По окончании обработки запроса СУБД уничтожает временную таблицу. На рис. 14.6 показан описанный процесс материализации. Понятно, что материализация содержимого представления может быть очень затратной операцией. Если типичная нагрузка на базу данных содержит много запросов, требующих материализации представлений, то общая пропускная способность СУБД может резко снизиться.

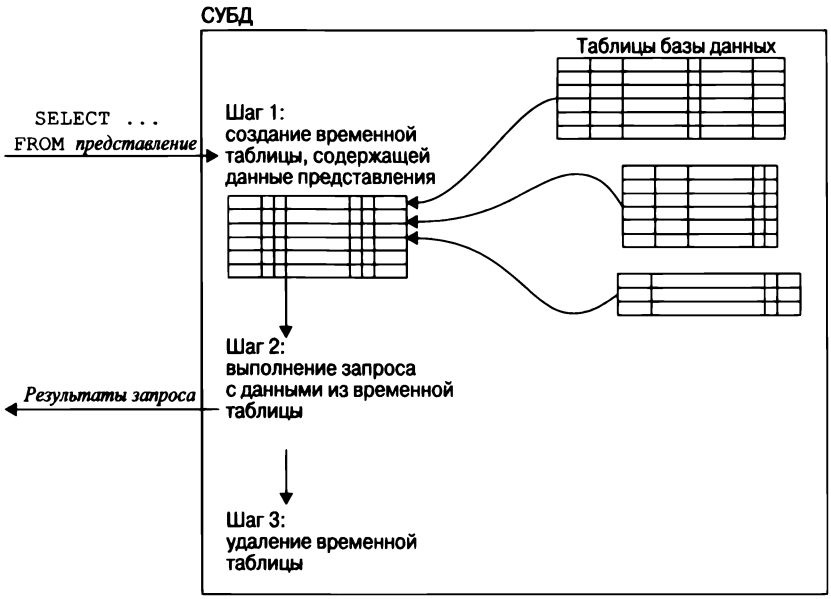


Рис. 14.6. Материализация представления для выполнения запроса

Для решения этой проблемы некоторые коммерческие СУБД поддерживают *материализованные представления*. Когда вы определяете представление как материализованное, СУБД выполняет определяющий его запрос (обычно в момент определения материализованного представления), сохраняет его результаты (т.е. данные, составляющие представление) в базе данных и постоянно поддерживает эту копию данных представления. Для поддержания точности данных материализованного представления СУБД должна автоматически проверять каждое изменение данных в исходных таблицах и выполнять соответствующие изменения в данных материализованного представления. В некоторых СУБД обновления материализованных представлений выполняются при обновлениях исходных таблиц, но более распространены СУБД, в которых изменения таблиц заносятся в журнал и применяются к материализованному представлению по расписанию, через определенные промежутки времени. Когда СУБД должна выполнить запрос к материализованному представлению, его данные уже готовы и запрос обрабатывается достаточно эффективно. На рис. 14.7 показана операция СУБД с материализованным представлением.

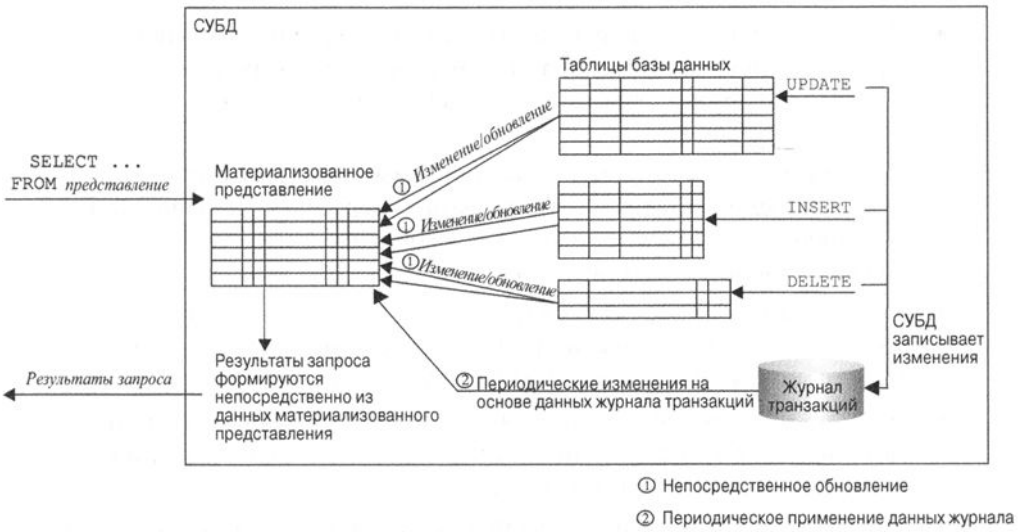


Рис. 14.7. Операция с материализованным представлением

Материализованные представления — это компромисс между эффективностью обновлений данных, содержащихся в представлении, и эффективностью запросов к данным представления. В случае нематериализованного представления на обновления исходных таблиц никак не влияет наличие представлений; они обрабатываются с обычной для СУБД скоростью. Однако запросы к нематериализованным представлениям могут быть гораздо менее эффективными по сравнению с запросами к обычным таблицам базы данных, поскольку СУБД должна на лету обработать запрос, что может потребовать выполнения большого количества работы.

Материализованные представления делают описанное соотношение обратным. При определении материализованного представления обновления исходных таблиц оказываются существенно менее эффективными, чем обновление обычных таблиц. Это связано с тем, что СУБД должна вычислить влияние обновления и соответствующим образом обновить данные материализованного представления. Однако запросы к материализованным представлениям могут обрабатываться с той же скоростью, что и запросы к реальным таблицам базы данных, поскольку материализованное представление есть не что иное, как обычная таблица. Таким образом, материализованное представление наиболее выгодно в том случае, когда количество обновлений лежащих в основе представления данных относительно мало, а количество запросов, напротив, велико.

Резюме

Представления дают возможность переопределять структуру базы данных, позволяя каждому пользователю видеть свою собственную структуру и свою часть содержимого базы данных.

- Представление — это виртуальная таблица, созданная на основе запроса. Представление, как и реальная таблица, содержит строки и столбцы данных, однако данные, видимые в представлении, на самом деле являются результатами запроса.
- Представление может быть простым подмножеством строк и столбцов одной таблицы, может резюмировать содержимое таблицы (сгруппированное представление) или содержать данные из двух или более таблиц (соединенное представление).
- В инструкциях `SELECT`, `INSERT`, `DELETE` и `UPDATE` к представлению можно обращаться, как к обычной таблице. Однако более сложные представления обновлять нельзя, они доступны только для чтения.
- Представления обычно используются для упрощения видимой структуры базы данных и запросов, а также для защиты некоторых строк и столбцов от несанкционированного доступа.
- Материализованные представления могут повысить эффективность работы базы данных в случае высокой активности запросов и низкой активности обновлений.

15
ГЛАВА

SQL и безопасность

При хранении данных в какой-либо СУБД одной из главных задач пользователя является обеспечение безопасности этих данных. Проблема безопасности особенно остро стоит в реляционных базах данных, поскольку интерактивный SQL позволяет легко получить к ним доступ. Требования, предъявляемые к системе безопасности в типичной реляционной базе данных, довольно разнообразны.

- Доступ к данным отдельной таблицы должен быть разрешен одним пользователям и запрещен другим.
- Одним пользователям должно быть позволено изменять данные в некоторой таблице, а другим — осуществлять только выборку данных из нее.
- К ряду таблиц доступ должен производиться только к отдельным столбцам.
- Определенным пользователям должно быть запрещено обращение к некоторой таблице с помощью интерактивного SQL, но разрешено пользоваться прикладными программами, изменяющими эту таблицу.

В настоящей главе рассматривается схема обеспечения безопасности базы данных с использованием SQL, позволяющая реализовать все упомянутые выше виды защиты данных в реляционной СУБД.

Принципы защиты данных, применяемые в SQL

За реализацию системы безопасности отвечает программное обеспечение СУБД. Язык SQL является фундаментом системы безопасности реляционной СУБД: требования, предъявляемые к системе защиты информации в базе данных, формулируются с помощью инструкций SQL. С защитой данных в SQL связаны три основные концепции.

- **Пользователи**. Действующими лицами в базе данных являются пользователи. Каждый раз, когда СУБД извлекает, вставляет, удаляет или об-

новляет данные, она делает это от имени какого-то пользователя. СУБД выполнит требуемое действие или откажется его выполнять в зависимости от того, какой пользователь запрашивает это действие.

- **Объекты базы данных**. Эти объекты являются теми элементами, защита которых может осуществляться посредством SQL. Обычно обеспечивается защита таблиц и представлений, но и другие объекты, такие как формы, прикладные программы и целые базы данных, также могут быть защищены. Большинству пользователей разрешается использовать одни объекты базы данных и запрещается использовать другие.
- **Привилегии**. Это — права пользователя на проведение тех или иных действий над определенным объектом базы данных. Например, пользователю может быть разрешено извлекать строки из некоторой таблицы и добавлять их в нее, но запрещено удалять или обновлять строки этой таблицы. У другого пользователя может быть иной набор привилегий.

На рис. 15.1 показано, как эти принципы могут применяться в учебной базе данных.

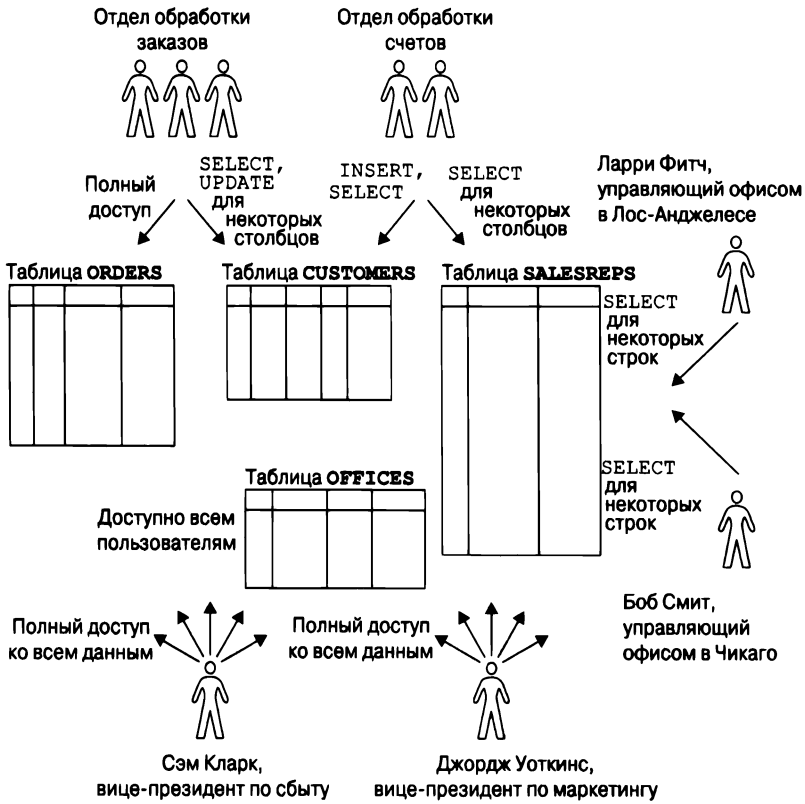


Рис. 15.1. Система безопасности учебной базы данных

Для введения элементов системы безопасности применяется инструкция GRANT, с помощью которой тем или иным пользователям предоставляются определенные привилегии на использование тех или иных объектов базы данных. Вот, например, инструкция GRANT, позволяющая Сэму Кларку (Sam Clark) извлекать данные из таблицы OFFICES учебной базы данных и добавлять их в нее.

Разрешить Сэму Кларку извлекать данные из таблицы OFFICES и добавлять их в нее.

```
GRANT SELECT, INSERT
   ON OFFICES
   TO SAM;
```

В инструкции GRANT задается комбинация идентификатора пользователя (SAM), объекта (таблица OFFICES) и привилегий (SELECT и INSERT). Предоставленные привилегии можно позднее аннулировать посредством инструкции REVOKE.

Отменить привилегии, предоставленные ранее Сэму Кларку.

```
REVOKE SELECT, INSERT
    ON OFFICES
  FROM SAM;
```

Инструкции GRANT и REVOKE более подробно описываются далее в данной главе.

Идентификаторы пользователей

Каждому пользователю в реляционной базе данных присваивается идентификатор — краткое имя, однозначно определяющее пользователя для программного обеспечения СУБД. Эти идентификаторы являются основой системы безопасности. Каждая инструкция SQL выполняется в СУБД от имени конкретного пользователя. От его идентификатора зависит, будет ли разрешено или запрещено выполнение конкретной инструкции. В промышленной базе данных идентификатор пользователя назначается ее администратором. В базе данных на персональном компьютере может быть только один идентификатор пользователя, который обозначает пользователя, создавшего базу данных и являющегося ее владельцем.

На практике ограничения на имена идентификаторов зависят от реализации СУБД. Стандарт SQL1 позволял идентификатору пользователя иметь длину до 18 символов и требовал, чтобы он был допустимым SQL-именем. В некоторых СУБД для мэйнфреймов длина идентификаторов ограничивалась восемью символами. В Sybase и SQL Server идентификатор пользователя может иметь до 30 символов. Если важна переносимость, то лучше всего, чтобы у идентификатора пользователя было не более восьми символов. На рис. 15.2 показаны различные пользователи, работающие с учебной базой данных, и типичные идентификаторы, присвоенные им. Обратите внимание на то, что всем пользователям в отделе обработки заказов можно присвоить один и тот же идентификатор, так как все они имеют идентичные привилегии.

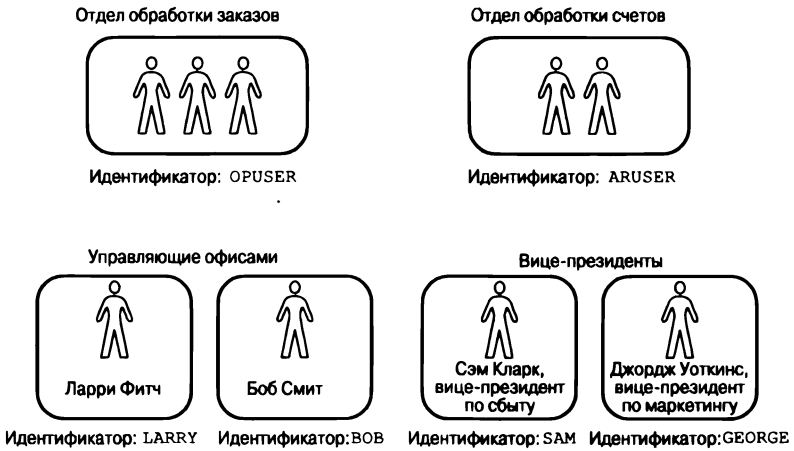


Рис. 15.2. Идентификаторы пользователей учебной базы данных

В стандарте ANSI/ISO вместо термина "идентификатор пользователя" (user-id) употребляется термин *идентификатор авторизации* (authorization-id), и он встречается иногда в документации по SQL. С технической точки зрения второй термин является более правильным, так как роль идентификатора заключается в определении прав или привилегий. Бывают ситуации (как на рис. 15.2), когда имеет смысл присвоить нескольким пользователям одинаковый идентификатор. В других ситуациях один пользователь может пользоваться двумя или тремя различными идентификаторами. В промышленной базе данных идентификатор прав доступа может относиться к программам и группам программ, а не к отдельным людям. В каждой из этих ситуаций термин "идентификатор авторизации" является более точным и ясным, чем термин "идентификатор пользователя". Однако наиболее распространена ситуация, когда каждому пользователю присваивается свой идентификатор, поэтому в документации большинства реляционных СУБД употребляется термин "идентификатор пользователя".

Аутентификация пользователей

Согласно стандарту SQL, безопасность базы данных обеспечивается с помощью идентификаторов пользователей. Однако в стандарте ничего не говорится о механизме связи идентификатора пользователя с инструкциями SQL. Например, если вы вводите инструкции SQL в интерактивном режиме, как СУБД определит, с каким идентификатором пользователя связаны эти инструкции? На сервере баз данных может существовать ежевечерне выполняющаяся программа генерации отчетов; каков в этом случае идентификатор, если нет никакого пользователя? Наконец, как обрабатываются запросы к базе данных по сети, если на локальном компьютере у пользователя может быть один идентификатор, а на удаленном компьютере — другой?

В большинстве коммерческих реляционных СУБД идентификатор пользователя создается для каждого *сеанса* (*сессии*) связи с базой данных. В интерактивном режиме сеанс начинается, когда пользователь запускает интерактивную програм-

му формирования запросов, и продолжается до тех пор, пока пользователь не выйдет из программы или не введет команду смены пользователя. В приложении, использующем программный SQL, сеанс начинается, когда приложение подключается к СУБД, и заканчивается по завершении приложения. В течение сеанса все инструкции SQL ассоциируются с идентификатором пользователя, установленным для этого сеанса. Однако в современных прикладных системах возможно также, что программа устанавливает несколько соединений с базой данных и выбирает одно из них для передачи инструкций SQL.

Как правило, в начале сеанса необходимо ввести как идентификатор пользователя, так и связанный с ним пароль. Пароль служит для подтверждения того, что пользователь действительно имеет право работать под введенным идентификатором. Однако ряд СУБД поддерживает аутентификацию операционной системы, т.е. СУБД получает мандат пользователя от операционной системы, без необходимости вводить пароль или выполнять иную аутентификацию. Хотя идентификаторы и пароли применяются в большинстве реляционных СУБД, способ, которым пользователь вводит свой идентификатор и пароль, зависит от конкретной СУБД.

В некоторых СУБД, в особенности тех, которые доступны для разных операционных систем, реализована своя собственная система безопасности с идентификаторами пользователей и паролями. Например, когда в СУБД Oracle вы работаете с интерактивной SQL-программой SQLPLUS, можете указать имя пользователя и соответствующий пароль в командной строке.

```
SQLPLUS SCOTT/TIGER
```

Заметим, что такой ввод пароля в командной строке использовать не рекомендуется, так как в этом случае пароль не шифруется и может быть перехвачен кем угодно в системе. Гораздо лучше опустить пароль (и разделительную косую черту) и позволить SQLPLUS самому запросить у вас пароль.

Интерактивный SQL-модуль СУБД Sybase, который называется ISQL, также может принимать имя пользователя и пароль в командной строке.

```
ISQL -U SCOTT -P TIGER
```

В каждом случае, прежде чем начать интерактивный сеанс связи, СУБД проверяет достоверность идентификатора пользователя (`SCOTT`) и пароля (`TIGER`). И вновь, лучше опустить пароль и позволить ISQL самому запросить его у вас. Старые версии SQL Server также поддерживают ISQL, но более новые используют для доступа к командной строке СУБД несколько отличающуюся утилиту OSQL.

Во многих других СУБД, включая Ingres и Informix, в качестве идентификаторов пользователей используются имена пользователей, регистрируемые в операционной системе. Например, когда вы регистрируетесь в вычислительной системе на основе UNIX, то вводите имя пользователя и пароль. Чтобы запустить затем интерактивный модуль в СУБД Ingres, вы просто даете команду

```
ISQL SALESDB
```

где `SALESDB` — это имя базы данных Ingres, с которой вы хотите работать. Ingres автоматически получает имя пользователя UNIX и делает его вашим идентификатором пользователя в текущем сеансе работы с СУБД. Таким образом, вам не тре-

буется задавать отдельный идентификатор пользователя и пароль для базы данных. Аналогичный прием применяется и в интерактивном SQL-модуле СУБД DB2, работающем в операционной системе MVS/TSO. Учетная запись TSO автоматически становится идентификатором пользователя интерактивного SQL-сеанса.

Большинство современных СУБД имеет для доступа к базе данных приложения с графическим пользовательским интерфейсом, такие как SQL Server Management Studio, DB2 UDB Command Editor или Oracle SQL Developer. Эти инструменты при подключении к базе данных запрашивают идентификатор пользователя и пароль.

Те же средства защиты используются при программном доступе к базе данных, так что СУБД должна определять и аутентифицировать идентификатор пользователя для каждой прикладной программы, пытающейся получить доступ к базе данных. И вновь, способы и правила применения идентификаторов пользователей меняются от одной СУБД к другой. Чаще всего программа в начале сеанса выдает пользователю диалоговое окно с запросом идентификатора и пароля. Специализированные или написанные пользователями программы могут использовать идентификатор, жестко закодированный в программе.

Стандарт SQL позволяет программе использовать идентификатор авторизации, связанный с конкретным набором инструкций SQL (такой набор называется *модулем*), а не идентификатор конкретного пользователя, запустившего программу. Такой механизм обеспечивает возможность выполнения программой различных задач от имени различных пользователей, даже если при прочих условиях этим пользователям запрещен доступ к соответствующей информации. Это удобная возможность, которая находит свое применение в основных реализациях SQL. Специфика безопасности SQL в случае программного обращения к базам данных рассматривается в главе 17, "Встроенный SQL".

Группы пользователей

В больших производственных базах данных часто имеются группы пользователей со схожими задачами. Например, в учебной базе данных три человека в отделе обработки заказов образуют естественную группу пользователей; два человека в финансовом отделе образуют другую естественную группу. В пределах каждой группы все пользователи работают с одинаковыми данными и должны иметь идентичные привилегии.

Согласно стандарту ANSI/ISO, с группами пользователей можно поступить одним из трех способов.

- Каждому члену группы можно присвоить один и тот же идентификатор пользователя, как показано на рис. 15.2. Это упрощает управление системой безопасности, так как позволяет установить привилегии доступа к данным один раз (в связи с тем, что идентификатор пользователя один). Однако в этом случае людей, совместно использующих один идентификатор пользователя, нельзя будет различить ни на дисплее системного оператора, ни в отчетах СУБД.
- Всем членам группы можно присвоить разные идентификаторы пользователя. Это позволит вам дифференцировать пользователей в отчетах СУБД и устанавливать в дальнейшем различные привилегии для отдель-

ных пользователей. Однако привилегии придется устанавливать для каждого пользователя индивидуально, что может быть утомительно и увеличивает вероятность возникновения ошибок.

- В тех, наиболее современных, СУБД, которые поддерживают эту возможность, можно создать *роль*, содержащую требуемые привилегии. Роль представляет собой именованную коллекцию привилегий. Можно назначить каждому пользователю его собственный идентификатор и связать с ним определенную роль. Очевидно, что это лучший выбор, поскольку можно различать пользователей, не усложняя при этом администрирование. Детальнее роли будут описаны позже в этой главе.

Выбор зависит от того, какие допускаются компромиссы в базе данных и прикладных программах.

До того, как была реализована поддержка ролей, в Sybase и SQL Server по отношению к группам пользователей был возможен иной, третий, вариант. В этих СУБД существуют идентификаторы групп, состоящих из связанных каким-либо образом пользователей. Привилегии могут быть предоставлены как пользователям, имеющим персональные идентификаторы, так и группам, также обладающим идентификаторами, и пользователи могут осуществлять действия, разрешенные и теми и другими привилегиями. Таким образом, идентификаторы групп упрощают управление системой безопасности. Однако они не соответствуют стандарту, и базы данных, в которых они применяются, могут не быть переносимы в другую СУБД.

В некоторых версиях DB2 также осуществляется поддержка групп пользователей, но другим способом. Администратор базы данных DB2 может конфигурировать ее таким образом, что когда вы первоначально подключаетесь к DB2 и сообщаете свой идентификатор пользователя (известный как ваш *первичный идентификатор авторизации*), DB2 автоматически ищет набор дополнительных идентификаторов пользователя (известных как *вторичные идентификаторы авторизации*), которыми вы можете пользоваться. Когда DB2 проверяет позднее ваши привилегии, она проверяет привилегии для всех идентификаторов прав доступа — как первичных, так и вторичных. Администратор базы данных DB2 обычно устанавливает вторичные идентификаторы прав доступа, совпадающие с именами групп пользователей, которые используются в RACF (Resource Access Control Facility, средство управления доступом к ресурсам — средство безопасности для мэйнфреймов компании IBM). Таким образом, применяемый в DB2 подход фактически вводит идентификаторы групп, не расширяя явно механизм идентификаторов пользователя.

Защищаемые объекты

Средства защиты в SQL применяются по отношению к отдельным *объектам* базы данных. В стандарте SQL1 указаны два типа защищаемых объектов — таблицы и представления. Таким образом, каждая таблица и каждое представление могут быть защищены индивидуально. Доступ к ним может быть разрешен для одних пользователей и запрещен для других. Последующие версии стандарта SQL рас-

ширяют круг защищаемых объектов, включая в него домены и пользовательские наборы символов, и добавляют новые типы защиты таблиц и представлений.

В большинстве коммерческих СУБД дополнительно могут быть защищены и другие типы объектов. Например, в SQL Server важным объектом базы данных является *хранимая процедура*. С помощью средств защиты SQL устанавливается, какие пользователи могут создавать и удалять хранимые процедуры и какие пользователи могут их выполнять. В DB2 защищаемыми объектами являются *табличные пространства* — физические области хранения таблиц. Администратор базы данных может для одних пользователей разрешать создание новых таблиц в некоторой физической области, а для других — запрещать. Другие реляционные СУБД могут защищать иные объекты. В целом, однако, базовый механизм обеспечения безопасности в SQL — выдача и отмена привилегий на определенные объекты с помощью определенных инструкций SQL — одинаковым образом реализован практически во всех СУБД.

Привилегии

Множество действий, которые пользователь имеет право выполнять над объектом базы данных, называется *привилегиями* пользователя по отношению к данному объекту. В стандарте SQL1 для таблиц и представлений определены четыре привилегии.

- Привилегия `SELECT` позволяет извлекать данные из таблицы или представления. Имея эту привилегию, можно задавать имя таблицы или представления в предложении `FROM` инструкции `SELECT` или подзапроса.
- Привилегия `INSERT` позволяет вставлять новые записи в таблицу или представление. Имея эту привилегию, можно задавать имя таблицы или представления в предложении `INTO` инструкции `INSERT`.
- Привилегия `DELETE` позволяет удалять записи из таблицы или представления. Имея эту привилегию, можно задавать имя таблицы или представления в предложении `FROM` инструкции `DELETE`.
- Привилегия `UPDATE` позволяет модифицировать записи в таблице или представлении. Имея эту привилегию, можно задавать таблицу или представление в инструкции `UPDATE` как целевую таблицу. Привилегия `UPDATE` может быть ограничена отдельными столбцами таблицы или представления, позволяя тем самым обновлять только эти столбцы и запрещая обновлять другие.

Эти четыре привилегии поддерживаются почти всеми коммерческими реализациями SQL.

Расширенные привилегии SQL

В последующих версиях стандарта SQL базовый механизм управления привилегиями стандарта SQL1 значительно расширен. Определенные в SQL1 привилегии `SELECT`, `INSERT` и `UPDATE` дополнены новыми возможностями. Появилась новая привилегия `REFERENCES`, ограничивающая право пользователя на создание

внешних ключей доступной ему таблицы для ссылок на другие таблицы. Кроме того, добавлена новая привилегия USAGE, управляющая доступом к новым структурам баз данных SQL — доменам, наборам символов, порядкам сортировки и правилам конвертирования текста.

Расширения привилегий SELECT, INSERT и UPDATE достаточно просты. Теперь они могут относиться к конкретному столбцу или столбцам таблицы, а не только ко всей таблице целиком. Давайте рассмотрим их полезность на примере нашей учебной базы данных. Предположим, вы хотите, чтобы за добавление новых служащих в таблицу SALESREPS отвечал начальник отдела кадров. Он должен вводить идентификатор принятого на работу служащего, его имя и прочую учетную информацию. Однако за назначение новому служащему планового объема продаж, т.е. за заполнение столбца QUOTA, отвечает не он, а начальник отдела сбыта. Аналогично начальник отдела кадров не должен иметь доступа к столбцу SALES имеющихся записей таблицы SALESREPS. Используя новые возможности стандарта SQL, можно реализовать эту схему, предоставив начальнику отдела кадров привилегию INSERT для соответствующих столбцов. Остальные столбцы (SALES и QUOTA) будут при создании новой записи принимать значения NULL. Начальник же отдела сбыта получает привилегию UPDATE для столбцов SALES и QUOTA, чтобы он мог назначать служащим плановые объемы продаж. При отсутствии возможности указывать привилегии для конкретных столбцов вам придется либо ослаблять упомянутые ограничения на доступ к отдельным столбцам, либо определять дополнительные представления просто для ограничения доступа.

Введение новой привилегии REFERENCES связано с более тонкой особенностью системы безопасности баз данных, касающейся определения внешних ключей и ограничений на значения столбцов. Проанализируем это на примере нашей учебной базы данных. Предположим, что служащий компании может создавать в базе данных новые таблицы (например, ему может понадобиться таблица с информацией о новом товаре), но у него нет доступа к информации о служащих в таблице SALESREPS. При такой схеме защиты может показаться, что он никак не сможет узнать используемые компанией идентификаторы служащих или выяснить, был ли принят на работу новый сотрудник.

Однако это не совсем верно. Служащий может создать новую таблицу и определить один из ее столбцов как внешний ключ для связи с таблицей SALESREPS. Как вы помните, это означает, что допустимыми значениями этого столбца будут только значения первичного ключа из таблицы SALESREPS, т.е. допустимые идентификаторы служащих. Чтобы выяснить, работает ли в компании служащий с некоторым идентификатором, нашему "хакеру" достаточно будет попытаться добавить в свою таблицу новую строку с этим значением внешнего ключа. Если это получится, значит, такой служащий в компании есть, если нет — то и служащего с таким идентификатором в компании нет.

Еще более серьезную проблему порождает возможность создания новых таблиц с ограничениями на значения столбцов. Предположим, например, что служащий пытается выполнить такую инструкцию.

```
CREATE TABLE XYZ (TRYIT DECIMAL(9,2),
       CHECK ((SELECT QUOTA
                 FROM SALESREPS
                WHERE NAME = 'VP Sales')
     BETWEEN 400000 AND 500000));
```

Поскольку значение столбца TRYIT созданной служащим таблицы теперь связано со значением столбца QUOTA конкретной строки таблицы SALESREPS, то успешное выполнение приведенной инструкции означает, что квота вице-президента по сбыту находится в указанном диапазоне! В противном случае можно попробовать аналогичную инструкцию, подкорректировав границы диапазона. Заметим, впрочем, что только некоторые реализации SQL поддерживают проверку с обращениями к другим таблицам. На момент написания данного материала такой поддержкой обладала только MySQL, но не DB2, SQL Server или Oracle.

Эту лазейку для доступа к данным и закрывает введенная в стандарт SQL новая привилегия REFERENCES. Подобно привилегиям SELECT, INSERT и UPDATE, она назначается на отдельные столбцы таблицы. Если у пользователя есть привилегия REFERENCES для некоторого столбца, он может создавать новые таблицы, связанные с этим столбцом тем или иным способом (например, через внешний ключ или ограничения на значения столбцов, как в предыдущих примерах). Если же СУБД не поддерживает привилегию REFERENCES, но поддерживает внешние ключи и ограничения на значения столбцов, то иногда эту же задачу может выполнить привилегия SELECT.

Наконец, стандарт SQL определяет новую привилегию USAGE, управляющую доступом к доменам (наборам допустимых значений столбцов), пользовательским наборам символов, порядкам сортировки и правилам конвертирования текста. Привилегия USAGE просто разрешает или запрещает использование этих объектов базы данных по имени для отдельных идентификаторов пользователей. Например, имея привилегию USAGE на некоторый домен, можно определить новую таблицу и указать этот домен в качестве типа данных какого-нибудь из ее столбцов. Без такой привилегии использовать этот домен для определения столбцов нельзя. Привилегия USAGE служит, главным образом, для упрощения администрирования больших корпоративных баз данных, которые используются и модифицируются несколькими различными командами разработчиков. Ее введение не связано с теми проблемами безопасности, с которыми связаны привилегии для доступа к таблицам и столбцам.

Привилегии владения

Когда вы создаете таблицу посредством инструкции CREATE TABLE, становитесь ее владельцем и получаете все привилегии для этой таблицы (SELECT, INSERT, DELETE, UPDATE и другие привилегии, имеющиеся в СУБД). Все прочие пользователи изначально не имеют никаких привилегий для работы со вновь созданной таблицей. Чтобы они получили доступ к таблице, вы должны явно предоставить им соответствующие привилегии с помощью инструкции GRANT.

Когда с помощью инструкции CREATE VIEW вы создаете представление, то становитесь его владельцем, но не обязательно получаете по отношению к нему все привилегии. Для успешного создания представления необходимо иметь привилегию SELECT для каждой исходной таблицы представления, поэтому привилегию SELECT для работы с созданным представлением вы получаете автоматически. Что

касается остальных привилегий (INSERT, DELETE и UPDATE), то вы получаете их только в том случае, если имели их для *каждой* исходной таблицы представления.

Другие привилегии

Во многих коммерческих СУБД, помимо привилегий SELECT, INSERT, DELETE и UPDATE, по отношению к таблицам и представлениям могут быть предоставлены дополнительные привилегии. Например, в Oracle и базах данных для мэйнфреймов компании IBM предусмотрены привилегии ALTER и INDEX. Имея привилегию ALTER для какой-либо таблицы, пользователь может с помощью инструкции ALTER TABLE модифицировать определение данной таблицы; имея привилегию INDEX, пользователь может посредством инструкции CREATE INDEX создать индекс для таблицы. В тех СУБД, где отсутствуют привилегии ALTER и INDEX, только владелец таблицы может пользоваться инструкциями ALTER TABLE и CREATE INDEX.

Часто в СУБД имеются дополнительные привилегии не для таблиц и представлений, а для других объектов. Например, Oracle, Sybase и SQL Server поддерживают привилегию EXECUTE для хранимых процедур, которая определяет, кто из пользователей имеет право ее выполнять. В DB2 имеется привилегия USE для табличных пространств, определяющая, кто из пользователей может создавать таблицы в определенном табличном пространстве.

Представления и безопасность SQL

Наряду с привилегиями, ограничивающими доступ к таблицам, представления также играют ключевую роль в защите данных. Создавая представление и давая пользователю разрешение на доступ к нему, а не к исходной таблице, можно тем самым ограничить доступ пользователя, позволив ему обращаться только к определенным столбцам и строкам. Таким образом, представления позволяют осуществлять четкий контроль над тем, какие данные доступны тому или иному пользователю.

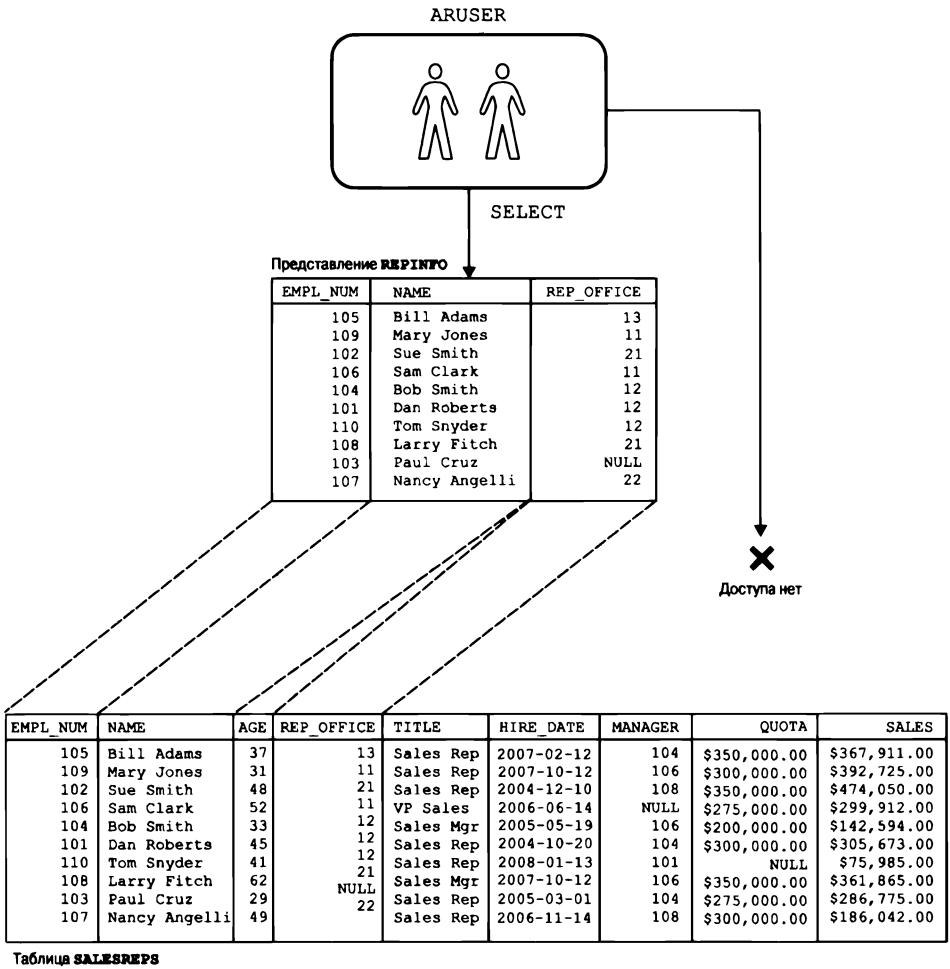
Предположим, например, что вы решили установить в учебной базе данных следующее правило защиты данных.

"Персонал финансового отдела имеет право извлекать из таблицы SALESREPS идентификаторы и имена служащих, а также идентификаторы офисов; информация об объеме продаж и личных планах должна быть для них недоступна."

Это правило можно реализовать, создав представление

```
CREATE VIEW REPINFO AS
     SELECT EMPL_NUM, NAME, REP_OFFICE
       FROM SALESREPS;
```

и назначив привилегию SELECT для этого представления пользователю с идентификатором ARUSER (рис. 15.3). В данном примере для ограничения доступа к отдельным столбцам используется вертикальное представление.



EMPL_NUM	NAME	REP_OFFICE
105	Bill Adams	13
109	Mary Jones	11
102	Sue Smith	21
106	Sam Clark	11
104	Bob Smith	12
101	Dan Roberts	12
110	Tom Snyder	12
108	Larry Fitch	21
103	Paul Cruz	NULL
107	Nancy Angelli	22

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	2007-02-12	104	$350,000.00	$367,911.00
109	Mary Jones	31	11	Sales Rep	2007-10-12	106	$300,000.00	$392,725.00
102	Sue Smith	48	21	Sales Rep	2004-12-10	108	$350,000.00	$474,050.00
106	Sam Clark	52	11	VP Sales	2006-06-14	NULL	$275,000.00	$299,912.00
104	Bob Smith	33	12	Sales Mgr	2005-05-19	106	$200,000.00	$142,594.00
101	Dan Roberts	45	12	Sales Rep	2004-10-20	104	$300,000.00	$305,673.00
110	Tom Snyder	41	12	Sales Rep	2008-01-13	101	NULL	$75,985.00
108	Larry Fitch	62	21	Sales Mgr	2007-10-12	106	$350,000.00	$361,865.00
103	Paul Cruz	29	NULL	Sales Rep	2005-03-01	104	$275,000.00	$286,775.00
107	Nancy Angelli	49	22	Sales Rep	2006-11-14	108	$300,000.00	$186,042.00

Рис. 15.3. Ограничение доступа к столбцам с помощью представления

С помощью горизонтальных представлений также можно эффективно реализовывать правила защиты данных. Рассмотрим следующее правило.

"Менеджеры по продажам каждого региона должны иметь доступ ко всем данным, касающимся служащих только их региона".

Как показано на рис. 15.4, можно создать два представления, EASTREPS и WESTREPS, в которых содержатся данные таблицы SALESREPS отдельно по каждому региону, а затем каждому менеджеру по продажам дать разрешение на доступ к соответствующему представлению.

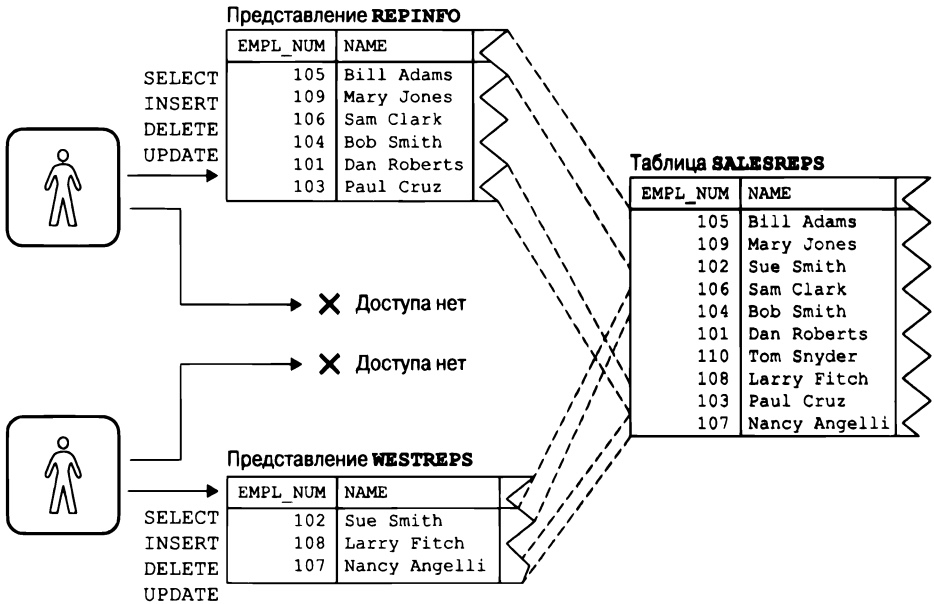


Рис. 15.4. Ограничение доступа к строкам с помощью представления

Конечно, представления могут быть гораздо сложнее, чем простое подмножество строк и столбцов одной таблицы, как в приведенном примере. Создавая представление посредством запроса с группировкой, пользователю можно предоставить доступ к итоговым данным, а не к подробным данным исходной таблицы. В представлении можно объединять данные из двух или более таблиц, разрешая пользователю доступ именно к тем данным, которые ему необходимы, и запрещая доступ ко всем остальным. Использование представлений для защиты данных в SQL ограничивается двумя важными факторами, упомянутыми ранее в главе 14, "Представления".

- **Ограничения на обновление данных**. Если представление доступно только для выборки, то для него можно установить привилегию SELECT, но привилегии INSERT, DELETE и UPDATE не имеют для него смысла. Когда пользователь должен обновлять данные, видимые в доступном только для выборки представлении, ему следует дать разрешение на обновление исходных таблиц и он должен иметь возможность выполнять инструкции INSERT, DELETE и UPDATE по отношению к этим таблицам.
- **Производительность**. Так как СУБД преобразует каждое обращение к представлению в соответствующие обращения к исходным таблицам, при использовании представлений может значительно увеличиться трудоемкость операций, проводимых в базе данных. Поэтому представления нельзя неосмотрительно применять для ограничения доступа к данным — это неизбежно влечет за собой снижение общей производительности СУБД.

Предоставление привилегий (GRANT)

Инструкция GRANT, синтаксическая диаграмма которой изображена на рис. 15.5, используется для предоставления пользователям привилегий для работы с объектами базы данных. Обычно инструкцией GRANT пользуется владелец таблицы или представления, чтобы разрешить другим пользователям доступ к этим данным. Как видно из рисунка, инструкция GRANT содержит список предоставляемых привилегий, имя таблицы или иного объекта, для которого назначаются привилегии (для всех объектов, кроме таблиц и представлений, требуется указание типа объекта), и идентификатор пользователя или роль, получающую эти привилегии. В большинстве реализаций SQL для предоставления привилегий пользователю его учетная запись должна уже иметься в базе данных.

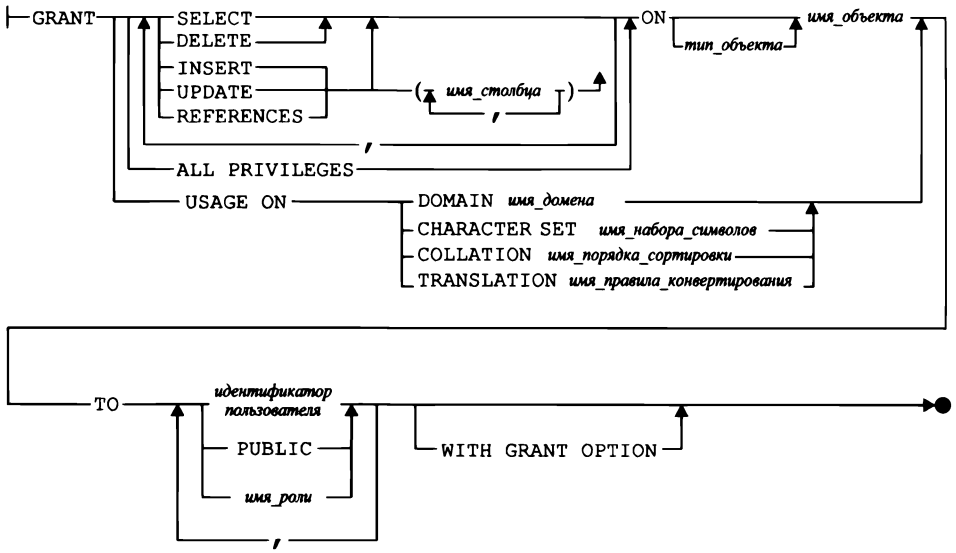


Рис. 15.5. Синтаксическая диаграмма инструкции GRANT

Синтаксическая диаграмма инструкции GRANT, изображенная на рисунке, соответствует стандарту ANSI/ISO. Многие СУБД придерживаются синтаксиса инструкции GRANT, который применяется в DB2. Он более гибок и позволяет задавать список идентификаторов пользователей и список таблиц, упрощая предоставление большого числа привилегий. Вот несколько примеров простых инструкций GRANT для учебной базы данных.

Предоставить пользователям из отдела обработки заказов полный доступ к таблице ORDERS.

```
GRANT SELECT, INSERT, DELETE, UPDATE
   ON ORDERS
   TO OPUSER;
```

Разрешить пользователям финансового отдела извлекать данные о заказчиках и добавлять новых заказчиков в таблицу CUSTOMERS; пользователям отдела обработки заказов разрешить только выборку из этой таблицы.

```
GRANT SELECT, INSERT
   ON CUSTOMERS
   TO ARUSER;

GRANT SELECT
   ON CUSTOMERS
   TO OPUSER;
```

Разрешить Сэму Кларку добавлять и удалять информацию об офисах.

```
GRANT INSERT, DELETE
   ON OFFICES
   TO SAM;
```

При предоставлении большого числа привилегий или предоставлении привилегий большому числу пользователей в инструкции GRANT для удобства применяются два сокращения. Вместо того чтобы перечислять все привилегии для некоторого объекта, можно использовать предложение ALL PRIVILEGES. Приведенная ниже инструкция GRANT разрешает вице-президенту по сбыту Сэму Кларку полный доступ к таблице SALESREPS.

Предоставить Сэму Кларку все привилегии по отношению к таблице SALESREPS.

```
GRANT ALL PRIVILEGES
   ON SALESREPS
   TO SAM;
```

Вместо того чтобы давать привилегии по отдельности каждому пользователю базы данных, можно с помощью ключевого слова PUBLIC предоставить их сразу всем пользователям, имеющим право работать с базой данных. Очевидно, что этой возможностью надо пользоваться крайне осторожно. Следующая инструкция GRANT разрешает всем пользователям извлекать данные из таблицы OFFICES.

Дать всем пользователям привилегию SELECT для таблицы OFFICES.

```
GRANT SELECT
   ON OFFICES
   TO PUBLIC;
```

Обратите внимание: такая инструкция GRANT дает привилегии всем нынешним и будущим пользователям базы данных, а не только тем, кто имеет право работать с базой данных в настоящее время. Это устраняет необходимость явно предоставлять привилегии новым пользователям при их появлении.

Привилегии для работы со столбцами

Стандарт SQL1 позволял предоставлять привилегию UPDATE для отдельных столбцов таблицы или представления, а новейшие версии стандарта дают возможность предоставлять таким же образом привилегии SELECT, INSERT и REFERENCES. Список столбцов располагается после ключевого слова SELECT, UPDATE, INSERT

или REFERENCES и заключается в круглые скобки. Вот как выглядит инструкция GRANT, разрешающая сотрудникам отдела обработки заказов обновлять в таблице CUSTOMERS только столбцы с названием компании (COMPANY) и идентификатором закрепленного за ней служащего (CUST_REP).

Разрешить пользователям отдела обработки заказов изменять названия компаний, а также закрепленных за компаниями служащих.

```
GRANT UPDATE (COMPANY, CUST_REP)
   ON CUSTOMERS
   TO OPUSER;
```

Если список столбцов опущен, то привилегия действительна для всех столбцов таблицы или представления.

Разрешить пользователям финансового отдела изменять всю информацию о клиентах.

```
GRANT UPDATE
   ON CUSTOMERS
   TO ARUSER;
```

Стандарты SQL после SQL1 поддерживают предоставление привилегии SELECT для списка столбцов, как в приведенном далее примере.

Разрешить пользователям финансового отдела доступ только для чтения к идентификаторам, именам и офисам служащих из таблицы SALESREPS.

```
GRANT SELECT (EMPL_NUM, NAME, REP_OFFICE)
   ON SALESREPS
   TO ARUSER;
```

Такая инструкция GRANT устраняет необходимость в представлении REPINFO, показанном на рис. 15.3, и на практике может сделать ненужным большое количество представлений в промышленных базах данных. Однако эта возможность пока что поддерживается далеко не всеми ведущими СУБД.

Передача привилегий (GRANT OPTION)

Если вы создаете в базе данных объект и становитесь его владельцем, только вы можете предоставлять привилегии для работы с этим объектом. Когда вы предоставляете привилегии другим пользователям, они могут пользоваться объектом, но не могут передавать эти привилегии кому-то еще. Таким образом, владелец объекта сохраняет строгий контроль над тем, кому какие формы доступа к объекту разрешены.

Но бывают ситуации, когда владелец может захотеть передать другим пользователям право предоставлять привилегии для работы со своим объектом. Например, обратимся еще раз к представлениям EASTREPS и WESTREPS в учебной базе данных. Их создал и владеет ими вице-президент по сбыту Сэм Кларк. Посредством следующей инструкции GRANT он может дать менеджеру лос-анжелесского офиса Ларри Фитчу разрешение пользоваться представлением WESTREPS.

```
GRANT SELECT
   ON WESTREPS
   TO LARRY;
```

Что произойдет, если Ларри захочет дать Сью Смит (идентификатор пользователя SUE) разрешение на доступ к представлению WESTREPS в связи с тем, что она составляет прогноз объема продаж для офиса в Лос-Анджелесе? Он не сможет этого сделать, так как предыдущая инструкция GRANT неявно запрещает подобное действие. Требуемую привилегию может предоставить только Сэм Кларк, так как именно он является владельцем представления.

Если Сэму требуется, чтобы Ларри по своему усмотрению решал, кому давать привилегию для работы с представлением WESTREPS, он может воспользоваться следующей разновидностью инструкции GRANT.

```
GRANT SELECT
   ON WESTREPS
   TO LARRY
 WITH GRANT OPTION
```

Наличие предложения WITH GRANT OPTION означает, что инструкция GRANT наряду с привилегиями дает право на предоставление этих привилегий другим пользователям.

Теперь Ларри может выполнить инструкцию GRANT

```
GRANT SELECT
   ON WESTREPS
   TO SUE
```

которая позволяет Сью Смит извлекать данные из представления WESTREPS. На рис. 15.6 процедура передачи привилегий изображена графически: сначала от Сэма к Ларри, а затем от Ларри к Сью. Так как Ларри не включил в свою инструкцию GRANT предложение WITH GRANT OPTION, цепочка заканчивается на Сью; она может извлекать данные из представления WESTREPS, но не может давать право на доступ к нему другому пользователю. Однако если бы Ларри включил в свою инструкцию GRANT предложение WITH GRANT OPTION, цепочка могла бы протянуться дальше, поскольку Сью получила бы возможность предоставлять право на доступ другим пользователям.

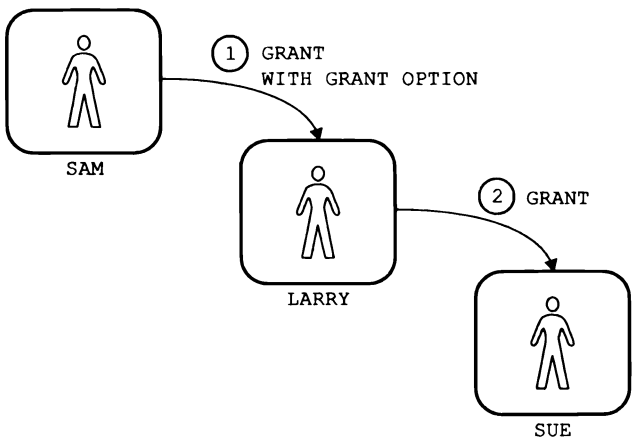


Рис. 15.6. Применение GRANT OPTION

Ларри мог бы поступить иначе: создать для Сью представление, включающее только служащих лос-анжелесского офиса, и разрешить ей доступ к этому представлению.

```
CREATE VIEW LAREPS AS
     SELECT *
       FROM WESTREPS
      WHERE OFFICE = 21;

GRANT ALL PRIVILEGES
   ON LAREPS
   TO SUE;
```

Ларри является владельцем представления LAREPS, но не является владельцем представления WESTREPS, на основании которого создается новое представление. Для эффективного обеспечения безопасности требуется, чтобы, прежде чем Ларри мог дать Сью привилегию SELECT для работы с представлением LAREPS, он имел не только привилегию SELECT для работы с представлением WESTREPS, но и право выдачи этой привилегии.

Если пользователь имеет какие-либо привилегии с указанием GRANT OPTION, то он может давать другим пользователям не только сами эти привилегии, но *и* право их передачи другим. Те, в свою очередь, могут дальше передавать привилегии также с правом передачи. По этой причине следует очень осторожно использовать возможность передачи привилегий. Обратите внимание: право передачи касается только тех привилегий, которые указаны в инструкции GRANT. Если вы хотите дать кому-либо одни привилегии с правом передачи, а другие — без, то должны воспользоваться двумя отдельными инструкциями GRANT.

Разрешить Ларри Фитчу извлекать, добавлять, обновлять и удалять данные из представления WESTREPS, а также позволить ему предоставлять другим пользователям разрешение на выборку данных.

```
GRANT SELECT
   ON WESTREPS
   TO LARRY
 WITH GRANT OPTION;

GRANT INSERT, DELETE, UPDATE
   ON WESTREPS
   TO LARRY;
```

Отмена привилегий (REVOKE)

В большинстве реляционных баз данных привилегии, предоставленные посредством инструкции GRANT, могут быть отобраны с помощью инструкции REVOKE, синтаксическая диаграмма которой изображена на рис. 15.7. Инструкция REVOKE, имеющая структуру, аналогичную структуре инструкции GRANT, содержит набор отбираемых привилегий, объект, к которому относятся эти привилегии, и идентификаторы пользователей, у которых отбираются привилегии.

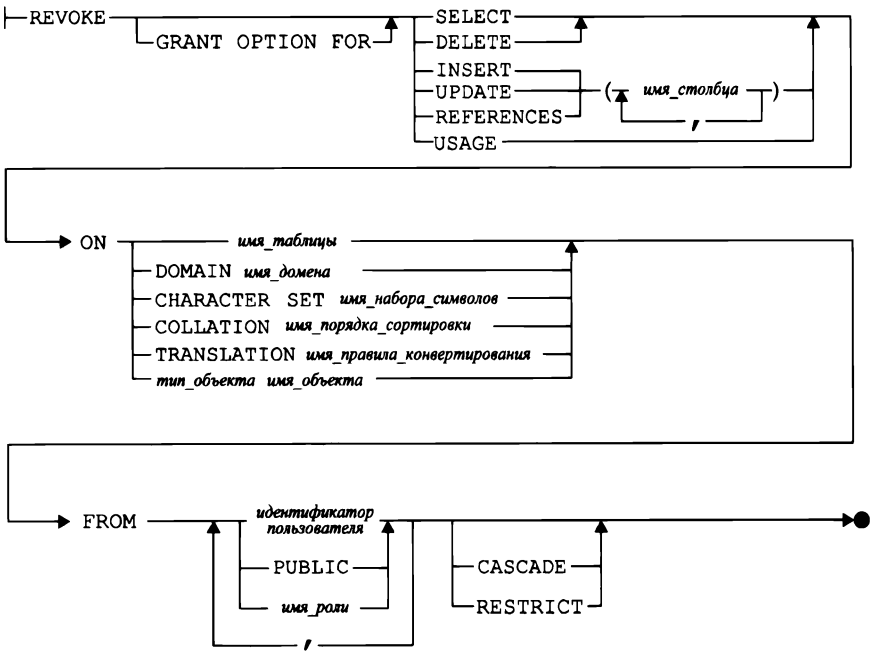


Рис. 15.7. Синтаксическая диаграмма инструкции REVOKE

Посредством инструкции REVOKE можно отобрать все или только некоторые из привилегий, предоставленных ранее пользователю. Например, рассмотрим такую последовательность инструкций.

Предоставить некоторые привилегии для работы с таблицей SALESREPS, а затем отменить часть из них.

```
GRANT SELECT, INSERT, UPDATE
   ON SALESREPS
   TO ARUSER, OPUSER;

REVOKE INSERT, UPDATE
    ON SALESREPS
  FROM OPUSER;
```

Сначала привилегии INSERT и UPDATE для работы с таблицей SALESREPS даются двум пользователям, а затем у одного из них отбираются. Однако привилегия SELECT остается у обоих пользователей. Вот еще несколько примеров инструкций REVOKE.

Отобрать все привилегии, предоставленные ранее для работы с таблицей OFFICES.

```
REVOKE ALL PRIVILEGES
    ON OFFICES
  FROM ARUSER;
```

Отобрать привилегии UPDATE и DELETE у двух пользователей.

```
REVOKE UPDATE, DELETE
    ON OFFICES
  FROM ARUSER, OPUSER;
```

Отобрать у всех пользователей все предоставленные ранее привилегии для работы с таблицей OFFICES.

```
REVOKE ALL PRIVILEGES
    ON OFFICES
  FROM PUBLIC;
```

Посредством инструкции REVOKE вы можете отобрать только те привилегии, которые *вы* предоставили ранее некоторому пользователю. У него могут быть также привилегии, предоставленные другими пользователями; ваша инструкция REVOKE на них не влияет. Обратите особое внимание на то, что если два разных пользователя предоставляют третьему пользователю одну и ту же привилегию на один и тот же объект, а затем один из них отменяет привилегию, то вторая привилегия остается в силе и по-прежнему разрешает пользователю доступ к объекту. Результат такого "перекрытия привилегий" иллюстрируется следующим примером.

Предположим, что вице-президент по сбыту Сэм Кларк предоставляет Ларри Фитчу привилегию SELECT для работы с таблицей SALESREPS и привилегии SELECT и UPDATE для работы с таблицей ORDERS, используя следующие инструкции.

```
GRANT SELECT
   ON SALESREPS
   TO LARRY;

GRANT SELECT, UPDATE
   ON ORDERS
   TO LARRY;
```

Через несколько дней вице-президент по маркетингу Джордж Уоткинс предоставляет Ларри привилегии SELECT и DELETE для работы с таблицей ORDERS и привилегию SELECT для работы с таблицей CUSTOMERS, используя такие инструкции.

```
GRANT SELECT, DELETE
   ON ORDERS
   TO LARRY;

GRANT SELECT
   ON CUSTOMERS
   TO LARRY;
```

Заметьте, что Ларри получил привилегии для работы с таблицей ORDERS из двух различных источников. Причем привилегию SELECT для работы с таблицей ORDERS он получил из обоих источников. Несколькими днями позже Сэм отменяет привилегии, предоставленные недавно Ларри для работы с таблицей ORDERS.

```
REVOKE SELECT, UPDATE
    ON ORDERS
  FROM LARRY;
```

После выполнения этой инструкции СУБД у Ларри остается привилегия SELECT для работы с таблицей SALESREPS, привилегии SELECT и DELETE для работы с таблицей ORDERS и привилегия SELECT для работы с таблицей CUSTOMERS; но он теряет привилегию UPDATE для работы с таблицей ORDERS.

REVOKE и GRANT OPTIONS

Когда вы даете кому-нибудь привилегии с правом последующего предоставления, а затем отменяете эти привилегии, то в большинстве СУБД *автоматически* отменяются все привилегии, которые являются производными от исходных. Рассмотрим еще раз цепочку привилегий, представленных на рис. 15.6, идущую от вице-президента по сбыту Сэма Кларка к менеджеру лос-анжелесского офиса Ларри Фитчу, а затем к Сью Смит. Если теперь Сэм отменит привилегии Ларри для работы с представлением WESTREPS, то привилегии Сью также будут автоматически отменены.

Ситуация становится более сложной, если привилегии предоставляются двумя или более пользователями, один из которых впоследствии отменяет привилегии. Рассмотрим рис. 15.8. Он представляет собой предыдущий пример в слегка измененном виде. Здесь Ларри получает привилегию SELECT с правом передачи *как* от Сэма (вице-президента по сбыту), *так* и от Джорджа (вице-президента по маркетингу), а затем дает привилегии Сью. На этот раз, когда Сэм отменяет привилегии Ларри, остаются привилегии, предоставленные Джорджем. Привилегии Сью также остаются, поскольку они могут происходить от привилегий Джорджа.

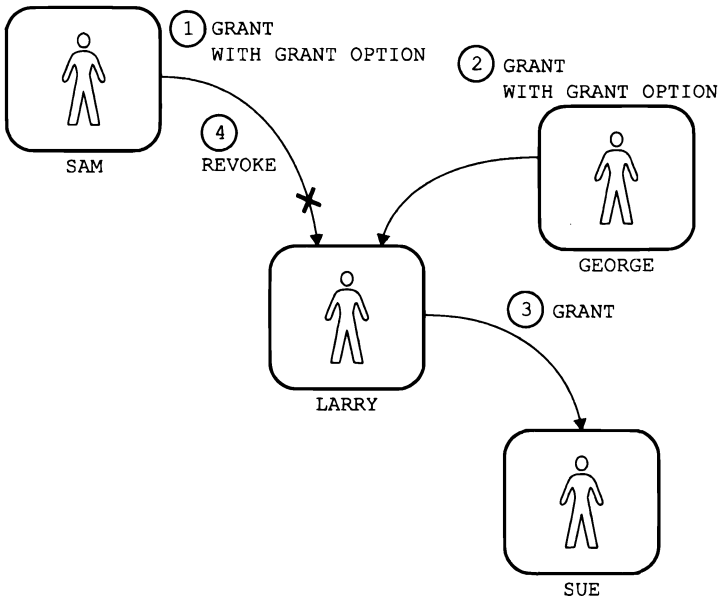


Рис. 15.8. Отмена привилегий, предоставленных двумя пользователями

Рассмотрим другой вариант этой цепочки привилегий (рис. 15.9), в котором изменен порядок событий. Здесь Ларри получает привилегию с правом передачи от Сэма, дает эту привилегию Сью и *после этого* получает привилегию с правом передачи от Джорджа. Когда на этот раз Сэм отменяет привилегии Ларри, результат будет другим и может отличаться у разных СУБД. Как и в случае, изображенном на рис. 15.8, у Ларри останется привилегия SELECT для работы с представлением WESTREPS, предоставленная Джорджем. Но в СУБД DB2 и SQL/DS Сью автоматически потеряет привилегию SELECT. Почему? Потому что привилегия Сью явно происходит от привилегии, которая предоставлена Ларри Сэмом и только что отменена. Она не может быть производной от привилегии, предоставленной Ларри Джорджем, так как эта привилегия отсутствовала в тот момент, когда Ларри давал привилегию Сью.

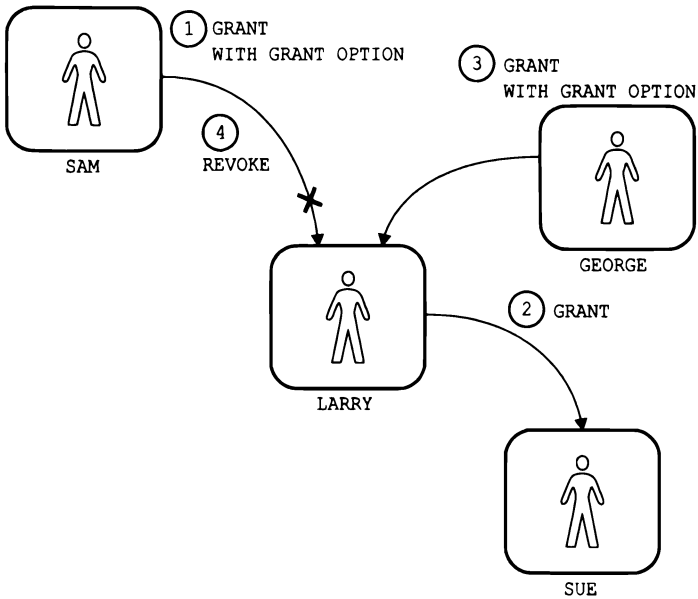


Рис. 15.9. Отмена привилегий, предоставленных в другой последовательности

В других СУБД привилегия Сью могла бы сохраниться, так как у Ларри сохранилась привилегия, предоставленная Джорджем. Таким образом, то, как далеко распространятся последствия выполнения инструкции REVOKE, зависит не только от самих привилегий, но и от хронологической последовательности инструкций GRANT и REVOKE. Для получения желаемых результатов выдача привилегий с правом передачи и аннулирование таких привилегий должны выполняться очень осторожно.

Следует не упускать из виду еще одно — дополнительные расходы на обработку каскадных отмен привилегий. При излишнем использовании GRANT OPTION отмены привилегий могут привести к серьезным проблемам с производительностью.

REVOKE и стандарт ANSI/ISO

Стандарт SQL1 определяет инструкцию GRANT как часть языка определения данных (DDL). Вспомним из главы 13, "Создание базы данных", что стандарт SQL1 рассматривает DDL как отдельное, статическое определение базы данных и не требует, чтобы в СУБД существовала возможность динамического изменения структуры базы данных. Такой же подход используется по отношению к безопасности базы данных. Согласно стандарту SQL1, возможность доступа к таблицам и представлениям определяется набором инструкций GRANT, входящих в схему базы данных. После того как создано определение базы данных, изменить ее схему невозможно. Поэтому инструкция REVOKE, так же как инструкция DROP TABLE, в стандарте SQL1 отсутствует.

Несмотря на это, инструкция REVOKE присутствует практически во всех коммерческих СУБД на базе SQL с самых первых их версий. Как и в случае с инструкциями DROP и ALTER, стандарт инструкции REVOKE был разработан позднее на основе диалекта SQL, используемого в DB2. Начиная с версии SQL2, в стандарт вошла спецификация инструкции REVOKE, образцом для которой, с некоторыми расширениями, послужила одноименная инструкция из DB2. Одно из изменений дает пользователю возможность осуществлять явный контроль над отменой привилегий, предоставленных, в свою очередь, другим пользователям. Другое изменение позволяет отменять право передачи привилегий, не отменяя при этом сами привилегии.

Стандарт SQL требует, чтобы в инструкции REVOKE присутствовал параметр CASCADE или RESTRICT, определяющий, каким образом должна производиться отмена привилегий, предоставленных другим пользователям. (Подобное требование, как упоминалось в главе 13, "Создание базы данных", предъявляется в стандарте SQL и ко многим инструкциям семейства DROP.) Предположим, что Ларри были даны привилегии SELECT и UPDATE для работы с таблицей ORDERS с правом дальнейшей передачи и что Ларри передал эти привилегии Биллу. Тогда инструкция REVOKE

```
REVOKE SELECT, UPDATE
    ON ORDERS
  FROM LARRY CASCADE;
```

отменяет привилегии не только Ларри, но и Билла. Таким образом, действие этой инструкции REVOKE распространяется на всех остальных пользователей, чьи привилегии являются производными от первоначальной инструкции GRANT.

Теперь предположим, что при тех же обстоятельствах выполняется инструкция REVOKE

```
REVOKE SELECT, UPDATE
    ON ORDERS
  FROM LARRY RESTRICT;
```

В этом случае инструкция REVOKE выполнится неуспешно. Параметр RESTRICT означает, что СУБД не должна выполнять эту инструкцию, если в базе данных имеются привилегии, производные от удаляемой. Сообщение об ошибке привлекает внимание пользователя к тому факту, что имеются (возможно, непреднаме-

ренные) побочные эффекты выполнения инструкции REVOKE, и дает возможность пересмотреть решение. Если пользователь все же решит отменить привилегии, то он может воспользоваться параметром CASCADE.

Новейшая версия инструкции REVOKE обеспечивает более детальный контроль над привилегиями и правом их предоставления. Вновь предположим, что Ларри получил привилегии для работы с таблицей ORDERS с правом их передачи. Обычная инструкция REVOKE для этих привилегий

```
REVOKE SELECT, UPDATE
    ON ORDERS
  FROM LARRY;
```

отменяет как привилегии, так и право передачи их другим пользователям. Стандарт SQL2 допускает такой вариант инструкции REVOKE.

```
REVOKE GRANT OPTION FOR SELECT, UPDATE
    ON ORDERS
  FROM LARRY CASCADE;
```

После успешного выполнения этой инструкции Ларри потеряет право передавать привилегии другим пользователям, но сам свои привилегии сохранит. Как и ранее, стандарт SQL требует наличия параметра CASCADE или RESTRICT для указания способа выполнения инструкции, если Ларри, в свою очередь, передал право предоставления привилегий другим пользователям.

Безопасность на основе ролей

Управление привилегиями отдельных пользователей может оказаться весьма монотонной и утомительной работой. В связи с этим в стандарт SQL была добавлена концепция ролей. Вспомним, что *роль* — это просто именованный набор привилегий. В большинстве современных реализаций SQL роли могут предоставляться отдельным идентификаторам пользователей точно так же, как и отдельные привилегии. Более того, большинство реализаций SQL поставляется с набором предопределенных ролей. Например, привилегии, которые обычно необходимы для работы администратору базы данных, зачастую предоставляются поставщиком СУБД в виде роли.

Преимущества использования ролей следующие.

- Роли могут существовать до идентификаторов пользователей. Например, мы можем создать роль для отдела обработки заказов, вместо того чтобы предоставлять всем его сотрудникам единый идентификатор пользователя OPUSER, как показано на рис. 15.2. Когда в отдел приходит новый сотрудник, одна инструкция GRANT с указанием роли предоставляет ему все необходимые для работы в отделе привилегии.
- Роли в состоянии "пережить" удаление пользователей. Администратору не надо беспокоиться об отслеживании потерь привилегий при удалении некоторого пользователя. Например, если привилегии вице-президента по сбыту Сэма Кларка получены им при помощи роли, то список приви-

легий данной роли останется неизменным даже после увольнения Сэма из компании и удаления его учетной записи. Роль может быть легко передана другому человеку, принятому на эту должность.

- Роли поддерживают стандартные привилегии. При применении ролей в организации легко обеспечить одинаковые привилегии для всех людей, выполняющих одинаковую работу.
- Роли устраняют монотонную и утомительную работу по предоставлению привилегий отдельным пользователям. Роли позволяют предоставлять множество привилегий одной простой командой. При добавлении привилегий к роли или удалении их из нее все изменения немедленно отражаются на всех пользователях, которым предоставлена данная роль.

Создание и назначение привилегий при помощи ролей выполняется очень просто, если вы знакомы с инструкциями GRANT и REVOKE. Обратите внимание: на рис. 15.5 инструкция GRANT допускает указание имени роли в предложении TO вместо идентификатора пользователя или ключевого слова PUBLIC.

Аналогично имя роли может использоваться в предложении FROM инструкции REVOKE, как показано на рис. 15.7. Вот несколько конкретных примеров.

Создать роль OPUSER для пользователей из отдела обработки заказов.

```
CREATE ROLE OPUSER;
```

Назначить роли привилегии, необходимые всем пользователям этой роли.

```
GRANT SELECT, INSERT, DELETE, UPDATE
   ON ORDERS
   TO OPUSER;

GRANT SELECT
   ON CUSTOMERS
   TO OPUSER;

GRANT UPDATE (COMPANY, CUST_REP)
   ON CUSTOMERS
   TO OPUSER;

GRANT SELECT
   ON SALESREPS
   TO OPUSER;
```

Предоставить роль пользователям Julio, Sumit и Yolanda из отдела обработки заказов.

```
GRANT OPUSER
   TO JULIO, SUMIT, YOLANDA;
```

Предоставить право UPDATE для работы с таблицей SALESREPS роли OPUSER. Обратите внимание, что все три текущих пользователя с данной ролью немедленно получат новую привилегию.

```
GRANT UPDATE
   ON SALESREPS
   TO OPUSER;
```

Предоставить роль OPUSER новому сотруднику отдела обработки заказов — Francois. Обратите внимание, что он немедленно получит все привилегии данной роли.

```
GRANT OPUSER
   TO FRANCOIS;
```

Отменить роль OPUSER пользователя Yolanda, который переведен в другой отдел.

```
REVOKE OPUSER
  FROM YOLANDA;
```

Заметим, что поддержка ролей в разных реализациях SQL несколько отличается. Например, на момент написания книги СУБД MySQL роли не поддерживала. В Oracle пользователь для создания новых ролей должен иметь привилегию CREATE ROLE.

Резюме

Безопасность реляционной базы данных обеспечивается с помощью языка SQL.

- Схема защиты данных в SQL основана на привилегиях (разрешенных действиях), предоставляемых имеющим отдельные идентификаторы пользователям (или группам пользователей) для работы с отдельными объектами базы данных (например, таблицами или представлениями).
- Представления играют ключевую роль в защите данных, так как они могут применяться для ограничения доступа к строкам и столбцам таблицы.
- Инструкция GRANT используется для предоставления привилегий; если пользователь получил привилегии с правом передачи, то, в свою очередь, он может предоставлять их другим пользователям.
- Инструкция REVOKE применяется для отмены привилегий, предоставленных ранее посредством инструкции GRANT.
- Роли могут применяться для создания списков привилегий, которые затем могут быть предоставлены пользователю (или забраны у него) одной командой.

16
ГЛАВА

Системный каталог

Чтобы осуществлять управление данными, СУБД должна отслеживать большое количество информации, определяющей структуру базы данных. В реляционной базе данных эта информация обычно хранится в *системном каталоге* — совокупности системных таблиц, используемых СУБД для собственных целей. Информация, хранящаяся в системном каталоге, описывает таблицы, представления, столбцы, привилегии и другие структурные элементы базы данных.

Хотя системный каталог предназначен, главным образом, для внутреннего применения, пользователи базы данных также могут получить доступ к системным таблицам с помощью стандартных запросов SQL. Таким образом, реляционная база представляет собой самоописываемую сущность; вы можете получить описание структуры базы данных, выполняя запросы к системным таблицам. Это используется в таких клиентских приложениях общего назначения, как модули формирования запросов и программы генерации отчетов, где для упрощения доступа к базе данных пользователям предоставляется на выбор список таблиц и столбцов.

В настоящей главе описываются системные каталоги нескольких популярных СУБД на базе SQL и содержащаяся в этих каталогах информация. Рассматривается также структура системного каталога в стандарте ANSI/ISO SQL.

Что такое системный каталог

Системным каталогом называется совокупность специальных таблиц базы данных. Их создает, сопровождает и владеет ими сама СУБД. Эти *системные таблицы* содержат информацию, которая описывает структуру базы данных. Таблицы системного каталога создаются автоматически при создании базы данных. Обычно они объединяются под специальным системным идентификатором пользователя с таким именем, как SYSTEM, SYSIBM, MASTER или DBA.

При обработке инструкций SQL СУБД постоянно обращается к данным системного каталога. Например, чтобы обработать двухтабличную инструкцию SELECT, СУБД должна выполнить следующее:

- проверить, существуют ли две указанные таблицы;
- убедиться, что пользователь имеет разрешение на доступ к ним;
- проверить, существуют ли столбцы, на которые имеются ссылки в данном запросе;
- разрешить неквалифицированные имена и установить, к каким таблицам они относятся;
- определить тип данных каждого столбца.

Так как информация о структуре базы данных хранится в системных таблицах, СУБД может использовать собственные методы и алгоритмы, чтобы быстро и эффективно извлекать информацию, необходимую для выполнения этих задач.

Если бы системные таблицы служили только для удовлетворения внутренних потребностей СУБД, то для пользователей базы данных они не представляли бы практически никакого интереса. Однако системные таблицы (или созданные на их основе представления), как правило, доступны также и для пользователей. Запросы к системным каталогам разрешены почти во всех базах данных для персональных и мини-компьютеров. В СУБД для мэйнфреймов или СУБД уровня предприятия такие запросы тоже допускаются, но администратор базы данных может ограничивать доступ к системному каталогу в качестве дополнительной меры обеспечения безопасности базы данных. С помощью запросов к системным каталогам вы можете получить информацию о структуре базы данных, даже если никогда раньше с ней не работали.

Пользователи могут только извлекать информацию из системного каталога. СУБД запрещает пользователям модифицировать системные таблицы непосредственно, так как это может нарушить целостность базы данных. СУБД сама вставляет, удаляет и обновляет строки системных таблиц во время модификации структуры базы данных. Изменения в системных таблицах происходят также в качестве побочного результата выполнения таких инструкций DDL, как CREATE, ALTER, DROP, GRANT и REVOKE. В некоторых СУБД даже инструкции DML, например INSERT и DELETE, могут модифицировать содержимое системных таблиц, отслеживающих количество записей в таблицах.

Системный каталог и средства формирования запросов

Одним из наиболее важных преимуществ системного каталога является то, что он позволяет создавать дружественные к пользователю программы формирования запросов, такие как Query Builder, являющаяся частью Oracle Application Express, поставляемой в составе Oracle Express Edition (рис. 16.1). Цель такой программы заключается в том, чтобы обеспечить простой и понятный способ доступа к базе данных пользователю, не знающему языка SQL. Обычно эта программа проводит пользователя через следующую последовательность действий.

1. Пользователь вводит свое имя и пароль, чтобы получить доступ к базе данных.
2. Программа формирования запросов отображает на экране список доступных таблиц.
3. Пользователь выбирает таблицу, после чего программа отображает на экране список столбцов данной таблицы.
4. Пользователь выбирает интересующие его столбцы — обычно при помощи щелчков мышью на именах столбцов, когда они появляются на экране.
5. Пользователь выбирает столбцы из других таблиц или ограничивает объем извлекаемых данных с помощью условия отбора.
6. Программа извлекает запрашиваемые данные и отображает их на экране.

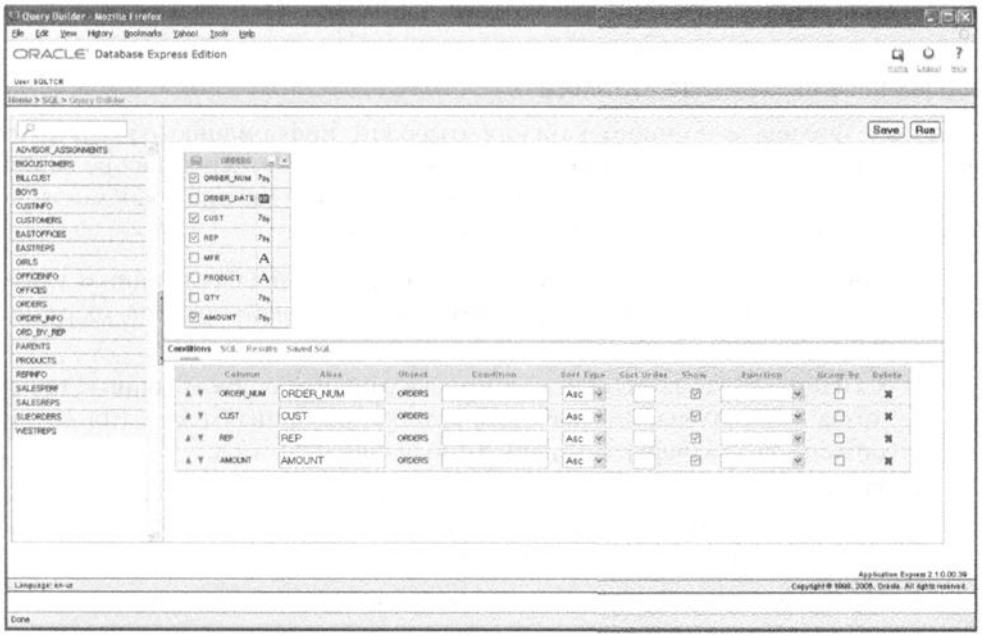


Рис. 16.1. Oracle Query Builder — пример программы формирования запросов с дружественным интерфейсом

Программа формирования запросов общего назначения, подобная той, что представлена на рис. 16.1, будет применяться многими пользователями для доступа к самым различным базам данных. Ей неизвестна заранее структура базы данных в каждом конкретном случае, поэтому необходимо иметь возможность динамически получать информацию о таблицах и столбцах базы данных. Для достижения этой цели такая программа использует системный каталог.

Системный каталог и стандарт ANSI/ISO

В стандарте SQL1 ничего не говорится о структуре и содержании системного каталога. Стандарт фактически не требует даже наличия самого системного ката-

лога. Однако во всех основных СУБД на базе SQL в той или иной форме он создается. Структура каталога и содержащиеся в нем таблицы значительно отличаются друг от друга в разных СУБД.

В связи с ростом популярности инструментальных программ общего назначения, предназначенных для работы с базами данных и требующих доступа к системному каталогу, в стандарт SQL (начиная с SQL2) включена спецификация набора представлений, обеспечивающая стандартизированный доступ к информации, которая обычно содержится в системном каталоге. СУБД, соответствующая стандарту SQL, должна поддерживать эти представления, все вместе именуемые INFORMATION_SCHEMA. Так как эта схема сложнее, чем реальные системные каталоги, применяемые в большинстве коммерческих СУБД, и пока еще не пользуется широкой поддержкой, она рассматривается в отдельном разделе в конце данной главы.

Содержимое системного каталога

Каждая таблица системного каталога содержит информацию об отдельном структурном элементе базы данных. В состав почти всех коммерческих реляционных СУБД входят, с небольшими различиями, системные таблицы, каждая из которых описывает один из следующих пяти элементов.

- **Таблицы**. В каталоге описывается каждая таблица базы данных: указывается ее имя, владелец, число содержащихся в ней столбцов, их размер и т.д.
- **Столбцы**. В каталоге описывается каждый столбец базы данных: приводится имя столбца, имя таблицы, которой он принадлежит, тип данных столбца, его размер, разрешены ли значения NULL и т.д.
- **Пользователи**. В каталоге описывается каждый зарегистрированный пользователь базы данных: указываются имя пользователя, его пароль в зашифрованном виде и другие данные.
- **Представления**. В каталоге описывается каждое представление, имеющееся в базе данных: указываются его имя, имя владельца, запрос, являющийся определением представления, и т.д.
- **Привилегии**. В каталоге описывается каждый набор привилегий, предоставляемых в базе данных: приводятся имена тех, кто предоставил привилегии, и тех, кому они предоставлены, указываются сами привилегии, объект, на которые они распространяются, и т.д.

В табл. 16.1 для основных реляционных СУБД даны имена системных таблиц, содержащих эту информацию. Некоторые типичные системные таблицы описываются более подробно далее; приводятся также примеры запросов к системному каталогу.

Таблица 16.1. Избранные системные таблицы популярных СУБД

СУБД	Таблицы	Столбцы	Пользователи	Представления	Привилегии
DB2[1]	SCHEMATA	COLUMNS	DBAUTH	VIEWS	DBAUTH
	TABLES	KEYCOLUSE			SCHEMAAUTH
	REFERENCES	COLOPTIONS			TABAUTH
	TABOPTIONS				COLAUTH
	TABDEP				
Oracle[2]	CATALOG	TAB_COLUMNS	USERS	VIEWS	TAB_PRIVS
	OBJECTS	TAB_COLS			COL_PRIVS
	TABLES	LOBS			SYS_PRIVS
	SYNONYMS				
Informix	SYSTABLES	SYSCOLUMNS	SYSUSERS	SYSVIEWS	SYSTABAUTH
	SYSREFERENCES			SYSDEPEND	SYSCOLAUTH
	SYSYNONYMS				
Sybase	SYSDATABASES	SYSCOLUMNS	SYSUSERS	SYSOBJECTS	
	SYSOBJECTS			SYSCOMMENTS	
	SYSKEYS				
SQL Server[3]	DATABASES	COLUMNS	DATABASE_ PRINCIPALS	OBJECTS	DATABASE_ PERMISSIONS
	OBJECTS	FOREIGN_KEY_ COLUMNS	SQL_LOGINS	VIEWS	
	FOREIGN_KEYS	IDENTITY_ COLUMNS			
	REFERENCES				

[1] *Таблицы DB2 используют квалификатор* `SYSCAT` *(например,* `SYSCAT.TABLES`*).*

[2] *Oracle предоставляет три версии многих представлений каталога с префиксами* `ALL_`, `DBA_` *или* `USER_` *(например,* `ALL_TABLES`, `DBA_TABLES` *или* `USER_TABLES`*). Версия* `ALL_` *показывает все объекты, к которым текущий пользователь имеет доступ,* `DBA_` *— все объекты во всей базе данных, а* `USER_` *— только объекты, владельцем которых является текущий пользователь.*

[3] *Представления каталога SQL Server имеют квалификатор* `SYS` *(например,* `SYS.DATABASES`*). Начиная с SQL Server 2000 происходит отказ от системных таблиц в пользу новых представлений каталога.*

Информация о таблицах

Во всех основных реляционных СУБД имеется системная таблица или представление, где отслеживается состояние всех таблиц базы данных. В DB2 это представление называется `SYSCAT.TABLES`. (Все системные представления в СУБД DB2 входят в схему, называющуюся `SYSCAT`, поэтому обладают именами вида `SYSCAT.XXX`)

В табл. 16.2 перечислены некоторые столбцы представления SYSCAT.TABLES. В нем для каждой таблицы, каждого представления или псевдонима, имеющихся в базе данных, отводится одна строка. Примерно такая же информация предоставляется в соответствующих системных таблицах других СУБД.

Таблица 16.2. Избранные столбцы таблицы SYSCAT.TABLES (СУБД DB2)

Столбец	Тип данных	Информация
TABSCHEMA	VARCHAR(128)	Схема, содержащая таблицу, представление или псевдоним
TABNAME	VARCHAR(128)	Имя таблицы, представления или псевдонима
DEFINER	VARCHAR(128)	Идентификатор создателя таблицы, представления или псевдонима
TYPE	CHAR(1)	Тип: 'T' — таблица, 'V' — представление, 'A' — псевдоним, 'H' — иерархическая таблица, 'S' — материализованный запрос, 'U' — типизированная таблица, 'W' — типизированное представление
STATUS	CHAR(1)	Статус объекта (используется системой)
DROPRULE	CHAR(1)	'N' — правила нет, 'R' — правило удаления RESTRICT
BASE_TABSCHEMA	VARCHAR(128)	Схема, содержащая базовую таблицу, на которую ссылается псевдоним
BASE_TABNAME	VARCHAR(128)	Имя базовой таблицы, на которую ссылается псевдоним
ROWTYPESCHEMA	VARCHAR(128)	Имя схемы для типа строк данной таблицы
ROWTYPENAME	VARCHAR(18)	Имя типа строк данной таблицы
CREATE_TIME	TIMESTAMP	Время создания объекта
STATS_TIME	TIMESTAMP	Время, когда последний раз вычислялись статистические данные объекта
COLCOUNT	SMALLINT	Количество столбцов в таблице
TABLEID	SMALLINT	Внутренний идентификатор таблицы
TBSPACEID	SMALLINT	Идентификатор первичного табличного пространства, в котором хранится таблица
CARD	INTEGER	Количество записей в таблице
NPAGES	INTEGER	Число страниц дисковой памяти, содержащих данные таблицы
FPAGES	INTEGER	Общее число страниц дисковой памяти, занимаемых таблицей
OVERFLOW	INTEGER	Число переполнения записей таблицы
TBSPACE	VARCHAR(18)	Первичное табличное пространство таблицы
INDEX_TBSPACE	VARCHAR(18)	Табличное пространство, в котором хранятся индексы таблицы
LONG_TBSPACE	VARCHAR(18)	Табличное пространство, в котором хранятся большие двоичные объекты
PARENTS	SMALLINT	Число родительских таблиц для данной таблицы
CHILDREN	SMALLINT	Число дочерних таблиц для данной таблицы

Окончание табл. 16.2

Столбец	Тип данных	Информация
SELFREFS	SMALLINT	Число ссылок таблицы на саму себя
KEYCOLUMNS	SMALLINT	Число столбцов в первичном ключе таблицы
KEYINDEXID	SMALLINT	Внутренний идентификатор первичного ключа таблицы
KEYUNIQUE	SMALLINT	Число ограничений уникальности на значения столбцов таблицы
CHECKCOUNT	SMALLINT	Число ограничений на значения столбцов таблицы
DATACAPTURE	CHAR(1)	Признак реплицированной таблицы
CONST_CHECKED	CHAR(32)	Флаги проверки ограничений
PMAP_ID	SMALLINT	Внутренний идентификатор схемы физической сегментации таблицы
PARTITION_MODE	CHAR(1)	Режим сегментации таблиц базы данных
LOG_ATTRIBUTE	CHAR(1)	Признак того, что для таблицы по умолчанию установлен режим ведения журнала
PCTFREE	SMALLINT	Часть страниц (в процентах), которую следует зарезервировать для будущих данных
REMARKS	VARCHAR(254)	Пользовательские примечания к таблице

С помощью таких запросов, как приведенные ниже, можно получить информацию о таблицах в СУБД DB2. Используя аналогичные запросы с другими именами таблиц и столбцов, можно получить ту же самую информацию в СУБД других типов.

Имена всех таблиц базы данных, а также имена владельцев этих таблиц.

```
SELECT DEFINER, TABNAME
  FROM SYSCAT.TABLES
 WHERE TYPE = 'T';
```

Имена всех таблиц, представлений и псевдонимов, имеющихся в базе данных.

```
SELECT TABNAME
  FROM SYSCAT.TABLES;
```

Имена и даты создания моих таблиц.

```
SELECT TABNAME, CREATE_TIME
  FROM SYSCAT.TABLES
 WHERE TYPE = 'T'
   AND DEFINER = USER;
```

В СУБД Oracle аналогичные функции выполняют системные представления ALL_TABLES, DBA_TABLES и USER_TABLES, описанные в табл. 16.3. Представление ALL_TABLES содержит по одной строке для каждой таблицы, к которой текущий пользователь имеет по крайней мере одну привилегию доступа. Представление DBA_TABLES содержит по строке для каждой таблицы базы данных. DBA-представления (т.е. те, имена которых начинаются на DBA_) обычно доступны только тем пользователям, которые имеют в базе данных высший уровень привилегий, таким как администратор базы данных. Представление USER_TABLES со-

держит по одной строке для каждой таблицы, принадлежащей текущему пользователю. Все три представления содержат одинаковые столбцы, за исключением столбца OWNER, отсутствующего в представлении USER_TABLES.

Таблица 16.3. Избранные столбцы представления каталога ALL_TABLES, DBA_TABLES и USER_TABLES СУБД Oracle

Столбец	Тип данных	Информация
OWNER	VARCHAR2(30)	Владелец таблицы (отсутствует в USER_TABLES)
TABLE_NAME	VARCHAR2(30)	Имя таблицы
TABLESPACE_NAME	VARCHAR2(30)	Имя табличного пространства, содержащего таблицу; NULL для сегментированных, временных и индексно-организованных таблиц
CLUSTER_NAME	VARCHAR2(30)	Имя кластера, к которому относится таблица (если таковой имеется)
IOT_NAME	VARCHAR2(30)	Имя индексно-организованной таблицы (если таковая имеется), к которой относятся записи с переполнением или таблицы отображения
STATUS	VARCHAR2(8)	Указывает, корректна таблица (VALID) или нет (UNUSABLE)
PCT_FREE	NUMBER	Минимальный процент свободного пространства в блоке; NULL для секционированных таблиц
PCT_USED	NUMBER	Минимальный процент используемого пространства в блоке; NULL для секционированных таблиц
INI_TRANS	NUMBER	Изначальное количество транзакций; NULL для секционированных таблиц
MAX_TRANS	NUMBER	Максимальное количество транзакций; NULL для секционированных таблиц
INITIAL_EXTENT	NUMBER	Размер начального экстента в байтах; NULL для секционированных таблиц
NEXT_EXTENT	NUMBER	Размер вторичных экстентов в байтах; NULL для секционированных таблиц
MIN_EXTENTS	NUMBER	Минимально допустимое количество экстентов в сегменте; NULL для секционированных таблиц
MAX_EXTENTS	NUMBER	Максимально допустимое количество экстентов в сегменте; NULL для секционированных таблиц
NUM_ROWS	NUMBER	Количество строк в таблице (NULL при обновлении статистики)
BLOCKS	NUMBER	Количество используемых блоков данных в таблице (NULL при обновлении статистики)
EMPTY_BLOCKS	NUMBER	Количество блоков в таблице, которые никогда не использовались (NULL при обновлении статистики)
AVG_SPACE	NUMBER	Среднее количество свободного пространства в байтах в блоке данных, выделенном таблице
PARTITIONED	VARCHAR2(3)	Указывает, секционирована таблица (YES) или нет (NO)
TEMPORARY	VARCHAR2(1)	Указывает, временная это таблица (Y) или нет (N)

Вот типичные запросы к представлениям системного каталога Oracle.

Имена и владельцы всех таблиц, к которым имеет доступ текущий пользователь.

```
SELECT TABLE_NAME, OWNER
  FROM ALL_TABLES;
```

Имена и владельцы всех таблиц базы данных.

```
SELECT TABLE_NAME, OWNER
  FROM DBA_TABLES;
```

Имена всех таблиц текущего пользователя.

```
SELECT TABLE_NAME
  FROM USER_TABLES;
```

В СУБД SQL Server эквивалентом представления SYSCAT.TABLES из DB2 является представление каталога SYS.OBJECTS (табл. 16.4). Представление SYS.OBJECTS хранит информацию о таблицах, представлениях и других объектах SQL Server, таких как хранимые процедуры, правила и триггеры. Обратите также внимание на то, что для идентификации владельца таблицы в представлении SYS.OBJECTS используется внутренний идентификатор пользователя (principal_id) вместо его имени.

Таблица 16.4. Столбцы представления SYS.OBJECTS каталога SQL Server

Имя столбца	Тип данных	Информация
Name	sysname	Имя объекта
object_id	int	Внутренний идентификатор объекта
schema_id	int	Идентификатор схемы, содержащей объект
principal_id	int	Идентификатор владельца индивидуального объекта (если он отличается от владельца схемы)
parent_object_id	int	Идентификатор объекта, которому принадлежит данный объект (0, если объект не является дочерним)
type	char(2)	Тип объекта: C — ограничение CHECK D — ограничение DEFAULT F — ограничение FOREIGN KEY P — хранимая процедура PK — ограничение PRIMARY KEY S — системная таблица TR — триггер U — пользовательская таблица (плюс много других значений)
create_date	datetime	Дата и время создания объекта
modify_date	datetime	Дата и время последнего изменения объекта
is_ms_shipped	bit	Был объект создан внутренним компонентом SQL Server или нет
is_published	bit	Опубликован объект или нет
is_schema_published	bit	Опубликована только схема объекта или нет

В Informix Universal Server системная таблица, содержащая информацию о таблицах базы данных, называется SYSTABLES. Как и каталог DB2, она содержит информацию только о таблицах, представлениях и псевдонимах; другие объекты базы данных описываются в других системных таблицах. Вот типичный запрос к системной таблице Informix.

Имена, владельцы и даты создания всех таблиц базы данных.

```
SELECT TABNAME, OWNER, CREATED
  FROM SYSTABLES
 WHERE TABTYPE = 'T';
```

Как показывают эти примеры, запросы на получение информации о таблицах имеют похожую структуру в различных СУБД. Тем не менее конкретные имена системных таблиц или представлений, а также их столбцов могут существенно отличаться.

Информация о столбцах

Во всех основных реляционных СУБД имеется системная таблица, в которой отслеживается состояние столбцов базы данных. В этой таблице отводится одна строка для каждого столбца каждой таблицы или представления базы данных. Большинство СУБД ограничивает доступ пользователей к этой таблице, вместо этого создавая системное представление, описывающее столбцы только тех таблиц, которыми владеет пользователь или к которым он имеет доступ. В Oracle эта информация содержится в трех представлениях системного каталога — USER_TAB_COLUMNS, включающем одну строку для каждого столбца в каждой таблице, принадлежащей текущему пользователю; ALL_TAB_COLUMNS, включающем одну строку для каждого столбца каждой таблицы, для работы с которой текущему пользователю предоставлена хотя бы одна привилегия доступа; и DBA_TAB_COLUMNS, включающем одну строку для каждого столбца каждой таблицы базы данных.

Большая часть информации в описываемой системной таблице или представлении относится к определению столбца. Здесь указывается его имя, тип данных, длина, могут ли в нем присутствовать значения NULL и т.д. Кроме того, иногда в таблице приводится информация о том, как распределены значения данных в каждом столбце. Эта статистическая информация помогает СУБД выполнять запросы оптимальным образом.

С помощью запросов, аналогичных приведенному ниже, можно получать информацию о столбцах в СУБД Oracle.

Имена и типы данных столбцов моей таблицы OFFICES.

```
SELECT COLUMN_NAME, DATA_TYPE
  FROM USER_TAB_COLUMNS
 WHERE TABLE_NAME = 'OFFICES';
```

Таблица 16.5. Избранные столбцы представления SYSCAT.COLUMNS (СУБД DB2)

Имя столбца	Тип данных	Информация
TABSCHEMA	VARCHAR(128)	Схема, которой принадлежит таблица, содержащая столбец
TABNAME	VARCHAR(128)	Имя таблицы, содержащей столбец
COLNAME	VARCHAR(128)	Имя столбца
COLNO	SMALLINT	Позиция столбца в таблице (первый столбец — 0)
TYPESCHEMA	VARCHAR(128)	Схема, которой принадлежит домен столбца (по умолчанию SYSIBM)
TYPENAME	VARCHAR(18)	Название типа данных или домена столбца
LENGTH	INTEGER	Максимальная длина для столбцов, содержащих значения переменной длины
SCALE	SMALLINT	Масштаб чисел типа DECIMAL
DEFAULT	VARCHAR(254)	Значение по умолчанию
NULLS	CHAR(1)	'Y' — разрешены значения NULL; 'N' — не разрешены
CODEPAGE	SMALLINT	Кодовая страница для текстовых данных, представленных в расширенной кодировке
LOGGED	CHAR(1)	'Y' — разрешено ведение журнала доступа к столбцам, содержащим большие двоичные объекты; 'N' — ведение журнала не разрешено
COMPACT	CHAR(1)	'Y' — столбец, содержащий большие двоичные объекты, является сжатым; 'N' — столбец не сжат
COLCARD	BIGINT	Число различных значений в столбце
HIGH2KEY	VARCHAR(254)	Второе наибольшее значение данных в столбце
LOW2KEY	VARCHAR(254)	Второе наименьшее значение данных в столбце
AVGCOLLEN	INTEGER	Средняя длина для столбцов, содержащих значения переменной длины
KEYSEQ	SMALLINT	Позиция столбца в первичном ключе (или 0)
PARTKEYSEQ	SMALLINT	Позиция столбца в ключе сегментирования (или 0)
NQUANTILES	SMALLINT	Число квантилей в статистике столбца
NMOSTFREQ	SMALLINT	Число часто встречающихся значений в статистике столбца
REMARKS	VARCHAR(254)	Пользовательские примечания к столбцу

Подобно информации в системном каталоге, информация о столбцах представлена по-разному в различных СУБД. В табл. 16.5 описано содержимое системной таблицы SYSCAT.COLUMNS, в которой помещаются сведения о столбцах базы данных в DB2. Вот несколько запросов, которые можно выполнить в этой СУБД.

Найти в базе данных все столбцы с типом данных DATE.

```
SELECT TABSCHEMA, TABNAME, COLNAME
  FROM SYSCAT.COLUMNS
 WHERE TYPESCHEMA = 'SYSIBM'
   AND TYPENAME = 'DATE';
```

Указать имя владельца, имя столбца, тип данных и длину для всех содержащих более десяти символов текстовых столбцов представлений.

```
SELECT DEFINER, COLS.TABNAME, COLNAME, TYPENAME, LENGTH
  FROM SYSCAT.COLUMNS COLS, SYSCAT.TABLES TBLS
 WHERE TBLS.TABSCHEMA = COLS.TABSCHEMA
   AND TBLS.TABNAME = COLS.TABNAME
   AND (TYPENAME = 'VARCHAR' OR TYPENAME = 'CHARACTER')
   AND LENGTH > 10
   AND TYPE = 'V';
```

Определения столбцов в системных каталогах СУБД различных типов значительно отличаются друг от друга. Для сравнения в табл. 16.6 приведено определение таблицы SYSCOLUMNS из Informix Universal Server. Некоторые различия между таблицами, содержащими описания столбцов, носят лишь стилистический характер.

- Имена столбцов в указанных таблицах различны, даже когда они содержат аналогичные данные.
- Для идентификации таблицы, содержащей данный столбец, в DB2 используется комбинация имен схемы и таблицы; в Informix для этой цели применяется внутренний идентификатор, который является внешним ключом для таблицы SYSTABLES.
- В системном каталоге DB2 типы данных задаются в текстовом виде (например, CHARACTER); в системном каталоге Informix используются целочисленные коды типов данных.

Таблица 16.6. Таблица SYSCOLUMNS (СУБД Informix)

Имя столбца	Тип данных	Информация
COLNAME	VARCHAR(128)	Имя столбца
TABID	INTEGER	Внутренний идентификатор таблицы, в которой содержится столбец
COLNO	SMALLINT	Позиция столбца в таблице
COLTYPE	SMALLINT	Код типа данных столбца, а также разрешены ли в нем значения NULL
COLLENGTH	SMALLINT	Длина столбца в байтах
COLMIN	INTEGER	Минимальная длина столбца в байтах
COLMAX	INTEGER	Максимальная длина столбца в байтах
EXTENDED_ID	INTEGER	Внутренний идентификатор расширенного типа данных
SECLABLID	INTEGER	Идентификатор метки безопасности столбца

Другие различия отражают разные возможности этих двух СУБД.

- DB2 позволяет делать примечания (длиной до 254 символов) к каждому столбцу; в Informix такой возможности нет.

- В Informix отслеживается минимальный и максимальный размер данных для столбцов, хранящих значения переменной длины; в DB2 такая информация непосредственно недоступна.

Информация о представлениях

Определения представлений, имеющихся в базе данных, хранятся в системном каталоге. Каталог СУБД DB2 содержит две системные таблицы, в которых отслеживается состояние представлений. Таблица SYSCAT.VIEWS (табл. 16.7) содержит SQL-определения всех представлений в текстовом виде. Старые версии DB2 поддерживали текст SQL длиной до 3600 символов, и в них определения, превышающие этот размер, хранились в нескольких последовательно пронумерованных строках (1, 2, 3 и т.д.). В новых версиях DB2 используется тип данных CLOB, способный принимать определения размером до 64 Кбайт, так что для каждого представления достаточно одной строки таблицы SYSCAT.VIEWS.

Таблица 16.7. Присваивание SYSCAT.VIEWS (СУБД DB2)

Имя столбца	Тип данных	Информация
VIEWSCHEMA	VARCHAR(128)	Схема, содержащая представление
VIEWNAME	VARCHAR(128)	Имя представления
DEFINER	VARCHAR(128)	Идентификатор создателя представления
SEQNO	SMALLINT	Последовательный номер данной строки SQL-текста (в DB2 UDB всегда 1)
VIEWCHECK	CHAR(1)	Тип проверки представления: 'N' — проверки нет 'L' — локальная проверка 'C' — каскадная проверка
READONLY	CHAR(1)	'Y', если представление доступно только для чтения; 'N' — в противном случае
VALID	CHAR(1)	'Y', если определение представления корректно; 'N' — в противном случае
QUALIFIER	VARCHAR(128)	Имя схемы по умолчанию в момент определения объекта
FUNC_PATH	VARCHAR(254)	Путь для разрешения вызовов функций в представлении
TEXT	CLOB(64K)	SQL-текст определения представления ("SELECT ..."); в старых версиях DB2 тип данных VARCHAR(3600)

С помощью этой таблицы можно посмотреть определения всех представлений базы данных. Как и в большинстве основных коммерческих СУБД, в DB2 информация о представлениях тесно связана с информацией о таблицах в каталоге DB2. Это означает, что нужные сведения часто можно получить несколькими способами. Вот, например, непосредственный запрос к системной таблице DB2 VIEWS, возвращающий имена всех представлений базы данных вместе с именами их создателей.

Список всех представлений, определенных в базе данных.

```
SELECT DISTINCT VIEWSCHEMA, VIEWNAME, DEFINER
  FROM SYSCAT.VIEWS;
```

Обратите внимание на применение предиката DISTINCT в старых версиях DB2 для удаления из списка тех представлений, определения которых слишком длинны для размещения в одной строке. Пожалуй, более простой способ получить ту же информацию — обратиться непосредственно к системной таблице TABLES, отбирая только те строки, которые соответствуют представлениям, согласно столбцу TYPE.

Список всех представлений, определенных в базе данных.

```
SELECT TABSCHEMA, TABNAME, DEFINER
  FROM SYSCAT.TABLES
 WHERE TYPE = 'V';
```

В большинстве ведущих СУБД представления описываются в структуре системного каталога схожим образом. В Informix Universal Server, например, имеется системная таблица SYSVIEWS, содержащая описания представлений. В каждой ее строке хранится 64-символьный фрагмент инструкции SELECT, формирующей представление. Если представление занимает несколько фрагментов, каждому из них присваивается порядковый номер, как и в DB2. В таблице SYSVIEWS содержится лишь один дополнительный столбец, в котором находятся идентификаторы таблиц, связывающие таблицу SYSVIEWS с соответствующими записями таблицы SYSTABLES. Таким образом дублируется меньше информации, но вам придется выполнять явное соединение обеих системных таблиц, чтобы получить информацию о представлении.

В Oracle SQL-текст определения представления также доступен через системное представление. Как и в случае таблиц и столбцов, имеется три системных представления — USER_VIEWS, содержащее информацию обо всех представлениях, принадлежащих текущему пользователю; ALL_VIEWS, содержащее информацию обо всех представлениях, к которым текущий пользователь имеет доступ; и DBA_VIEWS, содержащее информацию обо всех представлениях базы данных. В Oracle текст определения представления хранится в столбце типа LONG (специфичный для Oracle тип данных для хранения больших текстов) и может содержать много тысяч символов. Имеется также столбец TEXT_LENGTH, в котором записана длина этого определения. Вот как в СУБД Oracle можно получить информацию о представлениях.

Определения всех представлений, принадлежащих текущему пользователю.

```
SELECT VIEW_NAME, TEXT_LENGTH, TEXT
  FROM USER_VIEWS;
```

Заметим, что в большинстве интерактивных продуктов SQL (включая Oracle) при выводе на экран длинный текст обрезается. В базе данных же хранится полный текст.

Примечания

В СУБД DB2 вы можете создавать *примечания* (длиной до 254 символов) для каждой таблицы, каждого представления или столбца базы данных. Примечания позволяют сохранить в системном каталоге краткое описание таблицы или других данных. Примечания хранятся в системных таблицах SYSCAT.TABLES и SYSCAT.COLUMNS. В отличие от других элементов определения таблицы или столбца, примечания не задаются в инструкции CREATE TABLE. Вместо этого следует использовать инструкцию COMMENT (рис. 16.2). Ниже приведено несколько примеров использования инструкции COMMENT.

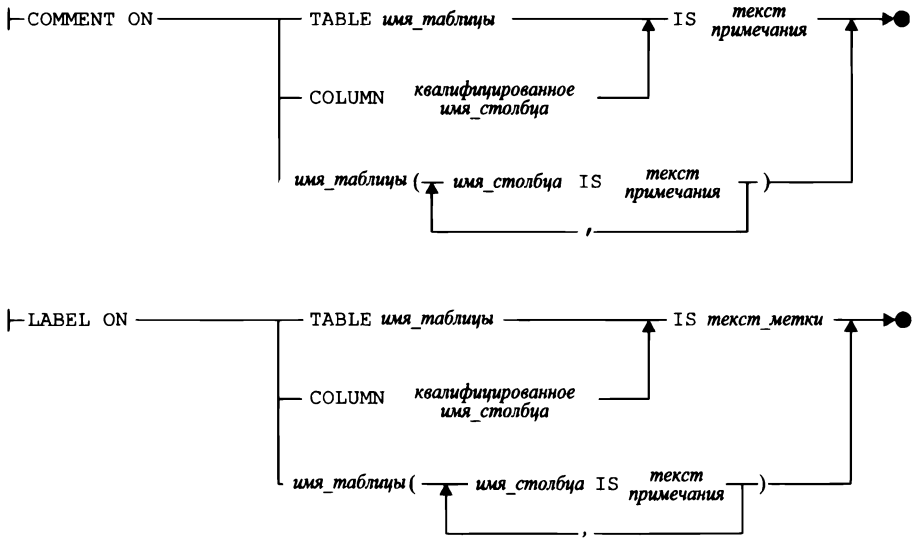


Рис. 16.2. Синтаксическая диаграмма инструкции COMMENT в DB2

Создать примечание для таблицы OFFICES.

```
COMMENT ON TABLE OFFICES
     IS 'Эта таблица содержит данные о наших офисах';
```

Создать примечания для столбцов TARGET и SALES таблицы OFFICES.

```
COMMENT ON OFFICES
(TARGET IS 'Это плановый годовой объем продаж офиса',
  SALES IS 'Это текущий объем продаж офиса');
```

Поскольку эта возможность берет свое начало из самых ранних продуктов SQL фирмы IBM, Oracle также поддерживает инструкцию COMMENT ON для назначения комментариев таблицам и столбцам. Однако эта информация хранится отдельно от прочей информации о таблицах и столбцах. Она доступна посредством системных представлений Oracle USER_TAB_COMMENTS и USER_COL_COMMENTS. В DB2 разрешается также создавать примечания для ограничений, хранимых процедур, схем, табличных пространств, триггеров и других объектов базы данных. Эта возможность не поддерживается ни в стандарте SQL, ни в большинстве других СУБД.

Информация об отношениях между таблицами

В середине 90-х годов, наряду с введением понятия ссылочной целостности, в ведущих корпоративных СУБД системные каталоги были расширены информацией о первичных ключах, внешних ключах и создаваемых ими отношениях "предок-потомок". В DB2, которая была одной из первых СУБД, поддерживающих ссылочную целостность, эта информация находится в системной таблице SYSCAT.REFERENCES (табл. 16.8). Каждое отношение "предок-потомок" между двумя таблицами базы данных представлено одной строкой. В этой строке содержатся имена родительской и дочерней таблиц, имя отношения, а также правила удаления и обновления для этого отношения. Чтобы получить информацию об отношениях в базе данных, вам следует выполнить запрос к этой таблице.

Список всех отношений "предок-потомок" между моими таблицами, включая имя отношения, имя таблицы-предка, имя таблицы-потомка и правила удаления и обновления для каждого отношения.

```
SELECT CONSTNAME, REFTABNAME, TABNAME,
       DELETERULE, UPDATERULE
  FROM SYSCAT.REFERENCES
 WHERE DEFINER = USER;
```

Список всех таблиц, связанных с таблицей SALESREPS (как предков, так и потомков).

```
SELECT REFTABNAME
  FROM SYSCAT.REFERENCES
 WHERE TABNAME = 'SALESREPS'
 UNION
SELECT TABNAME
  FROM SYSCAT.REFERENCES
 WHERE REFTABNAME = 'SALESREPS';
```

Таблица 16.8. Представление SYSCAT.REFERENCES (СУБД DB2)

Имя столбца	Тип данных	Информация
CONSTNAME	VARCHAR(128)	Имя отношения, описываемого данной строкой
TABSCHEMA	VARCHAR(128)	Схема, содержащая данное именованное отношение
TABNAME	VARCHAR(128)	Имя таблицы, к которой применяется ограничение
OWNER	VARCHAR(128)	Создатель таблицы, к которой применяется ограничение
REFKEYNAME	VARCHAR(128)	Имя родительского ключа
REFTABSCHEMA	VARCHAR(128)	Схема, содержащая родительскую таблицу
REFTABNAME	VARCHAR(128)	Имя родительской таблицы
COLCOUNT	SMALLINT	Число столбцов во внешнем ключе
DELETERULE	CHAR(1)	Правило удаления для ограничения внешнего ключа ('A' — отсутствие правила, 'C' — CASCADE, 'R' — RESTRICT)
UPDATERULE	CHAR(1)	Правило обновления для ограничения внешнего ключа ('A' — отсутствие правила, 'R' — RESTRICT)

Окончание табл. 16.8

Имя столбца	Тип данных	Информация
CREATE_TIME	TIMESTAMP	Время создания ограничения
FK_COLNAMES	VARCHAR(640)	Названия столбцов внешнего ключа
PK_COLNAMES	VARCHAR(640)	Названия столбцов первичного ключа
DEFINER	VARCHAR(128)	Идентификатор авторизации, под которым было создано данное ограничение

Имена столбцов внешнего ключа и соответствующих им столбцов первичного ключа перечислены в текстовом виде в столбцах FK_COLNAMES и PK_COLNAMES таблицы SYSCAT.REFERENCES. Более полезная информация содержится в другой системной таблице — SYSCAT.KEYCOLUSE (табл. 16.9). Для каждого столбца каждого внешнего ключа, первичного ключа и ограничения уникальности в этой системной таблице отводится одна строка. Специальная последовательность чисел определяет порядок следования столбцов в составном внешнем ключе. С помощью запроса к этой таблице, подобного приведенному ниже, можно узнать имена столбцов, связывающих некоторую таблицу с ее предком.

Список столбцов, связывающих таблицу ORDERS с таблицей PRODUCTS в отношении ISFOR.

```
SELECT COLNAME, COLSEQ
  FROM SYSCAT.KEYCOLUSE
 WHERE CONSTNAME = 'ISFOR'
 ORDER BY COLSEQ;
```

Таблица 16.9. Представление SYSCAT.KEYCOLUSE (СУБД DB2)

Имя столбца	Тип данных	Информация
CONSTNAME	VARCHAR(128)	Имя описываемого строкой ограничения (ограничения уникальности, первичного ключа или внешнего ключа)
TABSCHEMA	VARCHAR(128)	Схема, содержащая данное ограничение
TABNAME	VARCHAR(128)	Имя таблицы, к которой применяется ограничение
COLNAME	VARCHAR(128)	Имя столбца, участвующего в ограничении
COLSEQ	SMALLINT	Позиция, занимаемая столбцом в ограничении (первый столбец = 1)

Информация о первичных ключах и отношениях "предок-потомок", в которых они участвуют, содержится также в системных таблицах SYSCAT.TABLES и SYSCAT.COLUMNS, описанных ранее в табл. 16.2 и 16.5. Если таблица имеет первичный ключ, то ее строка в таблице SYSCAT.TABLES имеет в столбце KEYCOLUMNS значение, отличное от нуля и равное числу столбцов первичного ключа (1 — для простого ключа, 2 и более — для составного ключа). В таблице SYSCAT.COLUMNS строки столбцов, входящих в состав первичных ключей, имеют в столбце KEYSEQ значение, не равное нулю. Оно указывает позицию (1, 2 и т.д.), занимаемую столбцом в первичном ключе.

Чтобы найти первичный ключ какой-либо таблицы, вы можете сделать следующий запрос к таблице SYSCAT.COLUMNS.

Список столбцов, образующих первичный ключ таблицы PRODUCTS.

```
SELECT COLNAME, KEYSEQ, TYPENAME, REMARKS
  FROM SYSCAT.COLUMNS
 WHERE TABNAME = 'PRODUCTS'
   AND KEYSEQ > 0
 ORDER BY KEYSEQ;
```

В системных каталогах других основных СУБД поддержка первичных и внешних ключей осуществляется схожим образом. Например, в Oracle имеются системные представления ALL_CONSTRAINTS и USER_CONSTRAINTS, предоставляющие ту же информацию, что и системная таблица DB2 SYSCAT.REFERENCES. Информация о столбцах внешних и первичных ключей содержится в системных представлениях ALL_CONS_COLUMNS и USER_CONS_COLUMNS, которые аналогичны таблице SYSCAT.KEYCOLUSE DB2. В Microsoft SQL Server структура системного каталога подобна описанной выше, а информация о внешних ключах распределена между представлениями SYS.FOREIGN_KEYS и SYS.FOREIGN_KEY_COLUMNS.

В Informix Universal Server применяется подход, близкий используемому в DB2, но с теми же отличиями, которые уже были описаны при рассмотрении представления в системном каталоге информации о столбцах. Для каждого ограничения, определенного в базе данных, создается строка в системной таблице SYSCONSTRAINTS, где записано имя ограничения и его тип (условие на значение, первичный ключ, внешний ключ и т.д.). В этой таблице ограничению назначается также внутренний идентификатор, используемый при ссылке на данное ограничение в других таблицах системного каталога. Таблица, к которой применяется ограничение, тоже идентифицируется внутренним номером (он служит в качестве внешнего ключа к таблице SYSTABLES).

Более подробная информация о ссылочной целостности (внешних ключах) содержится в таблице SYSREFERENCES. В ней ограничение, первичный ключ и родительская таблица также определяются внутренними идентификаторами, создающими связи с таблицами SYSCONSTRAINTS и SYSTABLES. В таблице SYSREFERENCES описаны правила удаления и обновления для отношения, создаваемого внешним ключом, и другая подобная информация.

Информация о пользователях

В общем случае системный каталог содержит таблицу, в которой перечислены все пользователи, имеющие санкционированный доступ к базе данных. СУБД может использовать эту системную таблицу для проверки имени и пароля пользователя, когда он первый раз устанавливает соединение с базой данных. В таблице может также храниться и другая информация о пользователях.

В SQL Server информация о пользователях хранится в системном представлении SYS.DATABASE_PRINCIPALS (табл. 16.10). Каждая строка этой таблицы описывает одного пользователя или группу пользователей, входящих в схему защиты ба-

зы данных. В Informix применяется схожий подход; соответствующая системная таблица тоже называется SYSUSERS. В Oracle эта таблица называется DBA_USERS. Вот два эквивалентных запроса к SQL Server и Oracle, которые выводят список зарегистрированных пользователей.

Вывести все идентификаторы пользователей, зарегистрированных в SQL Server.

```
SELECT NAME
  FROM SYS.DATABASE_PRINCIPALS;
```

Вывести все идентификаторы пользователей, зарегистрированных в Oracle.

```
SELECT USERNAME
  FROM DBA_USERS;
```

Таблица 16.10. Столбцы представления каталога SYS.DATABASE_PRINCIPALS (СУБД SQL Server)

Имя столбца	Тип данных	Информация
name	sysname	Уникальное в пределах базы данных имя пользователя
principal_id	int	Уникальный в пределах базы данных идентификатор пользователя
type	char(1)	Тип пользователя: S — пользователь SQL U — пользователь Windows G — группа Windows A — роль приложения R — роль базы данных C — пользователь, отображенный на сертификат K — пользователь, отображенный на асимметричный ключ
type_desc	nvarchar(60)	Описание типа пользователя
default_schema_name	sysname	Имя, используемое в случае, когда имя SQL не определяет схему
create_date	datetime	Дата и время создания пользователя
modify_date	datetime	Дата и время последней модификации пользователя
owning_principal_id	int	Идентификатор владельца данного пользователя
sid	varbinary(85)	Идентификатор безопасности (Security identifier, SID), если пользователь определен как внешний по отношению к базе данных (типы S, U или G)
is_fixed_role	bit	Если значение равно 1, то строка представляет запись для одной из фиксированных ролей, таких как db_owner

В DB2 таблица SYSCAT.DBAUTH, содержащая имена пользователей, хранит также информацию о ролях и привилегиях пользователей в базе данных (т.е. являются ли они администраторами базы данных, могут ли создавать таблицы, могут ли

создавать программные модули для доступа к базе данных и т.д.). Вот запрос к СУБД DB2, эквивалентный показанным выше.

Вывести все идентификаторы пользователей, зарегистрированных в DB2.

```
SELECT DISTINCT GRANTEE
  FROM SYSCAT.DBAUTH
 WHERE GRANTEETYPE = 'U';
```

Информация о привилегиях

Помимо информации о структуре базы данных, системный каталог в общем случае хранит информацию, которая необходима СУБД для обеспечения безопасности базы данных. Как уже говорилось в главе 15, "SQL и безопасность", в разных СУБД применяются различные варианты базовой схемы выдачи привилегий SQL. Это находит свое отражение в структуре системных каталогов различных СУБД.

DB2 имеет одну из наиболее сложных схем пользовательских привилегий, которая охватывает даже отдельные столбцы таблиц. В табл. 16.11 приведены названия таблиц системного каталога DB2, которые хранят информацию о привилегиях, и кратко описана роль каждой из них.

Таблица 16.11. Представления системного каталога DB2, реализующие привилегии

Системная таблица	Роль
TABAUTH	Реализует привилегии на уровне таблиц, показывая, каким пользователям к каким таблицам и для каких операций (SELECT, INSERT, DELETE, UPDATE, ALTER и INDEX) разрешен доступ
COLAUTH	Реализует привилегии на уровне столбцов, показывая, какие пользователи имеют разрешение на обновление и создание внешних ссылок, а также на какие именно столбцы каких именно таблиц
DBAUTH	Определяет, какие пользователи имеют право устанавливать соединение с базой данных, создавать таблицы и выполнять различные функции администрирования базы данных
SCHEMAAUTH	Реализует привилегии на уровне схемы, показывая, какие пользователи имеют разрешения создавать, удалять или модифицировать объекты (таблицы, представления, домены и т.д.), входящие в схему
INDEXAUTH	Реализует привилегии на уровне индексов, показывая, какие пользователи имеют привилегии управления различными индексами
PACKAGEAUTH	Реализует привилегии программного доступа, показывая, какие пользователи могут создавать различные утилиты доступа к базам данных ("пакеты"), выполнять их и управлять ими

Схема авторизации пользователей, используемая в SQL Server, более фундаментальная и четкая, чем в DB2. Базы данных, таблицы, хранимые процедуры, триггеры и другие элементы рассматриваются в ней однотипно, как объекты, по отношению к которым назначаются привилегии. Четкая структура этой схемы отражена в представлении каталога SYS.DATABASE_PERMISSIONS (табл. 16.12), которое реализует всю схема привилегий в SQL Server. Каждая строка этого представления соответствует одной инструкции GRANT или REVOKE, выполненной в базе данных.

Таблица 16.12. Столбцы представления SYS.DATABASE_PERMISSIONS (СУБД SQL Server)

Имя столбца	Тип данных	Информация
class	tinyint	Класс, для которого имеется разрешение 0 — база данных 1 — объект или столбец 3 — схема 4 — пользователь базы данных 5 — пакет 6 — тип 10 — набор XML схем 15 — тип сообщения 16 — контракт службы 17 — служба 18 — подключение к удаленному сервису 19 — маршрут 23 — полнотекстовый каталог 24 — симметричный ключ 25 — сертификат 26 — асимметричный ключ
class_desc	nvarchar(60)	Описание класса, для которого имеется разрешение
major_id	int	Идентификатор сущности, для которой имеется разрешение
minor_id	int	Вторичный идентификатор сущности, для которой имеется разрешение
grantee_principal_id	int	Идентификатор пользователя базы данных, которому предоставлено разрешение
grantor_principal_id	int	Идентификатор пользователя базы данных, предоставившего разрешение
type	char(4)	Тип разрешения базы данных
permission_name	sysname	Имя разрешения
state	char(1)	Состояние разрешения
state_desc	nvarchar(60)	Описание состояния разрешения

Информационная схема SQL

В стандарте SQL форма системного каталога, которую должны поддерживать реляционные СУБД, явно не определена. В то время, когда принимался стандарт SQL2, уже существовал широкий разброс характеристик коммерческих СУБД различных типов и имели место огромные различия в их системных каталогах, так что невозможно было достичь согласия по вопросу стандартной спецификации системного каталога. Вместо этого авторы стандарта дали определение идеализированного системного каталога, который поставщики СУБД могли бы применять при разработке с нуля СУБД, соответствующих стандарту SQL. Поскольку СУБД MySQL разрабатывалась после того, как SQL Information Schema вошла в стандарт SQL, она соответствует стандарту в этой части. Microsoft добавила в SQL Server 2008 ряд представлений, совместимых с SQL Information Schema. Многие иные производители намерены поступить так же.

Таблицы этого идеализированного системного каталога (который в стандарте называется *схема определений*) приведены в табл. 16.13.

Таблица 16.13. Избранные столбцы стандартной схемы определений

Системная таблица	Содержимое
ASSERTIONS	Одна строка для каждого утверждения
AUTHORIZATIONS	Одна строка для каждого имени роли и одна строка для каждого идентификатора авторизации
CHARACTER_SETS	Одна строка для каждого дескриптора набора символов
CHECK_COLUMN_USAGE	Одна строка для каждого столбца, упомянутого в ограничении на значение, ограничении домена или утверждении
CHECK_CONSTRAINTS	Одна строка для каждого ограничения домена, ограничения на значения таблицы или каждого утверждения
CHECK_TABLE_USAGE	Одна строка для каждой таблицы, упомянутой в условиях отбора, ограничения на значение, ограничения домена или утверждения
COLLATIONS	Одна строка для каждого дескриптора порядка сортировки
COLUMN_PRIVILEGES	Одна строка для каждого дескриптора привилегии столбца
COLUMNS	Одна строка для каждого столбца в каждом определении таблицы или представления
DATA_TYPE_DESCRIPTOR	Одна строка для каждого домена или столбца, определенного с типом данных
DOMAIN_CONSTRAINTS	Одна строка для каждого ограничения домена
DOMAINS	Одна строка для каждого домена
KEY_COLUMN_USAGE	Одна или несколько строк для каждой строки в таблице TABLE_CONSTRAINTS, которая участвует в ограничениях на значение или ограничениях первичного или внешнего ключа
REFERENTIAL_CONSTRAINTS	Одна строка для каждой строки таблицы TABLE_CONSTRAINTS, которая участвует в ограничениях внешнего ключа
SCHEMATA	Одна строка для каждой схемы
TABLE_CONSTRAINTS	Одна строка для каждого ограничения таблицы, указанного в определении таблицы
TABLE_PRIVILEGES	Одна строка для каждой привилегии таблицы
TABLES	Одна строка для каждой таблицы или представления
TRIGGER_COLUMN_USAGE	Одна строка для каждого столбца, к которому обращается триггер
TRIGGER_TABLE_USAGE	Одна строка для каждой таблицы, к которой обращается триггер
TRIGGERS	Одна строка для каждого триггера
USAGE_PRIVILEGES	Одна строка для каждого дескриптора привилегий использования
USER_DEFINED_TYPES	Одна строка для каждого пользовательского типа данных
VIEW_COLUMN_USAGE	Одна строка для каждого столбца, к которому обращается представление
VIEW_TABLE_USAGE	Одна строка для каждой таблицы в определении каждого представления (если представление определено как запрос к нескольким таблицам, для каждой таблицы будет иметься отдельная строка)
VIEWS	Одна строка для каждой таблицы или представления

Стандарт SQL не требует, чтобы СУБД реально поддерживала описанные таблицы системного каталога, как и каталог вообще. Вместо этого в стандарте SQL определен ряд представлений, основанных на этих системных таблицах и описы-

вающих те объекты базы данных, которые доступны для текущего пользователя. (Эти представления каталога называются в стандарте *информационной схемой*.) Для того чтобы СУБД соответствовала стандарту SQL на уровне Intermediate или Full, она должна поддерживать эти представления. Такой подход дает пользователю стандартный способ получения информации о доступных ему объектах базы данных с помощью стандартных запросов к представлениям системного каталога. Следует отметить, что на начальном уровне (Entry Level) совместимости со стандартом SQL поддержка системных представлений не требуется.

На практике основные коммерческие реляционные СУБД постепенно переходят к поддержке информационной схемы SQL, обычно путем создания соответствующих представлений на основе таблиц своих собственных системных каталогов. Структура системных каталогов большинства СУБД достаточно близка к требуемой в стандарте; по крайней мере, 90 процентов требований, связанных с совместимостью со стандартом SQL, можно реализовать относительно просто. Реализация остальных 10 процентов существенно труднее, учитывая значительные различия между СУБД и то, что даже обобщенные системные представления затрагивают внутренние особенности реализации самой СУБД.

В результате полная поддержка представлений каталога SQL обычно реализуется в крупных новых версиях СУБД, вместе с соответствующими изменениями в ядре программного обеспечения СУБД. Представления каталога, требуемые стандартом SQL, приведены в табл. 16.14, вместе с кратким описанием информации, содержащейся в каждом представлении. Вот несколько примеров запросов, которые могут использоваться для получения информации о структуре базы данных из представлений системного каталога.

Имена всех таблиц и представлений, владельцем которых является текущий пользователь.

```
SELECT TABLE_NAME
  FROM TABLES;
```

Имена, позиции и типы данных всех столбцов во всех представлениях.

```
SELECT TABLE_NAME, COLUMN_NAME, ORDINAL_POSITION, DATA_TYPE
  FROM COLUMNS
 WHERE (COLUMNS.TABLE_NAME IN
        (SELECT TABLE_NAME FROM VIEWS));
```

Количество столбцов в таблице `OFFICES`.

```
SELECT COUNT(*)
  FROM COLUMNS
 WHERE (TABLE_NAME = 'OFFICES');
```

Заметим, что в случае MySQL имена представлений должны быть квалифицированы именем схемы `INFORMATION_SCHEMA` (например, `INFORMATION_SCHEMA.TABLES`), если только вы уже не находитесь в этой базе данных.

```
SELECT TABLE_NAME
  FROM INFORMATION_SCHEMA.TABLES;
```

Таблица 16.14. Представления системного каталога в соответствии со стандартом SQL

Представление	Содержимое
ADMINISTRATIVE_ROLE_AUTHORIZATIONS	Одна строка для каждой авторизации роли, включающей WITH ADMIN OPTION
APPLICABLE_ROLES	Одна строка для каждой роли, применимой к текущему пользователю
ASSERTIONS	Одна строка для каждого утверждения, которым владеет текущий пользователь; содержит имя утверждения и его характеристики
ATTRIBUTES	Одна строка для каждого пользовательского типа данных, определенного в каталоге
CHARACTER_SETS	Одна строка для каждого определения набора символов, доступного текущему пользователю
CHECK_CONSTRAINT_ROUTINE_USAGE	Одна строка для каждой выполнимой подпрограммы, владельцем которой является текущий пользователь и от которой зависит ограничение домена, таблицы или утверждение
CHECK_CONSTRAINTS	Одна строка для каждого ограничения на значение столбца таблицы, которой владеет текущий пользователь
COLLATIONS	Одна строка для каждого определения порядка сортировки, доступного текущему пользователю
COLLATION_CHARACTER_SET_APPLICABILITY	Одна строка для каждого набора символов, к которому применима сортировка
COLUMN_COLUMN_USAGE	Одна строка для каждого генерируемого столбца, который зависит от базового столбца
COLUMN_DOMAIN_USAGE	Одна строка для каждого столбца, определенного как зависимый от домена
COLUMN_PRIVILEGES	Одна строка для каждой привилегии для работы со столбцом, предоставленной текущему пользователю или предоставленной им другому пользователю; содержит имя таблицы и столбца, тип привилегии, указание на то, кто предоставил привилегию, кому она предоставлена и имеет ли текущий пользователь право передачи этой привилегии
COLUMN_UDT_USAGE	Одна строка для каждого столбца, который зависит от пользовательского типа данных
COLUMNS	Одна строка для каждого столбца, доступного текущему пользователю; содержит имя столбца, имя таблицы или представлений, включающих в себя данный столбец, тип его данных и другую информацию
CONSTRAINT_COLUMN_USAGE	Одна строка для каждого столбца, на который имеется ссылка в условии на значение, условии уникальности, утверждении или определении внешнего ключа, принадлежащих текущему пользователю
CONSTRAINT_TABLE_USAGE	Одна строка для каждой таблицы, на которую имеется ссылка в условии на значение, условии уникальности, утверждении или определении внешнего ключа, принадлежащих текущему пользователю
DATA_TYPE_PRIVILEGES	Одна строка для каждого объекта схемы, который включает дескриптор типа данных, доступного данному пользователю или роли
DIRECT_SUPERTABLES	Одна строка для каждой непосредственной надтаблицы, связанной с таблицей, определенной в данном каталоге, и владельцем которой является данный пользователь или роль
DIRECT_SUPERTYPES	Одна строка для каждого непосредственного определенного в данном каталоге надтипа, владельцем которого является данный пользователь или роль

Продолжение табл. 16.14

Представление	Содержимое
DOMAIN_CONSTRAINTS	Одна строка для каждого ограничения домена; содержит имя ограничения и его характеристики
DOMAINS	Одна строка для каждого домена, доступного текущему пользователю; содержит имя домена, базовый тип данных и его характеристики
ELEMENT_TYPES	Одна строка для каждого определенного в данном каталоге типа элемента, владельцем которого является данный пользователь или роль
ENABLED_ROLES	Одна строка для каждой роли, действующей в текущей сессии SQL
FIELDS	Одна строка для каждого определенного в данном каталоге типа поля, владельцем которого является данный пользователь или роль
INFORMATION_SCHEMA_CATALOG_NAME	Одна строка с именем базы данных для каждого пользователя ("каталога", по терминологии стандарта SQL), описываемого данной информационной схемой
KEY_COLUMN_USAGE	Одна строка для каждого столбца, на который наложено ограничение первичного или внешнего ключа либо ограничение уникальности и который входит в таблицу, принадлежащую текущему пользователю; строка содержит имя таблицы, имя столбца и позицию столбца в ключе
METHOD_SPECIFICATION_PARAMETERS	Одна строка для каждого SQL-параметра спецификации метода, определенной в представлении METHOD_SPECIFICATIONS
METHOD_SPECIFICATIONS	Одна строка для каждого SQL-метода каталога, доступного данному пользователю или роли
PARAMETERS	Одна строка для каждого SQL-параметра определенной в данном каталоге подпрограммы, который доступен данному пользователю или роли
REFERENCED_TYPES	Одна строка для каждого определенного в данном каталоге ссылочного типа, который доступен данному пользователю или роли
REFERENTIAL_CONSTRAINTS	Одна строка для каждого ссылочного ограничения (определения внешнего ключа) на таблицу, которой владеет текущий пользователь; содержит имена ограничений и имена родительских и дочерних таблиц
ROLE_COLUMN_GRANTS	Одна строка для каждой привилегии для работы с определенным в данном каталоге столбцом, который доступен (или права на который переданы) активным в настоящий момент ролям
ROLE_ROUTINE_GRANTS	Одна строка для каждой привилегии для работы с вызываемой SQL-подпрограммой, которая определена в данном каталоге и доступна (или права на нее переданы) активным в настоящий момент ролям
ROLE_TABLE_GRANTS	Одна строка для каждой привилегии для работы с определенной в данном каталоге таблицей, которая доступна (или права на которую переданы) активным в настоящий момент ролям
ROLE_TABLE_METHOD_GRANTS	Одна строка для каждой привилегии для работы с определенным над таблицами структурированных типов методом, который доступен (или права на который переданы) активным в настоящий момент ролям
ROLE_USAGE_GRANTS	Одна строка для каждой определенной в данном каталоге привилегии USAGE, которая доступна (или права на которую переданы) активным в настоящий момент ролям
ROLE_UDT_GRANTS	Одна строка для каждой привилегии для работы с определенным в данном каталоге пользовательским типом данных, который доступен (или права на который переданы) активным в настоящий момент ролям

Продолжение табл. 16.14

Представление	Содержимое
ROUTINE_COLUMN_USAGE	Одна строка для каждого столбца, владельцем которого является текущий пользователь или роль и от которого зависят подпрограммы SQL, определенные в данном каталоге
ROUTINE_PRIVILEGES	Одна строка для каждой привилегии для работы с вызываемыми SQL-подпрограммами, которые определены в данном каталоге и доступны (или права на них переданы) активным в настоящий момент ролям
ROUTINE_ROUTINE_USAGE	Одна строка для каждой вызываемой SQL-подпрограммы, владельцем которой является данный пользователь или роль, от которой зависит SQL-подпрограмма, определенная в данном каталоге
ROUTINE_SEQUENCE_USAGE	Одна строка для каждого генератора внешней последовательности, владельцем которого является данный пользователь или роль и от которого зависит SQL-подпрограмма, определенная в данном каталоге
ROUTINE_TABLE_USAGE	Одна строка для каждой таблицы, владельцем которой является данный пользователь или роль и от которой зависит SQL-подпрограмма, определенная в данном каталоге
ROUTINES	Одна строка для каждой вызываемой SQL-подпрограммы в каталоге, которая доступна данному пользователю или роли
SCHEMATA	Одна строка для каждой схемы в базе данных, принадлежащей текущему пользователю; содержит имя схемы, набор символов по умолчанию и т.д.
SEQUENCES	Одна строка для каждого генератора внешней последовательности, который определен в данном каталоге и доступен данному пользователю или роли
SQL_FEATURES	Одна строка для каждой возможности или подвозможности из стандарта SQL, указывающая, реализована ли она в данной реализации SQL
SQL_IMPLEMENTATION_INFO	Одна строка для каждого определенного в стандарте SQL элемента информации реализации SQL, указывающая наличие поддержки в данной реализации SQL
SQL_LANGUAGES	Одна строка для каждого языка (например, COBOL, C и т.д.), поддерживаемого СУБД данного типа; в строке указывается уровень соответствия стандарту SQL, тип поддерживаемого диалекта SQL и т.д.
SQL_PACKAGES	Одна строка для каждого пакета стандарта SQL, указывающая, поддерживается ли этот пакет в данной реализации SQL
SQL_PARTS	Одна строка для каждой части стандарта SQL, указывающая, поддерживается ли эта часть в данной реализации SQL
SQL_SIZING	Одна строка для каждого элемента размера стандарта SQL, указывающая, поддерживается ли этот размер в данной реализации SQL
SQL_SIZING_PROFILES	Одна строка для каждого элемента размера стандарта SQL, указывающая размер, требуемый одним или несколькими профилями стандарта
TABLE_CONSTRAINTS	Одна строка для каждого ограничения (первичный ключ, внешний ключ, условие уникальности или условие на значение), заданного для таблицы, которой владеет текущий пользователь; содержит имя ограничения и таблицы, тип ограничения и его характеристики
TABLE_METHOD_PRIVILEGES	Одна строка для каждой привилегии для работы с методом, определенным над таблицами структурных типов, которые определены в каталоге и доступны (или права на них переданы) данному пользователю или роли

Окончание табл. 16.14

Представление	Содержимое
TABLE_PRIVILEGES	Одна строка для каждой привилегии для работы с таблицей, предоставленной текущему пользователю или предоставленной им другому пользователю; содержит имя таблицы, тип привилегии, указание на то, кто предоставил привилегию, кому она предоставлена и имеет ли текущий пользователь право предоставления этой привилегии
TABLES	Одна строка для каждой таблицы или каждого представления, доступных текущему пользователю; содержит имя объекта и тип (т.е. идет ли речь о таблице или представлении)
TRANSFORMS	Одна строка для каждого преобразования определенных в этом каталоге пользовательских типов, которые доступны данному пользователю или роли
TRANSLATIONS	Одна строка для каждого определения правила конвертирования текста, доступного текущему пользователю
TRIGGERED_UPDATE_COLUMNS	Одна строка для каждого столбца в каталоге, который является явным столбцом триггера события UPDATE, определенного в этом каталоге, и который доступен данному пользователю или роли
TRIGGER_COLUMN_USAGE	Одна строка для каждого столбца, от которого зависит определенный в этом каталоге триггер, владельцем которого является данный пользователь
TRIGGER_ROUTINE_USAGE	Одна строка для выполняемой SQL-подпрограммы, владельцем которой является данный пользователь или роль, от которой зависит определенный в этом каталоге триггер
TRIGGER_SEQUENCE_USAGE	Одна строка для каждого генератора внешней последовательности, владельцем которого является данный пользователь или роль и от которого зависит некоторый определенный в этом каталоге триггер
TRIGGER_TABLE_USAGE	Одна строка для каждой таблицы, от которой зависит определенный в этом каталоге триггер, владельцем которого является данный пользователь или роль
TRIGGERS	Одна строка для каждого определенного над таблицей в этом каталоге триггера, доступного данному пользователю или роли
UDT_PRIVILEGES	Одна строка для каждой привилегии для работы с пользовательскими типами в данном каталоге, которые доступны (или права на которые переданы) данному пользователю или роли
USAGE_PRIVILEGES	Одна строка для каждой привилегии использования, предоставленной текущему пользователю или переданной им другому пользователю
USER_DEFINED_TYPES	Одна строка для каждого определенного в этом каталоге пользовательского типа, который доступен данному пользователю или роли
VIEW_COLUMN_USAGE	Одна строка для каждого столбца, на который имеется ссылка в представлениях, принадлежащих текущему пользователю; содержит имя столбца и таблицы, в которую входит столбец
VIEW_ROUTINE_USAGE	Одна строка для каждой подпрограммы, владельцем которой является данный пользователь или роль и от которой зависит представление, определенное в этом каталоге
VIEW_TABLE_USAGE	Одна строка для каждой таблицы, на которую имеется ссылка в определениях представлений, принадлежащих текущему пользователю; содержит имя таблицы
VIEWS	Одна строка для каждого представления, доступного текущему пользователю; содержит имя, информацию о проверках и возможности обновления

Стандарт определяет также четыре используемых в представлениях каталога домена, которые доступны пользователям. Эти домены приведены в табл. 16.15.

Таблица 16.15. Домены, описанные в стандарте SQL

Системный домен	Значения
CARDINAL_NUMBER	Домен всех неотрицательных чисел от нуля до максимального целого числа, представленного типом INTEGER в данной СУБД; значение этого домена является нулем или допустимым положительным числом
CHARACTER_DATA	Домен всех символьных строк переменной длины, имеющих длину от нуля до максимального значения, поддерживаемого данной СУБД; значение, взятое из этого домена, является допустимой символьной строкой
SQL_IDENTIFIER	Домен всех символьных строк переменной длины, которые являются допустимыми идентификаторами SQL согласно стандарту; любое значение, взятое из этого домена, является допустимым именем таблицы, столбца и т.д.
TIME_STAMP	Домен для всех временных отметок, каждая из которых включает дату и время суток

Прочая информация каталога

Системный каталог СУБД является отражением ее свойств и возможностей. Во многих популярных СУБД используются дополнительные, расширенные, возможности языка SQL, поэтому их системные каталоги всегда содержат несколько уникальных таблиц. Вот лишь несколько примеров.

- СУБД DB2 и Oracle поддерживают псевдонимы, или синонимы (альтернативные имена таблиц). В DB2 информация о псевдонимах хранится вместе с другой информацией о таблицах в системной таблице SYSCAT.TABLES. В Oracle информация о синонимах доступна через представление DBA_SYNONYMS.
- В SQL Server можно работать одновременно со многими базами данных. Эта СУБД имеет системную таблицу SYS.DATABASES, в которой хранятся имена баз данных, управляемых одним сервером.
- В настоящее время многие СУБД поддерживают хранимые процедуры, и в системном каталоге содержится одна или несколько таблиц с описанием имеющихся хранимых процедур. В Sybase информация о хранимых процедурах находится в системной таблице SYSPROCEDURES.
- СУБД Ingres может иметь таблицы, распределенные по нескольким дисковым томам. В системной таблице IIMULTI_LOCATIONS отслеживается расположение многотомных таблиц.

Резюме

Системный каталог представляет собой совокупность системных таблиц, описывающих структуру реляционной базы данных.

- Когда структура базы данных изменяется, СУБД обновляет данные, которые хранятся в системных таблицах.
- Пользователь может делать запросы к системным таблицам с целью получения информации о таблицах, столбцах и привилегиях.
- Средства формирования запросов при помощи дружественного интерфейса облегчают пользователю задачу поиска в базе данных необходимой информации.
- Системные таблицы в разных СУБД организованы по-разному, и их имена различны; даже у СУБД одного и того же производителя имеются различия в их системных каталогах, отражающие различную внутреннюю структуру и системные возможности продуктов.
- Стандарт SQL не требует от СУБД располагать предписанным в стандарте набором системных таблиц, но определяет набор стандартных представлений, которые должны поддерживаться в любой СУБД, претендующей на высокий уровень соответствия стандарту SQL.

V ЧАСТЬ

Программирование и SQL

Глава 17
Встроенный SQL

Глава 18
Динамический SQL

Глава 19
SQL API

Язык SQL используется для доступа к базам данных не только в интерактивном режиме, но и в прикладных программах. В следующих трех главах описываются особенности программного SQL. В главе 17, "Встроенный SQL", рассматривается встроенный SQL — разновидность программного SQL, наиболее широко применяемая в реляционных СУБД. В главе 18, "Динамический SQL", описывается динамический SQL — усовершенствованная форма встроенного SQL; с его помощью создаются утилиты общего назначения для работы с базами данных. И наконец, в главе 19, "SQL API", рассматривается альтернативная разновидность программного SQL — интерфейс вызовов функций, применяемый в нескольких популярных СУБД.

17
ГЛАВА

Встроенный SQL

Язык SQL можно использовать для доступа к базам данных в *двух режимах*. Это одновременно как язык для интерактивного общения с базой данных, выполнения запросов и обновлений, так и язык, при помощи которого с базами данных работают прикладные программы. Сам язык, по большей части, одинаков в обоих режимах. Такая двойственность SQL имеет следующие преимущества:

- программисты могут относительно легко научиться писать программы, которые в ходе своей работы обращаются к базам данных;
- все возможности, доступные в интерактивном языке запросов, автоматически доступны и в прикладных программах;
- инструкции SQL, предназначенные для использования в программах, вначале могут быть проверены в интерактивном режиме, а затем вставлены в исходный текст программы;
- программы могут работать с базами данных на уровне таблиц и результатов запросов.

В настоящей главе кратко описываются типы программного SQL, предлагаемые ведущими производителями реляционных СУБД, а затем рассматривается используемый в реляционных СУБД компании IBM программный SQL, который называется *встроенный SQL*.

Методы программного SQL

SQL действительно является языком, который можно использовать при написании программ, но было бы неверно называть его языком программирования. В нем отсутствуют даже самые элементарные возможности настоящих языков программирования. В SQL нельзя объявлять переменные, в нем отсутствуют инструкция безусловного перехода `GOTO`, инструкция `IF` для проверки условий, инструкции `FOR`, `DO` и `WHILE` для организации циклов, нет блочной структуры программ и т.д. SQL мож-

но назвать *подъязыком*, который служит исключительно для работы с базами данных. Чтобы создать программу, которая в процессе работы должна обращаться к базе данных, необходимо написать ее на обычном языке программирования, таком как COBOL, PL/1, FORTRAN, Pascal, C, C++ или Java, или с применением языка сценариев, такого как Perl, PHP или Ruby, а затем добавить в нее инструкции SQL.

Изначально стандарт ANSI/ISO SQL описывал только программное применение SQL. В него не была включена даже интерактивная инструкция SELECT, описанная в главах 6–9, а вошла лишь программная инструкция SELECT, которая будет рассмотрена далее в настоящей главе. Стандарт SQL2, опубликованный в 1992 году, включил в себя спецификацию интерактивного SQL (названного в стандарте *прямым вызовом SQL*) и расширенных форм программного SQL (*динамического SQL*, описываемого в главе 18, "Динамический SQL".)

Производители коммерческих реляционных СУБД предлагают два основных метода применения SQL в прикладных программах.

- **Встроенный SQL**. При таком подходе инструкции SQL встраиваются непосредственно в исходный текст программы, создаваемой на другом языке программирования. Для пересылки информации из базы данных в программу используются специальные инструкции встроенного SQL. Исходный текст программы, включающий в себя инструкции встроенного SQL, перед компиляцией подается на вход специального препроцессора SQL, который вместе с другими программными инструментами преобразует этот исходный текст в исполняемую программу.
- **Интерфейс прикладного программирования**. При этом подходе программа взаимодействует с СУБД с применением набора функций, называемого *интерфейсом прикладного программирования* (Application Program Interface — API). Вызывая функции API, программа передает в СУБД инструкции SQL и получает обратно результаты запросов. В этом случае специальный препроцессор не требуется.

Встроенный SQL использовался в первых реляционных СУБД компании IBM; в 1980-х годах этот подход был принят также в большинстве коммерческих реализаций SQL. Первоначально в стандарте ANSI/ISO для программного SQL был определен отдельный *модульный язык*, но коммерческие SQL-продукты продолжали следовать стандарту де-факто от IBM. Но в 1989 году стандарт был расширен; в него вошло определение встроенного SQL для таких языков программирования, как Ada, C, COBOL, FORTRAN, Pascal и PL/1. Эта спецификация поддерживается и в последующих версиях стандарта SQL.

Параллельно с развитием встроенного SQL производители некоторых СУБД для мини-компьютерных систем в 1980-х годах занялись разработкой API для баз данных. СУБД Sybase, появившаяся в то время на рынке, предлагала *только* такой интерфейс и не поддерживала встроенный SQL. Созданная на ее основе СУБД SQL Server компании Microsoft придерживалась такой же идеологии. Вскоре после появления SQL Server компания Microsoft представила протокол ODBC (Open Database Connectivity) — еще один API, в основном, основанный на API SQL

Server, но обеспечивающий независимость от конкретной СУБД и разрешающий доступ к нескольким СУБД посредством единого набора функций.

Не так давно появился новый протокол JDBC (Java Database Connectivity), предназначенный для обеспечения доступа к реляционным базам данных из программ, написанных на языке Java. При растущей популярности API в настоящее время в мире баз данных используются оба подхода — как вызываемый, так и встроенный SQL. В общем случае программисты на старых языках программирования типа COBOL и Assembler склонны применять встроенный SQL. Программисты на современных языках наподобие C++ и Java предпочитают API. Спецификации встроенного SQL в Java были добавлены в стандарт в 1999 году и пересмотрены в нескольких последующих версиях. Изначально стандарт назывался SQLJ (позже он был переименован в SQL/JRT), а некоторые производители реализовали решения, известные как JSQL. Большинство реализаций JSQL использует препроцессор для трансляции встроенного в исходный текст Java SQL в программы Java с использованием JDBC API.

В табл. 17.1 перечислены ведущие реляционные СУБД и указано, какие интерфейсы программирования они поддерживают.

Таблица 17.1. Поддержка программного SQL в ведущих СУБД

СУБД	API	Поддержка встроенного SQL
DB2	ODBC, JDBC, JSQL	APL, Assembler, BASIC, COBOL, FORTRAN, Java, PL/1
Informix	ODBC, JDBC	C, COBOL, Java
Microsoft SQL Server	Библиотека DB (`dblib`), ODBC	C
MySQL	Свой API для C, ODBC, JDBC, Perl, PHP, Ruby, другие языки сценариев	Отсутствует
Oracle	OCI (Oracle Call Interface), ODBC, JDBC, JSQL, PHP, Perl	C, COBOL, FORTRAN, Pascal, PL/1, Java
Sybase	Библиотека DB (`dblib`), ODBC, JDBC, JSQL	C, COBOL, Java

Базовая технология встроенного SQL, называемая *статическим* SQL, рассматривается в настоящей главе. Некоторые дополнительные возможности встроенного SQL, называемые *динамическим* SQL, описываются в главе 18, "Динамический SQL". Вызываемые API, включая Sybase/SQL Server API, ODBC и JDBC, рассматриваются в главе 19, "SQL API".

Обработка инструкций в СУБД

Чтобы лучше понимать методы программного SQL, давайте рассмотрим, как СУБД выполняет инструкции SQL. Для обработки инструкции SQL СУБД проходит пять этапов, показанных на рис. 17.1.

1. Выполняется *синтаксический анализ* инструкции SQL. СУБД разделяет инструкцию на отдельные слова, затем проверяет, правильно ли в инструкции указана команда, используются ли допустимые предложения и т.д. На этом этапе обнаруживаются синтаксические ошибки.

2. Осуществляется *проверка правильности* инструкции SQL. Посредством обращения к системному каталогу выясняется, существуют ли в базе данных таблицы, указанные в инструкции, существуют ли указанные столбцы и являются ли их имена однозначными, имеет ли пользователь привилегии, необходимые для выполнения инструкции. На этом этапе обнаруживаются семантические ошибки.
3. Выполняется *оптимизация* инструкции — СУБД исследует возможные способы ее выполнения. Выясняется, может ли быть использован индекс для ускорения поиска; следует ли вначале применить условие отбора к таблице *A*, а затем соединить ее с таблицей *B* или удобнее начать с соединения таблиц, а условия отбора применить потом; можно ли избежать последовательного поиска по всей таблице или хотя бы ограничить его некоторой частью таблицы; можно ли избежать сортировки, применяя индекс. После исследования возможных альтернатив СУБД выбирает одну из них.
4. СУБД генерирует *план выполнения* инструкции, который является бинарным представлением действий, необходимых для выполнения инструкции. План выполнения в СУБД является эквивалентом объектного кода программы.
5. И наконец, СУБД реализует разработанный план, тем самым выполняя инструкцию.

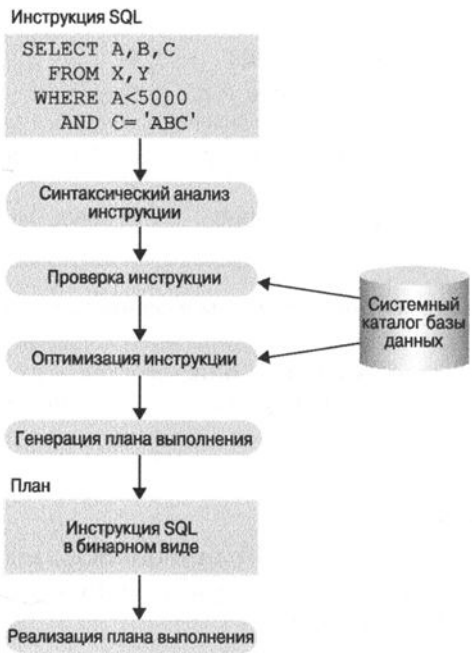


Рис. 17.1. Обработка СУБД инструкции SQL

Заметим, что использованные в данном описании термины могут отличаться в разных СУБД.

Перечисленные на рис. 17.1 этапы выполнения отличаются по числу обращений к базе данных и по времени работы процессора, требуемому для их выполнения. Синтаксический анализ инструкции SQL не требует обращения к базе данных и, как правило, проводится очень быстро. Оптимизация же, наоборот, требует интенсивной работы процессора и обращения к системному каталогу базы данных. Для сложных многотабличных запросов оптимизатор может исследовать более десятка альтернативных путей выполнения запроса. Однако стоимость выполнения запроса неправильным способом так высока, по сравнению со стоимостью его выполнения правильным (или, по крайней мере, лучшим из найденных) способом, что время, затраченное на оптимизацию, с избытком компенсируется более быстрым выполнением запроса.

При вводе инструкции SQL в интерактивном режиме СУБД последовательно проходит через все пять этапов, а пользователь в это время ожидает ответа. У СУБД нет выбора — она не знает, какую инструкцию вы введете, до тех пор пока вы этого не сделаете. Поэтому СУБД не может ничего делать заранее. Тем не менее некоторые СУБД, такие как Oracle, автоматически поддерживают кеш SQL, в котором хранятся последние выполненные инструкции. Если SQL-процессору будет передана та же инструкция, что и ранее, этап синтаксического анализа, а в некоторых случаях даже проверка правильности и оптимизация могут быть пропущены. Более того, если результаты предыдущего идентичного запроса все еще находятся в памяти, повторного выполнения запроса можно полностью избежать.

Однако в случае с программным SQL ситуация в корне меняется. Ряд описанных этапов может быть реализован уже во *время компиляции*. Тогда во *время выполнения*, при эксплуатации программы пользователем, необходимо будет осуществить только оставшиеся этапы. СУБД пытается в процессе компиляции завершить как можно больший объем работы, поскольку после создания окончательной версии программы она может выполняться пользователями в промышленных приложениях тысячи раз. В частности, СУБД, насколько это возможно, стремится произвести оптимизацию еще во время компиляции.

Основные концепции встроенного SQL

Главная идея встроенного SQL состоит в непосредственном объединении инструкций SQL с программой, написанной на принимающем языке программирования, таком как C, Java, Pascal, COBOL, FORTRAN, PL/1 или Assembler. Для этого используется следующий подход.

- Инструкции SQL смешиваются с инструкциями принимающего языка в исходном тексте программы. Исходная программа со встроенным SQL поступает на вход препроцессора SQL, который обрабатывает инструкции SQL.
- Встроенные инструкции SQL могут ссылаться на переменные принимающего языка программирования; это позволяет использовать в инструкциях SQL данные из программы.

- Встроенные инструкции SQL получают результаты SQL-запросов с помощью переменных принимающего языка; это позволяет программе пользоваться полученными данными и обрабатывать их.
- Для присвоения значений NULL столбцам базы данных и получения значений NULL из базы данных в программе применяются специальные переменные.
- Для построчной обработки результатов запроса во встроенный SQL добавляется несколько новых инструкций, которых нет в интерактивном SQL.

На рис. 17.2 приведена простая программа со встроенным SQL, написанная на языке С. В этой программе отражены многие (но не все) концепции встроенного SQL. Программа просит пользователя ввести номер офиса и выводит на экран информацию об этом офисе: город, регион, объем продаж и план продаж.

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        int officenum;        /* Номер офиса
                                 (задает пользователь) */
        char cityname[16];    /* Город                 */
        char regionname[11];  /* Регион                */
        float targetval;      /* План продаж           */
        float salesval;       /* Объем продаж          */
    exec sql end declare section;

    /* Настройка обработки ошибок */
    exec sql whenever sqlerror goto query_error;
    exec sql whenever not found goto bad_number;

    /* Запрос номера офиса у пользователя */
    printf("Введите номер офиса:");
    scanf("%d", &officenum);

    /* Выполнение SQL-запроса */
    exec sql select city, region, target, sales
              from offices
             where office=:officenum
              into :cityname, :regionname,
                   :targetval, :salesval;

    /* Вывод результатов */
    printf("Город  : %s\n", cityname);
    printf("Регион : %s\n", regionname);
    printf("План   : %f\n", targetval);
    printf("Продажи: %f\n", salesval);
    exit();
query_error:
    printf("Ошибка SQL: %ld\n", sqlca.sqlcode);
    exit();
bad_number:
    printf("Неверный номер офиса.\n");
    exit();
}
```

Рис. 17.2. Типичная программа со встроенным SQL

Не беспокойтесь, если программа покажется вам необычной или вы не сможете понять смысл некоторых инструкций, пока не прочитаете до конца настоящую главу. Один из недостатков встроенного SQL заключается в том, что исходный текст программы становится неоднородной смесью двух различных языков, что делает программу трудной для понимания без знания как SQL, так и принимающего языка. Другим недостатком встроенного SQL является то, что в нем используются конструкции, которых нет в интерактивном SQL; например, в данной программе присутствуют инструкция `WHENEVER` и предложение `INTO` инструкции `SELECT`.

Разработка программы со встроенным SQL

Программа со встроенным SQL содержит смесь инструкций SQL и принимающего языка программирования, поэтому ее нельзя просто скомпилировать с помощью компилятора принимающего языка. Вместо этого программа проходит через многоэтапный процесс, схематически изображенный на рис. 17.3. Фактически этот рисунок иллюстрирует процесс, применяемый в СУБД компании IBM (DB2, SQL/DS), однако во всех СУБД, поддерживающих встроенный SQL, применяется аналогичный алгоритм.

Рис. 17.3. Процесс разработки программы со встроенным SQL

1. Исходная программа со встроенным SQL подается на вход *препроцессора* SQL, представляющего собой специальный инструмент, который просматривает программу, находит в ней встроенные инструкции SQL и обрабатывает их. Для каждого языка программирования, поддерживаемого конкретной СУБД, требуется свой препроцессор. В коммерческие реляционные СУБД, как правило, входят препроцессоры для нескольких языков программирования, включая C, Pascal, COBOL, FORTRAN, Ada, PL/1, RPG и различные версии ассемблера.
2. Препроцессор создает на выходе два файла. Первый файл представляет собой исходный текст программы, из которого удалены встроенные инструкции SQL. Препроцессор заменяет их вызовами закрытых функций СУБД, обеспечивающих связь между программой и СУБД на этапе выполнения. Имена и соглашения о вызове этих функций известны только препроцессору и СУБД; они не являются открытым интерфейсом СУБД. Второй файл содержит копии всех инструкций SQL, встроенных в программу. Этот файл иногда называют *модулем запросов к базе данных* (Database Request Module — DBRM).
3. После препроцессора файл с исходным текстом программы компилируется стандартным компилятором принимающего языка программирования (например, компилятором C или COBOL). Компилятор преобразует исходный текст программы в объектный код. Обратите внимание: этот этап не имеет никакого отношения ни к СУБД, ни к SQL.
4. Объектные модули, созданные компилятором, поступают на вход *компоновщика*. Он связывает их с различными библиотечными функциями и на выходе создает исполняемую программу. В число библиотечных функций, связанных с исполняемой программой, входят и закрытые функции СУБД, упомянутые в п. 2.
5. Модуль запросов к базе данных, подготовленный препроцессором, передается специальной утилите `BIND`. Она исследует инструкции SQL, анализирует, проверяет, оптимизирует их и для каждой инструкции создает план выполнения. Результатом работы утилиты BIND является объединенный план выполнения для всех инструкций SQL, встроенных в программу. Он представляет собой выполняемую в СУБД версию встроенных инструкций SQL. Утилита BIND сохраняет этот план в базе данных и обычно присваивает ему имя прикладной программы, создавшей его.

Программы SQLJ следуют несколько более простому процессу, в основном, потому что перед выполнением компоновка программ Java не выполняется.

1. Программа на языке программирования Java со встроенным SQL передается препроцессору SQLJ (который именуется также транслятором). Транслятор, также написанный на Java, генерирует `.java`-файл, содержащий исходный текст программы на Java с инструкциями SQL, транслированными в стандартный код Java (обычно в виде вызовов API JDBC), и один или несколько профилей SQLJ, которые содержат информацию об операциях SQL.

2. Компилятор Java обрабатывает `.java`-файл и генерирует `.class`-файлы.
3. Шаги, связанные с компоновкой, в этом случае не требуются.
4. Компонент времени выполнения вызывается автоматически при запуске программы. Он использует профили SQLJ, которые помогают при выполнении команд SQL, включенных в исходную программу.

Следует отметить, что процесс компиляции программы, представленный на рис. 17.3, коррелирует с процессом выполнения инструкций SQL в СУБД, схематически изображенным на рис. 17.1. В частности, препроцессор обычно выполняет синтаксический анализ инструкции (первый этап), а утилита `BIND` проверяет правильность инструкций, производит их оптимизацию и создает план выполнения (второй, третий и четвертый этапы). Следовательно, при применении встроенного SQL первые четыре этапа из числа представленных на рис. 17.1 выполняются во время компиляции. И только пятый этап — фактическая реализация плана — осуществляется во время выполнения.

Таким образом, процесс разработки со встроенным SQL преобразует исходный текст в два исполняемых компонента:

- **исполняемую программу**, которая хранится в виде файла того же формата, что и любая другая исполняемая программа;
- **план выполнения**, который хранится внутри базы данных в формате, необходимом для данной СУБД.

Процесс компиляции программы со встроенным SQL может показаться запутанным; во всяком случае, он менее понятен, чем компиляция стандартной программы на С или COBOL. В большинстве случаев все этапы, представленные на рис. 17.3, выполняются с помощью одной управляющей процедуры, так что отдельные этапы оказываются скрыты от прикладного программиста. Но с точки зрения СУБД этот процесс имеет несколько существенных преимуществ.

- Смешение инструкций SQL и языка программирования в исходном тексте программы является эффективным методом слияния двух языков. Принимающий язык обеспечивает управление, блочную структуру, использование переменных и функций ввода-вывода, а SQL — только доступ к базе данных.
- Применение препроцессора означает, что трудоемкая работа по синтаксическому анализу и оптимизации может быть выполнена на **этапе разработки**. Полученная в результате исполняемая программа эффективно использует ресурсы центрального процессора.
- Модуль запросов к базе данных, создаваемый препроцессором, обеспечивает переносимость приложений. Прикладная программа может быть написана и протестирована в одной компьютерной системе, а затем ее исполняемая часть и модуль запросов могут быть перенесены в другую систему. После того как утилита `BIND` в новой системе создаст план выполнения и запишет его в базу данных, саму прикладную программу можно использовать без перекомпиляции.

- Интерфейс программы к закрытым функциям СУБД во время выполнения полностью скрыт от прикладного программиста. Программист работает со встроенным SQL на уровне исходного текста и может не беспокоиться о других, более сложных, интерфейсах.

Выполнение программы со встроенным SQL

В процессе компиляции программы со встроенным SQL (рис. 17.3) создаются два исполняемых компонента: собственно исполняемая программа и план ее выполнения, сохраняющийся в базе данных. При запуске программы со встроенным SQL два указанных компонента выполняют необходимую работу совместно.

1. Исполняемая программа загружается операционной системой как обычно, и начинается выполнение ее инструкций.
2. Один из первых вызовов, генерируемых препроцессором, — это вызов функции СУБД, которая находит и загружает план выполнения для данной программы.
3. Для каждой встроенной инструкции SQL программа вызывает одну или несколько закрытых функций СУБД, запрашивая выполнение соответствующей инструкции в имеющемся плане. СУБД находит инструкцию, выполняет соответствующую часть плана и возвращает управление программе.
4. Программа продолжает работать, как описано в пп. 1–3, выполняя совместно с СУБД задачу, определенную исходной программой со встроенным SQL.

Защита данных во время выполнения

При работе с базой данных в интерактивном режиме СУБД обеспечивает защиту данных, используя идентификатор, который пользователь вводит в начале сеанса работы. Вы можете запрашивать выполнение любой инструкции SQL, но выполнит СУБД эту инструкцию или нет, зависит от привилегий, предоставленных пользователю с данным идентификатором. При выполнении программы со встроенным SQL СУБД должна учитывать два идентификатора:

- идентификатор пользователя, который разработал программу, или, более точно, того, кто запускал утилиту `BIND`, чтобы создать план выполнения;
- идентификатор пользователя, который в настоящий момент выполняет программу и реализует соответствующий план.

Может показаться странным, что СУБД должна учитывать идентификатор того пользователя, который запускал утилиту `BIND` (или, говоря более обобщенно, того, кто разработал прикладную программу или инсталлировал ее в системе), но фактически в системе безопасности DB2 и ряда других коммерческих СУБД применяются оба идентификатора пользователя. Чтобы разобраться в принципе работы системы безопасности, предположим, что пользователь `JOE` выполняет программу `ORDMAINT` (программу обслуживания заказов), которая обновляет таблицы `ORDERS`, `SALES` и `OFFICES`. План выполнения для программы `ORDMAINT` был создан администратором по обработке заказов, имеющим идентификатор пользователя `OPADMIN`.

В DB2 план выполнения является защищаемым объектом базы данных. Чтобы реализовать план, пользователь JOE должен иметь привилегию EXECUTE для работы с ним. Если такой привилегии у него нет, программа выполняться не будет. Когда программа выполняется, ее встроенные инструкции INSERT, UPDATE и DELETE изменяют содержимое базы данных. Привилегии пользователя с идентификатором OPADMIN определяют, осуществит ли исполняемый файл, содержащий план выполнения, эти изменения. Обратите внимание на то, что названный файл может модифицировать таблицы, даже если у пользователя JOE нет требуемых привилегий. Однако могут быть сделаны *только те* изменения, которые явно заданы во встроенных инструкциях SQL данной программы. Таким образом, DB2 осуществляет четкое управление схемой защиты базы данных. Можно значительно ограничить привилегии пользователей на доступ к таблицам, не лишая их возможности пользоваться готовыми программами.

Не во всех СУБД осуществляется защита плана выполнения. Там, где ее нет, привилегии плана выполнения определяются привилегиями пользователя, выполняющего программу. Согласно этой схеме, у пользователя должны быть привилегии на выполнение всех действий, имеющихся в плане, иначе программа завершится неуспешно. Если пользователь должен иметь привилегии, отличные от интерактивного режима, то необходимо ограничить для него доступ к самой интерактивной программе, что является недостатком данной системы защиты.

Автоматическая перекомпоновка

Следует отметить, что план выполнения оптимизируется для структуры базы данных, существующей в тот момент, когда утилита BIND помещает план в базу данных. Если позднее структура изменится (например, будет удален индекс или столбец из таблицы), любой план выполнения, в котором имеются ссылки на измененные структурные элементы, может стать недействительным. Для обработки таких ситуаций в СУБД вместе с планом выполнения хранится копия его исходных инструкций SQL.

СУБД также отслеживает состояние всех объектов базы данных, от которых зависит план выполнения. Если какой-нибудь из этих объектов модифицируется инструкцией DDL, СУБД автоматически отмечает план как недействительный. Когда в следующий раз программа попытается использовать этот план, СУБД обнаружит такую ситуацию и *автоматически перекомпонует* инструкции, создав новый образ плана. Автоматическая перекомпоновка является совершенно незаметной для прикладной программы (за исключением того, что при перекомпоновке может увеличиться время выполнения).

Хотя СУБД автоматически перекомпоновывает план при изменении одного из структурных элементов, с которым связан данный план, она не способна обнаружить изменения в структуре базы данных, позволяющие улучшить план. Предположим, что план использует последовательное сканирование таблицы для поиска некоторых строк, поскольку соответствующий индекс во время компоновки отсутствовал. Вполне возможно, что позднее этот индекс будет создан с помощью инструкции CREATE INDEX. Но чтобы воспользоваться преимуществами новой структуры, необходимо явно запустить утилиту BIND и перекомпоновать план.

Простые инструкции встроенного SQL

Проще всего встраиваются в программу инструкции SQL, которые являются самодостаточными и не возвращают таблицу результатов запроса. Рассмотрим, например, следующую инструкцию интерактивного SQL.

Удалить данные обо всех служащих с объемом продаж ниже $150 000.

```
DELETE FROM SALESREPS
 WHERE SALES < 150000.00;
```

На рис. 17.4–17.6 приведены три программы, которые с помощью встроенного SQL выполняют ту же задачу, что и данная инструкция интерактивного SQL. Эти программы написаны на C, COBOL и FORTRAN соответственно. Хотя программы являются очень простыми, они иллюстрируют основные особенности встроенного SQL.

- Инструкции встроенного SQL могут находиться между инструкциями принимающего языка программирования. Обычно не имеет значения, прописными или строчными буквами набраны инструкции SQL. Это определяется стилем принимающего языка.
- Все встроенные инструкции SQL начинаются со *спецификатора*, который указывает, что данная инструкция является инструкцией SQL. В реляционных СУБД компании IBM для большинства принимающих языков применяется спецификатор EXEC SQL, который является спецификатором и в стандарте ANSI/ISO SQL. В некоторых СУБД в целях обратной совместимости по-прежнему могут использоваться и другие спецификаторы.

```
main()
{
    exec sql include sqlca;
    exec sql declare salesreps
            table (empl_num integer not null,
                       name varchar(15) not null,
                        age integer,
                  rep_office integer,
                       title varchar(10),
                   hire_date date not null,
                     manager integer,
                       quota decimal(9,2),
                       sales decimal(9,2) not null);

    /* Вывод сообщения пользователю */
    printf("Удаление персонала с низкими продажами.\n");

    /* Выполнение инструкции SQL    */
    exec sql delete from salesreps
              where sales < 150000.00;
    exec sql commit;

    /* Вывод другого сообщения      */
    printf("Удаление завершено.\n");
    exit();}
```

Рис. 17.4. Программа со встроенным SQL на языке программирования C

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
        EXEC SQL INCLUDE SQLCA.
        EXEC SQL DECLARE SALESREPS TABLE
                          EMPL_NUM INTEGER NOT NULL,
                              NAME VARCHAR(15) NOT NULL,
                               AGE INTEGER,
                       REP_OFFICE INTEGER,
                             TITLE VARCHAR(10),
                           RE_DATE DATE NOT NULL,
                           MANAGER INTEGER,
                             QUOTA DECIMAL,(9,2)
                             SALES DECIMAL(9,2) NOT NULL)
        END-EXEC.
PROCEDURE DIVISION.
*
* ВЫВОД СООБЩЕНИЯ ПОЛЬЗОВАТЕЛЮ
  DISPLAY "Удаление персонала с низкими продажами.".
*
* ВЫПОЛНЕНИЕ ИНСТРУКЦИИ SQL
  EXEC SQL DELETE FROM SALESREPS
           WHERE QUOTA < 150000
  END EXEC.
  EXEC SQL COMMIT
  END EXEC.
*
* ВЫВОД ДРУГОГО СООБЩЕНИЯ
DISPLAY "Удаление завершено.".
```

Рис. 17.5. Программа со встроенным SQL на языке программирования COBOL

```
PROGRAM SAMPLE
      100 FORMAT (' ',A35)
          EXEC SQL INCLUDE SQLCA
          EXEC SQL DECLARE SALESREPS TABLE
         C                 (EMPL_NUM INTEGER NOT NULL,
         C                      NAME VARCHAR(15) NOT NULL,
         C                       AGE INTEGER,
         C                REP_OFFICE INTEGER,
         C                     TITLE VARCHAR(10),
         C                 HIRE_DATE DATE NOT NULL,
         C                   MANAGER INTEGER,
         C                     QUOTA DECIMAL(9,2),
         C                     SALES DECIMAL(9,2) NOT NULL)
    *
    *     ВЫВОД СООБЩЕНИЯ ПОЛЬЗОВАТЕЛЮ
          WRITE (6,100) 'Удаление персонала с низкими продажами.'
    *
    *     ВЫПОЛНЕНИЕ ИНСТРУКЦИИ SQL
          EXEC SQL DELETE FROM REPS
         C          WHERE QUOTA<150000
          EXEC SQL COMMIT
    *
    *     ВЫВОД ДРУГОГО СООБЩЕНИЯ
          WRITE (6,100) 'Удаление завершено.'
          RETURN
          END
```

Рис. 17.6. Программа со встроенным SQL на языке программирования FORTRAN

- Если встроенная инструкция SQL занимает несколько строк, то факт продолжения инструкции на следующей строке обозначается в соответствии с правилами принимающего языка. В программах на C, PL/1 и COBOL никакого специального символа не требуется. В программах на FORTRAN вторая и последующие строки инструкции должны иметь в шестом столбце символ продолжения.
- Каждая встроенная инструкция SQL заканчивается *ограничителем*, обозначающим ее окончание. В разных принимающих языках применяются различные ограничители. В языке COBOL ограничителем является строка `END-EXEC.`, заканчивающаяся точкой, как и другие инструкции COBOL. В PL/1 и C ограничитель представляет собой точку с запятой, которая в этих языках является символом окончания инструкции. В языке FORTRAN встроенная инструкция SQL заканчивается, когда отсутствует символ продолжения.

Метод встраивания, продемонстрированный на этих трех рисунках, справедлив для любой инструкции SQL, которая (а) не зависит от значений переменных принимающего языка и (б) не извлекает информацию из базы данных. Например, в приведенной на рис. 17.7 программе на языке C, которая создает новую таблицу `REGIONS` и добавляет в нее две строки, для встраивания инструкций SQL применяется та же методика, что и в программе, представленной на рис. 17.4. Во всех дальнейших примерах используются программы на языке C, за исключением тех случаев, когда требуется проиллюстрировать особенности конкретного принимающего языка.

```
main()
{
    exec sql include sqlca;
    /* Создание новой таблицы REGIONS */
    exec sql create table regions
                    (name char(15),
                 hq_city char(15),
                 manager integer,
                  target decimal(9,2),
                   sales decimal(9.2),
              primary key name,
              foreign key manager
               references salesreps);
    printf("Таблица создана.\n");

    /* Вставка двух строк в таблицу   */
    exec sql insert into regions
        values ('Eastern', 'New York', 106, 0.00, 0.00);
    exec sql insert into regions
        values ('Western', 'Los Angeles', 108, 0.00, 0.00);
    printf("Таблица заполнена.\n");

    exit();
}
```

Рис. 17.7. Создание таблицы с помощью встроенного SQL

Объявления таблиц

Инструкция DECLARE TABLE, представленная на рис. 17.8, объявляет таблицу, на которую будут ссылаться одна или несколько встроенных инструкций SQL. Это необязательная инструкция, которая помогает препроцессору произвести синтаксический анализ и проверку правильности встроенных инструкций SQL. С помощью инструкции DECLARE TABLE программа явно указывает, какие столбцы имеются в таблице и какие они имеют размеры и типы данных. Препроцессор проверяет соответствие между ссылками на таблицы и столбцы в программе и объявлениями таблиц.

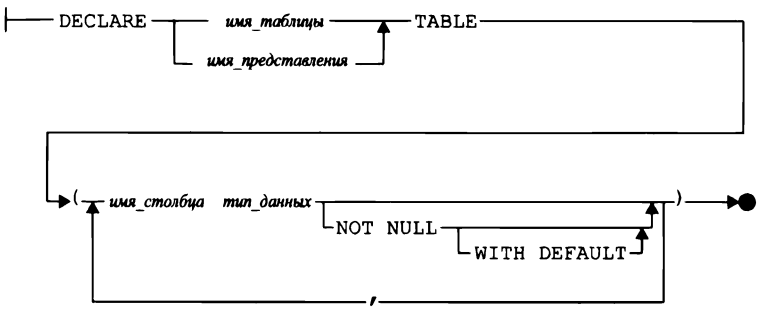


Рис. 17.8. Синтаксическая диаграмма инструкции DECLARE TABLE

Во всех трех программах, приведенных на рис. 17.4–17.6, применяется инструкция DECLARE TABLE. Важно отметить, что данная инструкция предназначена исключительно для препроцессора и для документирования программы. Она не является ни исполняемой, ни обязательной инструкцией, и вы не обязаны явно объявлять таблицу перед обращением к ней во встроенных инструкциях DML или DDL. Однако применение инструкции DECLARE TABLE делает вашу программу более понятной и облегчает ее сопровождение. Все реляционные СУБД компании IBM поддерживают инструкцию DECLARE TABLE; большинство же других СУБД ее не поддерживает, и в этом случае препроцессор выдаст сообщение об ошибке.

Обработка ошибок

Когда в интерактивном режиме вы вводите инструкцию SQL, вызывающую ошибку, интерактивная программа сразу же отображает на экране сообщение об ошибке, прерывает выполнение инструкции и просит вас ввести новую инструкцию. В случае встроенного SQL за обработку ошибок отвечает прикладная программа. Встроенные инструкции SQL могут вызывать сообщения об ошибках двух типов.

- **Ошибки компиляции**. Неправильно поставленные запятые, ошибки в правописании ключевых слов SQL и другие подобные ошибки во встроенных инструкциях SQL обнаруживаются препроцессором SQL, и программист получает соответствующее сообщение. Программист может устранить эти ошибки и заново скомпилировать прикладную программу.
- **Ошибки выполнения**. Попытки вставить некорректные данные или отсутствие доступа на обновление таблицы могут быть обнаружены только во время выполнения. Такие ошибки должны обнаруживаться и обрабатываться прикладной программой.

В программах со встроенным SQL СУБД сообщает программе об ошибках времени выполнения при помощи возврата кода ошибки. При обнаружении ошибки дальнейшее ее описание и прочая информация о выполнявшейся инструкции доступны посредством дополнительной диагностической информации. Впервые подобный механизм обработки ошибок появился в ранних СУБД компании IBM, а затем, с некоторыми изменениями, был принят большинством ведущих производителей СУБД. Центральный элемент этого механизма — значение SQLCODE, возвращающее код ошибки, — был закреплен в стандарте ANSI/ISO. В стандарте SQL2, опубликованном в 1992 году, был описан принципиально новый, параллельный, механизм обработки ошибок, построенный на использовании значений SQLSTATE. Оба механизма описаны в следующих двух разделах.

Обработка ошибок с использованием SQLCODE

В этой схеме, первоначально разработанной для ранних продуктов IBM, СУБД передает программе со встроенным SQL информацию о состоянии при помощи *области коммуникации SQL* (SQL Communications Area — SQLCA). SQLCA представляет собой структуру данных, которая содержит поля с информацией об ошибках и индикаторы состояния. Извлекая информацию из SQLCA, прикладная программа может узнать, успешно ли была выполнена встроенная инструкция SQL, и действовать соответственно полученной информации.

Обратите внимание на то, что на рис. 17.4–17.7 первой встроенной инструкцией является инструкция INCLUDE SQLCA. Она указывает препроцессору SQL, чтобы тот включил в данную программу область коммуникации SQL. В разных СУБД конкретное содержимое области связей слегка различается, но она всегда содержит информацию одного и того же типа. На рис. 17.9 показана структура SQLCA, используемая в СУБД компании IBM. Наиболее важная часть области связей SQL — значение SQLCODE — поддерживается всеми основными СУБД, использующими встроенный SQL; она была определена в стандарте ANSI/ISO SQL1.

При выполнении каждой встроенной инструкции SQL СУБД присваивает значение полю SQLCODE в области коммуникации SQL, показывающее, как завершилось выполнение инструкции.

- Нулевое значение SQLCODE указывает на успешное выполнение инструкции без каких-либо ошибок или предупреждений.
- Отрицательное значение SQLCODE указывает на серьезную ошибку, которая не позволила правильно выполнить инструкцию. Например, попытка модифицировать представление, доступное только для чтения, приведет к присвоению отрицательного значения SQLCODE. Каждой ошибке, которая может возникнуть во время выполнения, соответствует свое отрицательное значение SQLCODE.
- Положительное значение SQLCODE указывает на необходимость предупреждения пользователя о происшедшем в процессе выполнения инструкции. Например, усечение или округление значения элемента данных, полученного программой, вызывает появление предупреждения. Каждому предупреждению, которое может возникнуть во время выполнения, соответствует отдельное положительное значение SQLCODE. Одно из наиболее распространенных предупреждений, имеющее код +100 в большинстве

реализаций и в стандарте SQL, означает отсутствие данных; оно генерируется в ситуации, когда программа пытается получить очередную строку результатов запроса, хотя предыдущая строка была последней.

```
struct sqlca
{
  unsigned char sqlcaid[8];    /* Строка "SQLCA "                  */
  long          sqlcabc;       /* Длина SQLCA, в байтах            */
  long          sqlcode;       /* Код состояния SQL                */
  short         sqlerrml;      /* Длина массива sqlerrmc           */
  unsigned char sqlerrmc[70];  /* Имя объекта (имена объектов),
                                  вызвавшего ошибку                */
  unsigned char sqlerrp[8];    /* Диагностическая информация       */
  long          sqlerrd[6];    /* Разные счетчики и коды ошибки    */
  unsigned char sqlwarn[8];    /* Массив флагов предупреждения     */
  unsigned char sqlext[8];     /* Расширения массива sqlwarn       */
};

#define SQLCODE sqlca.sqlcode /* Код состояния SQL                 */

/* 'W' в любом из полей SQLWARN сигнализирует о наличии
   предупреждения; в противном случае в полях находятся пробелы  */

#define SQLWARN0 sqlca.sqlwarn[0] /* Главный флаг предупреждения*/
#define SQLWARN1 sqlca.sqlwarn[1] /* Строка обрезана            */
#define SQLWARN2 sqlca.sqlwarn[2] /* Значения NULL не учтены в
                                     статистической функции     */
#define SQLWARN3 sqlca.sqlwarn[3] /* Слишком много/слишком мало
                                     переменных                 */
#define SQLWARN4 sqlca.sqlwarn[4] /* UPDATE/DELETE без WHERE    */
#define SQLWARN5 sqlca.sqlwarn[5] /* SQL/DS-DB2 несовместимость */
#define SQLWARN6 sqlca.sqlwarn[6] /* Неверные данные в
                                     арифметическом выражении   */
#define SQLWARN7 sqlca.sqlwarn[7] /* Зарезервировано            */
```

Рис. 17.9. Область коммуникации SQL для баз данных IBM (язык программирования C)

Так как любая выполняемая инструкция встроенного SQL потенциально может вызвать ошибку, правильно написанная программа будет проверять значение SQLCODE после *каждой* такой инструкции. На рис. 17.10 показан отрывок программы, написанной на C, в котором проверяется значение SQLCODE. На рис. 17.11 показан аналогичный отрывок программы на языке COBOL.

```
        .
        .
        .
exec sql delete from salesreps
         where quota < 150000;
if (sqlca.sqlcode < 0)
    goto error_routine;
        .
        .
        .
error_routine:
    printf("Ошибка SQL: %ld\n", sqlca.sqlcode);
    exit();
        .
        .
        .
```

Рис. 17.10. Отрывок программы на языке C с проверкой SQLCODE

```
        .
        .
        .
01 PRINT_MESSAGE.
    02 FILLER PIC X(11) VALUE 'Ошибка SQL:'.
    02 PRINT-CODE PIC SZ(9).
        .
        .
        .
    EXEC SQL DELETE FROM SALESREPS
             WHERE QUOTA < 150000
    END EXEC.
    IF SQLCODE NOT = ZERO GOTO ERROR-ROUTINE.
        .
        .
        .
ERROR-ROUTINE.
    MOVE SQLCODE TO PRINT-CODE.
    DISPLAY PRINT_MESSAGE.
        .
        .
        .
```

Рис. 17.11. Отрывок программы на языке COBOL с проверкой SQLCODE

Обработка ошибок с использованием SQLSTATE

К тому времени, когда появился стандарт SQL2, практически во всех коммерческих реляционных СУБД для сообщения об ошибках в программах со встроенным SQL использовалась переменная SQLCODE. Однако номера ошибок не были стандартизированы, и в разных СУБД одним и тем же ошибкам соответствовали различные номера. Кроме того, в связи с тем, что стандарт SQL1 допускал значительные различия в реализациях SQL, в разных СУБД возможные ошибки заметно отличаются друг от друга. Наконец, определение SQLCA в различных СУБД имело разный вид, а к моменту появления стандарта SQL2 существовало огромное количество инсталлированных приложений для работы с базами данных, так что любое изменение области коммуникации SQL могло нарушить их работоспособность.

Понимая, что невозможно согласовать значения SQLCODE в разных СУБД и сделать их соответствующими стандарту, разработчики стандарта SQL2 пошли другим путем. Они включили в стандарт переменную SQLCODE, но обозначили ее как *нежелательную*, имея в виду, что она устарела и в будущем будет исключена из стандарта. Вместо нее они ввели новую переменную SQLSTATE. В стандарте подробно определены ошибки, о которых можно сообщать с помощью переменной SQLSTATE, и каждой ошибке присвоен код. Чтобы соответствовать стандарту SQL2, СУБД должна сообщать об ошибке как с помощью переменной SQLCODE, так и с помощью переменной SQLSTATE. Таким образом, программы, использующие переменную SQLCODE, могут функционировать по-прежнему, а новые программы можно писать, используя стандартизированные коды ошибок SQLSTATE.

Переменная SQLSTATE состоит из двух частей.

- Двухсимвольный *класс ошибки*, который дает общую классификацию ошибки (например, "ошибка соединения", "недействительные данные" или "предупреждение").

- Трехсимвольный *подкласс ошибки*, который определяет конкретный тип ошибки. Например, в классе "недействительные данные" могут быть такие подклассы ошибок, как "деление на нуль", "недействительное числовое значение" или "недействительные дата/время".

Первый символ класса ошибки, согласно стандарту SQL, может быть цифрой от нуля до четырех (включительно) или буквой от A до H (включительно). Например, ошибки данных относятся к классу 22, нарушение ограничений целостности данных — к классу 23, а отмена транзакции — к классу 40. Стандартные коды подклассов ошибок также подчиняются указанным соглашениям. В частности, в классе ошибок с номером 40 существуют подклассы 001 — ошибка сериализации (означает, что программа объявлена проигравшей в тупиковой ситуации), 002 — нарушение ограничений целостности данных, 003 — статус завершения инструкции SQL не определен (например, произошел разрыв сетевого соединения или сбой сервера до завершения инструкции). На рис. 17.12 представлен текст той же программы на языке C, что и на рис. 17.10, но в этот раз для проверки ошибок применяется переменная SQLSTATE, а не SQLCODE.

```
    .
    .
    .
exec sql delete from salesreps
         where quota < 150000;
if (strcmp(sqlca.sqlstate,"00000"))
    goto error_routine;
    .
    .
    .
error_routine:
    printf("Ошибка SQL: %s\n", sqlca.sqlstate);
    exit();
    .
    .
    .
```

Рис. 17.12. Отрывок программы на языке C с проверкой SQLSTATE

Коды классов ошибок, первый символ которых является цифрой от пяти до девяти (включительно) или буквой от I до Z (включительно), не стандартизированы и имеют свой смысл в каждой конкретной реализации. Иными словами, стандарт по-прежнему допускает различия между СУБД, однако большинство обычных ошибок, возникающих при выполнении инструкций SQL, охватывается стандартными кодами классов ошибок. По мере перехода коммерческих СУБД к использованию SQLSTATE один из самых неприятных факторов несовместимости СУБД будет постепенно устраняться.

Стандарт SQL позволяет получать дополнительную информацию диагностического характера с помощью инструкции GET DIAGNOSTICS (рис. 17.13). Благодаря этой инструкции программа, использующая встроенный SQL, может узнать, как завершилась последняя выполненная инструкция встроенного SQL, а также получить подробные сведения о возникшей ошибке. Поддержка инструкции GET DIAGNOSTICS требуется уровнями Intermediate и Full совместимости со стан-

дартом SQL2 и не требуется уровнем Entry. На рис. 17.14 представлен текст той же программы на языке C, что и на рис. 17.12, но на этот раз в ней используется инструкция GET DIAGNOSTICS.

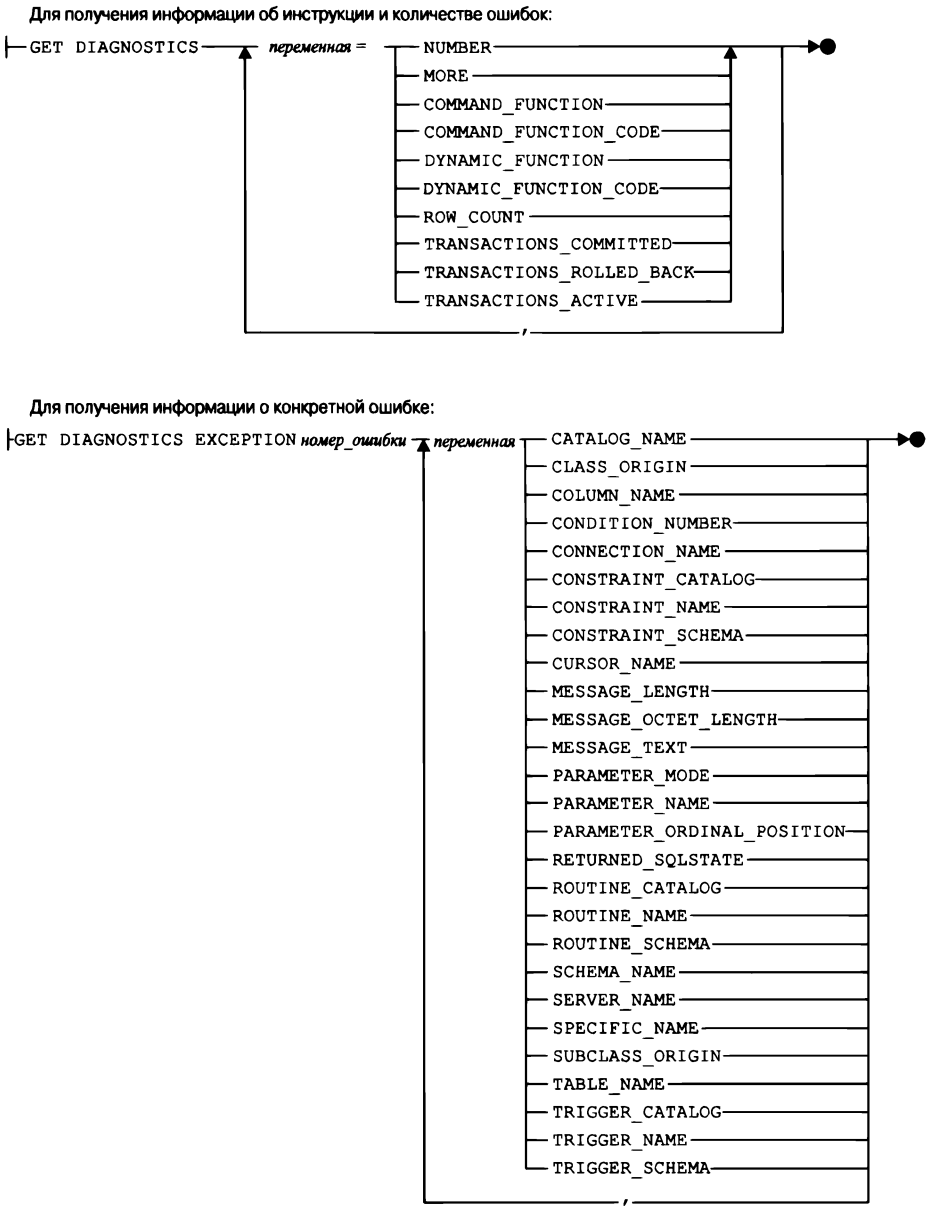


Рис. 17.13. Синтаксическая диаграмма инструкции GET DIAGNOSTICS

```
        .
        .
        .
        /* Выполнение инструкции DELETE и проверка на ошибки  */
        exec sql delete from salesreps
                 where quota < 150000;
        if (strcmp(sqlca.sqlstate,"00000"))
            goto error_routine;

        /* Удаление успешное; ищем количество удаленных строк */
        exec sql get diagnostics :numrows = ROW_COUNT;
        printf("Удалено %ld строк\n",numrows);
        .
        .
        .
    error_routine:
        /* Находим количество сообщений об ошибках            */
        exec sql get diagnostics :count = NUMBER;
        for (i=1; i<count; i++)
        {
            exec sql get diagnostics EXCEPTION :I
                                :err = RETURNED_SQLSTATE,
                                :msg = MESSAGE_TEXT;
            printf("Ошибка SQL # %d: код: %s сообщение: %s\n",
                   i, err, msg);
        }
        exit();
        .
        .
        .
```

Рис. 17.14. Отрывок программы на языке С с применением инструкции GET DIAGNOSTICS

Инструкция WHENEVER

Программиста быстро утомит написание программы, в которой после каждой встроенной инструкции SQL приходится явно проверять значение переменной SQLCODE. Для упрощения обработки ошибок во встроенном SQL имеется инструкция WHENEVER (рис. 17.15). Она является не исполняемой инструкцией, а директивой для препроцессора SQL. Инструкция WHENEVER дает препроцессору команду после каждой выполняемой инструкции SQL *автоматически* генерировать код, связанный с обработкой ошибок, и определяет, каким должен быть этот код.

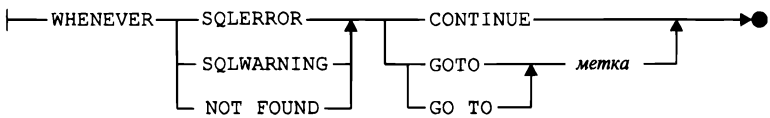


Рис. 17.15. Синтаксическая диаграмма инструкции WHENEVER

Имеется три формы инструкции WHENEVER, предназначенные для трех различных ситуаций.

- Инструкция WHENEVER SQLERROR служит для обработки ситуаций, связанных с возникновением серьезных ошибок (отрицательные значения SQLCODE).
- Инструкция WHENEVER SQLWARNING служит для обработки ситуаций, связанных с возникновением предупреждений (положительные значения SQLCODE).

- Инструкция WHENEVER NOT FOUND служит для обработки конкретного предупреждения, генерируемого СУБД в случае, когда программа пытается извлечь результаты запроса, а их больше не осталось. Данная форма инструкции предназначена для одиночных инструкций SELECT и FETCH и описывается ниже в настоящей главе.

Инструкция WHENEVER в любой из этих трех форм может потребовать от препроцессора генерации кода для одного из двух действий:

- директива GOTO указывает препроцессору сгенерировать переход к указанной *метке*, которая должна быть корректной меткой (или номером инструкции в соответствующих языках) в программе;
- директива CONTINUE служит указанием препроцессору не нарушать ход выполнения программы и осуществить переход к следующей инструкции принимающего языка.

Так как инструкция WHENEVER является директивой препроцессора, ее действие может быть перекрыто другой инструкцией WHENEVER, появляющейся в программе позже. На рис. 17.16 приведен отрывок программы, содержащий три инструкции WHENEVER и четыре исполняемые инструкции SQL. Первая инструкция WHENEVER вызывает переход к метке error1 при возникновении ошибки в любой из двух инструкций DELETE. Ошибка в инструкции UPDATE приводит к передаче управления следующей инструкции программы. Ошибка в инструкции INSERT вызывает переход к метке error2. Как видно из этого примера, главным назначением инструкции WHENEVER/CONTINUE является отмена действия предыдущей инструкции WHENEVER.

```
        .
        .
        .
        exec sql whenever sqlerror goto error1;

        exec sql delete from salesreps
                  where quota < 150000;
        exec sql delete from customers
                  where credit_limit < 20000;

        exec sql whenever sqlerror continue;

        exec sql update salesreps
                     set quota = quota * 1.05;

        exec sql whenever sqlerror goto error2;

        exec sql insert into salesreps (empl_num, name, quota)
                 values (116, 'Jan Hamilton', 100000.00);
        .
        .
        .
error1:
        printf("Ошибка SQL DELETE: %dl\n", sqlca.sqlcode);
        exit();
error2:
        printf("Ошибка SQL INSERT: %ld\n", sqlca.sqlcode);
        exit();
        .
        .
        .
```

Рис. 17.16. Использование инструкции WHENEVER

Инструкция WHENEVER значительно упрощает обработку ошибок встроенного SQL, и в прикладных программах обычно используют эту инструкцию, а не проверяют значение SQLCODE непосредственно. Помните, однако, что после того, как в программе появляется инструкция WHENEVER/GOTO, препроцессор будет осуществлять переход к заданной метке для *каждой* встроенной инструкции SQL, следующей за инструкцией WHENEVER/GOTO. Вы должны построить свою программу так, чтобы заданная метка была действительным адресом перехода для всех этих инструкций SQL, иначе необходимо вставить в программу другую инструкцию WHENEVER с иной меткой или вовремя отменить действие инструкции WHENEVER/GOTO.

Использование базовых переменных

В программах со встроенным SQL, показанных на предыдущих рисунках, отсутствовало какое-либо реальное взаимодействие между инструкциями встроенного SQL и программными инструкциями. Однако во многих приложениях требуется использовать во встроенных инструкциях SQL программные переменные. Предположим, например, что необходимо написать программу, которая увеличивает или уменьшает все личные планы продаж на некоторую сумму, выраженную в долларах. Программа должна запросить у пользователя эту сумму и с помощью встроенной инструкции UPDATE обновить столбец QUOTA таблицы SALESREPS.

Для выполнения этой задачи во встроенном SQL вводится концепция *базовых переменных*. Базовая переменная — это программная переменная, которая объявляется в принимающем языке программирования (например, COBOL или C) и на которую ссылается встроенная инструкция SQL. Для идентификации базовой переменной во встроенной инструкции SQL перед именем переменной ставится двоеточие. Двоеточие позволяет препроцессору легко отличать базовые переменные и объекты базы данных (например, таблицы или столбцы), имена которых могут совпадать.

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        float amount;    /* Вводится пользователем */
    exec sql end declare section;

    /* Приглашение ввести изменение плана  продаж  */
    printf("На сколько изменить объем продаж: ");
    scanf("%f", &amount);

    /* Обновление столбца QUOTA таблицы SALESREPS  */
    exec sql update salesreps
                set quota = quota + :amount;

    /* Проверка результата выполнения инструкции   */
    if (sqlca.sqlcode != 0)
        printf("Ошибка при обновлении.\n");
    else
        printf("Обновление успешное.\n");

    exit();
}
```

Рис. 17.17. Использование базовой переменной

На рис. 17.17 приведена написанная на С программа, которая модифицирует столбец QUOTA, используя базовую переменную. Программа запрашивает у пользователя сумму и запоминает введенное значение в переменной amount. Встроенная инструкция UPDATE обращается к этой базовой переменной. Перед выполнением инструкции UPDATE значение переменной amount вводится в программу и подставляется в инструкцию SQL. Например, если в ответ на запрос вы введете значение 500, то фактически СУБД выполнит такую инструкцию UPDATE.

```
exec sql update salesreps
            set quota = quota + 500;
```

Базовая переменная может находиться во встроенной инструкции SQL на месте любой константы. В частности, базовую переменную можно использовать в выражении присваивания.

```
exec sql update salesreps
            set quota = quota + :amount;
```

Базовая переменная может присутствовать в условии отбора.

```
exec sql delete from salesreps
          where quota < :amount;
```

Базовую переменную можно применять в предложении VALUES инструкции INSERT.

```
exec sql insert into salesreps (empl_num, name, quota)
         values (116, 'Bill Roberts', :amount);
```

Обратите внимание на то, что в каждом случае базовая переменная является частью информации, вводимой программой в базу данных; она входит в состав инструкции SQL, передаваемой в СУБД для выполнения. Ниже в настоящей главе вы увидите, что базовые переменные используются также для хранения информации, извлекаемой из базы данных; они принимают результаты запроса, возвращаемые из СУБД в программу.

Следует также отметить, что базовая переменная не может использоваться в качестве идентификатора SQL. Приведенный ниже пример использования переменной colname является недопустимым.

```
char *colname = "quota";

exec sql insert into salesreps (empl_num, name, :colname)
              values (116, 'Bill Roberts', 0.00);
```

Объявление базовых переменных

Прежде чем применять базовую переменную во встроенной инструкции SQL, ее необходимо объявить, используя обычные правила объявления переменных в базовом языке программирования. Например, на рис. 17.17 базовая переменная amount объявлена в соответствии с правилами языка С (float amount;). Когда препроцессор обрабатывает исходный текст программы, он запоминает имена всех встречающихся ему переменных вместе с типом данных и размером. Эта информация необходима препроцессору для того, чтобы, встретив в инструкции SQL базовую переменную, он мог сгенерировать правильный код.

Как показано на рис. 17.17, объявления базовых переменных должны находиться между двумя встроенными инструкциями SQL BEGIN DECLARE SECTION и END DECLARE SECTION. Эти две инструкции имеются только во встроенном SQL и являются не исполняемыми инструкциями, а директивами препроцессора. Они сигнализируют препроцессору, когда он должен уделять внимание объявлениям переменных, а когда может их игнорировать.

В простой программе можно собрать все объявления базовых переменных в одном разделе объявлений. Однако обычно базовые переменные приходится объявлять в различных точках программы, особенно в таких структурированных языках, как C, Pascal и PL/1. В этом случае каждое объявление базовых переменных должно охватываться парой инструкций BEGIN DECLARE SECTION / END DECLARE SECTION.

Инструкции BEGIN DECLARE SECTION и END DECLARE SECTION являются относительно новыми во встроенном SQL. Они определены в стандарте ANSI/ISO и требуются в DB2 в новейших реализациях встроенного SQL. Однако исторически сложилось так, что DB2 и многие другие СУБД не требуют наличия разделов объявления переменных, поэтому в некоторых препроцессорах SQL эти инструкции отсутствуют. В таком случае препроцессор сканирует и обрабатывает все объявления переменных в программе.

Когда используется базовая переменная, препроцессор может накладывать ограничения на объявление этой переменной в базовом языке программирования. Рассмотрим, например, следующий исходный текст на языке C.

```
#define BIGBUFSIZE 256
    .
    .
    .
exec sql begin declare section;
    char bigbuffer[BIGBUFSIZE+1];
exec sql end declare section;
```

Это правильное объявление переменной bigbuffer в языке C. Однако если вы попытаетесь использовать переменную bigbuffer как базовую переменную в инструкции встроенного SQL, например так:

```
exec sql update salesreps
            set quota = 300000
          where name = :bigbuffer;
```

то многие препроцессоры выдадут сообщение об ошибке (недопустимое объявление переменной bigbuffer). Проблема заключается в том, что эти препроцессоры не распознают символьные константы наподобие BIGBUFSIZE. Это всего лишь один пример тех особых ситуаций, что могут случиться при использовании встроенного SQL и препроцессора. К счастью, производители СУБД постоянно совершенствуют предлагаемые препроцессоры, и число подобных проблем уменьшается.

Базовые переменные и типы данных

Типы данных, используемые в реляционных СУБД и языках программирования (например, C или FORTRAN), очень часто не совпадают. Эти различия оказывают влияние на базовые переменные (тип базовой переменной при ее объявлении следует выбирать с учетом соответствия типов между принимающим языком и SQL),

так как последние играют двойную роль. С одной стороны, базовая переменная является программной переменной, объявляемой в соответствии с типами данных языка программирования и обрабатываемой программой на этом языке. С другой стороны, базовая переменная используется во встроенных инструкциях SQL и содержит информацию из базы данных.

```
     .
     .
     .
exec sql begin declare section;
    int   hostvar1 = 106;
    char *hostvar2 = "Joe Smith";
    float hostvar3 = 150000.00;
    char *hostvar4 = "01-JUN-1990";
exec sql end declare section;

exec sql update salesreps
            set manager = :hostvar1
          where empl_num = 102;

exec sql update salesreps
            set name = :hostvar2
          where empl_num = 102;

exec sql update salesreps
            set quota = :hostvar3
          where empl_num = 102;

exec sql update salesreps
            set hire_date = :hostvar4
          where empl_num = 102;
     .
     .
     .
```

Рис. 17.18. Базовые переменные и типы данных

Рассмотрим четыре встроенные инструкции UPDATE, представленные на рис. 17.18. В первой инструкции UPDATE столбец MANAGER имеет тип INTEGER, поэтому переменная hostvar1 должна быть объявлена в языке С как целочисленная. Во второй инструкции столбец NAME имеет тип VARCHAR, поэтому переменная hostvar2 должна содержать строковые данные. Программа на языке С должна объявлять переменную hostvar2 как массив символов, и для большинства СУБД необходимо, чтобы данные в массиве заканчивались нулевым символом (код 0). В третьей инструкции UPDATE столбец QUOTA имеет тип DECIMAL. Но соответствующего типа данных в языке С нет, так как в нем отсутствуют форматированные десятичные числа. Для большинства СУБД переменную hostvar3 в языке С необходимо объявлять как переменную с плавающей точкой, а СУБД должна автоматически преобразовывать значение с плавающей точкой в формат DECIMAL этой СУБД. Наконец, в четвертой инструкции UPDATE столбец HIRE_DATE имеет тип данных DATE. Для большинства СУБД необходимо объявлять переменную hostvar4 в языке С как массив символов и заполнять этот массив датами в приемлемом для СУБД формате.

Как видно из рис. 17.18, типы данных для переменных принимающего языка следует выбирать осторожно, с учетом их предполагаемого использования во встроенных инструкциях SQL. В табл. 17.2 приведены типы данных, определенные в стан-

дарте ANSI/ISO SQL, и соответствующие им типы данных в четырех из наиболее популярных языков программирования, упомянутых в стандарте. Стандарт определяет соответствие типов данных и правила встроенного SQL для языков Ada, C (охватывает C++), COBOL, FORTRAN, MUMPS, Pascal и PL/1.

Однако обратите внимание на то, что во многих случаях между типами данных нет однозначного соответствия. Кроме того, каждая СУБД имеет собственные предпочтения и неприятия типов данных и свои правила преобразования типов данных при использовании базовых переменных. Перед тем как полагаться на некоторые правила преобразования данных, проконсультируйтесь с документацией на вашу конкретную СУБД и внимательно прочтите описание используемого вами языка программирования.

Таблица 17.2. Типы данных в SQL и принимающих языках

SQL	C и C++	COBOL	FORTRAN	PL/1
SMALLINT	short	PIC S9 (4) COMP	INTEGER*2	FIXED BIN(15)
INTEGER	long	PIC S9 (9) COMP	INTEGER*4	FIXED BIN(31)
REAL	float	COMP-1	REAL*4	BIN FLOAT(21)
DOUBLE PRECISION	double	COMP-2	REAL*8	BIN FLOAT(53)
NUMERIC(p, s) DECIMAL(p, s)	double[1]	PIC S9 (p-s) V9(s) COMP-3	REAL*8[1]	FIXED DEC(p, s)
CHAR(n)	char x[n+1][2]	PIC X (n)	CHARACTER*n	CHAR(n)
VARCHAR(n)	char x[n+1][2]	Требуется преобразование[4]	Требуется преобразование[4]	CHAR(n) VAR
BIT(n)	char x[l][3]	PIC X (l)	CHARACTER*L3	BIT(n)
BIT VARYING(n)	char x[l][3]	Требуется преобразование[4]	Требуется преобразование[4]	BIN(n) VAR
DATE	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]
TIME	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]
TIMESTAMP	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]
INTERVAL	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]	Требуется преобразование[5]

[1] *Принимающий язык не поддерживает упакованные десятичные данные; приведение к типу данных с плавающей запятой и обратно может вызывать ошибки усечения и округления.*

[2] *В стандарте определено, что в языке C строки оканчиваются нулевым символом; старые СУБД отдельно возвращали длину строки.*

[3] *В данном случае длина строки символов (l) — это число битов (n), разделенное на количество битов в представлении одного символа (обычно 8), с округлением вверх.*

[4] *Принимающий язык не поддерживает строки символов переменной длины; большинство СУБД преобразует значения данного типа в строки постоянной длины.*

[5] *Принимающий язык не поддерживает типы данных, учитывающие региональные стандарты даты/времени; необходимо использовать текстовое представление соответствующих значений даты/времени.*

Базовые переменные и значения NULL

В большинстве языков программирования, в отличие от SQL, нет неизвестных или отсутствующих значений. Например, в языках COBOL, C или FORTRAN переменная всегда имеет некоторое значение, а такое понятие, как неизвестное или отсутствующее значение (значение NULL), не используется. Поэтому, когда с помощью программного SQL необходимо записать в СУБД значение NULL или извлечь его оттуда, приходится решать описанную проблему. Для ее решения во встроенном SQL вводится понятие *переменной-индикатора*. Во встроенной инструкции одно значение SQL определяется совместно парой переменных — базовой переменной и переменной-индикатором.

- Значение индикатора, равное нулю, означает, что базовая переменная содержит действительное значение, которое можно использовать.
- Отрицательное значение индикатора означает, что базовая переменная содержит значение NULL; значение базовой переменной в этом случае должно игнорироваться.
- Положительное значение индикатора означает, что базовая переменная содержит действительное значение, которое, возможно, было округлено или усечено. Такая ситуация может возникнуть только при извлечении информации из базы данных и рассматривается ниже в настоящей главе.

Когда во встроенной инструкции SQL вы используете базовую переменную, сразу за ней можно указать имя соответствующей переменной-индикатора. Перед именами обеих переменных ставится двоеточие. Ниже показана встроенная инструкция UPDATE, в которой используется базовая переменная amount и сопутствующая ей переменная-индикатор amount_ind.

```
exec sql update salesreps
            set quota = :amount :amount_ind, sales = :amount2
          where quota < 20000.00;
```

Если переменная amount_ind при выполнении инструкции UPDATE имеет неотрицательное значение, то СУБД трактует эту инструкцию, как если бы она имела такой вид.

```
exec sql update salesreps
            set quota = :amount, sales = :amount2
          where quota < 20000.00;
```

Если при выполнении инструкции UPDATE переменная-индикатор amount_ind имеет отрицательное значение, то СУБД трактует эту инструкцию, как если бы она имела такой вид.

```
exec sql update salesreps
            set quota = NULL, sales = :amount2
          where quota < 20000.00;
```

Пара "базовая переменная/переменная-индикатор" может присутствовать в предложении SET встроенной инструкции UPDATE (как в данном случае) или в предложении VALUES встроенной инструкции INSERT. Переменную-индикатор нельзя исполь-

зовать в условии отбора, так что следующая встроенная инструкция SQL является недопустимой.

```
exec sql delete from salesreps
         where quota = :amount :amount_ind;
```

Данный запрет существует по той же причине, по которой в условии отбора не разрешается использовать ключевое слово NULL, — нет смысла сравнивать значения QUOTA и NULL, так как ответ всегда будет NULL (неизвестно). Вместо переменной-индикатора необходимо явно задать проверку IS NULL. Ниже приведены две встроенные инструкции SQL, выполняющие задачу предыдущей недопустимой инструкции.

```
if (amount_ind < 0){
   exec sql delete from salesreps
            where quota is null;
}
else {
   exec sql delete from salesreps
            where quota = :amount;
}
```

Переменные-индикаторы особенно полезны в том случае, когда в программу из базы данных извлекаются элементы, которые могут иметь значения NULL. Подобное применение переменных-индикаторов описывается ниже в настоящей главе.

Выборка данных с помощью встроенного SQL

По методике, описанной в предыдущем разделе, в прикладную программу можно вставить любую инструкцию интерактивного SQL, *за исключением* инструкции SELECT. Чтобы в программе со встроенным SQL можно было извлекать данные, необходимо специальным образом расширить инструкцию SELECT. Причиной этого является принципиальное отличие между языком SQL и языками программирования (например, C и COBOL): запрос SQL создает таблицу результатов запроса, а большинство языков программирования могут обрабатывать только отдельные элементы данных или отдельные записи (строки данных).

Встроенный SQL должен создать мост между табличной логикой инструкции SELECT и последовательной обработкой строк в C, COBOL и других принимающих языках программирования. Поэтому во встроенном SQL запросы делятся на две группы.

- **Запросы, возвращающие одну запись**, когда ожидаемые результаты запроса представляют собой одну строку данных. Запрос о лимите кредита одного клиента или об объеме продаж и плане продаж одного служащего является примером такого запроса.
- **Запросы, возвращающие набор записей**, где ожидаемые результаты запроса могут содержать одну, ни одной или несколько строк данных. Запрос о заказах на сумму свыше $20000 или об именах служащих, работающих с опережением плана, является примером такого запроса.

В интерактивном SQL запросы этих двух типов не различаются: они выполняются с помощью одинаковых инструкций SELECT. Однако во встроенном SQL эти запросы выполняются совершенно по-разному. Запросы первого типа проще и обсуждаются в следующем разделе. Наборы записей рассматриваются ниже в настоящей главе.

Запросы, возвращающие одну запись

Многие запросы SQL возвращают одну строку результата. Такие запросы особенно распространены в программах обработки транзакций, где пользователь вводит идентификатор клиента или номер заказа и программа извлекает соответствующие данные о клиенте или заказе. Во встроенном SQL запросы этого типа выполняются с помощью одиночной инструкции SELECT, синтаксическая диаграмма которой изображена на рис. 17.19. Одиночная инструкция SELECT очень похожа на интерактивную: она содержит предложение SELECT, предложение FROM и необязательное предложение WHERE. Поскольку одиночная инструкция SELECT возвращает одну строку данных, нет необходимости в предложениях GROUP BY, HAVING или ORDER BY. В предложении INTO указывается базовая переменная, которая принимает данные, извлеченные инструкцией.

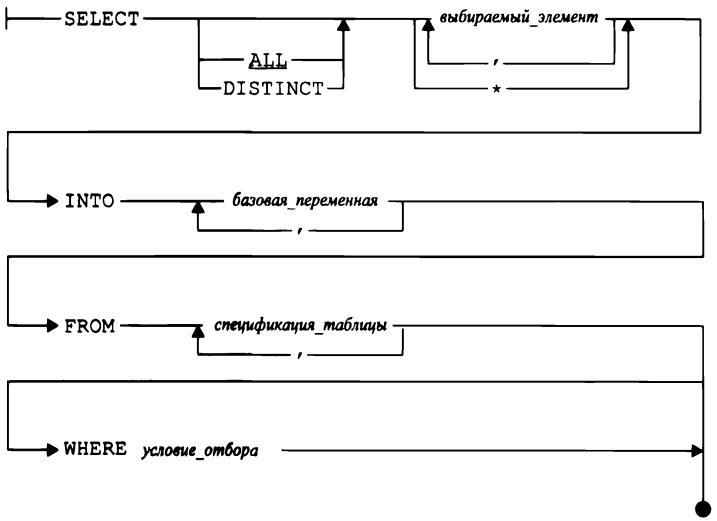


Рис. 17.19. Синтаксическая диаграмма одиночной инструкции SELECT

На рис. 17.20 представлена простая программа с одиночной инструкцией SELECT. Программа запрашивает у пользователя идентификатор и затем извлекает имя, план продаж и объем продаж служащего с таким идентификатором. Эти три извлекаемых элемента данных СУБД помещает в базовые переменные repname, repquota и repsales соответственно.

```
main()
{
    exec sql begin declare section;
        int   repnum;       /* Номер служащего (вводится) */
        char  repname[16]; /* Полученное имя служащего   */
        float repquota;     /* Полученный план продаж     */
        float repsales;     /* Полученные продажи         */
    exec sql end declare section;

    /* Запрос номера служащего у пользователя программы  */
    printf("Введите номер служащего:");
    scanf("%d", &repnum);

    /* Выполнение запроса SQL                            */
    exec sql select name, quota, sales
               into :repname, :repquota, :repsales
               from salesreps
              where empl_num = :repnum;

    /* Вывод полученных данных                           */
    if (sqlca.sqlcode = = 0)
    {
        printf("Имя:     %s\n", repname);
        printf("План:    %f\n", repquota);
        printf("Продажи: %f\n", repsales);
    }
    else
        if (sqlca.sqlcode = = 100)
            printf("Служащего с таким номером нет.\n");
        else
            printf("Ошибка SQL: %ld\n", sqlca.sqlcode);

    exit();
}
```

Рис. 17.20. Применение одиночной инструкции SELECT

Вспомним, что базовые переменные, которые использовались в инструкциях INSERT, DELETE и UPDATE, были входными переменными. В противоположность этому, базовые переменные, заданные в предложении INTO одиночной инструкции SELECT, являются выходными базовыми переменными. Каждая базовая переменная, указанная в предложении INTO, получает один столбец из строки результатов запроса. Выбираемые столбцы и соответствующие им базовые переменные образуют пары в порядке расположения в своих предложениях, и число столбцов должно равняться числу переменных. Кроме того, тип данных каждой переменной должен быть совместимым с типом данных соответствующего столбца.

Большинство СУБД автоматически выполняет разумные преобразования между типами данных СУБД и языка программирования. Например, большинство СУБД, прежде чем записывать данные типа DECIMAL в переменную языка COBOL, преобразует их в упакованные десятичные числа (COMP-3), а прежде чем записывать их в переменную языка С — в числа с плавающей запятой. Препроцессор знает тип данных базовой переменной и правильно выполняет преобразование.

Текстовые данные переменной длины также необходимо преобразовывать перед тем, как записывать их в базовую переменную. СУБД, как правило, преобразует данные типа VARCHAR в строку, заканчивающуюся нулевым символом, для про-

грамм на языке С и в строку переменной длины (первый символ указывает длину) для программ на языке Pascal. В программах на языках COBOL и FORTRAN базовая переменная должна быть объявлена, в общем случае, как структура данных, состоящая из целочисленного поля счетчика и массива символов. СУБД возвращает символы данных в массив, а длину данных — в поле счетчика этой структуры.

Если в СУБД имеются данные типов DATE, TIME или иные, то необходимы другие преобразования. Некоторые СУБД возвращают представленные во внутреннем формате значения даты и времени в целочисленную базовую переменную. Другие преобразуют данные типа DATE/TIME в текстовый формат и возвращают их в строковую переменную принимающего языка. В табл. 17.2 были описаны преобразования типов данных, обычно осуществляемые в СУБД, но, чтобы получить полную информацию, необходимо обратиться к документации по встроенному SQL для СУБД конкретного типа.

Условие NOT FOUND

Как и все прочие встроенные инструкции SQL, одиночная инструкция SELECT устанавливает значения переменных SQLCODE и SQLSTATE, указывающие статус завершения операции.

- Если успешно получена *одна* строка, SQLCODE устанавливается равной нулю, а SQLSTATE — равной 00000; базовые переменные, указанные в предложении INTO, содержат полученные из базы данных значения.
- Если при обработке запроса произошла *ошибка*, SQLCODE получает отрицательное значение, а SQLSTATE устанавливается равной ненулевому классу ошибки (первые два символа пятисимвольной строки SQLSTATE); базовые переменные не содержат полученных из базы данных значений. Однако эти базовые переменные могут содержать значения, оставшиеся от последнего успешного выполнения инструкции SELECT, обращавшейся к ним, так что перед использованием значений, хранящихся в базовых переменных, необходимо проверить значения переменных SQLCODE или SQLSTATE.
- Если запрос *не* возвращает ни одной строки, в переменной SQLCODE возвращается специальное предупреждающее значение NOT FOUND, а SQLSTATE возвращает класс ошибки NO DATA.
- Если запрос возвращает *более одной* строки, такая ситуация рассматривается как ошибка, и SQLCODE содержит отрицательное значение.

Стандарт SQL описывает предупреждение NOT FOUND, но не оговаривает его конкретное значение. DB2 использует значение +100, и этому соглашению следует большинство реализаций SQL, включая другие продукты IBM, Ingres, и SQLBase. Это значение указано и в стандарте SQL, но, как уже говорилось, стандарт строго рекомендует использовать новую переменную SQLSTATE вместо старой SQLCODE.

Выборка значений NULL

Если записи, которые должны быть извлечены из базы данных, могут содержать значения NULL, то одиночная инструкция SELECT должна обеспечить передачу значений NULL из СУБД в прикладную программу. Для этого в предложении INTO ис-

пользуются переменные-индикаторы, точно так же как они использовались в предложении VALUES инструкции INSERT и в предложении SET инструкции UPDATE.

Если в предложении INTO имеется базовая переменная, то сразу же за ней должно следовать имя сопутствующей переменной-индикатора. На рис. 17.21 показана измененная программа, представленная ранее на рис. 17.20. В этом варианте программы используется базовая переменная repquota и переменная-индикатор repquota_ind. Так как в определении таблицы SALESREPS столбцы NAME и SALES объявлены как NOT NULL, они не могут содержать значений NULL, и поэтому переменные-индикаторы для этих столбцов не требуются.

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        int   repnum;       /* Номер служащего (вводится) */
        char  repname[16];  /* Полученное имя служащего   */
        float repquota;     /* Полученный план продаж     */
        float repsales;     /* Полученные продажи         */
        short repquota_ind; /* Индикатор значения NULL    */
    exec sql end declare section;

    /* Приглашение пользователю ввести номер служащего    */
    printf("Введите номер служащего:");
    scanf("%d", &repnum);

    /* Выполнение запроса SQL                             */
    exec sql select name, quota, sales
        into :repname, :repquota :repquota_ind, :repsales
        from salesreps where empl_num = :repnum;

    /* Вывод полученных данных                            */
    if (sqlca.sqlcode = = 0)
    {
        printf("Имя: %s\n", repname);
        if (repquota_ind < 0)
            printf("План продаж - NULL\n");
        else
            printf("План продаж: %f\n", repquota);
        printf("Продажи: %f\n", repsales);
    }
    else if (sqlca.sqlcode = = 100)
        printf("Нет служащего с таким номером.\n");
    else printf("Ошибка SQL: %ld\n", sqlca.sqlcode);

    exit();
}
```

Рис. 17.21. Одиночная инструкция SELECT с переменными-индикаторами

Содержимое переменной-индикатора после выполнения инструкции SELECT информирует программу о том, как интерпретировать возвращаемые данные.

- Нулевое значение индикатора означает, что базовой переменной было присвоено значение, извлеченное из базы данных. Прикладная программа может использовать это значение в своих целях.

- Отрицательное значение индикатора означает, что извлеченное значение равно NULL. Значение базовой переменной не соответствует извлеченному значению и не должно использоваться прикладной программой.
- Положительное значение индикатора означает некоторое предупреждение, например, об округлении чисел или усечении строк.

Так как заранее неизвестно, когда будет извлечено значение NULL, для каждого столбца результатов запроса, который может содержать значение NULL, в предложении INTO должна быть задана переменная-индикатор. Если инструкция SELECT извлекает столбец со значением NULL, а переменная-индикатор для данного столбца в инструкции отсутствует, то СУБД трактует это как ошибку и возвращает отрицательную переменную SQLCODE. Таким образом, для успешного извлечения строк со значениями NULL обязательно должны применяться переменные-индикаторы. Кроме того, когда база данных возвращает значения NULL, значения базовых переменных остаются неизменными, так что разработчик программы должен обязательно проверить переменную-индикатор, перед тем как работать со значением базовой переменной, — в противном случае могут использоваться значения, которые вернула база данных при последнем успешно обработанном запросе.

Хотя основным применением переменных-индикаторов является обработка значений NULL, переменные-индикаторы используются в СУБД и для сигнализации о ненормальных ситуациях. Например, если в результате арифметического переполнения или деления на нуль содержимое какого-нибудь столбца результатов запроса становится недействительным, DB2 возвращает в переменной SQLCODE значение +802, означающее предупреждение, и устанавливает значение переменной-индикатора равным -2. Прикладная программа может просмотреть переменные-индикаторы, чтобы определить, какой именно столбец содержит недействительные данные.

С помощью переменных-индикаторов СУБД сигнализирует также об усечении символьных данных. Если какой-нибудь столбец результатов запроса содержит слишком длинные строки и они не умещаются в соответствующей базовой переменной, DB2 копирует в базовую переменную первую часть строки символов и заносит в соответствующую переменную-индикатор полную длину строки. Прикладная программа может просмотреть переменную-индикатор и при необходимости повторить инструкцию SELECT с другой базовой переменной, позволяющей хранить более длинную строку.

Такое дополнительное использование переменных-индикаторов является обычным делом в коммерческих СУБД, однако конкретные коды предупреждений в разных СУБД различны. Они не устанавливаются стандартом ANSI/ISO. Вместо этого стандарт SQL определяет классы и подклассы ошибок, а прикладная программа должна выполнить инструкцию GET DIAGNOSTICS, чтобы получить детальную информацию о базовой переменной, вызвавшей ошибку.

Выборка данных с использованием структур

В некоторых языках программирования существуют *структуры данных*, которые представляют собой именованные коллекции переменных. В таких языках программирования в предложении INTO можно указывать в качестве единствен-

ной базовой переменной структуру. Вместо того чтобы задавать отдельную базовую переменную для каждого столбца результатов запроса, можно задать структуру, в которую будет помещена вся запись целиком. На рис. 17.22 изображена программа, представленная на рис. 17.21, но измененная так, чтобы в ней использовалась базовая переменная типа struct языка С.

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        int repnum;         /* Номер служащего (вводится) */
        struct{
            char name[16]; /* Полученное имя служащего   */
            float quota;   /* Полученный план продаж     */
            float sales;   /* Полученные продажи         */
        } repinfo;
    short rep_ind[3];       /* Индикатор значения NULL    */
    exec sql end declare section;

    /* Приглашение пользователю ввести номер служащего   */
    printf("Введите номер служащего:");
    scanf("%d", &repnum);

    /* Выполнение запроса SQL                             */
    exec sql select name, quota, sales
              into :repinfo :rep_ind
              from salesreps where empl_num = :repnum;

    /* Вывод полученных данных                            */
    if (sqlca.sqlcode = = 0)
    {
        printf("Имя: %s\n", repinfo.name);
        if (repquota_ind < 0)
            printf("План продаж - NULL\n");
        else
            printf("План продаж: %f\n", repinfo.quota);
        printf("Продажи: %f\n", repinfo.sales);
    }
    else if (sqlca.sqlcode = = 100)
        printf("Нет служащего с таким номером.\n");
    else printf("Ошибка SQL: %ld\n", sqlca.sqlcode);

    exit();
}
```

Рис. 17.22. Применение структуры в качестве базовой переменной

Когда препроцессор встречает в предложении INTO ссылку на структуру, он заменяет ссылку списком переменных, входящих в структуру; переменные в списке располагаются в том порядке, в каком они объявлялись в этой структуре. Таким образом, число элементов структуры и их типы данных должны соответствовать столбцам результатов запроса. Структура данных в предложении INTO фактически является лишь сокращенной записью базовых переменных и никаких фундаментальных изменений в это предложение не вносит.

Поддержка структур в качестве базовых переменных по-разному реализуется в различных СУБД. Кроме того, структуры можно применять не во всех языках

программирования. Например, DB2 поддерживает структуры в языках C и PL/1, но не поддерживает в языке ассемблера и в COBOL.

Входные и выходные базовые переменные

Базовые переменные обеспечивают двустороннюю связь между программой и СУБД. В программе, представленной на рис. 17.21, две базовые переменные, `repnum` и `repname`, иллюстрируют две различные роли, которые могут играть базовые переменные.

- Переменная `repnum` — это входная базовая переменная, используемая для передачи данных из программы в СУБД. Программа присваивает переменной значение перед выполнением встроенной инструкции, и это значение становится частью инструкции `SELECT`, которая будет выполняться в СУБД. Значение переменной в СУБД не изменяется.
- Переменная `repname` — это выходная базовая переменная, используемая для передачи данных из СУБД в программу. СУБД присваивает этой переменной значение во время выполнения встроенной инструкции `SELECT`. После того как инструкция будет выполнена, программа может использовать полученное значение.

Входные и выходные базовые переменные объявляются в программе одинаково и во встроенной инструкции `SELECT` обозначаются также одинаково: с помощью двоеточия. Однако при написании программы полезно не забывать о различии между ними. Входные переменные можно применять в любой инструкции SQL, где присутствует константа. Выходные переменные можно применять только в одиночной инструкции `SELECT` и в инструкции `FETCH`, рассматриваемой ниже в настоящей главе.

Многострочные запросы

Когда результатом выполнения запроса является не одна запись, а целая таблица, встроенный SQL должен обеспечить для прикладной программы возможность построчного получения результатов запроса. Для этого во встроенном SQL вводится понятие указателя набора записей, или *курсора* (cursor), и добавляется несколько новых инструкций. Вот как вкратце выглядит механизм выполнения запроса данного типа и требуемые для этого новые инструкции.

1. Инструкция `DECLARE CURSOR` определяет выполняемый запрос и связывает имя курсора с данным запросом.
2. Инструкция `OPEN` дает команду СУБД, чтобы она начала выполнять запрос и создавать таблицу результатов запроса. Эта инструкция позиционирует курсор перед первой строкой таблицы результатов.
3. Инструкция `FETCH` перемещает курсор на первую строку таблицы результатов и извлекает данные, записывая их в базовые переменные прикладной программы. Последующие инструкции `FETCH` перемещаются по таблице результатов строка за строкой, смещая курсор на следующую строку и извлекая из нее данные с последующей их записью в базовые переменные.

4. Инструкция CLOSE прекращает доступ к таблице результатов и ликвидирует связь между курсором и таблицей результатов.

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        char  repname[16];  /* Полученное имя служащего */
        float repquota;     /* Полученный план продаж   */
        float repsales;     /* Полученные продажи       */
        short repquota_ind; /* Индикатор значения NULL  */
    exec sql end declare section;

    /* Объявление курсора */
    exec sql declare repcurs cursor for     <------------------①
              select name, quota, sales
                from salesreps
               where sales > quota
               order by name;

    /* Обработка ошибок */
    whenever sqlerror goto error;
    whenever not found goto done;

    /* Открытие курсора для начала выполнения запроса */
    exec sql open repcurs;     <------------------------------②

    /* Цикл по всем строкам результатов запроса */
    for (;;)
    {
        /* Выборка очередной строки результатов запроса */
        exec sql fetch repcurs     <------------------------③
            into :repname, :repquota :repquota_ind,
                 :repsales;

        /*Вывод полученных данных */
        printf("Имя: %s\n", repname);
        if (repquota_ind < 0)
            printf("План продаж - NULL\n");
        else
            printf("План продаж: %f\n", repquota);
        printf("Продажи: %f\n", repsales);
    }
error:
    printf("Ошибка SQL: %ld\n", sqlca.sqlcode);
    exit();
done:
    /* Запрос завершен; закрытие курсора */
    exec sql close repcurs;     <------------------------------④
    exit();
}
```

Рис. 17.23. Выполнение многострочного запроса

На рис. 17.23 приведен текст программы, выполняющей с помощью встроенного SQL простой запрос, возвращающий набор записей. Цифры в кружках на рисунке соответствуют номерам перечисленных выше этапов. Программа извлекает и отображает на экране в алфавитном порядке имя, личный план и текущий объем продаж для каждого служащего, чей текущий объем продаж превышает лич-

ный план. Соответствующий интерактивный запрос SQL, извлекающий эту информацию, имеет следующий вид.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS
 WHERE SALES > QUOTA
 ORDER BY NAME;
```

Обратите внимание на то, что точно такой же запрос, слово в слово, присутствует во встроенной инструкции DECLARE CURSOR на рис. 17.23. Инструкция также связывает имя курсора repcurs с данным запросом. Это имя курсора используется затем в инструкции OPEN, чтобы начать выполнение запроса и позиционировать курсор перед первой строкой таблицы результатов запроса.

Инструкция FETCH внутри цикла for при каждом прохождении цикла передает в программу очередную запись из таблицы результатов запроса. Предложение INTO в инструкции FETCH играет ту же роль, что и в одиночной инструкции SELECT. В этом предложении задаются базовые переменные, принимающие данные, — по одной базовой переменной для каждого столбца таблицы результатов запроса. Как и в предыдущих примерах, для получения данных, которые могут содержать значения NULL, используется переменная-индикатор (repquota_ind).

Когда в программу переданы все строки и больше передавать нечего, инструкция FETCH возвращает предупреждение NOT FOUND. Код предупреждения здесь такой же, как и в случае, когда одиночная инструкция SELECT не может извлечь строку данных. Инструкция WHENEVER NOT FOUND дает препроцессору команду сгенерировать подпрограмму, которая проверяет значение переменной SQLCODE после выполнения инструкции FETCH. Эта подпрограмма осуществляет переход к метке done при наличии предупреждения NOT FOUND и к метке error при возникновении ошибки. В конце программы инструкция CLOSE завершает запрос и прекращает доступ программы к таблице результатов запроса.

Курсоры

Как видно из программы, представленной на рис. 17.23, курсор во встроенном SQL очень похож на имя или дескриптор файла в таких языках программирования, как C или COBOL. Программа открывает курсор, чтобы получить доступ к таблице результатов запроса, точно так же как она открывает файл, чтобы получить доступ к его содержимому. Аналогично программа закрывает файл, чтобы прекратить к нему доступ, и закрывает курсор, чтобы прекратить доступ к результатам запроса. И наконец, курсор отслеживает текущую позицию в таблице результатов запроса точно так же, как дескриптор файла отслеживает текущую позицию внутри открытого файла. Эти параллели между файлами и курсорами в SQL облегчают прикладным программистам понимание того, что такое курсор.

Однако, помимо сходства, между файлами и курсорами имеются и некоторые различия. Как правило, открытие курсора связано с большими затратами, чем открытие файла, так как в результате открытия курсора СУБД начинает выполнять соответствующий запрос. Кроме того, курсоры обычно могут перемещаться только вперед, последовательно через все строки таблицы результатов, — как при последовательном доступе к файлу.

Курсоры обеспечивают большую гибкость при выполнении запросов в программах со встроенным SQL. Объявляя и открывая в программе несколько курсоров, можно параллельно обрабатывать несколько таблиц результатов запроса. Например, программа может извлечь результаты запроса, отобразить их на экране, а затем, в ответ на запрос пользователя о предоставлении более подробных данных, запустить второй запрос. Далее подробно описываются четыре встроенные инструкции SQL, осуществляющие управление курсорами.

Инструкция DECLARE CURSOR

Инструкция DECLARE CURSOR, синтаксическая диаграмма которой изображена на рис. 17.24, определяет запрос, который будет выполняться при открытии курсора. Инструкция также связывает имя курсора (которое должно быть допустимым идентификатором SQL) и запрос. Имя используется для идентификации запроса и его результатов в других встроенных инструкциях SQL и *не* является базовой переменной.

Рис. 17.24. Синтаксическая диаграмма инструкции DECLARE CURSOR

Инструкция SELECT, вложенная в инструкцию DECLARE CURSOR, определяет запрос, связанный с курсором. Она может быть любой допустимой интерактивной инструкцией SELECT, описанной в главах 6–9. В частности, эта инструкция должна содержать предложение FROM и может содержать предложения WHERE, GROUP BY, HAVING и ORDER BY. Инструкция SELECT может также включать операцию UNION, описанную в главе 6, "Простые запросы". Таким образом, во встроенных запросах SQL можно использовать все возможности, предоставляемые интерактивным SQL.

Запрос в инструкции DECLARE CURSOR может содержать входные базовые переменные. Они выполняют ту же функцию, что и во встроенных инструкциях INSERT, DELETE, UPDATE, а также в одиночной инструкции SELECT. Входная базовая переменная в запросе может находиться там, где может использоваться константа. Обратите внимание на то, что выходные базовые переменные не могут использоваться в этом запросе. В отличие от одиночной инструкции SELECT, инструкция SELECT, помещенная внутри инструкции DECLARE CURSOR, не содержит предложения INTO и самостоятельно не извлекает данные. Предложение INTO является частью инструкции FETCH, которая рассматривается ниже в настоящей главе.

Как следует из названия, инструкция DECLARE CURSOR представляет собой объявление курсора. В большинстве СУБД, включая СУБД компании IBM, эта инструкция является директивой препроцессора SQL; она не является исполняемой инструкцией, и препроцессор не создает для нее объектный код. Как и все объявления, инструкция DECLARE CURSOR должна располагаться в программе перед любыми инструкциями, которые содержат ссылки на объявляемый ею курсор. В большинстве реализаций SQL имя курсора рассматривается как глобальное имя, на которое можно ссылаться в любых процедурах, функциях или подпрограммах, расположенных после инструкции DECLARE CURSOR.

Следует отметить, что не во всех СУБД инструкция DECLARE CURSOR трактуется строго как декларативная инструкция, что может приводить к тонким пробле-

мам, которые трудно предвидеть. Некоторые препроцессоры SQL для инструкции DECLARE CURSOR генерируют объектный код (или объявления принимающего языка, или вызовы СУБД, или то и другое), придавая ей некоторые качества исполняемой инструкции. Для этих препроцессоров инструкция DECLARE CURSOR должна не только физически предшествовать инструкциям OPEN, FETCH и CLOSE, которые ссылаются на данный курсор, но и выполняться перед ними или же находиться с ними в одном блоке.

Проблем с инструкцией DECLARE CURSOR можно избежать, придерживаясь следующих правил.

- Располагайте инструкцию DECLARE CURSOR для данного курсора прямо перед инструкцией OPEN. Это обеспечивает правильную физическую последовательность инструкций, размещение инструкций DECLARE CURSOR и OPEN в одном блоке и (в случае необходимости) выполнение инструкции DECLARE CURSOR ранее инструкции OPEN.
- Убедитесь, что инструкции FETCH и CLOSE для данного курсора следуют за инструкцией OPEN как физически, так и в смысле потока выполнения программы.

Инструкция OPEN

Концептуально инструкция OPEN, синтаксическая диаграмма которой изображена на рис. 17.25, открывает таблицу результатов запроса для прикладной программы. На практике же инструкция OPEN дает СУБД команду выполнить запрос или, по крайней мере, начать его выполнение. Таким образом, инструкция OPEN заставляет СУБД выполнить ту же работу, которую вынуждает ее выполнить интерактивная инструкция SELECT, и остановиться непосредственно перед получением первой строки результатов запроса.

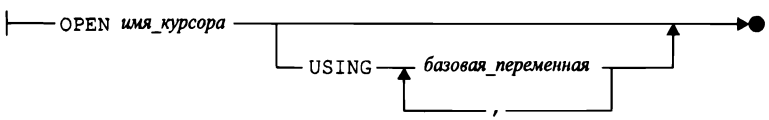


Рис. 17.25. Синтаксическая диаграмма инструкции OPEN

Единственным параметром инструкции OPEN является имя курсора, который должен быть открыт. Этот курсор необходимо предварительно объявить с помощью инструкции DECLARE CURSOR. Если при выполнении запроса, связанного с курсором, происходит ошибка, инструкция OPEN присваивает отрицательное значение переменной SQLCODE. Именно инструкция OPEN выдает сообщения о большинстве ошибок, возникающих во время выполнения запроса, таких как ссылка на неизвестную таблицу, неоднозначное имя столбца или попытка извлечь данные из таблицы без соответствующего разрешения. Во время выполнения последующих инструкций FETCH обычно происходит очень мало ошибок.

Будучи однажды открытым, курсор остается в таком состоянии до его закрытия с помощью инструкции CLOSE. СУБД автоматически закрывает все открытые курсоры в конце транзакции (т.е. когда СУБД выполняет инструкцию COMMIT или

ROLLBACK). После того как курсор был закрыт, его можно вновь открыть, выполнив инструкцию OPEN повторно. Обратите внимание: каждый раз, когда СУБД выполняет инструкцию OPEN, она запускает запрос заново, с самого начала.

Инструкция FETCH

Инструкция FETCH, синтаксическая диаграмма которой изображена на рис. 17.26, передает в прикладную программу *следующую* строку результатов запроса. Курсор, заданный в инструкции FETCH, определяет, какая строка результатов запроса должна быть извлечена. Этот курсор должен быть предварительно открыт с помощью инструкции OPEN.

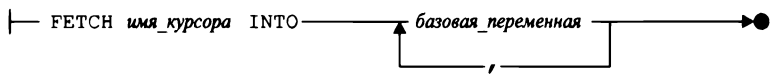


Рис. 17.26. Синтаксическая диаграмма инструкции FETCH

Инструкция FETCH записывает строку данных в базовые переменные, указанные в предложении INTO данной инструкции. Для обеспечения корректного извлечения значений NULL каждой базовой переменной может сопутствовать переменная-индикатор. Свойства переменной-индикатора и значения, которые она может принимать, идентичны описанным выше для одиночной инструкции SELECT. Число базовых переменных должно быть равно числу столбцов в таблице результатов запроса, а типы данных базовых переменных должны быть совместимыми с типами данных соответствующих столбцов из таблицы результатов запроса.

Как видно из рис. 17.27, инструкция FETCH обеспечивает перемещение курсора по таблице результатов запроса от строки к строке в соответствии со следующими правилами.

- Инструкция OPEN устанавливает курсор в положение, предшествующее первой строке таблицы результатов запроса. В этом положении у курсора нет текущей строки.
- Инструкция FETCH перемещает курсор на следующую доступную строку таблицы результатов запроса, если таковая имеется. Данная строка становится текущей строкой курсора.
- Если инструкция FETCH перемещает курсор в положение, следующее за последней строкой таблицы результатов запроса, то эта инструкция возвращает предупреждение NOT FOUND. В данной ситуации у курсора опять нет текущей строки.
- Инструкция CLOSE прекращает доступ к таблице результатов запроса и переводит курсор в закрытое состояние.

Если извлеченные строки отсутствуют (таблица результатов запроса пустая), то инструкция OPEN все равно устанавливает курсор *перед* таблицей результатов запроса и успешно завершается. Программа не может обнаружить, что инструкция OPEN извлекла пустую таблицу результатов запроса. Однако первая же инструкция FETCH возвращает предупреждение NOT FOUND и устанавливает курсор за таблицей результатов запроса.

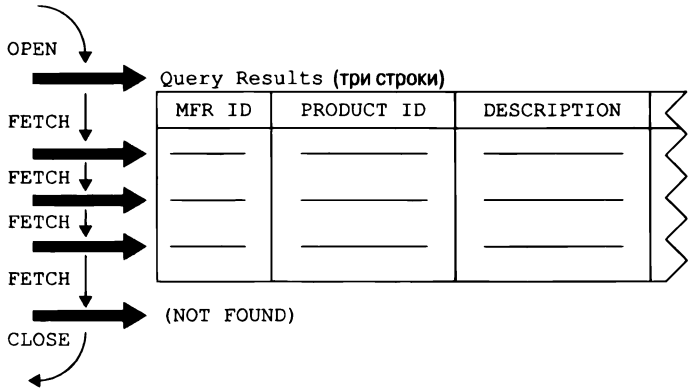


Рис. 17.27. Позиционирование курсора инструкциями OPEN, FETCH **и** CLOSE

Инструкция CLOSE

Можно сказать, что инструкция CLOSE (ее синтаксическая диаграмма изображена на рис. 17.28) закрывает таблицу результатов запроса, созданную инструкцией OPEN, и запрещает прикладной программе доступ к ней. Единственным параметром инструкции CLOSE является имя курсора, связанного с таблицей результатов запроса; это должен быть курсор, предварительно открытый с помощью инструкции OPEN. Инструкция CLOSE может быть выполнена в любой момент после того, как курсор был открыт. В частности, необязательно передавать в программу с помощью инструкции FETCH все строки результатов запроса перед тем, как закрывать курсор (хотя обычно поступают именно так). В конце транзакции все курсоры автоматически закрываются. После того как курсор закрыт, таблица результатов запроса становится недоступной прикладной программе.

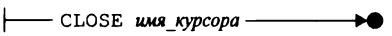


Рис. 17.28. Синтаксическая диаграмма инструкции CLOSE

Курсоры с произвольным доступом

Согласно стандарту SQL1, курсор мог перемещаться по таблице результатов запроса только вперед. До последнего времени подобного режима работы с курсорами придерживалось большинство коммерческих СУБД. Если программе необходимо повторно получить строку, которую курсор уже прошел, программа должна закрыть курсор, открыть его вновь (вынуждая СУБД еще раз выполнить запрос) и поочередно прочитать все строки вплоть до требуемой.

В начале 1990-х годов в нескольких коммерческих СУБД данная концепция была расширена и было введено понятие *курсора с произвольным доступом*, который, в отличие от стандартного курсора, обеспечивает произвольный доступ к строкам результатов запроса. Требуемую строку программа указывает в расширенной инструкции FETCH (рис. 17.29).

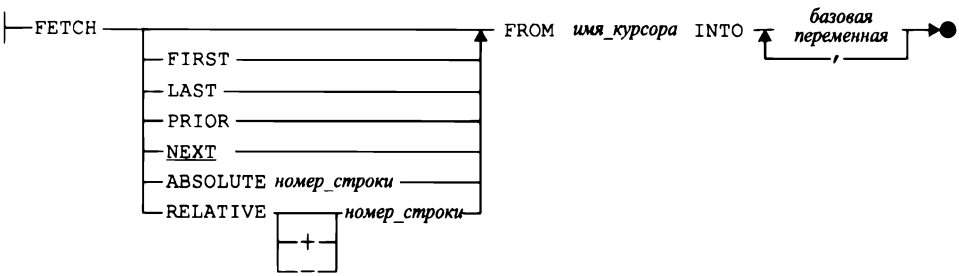


Рис. 17.29. Расширенная инструкция FETCH для курсоров с произвольным доступом

- Инструкция FETCH FIRST извлекает первую строку результатов запроса.
- Инструкция FETCH LAST извлекает последнюю строку результатов запроса.
- Инструкция FETCH PRIOR извлекает строку результатов запроса, которая непосредственно предшествует текущей строке курсора.
- Инструкция FETCH NEXT извлекает строку результатов запроса, которая непосредственно следует за текущей строкой курсора. Это соответствует стандартному перемещению курсора, и по умолчанию (если параметр перемещения не задан) инструкция FETCH выполняется именно таким образом.
- Инструкция FETCH ABSOLUTE извлекает отдельную строку по ее номеру.
- Инструкция FETCH RELATIVE перемещает курсор вперед или назад на определенное число строк по отношению к его текущей позиции.

Курсоры с произвольным доступом особенно полезны в программах, которые позволяют пользователю просматривать содержимое базы данных. В ответ на запрос пользователя переместиться вперед или назад на строку или на целый экран программа может легко извлечь требуемые строки результатов запроса. Однако реализовать в СУБД курсор с произвольным доступом гораздо труднее, чем обычный курсор с последовательным доступом. Для поддержки курсоров с произвольным доступом СУБД должна помнить все предыдущие результаты запросов, переданные в программу, и порядок, в котором были переданы эти строки. СУБД должна также обеспечить, чтобы ни одна параллельно выполняемая транзакция не модифицировала данные, уже полученные программой посредством курсора с произвольным доступом, так как, используя расширенную инструкцию FETCH, программа может повторно извлечь любую строку, через которую курсор уже прошел.

Необходимо понимать, что отдельные инструкции FETCH, использующие курсор с произвольным доступом, в некоторых СУБД могут потребовать очень большого объема работы. Выполнение инструкции FETCH NEXT, когда курсор установлен на первую строку таблицы результатов запроса, может занять гораздо больше времени, чем нормальное выполнение запроса, когда СУБД извлекает данные последовательно, шаг за шагом, а программа в это время с помощью инструкции FETCH отбирает очередные строки результатов запроса. Прежде чем пи-

сать промышленные приложения, использующие возможности курсора с произвольным доступом, следует узнать рабочие характеристики применяемой СУБД, познакомиться с ее производительностью и оценить ее достаточность для работы с такого рода приложениями.

В связи с тем, что курсоры с произвольным доступом очень удобны, а также из-за того, что различные СУБД, поставляемые на рынок, реализовывали данную концепцию немного по-разному, это понятие было включено в стандарт SQL. Если СУБД совместима со стандартом SQL только на уровне Entry, то поддержка курсоров с произвольным доступом не требуется, но она обязательна на уровнях Intermediate и Full. В стандарте также определено, что если используется любое перемещение курсора, отличное от FETCH NEXT (перемещение по умолчанию), то в инструкции DECLARE CURSOR должен быть явно задан курсор с произвольным доступом. Объявление курсора в программе на рис. 17.23 при использовании стандартного синтаксиса должно выглядеть следующим образом.

```
exec sql declare repcurs scroll cursor for
         select name, quota, sales
           from salesreps
          where sales > quota
          order by name;
```

Удаление и обновление данных на основе курсоров

Прикладные программы часто используют курсоры для того, чтобы дать пользователю возможность просматривать таблицу строка за строкой. Например, пользователь может попросить вывести все заказы, сделанные одним клиентом. Программа объявляет курсор для запроса к таблице ORDERS и отображает на экране каждый заказ (возможно, в виде какой-либо формы), ожидая от пользователя сигнала для перехода к следующей строке. Просмотр таблицы таким способом продолжается до тех пор, пока пользователь не достигнет последней строки в таблице результатов запроса. Курсор служит указателем на текущую строку результатов. Если запрос извлекает данные из одной таблицы и не является итоговым (как в данном примере), то курсор неявно адресует отдельную строку таблицы, так как одна строка результатов запроса строго соответствует одной строке данной таблицы.

В процессе просмотра пользователь может увидеть данные, которые необходимо изменить. Например, количество товара в каком-нибудь заказе может быть неправильным или клиент может захотеть удалить один из заказов. В такой ситуации пользователю требуется обновить или удалить этот заказ. При этом строка НЕ идентифицируется при помощи обычного условия отбора; программа использует курсор в качестве указателя на строку, которую требуется удалить или обновить.

Во встроенном SQL эта возможность обеспечивается посредством специальных версий инструкций DELETE и UPDATE, которые называются *позиционными*.

Позиционная инструкция DELETE, синтаксическая диаграмма которой изображена на рис. 17.30, удаляет из таблицы одну строку. Удаляемая строка является текущей строкой курсора, указатель которого ссылается на данную таблицу. Что-

бы выполнить инструкцию DELETE, СУБД локализует в базовой таблице строку, соответствующую текущей строке курсора, и удаляет ее. После удаления текущая строка у курсора отсутствует. Он фактически находится на пустом месте, оставшемся от удаленной строки, в ожидании продвижения на следующую строку с помощью очередной инструкции FETCH.

Рис. 17.30. Синтаксическая диаграмма позиционной инструкции DELETE

Позиционная инструкция UPDATE (ее синтаксическая диаграмма изображена на рис. 17.31) обновляет одну строку таблицы. Обновляемая строка является текущей строкой курсора, указатель которого ссылается на данную таблицу. Выполняя эту инструкцию, СУБД локализует в базовой таблице строку, соответствующую текущей строке курсора, и обновляет эту строку так, как это задано в предложении SET. После обновления строка остается текущей строкой курсора. На рис. 17.32 изображена программа просмотра заказов, в которой применяются позиционные инструкции UPDATE и DELETE.

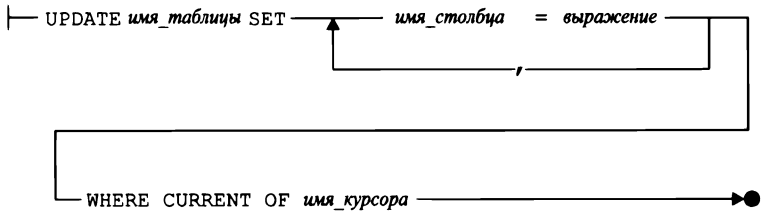


Рис. 17.31. Синтаксическая диаграмма позиционной инструкции UPDATE

1. Программа запрашивает у пользователя идентификатор клиента, после чего выполняет запрос к таблице ORDERS, чтобы отыскать все заказы, сделанные этим клиентом.
2. Программа извлекает одну строку результатов запроса, отображает на экране информацию о заказе и запрашивает дальнейшие инструкции.
3. Если пользователь вводит символ "N", программа не модифицирует текущий заказ, а сразу переходит к следующему.
4. Если пользователь вводит символ "D", программа удаляет текущий заказ с помощью позиционной инструкции DELETE.
5. Если пользователь вводит символ "U", программа запрашивает у пользователя новое количество товара и сумму заказа, а затем обновляет соответствующие два столбца текущей записи с помощью позиционной инструкции UPDATE.
6. Если пользователь вводит символ "X", программа прекращает выполнение запроса и завершает работу.

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        int   custnum;       /* Номер заказчика              */
        int   ordnum;        /* Номер заказа                 */
        char  orddate[12];   /* Дата заказа                  */
        char  ordmfr[4];     /* Идентификатор производителя */
        char  ordproduct[6]; /* Идентификатор товара         */
        int   ordqty;        /* Величина заказа              */
        float ordamount;     /* Сумма заказа                 */
    exec sql end declare section;
    char inbuf[101]          /* Вводимая строка              */

        /* Объявление запроса */
        exec sql declare ordcurs cursor for
                 select order_num, ord_date, mfr,
                        product, qty, amount
                   from orders where cust = custnum
                  order by order_num
                    for update of qty, amount;

    /* Приглашение ввести номер заказчика */
    printf("Введите номер заказчика:");
    scanf("%d", &custnum);

    /* Обработка ошибок */
    whenever sqlerror  goto error;                                  ①
    whenever not found goto done;

    /* Открытие курсора для запуска запроса */
    exec sql open ordcurs;

    /* Цикл по всем строкам результата */
    for (;;)
    {
        /* Выборка очередной строки из результатов запроса */
        exec sql fetch ordcurs into :ordnum, :orddate,
                                    :ordmfr, :ordproduct,
                                    :ordqty, :ordamount;

        /* Вывод полученных данных */
        printf("Номер заказа : %d\n", ordnum);
        printf("Дата заказа  : %s\n", orddate);
        printf("Производитель: %s\n", ordmfr);                      ②
        printf("Товар        : %s\n", ordproduct);
        printf("Количество   : %s\n", ordqty);
        printf("Сумма        : %f\n", ordamount);

        /* Приглашение выбрать действие */
        printf("Ваши действия (Next/Delete/Update/Exit): ");
        gets(inbuf);
```

Рис. 17.32. Применение позиционных инструкций DELETE **и** UPDATE

```
switch (inbuf[0])
        {
        case 'N':
            /* Переход к следующему заказу */
            break;
        case 'D':
            /* Удаление текущего заказа */
            exec sql delete from orders
                      where current of ordcurs;
            break;
        case 'U':
            /* Изменение текущего заказа */
            printf("Введите новое количество: ");
            scanf("%d", &ordqty);
            printf("Введите новую сумму: ");
            scanf("%f", &ordamount);
            exec sql update orders set qty = :ordqty,
                                       amount = :ordamount
                      where current of ordcurs;
            break;
        case 'X':
            /* Выход из программы */
            goto done;
        }
    }
done:
    exec sql close ordcurs;
    exec sql commit;
    exit();
error:
    printf("Ошибка SQL: %ld\n", sqlca.sqlcode);
    exit();
}
```

③ ④ ⑤ ⑥

Окончание рис. 17.32

Хотя пример, изображенный на рис. 17.32, очень прост по сравнению с реальными приложениями, в нем отражена вся логика встроенных инструкций SQL, необходимых для реализации программы, осуществляющей изменения в базе данных с помощью курсора.

Для того чтобы позиционные инструкции DELETE и UPDATE можно было использовать совместно с курсорами, эти курсоры, согласно стандарту SQL1, должны отвечать следующим очень строгим критериям.

- Запрос, связанный с курсором, должен извлекать данные из одной исходной таблицы; т.е. в предложении FROM запроса, который находится в инструкции DECLARE CURSOR, должна быть задана только одна таблица.
- В запросе не может присутствовать предложение ORDER BY; курсор не должен идентифицировать отсортированный набор строк результатов запроса.
- В запросе не может присутствовать предикат DISTINCT.
- Запрос не должен содержать предложение GROUP BY или HAVING.
- Пользователь должен иметь соответствующую привилегию (DELETE или UPDATE) для работы с базовой таблицей.

В базах данных компании IBM (DB2, SQL/DS) сделан еще один шаг в сторону усиления ограничений, установленных стандартом SQL1. Эти СУБД требуют, чтобы в инструкции DECLARE CURSOR курсор был явно объявлен как обновляемый. Расширенная форма инструкции DECLARE CURSOR, используемая компанией IBM, изображена на рис. 17.33. Помимо того, что предложение FOR UPDATE объявляет обновляемый курсор, оно может также содержать имена обновляемых столбцов. Если в предложении присутствует список столбцов, то позиционные инструкции UPDATE, применяемые для данного курсора, могут обновлять только указанные столбцы.

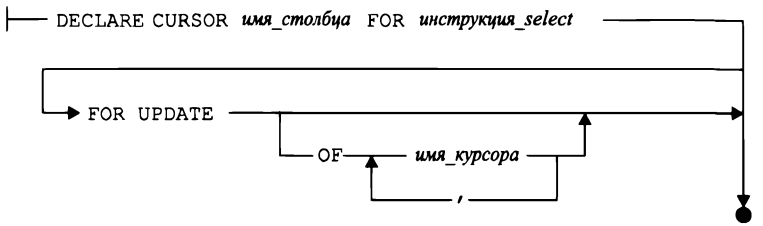


Рис. 17.33. Инструкция DECLARE CURSOR **с предложением** FOR UPDATE

На практике все коммерческие СУБД, в которых имеются позиционные инструкции DELETE и UPDATE, применяют их аналогично тому, как это делается в СУБД компании IBM. Если СУБД заранее "знает", что курсор будет использоваться не для обновления данных, а только для выборки, она может извлечь из этой информации огромное преимущество, так как выборка данных осуществляется гораздо проще. Предложение FOR UPDATE обеспечивает такое предварительное уведомление и *де-факто* может считаться стандартным элементом встроенного SQL.

В связи с широким применением предложения FOR UPDATE, в стандарте SQL подобное ему предложение было включено как необязательный параметр в инструкцию DECLARE CURSOR. Однако, в отличие от СУБД компании IBM, в стандарте SQL автоматически считается, что курсор открывается для обновления, если он не предназначен для произвольного доступа или явно не объявлен как FOR READ ONLY. Предложение FOR READ ONLY в инструкции DECLARE CURSOR располагается точно на том же месте, что и предложение FOR UPDATE, и явно информирует СУБД о том, что программа не будет выполнять позиционные операции DELETE или UPDATE для данного курсора. Так как в результате использования этих инструкций может повышаться нагрузка на СУБД и снижаться ее производительность, программисту следует хорошо понимать, какие предположения делает ваша конкретная СУБД по поводу обновлений с применением курсора и с помощью каких инструкций и предложений можно легче всего достичь поставленной цели.

Курсоры и обработка транзакций

То, как ваша программа работает с курсорами, может оказать большое влияние на производительность СУБД. Вспомним, что модель транзакций, описанная в главе 12, "Обработка транзакций", гарантирует непротиворечивость данных в течение всей транзакции. По отношению к курсорам это означает гарантию того,

что если ваша программа объявляет курсор, открывает его, принимает результаты запроса, закрывает курсор, открывает его опять и снова принимает результаты запроса, то оба раза результаты будут идентичными. Программе гарантируются идентичные результаты и в том случае, когда она извлекает одну и ту же строку с помощью двух различных курсоров. Фактически непротиворечивость данных гарантируется до тех пор, пока программа не закончит транзакцию с помощью инструкции COMMIT или ROLLBACK. Так как непротиворечивость между данными в различных транзакциях не гарантируется, то инструкции COMMIT и ROLLBACK автоматически закрывают все открытые курсоры.

С технической точки зрения, СУБД гарантирует непротиворечивость данных путем блокирования всех строк результатов запроса, предотвращая их модификацию другими пользователями. Если запрос извлекает много записей, то может быть заблокирована большая часть таблицы. Кроме того, если после получения каждой строки программа ожидает от пользователя ввода информации (например, чтобы пользователь проверил данные, увиденные им на экране), то отдельные области базы данных могут быть заблокированы в течение весьма длительного времени. Не исключено, что пользователь может не завершить транзакцию и уйти на обед, блокируя доступ других пользователей к информации в течение часа или даже больше!

Чтобы свести к минимуму количество требуемых блокировок, при написании интерактивных программ следует руководствоваться следующими принципами.

- Делайте транзакции как можно более короткими.
- Выполняйте инструкцию COMMIT через разумные промежутки времени. Иногда возникает соблазн выполнять COMMIT сразу же после каждой инструкции INSERT, UPDATE и DELETE, но обработка COMMIT приводит к повышенным накладным расходам, так что следует искать компромисс между освобождением блокировок по тайм-ауту и сниженной эффективностью программы.
- Избегайте программ, в которых осуществляется интенсивное взаимодействие с пользователем или выполняется просмотр большого количества записей.
- Если вы знаете, что программа не будет повторно обращаться к какой-либо записи, используйте один из наименее жестких режимов изоляции, описанных в главе 12, "Обработка транзакций"; это позволит СУБД разблокировать запись, как только начинает выполняться следующая инструкция FETCH.
- Избегайте применения курсоров с произвольным доступом, если вы не предприняли никаких мер, исключающих или сводящих к минимуму избыточные блокировки, вызываемые такими курсорами.
- Если есть такая возможность, явно определяйте курсор как FOR READ ONLY.

Резюме

Язык SQL применяется для осуществления не только интерактивного, но и программного доступа к реляционным базам данных.

- Наиболее распространенной разновидностью программного SQL является встроенный SQL, в котором инструкции SQL встраиваются в прикладную программу, смешиваясь с инструкциями принимающего языка программирования, такого как C или COBOL.
- Встроенные инструкции SQL обрабатываются специальным препроцессором SQL. Они начинаются со спецификатора (обычно `EXEC SQL`) и заканчиваются ограничителем, которые различны в разных принимающих языках.
- Везде, где во встроенных инструкциях SQL могут стоять константы, вместо них можно использовать переменные из прикладной программы, которые называются базовыми. С помощью этих входных переменных в базу данных можно передавать значения, вводимые пользователем.
- Базовые переменные применяются также и для получения результатов запросов. Значения этих выходных переменных могут затем обрабатываться прикладной программой.
- Запросы, которые извлекают одну запись, могут быть реализованы посредством одиночной инструкции `SELECT`, входящей во встроенный SQL. Одиночная инструкция `SELECT` содержит как запрос, так и базовые переменные, в которые передаются извлекаемые данные.
- Запросы, которые извлекают несколько записей, реализуются во встроенном SQL с помощью курсоров. Инструкция `DECLARE CURSOR` определяет запрос, инструкция `OPEN` начинает выполнение запроса, инструкция `FETCH` последовательно извлекает строки из таблицы результатов запроса, а инструкция `CLOSE` заканчивает выполнение запроса. В приложениях, в которых курсор должен перемещаться по результатам запроса не последовательным образом, могут использоваться курсоры с произвольным доступом (если таковые поддерживает СУБД).
- Позиционные инструкции `UPDATE` и `DELETE` используются для обновления или удаления строки, выбранной в данный момент курсором.

18
ГЛАВА

Динамический SQL*

Возможности встроенного SQL, описанные в предыдущей главе, известны как *статический SQL*. Статический SQL пригоден для написания типичных программ обработки данных. Например, для учебной базы данных с помощью статического SQL можно написать программу, которая позволит ввести новый или обновить существующий заказ, получить информацию о заказе и клиенте, а также позволит работать с файлом клиента и создавать все необходимые отчеты. Для решения каждой из перечисленных задач программист определяет схему доступа к базе данных и жестко зашивает ее в программу в виде ряда встроенных инструкций SQL.

Однако существует достаточно большой класс приложений, в которых невозможно заранее определить схему доступа к базе данных. Например, программа создания запросов или программа, генерирующая отчеты, должна иметь возможность решать, какие инструкции SQL она будет использовать для доступа к базе данных во время выполнения. Программа для работы с электронными таблицами, установленная на персональном компьютере и имеющая доступ к серверной базе данных, также должна иметь возможность сформировать запрос к этой базе данных на лету. Перечисленные программы, а также другие клиентские приложения общего назначения невозможно написать, используя инструкции статического SQL. Для создания этих программ необходима усовершенствованная разновидность встроенного SQL, которая называется *динамический SQL*. Она рассматривается в настоящей главе.

Недостатки статического SQL

Как следует из названия *статический SQL*, программа, использующая описанные в главе 17, "Встроенный SQL", возможности (базовые переменные, курсоры, инструкции DECLARE CURSOR, OPEN, FETCH и CLOSE), имеет заранее определенную, жестко фиксированную схему доступа к базе данных. В каждой встроенной инструкции SQL программист заранее указывает, к каким таблицам и столбцам он

будет обращаться. Входные базовые переменные придают статическому SQL определенную гибкость, но не могут коренным образом изменить его статическую природу. Вспомним, что базовую переменную разрешается применять в инструкции SQL везде, где может стоять константа. С помощью базовой переменной можно изменить условие отбора

```
exec sql select name, quota, sales
          from salesreps
         where quota > :cutoff_amount;
```

можно изменять добавляемые или обновляемые данные

```
exec sql update salesreps
           set quota = quota + :increase
         where quota > :cutoff_amount;
```

однако базовую переменную нельзя использовать вместо имени таблицы или столбца. Приведенные ниже примеры употребления базовых переменных `which_table` и `which_column` являются неправильными.

```
exec sql update :which_table
           set :which_column = 0;

exec sql declare cursor cursor7 for
         select *
           from :which_table;
```

Даже если бы вы могли использовать базовую переменную таким образом (а этого делать нельзя), немедленно возникла бы другая проблема. Количество столбцов, извлеченных второй инструкцией, будет изменяться в зависимости от того, какая таблица была задана в базовой переменной. Если задана таблица `OFFICES`, то результаты запроса будут иметь шесть столбцов, если задана таблица `SALESREPS`, то — девять столбцов. Кроме того, типы данных столбцов у двух указанных таблиц различны. Но чтобы написать инструкцию `FETCH` для какого-либо запроса, необходимо заранее знать, сколько будет столбцов в таблице результатов запроса и каковы их типы данных, так как для каждого столбца необходимо указать базовую переменную, в которую будет передаваться содержимое столбца.

```
exec sql fetch cursor7
         into :var1, :var2, :var3;
```

Таким образом, если программа может решить, какие инструкции SQL применять и к каким таблицам и столбцам обращаться, только на этапе выполнения, то статический SQL для нее непригоден. Описанная проблема решается с помощью динамического SQL, который свободен от указанных ограничений.

В реляционных СУБД компании IBM динамический SQL использовался изначально. Он много лет применялся также в СУБД для мини-компьютеров и UNIX. Однако динамический SQL не упоминается в стандарте ANSI/ISO SQL1, в нем описывается только статический SQL. По иронии судьбы, отсутствие динамического SQL в стандарте SQL1 объяснялось тем, что стандарт должен позволять создавать клиентские программы для работы с базами данных, переносимые между

СУБД различных типов. На самом же деле переносимые клиентские программы следует создавать именно с использованием динамического SQL.

В такой ситуации стандартом де-факто стал динамический SQL, применявшийся в DB2. Динамический SQL, поддерживаемый другими СУБД компании IBM того времени (SQL/DS и OS/2 Extended Edition), был почти идентичен тому, который использовался в DB2. Большинство других реляционных СУБД также последовало стандарту DB2. В 1992 году динамический SQL получил официальную поддержку со стороны стандарта SQL2, разработчики которого, в основном, придерживались пути, по которому шла компания IBM. На нижнем уровне (Entry Level) совместимости со стандартом SQL поддержка динамического SQL не требуется, но она обязательна для тех СУБД, которые претендуют на совместимость с SQL на уровне Intermediate или Full.

Концепции динамического SQL

Концепция, лежащая в основе динамического SQL, проста: встроенная инструкция SQL не записывается в исходный текст программы. Вместо этого программа формирует текст инструкции во время выполнения в одной из своих областей данных, а затем передает сформированную инструкцию в СУБД для динамического выполнения. Хотя детали реализации являются довольно сложными, весь динамический SQL построен на этом простом принципе, о котором не следует забывать.

Чтобы понять, как работает динамический SQL, и сравнить его со статическим SQL, полезно еще раз рассмотреть процесс выполнения инструкции SQL в СУБД, впервые представленный на рис. 17.1 и повторенный на рис. 18.1. Вспомним (см. главу 17, "Встроенный SQL"), что для статической инструкции SQL первые четыре действия выполняются на этапе компиляции. В процессе компиляции программы утилита `BIND` (или ее эквивалент) сохраняет в базе данных план выполнения инструкции, а во время выполнения программы СУБД просто исполняет план для данной инструкции.

В случае динамического SQL ситуация совершенно иная. Заранее неизвестно, какие инструкции SQL необходимо будет выполнять после запуска программы, поэтому СУБД не может приготовиться к их выполнению. Во время выполнения программы СУБД получает текст инструкции (называемый *строкой инструкции* или *строкой SQL*), которая должна быть динамически выполнена, и проходит через все пять этапов, изображенных на рис. 18.1.

Как и следует ожидать, динамический SQL менее эффективен, чем статический. По этой причине всегда, когда это возможно, используется статический SQL, и большинству прикладных программистов раньше никогда не приходилось иметь дело с динамическим SQL. Однако в связи с тем, что для работы с базами данных в настоящее время широко применяется архитектура "клиент/сервер", роль динамического SQL стала более важной. Пользователи персональных компьютеров все чаще работают с базами данных в таких приложениях, как электронные таблицы, текстовые процессоры и графические редакторы, а клиентские программы доступа к данным — это одна из наиболее быстро растущих областей баз данных. Во всех этих приложениях необходим динамический SQL.

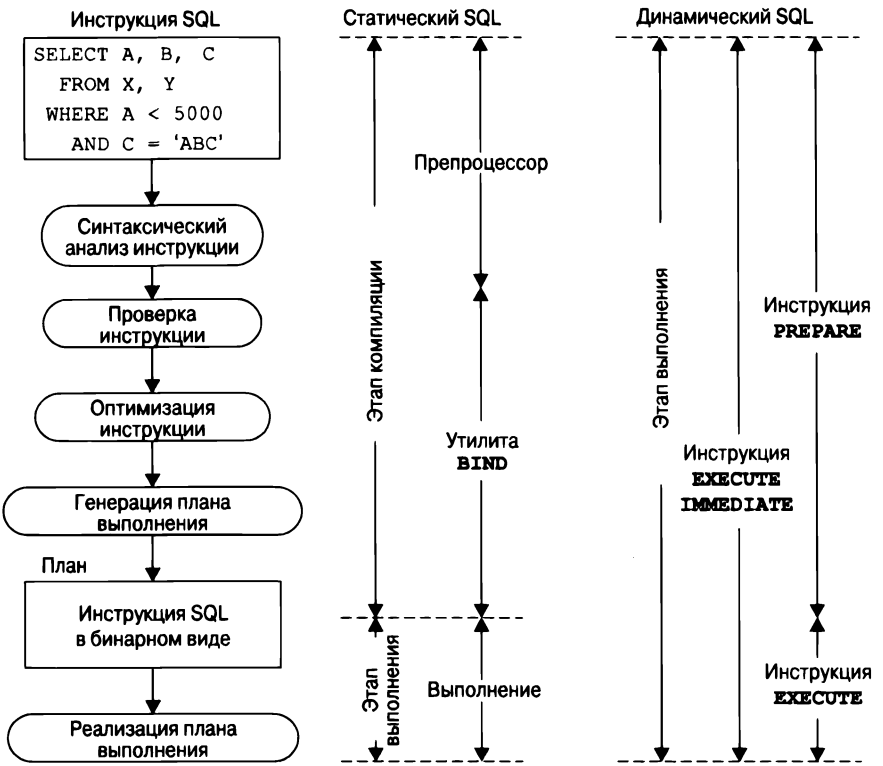


Рис. 18.1. Процесс выполнения инструкции SQL в СУБД

Важность динамического SQL еще больше возросла с развитием трехуровневых архитектур на базе Интернета, в которых логика приложения выполняется в одной системе (промежуточный уровень, зачастую включающий несколько серверов приложений), а логика базы данных — в другой (информационный, или серверный, уровень). В большинстве таких систем программная логика по своей природе динамична. Она должна адаптироваться к меняющимся условиям бизнеса, к появлению новых бизнес-правил. Регулярно изменяющаяся программная среда плохо сочетается со статическим SQL, в котором между программой и содержимым базы данных существует жесткая связь. Как следствие, в трехуровневых архитектурах для связи промежуточного уровня с базами данных серверного уровня применяются вызовы SQL API (которые описываются в главе 19, "SQL API"). Эти API для доступа к базам данных заимствуют многие концепции динамического SQL (например, разделение шагов `PREPARE` и `EXECUTE` и возможность вызова `EXECUTE IMMEDIATE`). Поэтому важно четко понимать принципы работы динамического SQL, чтобы представлять, что происходит за кулисами SQL API. В приложениях, производительность которых критична, такое понимание обеспечивает верный выбор дизайна приложения, обеспечивающего высокую производительность и низкое время отклика.

Динамическое выполнение инструкций (EXECUTE IMMEDIATE)

Наиболее простой формой динамического SQL является инструкция EXECUTE IMMEDIATE, синтаксическая диаграмма которой изображена на рис. 18.2. Эта инструкция передает в СУБД динамически сформированную строку инструкции SQL и дает команду немедленно ее выполнить. Схема применения инструкции EXECUTE IMMEDIATE выглядит так.

1. В одной из своих областей данных программа формирует инструкцию SQL в виде текстовой строки. (Вспомним, что для передачи информации в СУБД и получения от нее данных используются переменные языка программирования, называющиеся *базовыми переменными*.) Это может быть практически любая инструкция SQL, не извлекающая данные.
2. Программа передает сформированную инструкцию SQL в СУБД с помощью инструкции EXECUTE IMMEDIATE.
3. СУБД выполняет инструкцию и присваивает код завершения переменным SQLCODE/SQLSTATE точно так же, как при встраивании инструкции в программу с помощью статического SQL.

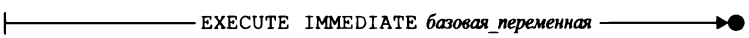


Рис. 18.2. Синтаксическая диаграмма инструкции EXECUTE IMMEDIATE

На рис. 18.3 приведена простая программа на языке С, осуществляющая эти действия. Программа запрашивает у пользователя имя таблицы и условие отбора, а затем на основе полученных ответов формирует текст инструкции DELETE. Для выполнения этой инструкции в программе применяется инструкция EXECUTE IMMEDIATE. Использовать статическую встроенную инструкцию DELETE нельзя, поскольку ни имя таблицы, ни условие отбора не известны до тех пор, пока пользователь не введет их во время выполнения программы.

Если запустить программу, текст которой приведен на рис. 18.3, и ввести информацию

```
Введите имя таблицы:        staff
Введите условие отбора:     quota < 20000
Удаление из таблицы staff успешное.
```

то программа передаст в СУБД такую строку.

```
delete from staff
 where quota < 20000
```

Если же ввести информацию

```
Введите имя таблицы:        orders
Введите условие отбора:     cust = 2105
Удаление из таблицы orders успешное.
```

то программа передаст такую строку.

```
delete from orders
 where cust = 2105
```

```
main()
{
    /* Программа удаляет строки из указанной
       пользователем таблицы в соответствии
       с пользовательскими условиями отбора. */
    exec sql include sqlca;
    exec sql begin declare section;
        char stmtbuf[301]; /* Выполняемый SQL-текст */
    exec sql end declare section;
    char tblname[101];      /* Имя таблицы
                               (вводит пользователь) */
    char search_cond[101]; /* Условие отбора
                               (вводит пользователь) */

    /* Построение инструкции DELETE в stmtbuf       */
    strcpy(stmtbuf, "delete from");

    /* Приглашение пользователю ввести имя таблицы
       и добавление его к инструкции DELETE         */
    printf("Введите имя таблицы: ");
    gets(tblname);
    strcat(stmtbuf, tblname);

    /* Приглашение пользователю ввести условие отбора
       и добавление его к инструкции DELETE         */
    printf("Введите условие отбора:");
    gets(search_cond);
    if (strlen(search_cond) > 0)
    {
        strcat(stmtbuf, " where ");
        strcat(stmtbuf, search_cond);
    }

    /* Обращение к СУБД для выполнения инструкции   */
    exec sql execute immediate :stmtbuf;
    if (sqlca.sqlcode < 0)
        printf("Ошибка SQL: %ld\n", sqlca.sqlcode);
    else
        printf("Удаление из %s успешное.\n", tblname);

    exit();
}
```

Рис. 18.3. Использование инструкции EXECUTE IMMEDIATE

Таким образом, инструкция EXECUTE IMMEDIATE предоставляет программе большую гибкость в выборе вида инструкции DELETE.

В инструкции EXECUTE IMMEDIATE применяется только одна базовая переменная, содержащая всю строку инструкции SQL. Сама строка инструкции не может содержать ссылки на базовые переменные, но в этом и нет необходимости. Вместо того чтобы использовать статическую инструкцию SQL с базовой переменной

```
exec sql delete from orders
         where cust = :cust_num;
```

программа с динамическим SQL получает тот же результат, формируя в буфере, а затем выполняя *целую* инструкцию.

```
sprintf(buffer, "delete from orders where cust = %d", cust_num)
exec sql execute immediate :buffer;
```

Инструкция EXECUTE IMMEDIATE представляет собой самую простую форму динамического SQL, но в то же время она оказывается очень гибкой. С ее помощью можно динамически выполнять большинство инструкций DML, включая инструкции INSERT, DELETE, UPDATE, COMMIT и ROLLBACK. Используя инструкцию EXECUTE IMMEDIATE, можно также динамически выполнять большинство инструкций DDL, в том числе CREATE, DROP, GRANT и REVOKE.

Однако у инструкции EXECUTE IMMEDIATE есть один существенный недостаток. С ее помощью нельзя динамически выполнить инструкцию SELECT, так как в инструкции EXECUTE IMMEDIATE отсутствует механизм обработки результатов запроса. Точно так же как в статическом SQL для запросов на выборку требуются курсоры и специальные инструкции (DECLARE CURSOR, OPEN, FETCH и CLOSE), так и в динамическом SQL для выполнения динамических запросов на выборку используются курсоры и несколько специальных инструкций. Ниже в настоящей главе рассматриваются средства динамического SQL, обеспечивающие реализацию динамических запросов на выборку.

Для предосторожности следует заметить, что пользовательский ввод не следует размещать непосредственно в SQL-инструкции (как это было сделано в приведенном выше упрощенном примере) без первоначального анализа управляющих и завершающих символов. Поступая так, как было сделано выше, вы позволяете хакерам включить во вводимую строку символы, которые могут завершить создаваемую инструкцию и добавить к ней другую, которая обеспечит неавторизованный доступ к другим данным базы. Эта методика известна как *sql-инжекция*.

Динамическое выполнение в два этапа

Инструкция EXECUTE IMMEDIATE обеспечивает одноэтапное выполнение динамической инструкции. Как уже говорилось, для выполнения динамической инструкции СУБД проходит через все пять этапов процесса, изображенного на рис. 18.1. Если в программе содержится много динамических инструкций, то накладные расходы на осуществление такого процесса могут стать значительными; и если выполняемые инструкции похожи друг на друга, то такой расход машинного времени будет просто расточительством. На практике инструкция EXECUTE IMMEDIATE применяется для одноразового выполнения инструкции, повторять которую в дальнейшем не планируется.

Для исправления ситуации в динамическом SQL существует альтернативный, двухэтапный, метод выполнения запросов SQL. На практике этот двухшаговый подход, разделяющий подготовку инструкции и ее выполнение, используется для *всех* SQL-инструкций программы, которые выполняются более одного раза, и в особенности для тех, которые выполняются многократно — сотни или тысячи раз, в ответ на запрос пользователя. Вот вкратце суть этого подхода.

1. Программа создает в буфере строку инструкции SQL, как и в случае с инструкцией EXECUTE IMMEDIATE. Любая константа в тексте инструкции может быть заменена вопросительным знаком; это говорит о том, что значение константы будет предоставлено позднее. Вопросительный знак называется *маркером параметра*, хотя очень часто используется термин *заполнитель*.

2. Инструкция PREPARE дает команду СУБД произвести синтаксический анализ инструкции, проверить ее правильность, оптимизировать и создать план выполнения (это первый шаг взаимодействия с СУБД). СУБД присваивает некоторое значение переменным SQLCODE/SQLSTATE, чтобы сообщить о любых ошибках, обнаруженных в инструкции, и сохраняет план выполнения. Обратите внимание на то, что, когда СУБД выполняет инструкцию PREPARE, она *не* выполняет соответствующий план.
3. Если программе требуется выполнить подготовленную ранее инструкцию, она передает в СУБД инструкцию EXECUTE вместе со значениями всех маркеров параметров (это второй шаг взаимодействия с СУБД). СУБД подставляет значения параметров, исполняет ранее подготовленный план и присваивает код завершения переменным SQLCODE/SQLSTATE.
4. Программа может многократно выполнять инструкцию EXECUTE, всякий раз изменяя значения параметров. СУБД просто повторяет второй шаг, поскольку первый уже выполнен, и его результат — план выполнения — остается корректен.

На рис. 18.4 представлен текст программы на языке С, которая осуществляет последовательность приведенных выше действий. Эта программа общего назначения выполняет обновление таблиц. Она запрашивает у пользователя имена таблицы и двух столбцов, а затем формирует для данной таблицы инструкцию UPDATE, имеющую следующий вид.

```
update имя_таблицы
   set имя_второго_столбца = ?
 where имя_первого_столбца = ?
```

Таким образом, информация, вводимая пользователем, задает обновляемую таблицу, обновляемый столбец и используемое при обновлении условие отбора. Сравниваемое значение в условии отбора и новые значения данных задаются как параметры, которые будут переданы позже, при выполнении инструкции EXECUTE.

После формирования в буфере текста инструкции UPDATE программа с помощью инструкции PREPARE дает СУБД команду произвести компиляцию инструкции UPDATE. Затем программа входит в цикл, запрашивая у пользователя значения параметров для выполнения ряда последовательных обновлений таблицы. Приведенный ниже диалог пользователя с компьютером показывает, как программа, представленная на рис. 18.4, обновляет личные планы служащих.

```
Введите имя обновляемой таблицы:     staff
Введите имя столбца для поиска:      empl_num
Введите имя столбца для обновления:  quota

Введите значение поиска empl_num:  106
Введите новое значение quota:      150000.00
Еще (y/n)? y

Введите значение поиска empl_num:  102
Введите новое значение quota:      225000.00
Еще (y/n)? y

Введите значение поиска empl_num:  107
Введите новое значение quota:      215000.00
Еще (y/n)? n

Обновления завершены.
```

```
main()
{
    /* Это программа общего назначения для обновления таблиц.
       Она может использоваться для любого обновления
       числового столбца во всех строках, где второй числовой
       столбец имеет определенное значение. Например, ее
       можно использовать для обновления плана продаж
       служащих или изменения предела кредита клиентов.    */

    exec sql include sqlca;
    exec sql begin declare section;
        char stmtbuf[301];  /* Выполняемый текст SQL        */
        float search_value; /* Значение искомого параметра */
        float new_value;    /* Обновленное значение         */
    exec sql end declare section;
    char tblname[31];       /* Обновляемая таблица          */
    char searchcol[31];     /* Имя столбца поиска значения */
    char updatecol[31];     /* Имя обновляемого столбца     */
    char yes_no[31];        /* Ответ yes/no пользователя    */

    /* Приглашение пользователю ввести имена
       таблицы и столбцов                                   */
    printf("Введите имя обновляемой таблицы: ");
    gets(tblname);
    printf("Введите имя столбца для поиска: ");
    gets(searchcol);
    printf("Введите имя столбца для обновления: ");
    gets(updatecol);

    /* Создание инструкции SQL в буфере; запрос к СУБД
       на ее компиляцию                                     */
    sprintf(stmtbuf, "update %s set %s = ? where %s = ?",   ①
            tblname, searchcol, updatecol);                 ②
    exec sql prepare mystmt from :stmtbuf;
    if (sqlca.sqlcode) {
        printf("Ошибка PREPARE: %ld\n", sqlca.sqlcode);
        exit();
    }

    /* Цикл работы с пользователем                          */
    for ( ; ; )
    {
        printf("\nВведите значение поиска %s: ", searchcol);
        scanf("%f", &search_value);
        printf("Введите новое значение %s: ", updatecol);
        scanf("%f", &new_value);

        /* Запрос СУБД на выполнение инструкции UPDATE      */
        execute mystmt using :search_value, :new_value;     ③
        if (sqlca.sqlcode)
        {
            printf("Ошибка EXECUTE: %ld\n", sqlca.sqlcode);
            exit();
        }

        /* Следует ли выполнить другое обновление?          */
        printf("Еще (y/n)? ");
        gets(yes_no);                                       ④
        if (yes_no[0] == 'n')
            break;
    }
    printf("\nОбновления завершены.\n");
    exit();
}
```

Рис. 18.4. Применение инструкций PREPARE **и** EXECUTE

Данная программа является хорошей иллюстрацией ситуации, в которой необходимо двухэтапное динамическое выполнение инструкции. СУБД компилирует динамическую инструкцию UPDATE только один раз, а выполняет три раза — по одному разу для каждого набора значений параметров, вводимых пользователем. Если бы в программе использовалась инструкция EXECUTE IMMEDIATE, то динамическая инструкция UPDATE компилировалась и выполнялась бы три раза. Таким образом, двухэтапное динамическое выполнение с помощью инструкций PREPARE и EXECUTE помогает устранить некоторые недостатки динамического SQL, снижающие производительность СУБД. Как уже упоминалось, такой же подход применяется во всех программных интерфейсах SQL, описываемых в следующей главе.

Инструкция PREPARE

Инструкция PREPARE, синтаксическая диаграмма которой изображена на рис. 18.5, применяется только в динамическом SQL. Она принимает базовую переменную, содержащую строку инструкции SQL, и передает эту инструкцию в СУБД. СУБД компилирует текст инструкции и подготавливает ее к выполнению, создавая план выполнения. СУБД присваивает также соответствующие значения переменным SQLCODE/SQLSTATE, чтобы сообщить пользователю обо всех ошибках, обнаруженных в тексте инструкции. Как уже говорилось, вместо любой константы строка инструкции может содержать маркер параметра, обозначаемый вопросительным знаком. Этот маркер дает СУБД сигнал о том, что значение параметра будет передано позднее, при выполнении инструкции.

Рис. 18.5. Синтаксическая диаграмма инструкции PREPARE

В результате выполнения инструкции PREPARE подготавливаемой инструкции присваивается *имя*. Это имя является идентификатором SQL, подобно имени курсора. Имя инструкции указывается в последующих инструкциях EXECUTE. Длительность хранения подготовленной инструкции, а также ее имени различается в конкретных СУБД. Иногда СУБД сохраняет инструкцию и связанное с нею имя до конца текущей транзакции (т.е. до последующей инструкции COMMIT или ROLLBACK). Если вам потребуется выполнить ту же самую динамическую инструкцию позднее, во время другой транзакции, то придется подготавливать ее еще раз. В других СУБД это ограничение ослаблено, а подготовленная инструкция сохраняется в течение всего сеанса подключения к базе данных. В стандарте ANSI/ISO SQL явно указано, что доступность подготовленной инструкции за пределами текущей транзакции зависит от конкретной реализации СУБД.

Инструкция PREPARE может быть использована для подготовки практически любой исполняемой инструкции DML или DDL, включая инструкцию SELECT. Инструкции встроенного SQL, которые являются директивами препроцессора (например, инструкции WHENEVER или DECLARE CURSOR), конечно, не могут быть подготовлены, так как это не исполняемые инструкции.

Инструкция EXECUTE

Инструкция EXECUTE, синтаксическая диаграмма которой изображена на рис. 18.6, применяется только в динамическом SQL. Она дает СУБД команду выполнить инструкцию, подготовленную ранее посредством инструкции PREPARE. С помощью инструкции EXECUTE можно выполнить любую подготовленную инструкцию, за одним исключением. Как и инструкция EXECUTE IMMEDIATE, инструкция EXECUTE не может использоваться для выполнения инструкции SELECT, поскольку не имеет механизма обработки результатов запроса.

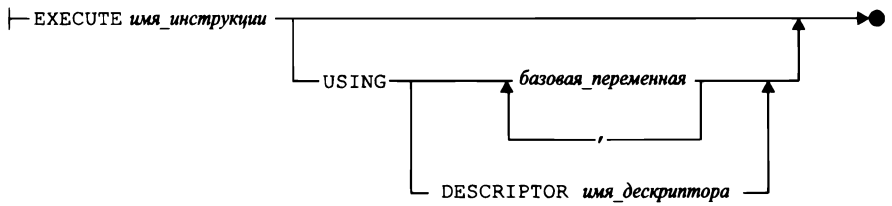


Рис. 18.6. Синтаксическая диаграмма инструкции EXECUTE

Если динамическая инструкция, которая должна быть выполнена, содержит один или несколько маркеров параметров, инструкция EXECUTE должна передать в СУБД значения всех этих параметров. Это можно сделать двумя различными способами, описанными в двух последующих разделах. В стандарте SQL поддерживаются оба способа.

Инструкция EXECUTE с базовыми переменными

Самый простой способ передать значения параметров — это задать список базовых переменных в предложении USING. Инструкция EXECUTE подставляет соответствующие значения базовых переменных вместо маркеров параметров в подготовленном тексте инструкции. Таким образом, эти базовые переменные являются для динамически выполняемой инструкции входными базовыми переменными. Такой метод применяется в программе, представленной на рис. 18.4. Он работает во всех популярных СУБД, которые поддерживают использование динамического SQL, и включен в стандарт ANSI/ISO SQL.

Число базовых переменных в предложении USING должно соответствовать числу маркеров параметров в динамической инструкции, а типы переменных должны быть совместимыми с типами данных соответствующих параметров. Каждая базовая переменная в списке может иметь сопутствующую переменную-индикатор. Если во время выполнения инструкции EXECUTE какая-нибудь переменная-индикатор содержит отрицательное значение, то соответствующему маркеру параметра присваивается значение NULL.

Инструкция EXECUTE, использующая SQLDA

Еще один способ передачи параметров заключается в применении специальной структуры, называемой *областью данных SQL* или, сокращенно, SQLDA (SQL Data Area). Эта область используется для передачи параметров, когда во время написания программы неизвестны ни число передаваемых параметров, ни их типы

данных. Например, вам необходимо так модифицировать программу обновления таблиц, представленную на рис. 18.4, чтобы пользователь получил возможность обновлять несколько столбцов. Можно легко модифицировать программу, чтобы она формировала инструкцию UPDATE с переменным числом обновляемых столбцов, но возникает проблема со списком базовых переменных в инструкции EXECUTE: он должен быть заменен списком переменной длины. Область SQLDA как раз обеспечивает способ задания такого списка.

На рис. 18.7 изображен формат области SQLDA, используемой во всех СУБД компании IBM, включая DB2, которая определяет стандарт "де-факто" динамического SQL. В большинстве других СУБД используется аналогичный формат или очень близкий к нему. В стандарте ANSI/ISO SQL описана похожая структура, которая называется *область дескриптора SQL*. В ней хранится та же информация, и обе структуры играют одинаковую роль в динамической обработке инструкций. Однако детали использования — размещение информации в области дескриптора, выборка информации, связь с параметрами инструкции SQL и т.п. — существенно различны. На практике предпочтение отдается структуре SQLDA из DB2, поскольку она появилась задолго до стандарта SQL и поддерживается большинством ведущих СУБД.

```
struct sqlda
{
    unsigned char sqldaid[8];
    long sqldabc;
    short sqln;
    short sqld;
    struct sqlvar
    {
        short sqltype;
        short sqllen;
        unsigned char *sqldata;
        short *sqlind;
        struct sqlname
        {
            short length;
            unsigned char data[30];
        } sqlname;
    } sqlvar[1];
};
```

Рис. 18.7. SQLDA **в СУБД компании IBM**

Область SQLDA представляет собой структуру данных переменной длины, состоящую из двух отдельных частей.

- **Постоянная часть** располагается в начале области. Ее поля идентифицируют структуру как область SQLDA и указывают размер данной конкретной области.
- **Переменная часть** представляет собой массив из одной или нескольких структур SQLVAR. Когда область SQLDA используется для передачи параметров в инструкцию EXECUTE, каждому параметру соответствует отдельная структура SQLVAR.

Поля структуры SQLVAR описывают данные, переданные инструкции EXECUTE в качестве значений параметров.

- Поле SQLTYPE содержит целочисленный *код типа данных*, который определяет тип передаваемого параметра. Например, в DB2 код 500 соответствует двухбайтовому целому числу, код 496 — четырехбайтовому целому числу, а код 448 — символьной строке переменной длины.
- Поле SQLLEN задает *длину* передаваемых данных. Оно будет содержать значение 2 для двухбайтового целого числа и 4 для четырехбайтового целого числа. Когда в качестве параметра передается символьная строка, поле SQLLEN содержит число символов в строке.
- Поле SQLDATA является указателем на *область данных* внутри программы, которая содержит значение параметра. СУБД использует этот указатель, чтобы найти требуемые данные при выполнении динамической инструкции SQL. Поля SQLTYPE и SQLLEN информируют СУБД о типе этих данных и их длине.
- Поле SQLIND является указателем на двухбайтовое целое число, которое используется в качестве переменной-индикатора для данного параметра. СУБД проверяет переменную-индикатор, чтобы определить, не передается ли значение NULL. Если переменная-индикатор для параметра отсутствует, то в поле SQLIND должно быть указано значение нуль.

Другие поля в структуре SQLVAR и SQLDA для передачи параметров посредством инструкции EXECUTE не используются. Они служат для извлечения информации из базы данных; этот процесс рассматривается позже в настоящей главе.

На рис. 18.8 изображена программа с динамическим SQL, в которой входные параметры задаются с помощью области SQLDA. Программа обновляет таблицу SALESREPS, позволяя пользователю указать, какие столбцы будут обновляться. Затем программа выполняет цикл, в котором запрашивает у пользователя идентификатор служащего и новые значения для обновляемых столбцов. Если в ответ на просьбу ввести новое значение пользователь вводит звездочку (*), программа присваивает соответствующему столбцу значение NULL.

```
main()
{
    /* Данная программа обновляет указанные пользователем
       столбцы таблицы SALESREPS. Вначале она запрашивает
       у пользователя имена столбцов, которые будут
       обновляться, а затем циклически запрашивает
       идентификатор служащего и новые значения выбранных
       столбцов.                                          */

    #define COLCNT 6 /* В таблице SALESREPS шесть столбцов */

    exec sql include sqlca;
    exec sql include sqlda;
    exec sql begin declare section;
        char stmtbuf[2001];  /* Текст инструкции SQL       */
```

Рис. 18.8. Применение инструкции EXECUTE и области SQLDA

```
exec sql end declare section;

struct {
    char prompt[31]; /* Приглашение для данного столбца*/
    char  name[31];  /* Имя данного столбца            */
    short typecode;  /* Код типа данных столбца        */
    short buflen;    /* Длина буфера столбца           */
    char  selected;  /* Флаг выбора (y/n)              */
} columns[] =
    {
        {"Name",       "NAME",       449, 16, 'n' },
        {"Office",     "REP_OFFICE", 497,  4, 'n' },
        {"Manager",    "MANAGER",    497,  4, 'n' },
        {"Hire Date",  "HIRE_DATE",  449, 12, 'n' },
        {"Quota",      "QUOTA",      481,  8, 'n' },
        {"Sales",      "SALES",      481,  8, 'n' }
    };

struct sqlda *parmda;      /* SQLDA для параметров       */
struct sqlvar*parmvar;     /* Структура SQLVAR для
                              текущего параметра         */
int           parmcnt;     /* Количество параметров      */
int           empl_num;    /* Введенный пользователем
                              идентификатор служащего    */
int           i;           /* Индекс массива columns[]   */
int           j;           /* Индекс массива sqlvar
                              в sqlda                    */
char          inbuf[101];  /* Ввод пользователя          */

/* Запрос у пользователя обновляемых столбцов            */
printf("*** Программа обновления данных"
       " о служащих ***\n\n");
parmcnt = 1;
for (i = 0; i < COLCNT; i++)
{
    /* Запрос о столбце */
    printf("Обновить столбец %s (y/n)? ",
           columns[i].name);
    gets(inbuf);

    if (inbuf[0] == 'y')
    {
        columns[i].selected = 'y';
        parmcnt++;
    }
}

/* Создать структуру SQLDA для передачи параметров       */
parmda = malloc(16 + (44 * parmcnt));  ←————————①
strcpy(parmda->sqldaid, "SQLDA  ");
parmda->sqldabc = (16 + (44 * parmcnt));
parmda->sqln    = parmcnt;

/* Начать формирование инструкции UPDATE                 */
strcpy(stmtbuf, "update salesreps set ");

/* Цикл по столбцам                                      */
for (i = 0, j = 0; i < COLCNT; i++)    ←————————②
{
```

Продолжение рис. 18.8

```
    /* Пропустить невыбранные столбцы                          */
    if (columns[i].selected == 'n')
        continue;

    /* Добавить в динамическую инструкцию
       UPDATE присваивание                                     */
    if (parmcnt > 0)
        strcat(stmtbuf, ", ");
    strcat(stmtbuf, columns[i].name);
    strcat(stmtbuf, " = ?");

    /* Выделить память для данных и индикатора
       и заполнить структуру SQLVAR информацией
       для данного столбца                                     */
    parmvar          = parmda->sqlvar + j;
    parmvar->sqltype = columns[i].typecode;  <------------③
    parmvar->sqllen  = columns[i].buflen;    <------------④
    parmvar->sqldata = malloc(columns[i].buflen); <--------⑤
    parmvar->sqlind  = malloc(2);  <----------------------⑥
    strcpy(parmvar->sqlname.data, columns[i].prompt);
    j++;
}

/* Заполнить последнюю структуру SQLVAR информацией
   о параметре, содержащемся в предложении WHERE            */
strcat(stmtbuf, " where empl_num = ?");
parmvar          = parmda + parmcnt;
parmvar->sqltype = 496;
parmvar->sqllen  = 4;
parmvar->sqldata = &empl_num;
parmvar->sqlind  = 0;  <------------------------------------⑦

/* Скомпилировать динамическую инструкцию UPDATE            */
exec sql prepare updatestmt from :stmtbuf;
if (sqlca.sqlcode < 0)
{
    printf("Ошибка PREPARE: %ld\n", sqlca.sqlcode);
    exit();
}

/* Цикл запроса параметров и выполнения обновлений          */
for ( ; ; )
{

    /* Запросить у пользователя идентификатор служащего,
       данные о котором будут обновлены                        */
    printf("\nВведите идентификатор служащего: ");
    scanf("%ld", &empl_num);
    if (empl_num == 0) break;

    /* Получить новые значения обновляемых столбцов            */
    for (j = 0; j < (parmcnt-1); j++)
    {
        parmvar = parmda + j;
        printf("Введите новое значение для %s: ",
               parmvar->sqlname.data);
        gets(inbuf);  <---------------------------------------⑧

        if (inbuf[0] == '*')
```

Продолжение рис. 18.8

```
            {
                /* Если пользователь вводит '*',
                   присвоить столбцу значение NULL          */
                *(parmvar->sqlind) = -1;
                continue;
            }
            else
            {
                /* В противном случае установить значение
                   переменной-индикатора                    */
                *(parmvar->sqlind) = 0;

                switch(parmvar->sqltype) {

                case 481:
                    /* Преобразовать введенные данные в
                       8-байтовое число с плавающей точкой */
                    sscanf(inbuf, "%lf", parmvar->sqldata);←⑧
                    break;

                case 449:
                    /* Передать введенные данные в виде
                       строки переменной длины              */
                    strcpy(parmvar->sqldata, inbuf);
                    parmvar->sqllen = strlen(inbuf);  ←———⑧
                    break;

                case 501:
                    /* Преобразовать введенные данные в
                       4-байтовое целое число               */
                    sscanf(inbuf, "%ld", parmvar->sqldata);←⑧
                    break;
                }
            }
        }

        /* Выполнить инструкцию                            */
        exec sql execute updatestmt using :parmda;  ←———⑨
        if (sqlca.sqlcode < 0)    {
            printf("Ошибка EXECUTE: %ld\n", sqlca.sqlcode);
            exit();
        }
    }

    /* Обновления завершены                                */
    exec sql execute immediate "commit work";
    if (sqlca.sqlcode)
        printf("Ошибка COMMIT: %ld\n", sqlca.sqlcode);
    else
        printf("\nВсе обновления завершены.\n");

    exit();
}
```

Окончание рис. 18.8

Так как при запуске программы пользователь может выбирать для обновления разные столбцы, то для передачи параметров при выполнении инструкции EXECUTE программа должна использовать область SQLDA. Данная программа ил-

люстрирует общую методику применения области SQLDA (кружки с цифрами на рис. 18.8 соответствуют пунктам данной методики).

1. Программа выделяет память для области SQLDA, достаточно большую для того, чтобы вместить все структуры SQLVAR с описанием передаваемых параметров. Она заполняет поле SQLN, указывающее, сколько структур SQLVAR может быть выделено.
2. Для каждого передаваемого параметра программа заполняет соответствующей информацией одну структуру SQLVAR.
3. Программа определяет, каков тип данных параметра, и помещает в поле SQLTYPE соответствующий код типа данных.
4. Программа определяет размер параметра и помещает его в поле SQLLEN.
5. Программа выделяет память для значения параметра и заносит адрес выделенной области памяти в поле SQLDATA.
6. Программа выделяет память для переменной-индикатора, сопровождающей данный параметр, и заносит адрес этой переменной в поле SQLIND.
7. Программа заполняет поле SQLD, указывающее, сколько параметров будет передано. Это поле сообщает СУБД, сколько структур SQLVAR в области SQLDA содержат корректные данные.
8. Программа запрашивает у пользователя значения данных и помещает их в области данных, выделенные при выполнении пп. 5 и 6.
9. Программа выполняет инструкцию EXECUTE с предложением USING, чтобы передать параметры посредством области SQLDA.

Обратите внимание на то, что данная программа копирует строку приглашения для каждого параметра в структуру SQLNAME. Это делается исключительно для удобства; когда область SQLDA применяется для передачи параметров, СУБД игнорирует структуру SQLNAME.

Вот пример диалога между пользователем и программой, исходный текст которой приведен на рис. 18.8.

```
*** Программа обновления данных о служащих ***

Обновить столбец NAME (y/n)? y
Обновить столбец REP_OFFICE (y/n)? y
Обновить столбец MANAGER (y/n)? n
Обновить столбец HIRE_DATE (y/n)? n
Обновить столбец QUOTA (y/n)? y
Обновить столбец SALES (y/n)? n

Введите идентификатор служащего: 106
Введите новое значение для NAME: Sue Jackson
Введите новое значение для REP_OFFICE: 22
Введите новое значение для QUOTA: 175000.00

Введите идентификатор служащего: 104
Введите новое значение для NAME: Joe Smith
Введите новое значение для REP_OFFICE: *
```

```
Введите новое значение для QUOTA: 275000.00

Введите идентификатор служащего: 0

Все обновления завершены.
```

На основе первоначальных ответов пользователя программа формирует динамическую инструкцию UPDATE и передает ее инструкции PREPARE.

```
update salesreps
   set name = ?, office = ?, quota = ?
 where empl_num = ?
```

В инструкции заданы четыре параметра, и программа выделяет область SQLDA, достаточно большую для того, чтобы разместить в ней четыре структуры SQLVAR. Когда пользователь вводит первый набор значений параметров, динамическая инструкция UPDATE принимает такой вид.

```
update salesreps
   set name = 'Sue Jackson', office = 22, quota = 175000.00
 where empl_num = 106
```

После ввода второго набора значений параметров она становится такой.

```
update salesreps
   set name = 'Joe Smith', office = NULL, quota = 275000.00
 where empl_num = 104
```

Приведенная программа кажется достаточно сложной, но по сравнению с реальными утилитами обновления баз данных она достаточно проста. В ней применяются все средства динамического SQL, необходимые для динамического выполнения инструкций с переменным числом параметров.

Динамические запросы

Рассмотренные до сих пор инструкции EXECUTE IMMEDIATE, PREPARE и EXECUTE обеспечивают динамическое выполнение большинства инструкций SQL. Но они не поддерживают динамические запросы на выборку, поскольку у них отсутствует механизм получения результатов запроса. Для выполнения динамических запросов инструкции PREPARE и EXECUTE динамического SQL *комбинируются* с расширенными инструкциями статического SQL, обеспечивающими обработку результатов запроса, а также вводится новая инструкция. Вот последовательность действий, осуществляемых программой при выполнении динамического запроса на выборку.

1. Динамическая версия инструкции DECLARE CURSOR объявляет курсор для запроса. В отличие от статической инструкции DECLARE CURSOR, которая содержит внутри себя саму инструкцию SELECT, динамическая форма инструкции DECLARE CURSOR содержит только имя динамической инструкции SELECT.

2. Программа формирует инструкцию SELECT в буфере аналогично тому, как она формирует динамическую инструкцию UPDATE или DELETE. Инструкция SELECT, как и другие динамические инструкции SQL, может содержать маркеры параметров.
3. С помощью инструкции PREPARE программа передает строку инструкции СУБД, которая проводит синтаксический анализ инструкции, проверяет ее правильность, оптимизирует инструкцию и создает план выполнения. Это аналогично выполнению инструкции PREPARE для других динамических инструкций SQL.
4. Посредством инструкции DESCRIBE программа запрашивает у СУБД описание таблицы результатов запроса, которая будет создана при выполнении запроса. СУБД возвращает описание каждого столбца таблицы в область данных SQL (SQLDA), указанную программой, информируя программу о том, сколько столбцов имеет таблица результатов запроса, каковы имя каждого столбца, тип и длина содержащихся в нем данных. Инструкция DESCRIBE используется исключительно для формирования динамических запросов на выборку.
5. Программа использует описания столбцов, находящиеся в области SQLDA, чтобы выделить блоки памяти для приема столбцов таблицы результатов запроса. Программа может также выделить место для переменных-индикаторов, сопровождающих столбцы. Чтобы сообщить СУБД, куда следует возвращать результаты запросов, программа помещает адреса этих блоков памяти и адреса переменных-индикаторов в SQLDA.
6. Динамическая версия инструкции OPEN дает СУБД команду начать выполнение запроса и передает ей значения параметров, заданные в динамической инструкции SELECT. Инструкция OPEN устанавливает курсор на позиции, предшествующей первой строке таблицы результатов запроса.
7. Динамическая версия инструкции FETCH перемещает курсор на первую строку таблицы результатов запроса, извлекает данные и записывает их в соответствующие области данных и в переменные-индикаторы. В отличие от статической инструкции FETCH, которая содержит список базовых переменных, принимающих данные, динамическая инструкция FETCH использует область SQLDA, чтобы сообщить СУБД, куда возвращать данные. Последующие инструкции FETCH продвигаются по таблице результатов запроса от строки к строке, перемещая курсор на следующую строку и извлекая из нее данные в соответствующие области данных программы.
8. Инструкция CLOSE прекращает доступ к таблице результатов запроса и ликвидирует связь между ею и курсором. Эта инструкция идентична инструкции CLOSE в статическом SQL; никакие расширения инструкции для обеспечения возможности реализации динамических запросов не требуются.

Программирование динамического запроса — это более трудоемкая задача по сравнению с формированием любой другой встроенной инструкции SQL. Однако этот процесс, как правило, является скорее утомительным, чем сложным. На

рис. 18.9 изображена небольшая программа формирования запросов к базе данных, которая с помощью динамического SQL извлекает данные и отображает на экране выбранные столбцы из заданной пользователем таблицы. Кружки с цифрами на рисунке соответствуют восьми пунктам, перечисленным выше.

```
main()
{
    /* Это простая программа общего назначения. Она
       запрашивает у пользователя имя таблицы, а затем -
       какие столбцы таблицы должны быть добавлены в
       запрос. После завершения диалога с пользователем
       программа выполняет запрос и выводит его результаты
       на экран.                                          */
    exec sql include sqlca;
    exec sql include sqlda;
    exec sql begin declare section;
        char stmtbuf[2001]; /* Текст SQL-запроса          */
        char querytbl[32];  /* Пользовательская таблица   */
        char querycol[32];  /* Пользовательский столбец   */
    exec sql end declare section;

    /* Курсор для получения имен столбцов
       из системного каталога                             */
    exec sql declare tblcurs cursor for
              select colname
                from system.syscolumns
               where tblname = :querytbl and owner = user;
    exec sql declare qrycurs cursor for querystmt; <------- ①

    /* Структуры данных программы */
    int    colcount = 0;    /* Количество столбцов        */
    struct sqlda *qry_da;   /* Область SQLDA запроса      */
    struct sqlvar *qry_var; /* SQLVAR текущего столбца    */
    int    i;               /* Индекс SQLVAR в SQLDA      */
    char   inbuf[101];      /* Ввод пользователя          */

    /* Приглашение пользователю указать таблицу           */
    printf("*** Программа с мини-запросом ***\n\n");
    printf("Введите имя таблицы: ");
    gets(querytbl);

    /* Создание инструкции SELECT в буфере                */
    strcpy(stmtbuf, "select ");  <------------------------- ②

    /* Настройка обработки ошибок                         */
    exec sql whenever sqlerror goto handle_error;
    exec sql whenever not found goto no_more_columns;

    /* Запрос системного каталога об именах столбцов      */
    exec sql open tblcurs;
    for ( ; ; )
    {
        /* Получение имени столбца и запрос пользователя  */
        exec sql fetch tblcurs into :querycol;
        printf("Включить столбец %s (y/n)? ", querycol);
        gets(inbuf);
        if (inbuf[0] == 'y')
```

Рис. 18.9. Выборка данных с помощью динамического SQL

```
{
            /* Пользователь просит включить данный столбец;
               добавляем его к списку                          */
            if (colcount++ > 0)
                strcat(stmtbuf, ", ");
            strcat(stmtbuf, querycol);    <------------------- ②
        }
    }

no_more_columns:
    exec sql close tblcurs;

    /* Завершаем инструкцию SELECT предложением FROM          */
    strcat(stmtbuf, "from ");
    strcat(stmtbuf, querytbl);

    /* Выделение SQLDA для динамического запроса */
    query_da = (SQLDA *)malloc(sizeof(SQLDA) +
                               colcount * sizeof(SQLVAR));
    query_da->sqln = colcount;

    /* Подготавливаем запрос
       и передаем его СУБД для описания                        */
    exec sql prepare querystmt from :stmtbuf;   <------------ ③
    exec sql describe querystmt into qry_da;    <------------ ④

    /* Цикл по всем SQLVAR,
       с выделением памяти для столбцов                        */
    for (i = 0; i < colcount; i++)
    {
        qry_var = qry_da->sqlvar + i;
        qry_var->sqldat = malloc(qry_var->sqllen);   <------- ⑤
        qry_var->sqlind = malloc(sizeof(short));
    }

    /* SQLDA создана; запрос и получение результатов          */
    exec sql open qrycurs;  <------------------------------- ⑥
    exec sql whenever not found goto no_more_data;
    for ( ; ; )
    {
        /* Получение строки результатов в наши буферы         */
        exec sql fetch sqlcurs using descriptor qry_da; <---- ⑦
        printf("\n");
        /* Цикл вывода данных для каждого столбца строки      */
        for (i = 0; i < colcount; i++)
        {
            /* Поиск SQLVAR для данного столбца;
               вывод метки столбца                             */
            qry_var = qry_da->sqlvar + i;
            printf(" Столбец # %d (%s): ",
                   i+1, qry_var->sqlname);
            /* Проверка переменной-индикатора                  */
            if (*(qry_var -> sqlind) != 0)
            {
                puts("равен NULL!\n");
                continue;
            }
```

Продолжение рис. 18.9

```
                /* Получение данных и обработка типа          */
                switch (qry_var -> sqltype) {
                case 448:
                case 449:
                    /* VARCHAR -- просто выводим              */
                    puts(qry_var -> sqldata);
                    break;
                case 496:
                case 497:
                    /* Четырехбайтовое целое -
                       преобразуем и выводим                  */
                    printf("%ld", *((int *) (qry_var->sqldata)));
                    break;
                case 500:
                case 501:
                    /* Двухбайтовое целое -
                       преобразуем и выводим                  */
                    printf("%d", *((short *)(qry_var->sqldata)));
                    break;
                case 480:
                case 481:
                    /* Данные с плавающей точкой -
                       преобразуем и выводим                  */
                    printf("%lf",
                           *((double *)(qry_var->sqldat)));
                    break;
                }
            }
        }
    no_more_data:
        printf("\nКонец данных.\n");

        /* Освобождение выделенной памяти                    */
        for (i = 0; i < colcount; i++)
        {
            qry_var = qry_da->sqlvar + i;
            free(qry_var->sqldata);
            free(qry_var->sqlind);
        }
        free(qry_da);
        close qrycurs;   <------------------------------------ ⑧
        exit();
    }
```

Окончание рис. 18.9

Вначале программа, представленная на рис. 18.9, запрашивает у пользователя имя таблицы, а затем обращается к системному каталогу, чтобы узнать имена столбцов этой таблицы. Она просит пользователя выбрать столбцы, которые будут извлекаться, и на основе его ответов формирует динамическую инструкцию SELECT. Последовательное, шаг за шагом, формирование списка выбираемых столбцов характерно для клиентских программ, использующих динамический SQL. В реальных приложениях этот список может включать выражения или статистические функции, поэтому в программе может присутствовать дополнительный код, формирующий предложения GROUP BY, HAVING или ORDER BY. Заметьте, что создаваемая инструкция SELECT идентична инструкции SELECT, которая применялась бы для выполнения данного запроса в интерактивном режиме.

Обработка инструкций PREPARE и DESCRIBE, выполняемая в этой программе, а также способ выделения памяти под извлекаемые данные являются характерными для программ формирования динамических запросов. Обратите внимание на то, как программа использует описания столбцов, помещенные в массив структур SQLVAR, чтобы выделить для каждого столбца блок памяти соответствующего размера. Для каждого столбца программа также выделяет в памяти место для переменной-индикатора. Адрес блока памяти для содержимого столбца и адрес переменной-индикатора программа помещает в структуру SQLVAR.

Как видно из данной программы, инструкции OPEN, FETCH и CLOSE играют в динамических запросах ту же роль, что и в статических. Обратите внимание: в инструкции FETCH вместо списка базовых переменных задана область SQLDA. Так как программа предварительно заполнила поля SQLDATA и SQLIND в массиве SQLVAR, СУБД знает, куда помещать каждый извлекаемый столбец данных.

Из этого примера следует, что программирование динамического запроса связано большей частью с созданием области SQLDA и выделением памяти для нее и извлекаемых данных. Кроме того, в программе необходимо отслеживать различные типы данных, которые могут быть возвращены запросом, и правильно их обрабатывать, учитывая возможность возвращения значений NULL. Эти характеристики программы, изображенной на рис. 18.9, типичны для промышленных приложений, использующих динамические запросы. Несмотря на кажущуюся сложность, программировать динамические запросы на языках C, C++, Pascal, PL/1 или Java не очень трудно. Использовать для программирования динамических запросов такие языки, как COBOL и FORTRAN, нельзя, так как в них отсутствует возможность динамического выделения памяти и использования структур переменной длины.

В последующих разделах рассматриваются инструкция DESCRIBE и динамические версии инструкций DECLARE CURSOR, OPEN, FETCH и CLOSE.

Инструкция DESCRIBE

Инструкция DESCRIBE, синтаксическая диаграмма которой изображена на рис. 18.10, применяется только в динамических запросах и служит для получения от СУБД описания динамического запроса. Эта инструкция выполняется после компиляции запроса инструкцией PREPARE, но до его выполнения инструкцией OPEN. Запрос, который должен быть описан, идентифицируется по имени инструкции. СУБД возвращает описание запроса в область SQLDA, указанную программой.

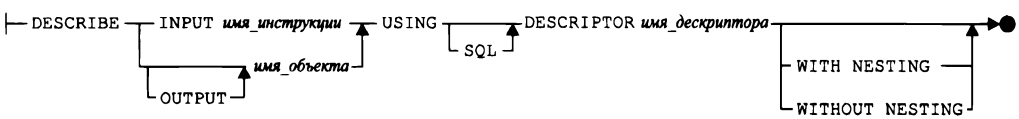


Рис. 18.10. Синтаксическая диаграмма инструкции DESCRIBE

Как уже говорилось в настоящей главе и было показано на рис. 18.7, область SQLDA — это структура переменной длины, содержащая массив, который состоит из одной или нескольких структур SQLVAR. Прежде чем использовать область

SQLDA в инструкции DESCRIBE, программа должна заполнить поле SQLN в ее заголовке, информирующее СУБД о размере массива SQLVAR в этой конкретной области. В качестве первого этапа обработки инструкции DESCRIBE СУБД заполняет поле SQLD в заголовке области SQLDA значением, соответствующим числу столбцов таблицы результатов запроса. Если размер массива SQLVAR (указанный в поле SQLN) слишком мал, чтобы вместить все описания столбцов, СУБД не заполняет оставшуюся часть области SQLDA. В противном случае СУБД заполняет данными по одной структуре SQLVAR для каждого столбца таблицы результатов запроса в порядке их следования слева направо. Каждая структура SQLVAR состоит из следующих полей, описывающих соответствующий столбец.

- Поле SQLNAME содержит имя столбца и, в свою очередь, состоит из двух полей: в поле DATA содержится собственно имя, а в поле LENGTH — длина имени. Если столбец является производным от выражения, то поле SQLNAME не используется.
- Поле SQLTYPE содержит целочисленный код типа данных столбца. Коды типов данных, используемые в различных СУБД, отличаются друг от друга. Как видно из табл. 18.1, коды типов данных в СУБД компании IBM указывают не только сам тип, но и то, допустимы ли в столбце значения NULL.
- Поле SQLLEN содержит длину данных столбца. Для типов данных переменной длины (таких, как VARCHAR) указывается максимально возможная длина данных в столбце; длина данных в любой строке этого столбца не должна превышать указанное значение. В DB2 (и многих других СУБД) значение, соответствующее типу DECIMAL, включает как точность десятичного числа (в старшем байте), так и его степень (в младшем байте).
- СУБД не заполняет поля SQLDATA и SQLIND. Прикладная программа записывает в эти поля адреса буфера данных и переменной-индикатора столбца перед тем, как использовать область SQLDA в инструкции FETCH.

Таблица 18.1. Коды типов данных SQLDA в DB2

Тип данных	Значение NULL разрешено	Значение NULL запрещено
CHAR	452	453
VARCHAR	448	449
LONG VARCHAR	456	457
SMALLINT	500	501
INTEGER	496	497
FLOAT	480	481
DECIMAL	484	485
DATE	384	385
TIME	388	389
TIMESTAMP	392	393
GRAPHIC	468	469
VARGRAPHIC	464	465

Сложность применения инструкции DESCRIBE заключается в том, что программа может не знать заранее, сколько столбцов будет содержать таблица результатов запроса, и, соответственно, не будет знать, насколько большой должна быть область SQLDA, чтобы в ней уместились все описания. Чтобы обеспечить достаточно большой размер области SQLDA, обычно пользуются одним из трех способов.

- При формировании списка извлекаемых столбцов программа может подсчитывать их количество. Затем программа может создать область SQLDA с таким числом структур SQLVAR, которое необходимо для получения описаний столбцов. Этот способ применяется в программе, представленной на рис. 18.9.
- Если программе неудобно подсчитывать количество извлекаемых элементов, она может выполнить инструкцию DESCRIBE, используя минимальную область SQLDA (с одной структурой SQLVAR). Когда инструкция DESCRIBE возвращает результат, поле SQLD содержит число столбцов, имеющихся в таблице результатов запроса. Теперь программа может создать область SQLDA необходимого размера и повторно выполнить инструкцию DESCRIBE, указывая новую область SQLDA. Нет никаких ограничений на число выполнений инструкции DESCRIBE для одной подготовленной инструкции.
- Программа может создать область SQLDA с массивом структур SQLVAR, достаточно большим для типичного запроса. В большинстве случаев инструкция DESCRIBE, использующая такую область, будет успешно выполняться. Если область SQLDA окажется слишком маленькой, то в поле SQLD будет содержаться информация о требуемом размере. Тогда программа может создать еще большую область SQLDA и повторно выполнить инструкцию DESCRIBE, используя новую область.

Инструкция DESCRIBE обычно применяется для выполнения динамических запросов, но с ее помощью можно получить описание любой предварительно подготовленной инструкции. Это полезно, например, в том случае, когда программе требуется выполнить неизвестную инструкцию SQL, введенную пользователем. Программа может выполнить инструкции PREPARE и DESCRIBE, а затем извлечь содержимое поля SQLD в области SQLDA. Если значение в этом поле равно нулю, то это означает, что инструкция не является запросом на выборку и ее можно выполнить с помощью инструкции EXECUTE. Если в поле SQLD содержится положительное число, то инструкция представляет собой запрос на выборку и для ее выполнения необходимо применить последовательность инструкций OPEN/FETCH/CLOSE.

Инструкция DECLARE CURSOR

Динамическая инструкция DECLARE CURSOR, синтаксическая диаграмма которой изображена на рис. 18.11, является разновидностью статической инструкции DECLARE CURSOR. Вспомним (см. главу 17, "Встроенный SQL"), что в статической инструкции DECLARE CURSOR запрос задается явно, в качестве одного из предложе-

ний инструкции. В динамической же инструкции DECLARE CURSOR запрос задается неявно в виде имени инструкции (связанной с запросом инструкцией PREPARE).

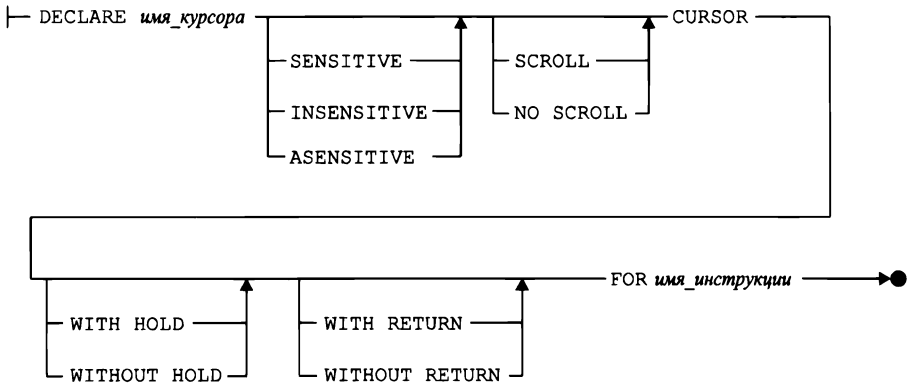


Рис. 18.11. Синтаксическая диаграмма динамической инструкции DECLARE CURSOR

Динамическая инструкция DECLARE CURSOR, как и ее статический аналог, является не исполняемой инструкцией, а директивой препроцессора SQL. В программе она должна стоять раньше любого обращения к объявляемому ею курсору. Имя курсора, объявленное этой инструкцией, используется последующими инструкциями OPEN, FETCH и CLOSE для обработки результатов динамического запроса.

Динамическая инструкция OPEN

Динамическая инструкция OPEN, синтаксическая диаграмма которой изображена на рис. 18.12, является разновидностью статической инструкции OPEN. Она дает СУБД команду начать выполнение запроса и устанавливает курсор в позицию, предшествующую первой строке таблицы результатов запроса. После выполнения инструкции OPEN курсор остается в открытом состоянии и может быть использован инструкцией FETCH.

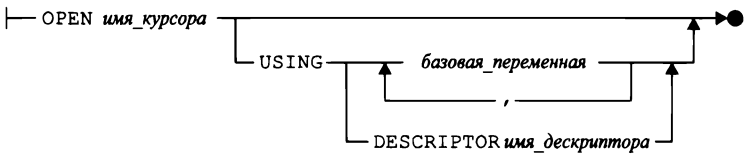


Рис. 18.12. Синтаксическая диаграмма инструкции OPEN

Роль инструкции OPEN для динамических запросов аналогична роли инструкции EXECUTE для выполнения других динамических инструкций SQL. Как инструкция EXECUTE, так и инструкция OPEN дает СУБД команду выполнить инструкцию, предварительно скомпилированную инструкцией PREPARE. Если текст динамического запроса содержит маркеры параметров, то инструкция OPEN, как и инструкция EXECUTE, должна передать в СУБД значения этих параметров. Для передачи параметров используется предложение USING, и оно имеет одинаковый формат как в инструкции EXECUTE, так и в инструкции OPEN.

Если число параметров динамического запроса известно заранее, программа может передать в СУБД значения параметров с помощью списка базовых переменных, содержащегося в предложении USING инструкции OPEN. Так же как и в инструкции EXECUTE, число базовых переменных должно соответствовать числу параметров, типы базовых переменных должны быть совместимыми с типами данных соответствующих параметров и, если необходимо, должны быть заданы переменные-индикаторы. На рис. 18.13 представлен фрагмент программы, в которой динамический запрос имеет параметры, значения которых указываются с помощью базовых переменных.

```
     .
     .
     .
/* Программа предварительно генерирует и подготавливает
   инструкцию SELECT наподобие

   SELECT A, B, C ...
     FROM SALESREPS
    WHERE SALES BETWEEN ? AND ?

   с двумя параметрами */

/* Приглашение пользователю ввести значения границ */
printf("Введите нижнюю границу диапазона: ");
scanf("%f", &low_end);
printf("Введите верхнюю границу диапазона: ");
scanf("%f", &high_end);

/* Открываем курсор для запуска запроса,
   передавая параметры */
exec sql open qrycursor using :low_end, :high_end;
     .
     .
     .
```

Рис. 18.13. Инструкция OPEN с базовыми переменными

Если до выполнения программы число параметров неизвестно, то программа должна передавать значения параметров с помощью области SQLDA. Эта методика передачи параметров была описана ранее в настоящей главе при рассмотрении инструкции EXECUTE. В инструкции OPEN применяется та же методика. На рис. 18.14 изображен фрагмент программы, в котором для передачи параметров используется область SQLDA.

Обратите особое внимание на то, что область SQLDA, используемая в инструкции OPEN, не имеет *ничего общего* с областью SQLDA, используемой в инструкциях DESCRIBE и FETCH.

- Область SQLDA в инструкции OPEN используется для передачи значений параметров динамического запроса *в* СУБД. Элементы массива SQLVAR соответствуют маркерам параметров в тексте динамической инструкции.
- Область SQLDA в инструкциях DESCRIBE и FETCH получает *от* СУБД описания столбцов таблицы результатов запроса и информирует СУБД о том, куда помещать извлекаемые результаты запроса. Элементы массива SQLVAR соответствуют *столбцам таблицы результатов запроса*, создаваемой динамическим запросом.

```
      .
      .
      .
/* Программа предварительно генерирует и подготавливает
   инструкцию SELECT наподобие

   SELECT A, B, C ...
     FROM SALESREPS
    WHERE EMPL_NUM IN (?, ?, ... ?)

   с переменным количеством параметров. Количество
   параметров хранится в переменной parmcnt */

SQLDA  *parmda;
SQLVAR *parmvar;
long    parm_value[101];

/* Выделение памяти для SQLDA для передачи
   значений параметров */
parmda = (SQLDA *)malloc(sizeof(SQLDA) +
                         parmcnt * sizeof(SQLVAR));
parmda->sqln = parmcnt;

/* Приглашение пользователю ввести значения параметров */
for (i = 0; i < parmcnt; i++)
{
    printf("Введите номер служащего: ");
    scanf("%ld", &(parm_value[i]));
    parmvar = parmda -> sqlvar + i;
    parmvar->sqltype = 496;
    parmvar->sqllen  = 4;
    parmvar->sqldata = &(parm_value[i]);
    parmvar->sqlind  = 0;
}

/* Откройте курсор для запуска запроса
   с передачей параметров */
exec sql open qrycursor using descriptor :parmda;
      .
      .
      .
```

Рис. 18.14. Инструкция OPEN, **использующая для передачи параметров** SQLDA

Динамическая инструкция FETCH

Динамическая инструкция FETCH, синтаксическая диаграмма которой изображена на рис. 18.15, является разновидностью статической инструкции FETCH. Она перемещает курсор на следующую доступную строку таблицы результатов запроса и извлекает значения ее столбцов в области данных, указанные программой. Вспомним (см. главу 17, "Встроенный SQL"), что в состав статической инструкции FETCH входит предложение INTO со списком базовых переменных, принимающих значения столбцов. В динамической инструкции FETCH базовые переменные заменяются областью SQLDA.

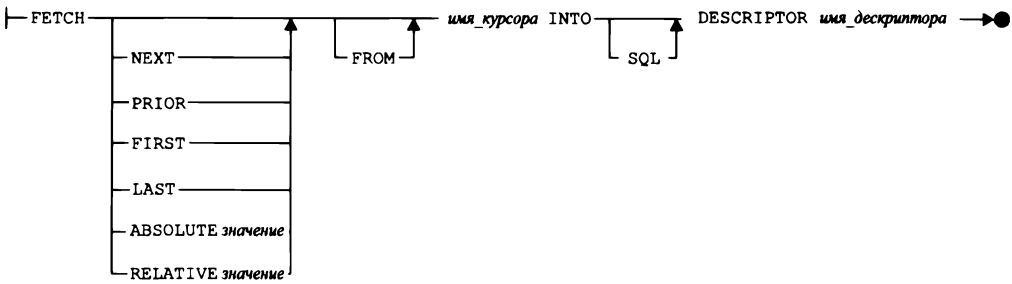


Рис. 18.15. Синтаксическая диаграмма динамической инструкции FETCH

Перед выполнением динамической инструкции FETCH прикладная программа должна выделить память для приема извлекаемых данных и для переменных-индикаторов каждого столбца. Прикладная программа должна также для каждого столбца заполнить в структуре SQLVAR поля SQLDATA, SQLIND и SQLLEN следующим образом.

- Поле SQLDATA должно указывать на область памяти, предназначенную для приема извлекаемых данных.
- Поле SQLLEN должно содержать размер области памяти, указанной в поле SQLDATA. Этот размер должен быть корректно задан, иначе СУБД будет копировать извлекаемые данные за пределы выделенной области.
- Поле SQLIND должно указывать на переменную-индикатор столбца (двухбайтовое целое число). Если для какого-либо столбца переменная-индикатор не используется, то поле SQLIND в соответствующей структуре SQLVAR должно содержать нуль.

Обычно прикладная программа выделяет память для области SQLDA, с помощью инструкции DESCRIBE получает описание таблицы результатов запроса, выделяет память для каждого столбца таблицы и устанавливает значения полей SQLDATA и SQLIND; все это она делает перед открытием курсора. Эта область SQLDA используется затем в инструкции FETCH. Необязательно использовать во всех инструкциях FETCH одну и ту же SQLDA, как необязательно, чтобы для каждой инструкции FETCH в области SQLDA указывались одни и те же области памяти. Прикладная программа может изменять указатели в полях SQLDATA и SQLIND (между двумя вызовами инструкции FETCH), извлекая две последовательные строки в разные области памяти.

Динамическая инструкция CLOSE

Функция, выполняемая динамической инструкцией CLOSE, и ее синтаксис идентичны функции и синтаксису статической инструкции CLOSE, изображенной на рис. 17.28. В обоих случаях инструкция CLOSE прекращает доступ к таблице результатов запроса. Когда программа закрывает курсор, созданный для динамического запроса, она, как правило, должна освобождать ресурсы, связанные с этим запросом. К таковым относятся следующие:

- область SQLDA, выделенная для динамического запроса и используемая в инструкциях DESCRIBE и FETCH;
- вторая область SQLDA (если таковая была выделена), используемая для передачи значений параметров в инструкции OPEN;
- области памяти, выделенные для приема столбцов таблицы результатов запроса, извлекаемых инструкцией FETCH;
- области памяти, выделенные для переменных-индикаторов, сопровождающих столбцы таблицы результатов запроса.

Эти ресурсы можно не освобождать, если программа заканчивает свою работу сразу же после выполнения инструкции CLOSE.

Диалекты динамического SQL

В разных СУБД используются различные диалекты динамического SQL. Эти различия более серьезны, чем в случае статического SQL, так как динамический SQL в большей степени зависит от особенностей конкретной СУБД, таких как используемые в ней типы данных, их форматы и т.п. Вот почему невозможно написать отдельную универсальную клиентскую программу для работы с базами данных, переносимую на различные СУБД. Такого рода программное обеспечение должно иметь для каждой поддерживаемой СУБД свой транслирующий слой, часто называемый *драйвером*, с помощью которого выполняется настройка под конкретную СУБД.

Раньше клиентское программное обеспечение поставлялось с отдельным драйвером для каждой из популярных СУБД. Появление протокола ODBC как единого программного интерфейса SQL упростило задачу, поскольку драйвер ODBC достаточно написать один раз для каждой СУБД, а клиентская программа может просто вызывать функции ODBC. Однако применение такого подхода на практике означало, что программа не могла использовать специфические особенности той или иной СУБД, что в конечном итоге приводило к снижению производительности программы. Поэтому современные клиентские приложения по-прежнему включают отдельные драйверы для популярных СУБД, а драйвер ODBC используется для обеспечения доступа к остальным СУБД.

Подробное описание вариантов динамического SQL для всех основных типов СУБД выходит за рамки данной книги. Однако будет поучительно рассмотреть в качестве примера диалект динамического SQL в Oracle.

Динамический SQL в Oracle*

СУБД Oracle появилась на рынке раньше, чем DB2, и ее динамический SQL базировался на прототипе System/R компании IBM. (В действительности СУБД Oracle была независимо разработана на основе общедоступных спецификаций System/R.) По этой причине динамический SQL в Oracle несколько отличается от стандарта IBM SQL. Хотя Oracle и DB2 в основном совместимы, у них имеются существенные отличия на уровне деталей. В эти отличия входит использование мар-

керов параметров, области SQLDA, формат этой области и поддержка преобразования типов данных. Различия между Oracle и DB2 сходны с теми, которые характерны по отношению к DB2 и для большинства других СУБД. Поэтому будет полезно кратко рассмотреть применяемый в Oracle динамический SQL и его отличия от динамического SQL в DB2.

Именованные параметры

Вспомним, что DB2 не допускает применения ссылок на базовые переменные в динамически подготавливаемых инструкциях. Вместо этого параметры в инструкциях идентифицируются знаками вопроса (маркерами параметров), а значения параметров задаются в инструкции EXECUTE или OPEN. Oracle же позволяет указывать параметры в динамически подготавливаемых инструкциях с использованием синтаксиса базовых переменных. Например, приведенная далее последовательность встроенных инструкций SQL в Oracle является допустимой.

```
exec sql begin declare section;
    char stmtbuf[1001];
     int employee_number;
exec sql end declare section;
    .
    .
    .
strcpy(stmtbuf, "delete from salesreps"
                " where empl_num = :rep_number;");
exec sql prepare delstmt from :stmtbuf;
exec sql execute delstmt using :employee_number;
```

Хотя rep_number выглядит как базовая переменная в динамической инструкции DELETE, фактически она является *именованным параметром*. Как видно из данного примера, именованные параметры аналогичны маркерам параметров в DB2. Значение параметра берется из "настоящей" базовой переменной в инструкции EXECUTE. Именованные параметры очень удобны, когда применяются динамические инструкции с переменным числом параметров.

Инструкция DESCRIBE

Инструкция DESCRIBE в Oracle, как и в DB2, применяется для описания таблицы результатов динамического запроса. Аналогично DB2, описание возвращается в область SQLDA. Инструкцию DESCRIBE в Oracle можно также использовать для описания именованных параметров в динамически подготовленной инструкции. Oracle тоже возвращает эти описания в область SQLDA.

Приведенная ниже инструкция DESCRIBE в Oracle запрашивает описание столбцов таблицы результатов предварительно подготовленной динамической инструкции.

```
exec sql describe select list for qrystmt into qry_sqlda;
```

Она соответствует инструкции DB2.

```
exec sql describe qrystmt into qry_sqlda;
```

Приведенная ниже инструкция DESCRIBE в Oracle запрашивает описание именованных параметров предварительно подготовленной динамической инструк-

ции. Подготовленная инструкция может быть как запросом на выборку, так и некоторой иной инструкцией SQL.

```
exec sql describe bind variables for thestmt into the_sqlda;
```

Эта инструкция эквивалента в DB2 не имеет. После нее программа, как правило, просматривает информацию в области SQLDA, записывает в нее указатели на значения параметров, которые она хочет передать в СУБД, и выполняет динамическую инструкцию, применяя инструкции EXECUTE или OPEN, использующие область SQLDA.

```
exec sql execute thestmt using descriptor the_sqlda;
exec sql open qrycursor using descriptor the_sqlda;
```

Информация, возвращаемая в Oracle инструкциями DESCRIBE обеих форм, является идентичной и рассматривается в следующем разделе.

Область SQLDA в Oracle

Область SQLDA в Oracle выполняет те же функции, что и в DB2, но ее формат, представленный на рис. 18.16, существенно отличается от формата, используемого в DB2. Два основных поля в заголовке области SQLDA из DB2 имеют аналоги в области SQLDA из Oracle.

- Поле N в Oracle задает размер массива, в котором размещаются описания столбцов. Оно соответствует полю SQLN в DB2.
- Поле F в Oracle указывает, сколько столбцов описано в настоящий момент в массивах области SQLDA. Оно соответствует полю SQLD в DB2.

```
struct sqlda
{
    long    N; /* Количество элементов
                  в массиве SQLDA                        */
    char  **V; /* Указатель на массив
                  указателей на области данных           */
    long   *L; /* Указатель на массив
                  длин буферов                           */
    short  *T; /* Указатель на массив
                  кодов типов данных                     */
    short **I; /* Указатель на массив
                  указателей на индикаторные переменные  */
    long    F; /* Количество активных
                  записей в массивах SQLDA               */
    char  **S; /* Указатель на массив
                  указателей на имена столбцов/параметров */
    short  *M; /* Указатель на массив
                  длин буферов имен                      */
    short  *C; /* Указатель на массив
                  текущих длин имен                      */
    char  **X; /* Указатель на массив
                  указателей на имена индикаторов        */
    short  *Y; /* Указатель на массив
                  длин буферов имен индикаторов          */
    short  *Z; /* Указатель на массив
                  текущих длин имен индикаторов          */
};
```

Рис. 18.16. SQLDA **в Oracle**

Вместо одного массива структур SQLVAR с описаниями столбцов, область SQLDA в Oracle содержит указатели на несколько массивов, описывающих различные параметры столбцов таблицы результатов запроса.

- Поле T указывает на массив целых чисел, представляющих собой коды типов данных столбцов или именованных параметров. Значения в этом массиве соответствуют полям SQLTYPE структур SQLVAR в DB2.
- Поле V адресует массив указателей на буферы для столбцов или передаваемых значений параметров. Указатели этого массива соответствуют полям SQLDATA структур SQLVAR в DB2.
- Поле L указывает на массив целых чисел, соответствующих размерам буферов, заданных в поле V. Значения в этом массиве соответствуют полям SQLLEN структур SQLVAR в DB2.
- Поле I адресует массив указателей на переменные-индикаторы для столбцов или именованных параметров. Указатели этого массива соответствуют полям SQLIND структур SQLVAR в DB2.
- Поле S адресует массив строковых указателей на буферы, в которых Oracle возвращает имена столбцов или именованных параметров. Эти буферы соответствуют структурам SQLNAME, которые являются членами структур SQLVAR в DB2.
- Поле M адресует массив целых чисел, соответствующих размерам буферов, указанных в поле S. Буферы для структур SQLNAME в DB2 имеют фиксированный размер, поэтому в DB2 отсутствует эквивалент поля M.
- Поле C указывает на массив целых чисел, которые представляют собой действительные длины имен, возвращаемых в буферы, заданные в поле S. Когда СУБД Oracle возвращает имена столбцов или параметров, она записывает в этот массив целые числа, указывающие действительные длины имен. Буферы для структур SQLNAME в DB2 имеют фиксированный размер, поэтому в DB2 отсутствует эквивалент поля C.
- Поле X адресует массив строковых указателей на буферы, в которые Oracle возвращает имена параметров-индикаторов. Эти буферы используются только в Oracle инструкцией DESCRIBE BLIND LIST; эквивалент для них в DB2 отсутствует.
- Поле Y указывает на массив целых чисел, представляющих собой размеры буферов, заданных в поле X. Эквивалент в DB2 отсутствует.
- Поле Z указывает на массив целых чисел, которые представляют собой действительные длины имен параметров-индикаторов, возвращаемых в буферы, заданные в поле X. Когда СУБД Oracle возвращает имена параметров-индикаторов, она записывает в этот массив целые числа, указывающие действительные длины имен. Эквивалент в DB2 отсутствует.

Преобразования типов данных

В DB2 применяются те же форматы типов данных (в частности, для приема значений параметров и возвращения результатов запроса), что и в мэйнфреймах S/370 компании IBM. Поскольку Oracle была спроектирована как переносимая СУБД, она использует свои собственные внутренние форматы типов данных. Когда Oracle получает значения параметров от программы или возвращает в программу результаты запроса, она автоматически проводит преобразования между внутренними форматами данных и форматами системы, в которой работает Oracle.

Используя область SQLDA, программа может управлять преобразованием типов данных, выполняемым в Oracle. Предположим, например, что ваша программа описывает результаты динамического запроса с помощью инструкции DESCRIBE и обнаруживает (по коду типа данных в области SQLDA), что первый столбец содержит числовые данные. Программа может запросить выполнение преобразования числовых данных, изменяя код типа данных в области SQLDA перед извлечением информации. Если программа, например, поместит в область SQLDA код типа данных символьной строки, то Oracle сам выполнит преобразование столбца таблицы результатов запроса и возвратит его программе как строку цифр.

Возможность преобразования типов данных в Oracle с помощью области SQLDA обеспечивает высокую степень переносимости программ как между различными компьютерными системами, так и между различными языками программирования. Аналогичная возможность имеется и в некоторых других СУБД, но в СУБД компании IBM такая возможность преобразования типов данных отсутствует.

Динамический SQL и стандарт SQL

Динамический SQL в стандарте SQL1 отсутствовал вовсе, поэтому стандартом де-факто для динамического SQL, как уже было сказано ранее, стала его реализация в СУБД DB2 компании IBM. В стандарте SQL2 динамическому SQL посвящена отдельная часть объемом примерно 50 страниц (при последнем обновлении в 2003 году она выросла и теперь превышает 400 страниц). В отношении наиболее простых аспектов стандарт SQL очень близок к тому динамическому SQL, который применяется в настоящее время в коммерческих СУБД. Но что касается других аспектов, включая даже базовые динамические запросы, новый стандарт несовместим с существующими СУБД. Чтобы уже разработанные программы соответствовали стандарту, их по сути придется переписать заново. В нескольких следующих разделах будут более подробно рассмотрены особенности стандартного динамического SQL, с акцентом на его отличиях от динамического SQL, применяемого в DB2 и описанного выше.

На практике динамический SQL, соответствующий стандарту, внедряется в коммерческие СУБД очень медленно, и старый динамический SQL в стиле DB2 по-прежнему широко используется в программировании. Даже если новая версия СУБД поддерживает стандарт SQL, в ней всегда предусматривается режим работы препроцессора, при котором он понимает старый динамический SQL, применяемый в данной СУБД. Часто именно этот режим является режимом препроцессора по умолчанию, так как необходимо, чтобы новые версии СУБД позволяли исполь-

зовать тысячи и тысячи уже существующих программ. Таким образом, реализация возможностей, предусмотренных стандартом SQL, которые не совместимы с существующей практикой, будет медленным эволюционным процессом.

Базовые динамические инструкции SQL

Синтаксические диаграммы базовых динамических инструкций (т.е. тех, которые не связаны с запросами на выборку) стандарта ANSI/ISO SQL изображены на рис. 18.17. Эти инструкции очень близки к диалекту DB2. В частности, они поддерживают одно- и двухэтапное выполнение динамических инструкций.

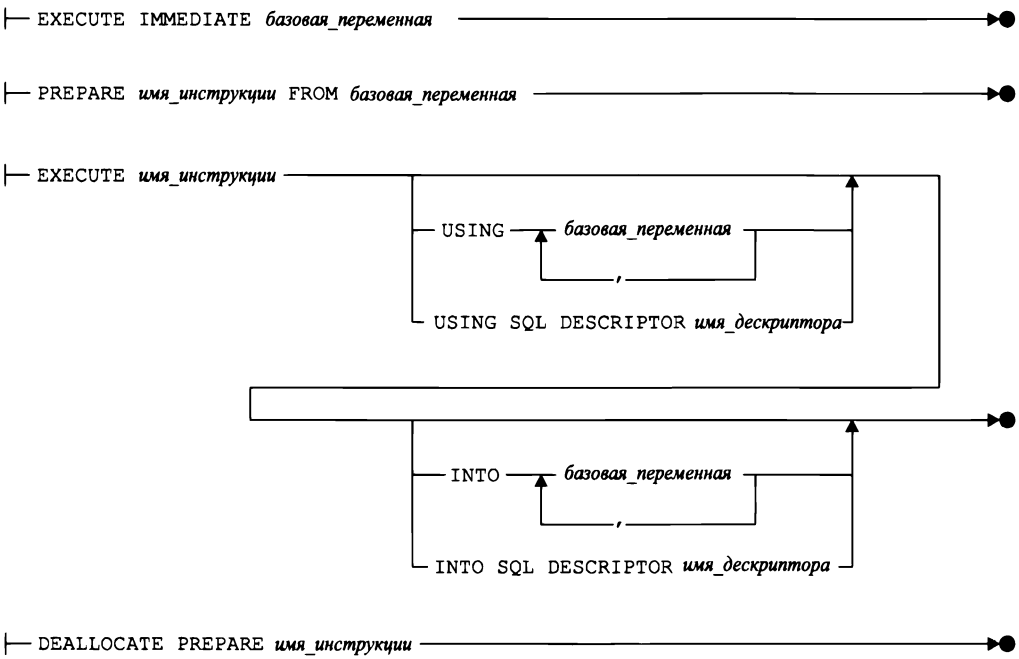


Рис. 18.17. Динамические инструкции стандарта SQL

Стандартная инструкция EXECUTE IMMEDIATE имеет синтаксис и назначение, идентичные синтаксису и назначению аналогичной инструкции в DB2. Она предназначена для немедленного выполнения инструкции SQL, переданной СУБД в виде строки символов. Таким образом, инструкция EXECUTE IMMEDIATE, представленная на рис. 18.2, соответствует стандарту SQL.

Имеющиеся в стандарте SQL инструкции PREPARE и EXECUTE также функционируют аналогично своим двойникам из DB2. Первая из них передает в СУБД текст инструкции SQL и дает команду проанализировать инструкцию, оптимизировать ее и создать для нее план выполнения. Вторая вызывает выполнение предварительно подготовленной инструкции. Эта инструкция EXECUTE, как и ее версия в DB2, может (необязательно) содержать базовые переменные, используемые для передачи значений параметров, необходимых при выполнении динамической

инструкции SQL. Таким образом, инструкции PREPARE и EXECUTE, представленные на рис. 18.4 (они обозначены цифрами 2 и 3), соответствуют стандарту SQL.

В разделе, касающемся полной совместимости со стандартом SQL (уровень Full), определены два полезных расширения инструкций PREPARE и EXECUTE. Первое из них — это инструкция DEALLOCATE PREPARE, дополняющая инструкцию PREPARE. Она отменяет подготовку ранее скомпилированной динамической инструкции SQL. Когда СУБД выполняет эту инструкцию, она может освобождать ресурсы, захваченные скомпилированной инструкцией; сюда, как правило, входит и некоторое внутреннее представление плана выполнения данной инструкции. Имя, указанное в инструкции DEALLOCATE PREPARE, должно быть идентичным имени в ранее выполненной инструкции PREPARE.

Учтите: если та или иная СУБД не реализует возможности, предоставляемые инструкцией DEALLOCATE PREPARE, то она никак не может узнать, будет ли выполняться подготовленная ранее инструкция еще, и поэтому должна хранить всю информацию, связанную с ней. На практике некоторые СУБД хранят скомпилированную версию инструкции только до конца транзакции; в последующих транзакциях инструкция должна быть подготовлена повторно. Поскольку это не очень эффективный подход, в ряде СУБД информация о скомпилированной инструкции хранится бесконечно долго. В таких случаях инструкция DEALLOCATE PREPARE может играть важную роль, особенно когда сеанс подключения к базе данных длится часами. Следует, однако, отметить, что в стандарте SQL четко указано: доступность подготовленной инструкции за пределами транзакции зависит от реализации.

Второе расширение касается инструкции EXECUTE, которая теперь может применяться для выполнения одиночной инструкции SELECT, возвращающей одну строку результатов запроса. Как и в DB2, стандартная инструкция EXECUTE включает предложение USING с именами базовых переменных, содержащих значения параметров запроса. Однако в стандарте добавлено дополнительное предложение INTO, содержащее имена базовых переменных, которые *принимают* результаты запроса, *возвращающего* одиночную строку.

Предположим, вы написали программу, которая динамически формирует инструкцию, возвращающую имя и план продаж служащего, чей идентификатор передан в качестве входного параметра. В DB2 даже столь простой запрос потребовал бы использовать область SQLDA, курсоры, цикл с инструкцией FETCH и т.д. При использовании стандартного динамического SQL достаточно выполнить всего две инструкции.

```
PREPARE qrystmt FROM  :statement_buffer;
EXECUTE qrystmt USING :emplnum INTO :name, :quota;
```

Как и в случае с любой другой подготовленной инструкцией, запрос qrystmt может выполняться многократно. Конечно, он все еще не избавлен от прежнего недостатка — число возвращаемых столбцов и их типы данных должны быть известны заранее, поскольку они должны в точности соответствовать количеству и типам базовых переменных, указанных в предложении INTO. Но этот недостаток можно устранить с помощью области дескриптора (эквивалент области SQLDA), заменяющей список базовых переменных, как описано в следующем разделе.

Стандартная SQLDA

Хотя обработка инструкций PREPARE/EXECUTE осуществляется схожим образом как в DB2, так и в SQL, обработка динамических запросов на выборку выполняется в стандарте совсем по-другому. В частности, большие различия связаны с областью SQLDA. Вспомним, что эта область обеспечивает выполнение двух важных функций:

- предоставление гибкого способа передачи параметров, которые будут использоваться при выполнении динамической инструкции SQL (что обеспечивает передачу данных из принимающей программы в СУБД);
- реализация способа получения результатов запроса при выполнении динамической инструкции SQL (что обеспечивает передачу данных из СУБД в принимающую программу).

В DB2 область SQLDA позволяет выполнять эти функции довольно гибко, однако она обладает рядом серьезных недостатков. Это структура данных очень низкого уровня, что приводит к ее тесной связи с конкретным языком программирования. Так, например, тот факт, например, что область SQLDA в DB2 имеет переменный размер, затрудняет ее представление в языке FORTRAN. Кроме того, при ее формировании были сделаны неявные предположения, касающиеся памяти компьютерной системы, в которой выполняется программа, использующая динамический SQL, — например, предположения о выравнивании данных в памяти. Для создателей стандарта SQL элементы низкого уровня явились помехой на пути обеспечения переносимости баз данных. Поэтому они заменили структуру SQLDA набором инструкций, предназначенных для работы с более абстрактной структурой данных, называемой динамическим SQL-*дескриптором*.

Структура стандартного дескриптора показана на рис. 18.18. Концептуально дескриптор играет точно такую же роль, что и область SQLDA в DB2, изображенная на рис. 18.7. В фиксированной части дескриптора находится счетчик числа элементов, расположенных в переменной части дескриптора. Каждый элемент переменной части содержит информацию об одном передаваемом параметре, такую как тип данных, длина, индикатор (указывающий, передается ли значение NULL) и т.п.

В отличие от области SQLDA в DB2, дескриптор не является структурой данных, определяемой в базовой программе. Он представляет собой коллекцию элементов данных, принадлежащих программному обеспечению СУБД. Программа выполняет операции над SQL-дескрипторами (создает, удаляет, заносит и извлекает данные) посредством нового набора динамических инструкций SQL, специально предназначенных для этой цели (рис. 18.19).

Чтобы понять, как работают инструкции управления дескрипторами, рассмотрим еще раз программу обновления таблицы, изображенную на рис. 18.8. Эта программа иллюстрирует использование области SQLDA в инструкции EXECUTE. Если вместо области SQLDA применить дескриптор стандарта SQL, то общий вид программы не изменится, но в нее будет внесено много мелких изменений.

Фиксированная часть	
COUNT	Количество описываемых элементов
Переменная часть — одна на каждый элемент (параметр или столбец таблицы результатов запроса)	
TYPE	Тип данных элемента
LENGTH	Длина элемента
OCTET_LENGTH	Длина элемента (в восьмибитовых октетах)
RETURNED_LENGTH	Длина возвращаемого элемента данных
RETURNED_OCTET_LENGTH	Длина возвращаемых данных (в восьмибитовых октетах)
PRECISION	Точность представления элемента данных
SCALE	Показатель масштабирования элемента данных
DATETIME_INTERVAL_CODE	Тип представления даты/времени
DATETIME_INTERVAL_PRECISION	Точность представления даты/времени
NULLABLE	Указывает, может ли элемент принимать значение NULL
INDICATOR	Указывает, содержит ли элемент данных значение NULL (индикатор)
DATA	Сам элемент данных
NAME	Имя элемента данных
UNNAMED	Указывает, отсутствует ли у элемента данных имя

Рис. 18.18. Структура дескриптора в стандарте SQL

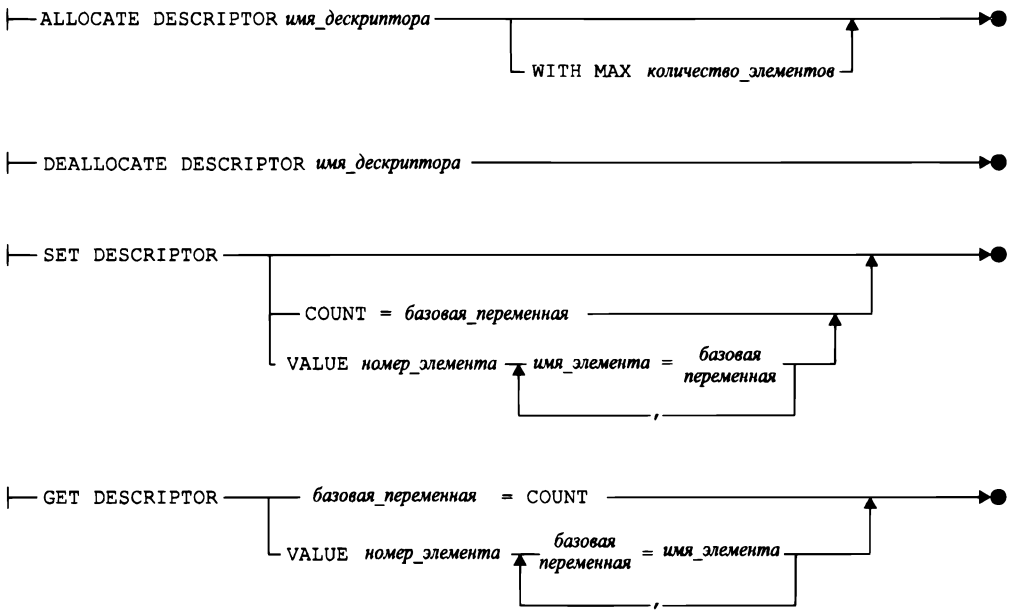


Рис. 18.19. Инструкции управления дескрипторами в стандарте SQL

Перед тем как использовать дескриптор, программа должна создать его с использованием следующей инструкции.

```
ALLOCATE DESCRIPTOR parmdesc WITH MAX :parmcnt;
```

Эта инструкция заменяет операцию выделения памяти для структуры parmda, обозначенную на рис. 18.8 цифрой 1. Дескриптор (названный parmdesc) будет выполнять те же функции, что и указанная структура. Обратите внимание на то, что программа, представленная на рис. 18.8, прежде чем выделять память для структуры parmda, должна сначала подсчитать требуемый ее объем. При использовании дескриптора эти вычисления не требуются, и базовая программа просто сообщает СУБД, сколько элементов должна вмещать переменная часть дескриптора.

Далее программа должна занести в дескриптор описания передаваемых параметров: их типы данных, длину и т.п. Цикл программы, обозначенный цифрой 2, остается без изменений, но детали передачи информации в дескриптор отличаются от аналогичной операции для области SQLDA. Операции занесения типа данных и длины параметра в область SQLDA (обозначенные цифрами 3 и 4) заменяются операцией передачи этой информации в дескриптор с помощью инструкций SET DESCRIPTOR, как показано ниже.

```
typecode = columns[i].typecode;
length = columns[j].buflen;
SET DESCRIPTOR parmdesc VALUE (:i + 1) TYPE = :typecode
SET DESCRIPTOR parmdesc VALUE (:i + 1) LENGTH = :length;
```

Отличия от рис. 18.8 весьма поучительны. Так как дескриптор находится в СУБД, тип данных и длина должны быть переданы в СУБД посредством инструкции SET DESCRIPTOR, использующей базовые переменные. В этом конкретном примере используются простые переменные typecode и length. Кроме того, в программе, представленной на рис. 18.8, используются коды типов данных, присущие только DB2. Различные коды типов данных, применяемые в разных СУБД, являются в динамическом SQL главной проблемой, препятствующей переносимости баз данных. Стандарт SQL устраняет эту проблему, так как в нем установлены целочисленные коды для всех типов данных, включенных в стандарт (табл. 18.2). Поэтому, помимо всех прочих изменений, коды типов данных в структуре columns, представленной на рис. 18.8, должны быть заменены кодами из стандарта SQL.

Инструкции, обозначенные на рис. 18.8 цифрами 5 и 6, привязывают структуру данных SQLDA к используемым программой буферам, в которых содержатся значения параметров и соответствующих переменных-индикаторов. Реально названные инструкции заносят в область SQLDA указатели на эти буферы, чтобы ими могла пользоваться СУБД. В случае дескрипторов такой тип привязки невозможен. Значения данных и индикаторов передаются позднее в программу как базовые переменные. Таким образом, при переходе к стандарту SQL инструкции, обозначенные цифрами 5 и 6, должны быть исключены из программы.

Инструкция, обозначенная на рис. 18.8 цифрой 7, записывает в область SQLDA число параметров, передаваемых в СУБД. В дескриптор также должно быть занесено количество передаваемых параметров. Это осуществляется с помощью следующей инструкции SET DESCRIPTOR.

```
SET DESCRIPTOR parmdesc COUNT = :parmcnt;
```

Таблица 18.2. Коды типов данных в SQL

Тип данных	Код
Коды типа данных (TYPE)	
INTEGER	4
SMALLINT	5
NUMERIC	2
DECIMAL	3
FLOAT	6
REAL	7
DOUBLE PRECISION	8
CHARACTER	1
CHARACTER VARYING	12
BIT	14
BIT VARYING	15
DATE/TIME/TIMESTAMP	9
INTERVAL	10
Подкоды формата даты/времени (INTERVAL_CODE)	
DATE	1
TIME	2
TIME WITH TIME ZONE	4
TIMESTAMP	3
TIMESTAMP WITH TIME ZONE	5
Подкоды формата даты/времени (INTERVAL_PRECISION)	
YEAR	1
MONTH	2
DAY	3
HOUR	4
MINUTE	5
SECOND	6
YEAR – MONTH	7
DAY – HOUR	8
DAY – MINUTE	9
DAY – SECOND	10
HOUR – MINUTE	11
HOUR – SECOND	12
MINUTE – SECOND	13

Строго говоря, эту инструкцию следует разместить в программе ранее и выполнить до инструкций для отдельных элементов. В стандарте SQL определен исчерпывающий набор правил, описывающих, как установка значений одних полей дескриптора влияет на сброс значений других его полей. По сути, эти правила устанавливают иерархию записи информации в дескриптор.

Например, если для какого-то параметра вы записываете в дескриптор код типа данных, указывающий на целое число, то поле LENGTH дескриптора будет сброшено в некое начальное значение, зависящее от реализации. Обычно это не влияет на процесс программирования, но необходимо четко понимать: если ранее в программе вы записали какое-то значение в дескриптор, то это не означает, что данное значение будет находиться там постоянно. Дескриптор необходимо заполнять по иерархическому принципу, начиная с информации высокого уровня (например, число элементов и их типы данных) и заканчивая более низкоуровневыми сведениями (длина элемента, подтип, разрешены ли значения NULL и т.п.).

Далее инструкция PREPARE компилирует динамическую инструкцию UPDATE; здесь никакие изменения не требуются. Затем программа входит в цикл for, запрашивая у пользователя значения параметров. Принципиальные различия здесь отсутствуют, но дескриптор используется не так, как область SQLDA.

Если пользователь указывает, что должно быть присвоено значение NULL (вводя звездочку в ответ на запрос), то программа на рис. 18.8 присваивает соответствующее значение буферу индикатора с помощью следующей инструкции.

```
*(parmvar -> sqlind) = -1;
```

Если же должно быть присвоено значение, отличное от NULL, то программа инициализирует буфер индикатора такой инструкцией.

```
*(parmvar -> sqlind) = 0;
```

В случае использования дескриптора эти инструкции заменяются парой инструкций SET DESCRIPTOR.

```
SET DESCRIPTOR parmdesc VALUE (:j + 1) INDICATOR = -1;
SET DESCRIPTOR parmdesc VALUE (:j + 1) INDICATOR = 0;
```

Обратите внимание на то, как счетчик цикла указывает, какой элемент заносится в дескриптор, а также на то, что осуществляется прямая передача данных (в данном случае констант), в отличие от использования указателей на буферы в области SQLDA.

Наконец, представленная на рис. 18.8 программа передает в СУБД через область SQLDA значение параметра, введенное пользователем. Инструкции, обозначенные цифрой 8, выполняют эту задачу для данных разных типов, преобразуя введенные символы в двоичные последовательности и помещая их в буферы, указанные в области SQLDA. При переходе к стандарту SQL указатели на буферы, а также функции, осуществляющие прямые операции над областью SQLDA, заменяются инструкцией SET DESCRIPTOR. Например, следующие инструкции передают в буфер строку символов переменной длины и указывают ее реальную длину.

```
length = strlen(inbuf);
SET DESCRIPTOR parmdesc VALUE (:j + 1) DATA = :inbuf;
SET DESCRIPTOR parmdesc VALUE (:j + 1) LENGTH = :length;
```

Если указание длины элемента данных не требуется, то передача данных упрощается, так как в этом случае необходима только одна инструкция SET DESCRIPTOR. Следует также отметить, что стандарт SQL подразумевает неявное приведение типов данных базовых переменных (таких, как inbuf) к типам данных SQL. Следуя стандарту SQL, программа, представленная на рис. 18.8, вынуждена была выполнять все эти преобразования в функциях sscanf(). Вместо этого данные можно передать в СУБД в символьном виде, что позволит выполнить автоматическое преобразование и обнаружить возможные ошибки.

Теперь, когда область SQLDA сформирована требуемым образом, программа на рис. 18.8 выполняет динамическую инструкцию UPDATE с передачей параметров при помощи инструкции EXECUTE, обозначенной цифрой 9. Эта инструкция довольно просто преобразуется при использовании стандартного дескриптора.

```
EXECUTE updatestmt USING SQL DESCRIPTOR parmdesc;
```

Ключевые слова в инструкции EXECUTE изменяются незначительно, а вместо имени области SQLDA указывается имя дескриптора.

И наконец, следует выполнить еще одну модификацию программы, представленной на рис. 18.8: освободить ресурсы, захваченные дескриптором. Это делается посредством инструкции

```
DEALLOCATE DESCRIPTOR parmdesc;
```

В такой простой программе, как программа, представленная на рис. 18.8, в освобождении ресурсов нет особой необходимости. Но в сложных реальных программах со множеством дескрипторов будет разумно освобождать ресурсы, занятые дескрипторами, когда они больше не требуются программе.

Стандарт SQL и динамические запросы на выборку

В предыдущем разделе дескриптор стандарта SQL, как и заменяемая им область SQLDA, использовался в динамических инструкциях SQL для передачи параметров, необходимых для выполнения динамической инструкции, из программы в СУБД. Стандартом SQL предусмотрено также использование дескрипторов в динамических запросах на выборку, где, подобно заменяемой ими области SQLDA, они управляют передачей результатов запросов из СУБД в базовую программу. На рис. 18.9 показана программа, выполняющая динамические запросы к СУБД DB2. Будет полезно рассмотреть, как следует преобразовать эту программу для того, чтобы она удовлетворяла требованиям стандарта SQL. Общая структура программы остается такой же, но в нее вновь будет внесено много мелких изменений. Варианты инструкций, связанных с выполнением динамических запросов на выборку и соответствующих стандарту SQL, представлены на рис. 18.20.

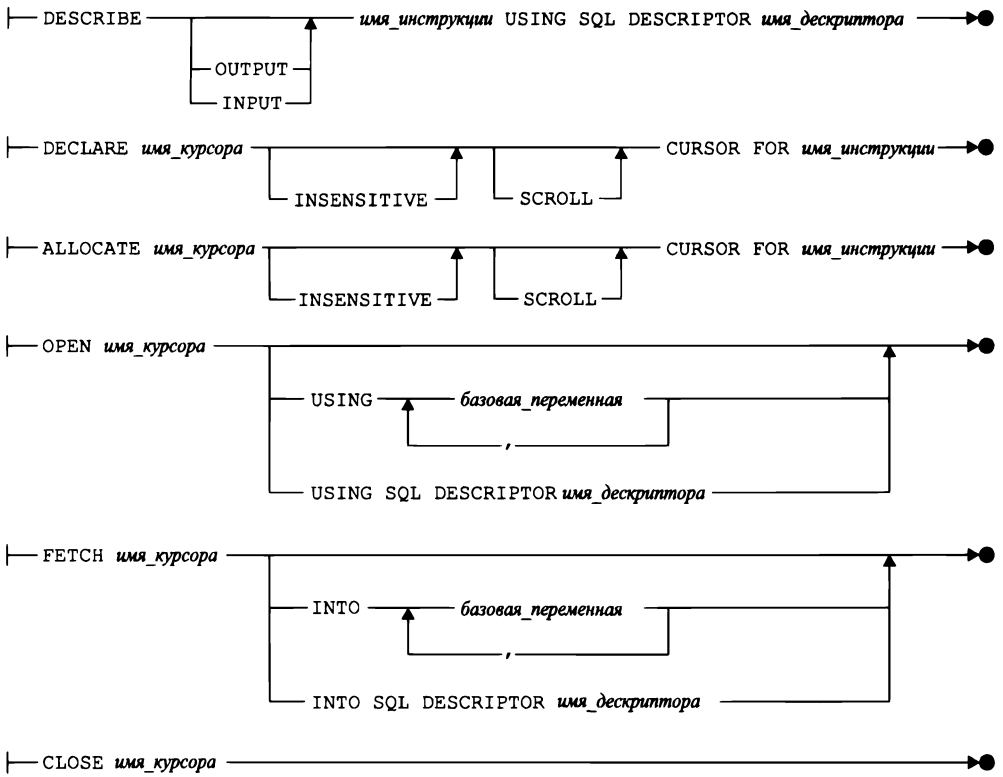


Рис. 18.20. Инструкции SQL, предназначенные для обработки динамических запросов на выборку

Объявление курсора для динамического запроса, обозначенное на рис. 18.9 цифрой 1, не изменяется. Формирование динамической инструкции SELECT (обозначено цифрой 2) и инструкция PREPARE (обозначена цифрой 3) также остаются без изменений. Изменения в программе начинаются с инструкции DESCRIBE (обозначена цифрой 4), возвращающей описание таблицы результатов запроса в область SQLDA с именем qry_da. Эта инструкция должна быть модифицирована так, чтобы в ней была ссылка на предварительно выделенный дескриптор. Предположим, что имя дескриптора qrydesc; тогда новые инструкции будут иметь следующий вид.

```
ALLOCATE DESCRIPTOR qrydesc WITH MAX :colcount;
DESCRIBE querystmt USING SQL DESCRIPTOR qrydesc;
```

Инструкция DESCRIBE стандарта SQL действует подобно замененной ею инструкции, только описания столбцов таблицы результатов запроса одно за другим возвращаются в дескриптор, а не в область SQLDA. Так как дескриптор принадлежит СУБД и не является структурой данных программы, последняя должна извлекать информацию из дескриптора. Эту функцию выполняет инструкция GET DESCRIPTOR, в отличие от инструкции SET DESCRIPTOR, заносящей информацию в дескриптор. В программе, представленной на рис. 18.9, инструкция, обо-

значенная цифрой 5 и получающая из области SQLDA длину отдельного столбца таблицы результатов запроса, должна быть заменена следующими инструкциями.

```
GET DESCRIPTOR qrydesc VALUE (:i + 1) :length = LENGTH;
qry_var->sqldata = malloc(length);
```

Выделение буферов для каждого элемента таблицы результатов запроса попрежнему необходимо, но способ передачи в СУБД адресов этих буферов в стандарте SQL меняется. Теперь эти адреса помещаются не в область SQLDA, а в дескриптор с помощью инструкции SET DESCRIPTOR. Буферы для переменных-индикаторов не требуются. Информацию о том, может ли столбец содержать значения NULL, можно получать из дескриптора непосредственно в процессе выборки каждой записи, как будет показано ниже в примере программы.

В данном конкретном примере число столбцов таблицы результатов запроса подсчитывается программой при формировании запроса. Но программа может также извлечь число столбцов из дескриптора с помощью инструкции GET DESCRIPTOR.

```
GET DESCRIPTOR qrydesc :colcount = COUNT;
```

После получения описания таблицы результатов запроса программа выполняет запрос, открывая курсор (обозначено цифрой 6). Простая форма инструкции OPEN без каких-либо входных параметров соответствует стандарту SQL. Если в динамическом запросе имеются параметры, они могут быть переданы в СУБД либо посредством базовых переменных, либо через дескриптор SQL. Инструкция OPEN стандарта SQL, использующая базовые переменные, идентична инструкции OPEN в DB2, приведенной на рис. 18.13. Инструкция OPEN стандарта SQL, использующая дескриптор, подобна инструкции EXECUTE стандарта SQL, также использующей дескриптор, и отличается от инструкции OPEN в DB2. Например, инструкция OPEN, изображенная на рис. 18.14,

```
OPEN qrycursor USING DESCRIPTOR :parmda;
```

заменяется, согласно стандарту SQL, инструкцией OPEN.

```
OPEN qrycursor USING SQL DESCRIPTOR parmdesc;
```

Методика передачи входных параметров в инструкцию OPEN через дескриптор та же, что и описанная ранее для инструкции EXECUTE.

Подобно реализации динамического SQL в СУБД Oracle, стандарт SQL предоставляет базовой программе возможность получать как описание параметров динамического запроса, так и описание таблицы результатов запроса. Для фрагмента программы, показанного на рис. 18.14, инструкция DESCRIBE

```
DESCRIBE INPUT querystmt USING SQL DESCRIPTOR parmdesc;
```

возвратит в дескриптор (с именем parmdesc) описание всех параметров динамического запроса. Число параметров может быть выяснено с помощью инструкции GET DESCRIPTOR, извлекающей из дескриптора элемент COUNT. Как и СУБД Oracle, стандарт SQL допускает существование двух дескрипторов, связанных с динамическим запросом. Входной дескриптор, получаемый с помощью инст-

рукции DESCRIBE INPUT, содержит описание параметров запроса. Выходной дескриптор содержит описание столбцов таблицы результатов запроса. Стандарт позволяет явно запрашивать выходной дескриптор

```
DESCRIBE OUTPUT querystmt USING SQL DESCRIPTOR qrydesc;
```

но по умолчанию используется инструкция DESCRIBE OUTPUT, так что ключевое слово OUTPUT обычно опускается.

Вернемся к программе, приведенной на рис. 18.9. Инструкция OPEN, обозначенная цифрой 6, открыла курсор, и теперь инструкция FETCH, обозначенная цифрой 7, извлекает строки из таблицы результатов запроса. Для соответствия стандарту SQL эта инструкция должна быть слегка модифицирована.

```
FETCH sqlcurs USING SQL DESCRIPTOR qrydesc;
```

Инструкция FETCH перемещает курсор на следующую строку таблицы результатов запроса и помещает все элементы этой строки в дескриптор, откуда программа может их затем извлечь. Программа по-прежнему должна получать дополнительную информацию о каждом столбце, в частности, его длину и то, может ли он содержать значения NULL. Например, определить длину столбца, содержащего символьные данные, программа может посредством инструкции

```
GET DESCRIPTOR qrydesc VALUE (:i + 1) :length = RETURNED_LENGTH;
```

Чтобы выяснить, содержится ли в столбце значение NULL, следует воспользоваться инструкцией

```
GET DESCRIPTOR qrydesc VALUE (:i + 1) :indbuf = INDICATOR;
```

а определить тип данных столбца можно так.

```
GET DESCRIPTOR qrydesc VALUE (:i + 1) :type = TYPE;
```

Как видите, построчная обработка результатов запроса внутри цикла for этой программы будет осуществляться совершенно иначе, чем в программе, представленной на рис. 18.9.

Обработав все строки таблицы результатов запроса, программа закрывает курсор. Инструкция CLOSE (на рис. 18.9 обозначена цифрой 8) соответствует стандарту SQL и потому не изменяется. После закрытия курсора хорошей практикой является освобождение всех дескрипторов, созданных в начале программы.

Изменения, которые необходимо произвести в программах динамического SQL, изображенных на рис. 18.8-18.9 и 18.14, для того чтобы они соответствовали стандарту SQL, детально иллюстрируют особенности нового стандарта и степень его отличия от динамического SQL, общеупотребительного в настоящее время. Вкратце, отличия стандартного динамического SQL от динамического SQL, применяемого в DB2, таковы.

- Структура данных SQLDA заменяется дескриптором, которому присваивается имя.
- Для создания и удаления дескрипторов служат инструкции ALLOCATE DESCRIPTOR и DEALLOCATE DESCRIPTOR. Они заменяют собой операции выделения памяти для области SQLDA и освобождения этой памяти.

- Программа передает информацию и значения параметров в СУБД с помощью инструкции SET DESCRIPTOR, а не использует для этой цели область SQLDA.
- Программа получает информацию о таблице результатов запроса и сами данные из этой таблицы с помощью инструкции GET DESCRIPTOR, а не использует для этого область SQLDA.
- Инструкция DESCRIBE применяется для получения описаний как таблицы результатов запроса (DESCRIBE OUTPUT), так и входных параметров запроса (DESCRIBE INPUT).
- Инструкции EXECUTE, OPEN и FETCH незначительно модифицируются — в них задается имя дескриптора вместо SQLDA.

Резюме

В настоящей главе был рассмотрен динамический SQL — усовершенствованная разновидность встроенного SQL. Динамический SQL не требуется при написании простых программ обработки данных, но он крайне необходим при создании клиентских приложений общего назначения, предназначенных для работы с базами данных. Статический и динамический SQL представляют собой классический пример компромисса между эффективностью и гибкостью, что выражается в следующем.

- **Простота**. Статический SQL относительно прост; даже самый сложный его элемент — курсоры — можно легко освоить, вспомнив концепцию файлового ввода-вывода. Динамический SQL довольно сложен; в нем осуществляется динамическое формирование инструкций, используются структуры данных переменной длины, выполняется распределение памяти и решаются другие сложные задачи.
- **Производительность**. Во время компиляции программы, использующей статический SQL, создается план выполнения всех встроенных инструкций; инструкции динамического SQL компилируются непосредственно на этапе выполнения. В результате производительность статического SQL, как правило, намного выше, чем динамического. Производительность динамического SQL существенно зависит от дизайна приложения; минимизация накладных расходов на компиляцию позволяет достичь производительности статического SQL.
- **Гибкость**. Динамический SQL дает программе возможность решать на этапе выполнения, какие конкретно инструкции SQL она будет выполнять. Статический SQL требует, чтобы все инструкции SQL были написаны заранее, на этапе создания программы; тем самым он ограничивает гибкость программы.

Функциональные возможности динамического SQL реализуются с помощью набора расширенных встроенных инструкций SQL.

- Инструкция EXECUTE IMMEDIATE обеспечивает передачу текста динамической инструкции SQL в СУБД, которая немедленно ее выполняет.
- Инструкция PREPARE передает текст динамической инструкции SQL в СУБД, которая компилирует ее (создает план выполнения), но не выполняет. Динамическая инструкция может содержать маркеры параметров, значения которых передаются в СУБД при выполнении инструкции.
- Инструкция EXECUTE дает СУБД команду выполнить динамическую инструкцию, скомпилированную ранее инструкцией PREPARE, а также передает значения параметров для выполняемой инструкции.
- Инструкция DESCRIBE возвращает в область SQLDA описание ранее подготовленной динамической инструкции. Если динамическая инструкция является запросом на выборку, то в ее описании приводятся все столбцы таблицы результатов запроса.
- Инструкция DECLARE CURSOR определяет курсор для запроса на выборку по имени, которое было присвоено запросу, когда он компилировался инструкцией PREPARE.
- Инструкция OPEN передает значения параметров, необходимых для выполнения динамической инструкции SELECT, и дает СУБД команду начать ее выполнение.
- Инструкция FETCH извлекает строку из таблицы результатов запроса и помещает ее элементы в области данных программы, указанные в SQLDA.
- Инструкция CLOSE завершает выполнение динамического запроса и прекращает доступ к таблице его результатов.

ГЛАВА 19

SQL API

Встроенный SQL как средство программного доступа к реляционным базам данных появился еще в первых реляционных СУБД компании IBM и с тех пор нашел широкое применение во многих популярных реляционных продуктах. Но в некоторых ведущих СУБД, начиная с самой первой реализации SQL Server компании Sybase, стал применяться принципиально иной подход. В таких СУБД пользователям предоставляется специальная библиотека функций, представляющая собой интерфейс прикладного программирования (API) между приложениями и СУБД. Прикладная программа вызывает функции SQL API для передачи в СУБД инструкций SQL и для получения от СУБД результатов запросов и служебной информации.

Для многих программистов применение SQL API — наиболее простой способ использования SQL. Большинство программистов имеют опыт работы с различными библиотеками функций, предназначенными для обработки строк, выполнения математических вычислений, организации файлового ввода-вывода или отображения информации на экране. В современных операционных системах, таких как UNIX или Windows, подобные библиотеки применяются чрезвычайно широко, так как они расширяют функциональные возможности самой операционной системы. Так что для программистов SQL API представляет собой просто новую библиотеку, которую необходимо освоить.

За последнее десятилетие популярность SQL API выросла и при разработке новых приложений стала равной, если не большей, популярности встроенного SQL. В настоящей главе описываются общие концепции, используемые всеми интерфейсами SQL API, и рассматриваются особенности API некоторых ведущих СУБД, а также стандарт ANSI/ISO SQL Call-Level Interface (CLI) и стандарт Microsoft Open Database Connectivity (ODBC), на базе которого был разработан упомянутый стандарт CLI. Наконец, в данной главе рассматривается JDBC — стандарт доступа к базам данных программ, написанных на Java, используемый большинством популярных серверов приложений Интернета.

Концепции API

Когда в СУБД поддерживается интерфейс вызова функций, приложение взаимодействует с СУБД только одним способом: вызывая функции, известные как *интерфейс прикладного программирования*, или API. Базовые операции типичного API СУБД показаны на рис. 19.1.

- Программа получает доступ к базе данных путем вызова одной или нескольких API-функций, подключающих программу к СУБД, а зачастую — и к конкретной базе данных или схеме.
- Для пересылки инструкции SQL в СУБД программа формирует инструкцию в виде текстовой строки (зачастую в переменной языка программирования) и затем передает эту строку в качестве параметра при вызове API-функции.
- Программа вызывает API-функции для проверки состояния переданной в СУБД инструкции и для обработки ошибок.
- Если инструкция SQL представляет собой запрос на выборку, то при вызове API-функций для получения результатов запроса программа записывает последние в свои переменные; обычно за один вызов возвращается одна строка или один столбец данных.
- Свое обращение к базе данных программа заканчивает вызовом API-функции, отключающей ее от СУБД.

SQL API часто используется в системах с архитектурой "клиент/сервер" (рис. 19.2). При этом библиотека, содержащая API-функции, расположена на клиентском компьютере, где выполняется приложение, а СУБД — на сервере, где находится база данных. Приложение вызывает API-функции локально, на клиентском компьютере, но связь между API-функциями и СУБД осуществляется по сети. Далее в настоящей главе будет показано, что применение SQL API в архитектуре "клиент/сервер" дает большие преимущества, поскольку позволяет свести к минимуму сетевой трафик.

Ранние интерфейсы программирования приложений в различных СУБД существенно отличались друг от друга. Как и в случае со многими другими аспектами языка SQL, эти интерфейсы просуществовали долгое время, прежде чем стали предприниматься попытки их стандартизировать. Кроме того, они в значительно большей степени связаны с особенностями конкретной СУБД, чем встроенный SQL. Тем не менее в основе всех интерфейсов, применяемых в коммерческих СУБД, лежат одни и те же принципы, проиллюстрированные на рис. 19.1 и 19.2. Это относится и к ODBC API, а также созданному позднее на его основе стандарту ANSI/ISO.

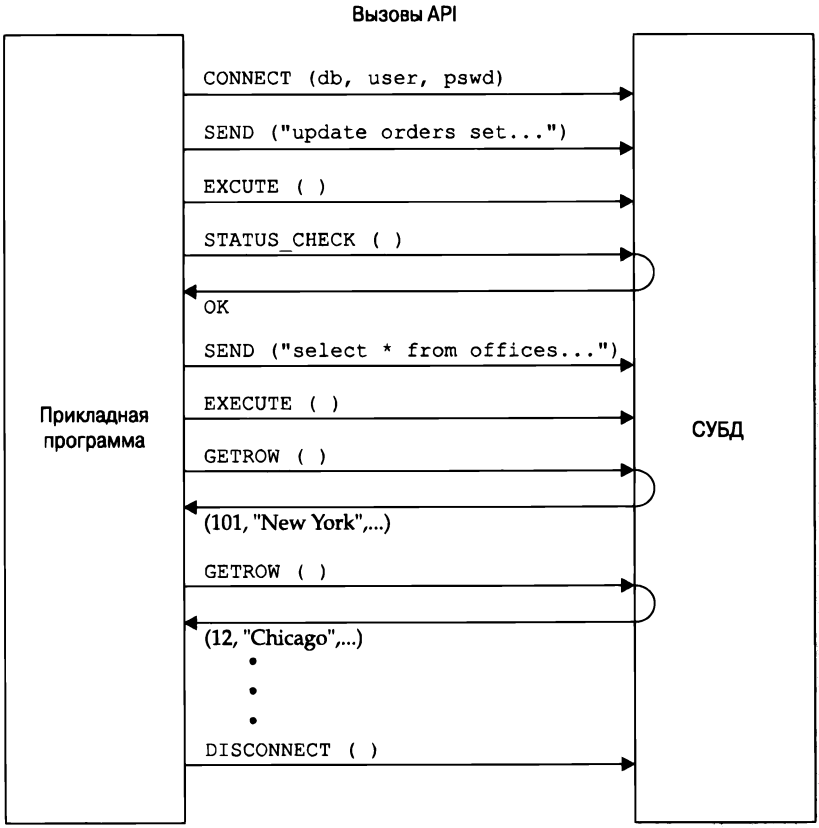


Рис. 19.1. Применение SQL API для доступа к базе данных

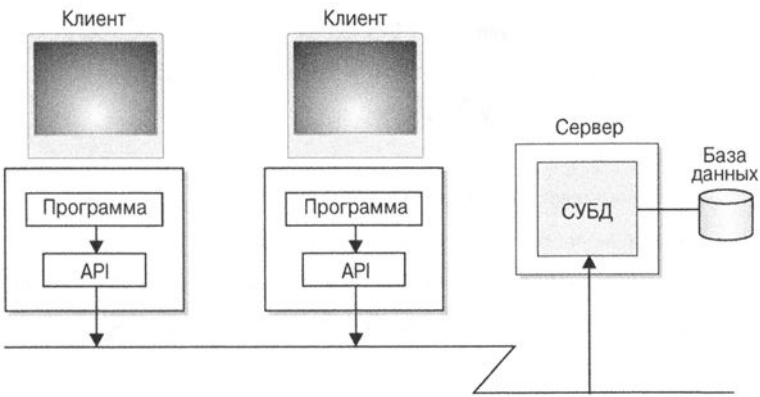


Рис. 19.2. SQL API в архитектуре “клиент/сервер”

dblib API (SQL Server)

Первой популярной СУБД, в которой появился интерфейс вызовов функций, была SQL Server, разработанная компаниями Sybase и Microsoft. Многие годы программный доступ к данной СУБД можно было осуществлять только посредством этого интерфейса. В настоящее время в SQL Server уже имеется встроенный SQL, а также новые интерфейсы более высокого уровня, но исходный SQL Server API все еще остается популярным средством доступа к базам данных, находящимся под управлением этой СУБД. Кроме того, он послужил основой, на которой впоследствии был сформирован Microsoft ODBC API. SQL Server и ее API, помимо всего прочего, являются еще и прекрасным примером системы, которая изначально проектировалась для архитектуры "клиент/сервер". Вот почему будет полезно начать изучение интерфейсов вызовов SQL-функций с базового интерфейса SQL Server.

Изначально SQL Server API, который называется *библиотекой базы данных*, сокращенно `dblib`, состоял примерно из 100 функций, доступных прикладным программам. Это весьма сложный и обширный интерфейс, однако типичная программа использует не более двух десятков функций, перечисленных в табл. 19.1. Остальные функции реализуют более сложные программные возможности, альтернативные методы взаимодействия с СУБД или обобщают несколько функций более низкого уровня.

Таблица 19.1. Основные API-функции в SQL Server

Функция	Описание
Подключение/отключение от базы данных	
`dblogin()`	Создает структуру, которая будет хранить информацию о подключении к SQL Server
`dbopen()`	Устанавливает соединение с SQL Server
`dbuse()`	Определяет базу данных, используемую по умолчанию
`dbexit()`	Разрывает соединение с SQL Server
Базовая обработка инструкций	
`dbcmd()`	Передает текст инструкции SQL в `dblib`
`dbsqlexec()`	Запрашивает выполнение пакета инструкций SQL
`dbresults()`	Получает результаты выполнения следующей инструкции SQL в пакете
`dbcancel()`	Отменяет выполнение оставшейся части пакета инструкций SQL
Обработка ошибок	
`dbmsghandle()`	Определяет пользовательскую процедуру обработки сообщений
`dberrhandle()`	Определяет пользовательскую процедуру обработки ошибок

Окончание табл. 19.1

Функция	Описание
Обработка результатов запроса на выборку	
dbbind()	Связывает столбец таблицы результатов запроса с программной переменной
dbnextrow()	Извлекает следующую строку из таблицы результатов запроса
dbnumcols()	Получает количество столбцов в таблице результатов запроса
dbcolname()	Получает имя столбца в таблице результатов запроса
dbcoltype()	Получает тип данных столбца в таблице результатов запроса
dbcollen()	Получает максимальную длину столбца в таблице результатов запроса
dbdata()	Возвращает указатель на извлеченные данные
dbdatlen()	Получает действительную длину извлеченных данных
dbcanquery()	Отменяет запрос, не дожидаясь, пока будут извлечены все строки

Основы работы с SQL Server

Простая программа, обновляющая базу данных SQL Server, может использовать всего несколько функций из dblib. Программа, текст которой приведен на рис. 19.3, обновляет столбец QUOTA таблицы SALESREPS в учебной базе данных. Эта программа выполняет ту же задачу, что и программа, представленная на рис. 17.17 (воспроизведенная на рис. 19.4), но здесь вместо встроенного SQL используется SQL Server API. На рис. 19.3 проиллюстрированы основные этапы взаимодействия между программой и SQL Server.

1. Программа с помощью функции dblogin() готовит учетную запись, вносся в нее имя пользователя, пароль и всю остальную информацию, которая необходима для подключения к СУБД.
2. Для подключения к базе данных программа вызывает функцию dbopen(). Подключение должно быть установлено до того, как программа начнет посылать в SQL Server инструкции SQL.
3. Программа формирует инструкцию SQL и вызывает функцию dbcmd(), чтобы передать текст этой инструкции в dblib. В результате последовательных вызовов функции dbcmd() очередной фрагмент текста добавляется к ранее переданному; необязательно, чтобы за один вызов функции dbcmd() передавался полностью весь текст инструкции SQL.
4. Программа вызывает функцию dbsqlexec(), давая SQL Server команду выполнить инструкцию SQL, переданную ранее функцией dbcmd().
5. Программа вызывает функцию dbresults(), чтобы определить, успешно ли была выполнена инструкция SQL.
6. Программа вызывает функцию dbexit(), чтобы разорвать соединение с SQL Server.

```
main()
{
    LOGINREC  *loginrec;  /* Структура данных учетной записи  */
    DBPROCESS *dbproc;    /* Структура данных для подключения */
    char amount_str[ 31 ];/* Вводимое пользователем значение  */
    int  status;          /* Состояние dblib                  */

    /* Создание учетной записи  */
    loginrec = dblogin();                          ←———— ①
    DBSETLUSER( loginrec, "scott" );               ←———— ①
    DBSETLPWD ( loginrec, "tiger" );               ←———— ①

    /* Подключение к SQL Server */
    dbproc = dbopen( loginrec, "" );               ←———— ②

    /* Приглашение пользователю ввести величину
       увеличения/уменьшения плана продаж       */
    printf( "На сколько изменить объем продаж:" );
    gets( amount_str );

    /* Передача инструкции SQL в dblib          */
    dbcmd( dbproc,
           "update salesreps set quota = quota + " ); ←———— ③
    dbcmd( dbproc, amount_str );                   ←———— ③

    /* Выполнение инструкции */
    dbsqlexec( dbproc );                           ←———— ④

    /* Получение результата выполнения */
    status = dbresults( dbproc );                  ←———— ⑤

    if ( status != SUCCEED )
        printf( "Ошибка обновления.\n" );
    else
        printf( "Обновление успешное.\n" );

    /* Отсоединение от SQL Server */
    dbexit( dbproc );                              ←———— ⑥

    exit();
}
```

Рис. 19.3. Простая программа, использующая `dblib`

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        float amount;    /* Вводится пользователем */
    exec sql end declare section;

    /* Приглашение ввести изменение плана  продаж  */
    printf("На сколько изменить объем продаж: ");
    scanf("%f", &amount);

    /* Обновление столбца QUOTA таблицы SALESREPS  */
    exec sql update salesreps
                set quota = quota + :amount;
```

Рис. 19.4. Программа с рис. 19.3 с применением встроенного SQL

```
    /* Проверка результата выполнения инструкции   */
    if (sqlca.sqlcode != 0)
        printf("Ошибка при обновлении.\n");
    else
        printf("Обновление успешное.\n");

    exit();
}
```

Окончание рис. 19.4

Будет полезно сравнить программы, приведенные на рис. 19.3 и 19.4, чтобы отметить различия между встроенным SQL и библиотекой dblib.

- Программа со встроенным SQL либо неявно подключается к единственной доступной базе данных (как в DB2), либо содержит встроенную инструкцию SQL для подключения (такую, как инструкция CONNECT в стандарте SQL). Программа, использующая библиотеку dblib, подключается к SQL Server путем вызова функции dbopen().
- Инструкции UPDATE, которые получает СУБД, в обеих программах идентичны. В программе со встроенным SQL эта инструкция является частью исходного текста программы. В программе, использующей библиотеку dblib, инструкция передается в виде одной строки или последовательности из нескольких символьных строк. Фактически методика, применяемая в dblib, больше напоминает инструкцию EXECUTE IMMEDIATE из динамического SQL, чем статический SQL.
- В программе со встроенным SQL базовые переменные обеспечивают связь между инструкциями SQL и значениями программных переменных. При использовании библиотеки dblib программа передает в СУБД значения переменных тем же способом, что и текст программы, — как часть строки инструкции SQL.
- В случае встроенного SQL ошибки возвращаются в полях SQLCODE или SQLSTATE структуры SQLCA. В случае библиотеки dblib вызов функции dbresults() позволяет определить код завершения любой инструкции SQL.

Подводя итоги сравнения, можно сказать, что программа со встроенным SQL (рис. 19.4) короче и более удобочитаема. Однако она не является программой, написанной на чистом C или чистом SQL, и программист должен знать встроенный SQL, чтобы ее понять. Использование базовых переменных означает, что интерактивная и встроенная формы инструкции SQL отличаются. Кроме того, программу со встроенным SQL необходимо дополнительно обрабатывать с помощью специального препроцессора SQL, что увеличивает время компиляции. В противоположность этому программа для SQL Server является простой программой на C без какого-либо "наполнителя", не требует специальной методики программирования и может сразу обрабатываться компилятором языка C.

Пакеты инструкций

Программа, представленная на рис. 19.3, посылает SQL Server одну инструкцию SQL и затем проверяет код ее завершения. Если прикладной программе необходимо выполнить несколько инструкций SQL, она может повторять цикл dbcmd()/dbsqlexec()/dbresults() для каждой инструкции. Но есть и другой способ — программа может послать в SQL Server несколько инструкций для одновременного выполнения в виде единого *пакета инструкций*.

На рис. 19.5 приведен текст программы, в которой используется пакет из трех инструкций. Как и предыдущая программа на рис. 19.3, эта программа вызывает функцию dbcmd() для передачи текста инструкций SQL в dblib. В библиотеке dblib строки инструкций просто объединяются. Обратите внимание: включать в передаваемый текст пробелы или другие необходимые знаки препинания должна программа. СУБД SQL Server не станет выполнять инструкции до тех пор, пока программа не вызовет функцию dbsqlexec(). В этом примере в SQL Server были посланы три инструкции, поэтому программа вызывает функцию dbresults() три раза подряд. При каждом вызове этой функции СУБД возвращает результаты очередной инструкции пакета, сообщая программе, успешно ли она выполнена.

```
main()
{
    LOGINREC  *loginrec;  /* Структура данных учетной записи  */
    DBPROCESS *dbproc;    /* Структура данных для подключения */
    .
    .
    .

    /* Удаление служащих с малыми продажами */
    dbcmd(dbproc,
          "delete from salesreps where sales < 10000.00");

    /* Повышение плана продаж служащим со средними продажами  */
    dbcmd(dbproc,
          "update salesreps set quota = quota + 10000.00 ");
    dbcmd(dbproc,"wheresales <= 150000.00");

    /* Повышение плана продаж служащим с высокими продажами   */
    dbcmd(dbproc,
          "update salesreps set quota = quota + 20000.00 ");
    dbcmd(dbproc,"where sales > 150000.00");

    /* Выполнение пакета инструкций */
    dbsqlexec(dbproc);

    /* Проверка результата выполнения
       каждой из трех инструкций */
    if(dbresults(dbproc) != SUCCEED) goto do_error;
    if(dbresults(dbproc) != SUCCEED) goto do_error;
    if(dbresults(dbproc) != SUCCEED) goto do_error;
    .
    .
    .
}
```

Рис. 19.5. Использование пакета инструкций в dblib

При разработке программы, текст которой приведен на рис. 19.5, программист заранее знает, что пакет содержит три инструкции, и может сделать в программе три соответствующих вызова функции dbresults(). Если число инструкций в пакете заранее неизвестно, программа может вызывать функцию dbresults() многократно — до тех пор, пока не получит код ошибки NO_MORE_RESULTS. Фрагмент программы, приведенный на рис. 19.6, иллюстрирует эту методику.

```
.
.
.

/* Выполнение пакета инструкций,
   созданного вызовами dbcmd()            */
dbsqlexec(dbproc);

/* Цикл проверки результатов выполнения
   каждой инструкции пакета */
while( status = dbresults(dbproc) != NO_MORE_RESULTS )
{
    if ( status == FAIL )
        goto handle_error;
    else
        printf("Инструкция выполнена успешно.\n");
}

/* Цикл завершен, пакет выполнен успешно */
printf("Пакет выполнен.\n");
exit();
.
.
.
```

Рис. 19.6. Обработка результатов выполнения пакета инструкций в dblib

Обработка ошибок

Значение, возвращаемое функцией dbresults(), сообщает программе о том, успешно ли была выполнена соответствующая инструкция пакета. Чтобы получить более детальную информацию об ошибке, программа должна иметь свою собственную функцию обработки сообщений. Когда в SQL Server при выполнении инструкции возникает ошибка, библиотека dblib автоматически вызывает функцию обработки сообщений. Отметим, что библиотека вызывает эту функцию во время выполнения функции dbsqlexec() или dbresults(), т.е. до того как они вернут управление программе (так называемые функции обратного вызова, вызываемые программным обеспечением SQL Server). Это позволяет функции обработки сообщений выполнить свою собственную обработку ошибки.

На рис. 19.7 показан фрагмент программы для SQL Server, который содержит функцию обработки сообщений msg_rtn(). В начале своего выполнения программа регистрирует эту функцию, вызывая функцию dbmsghandle(). Предположим, что ошибка происходит в то время, когда SQL Server обрабатывает инструкцию DELETE. Когда программа вызывает функцию dbsqlexec() или dbresults() и библиотека dblib получает от SQL Server сообщение об ошибке, она вызывает из выполняемой функции подпрограмму msg_rtn() данной программы, передавая ей пять параметров:

- dbproc — подключение, при котором произошла ошибка;
- msgno — номер ошибки;
- msgstate — информация о контексте ошибки;
- severity — категория сложности, показывающая, насколько серьезна данная ошибка;
- msgtext — сообщение об ошибке, соответствующее номеру msgno.

```
    .
    .
    .
/* Внешние переменные для хранения информации об ошибках   */
int  errcode;     /* Код ошибки                            */
char errmsg[256]; /* Сообщение об ошибке                   */

/* Определение собственной функции для обработки сообщений */
int msg_rtn( DBPROCESS *dbproc, DBINT  msgno,
             int msgstate, int  severity,
             char * msgtext)
{
    /* Вывод кода ошибки и сообщения */
    printf("*** Ошибка: %d Сообщение: %s\n", msgno, msgtext);

    /* Сохранение информации об ошибке для программы       */
    errcode = msgno;
    strcpy(errmsg,msgtext);

    /* Возврат в dblib для завершения вызова API           */
    return(0);
}


main()
{
    DBPROCESS *dbproc; /* Структура данных для подключения */
    .
    .
    .
    /* Установка собственной процедуры обработки ошибок    */
    dberrhandle(msg_rtn);
    .
    .
    .
    /* Выполнение инструкции удаления                      */
    dbcmd(dbproc,"delete from salesreps "
                 "where quota < 100000.00");
    dbsqlexec(dbproc);
    dbresults(dbproc);
    .
    .
    .
```

Рис. 19.7. Обработка ошибок при использовании dblib

Функция msg_rtn() данной программы обрабатывает сообщение, выводя его на экран и сохраняя код ошибки в программной переменной для дальнейшего использования. Когда функция обработки сообщений возвращает управление библиотеке dblib (которая ее вызвала), библиотека заканчивает выполнение функции, сгенерировавшей ошибку, и возвращает в программу флаг состояния FAIL. Программа может обнаружить этот флаг и при необходимости произвести дальнейшую обработку ошибки.

Процедура обработки ошибок, осуществляемая в SQL Server, представлена в рассматриваемой программе несколько упрощенно. Помимо ошибок инструкций SQL, обнаруживаемых в SQL Server, ошибки могут также происходить и в самой библиотеке dblib. Например, если соединение с SQL Server будет разорвано, время ожидания ответа библиотекой dblib может превысить установленный лимит, в результате чего возникнет ошибка. Библиотека обрабатывает такие ошибки путем вызова отдельной функции обработки ошибок, которая во многом подобна описанной выше функции обработки сообщений.

Из сравнения программы на рис. 19.7 с программами на рис. 17.10 и 17.14 (воспроизведены на рис. 19.8 и 19.9) можно увидеть различия в методике обработки ошибок между библиотекой dblib и встроенным SQL.

- Во встроенном SQL программа узнает об ошибках и предупреждениях, обращаясь к области SQLCA. SQL Server сообщает об ошибках и предупреждениях, вызывая из выполняемой функции специальные функции прикладной программы и передавая им код и описание возникшей ошибки.
- Во встроенном SQL обработка ошибок осуществляется синхронно. Когда во время выполнения встроенной инструкции SQL происходит ошибка, управление возвращается программе и проверяется значение переменных SQLCODE/SQLSTATE. Обработка ошибок в SQL Server происходит асинхронно. Когда при выполнении API-функции происходит ошибка, SQL Server вызывает функцию обработки ошибок или сообщений, входящую в прикладную программу, *во время* выполнения самой API-функции. Код ошибки в программу эта API-функция возвращает позднее.
- Во встроенном SQL определен только один тип ошибки и существует один механизм сообщения о ней. В SQL Server определены два типа ошибок и существуют два параллельных механизма.

Словом, обработка ошибок во встроенном SQL осуществляется просто и понятно, но прикладная программа имеет ограниченное число вариантов ответных действий при возникновении ошибок. Программа для SQL Server обрабатывает ошибки более гибко. Однако схема вызова из выполняемой функции, которая используется в библиотеке dblib, довольно сложна, и если системные программисты хорошо с ней знакомы, то прикладные программисты могут ее и не знать.

```
            .
            .
            .
        exec sql delete from salesreps
                 where quota < 150000;
        if (sqlca.sqlcode < 0)
            goto error_routine;
            .
            .
            .
        error_routine:
            printf("Ошибка SQL: %ld\n", sqlca.sqlcode);
            exit();
            .
            .
            .
```

Рис. 19.8. Отрывок программы на языке С с проверкой SQLCODE

```
    .
    .
    .
    /* Выполнение инструкции DELETE и проверка на ошибки  */
    exec sql delete from salesreps
             where quota < 150000;
    if (strcmp(sqlca.sqlstate,"00000"))
        goto error_routine;

    /* Удаление успешное; ищем количество удаленных строк */
    exec sql get diagnostics :numrows = ROW_COUNT;
    printf("Удалено %ld строк\n",numrows);
    .
    .
    .
error_routine:
    /* Находим количество сообщений об ошибках            */
    exec sql get diagnostics :count = NUMBER;
    for (i=1; i<count; i++)
    {
        exec sql get diagnostics EXCEPTION :I
                            :err = RETURNED_SQLSTATE,
                            :msg = MESSAGE_TEXT;
        printf("Ошибка SQL # %d: код: %s сообщение: %s\n",
               i, err, msg);
    }
    exit();
    .
    .
    .
```

Рис. 19.9. Отрывок программы на языке С с применением инструкции GET DIAGNOSTICS

Запросы на выборку в SQL Server

1. Методика выполнения программных запросов на выборку в SQL Server очень похожа на методику выполнения других инструкций SQL. Для выполнения запроса программа посылает в SQL Server инструкцию SELECT и с помощью библиотеки dblib строка за строкой извлекает таблицу ре-

зультатов запроса. Программа, представленная на рис. 19.10, иллюстрирует методику выполнения запроса в SQL Server.

2. Чтобы передать в SQL Server инструкцию SELECT и дать команду на ее выполнение, программа вызывает функции dbcmd() и dbsqlexec().
3. Когда после выполнения инструкции SELECT программа вызывает функцию dbresults(), библиотека dblib возвращает код завершения запроса и делает доступной для обработки таблицу результатов запроса.
4. Программа вызывает функцию dbbind() по одному разу для каждого столбца из таблицы результатов запроса, сообщая библиотеке dblib, куда она должна возвращать данные каждого отдельного столбца. Аргументы функции dbbind() указывают номер столбца, ожидаемый тип данных, размер буфера для приема данных и адрес буфера.
5. Программа выполняет цикл, многократно вызывая функцию dbnextrow() для получения строк из таблицы результатов запроса. Библиотека dblib помещает возвращаемые данные в буферы, указанные в предшествующих вызовах функции dbbind().
6. Когда все строки будут получены, функция dbnextrow() возвратит значение NO_MORE_ROWS. Если в пакете за инструкцией SELECT следуют другие инструкции, программа может вызвать функцию dbresults(), чтобы перейти к следующей инструкции.

```
main()
{
    LOGINREC  *loginrec; /* Структура данных учетной записи  */
    DBPROCESS *dbproc;   /* Структура данных для подключения */
    char repname[16];    /* Полученный город офиса           */
    short repquota;      /* Полученный план продаж           */
    float repsales;      /* Полученный объем продаж          */

    /* Создание учетной записи и подключение к SQL Server    */
    loginrec = dblogin();
    DBSETLUSER( loginrec, "scott" );
    DBSETLPWD ( loginrec, "tiger" );
    dbproc = dbopen( loginrec, "" );

    /* Передача запроса SQL Server для выполнения            */
    dbcmd(dbproc,"select name, quota, sales from salesreps ");
    dbcmd(dbproc,"where sales > quota order by name");  <----①
    dbsqlexec(dbproc);  <------------------------------------①

    /* Получение первой инструкции пакета                    */
    dbresults(dbproc);  <------------------------------------②

    /* Связываем каждый столбец с переменной программы       */
    dbbind(dbproc, 1, NTBSTRINGBIND, 16, &repname);  <----③
    dbbind(dbproc, 2, FLT4BIND,       0, &repquota);  <----③
    dbbind(dbproc, 3, FLT4BIND,       0, &repsales);  <----③

    /* Цикл получения строк результатов запроса              */
    while(status = dbnextrow(dbproc) == SUCCEED)  <----------④
```

Рис. 19.10. Получение результатов запроса в SQL Server

```
{
        /* Вывод данных о служащем                                  */
        printf("Имя         : %s\n",  repname);
        printf("План продаж: %f\n\n",repquota);
        printf("Продажи     : %f\n",  repsales);
    }

    /* Проверка на ошибки и закрытие соединения */          ←——⑤
    if (status == FAIL)
    {
        printf("Ошибка SQL.\n");
        dbexit(dbproc);
        exit();
    }
}
```

Окончание рис. 19.10

Обработка таблицы результатов запроса в рассматриваемой программе (рис. 19.10) осуществляется с помощью двух функций библиотеки dblib: dbbind() и dbnextrow(). Функция dbbind() устанавливает взаимно однозначное соответствие между столбцами таблицы результатов запроса и программными переменными, принимающими извлекаемые данные. Этот процесс называется *привязкой* столбцов. Первый столбец, NAME, привязывается к 16-символьному массиву и будет возвращен в виде строки, оканчивающейся нулевым символом. Второй и третий столбцы, QUOTA и SALES, привязываются к числовым переменным. Обязанность программиста — обеспечить, чтобы тип данных каждого столбца был совместим с типом данных программной переменной, к которой он привязывается.

И вновь будет полезно сравнить обработку запроса в SQL Server (рис. 19.10) с обработкой запроса во встроенном SQL (исходные тексты на рис. 17.20 и 17.23 воспроизведены на рис. 19.11 и 19.12).

- Во встроенном SQL существует два метода обработки запросов: для запросов, возвращающих отдельную запись (одиночная инструкция SELECT), и для запросов, возвращающих наборы записей. В SQL Server используется единый метод, независимо от количества строк в таблице результатов запроса.
- Во встроенном SQL запрос выражается в виде либо одиночной инструкции SELECT, либо инструкции DECLARE CURSOR, которые отличаются от интерактивной инструкции SELECT для данного запроса. В SQL Server инструкция SELECT, посылаемая программой, *идентична* интерактивной инструкции SELECT для данного запроса.
- Во встроенном SQL базовые переменные, принимающие результаты запроса, задаются в предложении INTO одиночной инструкции SELECT или инструкции FETCH. В случае с SQL Server соответствующие программные переменные задаются в функции dbbind().
- Во встроенном SQL построчный доступ к таблице результатов запроса обеспечивается с помощью специально предназначенных для этой цели инструкций SQL (OPEN, FETCH и CLOSE). В SQL Server доступ к таблице результатов запроса осуществляется посредством вызовов функций библиотеки dblib: dbresults() и dbnextrow().

```
main()
{
    exec sql begin declare section;
        int   repnum;       /* Номер служащего (вводится) */
        char  repname[16]; /* Полученное имя служащего   */
        float repquota;     /* Полученный план продаж     */
        float repsales;     /* Полученные продажи         */
    exec sql end declare section;

    /* Запрос номера служащего у пользователя программы  */
    printf("Введите номер служащего:");
    scanf("%d", &repnum);

    /* Выполнение запроса SQL                            */
    exec sql select name, quota, sales
              into :repname, :repquota, :repsales
              from salesreps
             where empl_num = :repnum;

    /* Вывод полученных данных                           */
    if (sqlca.sqlcode = = 0)
    {
        printf("Имя:     %s\n", repname);
        printf("План:    %f\n", repquota);
        printf("Продажи: %f\n", repsales);
    }
    else
        if (sqlca.sqlcode = = 100)
            printf("Служащего с таким номером нет.\n");
        else
            printf("Ошибка SQL: %ld\n", sqlca.sqlcode);

    exit();
}
```

Рис. 19.11. Применение одиночной инструкции SELECT

```
main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        char  repname[16];  /* Полученное имя служащего */
        float repquota;     /* Полученный план продаж   */
        float repsales;     /* Полученные продажи       */
        short repquota_ind; /* Индикатор значения NULL  */
    exec sql end declare section;

    /* Объявление курсора */
    exec sql declare repcurs cursor for   ←————————①
              select name, quota, sales
                from salesreps
               where sales > quota
               order by name;

    /* Обработка ошибок */
    whenever sqlerror goto error;
    whenever not found goto done;

    /* Открытие курсора для начала выполнения запроса */
    exec sql open repcurs;   ←————————————————②
```

Рис. 19.12. Выполнение многострочного запроса

```
    /* Цикл по всем строкам результатов запроса */
    for (;;)
    {
        /* Выборка очередной строки результатов запроса */
        exec sql fetch repcurs  <------------------------------③
            into :repname, :repquota :repquota_ind,
                 :repsales;

        /*Вывод полученных данных */
        printf("Имя: %s\n", repname);
        if (repquota_ind < 0)
            printf("План продаж - NULL\n");
        else
            printf("План продаж: %f\n", repquota);
        printf("Продажи: %f\n", repsales);
    }
error:
    printf("Ошибка SQL: %ld\n", sqlca.sqlcode);
    exit();
done:
    /* Запрос завершен; закрытие курсора */
    exec sql close repcurs; <------------------------------④
    exit();
}
```

Окончание рис. 19.12

В связи с относительной простотой интерфейса, применяемого в SQL Server, и его сходством с интерактивным SQL многие программисты считают, что API в SQL Server более удобен для обработки запросов, чем встроенный SQL.

Выборка значений NULL

Функции `dbnextrow()` и `dbbind()`, представленные на рис. 19.10, обеспечивают простой способ получения результатов запроса, но они не позволяют извлекать значения `NULL`. Если строка, извлекаемая функцией `dbnextrow()`, содержит столбец со значением `NULL`, SQL Server подставляет вместо него замещающее значение. По умолчанию замещающим значением в SQL Server для числовых типов данных является нуль, для строк фиксированной длины — строка пробелов, а для строк переменной длины — пустая строка. Приложение может изменить принимаемое по умолчанию замещающее значение для любого типа данных с помощью API-функции `dbsetnull()`.

Если в рассматриваемой программе на рис. 19.10 одна из строк таблицы содержит значение `NULL` в столбце `QUOTA`, функция `dbnextrow()` для этой строки запишет нуль в переменную `repquota`. Обратите внимание на то, что по извлекаемым данным программа не может определить, действительно ли столбец `QUOTA` содержит нуль или в нем находится значение `NULL`. В одних приложениях применение замещающих значений вполне приемлемо, но в других возможность обнаруживать значения `NULL` является очень важной. В таком случае следует использовать альтернативную схему извлечения результатов запроса, описанную в следующем разделе.

Извлечение результатов запроса с помощью указателей

При стандартном методе извлечения данных из SQL Server функция dbnextrow() копирует содержимое каждого столбца в одну из переменных программы. Если в таблице результатов запроса имеется много строк или много длинных столбцов с текстовыми данными, процесс копирования может занять много времени. Кроме того, функция dbnextrow() не имеет механизма передачи в программу значений NULL.

1. Для решения двух указанных проблем в библиотеке dblib применяется альтернативный метод извлечения результатов запроса. На рис. 19.13 приведена та же программа, что и на рис. 19.10, но переписанная с использованием альтернативного метода.
2. Программа посылает запрос в SQL Server и использует функцию dbresults() для доступа к результатам запроса (как и в случае с любой инструкцией SQL). Однако программа *не* вызывает функцию dbbind() для привязки столбцов к программным переменным.
3. Программа вызывает функцию dbnextrow() и последовательно перемещается по таблице результатов запроса от одной строки к другой.
4. Для каждого столбца каждой строки программа вызывает функцию dbdata(), чтобы получить *указатель* на значение очередной ячейки. Указатель адресует местоположение содержимого ячейки во внутренних областях памяти библиотеки dblib.
5. Если столбец содержит данные переменной длины (тип VARCHAR), программа вызывает функцию dbdatlen(), чтобы узнать длину элемента данных.
6. Если столбец содержит значение NULL, функция dbdata() возвращает нулевой указатель (0), а функция dbdatlen() возвращает нулевую длину элемента данных. Это позволяет программе обнаружить значения NULL и отреагировать надлежащим образом.

```
main()
{
    LOGINREC  *loginrec; /* Структура данных учетной записи  */
    DBPROCESS *dbproc;   /* Структура данных для подключения */
    char      *namep;    /* Указатель на данные столбца NAME */
    int        namelen;  /* Длина данных столбца NAME        */
    float     *quotap;   /* Указатель на данные столбца QUOTA*/
    float     *salesp;   /* Длина данных столбца SALES       */
    char      *namebuf;  /* Буфер для имени                  */

    /* Создание учетной записи и подключение к SQL Server    */
    loginrec = dblogin();
    DBSETLUSER( loginrec, "scott" );
    DBSETLPWD ( loginrec, "tiger" );
    dbproc = dbopen( loginrec, "" );

    /* Передача запроса SQL Server для выполнения            */
    dbcmd(dbproc,"select name, quota, sales from salesreps ");
    dbcmd(dbproc,"where sales > quota order by name");
    dbsqlexec(dbproc);
```

Рис. 19.13. Выборка данных с использованием функции dbdata()

```
    /* Получение первой инструкции пакета                    */
    dbresults(dbproc);                                        ①

    /* Цикл получения строк результатов запроса              */
    while(status = dbnextrow(dbproc) == SUCCEED)              ②
    {
        /* Получение адреса каждого элемента строки          */
        namep   = dbdata(dbproc,1);                           ③
        quotap  = dbdata(dbproc,2);                           ③
        salesp  = dbdata(dbproc,3);                           ③
        namelen = dbdatlen(dbproc,1);                         ④

        /*Копируем значение NAME в собственный буфер
         и завершаем его нулевым символом                    */
        strncpy(namebuf,namep,namelen);
        *(namebuf + namelen) = (char)0;

        /* Вывод данных о служащем                           */
        printf("Имя:%s\n",namebuf);
        if(quotap == 0)                                       ⑤
            printf("План продаж имеет значение NULL.\n");
        else
            printf("План продаж:%f\n",*quotap);
        printf("Продажи:%f\n",*salesp);
    }

    /* Проверка на ошибки и закрытие соединения              */
    if (status == FAIL)
    {
        printf("Ошибка SQL.\n");
        dbexit(dbproc);
        exit();
    }
}
```

Окончание рис. 19.13

Программа, представленная на рис. 19.13, более громоздка по сравнению с программой, текст которой приведен на рис. 19.10. В целом можно сказать, что если в программе не требуется различать значения NULL или обрабатывать результаты запроса большого объема, то лучше использовать функцию dbbind(), а не функцию dbdata().

Выборка строк в произвольном порядке

Обычно программа обрабатывает результаты запроса в SQL Server, последовательно извлекая строки с помощью функции dbnextrow(). Для приложений, связанных с просмотром баз данных, библиотека dblib обеспечивает возможность произвольного доступа к строкам таблицы результатов запроса. Программа должна явно разрешить произвольный доступ к строкам, устанавливая соответствующий режим в библиотеке dblib. После этого можно использовать функцию dbgetrow() для извлечения строк по их номерам.

Чтобы обеспечить выборку строк в произвольном порядке, библиотека dblib хранит строки таблицы результатов запроса во внутреннем буфере. Если в буфере

помещаются полностью все строки таблицы, функция dbgetrow() обеспечивает произвольный доступ к любой строке. Если размер таблицы превышает размер буфера, то в буфере запоминаются только первые строки. К этим строкам программа имеет произвольный доступ, но функция dbnextrow(), которая попытается извлечь строку, не вошедшую в буфер, возвратит специальный код ошибки BUF_FULL. Тогда программа должна удалить из буфера часть строк посредством функции dbclrbuf(), чтобы освободить место для новых строк. Функция dbgetrow() не может повторно извлечь строки, удаленные из буфера. Таким образом, как показано на рис. 19.14, библиотека dblib обеспечивает произвольный доступ к таблице результатов запроса в пределах ограниченного окна, размер которого определяется размером буфера строк. Вызывая функцию dbsetopt(), программа может задать размер буфера строк.

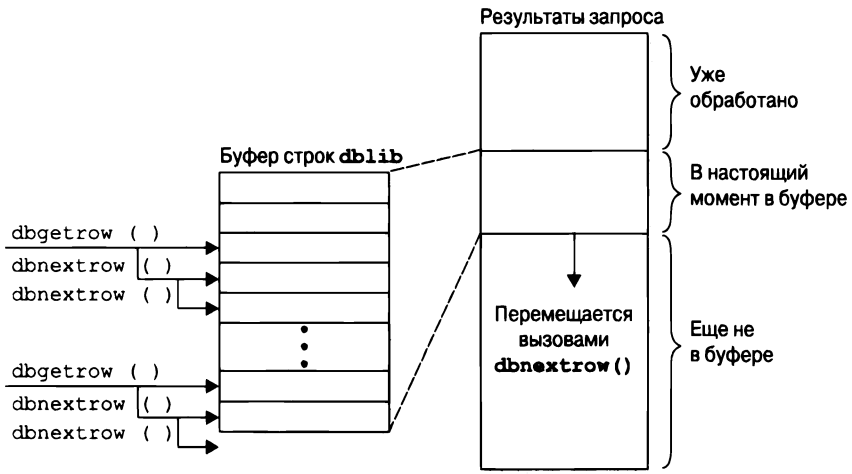


Рис. 19.14. Выборка строк в произвольном порядке в dblib

Произвольный доступ, обеспечиваемый функцией dbgetrow(), аналогичен курсорам с произвольным доступом, имеющимся в ряде СУБД и официально утвержденным в стандарте SQL. В обоих случаях реализуется произвольная выборка строк по их номерам. Однако курсор с произвольным доступом является настоящим указателем на всю таблицу результатов запроса от первой до последней строки, даже если она содержит тысячи строк. Функция же dbgetrow() обеспечивает произвольный доступ только в пределах ограниченного окна. Для небольших приложений этого вполне достаточно, но реализация произвольного доступа для больших запросов является непростой задачей.

Позиционные обновления

В программе со встроенным SQL курсор обеспечивает непосредственную и достаточно тесную связь между программой и СУБД, обрабатывающей запрос на выборку. Извлекая строку за строкой с помощью инструкции FETCH, программа каждый раз обращается к СУБД. Если запрос однотабличный, то СУБД может установить прямое соответствие между текущей строкой таблицы результатов запроса и

соответствующей строкой в базе данных. В этом случае с помощью позиционных инструкций (UPDATE ... WHERE CURRENT OF и DELETE ... WHERE CURRENT OF) программа может модифицировать или удалить текущую строку в таблице результатов запроса.

При обработке запроса в SQL Server связь между программой и СУБД является асинхронной и гораздо менее сильной. В ответ на получение пакета инструкций, содержащего одну или несколько инструкций SELECT, SQL Server посылает результаты запроса обратно в библиотеку dblib, которая их обрабатывает. Построчная выборка результатов запроса осуществляется с помощью функций библиотеки dblib, а не посредством инструкций SQL. В результате этого в ранних версиях SQL Server нельзя было выполнять позиционные обновления, так как понятие текущей строки относилось к результатам запроса в библиотеке, а не к строкам реальных таблиц базы данных.

В более поздних версиях SQL Server (и Sybase) была добавлена полная поддержка стандартных курсоров и связанных с ними инструкций DECLARE/OPEN/FETCH/CLOSE. Управление курсорами осуществляется внутри хранимых процедур, написанных на диалекте Transact-SQL. Действие инструкции FETCH заключается в том, чтобы передавать полученные данные в хранимую процедуру, а не в приложение, вызывающее эту процедуру. Хранимые процедуры и операции с ними в различных СУБД рассматриваются в главе 20, "Хранимые процедуры SQL".

Динамические запросы на выборку

Во всех примерах программ, рассмотренных ранее в настоящей главе, выполнялись запросы, структура которых была известна заранее. Столбцы в таблице результатов запроса можно было связывать с программными переменными посредством вызова функции dbbind(). Эту методику можно использовать для написания большинства программ, работающих с SQL Server. (Такая статическая привязка столбцов соответствует фиксированному списку базовых переменных, который применяется в статической инструкции FETCH, рассмотренной в главе 17, "Встроенный SQL".)

Если во время написания программы известна не вся информация, необходимая для выполнения запроса, то программа не может содержать вызовы функции dbbind(). Вместо этого программа должна с помощью специальных API-функций запросить у библиотеки dblib описание всех столбцов в таблице результатов запроса. После этого программа может на лету связать столбцы с буферами, выделяемыми для этой цели программой на этапе выполнения. (Такая динамическая привязка столбцов соответствует использованию динамической инструкции DESCRIBE и области SQLDA, рассматривавшихся в главе 18, "Динамический SQL".)

На рис. 19.15 приведен текст интерактивной программы формирования запросов, иллюстрирующей методику выполнения динамических запросов с помощью библиотеки dblib. Программа принимает имя таблицы, введенное пользователем, и предлагает ему выбрать столбцы, которые будут извлекаться из таблицы. Когда он это сделает, программа сформирует инструкцию SELECT, а затем описанным ниже способом выполнит эту инструкцию и отобразит на экране данные из выбранных столбцов.

1. Программа, как обычно, передает в SQL Server сформированную инструкцию SELECT с помощью функции dbcmd(), запрашивает ее выполнение посредством функции dbsqlexec() и вызывает функцию dbresults(), чтобы узнать, успешно ли была выполнена инструкция.
2. Программа вызывает функцию dbnumcols(), чтобы узнать, сколько столбцов содержит таблица результатов запроса, извлеченная инструкцией SELECT.
3. Для каждого столбца программа вызывает функцию dbcolname(), чтобы узнать имя столбца, и функцию dbcoltype(), чтобы узнать его тип данных.
4. Для приема каждого столбца программа выделяет буфер и вызывает функцию dbbind(), чтобы связать каждый столбец со своим буфером.
5. После привязки всех столбцов программа многократно вызывает функцию dbnextrow() для извлечения всех строк из таблицы результатов запроса.

```
main()
{
    /* Это простая программа общего назначения для
       выполнения запросов. Она предлагает пользователю
       ввести имя таблицы, после чего запрашивает, какие
       столбцы должны быть включены в запрос. По
       завершении выбора программа выполняет запрос и
       выводит его результаты.                          */
    LOGINREC  *loginrec; /* Учетная запись              */
    DBPROCESS *dbproc;   /* Данные для подключения к СУБД */
    char stmbuf[ 2001 ]; /* SQL-текст для выполнения    */
    char querytbl[ 32 ]; /* Выбранная таблица           */
    char querycol[ 32 ]; /* Выбранный столбец           */
    int  status;         /* Код состояния dblib         */
    int  first_col = 0;  /* Это первый выбранный столбец? */
    int  colcount;       /* Количество столбцов
                            в результатах запроса       */
    int  i;              /* Индекс столбца              */
    char inbuf[ 101 ];   /* Пользовательский ввод       */
    char*item_name[ 100 ]; /* Массив для имен столбцов  */
    char*item_data[ 100 ]; /* Массив буферов столбцов   */
    int  item_type[ 100 ]; /* Массив типов данных столбцов*/
    char*address;        /* Адрес буфера текущего столбца */
    int  length;         /* Длина буфера текущего столбца */

    /* Подключение к SQL Server                         */
    loginrec = dblogin();
    DBSETLUSER( loginrec, "scott" );
    DBSETLPWD( loginrec, "tiger" );
    dbproc = dbopen( loginrec, "" );

    /* Приглашение пользователю ввести имя таблицы      */
    printf( "*** Мини-программа для запросов ***\n" );
    printf( "Введите имя запрашиваемой таблицы:" );
    gets( querytbl );

    /* Создание запроса в буфере                        */
```

Рис. 19.15. Использование dblib для динамических запросов

```
strcpy( stmbuf, "select" );

/* Запрос SQL Server об именах столбцов                    */
dbcmd( dbproc, "select name from syscolumns " );
dbcmd( dbproc, "where id=(select id from sysobjects ");
dbcmd( dbproc, "where type = 'U' and name = " );
dbcmd( dbproc, querytbl );
dbcmd( dbproc, ")" );
dbsqlexec( dbproc );

/* Обработка результатов запроса                           */
dbresults( dbproc );
dbbind( dbproc, querycol );

while ( status = dbnextrow( dbproc ) == SUCCEED )
{
    printf( "Включить столбец %s (y/n)?", querycol );
    gets( inbuf );
    if ( inbuf[ 0 ] == 'y' )
    {
        /* Пользователь добавляет этот столбец             */
        if ( first_col++ > 0 ) strcat( stmbuf, "," );
        strcat( stmbuf, querycol );
    }
}

/* Завершаем инструкцию SELECT предложением FROM           */
strcat( stmbuf, "from" );
strcat( stmbuf, querytbl );

/* Выполнение запроса и переход к результатам              */
dbcmd( dbproc, stmbuf );        <------------------------①
dbsqlexec( dbproc );            <------------------------①
dbresults( dbproc );            <------------------------①

/* Запрос у dblib описания каждого столбца,
   выделение памяти и связывание                           */
colcount = dbnumcols( dbproc ); <------------------------②
for ( i = 0;i < colcount;i++ )
{
    item_name[ i ] = dbcolname( dbproc, i );    <--------③
    type = dbcoltype( dbproc, i );              <--------③
    switch ( type )
    {
    case SQLCHAR:
    case SQLTEXT:
    case SQLDATETIME:
        length = dbcollen( dbproc, i ) + 1;
        item_data[ i ] = address = malloc( length ); <-④
        item_type[ i ] = NTBSTRINGBIND;
        dbbind( dbproc, i, NTBSTRINGBIND,       <------④
                length, address );
        break;
    case SQLINT1:
    case SQLINT2:
    case SQLINT4:
        item_data[ i ] = address = malloc(sizeof(long));
        item_type[ i ] = INTBIND;
```

Продолжение рис. 19.15

```
            dbbind( dbproc, i, INTBIND,
                    sizeof( long ), address );
            break;
        case SQLFLT8:
        case SQLMONEY:
            item_data[ i ] = address
                           = malloc( sizeof( double ) );
            item_type[ i ] = FLT8BIND;
            dbbind( dbproc, i, FLT8BIND,
                    sizeof( double ), address );
            break;
        }
    }

    /* Выборка и вывод строк результатов запроса                    */
    while ( status = dbnextrow( dbproc ) == SUCCEED )  <----- 5
    {
        /* Цикл для вывода столбцов строки                          */
        printf( "\n" );
        for ( i = 0;i < colcount;i++ )
        {
            /* Вывод метки данного столбца                          */
            printf( "Столбец #%d (%s):", i + 1,
                    item_name[ i ]);
            /* Отдельная обработка каждого типа данных              */
            switch ( item_type[ i ] )
            {
            case NTBSTRINGBIND: /* Текст - просто выводим */
                puts( item_data[ i ] );
                break;
            case INTBIND: /* Четырехбайтовое целое                  */
                printf( "%lf",
                        *((double*)(item_data[ i ])));
                break;
            case FLT8BIND: /* Число с плавающей точкой              */
                printf( "%lf",
                        *((double*)(item_data[ i ])));
                break;
            }
        }
    }

    printf( "\nКонец данных.\n" );

    /* Освобождение памяти                                          */
    for ( i = 0;i < colcount;i++ )
    {
        free( item_data[ i ] );
    }
    dbexit( dbproc );

    exit();
}
```

Окончание рис. 19.15

Программа для SQL Server (рис. 19.15) выполняет ту же задачу, что и программа с динамическим SQL на рис. 18.9 (воспроизведена на рис. 19.16). Будет полезно сравнить эти программы и применяемые в них методы.

- Как в случае применения встроенного SQL, так и при использовании библиотеки dblib программа формирует инструкцию SELECT в своих буферах и передает ее в СУБД для обработки. В динамическом SQL эту задачу выполняет специальная инструкция PREPARE; в SQL Server используются функции dbcmd() и dbsqlexec().
- В обоих случаях программа должна запрашивать у СУБД описание столбцов таблицы результатов запроса. В динамическом SQL это делает специальная инструкция DESCRIBE, а описание возвращается в область SQLDA. В случае библиотеки dblib описание возвращается в программу посредством вызовов API-функций. Обратите внимание на то, что программа, приведенная на рис. 19.15, имеет свои собственные массивы для отслеживания информации о столбцах.
- В обоих случаях программа должна выделять буферы для приема результатов запроса и связывать столбцы с этими буферами. В динамическом SQL программа осуществляет привязку столбцов, помещая адреса буферов в структуры SQLVAR области SQLDA. В SQL Server для привязки столбцов программа использует функцию dbbind().
- В обоих случаях результаты запроса возвращаются в буферы программы строка за строкой. В динамическом SQL программа извлекает строки из таблицы результатов запроса с помощью специальной разновидности инструкции FETCH, в которой задана область SQLDA. В SQL Server программа извлекает строки, вызывая функцию dbnextrow().

```
main()
{
    /* Это простая программа общего назначения. Она
       запрашивает у пользователя имя таблицы, а затем -
       какие столбцы таблицы должны быть добавлены в
       запрос. После завершения диалога с пользователем
       программа выполняет запрос и выводит его результаты
       на экран.                                          */
    exec sql include sqlca;
    exec sql include sqlda;
    exec sql begin declare section;
        char stmtbuf[2001]; /* Текст SQL-запроса          */
        char querytbl[32];  /* Пользовательская таблица   */
        char querycol[32];  /* Пользовательский столбец   */
    exec sql end declare section;

    /* Курсор для получения имен столбцов
       из системного каталога                             */
    exec sql declare tblcurs cursor for
              select colname
                from system.syscolumns
               where tblname = :querytbl and owner = user;
    exec sql declare qrycurs cursor for querystmt; <------①

    /* Структуры данных программы */
    int    colcount = 0;    /* Количество столбцов        */
    struct sqlda *qry_da;   /* Область SQLDA запроса      */
    struct sqlvar *qry_var; /* SQLVAR текущего столбца    */
```

Рис. 19.16. Выборка данных с помощью динамического SQL

```
    int     i;                  /* Индекс SQLVAR в SQLDA        */
    char    inbuf[101];         /* Ввод пользователя            */

    /* Приглашение пользователю указать таблицу                  */
    printf("*** Программа с мини-запросом ***\n\n");
    printf("Введите имя таблицы: ");
    gets(querytbl);

    /* Создание инструкции SELECT в буфере                       */
    strcpy(stmtbuf, "select ");  <------------------------------②

    /* Настройка обработки ошибок                                */
    exec sql whenever sqlerror goto handle_error;
    exec sql whenever not found goto no_more_columns;

    /* Запрос системного каталога об именах столбцов             */
    exec sql open tblcurs;
    for ( ; ; )
    {
        /* Получение имени столбца и запрос пользователя         */
        exec sql fetch tblcurs into :querycol;
        printf("Включить столбец %s (y/n)? ", querycol);
        gets(inbuf);
        if (inbuf[0] == 'y')
        {
            /* Пользователь просит включить данный столбец;
               добавляем его к списку                            */
            if (colcount++ > 0)
                strcat(stmtbuf, ", ");
            strcat(stmtbuf, querycol);  <---------------------②
        }
    }

no_more_columns:
    exec sql close tblcurs;

    /* Завершаем инструкцию SELECT предложением FROM             */
    strcat(stmtbuf, "from ");
    strcat(stmtbuf, querytbl);

    /* Выделение SQLDA для динамического запроса */
    query_da = (SQLDA *)malloc(sizeof(SQLDA) +
                               colcount * sizeof(SQLVAR));
    query_da->sqln = colcount;

    /* Подготавливаем запрос
       и передаем его СУБД для описания                          */
    exec sql prepare querystmt from :stmtbuf;  <----------------③
    exec sql describe querystmt into qry_da;   <----------------④

    /* Цикл по всем SQLVAR,
       с выделением памяти для столбцов                          */
    for (i = 0; i < colcount; i++)
    {
        qry_var = qry_da->sqlvar + i;
        qry_var->sqldat = malloc(qry_var->sqllen);  <----------⑤
        qry_var->sqlind = malloc(sizeof(short));
    }
```

Продолжение рис. 19.16

```
    /* SQLDA создана; запрос и получение результатов        */
    exec sql open qrycurs;  <------------------------------⑥
    exec sql whenever not found goto no_more_data;
    for ( ; ; )
    {
        /* Получение строки результатов в наши буфера       */
        exec sql fetch sqlcurs using descriptor qry_da; <---⑦
        printf("\n");
        /* Цикл вывода данных для каждого столбца строки    */
        for (i = 0; i < colcount; i++)
        {
            /* Поиск SQLVAR для данного столбца;
               вывод метки столбца                          */
            qry_var = qry_da->sqlvar + i;
            printf(" Столбец # %d (%s): ",
                   i+1, qry_var->sqlname);
            /* Проверка переменной-индикатора               */
            if (*(qry_var -> sqlind) != 0)
            {
                puts("равен NULL!\n");
                continue;
            }
            /* Получение данных и обработка типа            */
            switch (qry_var -> sqltype) {
            case 448:
            case 449:
                /* VARCHAR -- просто выводим                */
                puts(qry_var -> sqldata);
                break;
            case 496:
            case 497:
                /* Четырехбайтовое целое -
                   преобразуем и выводим                    */
                printf("%ld", *((int *) (qry_var->sqldata)));
                break;
            case 500:
            case 501:
                /* Двухбайтовое целое -
                   преобразуем и выводим                    */
                printf("%d", *((short *)(qry_var->sqldata)));
                break;
            case 480:
            case 481:
                /* Данные с плавающей точкой -
                   преобразуем и выводим                    */
                printf("%lf",
                       *((double *)(qry_var->sqldat)));
                break;
            }
        }
    }
no_more_data:
    printf("\nКонец данных.\n");

    /* Освобождение выделенной памяти                       */
    for (i = 0; i < colcount; i++)
    {
        qry_var = qry_da->sqlvar + i;
        free(qry_var->sqldata);
```

Продолжение рис. 19.16

```
            free(qry_var->sqlind);
        }
        free(qry_da);
        close qrycurs;  <------------------------------------------⑧
        exit();
    }
```

Окончание рис. 19.16

В целом методика выполнения динамических запросов в обоих случаях очень похожа. Однако в динамическом SQL применяются специальные инструкции и структуры данных, совершенно не похожие на те, что используются для выполнения запросов в статическом SQL. А в SQL Server методика выполнения динамических запросов в основном та же, что и методика выполнения всех остальных запросов. Единственная особенность заключается в том, что добавляются функции библиотеки `dblib`, возвращающие информацию о столбцах в таблице результатов запроса. Это облегчает понимание подхода с применением API программистам, имеющим незначительный опыт работы с SQL.

ODBC API и стандарт SQL/CLI

ODBC (Open Database Connectivity, открытый доступ к базам данных) — это разработанный компанией Microsoft универсальный API для доступа к базам данных. Хотя в современном компьютерном мире Microsoft играет важную роль как производитель программного обеспечения для баз данных, все же в первую очередь она является одним из ведущих производителей операционных систем, и именно это послужило мотивом создания ODBC: Microsoft захотела облегчить разработчикам приложений Windows доступ к базам данных. Все дело в том, что различные СУБД существенно отличаются друг от друга, так же как и их API. Если разработчику нужно было написать приложение, работающее с базами данных нескольких СУБД, для каждой из них приходилось писать отдельный интерфейсный модуль (обычно называемый *драйвером*). Чтобы избавить программистов от выполнения рутинной и достаточно сложной работы, Microsoft решила на уровне операционной системы стандартизировать интерфейс взаимодействия между приложениями и СУБД, благодаря чему во всех программах мог бы использоваться один и тот же универсальный набор функций, поддерживаемый всеми производителями СУБД. Таким образом, от внедрения ODBC выиграли и разработчики приложений, и производители СУБД, для которых также решалась извечная проблема совместимости.

Стандартизация CLI

Даже если бы ODBC API был всего лишь собственным стандартом компании Microsoft, его значение все равно было бы очень велико. Однако Microsoft постаралась сделать его независимым от конкретной СУБД. Одновременно ассоциация производителей СУБД (SQL Access Group) работала над стандартизацией протоколов удаленного доступа к базам данных в архитектуре "клиент/сервер".

Microsoft убедила ассоциацию принять ODBC в качестве независимого стандарта доступа к базам данных. В дальнейшем этот стандарт перешел в ведение другой организации, Европейского консорциума X/Open, и был включен в ее комплект стандартов CAE (Common Application Environment — единая прикладная среда).

С ростом популярности программных интерфейсов доступа к базам данных организации, ответственные за принятие официальных стандартов, стали уделять этому аспекту SQL все более пристальное внимание. На основе стандарта X/Open (базировавшегося на исходной версии ODBC, разработанной компанией Microsoft) с небольшими модификациями был разработан официальный стандарт ANSI/ISO. Этот стандарт, известный как SQL/CLI (SQL/Call Level Interface — интерфейс уровня вызовов функций), был опубликован в 1995 году под названием ANSI/ISO/IEC 9075-3-1995. С небольшими модификациями SQL/CLI стал частью 3 стандарта SQL:1999 и благополучно прошел через все обновления, сохранившись во всех последующих версиях стандарта ANSI/ISO.

Microsoft привела ODBC в соответствие стандарту SQL/CLI. Стандарт CLI грубо образует ядро Microsoft ODBC 3. Однако полный набор высокоуровневых функций ODBC 3 выходит далеко за рамки спецификации CLI: он предоставляет разработчикам приложений гораздо более широкие возможности и решает ряд специфических задач, связанных с использованием ODBC как части операционной системы Windows. На практике возможности ядра ODBC и спецификация SQL/CLI вместе образуют эффективный стандарт API.

Преимущества связки ODBC/CLI как для разработчиков приложений, так и для производителей СУБД были настолько очевидны, что оба стандарта очень быстро получили самую широкую поддержку. Практически все производители СУБД на базе SQL включили в свои продукты поддержку соответствующих интерфейсов. Некоторые СУБД даже используют ODBC/CLI в качестве стандартного API для доступа к базе данных. ODBC и CLI поддерживаются тысячами приложений, включая ведущие пакеты инструментальных средств разработки, программ для работы с запросами и формами и программы генерации отчетов, а также такие популярные программы, как электронные таблицы и программы для построения диаграмм и графиков.

Стандарт SQL/CLI содержит около сорока функций, приведенных в табл. 19.2. Они служат для подключения к серверу баз данных, выполнения инструкций SQL, получения и обработки результатов запросов, а также обработки ошибок, происшедших в ходе выполнения инструкций. Эти функции обеспечивают полный набор возможностей, предоставляемых встроенным SQL, включая как статический, так и динамический SQL.

Таблица 19.2. Функции SQL/CLI API

Функция	Описание
Управление ресурсами и подключением к базе данных	
`SQLAllocHandle()`	Выделяет ресурсы для среды, подключения, описания или инструкции
`SQLFreeHandle()`	Освобождает ранее выделенные ресурсы
`SQLAllocEnv()`	Выделяет ресурсы для среды SQL

Продолжение табл. 19.2

Функция	Описание
SQLFreeEnv()	Освобождает ресурсы, выделенные для среды SQL
SQLAllocConnect()	Выделяет ресурсы для подключения к базе данных
SQLFreeConnect()	Освобождает ресурсы, выделенные для подключения к базе данных
SQLAllocStmt()	Выделяет ресурсы для инструкции SQL
SQLFreeStmt()	Освобождает ресурсы, выделенные для инструкции SQL
SQLConnect()	Устанавливает соединение с базой данных
SQLDisconnect()	Разрывает соединение с базой данных
Выполнение инструкций SQL	
SQLExecDirect()	Непосредственно выполняет инструкцию SQL
SQLPrepare()	Подготавливает инструкцию SQL к последующему выполнению
SQLExecute()	Выполняет ранее подготовленную инструкцию SQL
SQLRowCount()	Возвращает количество строк, обработанных последней инструкцией SQL
Управление транзакциями	
SQLEndTran()	Завершает транзакцию
SQLCancel()	Отменяет выполнение инструкции SQL
Обработка параметров	
SQLBindParam()	Связывает программный буфер со значением параметра
SQLParamData()	Обрабатывает отложенные значения параметров
SQLPutData()	Предоставляет отложенные значения параметров или часть строкового значения
Обработка результатов запроса	
SQLSetCursorName()	Назначает имя курсору
SQLGetCursorName()	Возвращает имя курсора
SQLFetch()	Возвращает строку из таблицы результатов запроса
SQLFetchScroll()	Возвращает указанную строку из таблицы результатов запроса
SQLCloseCursor()	Закрывает курсор
SQLGetData()	Возвращает значение столбца из таблицы результатов запроса
Описание результатов запроса	
SQLNumResultCols()	Возвращает количество столбцов в таблице результатов запроса
SQLDescribeCol()	Возвращает описание указанного столбца в таблице результатов запроса
SQLColAttribute()	Возвращает атрибут столбца в таблице результатов запроса
SQLGetDescField()	Возвращает значение поля дескриптора
SQLSetDescField()	Устанавливает значение поля дескриптора

Окончание табл. 19.2

Функция	Описание
`SQLGetDescRec()`	Возвращает значения из записи дескриптора
`SQLSetDescRec()`	Устанавливает значения в записи дескриптора
`SQLCopyDesc()`	Копирует области значений дескрипторов
Обработка ошибок	
`SQLError()`	Возвращает информацию об ошибке
`SQLGetDiagField()`	Возвращает значение поля записи диагностики
`SQLGetDiagRec()`	Возвращает значение записи диагностики
Управление атрибутами	
`SQLSetEnvAttr()`	Устанавливает значение атрибута среды SQL
`SQLGetEnvAttr()`	Возвращает значение атрибута среды SQL
`SQLSetStmtAttr()`	Устанавливает область дескриптора, используемую инструкцией SQL
`SQLGetStmtAttr()`	Возвращает область дескриптора инструкции SQL
Управление драйвером	
`SQLDataSources()`	Возвращает список доступных серверов баз данных
`SQLGetFunctions()`	Возвращает информацию о функциях, поддерживаемых текущей реализацией SQL
`SQLGetInfo()`	Возвращает информацию о возможностях, поддерживаемых текущей реализацией SQL

Простейшая CLI-программа, приведенная на рис. 19.17, повторяет программы, представленные на рис. 19.3 и 19.9, но в ней используются функции CLI. Реализованная в ней последовательность действий является основой большинства приложений CLI.

1. Программа подключается к библиотеке CLI и выделяет память под структуры данных, используемые функциями этой библиотеки.
2. Программа подключается к конкретному серверу баз данных.
3. Программа формирует инструкции SQL в собственных буферах памяти.
4. Программа вызывает функции CLI, с помощью которых она просит сервер выполнить инструкции SQL и узнает о завершении этих инструкций.
5. В случае успешного выполнения инструкций SQL программа с помощью еще одной функции CLI просит сервер завершить транзакцию.
6. Программа отключается от сервера баз данных и освобождает память, занимаемую структурами данных.

```
/* Программа изменяет все плановые объемы продаж
   на заданную пользователем сумму                                  */
#include <sqlcli.h>  /* Заголовочный файл с объявлениями
                        структур и функций CLI                     */

main()
{
    SQLHENV    env_hdl;            /* Дескриптор среды SQL     */
    SQLHDBC    conn_hdl;           /* Дескриптор подключения   */
    SQLHSTMT   stmt_hdl;           /* Дескриптор инструкции    */
    SQLRETURN  status;             /* Код возврата вызова CLI  */
    char      *svr_name = "demo";  /* Имя сервера              */
    char      *user_name = "joe";  /* Имя пользователя и       */
    char      *user_pswd = "xyz";  /* пароль для подключения   */
    char       amount_str[31];     /* Введенная пользователем
                                      сумма                    */
    char       stmt_buf[128];      /* Буфер инструкции SQL     */

    /* Выделяем память для среды SQL, подключения
       и инструкции и получаем их дескрипторы */
    SQLAllocHandle(SQL_HANDLE_ENV,
                   SQL_NULL_HANDLE,&env_hdl);        ←———— ①
    SQLAllocHandle(SQL_HANDLE_DBC,
                   env_hdl, &conn_hdl);              ←———— ①
    SQLAllocHandle(SQL_HANDLE_STMT,
                   conn_hdl, &stmt_hdl);             ←———— ①

    /* Подключаемся к базе данных, указав имя сервера,
       имя пользователя и пароль.
       Спецификатор SQL_NTS говорит о том, что мы передаем
       вместо длины строку с завершающим нулевым символом.   */
    SQLConnect(conn_hdl,  svr_name, SQL_NTS,             ←———— ②
               user_name, SQL_NTS,
               user_pswd, SQL_NTS);

    /* Запрашиваем у пользователя сумму, на которую нужно
       увеличить или уменьшить плановый объем продаж         */
    printf("Изменить объем продаж на: ");
    gets(amount_str);

    /* Формируем инструкцию UPDATE и просим СУБД ее выполнить*/
    strcpy(stmt_buf,
           "update salesreps set quota = quota + ");     ←———— ③
    strcat(stmt_buf, amount_str);                        ←———— ③
    status = SQLExecDirect(stmt_hdl, stmt_buf, SQL_NTS);  ←—— ④
    if (status)
    {
        SQLEndTran(SQL_HANDLE_ENV, env_hdl, SQL_ROLLBACK);
        printf("Ошибка при обновлении.\n");
    }
    else
    {
        SQLEndTran(SQL_HANDLE_ENV, env_hdl, SQL_COMMIT);  ←—— ⑤
        printf("Обновление завершено.\n");
    }

    /* Отключаемся от сервера                                */
    SQLDisconnect(conn_hdl);  ←————————————————————— ⑥
```

Рис. 19.17. Простая программа, использующая SQL/CLI

```
/* Освобождаем дескрипторы и выходим из программы          */
    SQLFreeHandle(SQL_HANDLE_STMT, stmt_hdl);          ←———⑥
    SQLFreeHandle(SQL_HANDLE_DBC,  conn_hdl);          ←———⑥
    SQLFreeHandle(SQL_HANDLE_ENV,  env_hdl);           ←———⑥
    exit();
}
```

Окончание рис. 19.17

Все функции CLI возвращают код состояния, указывающий, успешно ли функция выполнила свою работу. В случае каких-либо проблем это значение представляет собой код ошибки или предупреждения. Все возможные коды состояния, возвращаемые функциями CLI, описаны в табл. 19.3. В некоторых примерах программ, приведенных в этой главе, возвращаемые функциями CLI коды не проверяются, поскольку мы хотели сократить примеры и акцентировать ваше внимание на иллюстрируемых ими возможностях. Однако в реальных программах коды состояния следует обязательно проверять, чтобы убедиться, что каждая вызываемая функция CLI выполнена успешно. Как правило, для этих кодов, как и для многих других фиксированных значений (таких, как коды типов данных и идентификаторы инструкций), используются символьные константы, определяемые в файле заголовков, который включается в программу в начале исходного текста.

Таблица 19.3. Коды состояний, возвращаемые функциями CLI

Возвращаемое значение	Описание
0	Инструкция выполнена успешно
1	Инструкция выполнена успешно, но с предупреждением
100	Данные не найдены (при извлечении результатов запроса)
99	Требуются данные (отсутствует параметр динамической инструкции)
-1	Ошибка в ходе выполнения инструкции SQL
-2	Ошибка — функции передан неверный дескриптор

Структуры CLI

Принципы взаимодействия приложения с сервером баз данных посредством CLI основаны на ряде концепций, отраженных в иерархии структур данных этого интерфейса.

- **Среда SQL**. Это самый верхний уровень, на котором осуществляется доступ к базе данных. CLI связывает со средой SQL структуру, предназначенную для отслеживания приложений, работающих в этой среде.
- **Сеанс подключения**. Это логическое подключение к конкретному серверу баз данных. Концептуально CLI позволяет одному приложению одновременно подключаться к нескольким различным серверам. Каждый сеанс подключения имеет собственные структуры данных, которые CLI использует для отслеживания состояния сеанса.

- **Инструкция SQL**. Это отдельная инструкция SQL, обрабатываемая сервером баз данных. Каждая инструкция может обрабатываться в несколько этапов: СУБД ее готовит (компилирует), выполняет, обрабатывает ошибки и, если запрашивались данные, возвращает результаты запроса прикладной программе. Концептуально СУБД может одновременно обрабатывать несколько инструкций одного приложения, параллельно проходя различные описанные стадии обработки. С каждой из инструкций CLI связывает отдельную структуру, которая используется для отслеживания хода обработки инструкции.

Для управления этими концептуальными единицами CLI применяет ту же методику, которая используется большинством современных операционных систем и библиотек. Со средой SQL, с каждым сеансом подключения и с каждой выполняемой инструкцией SQL связывается указатель особого типа, называемый *дескриптором*. Дескриптор идентифицирует область памяти, используемую для хранения данных о конкретном объекте и управляемую непосредственно библиотекой CLI. Практически каждой функции CLI в качестве параметра передается один из таких дескрипторов. На рис. 19.18 представлены объявления функций CLI, предназначенных для управления дескрипторами.

Дескриптор создается с помощью функции `SQLAllocHandle()`. Первый ее параметр говорит CLI о том, дескриптор какого типа вы хотите получить. В последнем параметре приложению возвращается значение созданного дескриптора. Получив дескриптор, приложение может передавать его тем функциям CLI, которые работают в том же самом контексте. Таким образом, различные потоки одной программы или различные параллельно выполняющиеся программы (процессы) могут устанавливать посредством CLI свои собственные сеансы подключения и работать независимо друг от друга. Кроме того, эта технология позволяет одной программе поддерживать несколько сеансов подключения к различным серверам баз данных и параллельно выполнять несколько инструкций SQL. Когда дескриптор программе больше не нужен, она освобождает его с помощью функции `SQLFreeHandle()`.

В дополнение к универсальным функциям `SQLAllocHandle()` и `SQLFreeHandle()`, служащим для создания и освобождения дескрипторов любого типа, в библиотеке CLI определены отдельные функции для создания и освобождения дескрипторов среды, сеанса и инструкции. Эти функции (`SQLAllocEnv()`, `SQLAllocStmt()` и другие) были частью исходного варианта ODBC и по-прежнему поддерживаются текущими реализациями ODBC ради обратной совместимости. Однако Microsoft предупреждает программистов, что в новых программах им лучше пользоваться универсальными функциями, а специализированные функции в будущих версиях этого API могут не поддерживаться. Таким образом, использование универсальных функций гарантирует максимальную переносимость приложений.

```
/* Создает дескриптор для использования в последующих
   вызовах функций CLI                                        */
SQLSMALLINT SQLAllocHandle(
    SQLSMALLINT hdlType,  /* Входной:  код типа дескриптора */
    SQLINTEGER  inHdl,    /* Входной:  дескриптор среды или
                                       соединения            */
    SQLINTEGER *rtnHdl )  /* Выходной: дескриптор            */

/* Освобождает дескриптор, который был создан
   функцией SQLAllocHandle()                                  */
SQLSMALLINT SQLFreeHandle(
    SQLSMALLINT hdlType,  /* Входной:  код типа дескриптора */
    SQLINTEGER  inHdl )   /* Входной:  освобождаемый
                             дескриптор                      */

/* Создает дескриптор для новой среды SQL                     */
SQLSMALLINT SQLAllocEnv(
    SQLINTEGER *envHdl )  /* Выходной: дескриптор среды      */

/* Освобождает дескриптор, который был создан
   функцией SQLAllocEnv()                                     */
SQLSMALLINT SQLFreeEnv(
    SQLINTEGER  envHdl )  /* Входной:  дескриптор среды      */

/* Создает дескриптор для нового сеанса подключения           */
SQLSMALLINT SQLAllocConnect(
    SQLINTEGER  envHdl,   /* Входной:  дескриптор среды      */
    SQLINTEGER *connHdl ) /* Выходной: дескриптор сеанса     */

/* Освобождает дескриптор, который был создан
   функцией SQLAllocConnect()                                 */
SQLSMALLINT SQLFreeConnect(
    SQLINTEGER  connHdl ) /* Входной:  дескриптор сеанса     */

/* Создает дескриптор для новой инструкции SQL                */
SQLSMALLINT SQLAllocStmt(
    SQLINTEGER  envHdl,   /* Входной:  дескриптор среды      */
    SQLINTEGER *stmtHdl ) /* Выходной: дескриптор инструкции*/

/* Освобождает дескриптор, который был создан
   функцией SQLAllocStmt()                                    */
SQLSMALLINT SQLFreeStmt(
    SQLINTEGER  stmtHdl,  /* Входной: дескриптор инструкции */
    SQLINTEGER  option )  /* Входной: параметры, определяющие
                             освобождение буферов и закрытие
                             курсоров                        */
```

Рис. 19.18. Функции CLI для управления дескрипторами

Среда SQL

Среда SQL — это контекст самого высокого уровня, используемый приложением для вызова функций CLI. Если в программе только один поток, ей обычно достаточно одной среды. Если же потоков несколько, может быть создано по одной среде для каждого из них или одна общая среда на всех, в зависимости от архитектуры программы. Концептуально CLI допускает создание в пределах одной среды SQL нескольких сеансов подключения к разным серверам баз данных. Однако некоторые реализации CLI, созданные для конкретных СУБД, могут не допускать существования нескольких одновременных сеансов.

SQL-подключения

Внутри среды SQL программа может установить один или несколько сеансов подключения к базе данных. Сеанс — это соединение между программой и конкретным сервером баз данных, через которое выполняются инструкции SQL. На практике сеанс обычно представляет собой сетевое соединение с сервером баз данных, расположенным на другом компьютере. Однако это может быть и логическое соединение между программой и СУБД, расположенными в одной системе.

На рис. 19.19 приведены объявления функций CLI, используемых для управления сеансами подключения. Чтобы установить соединение, программа сначала создает дескриптор сеанса, вызывая функцию `SQLAllocHandle()` и указывая соответствующий тип дескриптора. Затем программа пытается подключиться к серверу с помощью функции `SQLConnect()`. Если соединение будет установлено успешно, через него будут выполняться инструкции SQL. Дескриптор сеанса передается в качестве параметра всем функциям CLI, связанным с обработкой инструкций SQL, — он указывает, через какое соединение им следует передавать и принимать данные. Когда сеанс больше не требуется, программа отключается от сервера с помощью функции `SQLDisconnect()` и освобождает дескриптор сеанса с помощью функции `SQLFreeHandle()`.

```
/* Устанавливает соединение с сервером баз данных          */
        SQLSMALLINT SQLConnect(
            SQLINTEGER   connHdl,  /* Входной: дескриптор сеанса    */
            SQLCHAR     *svrName,  /* Входной: имя целевого сервера */
            SQLSMALLINT  svrNamLen,/* Входной: длина имени сервера  */
            SQLCHAR     *userName, /* Входной: имя пользователя     */
            SQLSMALLINT  usrNamLen,/* Входной: длина имени          */
            SQLCHAR     *passwd,   /* Входной: пароль               */
            SQLSMALLINT  pswLen)   /* Входной: длина пароля         */

        /* Отключение от SQL-сервера                                */
        SQLSMALLINT SQLDisconnect(
            SQLINTEGER   connHdl)  /* Входной: дескриптор сеанса    */

        /* Возвращает имена доступных SQL-серверов                  */
        SQLSMALLINT SQLDataSources(
            SQLINTEGER   envHdl,   /* Входной : дескриптор среды    */
            SQLSMALLINT  direction,/* Входной : определяет, описание
                                       какого сервера из
                                       списка доступных
                                       следует получить              */
            SQLCHAR     *svrName,  /* Выходной: буфер имени сервера */
            SQLSMALLINT  bufLen,   /* Входной : длина буфера имени  */
            SQLSMALLINT *namLen,   /* Выходной: реальная длина имени*/
            SQLCHAR     *descrip,  /* Выходной: буфер для описания  */
            SQLSMALLINT  buf2Len,  /* Входной : длина буфера        */
            SQLSMALLINT *dscLen)   /* Выходной: реальная длина
                                       описания                      */
```

Рис. 19.19. Функции CLI для управления подключениями

Обычно приложение знает имя сервера баз данных, с которым оно собирается работать (по терминологии, используемой в стандарте, — SQL-сервера). Однако иногда выбор сервера предоставляется пользователю; например, так могут поступать универсальные программы формирования запросов. Такие программы могут воспользоваться функцией `SQLDataSources()`, возвращающей имена известных SQL-серверов, т.е. источников данных, которые могут быть заданы в вызовах функции `SQLConnect()`. Чтобы получить список всех доступных серверов, нужно соответствующее число раз вызвать функцию `SQLDataSources()`. Каждый вызов возвращает описание одного сервера, а когда все серверы перечислены, последний вызов возвращает код ошибки. Второй параметр функции `SQLDataSources()` позволяет изменить последовательность выбора имен серверов.

Обработка инструкций в CLI

Используемая в CLI методика обработки инструкций SQL очень похожа на описанную в предыдущей главе методику обработки инструкций в динамическом SQL. Приложение передает библиотеке CLI инструкцию SQL в текстовом виде как символьную строку. Она может быть выполнена либо сразу, либо в два этапа.

На рис. 19.20 приведены объявления базовых функций, связанных с обработкой инструкций SQL. Процесс их использования таков. Прежде всего приложение должно вызвать функцию `SQLAllocHandle()`, чтобы получить дескриптор инструкции, который идентифицирует выполняемую инструкцию как для программы, так и для CLI. Далее могут следовать вызовы функций `SQLExecDirect()`, `SQLPrepare()` и `SQLExecute()`, и всем им передается дескриптор инструкции. Когда дескриптор больше не нужен, он освобождается с помощью функции `SQLFreeHandle()`.

Для одношагового выполнения инструкции приложение вызывает функцию `SQLExecDirect()`, передавая ей текст инструкции в одном из параметров. СУБД обрабатывает инструкцию и возвращает код ее завершения. Этот процесс был проиллюстрирован в программе на рис. 19.17. Он соответствует одношаговой инструкции `EXECUTE IMMEDIATE` динамического SQL, описанной в главе 18, "Динамический SQL".

Для двухэтапного выполнения инструкции SQL приложение сначала вызывает функцию `SQLPrepare()`, которой также передается текст инструкции. В ответ СУБД анализирует инструкцию, генерирует план ее выполнения и сохраняет его. Немедленного выполнения инструкции не происходит — для этого служат отдельные вызовы функции `SQLExecute()`, которые могут следовать за функцией `SQLPrepare()`. Описанный процесс в точности соответствует двухэтапной схеме выполнения инструкции в динамическом SQL с помощью инструкций `PREPARE` и `EXECUTE`, описанных в предыдущей главе. Он лучше всего подходит для тех инструкций SQL, которые выполняются программой многократно, поскольку позволяет сэкономить время на их анализе и оптимизации, однократно выполняемых в ответ на вызов функции `SQLPrepare()`.

```
/* Непосредственное выполнение инструкции SQL                    */
SQLSMALLINT SQLExecDirect(
   SQLINTEGER  stmtHdl,  /* Входной: дескриптор инструкции  */
   SQLCHAR    *stmtText, /* Входной: текст инструкции       */
   SQLSMALLINT textLen)  /* Входной: длина текста           */

/* Подготовка инструкции SQL                                     */
SQLSMALLINT SQLPrepare(
   SQLINTEGER   stmtHdl, /* Входной: дескриптор инструкции  */
   SQLCHAR     *stmtText,/* Входной: текст инструкции       */
   SQLSMALLINT  textLen) /* Входной: длина текста           */

/* Выполнение подготовленной ранее инструкции SQL                */
SQLSMALLINT SQLExecute(
   SQLINTEGER   stmtHdl) /* Входной: дескриптор инструкции  */

/* Связывает параметр инструкции SQL
   с областью данных программы                                   */
SQLSMALLINT SQLBindParam(
   SQLINTEGER   stmtHdl, /* Входной: дескриптор инструкции  */
   SQLSMALLINT  parmNr,  /* Входной: номер параметра (1,...)*/
   SQLSMALLINT  valType, /* Входной: тип значения           */
   SQLSMALLINT  parmType,/* Входной: тип параметра в SQL    */
   SQLSMALLINT  colSize, /* Входной: размер столбца         */
   SQLSMALLINT  decDigs, /* Входной: количество цифр        */
   void        *value,   /* Входной: указатель на буфер для
                                      значения параметра     */
   SQLINTEGER  *lenInd)  /* Входной: указатель на буфер для
                                      реальной длины параметра
                                      или индикатора NULL    */

/* Возвращает указатель на буфер или специальный тег
   динамического параметра                                       */
SQLSMALLINT SQLParamData(
SQLINTEGER   stmtHdl, /* Входной : дескриптор инструкции с
                                    динамическими параметрами*/
void        *prmTag)  /* Выходной: возвращаемый указатель
                                    буфера или тег параметра */

/* Передача значения очередному динамическому параметру
   выполняемой инструкции                                        */
SQLSMALLINT SQLPutData(
   SQLINTEGER    stmtHdl,/* Входной: дескриптор инструкции с
                                     динамич. параметрами   */
   void         *prmData,/* Входной: буфер, содержащий
                                     значение параметра     */
   SQLSMALLINT   prLnInd)/* Входной: длина параметра или
                                     индикатор значения NULL*/
```

Рис. 19.20. Функции CLI для обработки инструкций SQL

Выполнение инструкций с параметрами

Зачастую одна и та же инструкция SQL может многократно выполняться с разными значениями параметров. Например, так бывает при добавлении заказов в базу данных, когда для каждого нового заказа выполняется она и та же инструкция INSERT, но всякий раз в ней указываются иные идентификаторы клиента, товара и производителя и иное количество заказанного товара. Как и в случае динамического SQL, такие инструкции могут быть выполнены гораздо эффективнее,

если определить их переменные части в виде входных параметров. В текст инструкции, передаваемый функции SQLPrepare(), вставляются маркеры параметров (вопросительные знаки), указывающие, куда должны помещаться значения параметров, которые позднее, при выполнении инструкции, будут отдельно предоставлены СУБД.

Простейшим способом задания параметров инструкции SQL является вызов функции SQLBindParam(). Каждый вызов этой функции устанавливает связь между одним из маркеров параметров (идентифицируемым номером) и программной переменной (идентифицируемой адресом в памяти). Дополнительно может указываться еще одна целочисленная переменная, которая будет содержать длину входного параметра (если таковая переменна). В случае, когда параметр представляет собой строку, оканчивающуюся нулевым символом (такие строки широко используются в программах на языке С), функции SQLBindParam() может быть передана специальная константа SQL_NTS, указывающая, что функция CLI должна сама определить длину строки. Другая константа, SQL_NULL_DATA, указывает, что значением входного параметра является NULL. Если инструкция использует три маркера параметров, функция SQLBindParam() должна быть вызвана трижды — по одному разу для каждого входного параметра.

После того как связь между программными переменными (точнее, между адресами буферов программы, предназначенных для хранения данных) и входными параметрами инструкции SQL установлена, можно выполнить эту инструкцию с помощью функции SQLExecute(). Для изменения значений параметров достаточно перед очередным вызовом функции SQLExecute() поместить в связанные буферы новые данные. При желании любой параметр можно связать с другим буфером, вызвав функцию SQLBindParam() еще раз. На рис. 19.21 приведена программа, выполняющая инструкцию SQL с двумя входными параметрами. Программа циклически запрашивает у пользователя идентификатор клиента и его новый лимит кредита и заносит эту информацию в таблицу CUSTOMERS с помощью инструкции UPDATE.

```
/* Программа заносит в базу данных введенный
   пользователем лимит кредита клиента                    */
#include <sqlcli.h> /* файл с объявлениями CLI             */

main()
{
    SQLHENV   env_hdl;          /* Дескриптор среды SQL     */
    SQLHDBC   conn_hdl;         /* Дескриптор подключения   */
    SQLHSTMT  stmt_hdl;         /* Дескриптор инструкции    */
    SQLRETURN status;           /* Код возврата функции CLI */
    SQLCHAR * svr_name = "demo";/* Имя сервера              */
    SQLCHAR * user_name = "joe";/* Имя пользователя         */
    SQLCHAR * user_pswd = "xyz";/* Пароль для подключения   */
    char      amt_buf[ 31 ];    /* Введенная сумма          */
    SQLINTEGER amt_ind = SQL_NTS; /* Индикатор строки с
                                     завершающим нулевым
                                     символом               */
    char       cust_buf[ 31 ];    /* Введенный идентификатор*/
    SQLINTEGER cust_ind = SQL_NTS;/* Индикатор строки с
                                     завершающим нулевым
                                     символом               */
```

Рис. 19.21. CLI-программа с входными параметрами

```
    char stmt_buf[ 128 ];          /* Буфер инструкции SQL   */

    /* Выделяем память для среды SQL, сеанса подключения
       и инструкции и получаем их дескрипторы               */
    SQLAllocHandle( SQL_HANDLE_ENV,  SQL_NULL_HANDLE,
                    &env_hdl);
    SQLAllocHandle( SQL_HANDLE_DBC,  env_hdl,  &conn_hdl );
    SQLAllocHandle( SQL_HANDLE_STMT, conn_hdl, &stmt_hdl );

    /* Подключаемся к базе данных, указав имя сервера,
       имя пользователя и пароль. Спецификатор SQL_NTS говорит
       о том, что мы передаем строку переменной длины с
       завершающим нулевым символом.                        */
    SQLConnect( conn_hdl, svr_name, SQL_NTS,
                user_name, SQL_NTS,
                user_pswd, SQL_NTS );

    /* Подготавливаем инструкцию UPDATE
       с маркерами параметров                               */
    strcpy(stmt_buf,
           "update customers set credit limit = ? ");
    strcat(stmt_buf, "where cust_num = ?" );
    SQLPrepare( stmt_hdl, stmt_buf, SQL_NTS );

    /* Связываем параметры с программными буферами          */
    SQLBindParam( stmt_hdl, 1, SQL_C_CHAR, SQL_DECIMAL,
                  9, 2, &amt_buf, &amt_ind );
    SQLBindParam( stmt_hdl, 2, SQL_C_CHAR, SQL_INTEGER,
                  0, 0, &cust_buf, &cust_ind );

    /* Цикл обработки вводимых пользователем значений       */
    for ( ; ; )
        {
            /* Запрашиваем у пользователя идентификатор
               клиента и его лимит кредита                  */
            printf( "Введите идентификатор клиента: " );
            gets( cust_buf );
            if ( strlen( cust_buf ) == 0 )
                break;
            printf( "Введите новый лимит кредита: " );
            gets( amt_buf );

            /* Выполняем инструкцию с параметрами           */
            status = SQLExecute( stmt_hdl );

            if ( status )
                printf( "Ошибка при обновлении.\n" );
            else
                printf( "Обновление выполнено успешно.\n" );

            /* Завершаем транзакцию                         */
            SQLEndTran(SQL_HANDLE_ENV, env_hdl, SQL_COMMIT);
        }

    /* Отключаемся от сервера, освобождаем дескрипторы
       и выходим из программы                               */
    SQLDisconnect( conn_hdl );
    SQLFreeHandle( SQL_HANDLE_STMT, stmt_hdl );
    SQLFreeHandle( SQL_HANDLE_DBC,  conn_hdl );
    SQLFreeHandle( SQL_HANDLE_ENV,  env_hdl );

    exit();
}
```

Окончание рис. 19.21

Функции SQLParamData() и SQLPutData() реализуют альтернативный метод *отложенной передачи параметров* инструкции SQL. Суть его заключается в следующем. Вы, как обычно, вызываете для очередного параметра функцию SQLBindParam(), но вместо адреса буфера указываете, что для данного параметра будет использоваться режим отложенной передачи, и указываете специальное значение, которое будет идентифицировать этот параметр в дальнейшем.

Запросив выполнение инструкции с помощью функции SQLExecDirect() или SQLExecute(), программа вызывает функцию SQLParamData(), чтобы определить, требуются ли этой инструкции отложенные параметры. Если таковые действительно требуются, то функция SQLParamData() возвращает специальный код (SQL_NEED_DATA) и индикатор, указывающий, какой именно отложенный параметр требуется инструкции. Получив эту информацию, программа передает нужное значение с помощью функции SQLPutData(). Обычно после этого программа снова вызывает функцию SQLParamData(), чтобы выяснить, для какого еще параметра требуются динамические данные. Весь этот процесс циклически повторяется до тех пор, пока библиотеке CLI не будут предоставлены все значения параметров, после чего инструкция SQL будет, наконец, выполнена.

Описанный альтернативный метод передачи параметров значительно сложнее простого связывания параметров с программными буферами. Однако у него есть два важных преимущества. Первое из них заключается в том, что реальная передача данных (и выделение для них памяти) может быть отложена до того момента, когда эти данные потребуются. А второе преимущество состоит в том, что эта методика позволяет передавать очень длинные значения параметров по частям. Функцию SQLPutData() можно вызывать для одного и того же параметра несколько раз, в каждом вызове задавая очередную порцию данных. Например, если в предложении VALUES инструкции INSERT должен быть передан текст документа, его можно передавать по тысяче символов за раз, повторяя вызовы функции SQLPutData() до тех пор, пока весь документ не будет отправлен на сервер. Это позволяет избежать создания в программе слишком большого буфера для хранения всего документа.

Управление транзакциями в CLI

Операции завершения и отмены транзакции, реализуемые инструкциями COMMIT и ROLLBACK, можно выполнить и посредством CLI. Обе эти инструкции заменены функцией SQLEndTran(), объявление которой приведено на рис. 19.22. Эта функция использовалась для завершения транзакции в программах, представленных на рис. 19.17 и 19.21. Какую конкретно операцию должна выполнить функция SQLEndTran(), следует указать в параметре функции.

Функция SQLCancel(), объявление которой также приведено на рис. 19.22, непосредственно в управлении транзакциями не участвует, но на практике используется совместно с операцией отмены транзакции. Она отменяет инициированное функцией SQLExecDirect() или SQLExecute() выполнение инструкции SQL, для которой используется отложенная передача параметров. Если программа решает, что вместо передачи значения очередного параметра следует просто отменить выполнение инструкции, она вызывает функцию SQLCancel().

```
/* Завершение или отмена транзакции                          */
SQLSMALLINT SQLEndTran(
    SQLSMALLINT hdlType,    /* Входной: тип дескриптора      */
    SQLINTEGER  txnHdl,     /* Входной: дескриптор среды,
                               сеанса или инструкции          */
    SQLSMALLINT complType)  /* Входной: тип операции
                               (COMMIT или ROLLBACK)          */

/* Отмена выполнения текущей инструкции SQL                  */
SQLSMALLINT SQLCancel(
    SQLSMALLINT stmtHdl)    /* Входной: дескриптор
                                        инструкции            */
```

Рис. 19.22. Функции CLI для управления транзакциями

Кроме того, функция `SQLCancel()` может использоваться в многопоточном приложении для отмены еще не завершенной инструкции, выполняемой функцией `SQLExecDirect()` или `SQLExecute()`. Пока поток, вызвавший одну из этих функций, ждет завершения инструкции, другой параллельно выполняющийся поток может вызвать функцию `SQLCancel()`, передав ей тот же дескриптор инструкции. Особенности этого процесса и то, насколько прерываемыми являются функции CLI, зависит от конкретной реализации библиотеки.

Обработка результатов запроса

Функции CLI, о которых мы говорили до сих пор, могут использоваться для выполнения любых инструкций DML и DDL, не связанных с выборкой данных (т.е. инструкций `UPDATE`, `DELETE` и `INSERT`). Для обработки запросов на выборку требуются некоторые дополнительные функции CLI, объявления которых приведены на рис. 19.23, — ведь нужно не только выполнить запрос, но и получить его результаты. Проще всего это сделать с помощью функций `SQLBindCol()` и `SQLFetch()`. Чтобы получить результаты запроса, приложению нужно выполнить следующие действия (предполагается, что соединение с базой данных уже установлено).

1. Получить дескриптор инструкции при помощи функции `SQLAllocHandle()`.
2. Вызвать функцию `SQLExecDirect()` для передачи текста инструкции `SELECT` и выполнения запроса.
3. Вызвать функцию `SQLBindCol()` по одному разу для каждого столбца таблицы результатов запроса, который должен быть возвращен приложению. Каждый вызов функции связывает возвращаемый столбец с программным буфером.
4. С помощью функции `SQLFetch()` извлечь строку таблицы результатов запроса. Полученные данные помещаются в соответствующие буферы программы, указанные в вызовах функции `SQLBindCol()`.
5. Если запрос возвращает несколько строк результатов, повторять шаг 4 до тех пор, пока при очередном вызове функции не будет возвращено значение, указывающее, что все строки получены.
6. Когда все результирующие данные получены, программа должна вызвать функцию `SQLDisconnect()` для завершения сеанса подключения к базе данных.

```
/* Связывает столбец в таблице результатов запроса
   с буфером программы                                    */
SQLSMALLINT SQLBindCol(
   SQLINTEGER   stmtHdl, /* Входной: дескриптор инструкции */
   SQLSMALLINT  colNr,   /* Входной: номер столбца         */
   SQLSMALLINT  tgtType, /* Входной: тип размещаемых данных*/
   void        *value,   /* Входной: указатель на буфер    */
   SQLINTEGER   bufLen,  /* Входной: длина буфера          */
   SQLINTEGER  *lenInd)  /* Входной: указатель на буфер
                                      для реальной длины или
                                      индикатора NULL       */

/* Перемещает курсор на следующую
   строку в таблице результатов запроса                   */
SQLSMALLINT SQLFetch(
   SQLINTEGER   stmtHdl) /* Входной: дескриптор инструкции */

/* Перемещает курсор вверх или вниз
   по таблице результатов запроса                         */
SQLSMALLINT SQLFetchScroll(
   SQLINTEGER   stmtHdl, /* Входной: дескриптор инструкции */
   SQLSMALLINT  fetchDir,/* Входной: направление           */
   SQLINTEGER   offset)  /* Входной: смещение (строки)     */

/* Получение данных из одного столбца
   в таблице результатов запроса                          */
SQLSMALLINT SQLGetData(
   SQLINTEGER   stmtHdl, /* Входной: дескриптор инструкции */
   SQLSMALLINT  colNr,   /* Входной: номер столбца         */
   SQLSMALLINT  tgtType, /* Входной: тип данных            */
   void        *value,   /* Входной: указатель на буфер    */
   SQLINTEGER   bufLen,  /* Входной: длина буфера          */
   SQLINTEGER  *lenInd)  /* Выходной: указатель на буфер
                                       для реальной длины или
                                       индикатора NULL      */

/* Закрытие курсора - прекращение доступа
   к таблице результатов запроса                          */
SQLSMALLINT SQLCloseCursor(
   SQLINTEGER   stmtHdl) /* Входной: дескриптор инструкции */

/* Назначение имени открытому курсору                     */
SQLSMALLINT SQLSetCursorName(
   SQLINTEGER   stmtHdl, /* Входной: дескриптор инструкции */
   SQLCHAR     *cursName,/* Входной: имя курсора           */
   SQLSMALLINT  nameLen) /* Входной: длина имени           */

/* Получение имени открытого курсора                      */
SQLSMALLINT SQLGetCursorName(
   SQLINTEGER   stmtHdl, /* Входной: дескриптор инструкции */
   SQLCHAR     *cursName,/* Выходной: буфер для имени      */
   SQLSMALLINT  bufLen,  /* Входной:  длина буфера         */
   SQLSMALLINT *namLen)  /* Выходной: указатель на буфер
                                       для реальной длины
                                       полученного имени    */
```

Рис. 19.23. Функции CLI для обработки результатов запроса

Пример программы, выполняющей запрос на выборку с помощью функций CLI, приведен на рис. 19.24. Функционально он идентичен программе, представленной на рис. 19.10, использующей функции `dblib` в SQL Server. Поучительно

сравнить две эти программы. Вы увидите, что в них используются вызовы совершенно разных библиотечных функций (и с разными параметрами), но логическая схема их работы одинакова.

```
/* Эта программа выводит список служащих, у которых
   фактический объем продаж превышает план                    */
#include <sqlcli.h>   /* файл с объявлениями CLI               */

main()
{
    SQLHENV   env_hdl;             /* Дескриптор среды SQL  */
    SQLHDBC   conn_hdl;            /* Дескриптор подключения*/
    SQLHSTMT  stmt_hdl;            /* Дескриптор инструкции */
    SQLRETURN status;              /* Код возврата CLI      */
    SQLCHAR  *svr_name =  "demo"; /* Имя сервера           */
    SQLCHAR  *user_name = "joe";  /* Имя пользователя      */
    SQLCHAR  *user_pswd = "xyz";  /* Пароль для подключения*/
    char      repname[16];         /* Имя служащего         */
    float     repquota;            /* Плановый объем продаж */
    float     repsales;            /* Фактические продажи   */
    SQLSMALLINT repquota_ind;      /* Индикатор значения NULL
                                      для планового объема
                                      продаж                */
    char      stmt_buf[128];       /* Буфер инструкции SQL  */

    /* Получаем дескрипторы и подключаемся к базе данных   */
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,&env_hdl);
    SQLAllocHandle(SQL_HANDLE_DBC, env_hdl, &conn_hdl);
    SQLAllocHandle(SQL_HANDLE_STMT, conn_hdl, &stmt_hdl);
    SQLConnect(conn_hdl, svr_name,  SQL_NTS,
               user_name, SQL_NTS,
               user_pswd,  SQL_NTS);

    /* Запрашиваем выполнение инструкции SQL                */
    strcpy(stmt_buf,
           "select name, quota, sales from salesreps ");
    strcat(stmt_buf, "where sales > quota order by name");
    SQLExecDirect(stmt_hdl, stmt_buf, SQL_NTS);

    /* Связываем извлекаемые столбцы с буферами программы  */
    SQLBindCol(stmt_hdl, 1, SQL_C_CHAR, repname, 15, NULL);
    SQLBindCol(stmt_hdl, 2, SQL_C_FLOAT, &repquota,
                         0, &repquota_ind);
    SQLBindCol(stmt_hdl, 3, SQL_C_FLOAT, &repsales, 0, NULL);

    /* Цикл обработки результатов запроса                   */
    for ( ; ; )
    {
        /* Извлекаем следующую строку из
           таблицы результатов запроса                      */
        if (SQLFetch(stmt_hdl) != SQL_SUCCESS)
            break;

        /* Отображаем полученные данные                     */
        printf("Имя: %s\n", repname);
        if (repquota_ind < 0)
            printf("План не назначен.\n");
        else
```

Рис. 19.24. Получение результатов запроса с помощью функций CLI

```
            printf("План: %f\n", repquota);
        printf("Объем продаж: %f\n", repsales);
    }

    /* Отключаемся от сервера, освобождаем дескрипторы
       и выходим из программы                                  */
    SQLDisconnect(conn_hdl);
    SQLFreeHandle(SQL_HANDLE_STMT, stmt_hdl);
    SQLFreeHandle(SQL_HANDLE_DBC,  conn_hdl);
    SQLFreeHandle(SQL_HANDLE_ENV,  env_hdl);
    exit();
}
```

Окончание рис. 19.24

Каждый вызов функции `SQLBindCol()` устанавливает связь между одним столбцом в таблице результатов запроса (идентифицируемым по номеру) и программным буфером (идентифицируемым по адресу). Функция `SQLFetch()` использует эти связи для копирования данных из столбцов таблицы результатов запроса в соответствующие программные буферы. Если буфер предназначен для получения строк переменной длины, то используется еще один буфер, в который помещается значение длины реально записанных в первый буфер данных. Если же столбец содержит значение `NULL`, во второй буфер записывается соответствующая константа-индикатор.

С помощью функций, приведенных на рис. 19.23, можно реализовать альтернативный способ получения результатов запроса, который заключается в следующем. Столбцы в таблице результатов запроса не привязываются заранее к программным буферам. Вместо этого функция `SQLFetch()` просто перемещает курсор к следующей строке таблицы, но данные в буферы не заносятся. Чтобы их получить, нужно вызвать другую функцию — `SQLGetData()`. Один из параметров этой функции определяет, какой столбец требуется получить. Остальные задают тип возвращаемых данных и адрес буфера, в который их нужно записать, а также адрес сопутствующего буфера для записи длины полученных данных или индикатора значения `NULL`.

В принципе, функция `SQLGetData()` дает тот же результат, что и связывание столбцов с программными буферами с помощью функции `SQLBindCol()`. Однако при работе с большими элементами данных у ее применения есть важное преимущество. Некоторые СУБД позволяют определять столбцы, которые могут содержать тысячи или даже миллионы байтов данных. Выделять в приложении буфер для хранения данных такого объема зачастую неудобно, особенно если это не требуется с точки зрения программной логики — например, в случае, когда программа могла бы обрабатывать эти данные по частям. Именно такую возможность и предоставляет программисту функция `SQLGetData()`: вы можете зарезервировать буфер разумного размера и обрабатывать получаемые данные ограниченными порциями.

Ничто не препятствует смешиванию обеих методик для обработки результатов одного запроса. Если часть столбцов из таблицы результатов запроса вы свяжете с буферами программы посредством функции `SQLBindCol()`, а остальные оставите несвязанными и затем вызовете функцию `SQLFetch()`, она поместит содержимое связанных столбцов текущей строки в буферы, а значения остальных столбцов

можно будет извлечь отдельно с помощью функции SQLGetData(). Это удобно, когда запрос извлекает из базы данных часть столбцов небольшого размера, например фамилии, даты, денежные суммы, а один или два столбца — с данными большого объема, такими, например, как текст контракта. Однако учтите, что в некоторых реализациях CLI возможности смешивания этих двух способов обработки результатов запроса могут быть ограничены. В частности, иногда требуется, чтобы все связанные столбцы размещались подряд в начале списка возвращаемых столбцов, а несвязанные столбцы следовали за ними.

Курсоры с произвольным доступом

Стандарт SQL/CLI поддерживает курсоры с произвольным доступом, аналогичные тем, которые с самого начала были включены в стандарт SQL для встроенного SQL. Перемещение по набору записей обеспечивает функция SQLFetchScroll(), объявление которой приведено на рис. 19.23. Эта функция представляет собой реализацию инструкции FETCH с несколько расширенными возможностями. Она позволяет перемещать указатель вперед, назад и на указанную строку. Первым параметром функции SQLFetchScroll() передается дескриптор инструкции SQL, как и в случае функции SQLFetch(). Два других параметра определяют направление перехода и величину смещения (которое может быть абсолютным и относительным, т.е. отсчитываться от первой или от текущей записи набора). Что касается использования функций SQLBindCol() и SQLGetData(), то в комплексе с функцией SQLFetchScroll() они используются точно так же, как и с функцией SQLFetch().

Именованные курсоры

Обратите внимание на то, что в стандарте SQL/CLI отсутствует функция для объявления курсора, эквивалентная инструкции DECLARE CURSOR встроенного SQL. Текст запроса на выборку просто передается CLI для выполнения (точно так же как текст любой другой инструкции SQL) с помощью функции SQLExecDirect() или пары функций SQLPrepare()/SQLExecute(). Таблица результатов запроса идентифицируется дескриптором инструкции, и доступ к ней осуществляется с помощью функций SQLFetch(), SQLBindCol() и т.п. Таким образом, роль имени курсора играет дескриптор инструкции SQL.

Слабым местом этого подхода являются позиционные обновления и удаления записей. Как рассказывалось в главе 17, "Встроенный SQL", встроенный SQL позволяет модифицировать или удалить текущую строку из таблицы результатов запроса (выбранную с помощью инструкции FETCH) посредством специальных инструкций UPDATE ... WHERE CURRENT OF и DELETE ... WHERE CURRENT OF. В этих инструкциях указывается имя курсора, поскольку программа может одновременно открыть несколько курсоров, чтобы обрабатывать результаты нескольких запросов.

Для поддержки аналогичных возможностей в приложениях, использующих CLI, стандартом определена функция SQLSetCursorName(), объявление которой приведено на рис. 19.23. Она назначает курсору заданное вами имя, указанное в одном параметре функции, при этом во втором параметре передается дескриптор инструкции SQL, идентифицирующий курсор. Назначенное курсору имя можно затем использовать в позиционных инструкциях UPDATE и DELETE, которые вы передаете CLI для выполнения. Сопутствующая функция SQLGetCursorName() позволяет получить заданное ранее имя курсора, указав дескриптор инструкции.

Выполнение динамических запросов

Если столбцы в таблице результатов запроса заранее, в процессе разработки программы, не известны, то их характеристики можно узнать в процессе выполнения программы с помощью функций, объявления которых приведены на рис. 19.25. Эти функции реализуют возможности динамического SQL, который для встроенного SQL был описан в главе 18, "Динамический SQL".

```
/* Определение количества столбцов
   в таблице результатов запроса                                  */
SQLSMALLINT SQLNumResultCols(
  SQLINTEGER   stmtHdl,  /* Входной:  дескриптор инструкции */
  SQLSMALLINT *colCount) /* Выходной: указатель на буфер, в
                                      который записывается
                                      количество столбцов   */

/* Получение характеристик столбца
   таблицы результатов запроса                                    */
SQLSMALLINT SQLDescribeCol(
  SQLINTEGER   stmtHdl,  /* Входной:  дескриптор инструкции */
  SQLSMALLINT  colNr,    /* Входной:  номер столбца, о
                                      котором нужна
                                      информация            */
  SQLCHAR     *colName,  /* Выходной: имя столбца           */
  SQLSMALLINT  bufLen,   /* Входной:  длина буфера имени    */
  SQLSMALLINT *namLen,   /* Выходной: указатель на буфер с
                                      реальной длиной имени */
  SQLSMALLINT *colType,  /* Выходной: указатель на буфер типа
                                      данных столбца        */
  SQLSMALLINT *colSize,  /* Выходной: указатель на буфер для
                                      размера данных столбца*/
  SQLSMALLINT *decDigits,/* Выходной: указатель на буфер для
                                      количества десятичных
                                      цифр в столбце        */
  SQLSMALLINT *nullable) /* Выходной: указатель на буфер для
                                      признака того, что
                                      столбец может содержать
                                      значения NULL         */

/* Получение информации об указанном атрибуте
   столбца в таблице результатов запроса                          */
SQLSMALLINT SQLColAttribute(
  SQLINTEGER   stmtHdl,  /* Входной:  дескриптор инструкции */
  SQLSMALLINT  colNr,    /* Входной:  номер столбца,  о
                                      котором нужна
                                      информация            */
  SQLSMALLINT  attrCode, /* Входной:  код атрибута          */
  SQLCHAR     *attrInfo, /* Выходной: буфер для строкового
                                      значения атрибута     */
  SQLSMALLINT  bufLen,   /* Входной:  размер буфера для
                                      строкового значения   */
  SQLSMALLINT *actLen,   /* Выходной: указатель на буфер для
                                      реальной длины
                                      строкового значения   */
  SQLINTEGER  *numAttr)  /* Выходной: указатель на буфер для
                                      числового значения
                                      атрибута              */
```

Рис. 19.25. Функции CLI для обработки динамических запросов

```
/* Получение значений набора полей из описателя CLI          */
SQLSMALLINT SQLGetDescRec(
  SQLINTEGER   descHdl,  /* Входной:  дескриптор описателя  */
  SQLSMALLINT  recNr,    /* Входной:  номер записи          */
  SQLCHAR  *name,        /* Выходной: имя описываемого
                                       элемента              */
  SQLSMALLINT  bufLen,   /* Входной:  длина буфера для имени
                                       описываемого элемента */
  SQLSMALLINT *namLen,   /* Выходной: указатель на буфер для
                                       реальной длины имени  */
  SQLSMALLINT *dataType, /* Выходной: указатель на буфер для
                                       кода типа данных
                                       описываемого элемента */
  SQLSMALLINT *subType,  /* Выходной: указатель на буфер для
                                       дополнительного кода
                                       типа данных
                                       описываемого элемента */
  SQLSMALLINT *length,   /* Выходной: указатель на буфер для
                                       длины описываемого
                                       элемента              */
  SQLSMALLINT *precis,   /* Выходной: указатель на буфер для
                                       точности представления
                                       описываемого элемента */
  SQLSMALLINT *scale,    /* Выходной: указатель на буфер для
                                       фактора масштабирования
                                       описываемого элемента */
  SQLSMALLINT *nullable) /* Выходной: указатель на буфер для
                                       признака того, может ли
                                       описываемый элемент
                                       содержать NULL        */

/* Установка значений набора полей в описателе CLI           */
SQLSMALLINT SQLSetDescRec(
  SQLINTEGER   descHdl,  /* Входной:  дескриптор описателя  */
  SQLSMALLINT  recNr,    /* Входной:  номер записи          */
  SQLSMALLINT  dataType, /* Входной:  код типа данных
                                       описываемого элемента */
  SQLSMALLINT  subType,  /* Входной:  дополнительный код типа
                                       данных описываемого
                                       элемента              */
  SQLSMALLINT  length,   /* Входной:  длина элемента        */
  SQLSMALLINT  precis,   /* Входной:  точность представления*/
  SQLSMALLINT  scale,    /* Входной:  фактор масштабирования*/
  void        *dataBuf,  /* Входной:  адрес буфера для
                                       описываемого элемента */
  SQLSMALLINT  bufLen,   /* Входной:  длина буфера данных   */
  SQLSMALLINT *indBuf)   /* Входной:  указатель на буфер для
                                       признака того, может ли
                                       описываемый элемент
                                       содержать NULL        */

/* Получение значения указанного поля из описателя CLI       */
SQLSMALLINT SQLGetDescField(
  SQLINTEGER   descHdl,  /* Входной:  дескриптор описателя  */
  SQLSMALLINT  recNr,    /* Входной:  номер записи          */
  SQLSMALLINT  attrCode, /* Входной:  код запрашиваемого
                                       атрибута              */
  void        *attrInfo, /* Выходной: буфер для значения
                                       атрибута              */
  SQLSMALLINT  bufLen,   /* Входной:  длина буфера          */
```

Продолжение рис. 19.25

```
  SQLSMALLINT *actLen)    /* Выходной: указатель на буфер для
                                       реальной длины данных */

/* Задание значения указанного поля в описателе CLI          */
SQLSMALLINT SQLSetDescField(
  SQLINTEGER   descHdl,  /* Входной:  дескриптор описателя  */
  SQLSMALLINT  recNr,    /* Входной:  номер записи          */
  SQLSMALLINT  attrCode, /* Входной:  код описываемого
                                       атрибута              */
  void        *attrInfo, /* Выходной: буфер с новым значением
                                       атрибута              */
  SQLSMALLINT  bufLen)   /* Входной:  длина буфера          */

/* Копирует содержимое одного описателя CLI в другой         */
SQLSMALLINT SQLCopyDesc(
  SQLINTEGER   inDscHdl, /* Входной:  дескриптор исходного
                                       описателя CLI         */
  SQLINTEGER   outDscHdl)/* Входной:  дескриптор целевого
                                       описателя CLI         */
```

Окончание рис. 19.25

Чтобы выполнить с их помощью динамически определяемый запрос, приложению нужно предпринять следующие действия (предполагается, что соединение с базой данных уже установлено).

1. Получить дескриптор инструкции с помощью функции `SQLAllocHandle()`.
2. Вызвать функцию `SQLPrepare()`, передав ей текст инструкции `SELECT`.
3. Вызвать функцию `SQLExecute()` для выполнения запроса.
4. Вызвать функцию `SQLNumResultCols()`, чтобы узнать количество столбцов в таблице результатов запроса.
5. Вызвать функцию `SQLDescribeCol()` по одному разу для каждого столбца, который должен быть возвращен приложению. При каждом вызове функции возвращается тип данных столбца, его размер, специальная константа-индикатор, указывающая, может ли столбец содержать значения `NULL`, и т.п.
6. Выделить буферы для получения результатов запроса и связать их со столбцами с помощью функции `SQLBindCol()` (по одному вызову для каждого столбца).
7. Используя функцию `SQLFetch()`, извлечь строку таблицы результатов запроса. Функция `SQLFetch()` перемещает курсор к следующей строке результатов запроса и помещает содержимое строки в программные буферы, предварительно связанные со столбцами таблицы результатов запроса с помощью функции `SQLBindCol()`.
8. Если запрос возвращает не одну строку, программа повторяет шаг 7 до тех пор, пока возвращаемое функцией `SQLFetch()` значение не покажет, что все строки получены.
9. Когда все данные получены, программа должна вызвать функцию `SQLCloseCursor()` для прекращения доступа к таблице результатов запроса.

На рис. 19.26 приведена программа, которая выполняет динамический запрос с использованием описанной методики. Концептуально эта программа идентична двум другим примерам программ, выполняющим динамические запросы: с помощью встроенного динамического SQL (рис. 19.16) и с помощью dblib (рис. 19.15). И снова сравнение работы этих программ очень познавательно и обеспечит лучшее понимание вами динамического SQL. Несмотря на разные имена вызовов API, последовательности вызовов функций в программе с применением dblib (рис. 19.15) и программе, использующей CLI (рис. 19.26), почти идентичны. Связка dbcmd()/dbsqlexec()/dbresults() заменяется функцией SQLExecDirect() (в данном случае запрос будет выполнен только один раз, поэтому нет необходимости в предварительной компиляции инструкции). Функция dbnumcols() заменяется функцией SQLNumResultCols(). Функции, извлекающие информацию о столбце таблицы результатов запроса (dbcolname(), dbcoltype(), dbcollen()), заменяются одной функцией SQLDescribeCol(). Роль функции dbnextrow() выполняет функция SQLFetch(). Остальные же модификации связаны с изменением синтаксиса API-функций.

```
main()
{
    /* Эта программа запрашивает у пользователя имя таблицы
       и перечень столбцов, которые он хочет получить.
       После этого программа формирует и выполняет запрос
       и отображает его результаты.                          */

    SQLHENV  env_hdl;            /* Дескриптор среды SQL    */
    SQLHDBC  conn_hdl;           /* Дескриптор подключения  */
    SQLHSTMT stmt1_hdl;          /* Дескриптор инструкции
                                    главного запроса         */
    SQLHSTMT stmt2_hdl;          /* Дескриптор инструкции
                                    запроса имен столбцов    */
    SQLRETURN status;            /* Код возврата CLI         */
    SQLCHAR *svr_name = "demo"; /* Имя сервера              */
    SQLCHAR *user_name = "joe"; /* Имя пользователя         */
    SQLCHAR *user_pswd = "xyz"; /* Пароль для подключения   */
    char stmtbuf[ 2001 ];        /* Текст главного запроса   */
    char stmt2buf[ 2001 ];       /* Текст запроса
                                    имен столбцов            */
    char querytbl[ 32 ];         /* Введенное пользователем
                                    имя таблицы              */
    char querycol[ 32 ];         /* Заданный столбец         */
    int  first_col = 0;          /* Это первый столбец?      */
    SQLSMALLINT colcount;        /* Количество столбцов в
                                    таблице результатов      */
    SQLCHAR    *nameptr;         /* Адрес буфера, в который
                                    записывается имя столбца*/
    SQLSMALLINT namelen;         /* Длина имени столбца      */
    SQLSMALLINT type;            /* Код типа данных,
                                    содержащихся в столбце   */
    SQLSMALLINT size;            /* Размер столбца           */
    SQLSMALLINT digits;          /* Количество цифр          */
    SQLSMALLINT nullable;        /* Признак допустимости
                                    значений NULL            */
    short i;                     /* Индекс столбца           */
    char  inbuf[ 101 ];          /* Данные, вводимые
                                    пользователем            */
```

Рис. 19.26. Использование CLI для динамических запросов

```
SQLCHAR *item_name[ 100 ];  /* Массив имен столбцов    */
SQLCHAR *item_data[ 100 ];  /* Массив буферов с данными
                               для столбцов            */
int   item_ind[ 100 ];      /* Массив индикаторов
                               для столбцов            */
SQLSMALLINT item_type[100]; /* Массив типов данных
                               столбцов                */
SQLCHAR *dataptr;           /* Адрес буфера
                               текущего столбца        */

/* Получаем необходимые дескрипторы
   и подключаемся к базе данных                        */
SQLAllocHandle( SQL_HANDLE_ENV,  SQL_NULL_HANDLE,
                &env_hdl );
SQLAllocHandle( SQL_HANDLE_DBC,  env_hdl, &conn_hdl );
SQLAllocHandle( SQL_HANDLE_STMT, conn_hdl,&stmt1_hdl );
SQLAllocHandle( SQL_HANDLE_STMT, conn_hdl,&stmt2_hdl );
SQLConnect( conn_hdl, svr_name,  SQL_NTS,
            user_name, SQL_NTS,
            user_pswd, SQL_NTS );

/* Спрашиваем у пользователя, из какой таблицы он хочет
   извлечь данные                                      */
printf( "*** Программа формирования запросов ***\n" );
printf( "Введите имя таблицы для запроса: " );
gets( querytbl );

/* Начинаем формирование инструкции SELECT в буфере    */
strcpy( stmtbuf, "select " );

/* Запрашиваем имена столбцов                          */
strcpy( stmt2buf, "select column_name from columns "
                  "where table_name = " );
strcat( stmt2buf, querytbl );
SQLExecDirect( stmt2_hdl, stmt2buf, SQL_NTS );

/* Обрабатываем результаты запроса                     */
SQLBindCol( stmt2_hdl, 1, SQL_C_CHAR, querycol,
            31, (int *)0 );

while ( status = SQLFetch( stmt2_hdl ) == SQL_SUCCESS )
{
    printf( "Включить столбец %s (y/n)?", querycol );
    gets( inbuf );

    if ( inbuf[ 0 ] = "y" )
    {
        /* Пользователь выбрал столбец; включаем его   */
        if ( first_col++ > 0 )
            strcat( stmtbuf, ", " );
        strcat( stmtbuf, querycol );
    }
}

/* Заканчиваем инструкцию SQL предложением FROM        */
strcat( stmtbuf, "from " );
strcat( stmtbuf, querytbl );

/* Выполняем запрос и готовимся
   к получению его результатов                         */
SQLExecDirect( stmt1_hdl, stmtbuf, SQL_NTS );
```

Продолжение рис. 19.26

```
/* Запрашиваем информацию о каждом столбце,
   выделяем память и связываем столбцы с буферами      */
SQLNumResultCols( stmt1_hdl, &colcount );
for ( i = 0; i < colcount; i++ )
{
    item_name[ i ] = nameptr = malloc( 32 );
    indptr = &item_ind[ i ];
    SQLDescribeCol( stmt1_hdl, i, nameptr, 32, &namelen,
                    &type, &size, &digits, &nullable );
    switch ( type )
    {
    case SQL_CHAR:
    case SQL_VARCHAR:
        /* Выделяем строковый буфер
           и связываем с ним столбец                     */
        item_data[ i ] = dataptr = malloc( size + 1 );
        item_type[ i ] = SQL_C_CHAR;
        SQLBindCol( stmt1_hdl, i, SQL_C_CHAR, dataptr,
                    size + 1, indptr );
        break;
    case SQL_TYPE_DATE:
    case SQL_TYPE_TIME:
    case SQL_TYPE_TIME_WITH_TIMEZONE:
    case SQL_TYPE_TIMESTAMP:
    case SQL_TYPE_TIMESTAMP_WITH_TIMEZONE:
    case SQL_INTERVAL_DAY:
    case SQL_INTERVAL_DAY_TO_HOUR:
    case SQL_INTERVAL_DAY_TO_MINUTE:
    case SQL_INTERVAL_DAY_TO_SECOND:
    case SQL_INTERVAL_HOUR:
    case SQL_INTERVAL_HOUR_TO_MINUTE:
    case SQL_INTERVAL_HOUR_TO_SECOND:
    case SQL_INTERVAL_MINUTE:
    case SQL_INTERVAL_MINUTE_TO_SECOND:
    case SQL_INTERVAL_MONTH:
    case SQL_INTERVAL_SECOND:
    case SQL_INTERVAL_YEAR:
    case SQL_INTERVAL_YEAR_TO_MONTH:
        /* Просим CLI преобразовать значение
           даты/времени в С-строку                       */
        item_data[ i ] = dataptr = malloc( 31 );
        item_type[ i ] = SQL_C_CHAR;
        SQLBindCol( stmt1_hdl, i, SQL_C_CHAR, dataptr,
                    31, indptr );
        break;
    case SQL_INTEGER:
    case SQL_SMALLINT:
        /* Преобразовать целое число в long языка С      */
        item_data[ i ] = dataptr
                       = malloc( sizeof( integer ) );
        item_type[ i ] = SQL_C_SLONG;
        SQLBindCol( stmt1_hdl, i, SQL_C_SLONG, dataptr,
                    sizeof( integer ), indptr );
        break;
    case SQL_NUMERIC:
    case SQL_DECIMAL:
    case SQL_FLOAT:
    case SQL_REAL:
    case SQL_DOUBLE:
        /* В демонстрационных целях конвертируем числа,
           относящиеся к этим типам данных, в формат
```

Продолжение рис. 19.26

```
            чисел с плавающей точкой языка C                    */
        item_data[ i ] = dataptr
                       = malloc( sizeof( long ) );
        item_type[ i ] = SQL_C_DOUBLE;
        SQLBindCol( stmt1_hdl, i, SQL_C_DOUBLE, dataptr,
                    sizeof( double ), indptr );
        break;
    default:
        /* Остальные типы данных не обрабатываются */
        printf( "Не могу обработать тип данных %d\n",
                type );
        exit();
    }
}

/* Извлекаем строки из таблицы результатов запроса
 и отображаем их на экране                                      */
while ( status = SQLFetch( stmt1_hdl ) == SQL_SUCCESS )
{

    /* Цикл вывода данных для каждого
       столбца текущей строки                                   */
    printf( "\n" );
    for ( i = 0; i < colcount; i++ )
    {
        /* Выводим имя столбца */
        printf( "Столбец № %d (%s): ", i + 1,
                item_name[ i ] );
        /* Проверяем, не содержит ли столбец
           значение NULL                                        */
        if ( item_ind[ i ] == SQL_NULL_DATA )
        {
            puts( "содержит NULL!\n" );
            continue;
        }
        /* Обрабатываем данные каждого возвращаемого
         типа (возможно, преобразованного) отдельно             */
        switch ( item_type[ i ] )
        {
        case SQL_C_CHAR:
            /* Получены текстовые данные - выводим их          */
            puts( item_data[ i ] );
            break;
        case SQL_C_LONG:
            /* Целое - конвертируем и выводим                  */
            printf( "%ld", *((int *)(item_data[i])));
            break;
        case SQL_C_DOUBLE:
            /* Число с плавающей точкой -
               конвертируем и выводим                          */
            printf( "%ld", *((double*)(item_data[i])));
            break;
        }
    }
}
printf( "\nКонец данных.\n" );

/* Освобождаем выделенную память                                */
for ( i = 0; i < colcount; i++ )
```

Продолжение рис. 19.26

```
    {
        free( item_data[ i ] );
        free( item_name[ i ] );
    }

    /* Отключаемся от сервера, освобождаем дескрипторы
       и выходим из программы                                  */
    SQLDisconnect( conn_hdl );
    SQLFreeHandle( SQL_HANDLE_STMT, stmt1_hdl );
    SQLFreeHandle( SQL_HANDLE_STMT, stmt2_hdl );
    SQLFreeHandle( SQL_HANDLE_DBC, conn_hdl );
    SQLFreeHandle( SQL_HANDLE_ENV, env_hdl );
    exit();
}
```

Окончание рис. 19.26

Если сравнить программу, представленную на рис. 19.26, с программой, использующей встроенный SQL (рис. 19.16), то одним из главных отличий окажется то, что во встроенном SQL применяется специальная область данных SQL (SQL Data Area — `SQLDA`), предназначенная для привязки столбцов и получения их описания. В CLI эту задачу выполняют несколько функций: `SQLNumResultCols()`, `SQLDescribeCol()` и `SQLBindCol()`. Большинству программистов удобнее пользоваться функциями CLI. Однако CLI предоставляет в распоряжение программистов и альтернативный низкоуровневый метод, дублирующий возможности `SQLDA`.

В этом альтернативном методе выполнения динамических запросов используются специальные структуры CLI, называемые *описателями* (*дескрипторами*, descriptor). Описатель CLI содержит низкоуровневую информацию о параметрах инструкции SQL (описатель параметров) или о столбцах таблицы результатов запроса (описатель строк). Информация в описателе подобна той, что содержится в переменной части области `SQLDA`, — имя столбца или параметра, тип и дополнительный тип данных, длина данных, адрес буфера, адрес для индикатора значения `NULL` и т.д. Описатели параметров и строк соответствуют входным и выходным областям `SQLDA`, которые имеются в реализациях динамического SQL некоторых популярных СУБД.

Описатели CLI идентифицируются своими дескрипторами. При подготовке инструкции SQL к выполнению библиотека CLI создает для ее параметров и для столбцов таблицы результатов запроса по два описателя: один из них представляет собой буфер прикладных значений, т.е. тех значений, которыми оперирует непосредственно приложение, а второй — это буфер системных значений, которыми оперирует уже СУБД (над этими значениями могут выполняться дополнительные операции преобразования типов данных). В качестве альтернативы программа может создать и свои собственные описатели. Дескрипторы описателей, связанных с конкретной инструкцией SQL, считаются ее атрибутами, и доступ к ним можно получить через дескриптор самой инструкции. Приложение может получать и устанавливать значения дескрипторов описателей с помощью функций управления атрибутами, описанных далее в настоящей главе.

Для получения хранящейся в описателе информации используются две функции. Функция `SQLGetDescField()` извлекает из описателя отдельное поле, идентифицируемое значением кода. Она обычно применяется для определения типов данных и размеров столбцов в таблице результатов запроса. Функция `SQLGetDescRec()` воз-

вращает набор наиболее часто используемых полей, включая имя столбца или параметра, основной и дополнительный тип данных, длину, точность и фактор масштабирования (если описываемый элемент является десятичным числом), а также индикатор, указывающий, может ли столбец содержать значения NULL. Аналогичная пара функций предназначена для занесения информации в описатель. Функция SQLSetDescField() устанавливает значение указанного поля описателя, а функция SQLSetDescRec() задает значения набора наиболее часто используемых полей. Для удобства в CLI включена также функция SQLCopyDesc(), копирующая всю информацию из одного описателя в другой.

Ошибки CLI и диагностическая информация

Каждая функция CLI возвращает целочисленное значение типа short, указывающее, как завершилась ее работа. Если код завершения свидетельствует об ошибке, можно воспользоваться функциями обработки ошибок (объявления которых приведены на рис. 19.27), чтобы выяснить, что именно произошло. Чаще всего с этой целью вызывается функция SQLError(). Приложение передает ей дескрипторы среды SQL, сеанса подключения и инструкции и получает значение переменной SQLSTATE, системный код ошибки (сгенерированный подсистемой, в которой она произошла) и сообщение об ошибке в текстовой форме.

```
/* Получение информации об ошибке, происшедшей
   во время последнего вызова функции CLI                    */
SQLSMALLINT SQLError(
  SQLINTEGER   envHdl,    /* Входной:  дескриптор среды      */
  SQLINTEGER   connHdl,   /* Входной:  дескриптор сеанса
                                       подключения            */
  SQLINTEGER   stmtHdl,   /* Входной:  дескриптор инструкции*/
  SQLCHAR     *sqlState,  /* Выходной: пятисимвольное
                                       значение переменной
                                       SQLSTATE               */
  SQLINTEGER  *nativeErr,/* Выходной: указатель на буфер для
                                       системно-зависимого
                                       кода ошибки            */
  SQLCHAR     *msgBuf,    /* Выходной: буфер для сообщения
                                       об ошибке              */
  SQLSMALLINT  bufLen,    /* Входной:  длина буфера          */
  SQLSMALLINT *msgLen)    /* Выходной: указатель на буфер для
                                       реальной длины полу-
                                       ченного сообщения      */

/* Определение количества строк, обработанных последней
   инструкцией SQL                                           */
SQLSMALLINT SQLRowCount(
  SQLINTEGER   stmtHdl,   /* Входной:  дескриптор инструкции*/
  SQLINTEGER  *rowCnt)    /* Выходной: указатель на буфер для
                                       количества строк       */

/* Получение значений набора полей из структуры CLI,
   содержащей диагностическую информацию                     */
SQLSMALLINT SQLGetDiagRec(
  SQLSMALLINT  hdlType,   /* Входной:  код типа дескриптора */
  SQLINTEGER   inHdl,     /* Входной:  дескриптор CLI        */
```

Рис. 19.27. Функции CLI для обработки ошибок

```
  SQLSMALLINT  recNr,     /* Входной:  номер запрашиваемой
                                       записи с информацией
                                       об ошибке            */
  SQLCHAR     *sqlState, /* Выходной: пятисимвольное значение
                                       переменной SQLSTATE  */
  SQLINTEGER  *nativeErr,/* Выходной: указатель на буфер для
                                       системно-зависимого
                                       кода ошибки          */
  SQLCHAR     *msgBuf,    /* Выходной: буфер для сообщения
                                       об ошибке            */
  SQLSMALLINT  bufLen,    /* Входной:  длина буфера         */
  SQLSMALLINT *msgLen)    /* Выходной: указатель на буфер для
                                       реальной длины полу-
                                       ченного сообщения    */

/* Получение значения указанного поля из структуры CLI,
   содержащей диагностическую информацию                    */
SQLSMALLINT SQLGetDiagField(
  SQLSMALLINT  hdlType,   /* Входной:  код типа дескриптора */
  SQLINTEGER   inHdl,     /* Входной:  дескриптор CLI       */
  SQLSMALLINT  recNr,     /* Входной:  номер запрашиваемой
                                       записи с информацией
                                       об ошибке            */
  SQLSMALLINT  diagId,    /* Входной:  идентификатор требуе-
                                       мого поля записи     */
  void        *diagInfo, /* Выходной: полученная диагности-
                                       ческая информация    */
  SQLSMALLINT  bufLen,    /* Входной:  длина буфера диагнос-
                                       тической информации  */
  SQLSMALLINT *actLen)    /* Выходной: указатель на буфер для
                                       реальной длины
                                       полученной информации*/
```

Окончание рис. 19.27

Функция `SQLError()` на самом деле возвращает только наиболее часто используемую информацию из диагностической структуры CLI. Другие функции обработки ошибок предоставляют более полную информацию, которую они получают путем непосредственного доступа к диагностическим записям, создаваемым и поддерживаемым CLI. В общем случае CLI может сгенерировать несколько ошибок и соответственно создать несколько диагностических записей. Функция `SQLGetDiagRec()` возвращает одну диагностическую запись, идентифицируемую ее номером. Вызвав эту функцию несколько раз, приложение может получить основную информацию обо всех ошибках, происшедших в ходе выполнения последней вызванной функции CLI. Более полная информация может быть получена путем исследования отдельных полей диагностической записи. Эту возможность предоставляет функция `SQLGetDiagField()`.

Еще одна функция, `SQLRowCount()`, хотя и не имеет прямого отношения к обработке ошибок, тоже вызывается *после* функции `SQLExecute()`. Она возвращает количество строк, обработанных инструкцией SQL (например, значение 4 будет возвращено после выполнения инструкции `UPDATE`, обновившей четыре записи).

Атрибуты CLI

CLI поддерживает ряд опций, управляющих выполнением библиотечных функций. Некоторые из них определяют относительно мелкие, но важные детали, как, например, должна ли библиотека CLI считать, что все передаваемые ее функциям строковые значения оканчиваются нулевым символом. Другие опции влияют на более глобальные аспекты работы CLI, например на возможность произвольного доступа курсоров.

Приложения управляют всеми этими опциями посредством *атрибутов* CLI. Атрибуты организованы в иерархию, аналогичную иерархии дескрипторов CLI, — среда/сеанс/инструкция. Атрибуты среды определяют общие аспекты работы CLI. Атрибуты сеанса играют роль только в контексте конкретного сеанса подключения, созданного с помощью функции `SQLConnect()`, и для разных сеансов могут быть различными. Наконец, атрибуты инструкции влияют на выполнение конкретной инструкции, идентифицируемой ее дескриптором.

Значения атрибутов устанавливаются и извлекаются с помощью набора функций, объявления которых приведены на рис. 19.28. Функции `SQLGetEnvAttr()`, `SQLGetConnectAttr()` и `SQLGetStmtAttr()` считывают значения атрибутов, а функции `SQLSetEnvAttr()`, `SQLSetConnectAttr()` и `SQLSetStmtAttr()` их устанавливают. Во всех этих функциях конкретный обрабатываемый атрибут идентифицируется значением кода.

```
/* Получение значения указанного атрибута среды SQL        */
SQLSMALLINT SQLGetEnvAttr(
  SQLINTEGER  envHdl,  /* Входной:  дескриптор среды      */
  SQLINTEGER  attrCode,/* Входной:  код атрибута          */
  void       *rtnVal,  /* Выходной: возвращаемое значение */
  SQLINTEGER  bufLen,  /* Входной:  длина буфера          */
  SQLINTEGER *strLen)  /* Выходной: указатель на буфер для
                                    реальной длины данных */

/* Установка значения указанного атрибута среды SQL        */
SQLSMALLINT SQLSetEnvAttr(
  SQLINTEGER  envHdl,  /* Входной:  дескриптор среды      */
  SQLINTEGER  attrCode,/* Входной:  код атрибута          */
  void       *attrVal, /* Входной:  новое значение
                                    атрибута              */
  SQLINTEGER  strLen)  /* Выходной: длина данных          */

/* Получение значения указанного атрибута подключения      */
SQLSMALLINT SQLGetConnectAttr(
  SQLINTEGER  connHdl, /* Входной:  дескриптор подключения*/
  SQLINTEGER  attrCode,/* Входной:  код атрибута          */
  void       *rtnVal,  /* Выходной: возвращаемое значение */
  SQLINTEGER  bufLen,  /* Входной:  длина буфера          */
  SQLINTEGER *strLen)  /* Выходной: указатель на буфер для
                                    реальной длины данных */

/* Установка значения указанного атрибута подключения      */
SQLSMALLINT SQLSetConnectAttr(
  SQLINTEGER  connHdl, /* Входной:  дескриптор подключения*/
  SQLINTEGER  attrCode,/* Входной:  код атрибута          */
```

Рис. 19.28. Функции CLI для управления атрибутами

```
  void        *attrVal, /* Входной:  новое значение
                                     атрибута              */
  SQLINTEGER  strLen)   /* Входной:  длина буфера          */

/* Получение значения указанного атрибута инструкции SQL  */
SQLSMALLINT SQLGetStmtAttr(
  SQLINTEGER  stmtHdl, /* Входной:  дескриптор инструкции */
  SQLINTEGER  attrCode,/* Входной:  код атрибута          */
  void        *rtnVal,  /* Выходной: возвращаемое значение */
  SQLINTEGER  bufLen,   /* Входной:  длина буфера */
  SQLINTEGER *strLen)   /* Выходной: указатель на буфер для
                                     реальной длины данных */

/* Установка значения указанного атрибута инструкции SQL  */
SQLSMALLINT SQLSetStmtAttr(
  SQLINTEGER  stmtHdl, /* Входной:  дескриптор инструкции */
  SQLINTEGER  attrCode,/* Входной:  код атрибута          */
  void        *attrVal, /* Входной:  новое значение
                                     атрибута              */
  SQLINTEGER  strLen)   /* Входной:  длина буфера          */
```

Окончание рис. 19.28

Хотя в стандарте и описана сложная структура атрибутов, на самом деле их не так уж и много. Единственный определенный стандартом CLI атрибут среды управляет интерпретацией строк, оканчивающихся нулевым символом. Единственный атрибут сеанса определяет, будет ли CLI автоматически заполнять описатели параметров при подготовке инструкции SQL к выполнению. Что касается атрибутов инструкции, то они в основном управляют свойствами курсоров. Вероятно, самыми важными из определенных в CLI атрибутов являются дескрипторы четырех описателей, которые могут быть связаны с инструкцией SQL: два описателя параметров и два описателя записей. Вызовы на рис. 19.28 используются для получения и установки этих дескрипторов при обработке инструкций на основе дескрипторов.

ODBC API, на котором изначально основывался стандарт SQL/CLI, включает множество других атрибутов. Например, атрибуты сеанса в ODBC могут определять сеанс как используемый только для выборки данных, но не для их изменения, устанавливать режим асинхронной обработки запросов, задавать время ожидания для запросов, направляемых серверу через указанное подключение, и т.п. Атрибуты среды в ODBC управляют автоматической трансляцией вызовов ODBC, написанных для ранних версий этого API. Атрибуты инструкции в ODBC задают уровень изоляции транзакций, разрешают или запрещают использование курсоров с произвольным доступом и ограничивают количество строк в таблице результатов запроса, которые могут быть сгенерированы в ответ на запрос удаленного приложения.

Информационные функции CLI

Стандарт CLI определяет три специфические функции, которые могут использоваться для получения информации о конкретной реализации CLI. Обычно в специализированных приложениях эти функции не нужны. Они требуются в программах общего назначения (таких, как утилиты формирования запросов или генераторы отчетов), которым необходимо знать точные характеристики используемого программного интерфейса. Объявления этих трех функций приведены на рис. 19.29.

```
/* Получение информации о конкретной реализации CLI        */
SQLSMALLINT SQLGetInfo(
   SQLINTEGER   connHdl,    /* Входной:  дескриптор сеанса */
   SQLSMALLINT  infoType,   /* Входной:  тип информации    */
   void        *infoVal,    /* Выходной: буфер для получения
                                         информации        */
   SQLSMALLINT  bufLen,     /* Входной:  длина буфера      */
   SQLSMALLINT *infoLen)    /* Выходной: реальная длина
                                         информации        */

/* Получение информации о поддержке конкретной функции CLI*/
SQLSMALLINT SQLGetFunctions(
   SQLINTEGER   connHdl,    /* Входной:  дескриптор сеанса */
   SQLSMALLINT  functionId,/* Входной:  идентификатор
                                         функции           */
   SQLSMALLINT *supported) /* Выходной: поддерживается ли
                                         функция           */

/* Получение информации о поддерживаемых типах данных      */
SQLSMALLINT SQLGetTypeInfo(
   SQLINTEGER   stmtHdl,    /* Входной:  дескриптор
                                         инструкции        */
   SQLSMALLINT  dataType)   /* Входной:  запрашиваемый тип
                                         или ALL TYPES     */
```

Рис. 19.29. Информационные функции CLI

Функция `SQLGetInfo()` используется для получения более подробной информации о реализации библиотеки, например максимальная длина имен таблиц и пользователей, поддерживает ли СУБД операции внешнего объединения или транзакции и учитывается ли регистр символов в идентификаторах SQL. Функция `SQLGetFunctions()` позволяет определить, поддерживает ли текущая реализация библиотеки CLI конкретную функцию. Вы задаете константу, соответствующую одной из функций CLI, и получаете значение, указывающее, поддерживается ли эта функция.

Функция `SQLGetTypeInfo()` используется для получения информации о поддержке конкретного типа данных или обо всех типах данных, поддерживаемых библиотекой CLI. Вызов этой функции подобен запросу к системному каталогу на получение информации о типах данных. Функция возвращает курсор, в котором каждая запись содержит информацию об одном типе данных: имя, размер, допускаются ли значения `NULL` и т.п.

ODBC API

Как уже говорилось, первоначально Microsoft разрабатывала свой ODBC API для того, чтобы предоставить разработчикам приложений Windows универсальный интерфейс доступа к базам данных, не зависящий от конкретной СУБД. Эта исходная версия ODBC была положена в основу стандарта SQL/CLI, который в настоящее время является официальным стандартом ANSI/ISO для программных интерфейсов SQL. Одновременно с разработкой спецификации SQL/CLI ODBC API был значительно расширен и модифицирован. Начиная с версии ODBC 3.0, он

соответствует стандарту SQL/CLI, но значительно превосходит его по своим возможностям, представляя собой надмножество SQL/CLI.

В некоторых областях ODBC выходит далеко за рамки SQL/CLI, поскольку цели Microsoft изначально были шире, чем просто создание стандартного интерфейса для доступа к базам данных. Компания Microsoft хотела, чтобы одно приложение Windows могло посредством ODBC одновременно работать с несколькими базами данных. Кроме того, ее задачей была разработка среды, позволяющей производителям СУБД поддерживать ODBC и при этом сохранять свои собственные программные интерфейсы, а кроме того, распространять драйверы ODBC для своих СУБД и при необходимости устанавливать их в клиентских Windows-системах. Все эти возможности обеспечиваются многоуровневой структурой ODBC и включенным в этот API набором специальных управляющих функций.

Структура ODBC

Структура ODBC API изображена на рис. 19.30. Программное обеспечение ODBC состоит из трех основных уровней.

- **Интерфейс вызовов функций**. На самом верхнем уровне ODBC находится единый программный интерфейс, который может использоваться всеми приложениями. Этот API реализован в виде динамически компонуемой библиотеки (DLL), которая является неотъемлемой частью операционной системы Windows.
- **Драйверы ODBC**. На нижнем уровне располагается набор драйверов — каждой поддерживаемой СУБД предназначается свой драйвер. Задачей драйвера является трансляция стандартных вызовов функций ODBC в вызовы соответствующих функций, поддерживаемых конкретной СУБД (их может быть и несколько для одной функции ODBC). Каждый драйвер устанавливается в операционной системе независимо. Это позволяет производителям СУБД разрабатывать для своих продуктов собственные драйверы и распространять их независимо от Microsoft. Если СУБД располагается в той же системе, что и драйвер ODBC, то драйвер обычно напрямую вызывает внутренние API-функции СУБД. Если же доступ к базе данных осуществляется по сети, то драйвер может направлять все вызовы в клиентскую часть СУБД, которая будет переадресовывать их на сервер, или самостоятельно управлять сеансом сетевого подключения к удаленной базе данных.
- **Диспетчер драйверов**. Средний уровень занят диспетчером драйверов ODBC, который отвечает за загрузку и выгрузку драйверов по запросам приложений, а также за передачу вызовов функций ODBC, производимых приложениями, соответствующим драйверам для выполнения.

Когда приложению нужно получить доступ к базе данных посредством ODBC, оно выполняет ту же самую последовательность начальных действий, которая предусматривается стандартом SQL/CLI. Прежде всего программа получает дескрипторы среды и сеанса, а затем вызывает функцию `SQLConnect()`, указав конкретный

источник данных, с которым она хочет работать. В ответ на это диспетчер драйверов ODBC анализирует переданную программой информацию о подключении и определяет, какой драйвер ей нужен. При необходимости диспетчер загружает этот драйвер в память (если этот драйвер еще не используется другой программой).

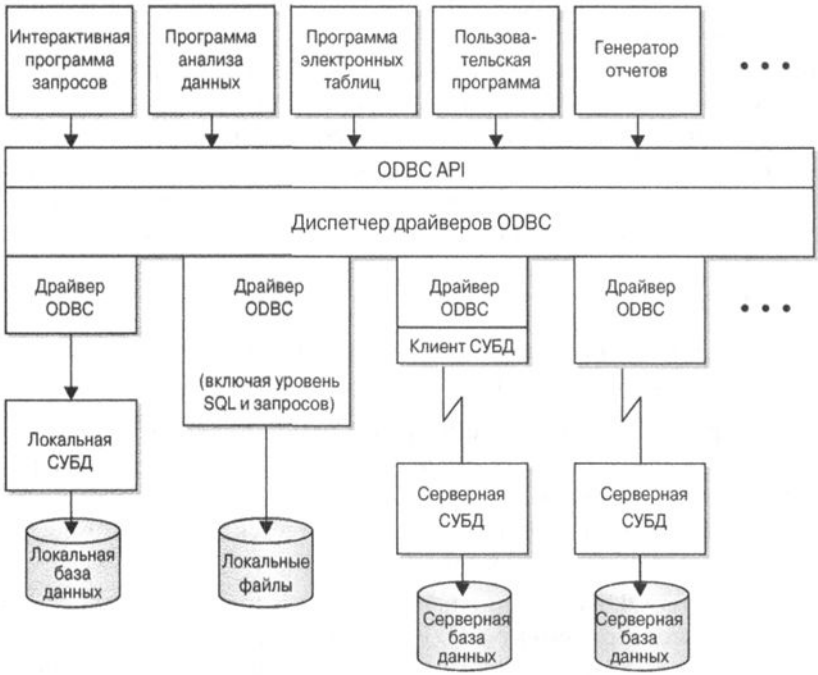


Рис. 19.30. Архитектура ODBC

Все последующие вызовы функций в пределах данного сеанса обрабатываются этим драйвером. Программа может затем вызывать функцию `SQLConnect()` для подключения к другим источникам данных, что вызовет загрузку дополнительных драйверов. После этого программа может использовать ODBC для обращения к нескольким базам данных и различным СУБД посредством единого набора функций.

ODBC и независимость от СУБД

Разработав универсальный программный интерфейс и многоуровневую архитектуру ODBC с диспетчеризацией вызовов, Microsoft значительно продвинулась в направлении независимого от СУБД доступа к базам данных, но сделать доступ абсолютно прозрачным невозможно. Драйверы ODBC для различных СУБД могут легко замаскировать незначительные отличия их программных интерфейсов и диалектов SQL, но более фундаментальные отличия скрыть трудно или даже невозможно. ODBC частично решает эту проблему, разделяя полный набор предлагаемых возможностей на несколько функциональных уровней и вводя набор функций, позволяющих драйверу ODBC рассказать о себе приложению: предоставить ему информацию о своих возможностях, наборе поддерживаемых функций и типов данных. Однако само наличие нескольких уровней функциональных воз-

можностей драйверов и выборочная поддержка драйверами этих возможностей ставят приложение в зависимость от СУБД, хотя и на качественно ином уровне. Поэтому на практике большинство приложений использует только базовые функции ODBC: программисты не хотят без крайней необходимости возиться с особенностями отдельных драйверов.

Функции ODBC для работы с системными каталогами

Одной из областей, в которых ODBC предлагает возможности, выходящие за рамки стандарта SQL/CLI, является получение информации о структуре базы данных из системного каталога. Будучи частью стандарта ANSI/ISO, библиотека CLI предполагает, что эта информация (о таблицах, столбцах, привилегиях и т.п.) доступна через информационную схему, описанную в главе 16, "Системный каталог". ODBC же не полагается на наличие в базе данных информационной схемы. Взамен этот API включает набор специальных функций (табл. 19.4), которые предоставляют приложению информацию о структуре источника данных. Вызывая эти функции, приложение может в процессе своего выполнения получать информацию о таблицах, столбцах, привилегиях, первичных и внешних ключах и хранимых процедурах, составляющих структуру источника данных. Однако для обеспечения безопасности эти функции возвращают информацию только о тех объектах, к которым данный пользователь имеет доступ.

Таблица 19.4. Функции ODBC для работы с системными каталогами

Функция	Описание
`SQLTables()`	Возвращает список каталогов, схем, таблиц или типов таблиц источника данных
`SQLColumns()`	Возвращает список имен столбцов одной или нескольких таблиц
`SQLStatisics()`	Возвращает статистические данные о таблице вместе со списком ее индексов
`SQLSpecialColumns()`	Возвращает список столбцов, которые однозначно идентифицируют строку таблицы; а также список столбцов, которые автоматически обновляются при обновлении строки
`SQLPrimaryKeys()`	Возвращает список столбцов, составляющих первичный ключ заданной таблицы
`SQLForeignKeys()`	Возвращает список внешних ключей заданной таблицы или список внешних ключей других таблиц, которые связаны с заданной таблицей
`SQLTablePrivileges()`	Возвращает список привилегий, связанных с одной или несколькими таблицами
`SQLColumnPrivileges()`	Возвращает список привилегий, связанных с одним или несколькими столбцами одной таблицы
`SQLProcedures()`	Возвращает список хранимых процедур источника данных
`SQLProcedureColumns()`	Возвращает список входных и выходных параметров, возвращаемое значение и имена столбцов результирующего множества хранимой процедуры
`SQLGetTypeInfo()`	Возвращает список типов данных SQL, поддерживаемых источником данных

Функции ODBC, предназначенные для работы с системными каталогами, обычно не нужны в специализированных приложениях, а вот программам общего назначения, таким как модули формирования запросов, генераторы отчетов и утилиты анализа данных, без подобных функций просто не обойтись. Эти функции можно вызывать в любое время после подключения к источнику данных. Например, генератор отчетов может вызвать функцию `SQLConnect()` и сразу же вслед за ней — функцию `SQLTables()`, чтобы узнать, какие таблицы имеются в базе данных. Перечень этих таблиц может быть выведен на экран, чтобы пользователь мог выбрать те из них, информация из которых ему нужна для отчета.

Все функции, работающие с каталогами, возвращают информацию в виде таблицы результатов запроса. Для получения этой информации приложения применяют те же методы, которые используются для получения результатов обычных запросов, выполняемых посредством CLI. Например, набор записей, сформированный функцией `SQLTables()`, будет содержать по одной записи для каждой таблицы базы данных, и функция `SQLFetch()` будет автоматически помещать в программные переменные сведения об очередной таблице.

Расширенные возможности ODBC

Как уже упоминалось, ODBC предоставляет дополнительные возможности помимо тех, что определены стандартом SQL/CLI. Многие из них предназначены для повышения производительности приложений, использующих ODBC, за счет минимизации количества вызовов функций ODBC, необходимых для выполнения типичных задач, а также за счет сокращения сетевого трафика при работе с ODBC. Кроме того, ODBC предоставляет ряд возможностей, обеспечивающих большую независимость приложения от конкретной СУБД и помогающих приложению устанавливать соединение с источником данных. Часть дополнительных возможностей ODBC реализована в виде функций, коротко описанных в табл. 19.5. Другие управляются атрибутами инструкций и сеансов. Учтите, что многие из расширенных возможностей ODBC появились только в версии 3.0 этого API и еще не поддерживаются большинством драйверов и приложений.

Таблица 19.5. Некоторые дополнительные функции ODBC

Функция	Описание
`SQLBrowseConnect()`	Возвращает информацию о доступных источниках данных ODBC и атрибутах, необходимых для подключения к каждому из них
`SQLDrivers()`	Возвращает список доступных драйверов и имен их атрибутов
`SQLDriverConnect()`	Расширенная форма функции `SQLConnect()`, предназначенная для передачи дополнительной информации о сеансе подключения
`SQLNumParams()`	Возвращает количество параметров предварительно подготовленной инструкции SQL
`SQLBindParameter()`	Дополняет возможности функции `SQLBindParam()`
`SQLDescribeParam()`	Возвращает информацию о параметре инструкции SQL
`SQLBulkOperations()`	Выполняет пакетные операции

Окончание табл. 19.5

Функция	Описание
`SQLMoreResults()`	Определяет, остались ли еще необработанные записи в таблице результатов запроса
`SQLSetPos()`	Задает позицию курсора в результирующем наборе записей, разрешая приложению выполнять позиционные операции над этим набором
`SQLNativeSQL()`	Возвращает перевод заданной инструкции SQL на диалект SQL той СУБД, с которой ведется работа

Управление подключениями

Две из расширенных возможностей ODBC связаны с организацией подключений. Механизм *просмотра информации о подключении* предназначен для упрощения процесса подключения к источнику данных. В основе этого механизма лежит функция `SQLBrowseConnect()`. Сначала приложение вызывает эту функцию, указывая имя источника данных, и в ответ получает описание необходимых для подключения атрибутов (таких, скажем, как имя пользователя и пароль). Программа собирает нужную информацию (например, запросив ее у пользователя) и передает ее функции `SQLBrowseConnect()`, которая возвращает описание последующих атрибутов. Цикл продолжается до тех пор, пока приложение не предоставит ODBC всю информацию, необходимую для подключения к заданному источнику данных. Как только это будет сделано, соединение будет установлено.

Механизм *группировки подключений* предназначен для более эффективного управления процессами установления/разрыва соединения в среде "клиент/сервер". Когда режим группировки подключений активизирован, ODBC, получив вызов функции `SQLDisconnect()`, не завершает сеанс подключения. Он остается в неактивном состоянии в течение некоторого времени, и если за это время поступит новый вызов функции `SQLConnect()`, ODBC просто активизирует имеющееся подключение (если, конечно, приложению нужен тот же источник данных). Повторное использование подключений позволяет существенно снизить расходы, связанные с многократным входом в серверную систему (и последующим выходом из нее) в приложениях, выполняющих большое число коротких транзакций.

Трансляция диалектов SQL

ODBC API определяет не только набор API-функций, но и стандартный диалект SQL, который является подмножеством стандарта SQL. Драйвер ODBC отвечает за преобразование инструкций этого диалекта в такие инструкции, с которыми может работать конкретный источник данных (например, выполняется модификация литеральных значений даты/времени, кавычек, ключевых слов и т.п.). Функция `SQLNativeSQL()` позволяет приложению увидеть результат этой трансляции. Кроме того, ODBC поддерживает управляющие последовательности символов, позволяющие приложению более точно управлять трансляцией специфических элементов SQL, которые сильно различаются в разных диалектах этого языка (внешние соединения, условия отбора с подстановочными знаками и т.п.).

Асинхронное выполнение функций

Драйвер ODBC может поддерживать асинхронное выполнение функций. Когда приложение вызывает функцию ODBC (обычно это функция, подготавливающая или выполняющая инструкцию SQL) в асинхронном режиме, ODBC запускает ее и тут же возвращает управление программе. Программа может продолжить свою работу и через какое-то время узнать, как завершилась функция. Асинхронное выполнение нескольких функций может осуществляться для разных сеансов или даже в пределах одного сеанса для разных инструкций. В некоторых случаях работа асинхронно выполняемой функции может быть прервана с помощью функции `SQLCancel()`, что дает приложению возможность отменять слишком долго длящиеся операции ODBC.

Эффективное выполнение инструкций

Выполнение каждой функции ODBC, запрашивающей выполнение инструкции SQL, сопряжено со значительными издержками, особенно если подключение к источнику данных осуществляется по сети. Чтобы снизить накладные расходы, драйвер ODBC может поддерживать *пакетное выполнение инструкций*, когда две или более инструкций SQL, переданных приложением, объединяются в пакет, выполняемый за один вызов функции `SQLExecDirect()` или `SQLExecute()`. Например, последовательность из десятка инструкций `INSERT` или `UPDATE` может быть таким образом выполнена как одна транзакция пакетного добавления или обновления. Это значительно уменьшит сетевой трафик в среде "клиент/сервер", но вместе с тем и усложнит выявление и обработку ошибок. Обработка ошибок, происходящих при пакетном выполнении инструкций, выполняется по-разному, в зависимости от конкретного драйвера.

Многие СУБД поддерживают другой способ пакетного выполнения инструкций — хранимые процедуры, находящиеся непосредственно в базе данных. Хранимая процедура может содержать не только последовательность инструкций SQL, но и некоторый дополнительный управляющий код, причем все это выполняется за один вызов процедуры. ODBC API позволяет приложениям непосредственно вызывать хранимые процедуры из заданного источника. Для тех СУБД, которые поддерживают передачу параметров хранимым процедурам по именам, ODBC разрешает указывать имена параметров, а не их порядковые номера. Для тех источников данных, которые предоставляют расширенную информацию о параметрах, функция `SQLDescribeParam()` позволяет приложению на этапе выполнения определять типы параметров. Получить значения выходных параметров хранимых процедур можно с помощью функции `SQLBindParam()` или `SQLGetData()`. Функция `SQLBindParam()` связывает выходные параметры процедуры с программными буферами, которые заполняются после завершения функции `SQLExecDirect()` или `SQLExecute()`. Функция `SQLGetData()` позволяет получать длинные значения строковых параметров по частям.

Другие дополнительные возможности ODBC связаны с многократным выполнением одиночной инструкции SQL (такой, как `INSERT` или `UPDATE`). Методика *смещения привязки* позволяет приложению перед очередным выполнением инструкции указать для привязки ее параметров не абсолютный адрес, а смещение нового адреса

относительно предыдущего. Благодаря этому можно создать массив значений одного параметра и использовать его для выполнения последовательности вызовов инструкции. В общем же случае модификация смещения параметра гораздо удобнее, чем его повторная привязка с помощью функции `SQLBindParam()`.

Массивы параметров в ODBC предоставляют альтернативный метод многократного выполнения одной инструкции с различными параметрами. В данном случае несколько наборов параметров передаются ODBC в одном вызове. Например, если приложению нужно добавить в таблицу несколько записей, оно может запросить выполнение инструкции `INSERT` с параметрами и связать эти параметры с массивами данных. Результат будет таким же, как при выполнении нескольких инструкций `INSERT` по отдельности, но достигаться будет быстрее. ODBC поддерживает массивы параметров двух типов: когда каждая строка массива содержит один набор значений всех параметров (горизонтальный массив) и когда каждая строка массива содержит набор всех значений одного параметра (вертикальный массив).

Эффективная обработка запросов

В среде "клиент/сервер" передача по сети объемных результатов запроса связана с большими издержками. Для сокращения этих издержек драйвер ODBC может поддерживать специальный режим выборки данных, осуществляемой посредством *блочных курсоров*. В этом режиме при каждом вызове функции `SQLFetch()` или `SQLFetchScroll()` может возвращаться несколько строк (называемых *текущим набором строк*) из результирующего набора. Приложение должно связать возвращаемые столбцы с массивами, чтобы в них могли помещаться данные сразу из нескольких записей. Как и в случае параметров, ODBC поддерживает как горизонтальные, так и вертикальные массивы записей. Кроме того, с помощью функции `SQLSetPos()` одну из строк текущего блока записей можно сделать текущей, если вам требуется выполнить позиционное обновление или удаление данных.

Механизм *закладок* в ODBC обеспечивает еще одну возможность повышения эффективности работы приложений, в которых требуется обрабатывать получаемые записи. Закладка — это не зависящий от конкретной СУБД уникальный идентификатор строки, используемый при выполнении инструкций SQL. Для реализации механизма закладок драйвер может использовать первичные ключи или внутренние идентификаторы строк, но для приложения это совершенно не важно. Когда включен режим использования закладок, для каждой строки в таблице результатов запроса создается отдельная закладка-идентификатор. Закладки часто создаются в курсорах с произвольным доступом, для того чтобы можно было многократно возвращаться к определенной строке. Кроме того, они удобны для выполнения позиционных удалений и обновлений.

С их помощью можно также определить, являются ли две записи, возвращенные двумя разными запросами, одной и той же записью или двумя разными записями с одинаковыми значениями данных. Закладки не только облегчают, но и ускоряют выполнение ряда операций (например, позиционное обновление записи, идентифицируемой закладкой, выполняется гораздо быстрее, чем та же операция со сложным условием отбора). Однако создание и обработка закладок — это тоже

немалая работа, и в зависимости от особенностей конкретных СУБД и драйверов она сама может привести к большим накладным расходам. Так что закладками нужно пользоваться осмотрительно.

Закладки составляют основу пакетных операций ODBC — еще одной важной технологии, предназначенной для повышения эффективности работы приложений. Функция `SQLBulkOperations()` позволяет приложению эффективно обновлять, добавлять, удалять или повторно извлекать группы записей, идентифицируя их местоположение закладками. Эта функция используется совместно с блочными курсорами и работает со строками текущего блока записей. Приложение помещает закладки тех записей, которые оно хочет обработать, в отдельный массив, а в другие массивы записывает добавляемые или обновляемые значения. Затем оно вызывает функцию `SQLBulkOperations()`, передав ей код, указывающий, какая операция должна быть выполнена (обновление, удаление, добавление или выборка данных). Таким образом, синтаксис выполнения всех этих операций совершенно иной, чем обычно. Описанный механизм выполнения пакетных операций обновления, удаления и добавления не только очень удобен, но и чрезвычайно эффективен.

Oracle Call Interface (OCI)

Основным программным интерфейсом в Oracle является встроенный SQL. Однако в этой СУБД имеется также альтернативный API, известный как *интерфейс вызовов Oracle* (Oracle Call Interface — OCI). Он существовал много лет, оставаясь практически неизменным, несмотря на появление новых версий Oracle. Лишь с выходом Oracle8 этот интерфейс был подвергнут значительной ревизии, и многие его функции были заменены новыми, улучшенными, версиями. Тем не менее первоначальные функции OCI по-прежнему поддерживаются, так как от них зависит работоспособность десятков тысяч приложений. Новые функции OCI, доступные в версиях Oracle 9*i*, 10*g*, 11*g*, позволяют существенно повысить эффективность новых программ, работающих с Oracle.

"Старый OCI" (из Oracle 7 и более ранних версий) остается только для старых программ, которые были исходно разработаны для его применения. В качестве небольшого справочника избранные функции "старого OCI" приведены в табл. 19.6, так что вы можете сразу распознать программу, которая работает со старой версией OCI. Концептуально эти функции повторяют возможности встроенного динамического SQL, описанного в главе 18, "Динамический SQL".

Новый OCI использует многие концепции стандарта SQL/CLI и ODBC, включая применение дескрипторов для идентификации объектов интерфейса. В API определены несколько сотен функций, полное описание которых выходит за рамки данной книги. В приведенных далее разделах рассматриваются только основные вызовы, используемые большинством прикладных программ.

Таблица 19.6. Функции старого OCI (Oracle 7 и более ранние)

Функция	Описание
Подключение к базе данных /отключение от базы данных	
`olon()`	Регистрирует пользователя в базе данных Oracle
`oopen()`	Открывает курсор (создает подключение) для обработки инструкции SQL
`oclose()`	Закрывает открытый курсор (завершает сеанс)
`ologof()`	Отменяет регистрацию пользователя в базе данных Oracle
Базовая обработка инструкций	
`osql3()`	Подготавливает (компилирует) строку инструкции SQL
`oexec()`	Выполняет ранее скомпилированную инструкцию
`oexn()`	Выполняет инструкцию с параметрами, для которых используется массив связанных переменных
`obreak()`	Прекращает выполнение текущей функции OCI
`oermsg()`	Получает текст сообщения об ошибке
Обработка параметров инструкции	
`obndrv()`	Связывает параметр с программной переменной (по его имени)
`obndrn()`	Связывает параметр с программной переменной (по его номеру)
Обработка транзакций	
`ocom()`	Завершает текущую транзакцию
`orol()`	Отменяет текущую транзакцию
`ocon()`	Включает режим автозавершения (каждая инструкция считается транзакцией)
`ocof()`	Выключает режим автозавершения
Обработка результатов запроса	
`odsc()`	Получает описание столбцов в таблице результатов запроса
`oname()`	Получает имя столбца в таблице результатов запроса
`odefin()`	Связывает столбец таблицы результатов запроса с программной переменной
`ofetch()`	Извлекает следующую строку из таблицы результатов запроса
`ofen()`	Извлекает в массив несколько строк из таблицы результатов запроса
`ocan()`	Отменяет выполнение запроса, не дожидаясь получения всех записей

Дескрипторы OCI

В новом варианте OCI поддерживается иерархия дескрипторов, помогающих управлять взаимодействием между приложением и базой данных, подобная иерархии дескрипторов SQL/CLI, описанной ранее в этой главе. Вот перечень этих дескрипторов.

- **Дескриптор среды**. Дескриптор самого верхнего уровня, связанный со средой взаимодействия между приложением и OCI.
- **Дескриптор контекста службы**. Идентифицирует подключение к серверу Oracle для обработки инструкций.
- **Дескриптор сервера**. Идентифицирует сервер баз данных Oracle (для приложений, поддерживающих одновременно несколько сеансов подключения).
- **Дескриптор сессии**. Идентифицирует активный сеанс подключения к Oracle (для приложений, поддерживающих одновременно несколько сеансов).
- **Дескриптор инструкции**. Идентифицирует обрабатываемую инструкцию SQL.
- **Дескриптор связывания**. Идентифицирует входной параметр инструкции.
- **Дескриптор результирующего столбца**. Идентифицирует столбец таблицы результатов запроса.
- **Дескриптор транзакции**. Идентифицирует выполняемую транзакцию SQL.
- **Дескриптор сложного объекта**. Используется для получения данных из объекта базы данных Oracle.
- **Дескриптор ошибки**. Используется для обработки ошибок OCI.

Приложение управляет дескрипторами OCI посредством функций, описанных в табл. 19.7. Функции создания и освобождения дескрипторов работают так же, как и их аналоги из SQL/CLI. Функции получения и установки значений атрибутов тоже аналогичны функциям SQL/CLI, устанавливающим и считывающим значения атрибутов среды, сеанса и инструкции.

Таблица 19.7. Функции OCI для управления дескрипторами

Функция	Описание
`OCIHandleAlloc()`	Создает дескриптор для использования в программе
`OCIHandleFree()`	Освобождает ранее созданный дескриптор
`OCIAttrGet()`	Возвращает значение заданного атрибута дескриптора
`OCIAttrSet()`	Устанавливает значение заданного атрибута дескриптора

Дескриптор ошибки используется для получения информации от OCI. Как правило, он передается функциям OCI в качестве параметра. Программа проверяет код завершения функции OCI и, если он указывает на наличие ошибки, извлекает информацию об ошибке из определяемой дескриптором диагностической структуры посредством функции `OCIErrorGet()`.

Подключение к серверу Oracle

Процедура инициализации OCI и подключения к базе данных Oracle аналогична процедуре подключения к базе данных в SQL/CLI, ODBC или dblib. Функции OCI, предназначенные для управления подключением, перечислены в табл. 19.8. Прежде всего приложение вызывает функцию OCIInitialize() для инициализации интерфейса. Далее вызывается функция OCIEnvInit() для инициализации дескриптора среды. Как и в ODBC, все взаимодействие программы с сервером осуществляется в контексте среды, идентифицируемом этим дескриптором.

Таблица 19.8. Функции для инициализации OCI и подключения к Oracle

Функция	Описание
OCIInitialize()	Инициализирует OCI
OCIEnvInit()	Инициализирует дескриптор среды для взаимодействия с OCI
OCIConnectionPoolCreate()	Инициализирует пул подключений
OCIConnectionPoolDestroy()	Уничтожает пул подключений
OCILogon()	Подключается к серверу Oracle
OCILogon2()	Получение сеанса (нового или виртуального) из пула подключений
OCILogoff()	Завершает сеанс подключения
OCIServerAttach()	Подключается к серверу Oracle для многосессионных операций
OCIServerDetach()	Отключается от сервера Oracle
OCIServerVersion()	Возвращает информацию о версии сервера
OCISessionBegin()	Начинает новый сеанс с уже подключенным сервером
OCIPasswordChange()	Изменяет пароль пользователя для доступа к серверу
OCISessionEnd()	Завершает начатый ранее пользовательский сеанс

После этих начальных действий большинство приложений вызывает функцию OCILogon() для подключения к серверу Oracle. Последующие функции OCI выполняются в контексте этого сеанса, и для определения привилегий доступа к базе данных Oracle в них используется указанный в функции OCILogon() идентификатор пользователя. Для завершения сеанса вызывается функция OCILogoff(). Другие функции предназначены для более гибкого управления сеансами в приложениях с несколькими потоками и подключениями. Функция OCIServerVersion() позволяет определить версию Oracle, а функция OCIPasswordChange() изменяет пароль текущего пользователя. При использовании пула подключений приложение может вызвать OCIConnectionPoolCreate() для его создания, после чего вызов OCILogon2() создает подключение в этом пуле. Когда пул подключений становится не нужен, его можно уничтожить вызовом OCIConnectionPoolDestroy().

Выполнение инструкций

Функции OCI, перечисленные в табл. 19.9, используются для выполнения инструкций SQL. Функции `OCIStmtPrepare()` и `OCIStmtExecute()` служат для двухэтапного процесса подготовки и выполнения инструкции. Кроме того, функция `OCIStmtExecute()` может дополнительно использоваться для того, чтобы получить описание таблицы результатов запроса (подобно инструкции `DESCRIBE` встроенного SQL), не выполняя сам запрос, — для этого ей передается специальный флаг. Если же функция `OCIStmtExecute()` выполняется в нормальном режиме, OCI автоматически предоставляет программе описание таблицы результатов запроса. Оно доступно через дескриптор выполняемой инструкции и является одним из его атрибутов.

Таблица 19.9. Функции OCI для обработки инструкций и их параметров

Функция	Описание
`OCIStmtPrepare()`	Подготавливает инструкцию к выполнению
`OCIStmtExecute()`	Выполняет ранее подготовленную инструкцию
`OCIBreak()`	Отменяет текущую выполняемую сервером функцию OCI
`OCIBindbyPos()`	Выполняет привязку параметра по его номеру
`OCIBindbyName()`	Выполняет привязку параметра по его имени
`OCIStmtGetBindInfo()`	Получает имена связанной и индикаторной переменных
`OCIBindArrayOfStruct()`	Выполняет привязку массива для передачи набора значений параметров
`OCIBindDynamic()`	Регистрирует функцию обратного вызова, которая будет выполнять динамическую привязку параметра
`OCIBindObject()`	Предоставляет дополнительную информацию о ранее связанном параметре, имеющем сложный объектный тип данных
`OCIStmtGetPieceInfo()`	Возвращает информацию о значении динамического параметра (или динамически связываемого столбца в таблице результатов запроса), которое потребовалось OCI при выполнении инструкции
`OCIStmtSetPieceInfo()`	Передает OCI информацию (буфер, длина, индикатор значений `NULL` и т.п.) о значении динамического параметра (или динамически связываемого столбца в таблице результатов запроса), которое потребовалось OCI при выполнении инструкции

Функции `OCIBindbyPos()` и `OCIBindbyName()` используются для связывания программных переменных с параметрами инструкции, задаваемыми либо по номерам, либо по именам. Эти функции автоматически создают дескрипторы контекста привязки, но им можно передавать и явно создаваемые дескрипторы. Остальные функции реализуют более сложные механизмы привязки, включая привязку массивов значений параметров и сложных объектных типов данных. OCI поддерживает также динамическую привязку параметров и столбцов таблицы результатов запроса, соответствующую технологии отложенной передачи параметров, поддерживаемой в SQL/CLI и ODBC и описанной ранее в этой главе.

Обработка результатов запроса

Функции OCI, приведенные в табл. 19.10, используются для обработки результатов запроса. Функция `OCIDefineByPos()` связывает программную переменную со столбцом таблицы результатов запроса, идентифицируемым по номеру. (В терминологии OCI этот процесс называется *определением* (define); термин *привязка* (binding) используется для входных параметров.) Другие функции поддерживают динамическую (на этапе выполнения) привязку столбцов, привязку массивов значений столбцов (для многострочных операций) и привязку столбцов, имеющих сложные объектные типы данных. Функция `OCIStmtFetch()` является аналогом инструкции `FETCH` — она извлекает строку из таблицы результатов запроса.

Таблица 19.10. Функции OCI для обработки результатов запроса

Функция	Описание
`OCIStmtFetch()`	Извлекает строку или строки из таблицы результатов запроса
`OCIDefineByPos()`	Выполняет привязку столбца таблицы результатов запроса
`OCIDefineArrayofStruct()`	Задает массив для получения групп строк из таблицы результатов запроса
`OCIDefineDynamic()`	Регистрирует функцию обратного вызова для динамической обработки столбца таблицы результатов запроса
`OCIDefineObject()`	Предоставляет дополнительную информацию о привязке столбца, имеющего сложный объектный тип данных

Управление описателями

В OCI *описатели* (descriptor) используются для хранения информации о параметрах, объектах базы данных Oracle (таблицах, представлениях, хранимых процедурах и т.п.), больших двоичных объектах, идентификаторах записей и других объектах OCI. Описатели служат не только для передачи приложению информации, но и позволяют управлять обработкой перечисленных объектов. В табл. 19.11 показаны функции для работы с описателями. Они создают и освобождают описатели, а также считывают и устанавливают значения отдельных их атрибутов.

Таблица 19.11. Функции OCI для управления описателями

Функция	Описание
`OCIDescriptorAlloc()`	Создает описатель или локатор объекта типа `LOB`
`OCIDescriptorFree()`	Освобождает ранее созданный описатель
`OCIParamGet()`	Возвращает описатель параметра
`OCIParamSet()`	Задает атрибуты описателя параметра

Управление транзакциями

Для управления транзакциями в приложениях Oracle используются функции, описанные в табл. 19.12. Функции `OCITransCommit()` и `OCITransRollback()` служат для завершения и отмены транзакций — они соответствуют инструкциям SQL `COMMIT` и `ROLLBACK`. Другие функции обеспечивают более гибкую и сложную

схему управления транзакциями, включая спецификацию транзакций только для чтения, сериализуемых транзакций, а также контроль над распределенными транзакциями. Функции управления транзакциями получают в качестве параметра дескриптор контекста службы, идентифицирующий текущее подключение.

Таблица 19.12. Функции OCI для управления транзакциями

Функция	Описание
`OCITransCommit()`	Завершает транзакцию
`OCITransRollback()`	Отменяет транзакцию
`OCITransStart()`	Инициализирует определенную транзакцию
`OCITransPrepare()`	Готовит к завершению транзакцию в распределенной среде
`OCITransMultiPrepare()`	Готовит транзакцию с несколькими ветвями в одном вызове
`OCITransForget()`	Отменяет ранее подготовленную транзакцию
`OCITransDetach()`	Отключает распределенную транзакцию

Обработка ошибок

Каждая функция OCI возвращает код состояния, указывающий, успешно ли завершена ее работа. Кроме того, большинство функций OCI принимает в качестве входного параметра дескриптор ошибки. Если в ходе выполнения функции произошла ошибка, информацию о ней можно получить через указанный дескриптор. Для этого после завершения функции приложение вызывает функцию `OCIErrorGet()` и получает подробное описание происшедшего, включая код ошибки и соответствующее текстовое сообщение.

Получение информации из системного каталога

Функция `OCIDescribeAny()` обеспечивает доступ к информации из системного каталога Oracle. Прикладная программа передает этой функции имя таблицы, представления, хранимой процедуры, типа данных или другого объекта, входящего в схему базы данных Oracle. В ответ функция заполняет описатель (идентифицируемый дескриптором) информацией об атрибутах объекта. Эту информацию можно затем получить с помощью функции `OCIAttrGet()`.

Работа с большими объектами

OCI включает большую группу функций (некоторые из них приведены в табл. 19.13), предназначенных для обработки больших объектов Oracle (тип `LOB`), а также больших объектов, хранящихся в файлах, имена которых указываются в столбцах таблиц Oracle. Поскольку большие объекты могут занимать тысячи и миллионы байтов памяти, их обычно нельзя непосредственно связывать с программными буферами. Вместо этого для доступа к таким объектам OCI использует *локаторы* `LOB`, которые функционируют как дескрипторы `LOB`-объектов. Локаторы возвращаются в таблице результатов запроса, а также задаются в качестве

входных параметров при добавлении и обновлении больших объектов. Специальные функции OCI поддерживают обработку объектов типа LOB по частям, позволяя перемещать их между базой данных Oracle и приложением. Эти функции получают в качестве входных параметров один или несколько локаторов LOB.

Таблица 19.13. Функции OCI для работы с большими объектами

Функция	Описание
OCILobRead()	Извлекает фрагмент большого объекта и помещает его в буфер программы
OCILobWrite()	Записывает данные из буфера программы в большой объект
OCILobAppend()	Добавляет данные в конец большого объекта
OCILobErase()	Удаляет данные из большого объекта
OCILobTrim()	Удаляет данные, выходящие за пределы локатора большого объекта
OCILobGetLength()	Возвращает длину большого объекта
OCILobLocatorIsInit()	Проверяет, является ли переданное значение корректным локатором большого объекта
OCILobCopy()	Копирует данные из одного большого объекта в другой
OCILobAssign()	Присваивает локатор одного большого объекта другому
OCILobIsEqual()	Сравнивает два локатора больших объектов
OCILobFileOpen()	Открывает файл, содержащий данные большого объекта
OCILobFileClose()	Закрывает открытый файл большого объекта
OCILobFileCloseAll()	Закрывает все ранее открытые файлы больших объектов
OCILobFileIsOpen()	Проверяет, открыт ли файл большого объекта
OCILobFileGetName()	Возвращает имя файла большого объекта для заданного локатора
OCILobFileSetName()	Задает имя файла в локаторе большого объекта
OCILobFileExists()	Проверяет, существует ли файл большого объекта
OCILobLoadFromFile()	Загружает большой объект из файла

Java Database Connectivity (JDBC)

JDBC представляет собой SQL API для языка программирования Java. JDBC является как официальным стандартом доступа к SQL-базам данных из Java, так и стандартом де-факто. Для языка программирования C разработчики СУБД создавали собственные API до того, как были созданы ODBC или SQL/CLI API. В случае же Java компания Sun Microsystems разработала JDBC API как часть Java API, встроенного в различные версии Java. В результате все основные производители СУБД поддерживают Java посредством JDBC; никакого представляющего хоть какой-то интерес конкурирующего API просто нет.

История и версии JDBC

JDBC API прошел путь от первоначальной версии через несколько основных этапов. JDBC 1.0 предоставлял лишь базовую функциональность, включая диспетчер драйверов для подключения к нескольким СУБД, управление подключениями для доступа к отдельным базам данных, управление инструкциями для передачи SQL-команд в СУБД, а также доступ к результатам запросов из Java.

JDBC 2.0 API и последующие версии расширяли JDBC 1.0 и разделили имеющуюся функциональность на базовую и расширенную. В версии 2.0 были добавлены некоторые новые возможности.

- **Пакетные операции**. Программа на Java может передавать несколько строк данных для вставки или обновления при помощи одного вызова API, повышая тем самым производительность и эффективность приложения.
- **Результирующие множества с произвольным доступом**. Подобно курсорам с произвольным доступом, имеющимся в других API, это новое средство JDBC обеспечивает возможность перемещения вперед и назад по результатам запроса.
- **Обновляемые результирующие множества**. Программа на Java может выполнить обновление в базе данных путем обновления требуемой строки результатов запроса или вставки новой строки в результаты.
- **Пул подключений**. Подключения к базе данных могут совместно использоваться программами Java, снижая тем самым накладные расходы на частые подключения и отключения.
- **Распределенные транзакции**. API предоставляет возможность синхронизации обновлений между несколькими базами данных с применением транзакционного принципа "все или ничего", пересекающего границы баз данных.
- **Источники данных**. Новый тип объекта, инкапсулирующий детали соединения с базой данных, который позволяет прикладному программисту работать без подробных знаний о специфике подключения.
- **Наборы строк**. Абстрагируя результаты запросов, наборы строк (rowset) позволяют обрабатывать эти результаты, даже когда программа отключена от исходной базы данных, а позднее выполнять синхронизацию.
- **Поддержка интерфейса JNDI**. Базы данных и драйверы могут быть именованы и каталогизированы в сетевой каталог и доступны в виде записей этого каталога.

JDBC 3.0 API был завершен и формально анонсирован компанией Sun в феврале 2002 года и вошел в качестве части в Java2 Standard Edition (J2SE) 1.4. Новые возможности в версии 3.0 включают следующее.

- **Объектно-реляционные расширения SQL**. API добавляет поддержку абстрактных типов данных и связанных с ними возможностей, добавленных в стандарт SQL в 1999 году.

- **Точки сохранения**. API допускает частичный откат к определенной точке в средине транзакции.
- **Сохранение курсора**. API позволяет курсору оставаться открытым между транзакциями.
- **Метаданные подготовленных инструкций**. Программы могут получать информацию о подготовленных инструкциях, такую как количество и типы данных параметров и столбцов результатов запроса.

Реализация JDBC и типы драйверов

JDBC предполагает наличие архитектуры драйверов, подобной используемой в стандарте ODBC, на котором он основан. На рис. 19.31 показаны основные составные блоки. Программа на Java подключается к диспетчеру драйверов JDBC посредством JDBC API. Системное программное обеспечение JDBC отвечает за загрузку одного или нескольких драйверов JDBC, обычно по запросу Java-программы. Концептуально каждый драйвер обеспечивает доступ к СУБД одного конкретного вида, используя специфичные для него вызовы API и посылая инструкции SQL, необходимые для выполнения запроса JDBC. Программное обеспечение JDBC распространяется в виде пакета Java, который импортируется в программу на языке программирования Java, использующую JDBC.

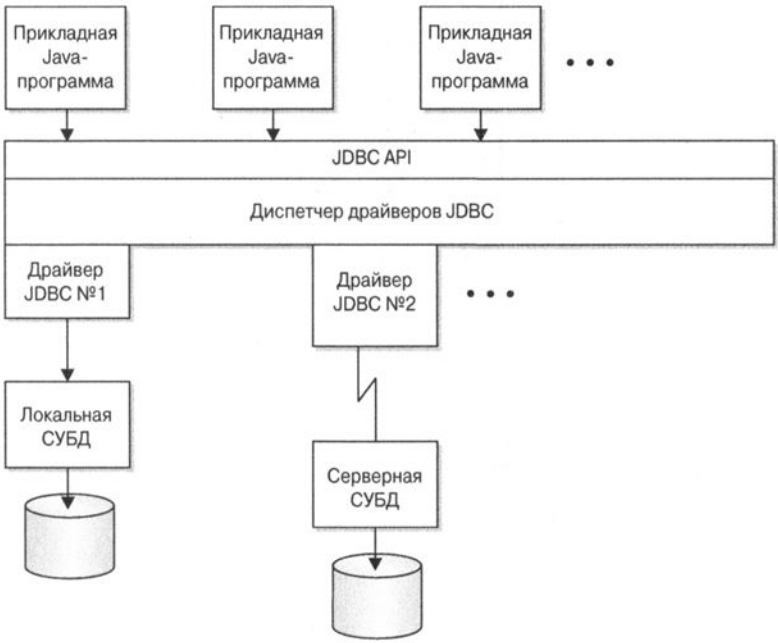


Рис. 19.31. Архитектура JDBC

Спецификация JDBC не рассматривает конкретные детали реализации драйверов JDBC. Однако со времени создания JDBC разработчики обычно подразделяют драйверы JDBC на четыре типа. Описания типов драйверов предполагают наличие

подключения "клиент/сервер" JDBC API (на клиентской системе) к серверу базы данных. Типы драйверов различаются по тому, как они транслируют вызовы JDBC (вызовы методов) в конкретные действия с СУБД.

Драйвер первого типа, именуемый также *мостом JDBC/ODBC*, показан на рис. 19.32. Драйвер преобразует вызовы JDBC в API, независимый от конкретной СУБД (на практике это всегда ODBC). Запрос передается конкретному драйверу ODBC для целевой СУБД. (При этом диспетчер драйверов ODBC может быть (необязательно) устранен, поскольку интерфейс ODBC к диспетчеру драйверов тот же, что и к самому драйверу.) Наконец, драйвер ODBC вызывает интерфейс конкретной СУБД. Если база данных находится в локальной системе, СУБД выполняет запрос. Если же она находится на удаленной системе (сервере), то код клиента представляет собой заглушку сетевого доступа, которая транслирует запрос в сетевое сообщение (специализированное для данной СУБД) и отправляет его на сервер.

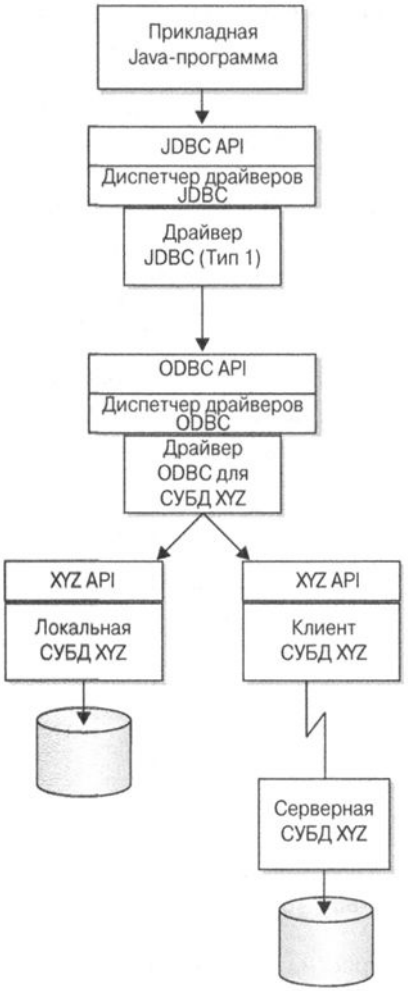


Рис. 19.32. Драйвер JDBC первого типа

Драйвер первого типа имеет одно важное преимущество. Поскольку практически все популярные СУБД поддерживают ODBC, единственный драйвер первого типа в состоянии обеспечить доступ к десяткам различных СУБД. Такие драйвера легкодоступны, включая драйвер, поставляемый компанией Sun.

Драйвер первого типа не лишен и недостатков. Каждый запрос JDBC проходит через множество уровней на пути до СУБД и обратно, так что у такого драйвера накладные расходы оказываются достаточно высоки, а производительность снижена. Использование ODBC в качестве промежуточной стадии, кроме того, может ограничивать функциональность драйвера — некоторые из возможностей СУБД, которые могут быть доступны непосредственно через интерфейс JDBC, через ODBC могут оказаться недоступными. Наконец, драйвер ODBC требует поставки драйвера первого типа в бинарном виде, а не в виде выполнимого файла Java. Таким образом, драйвер первого типа оказывается привязан к аппаратному обеспечению компьютера клиента и его операционной системе, и тем самым теряется важнейшее свойство Java — переносимость.

Драйвер второго типа, именуемый также *драйвером "родного" API*, транслирует запросы JDBC непосредственно в API СУБД, как показано на рис. 19.33. В отличие от драйвера первого типа, в этом случае отсутствует промежуточный уровень ODBC или какой-либо иной, независимый от конкретной СУБД. Если база данных располагается в той же системе, что и программа на языке программирования Java, вызовы API СУБД драйвером второго типа поступают непосредственно к ней. В сети "клиент/сервер" код СУБД на компьютере клиента вновь представляет собой заглушку сетевого доступа, которая транслирует запрос в сетевое сообщение (специализированное для данной СУБД) и отправляет его на сервер, как и при работе драйвера первого типа.

Драйвер второго типа представляет собой иное компромиссное решение, отличающееся от решения первого типа. У драйвера второго типа меньше уровней, так что обычно его производительность выше. Но у него остается присущий драйверу первого типа недостаток, заключающийся в требовании установки бинарного кода в системе клиента, так что каждый драйвер второго типа жестко привязан к конкретной аппаратной архитектуре и операционной системе. В отличие от драйвера первого типа, драйвер второго типа жестко привязан и к конкретной СУБД. Если вы хотите работать с несколькими СУБД разных производителей, вам потребуется соответствующее количество драйверов. Наконец, следует заметить, что различие драйверов первого и второго типов подразумевает то, что ODBC не является "родным" API СУБД. Если у СУБД имеется "родной" интерфейс ODBC, то его использование не требует применения промежуточного уровня, и драйвер второго типа в действительности будет использовать для доступа к СУБД интерфейс ODBC.

Драйвер третьего типа является *нейтральным по отношению к сети*. Этот драйвер транслирует запросы JDBC в сетевые сообщения в независимом от конкретного производителя формате и отправляет их по сети на сервер, как показано на рис. 19.34. На сервере промежуточное программное обеспечение получает сетевые запросы и транслирует их в вызовы API СУБД. Результаты запросов передаются по сети назад опять же в независимом от конкретного производителя формате.

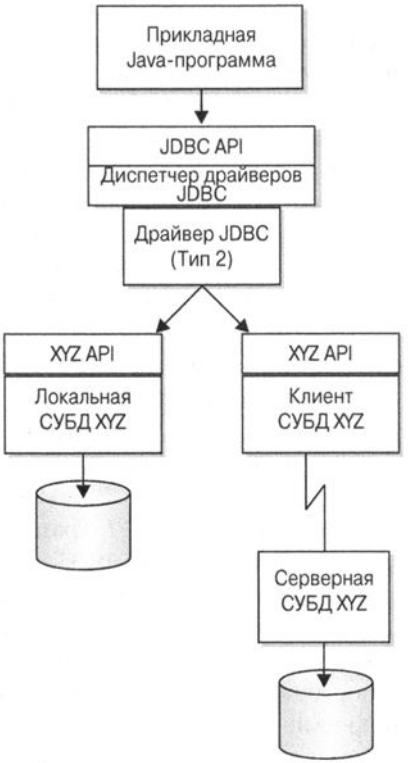


Рис. 19.33. Драйвер JDBC второго типа

Рис. 19.34. Драйвер JDBC третьего типа

Драйвер третьего типа представляет собой еще один компромисс. Основным преимуществом архитектуры третьего типа является то, что код клиента может быть полностью написан на Java, с применением сетевых интерфейсов, предоставляемых другими Java API. Обратите внимание и на независимость клиентского кода от конкретной СУБД; он работает одинаково, какой бы ни была целевая СУБД на сервере. Это означает, что код клиента переносим и способен работать в любой системе, поддерживающей виртуальную машину Java (Java Virtual Machine, JVM) и сетевой API Java. Недостатки драйверов третьего типа те же, что и драйверов первого типа: использование независимого от производителей сетевого уровня, аналогичного независимому уровню ODBC, что в свою очередь означает недоступность некоторых возможностей СУБД. Архитектура третьего типа требует двойной трансляции каждого запроса JDBC, как и при применении первого типа; однако одна из трансляций выполняется на сервере, что снижает нагрузку на систему клиента.

Драйвер четвертого типа является *"родным" сетевым драйвером*, который транслирует запросы JDBC в сетевые сообщения, но в этот раз — в формате, специфичном для конкретной СУБД, как показано на рис. 19.35. Драйвер написан на Java и реализует клиента для сетевого программного обеспечения СУБД, такого как Oracle SQL*Net. Промежуточный уровень на сервере при этом не требуется, так как сервер СУБД уже поддерживает собственную архитектуру "клиент/сервер". Результаты запросов передаются по сети назад также в специфичном для данной СУБД формате и доставляются драйвером запросившей их программе.

Драйверы четвертого типа обладают одним из важных преимуществ драйверов третьего типа, а именно — они могут быть реализованы на чистом Java, так что они, как и драйверы третьего типа, переносимы между различными аппаратными и операционными системами. Однако, в отличие от драйверов третьего типа, они привязаны к конкретным СУБД, поэтому для доступа к СУБД разных производителей требуется свой клиентский код. Архитектура четвертого типа уменьшает накладные расходы и потому имеет несколько более высокую производительность. На практике сетевые накладные расходы почти всегда обесценивают это преимущество, за исключением приложений с очень высоким количеством транзакций.

На рис. 19.36 показаны все четыре типа драйверов JDBC и их взаимоотношения. Два столбца делят типы драйверов по признаку наличия промежуточного уровня (левый столбец) или трансляции вызовов JDBC API непосредственно в интерфейс конкретной СУБД. Две строки делят типы драйверов по тому, где выполняется трансляция в вызовы API конкретной СУБД — на стороне клиента (верхняя строка) или на сервере (нижняя строка). В результате получаются 2×2=4 типа драйверов.

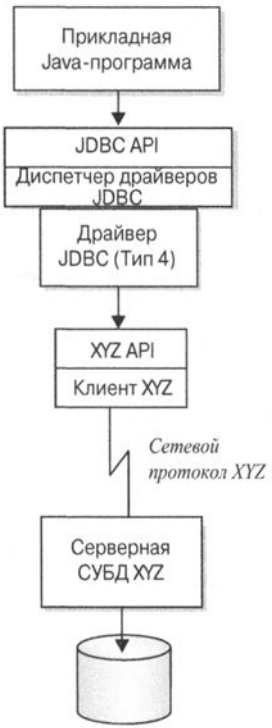


Рис. 19.35. Драйвер JDBC четвертого типа

Размещение API СУБД	Доступ к базе данных: Через нейтральный (ODBC) API	Доступ к базе данных: Через API СУБД
На клиенте	Драйвер первого типа	Драйвер второго типа
На сервере	Драйвер третьего типа	Драйвер четвертого типа

Рис. 19.36. Типы драйверов JDBC

JDBC API

Java представляет собой объектно-ориентированный язык программирования, так что вряд ли окажется сюрпризом то, что JDBC организует функции API как коллекцию объектов для работы с базами данных и предоставляемых ими методов.

- **Объект диспетчера драйверов**. Входная точка JDBC.
- **Объекты подключения**. Представляет индивидуальные активные подключения к целевым базам данных.
- **Объекты инструкций**. Представляет выполняемые инструкции SQL.
- **Объекты результирующего множества**. Представляет результаты SQL-запроса.
- **Объекты метаданных**. Представляют метаданные баз данных, результатов запросов и инструкций.
- **Объекты исключений**. Представляют ошибки при выполнении SQL-инструкций.

Эти объекты логически связаны так, как показано на рис. 19.37. В последующих разделах будут рассмотрены все указанные объекты, будет описано применение их методов для доступа к базам данных, выполнения SQL-инструкций и обработки результатов запросов. Полное описание JDBC API выходит за рамки данной книги, но описанные в ней концепции должны помочь вам эффективно использовать JDBC и разбираться в документации, поставляемой с этим пакетом.

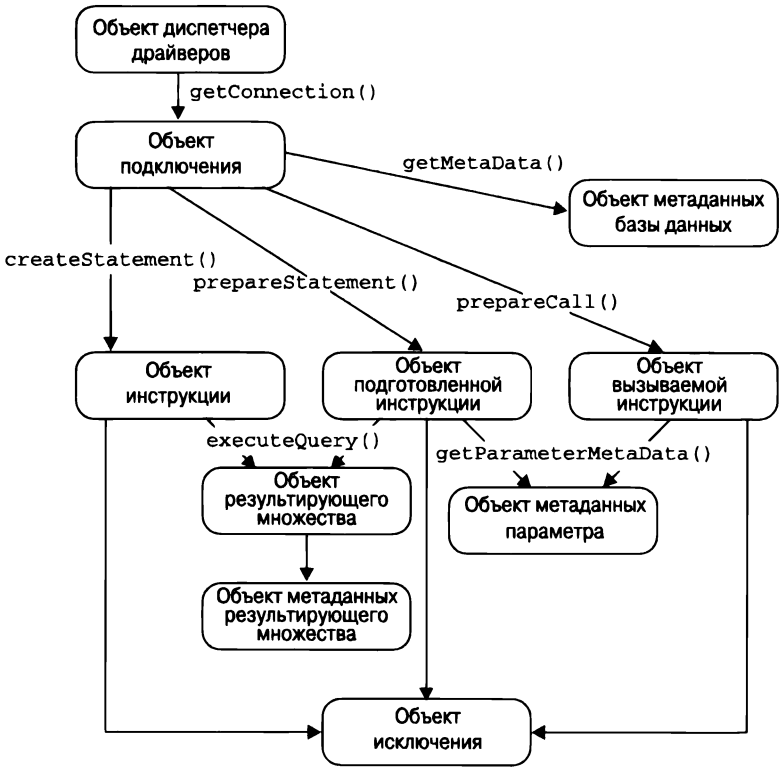


Рис. 19.37. Ключевые объекты JDBC API

Объект `DriverManager` представляет собой основной интерфейс пакета JDBC. Некоторые из наиболее важных его методов показаны в табл. 19.14. После загрузки класса драйвера JDBC, который вы хотите использовать (обычно при помощи

метода Class.forName()), ваша программа запрашивает у объекта DriverManager подключение к определенному драйверу и конкретной базе данных при помощи метода getConnection().

```
// Создание подключения к драйверу Oracle JDBC
String url = "... зависит от конкретной системы и т.п. ..."
String user = "Scott";
String pswd = "Tiger";
Connection dbconn =
    DriverManager.getConnection(url, user, pswd);
```

Таблица 19.14. Методы объекта DriverManager

Метод	Описание
getConnection()	Создает и возвращает объект подключения к базе данных для указанного URL источника данных, а также необязательных имени пользователя, пароля и свойств подключения
registerDriver()	Регистрирует драйвер у диспетчера драйверов JDBC
setLoginTimeout()	Устанавливает тайм-аут для подключения
getLoginTimeout()	Получает значение тайм-аута подключения
setLogWriter()	Обеспечивает трассировку вызовов JDBC

Метод getConnection() возвращает объект Connection, который представляет только что созданное подключение и базу данных на другом конце этого подключения. Другие методы DriverManager обеспечивают программный контроль над тайм-аутами подключения, включением трассировки вызовов JDBC для отладки и выполнением других функций. Если в процессе подключения происходит ошибка, объект DriverManager генерирует исключение. Обработка ошибок описана ниже в данной главе.

Базовая обработка инструкций в JDBC

Основные функции объекта JDBC Connection заключаются в управлении подключением к базе данных, создании SQL-инструкций для базы данных и управления транзакциями данного подключения. В табл. 19.15 показаны методы объекта Connection, выполняющие указанные функции. В большинстве простых программ с использованием JDBC следующим шагом после подключения является вызов метода createStatement() объекта Connection для создания объекта Statement.

Таблица 19.15. Методы объекта Connection

Метод	Описание
close()	Закрывает подключение к источнику данных
createStatement()	Создает объект Statement для подключения
prepareStatement()	Подготавливает параметризованную инструкцию SQL для выполнения в PreparedStatement
prepareStatement()	Подготавливает параметризованную инструкцию SQL для выполнения в PreparedStatement

Окончание табл. 19.15

Метод	Описание
`prepareStatement()`	Подготавливает параметризованную инструкцию SQL для выполнения в `PreparedStatement`
`prepareCall()`	Подготавливает параметризованный вызов хранимой процедуры или функции в `CallableStatement`
`commit()`	Завершает текущую транзакцию подключения
`rollback()`	Выполняет откат текущей транзакции подключения
`setAutoCommit()`	Устанавливает/сбрасывает режим автозавершения транзакций подключения
`getWarnings()`	Получает предупреждения, связанные с подключением
`getMetaData`	Возвращает объект `DatabaseMetaData` с информацией о базе данных

Основная функция объекта `Statement` заключается в выполнении инструкции SQL. В табл. 19.16 приведены методы объекта `Statement`, которые используются для управления выполнением инструкции. Имеется несколько различных методов `execute()`, зависящих от конкретного типа SQL-инструкции. Простая инструкция, не возвращающая результатов запроса (такая, как `UPDATE`, `DELETE`, `INSERT`, `CREATE TABLE`), может использовать метод `executeUpdate()`. Запросы используют метод `executeQuery()`, поскольку он обеспечивает механизм возврата результатов запроса. Прочие методы `execute()` поддерживают подготовленные SQL-инструкции, параметры инструкций и хранимые процедуры.

Таблица 19.16. Методы объекта `Statement`

Метод	Описание
Выполнение инструкций	
`executeUpdate()`	Выполняет инструкцию SQL, не возвращающую результатов, и возвращает количество строк, затронутых ею
`executeQuery()`	Выполняет SQL-запрос, возвращающий единственную строку, и возвращает ее
`execute()`	Выполнение одной или нескольких SQL-инструкций общего назначения
Выполнение пакетных инструкций	
`addBatch()`	Сохраняет ранее переданные значения параметров как часть пакета значений для выполнения
`executeBatch()`	Выполняет последовательность инструкций SQL; возвращает массив целых чисел, указывающих количество строк, на которые воздействовала каждая из инструкций
Ограничение результатов запроса	
`setMaxRows()`	Ограничивает количество строк, получаемых запросом
`getMaxRows()`	Возвращает текущее значение ограничения количества строк
`setMaxFieldSize()`	Ограничивает максимальный размер каждого из получаемых столбцов
`getMaxFieldSize()`	Возвращает текущее ограничение на размер столбцов
`setQueryTimeout()`	Ограничивает максимальное время выполнения запроса
`getQueryTimeout()`	Возвращает текущее максимальное время выполнения запроса
Обработка ошибок	
`getWarnings()`	Получает предупреждения, связанные с выполнением инструкции

Для иллюстрации основ применения объектов Connection и Statement ниже приведена простая программа на языке программирования Java, которая создает подключение к базе данных, выполняет два обновления базы данных, сохраняет внесенные изменения и закрывает соединение.

```
// Объекты подключения и строки, которые будут использованы
// в приведенном фрагменте исходного текста
Connection dbconn;      // Подключение к базе данных
String str1 = "UPDATE OFFICES SET TARGET = 0";
String str2 = "DELETE FROM SALESREPS WHERE EMPL_NUM = 106";

< Код, создающий подключение >

// Создание объекта выполняемой инструкции
Statement stmt = dbconn.createStatement();

// Обновление таблицы OFFICES
stmt.executeUpdate(str1);

// Обновление таблицы SALESREPS
stmt.executeUpdate(str2);

// Сохранение изменений в базе данных
dbconn.commit();

// Обновление таблицы SALESREPS с использованием
// имеющегося объекта инструкции
stmt.executeUpdate(str2);

// Закрытие подключения
dbconn.close();
```

Как показано в приведенном примере, операции SQL, связанные с транзакциями (принятие изменений и откат), обрабатываются вызовами методов объекта Connection, а не посредством выполнения инструкций COMMIT и ROLLBACK. Это позволяет драйверу JDBC получать информацию о состоянии обрабатываемой транзакции без изучения выполняемых SQL-инструкций. JDBC поддерживает также режим автозавершения, когда каждая инструкция рассматривается как отдельная транзакция (этот режим также управляется методом объекта Connection).

Заметим, что методы Connection и Statement, вызываемые в приведенном фрагменте программы, могут вызвать ошибки, хотя в фрагменте нет никакого кода для их обработки. В случае возникновения ошибки драйвер JDBC генерирует исключение SQLException. Обычно фрагмент наподобие приведенного (или его части) размещается в структуре try/catch для обработки возможных исключений. Для простоты охватывающая структура try/catch в этом и приводимых далее примерах не показана. Методы обработки ошибок в JDBC рассматриваются ниже в данной главе.

Обработка простых запросов

Как и в других SQL API и встроенном SQL, обработка запросов требует наличие дополнительного механизма, используемого базой данных для возврата результатов запросов. В JDBC такой дополнительный механизм обеспечивает объект

ResultSet. Для выполнения простого запроса программа на языке программирования Java вызывает метод executeQuery() объекта Statement, передавая ему текст запроса. Метод executeQuery() возвращает объект ResultSet, который включает результаты запроса. Затем Java-программа вызывает методы этого объек-та ResultSet для доступа к результатам запроса, строка за строкой и столбец за столбцом. В табл. 19.17 приведены некоторые из методов объекта ResultSet.

Таблица 19.17. Методы объекта ResultSet

Метод	Описание
Перемещение курсора	
next()	Перемещает курсор к следующей строке результатов запроса
close()	Завершает обработку запроса, закрывает курсор
Получение значений из столбцов	
getInt()	Получение целочисленного значения из указанного столбца
getShort()	Получение короткого целочисленного значения из указанного столбца
getLong()	Получение длинного целочисленного значения из указанного столбца
getFloat()	Получение значения с плавающей точкой из указанного столбца
getDouble()	Получение значения с плавающей точкой двойной точности из указанного столбца
getString()	Получение строкового значения из указанного столбца
getBoolean()	Получение логического значения из указанного столбца
getDate()	Получение значения даты из указанного столбца
getTime()	Получение значения времени из указанного столбца
getTimestamp()	Получение значения временной метки из указанного столбца
getByte()	Получение значения байта из указанного столбца
getBytes()	Получение бинарных данных фиксированной или переменной длины из указанного столбца
getObject()	Получение любого типа данных из указанного столбца
Получение больших объектов	
getAsciiStream()	Получение объекта входного потока для обработки большого символьного объекта (CLOB)
GetBinaryStream()	Получение объекта входного потока для обработки большого бинарного объекта (BLOB)
Другие функции	
getMetaData()	Возвращает объект ResultSetMetaData для данного запроса
getWarnings()	Возвращает предупреждения SQL, связанные с данным ResultSet

Ниже приведен очень простой фрагмент Java-программы, иллюстрирующий, как рассмотренные объекты и методы выполняют обработку простых запросов. В нем запрашивается и выводится на экран номер офиса, город и регион для каждого офиса из таблицы OFFICES.

```
// Объекты, строки и переменные, используемые в фрагменте
Connection dbconn;      // Подключение к базе данных
Int        num;         // Номер офиса
String     city;        // Город
String     reg;         // Регион
String     str1 = "SELECT OFFICE, CITY, REGION FROM OFFICES";

< Код, создающий подключение >

// Создание объекта выполняемого запроса
Statement stmt = dbconn.createStatement();

// Выполнение запроса. Метод возвращает объект ResultSet
ResultSet answer = stmt.executeQuery(str1);

// Цикл по строкам объекта ResultSet
while (answer.next()) {
    // Получение всех столбцов
    num  = answer.getInt("OFFICE");
    city = answer.getString("CITY");
    reg  = answer.getString(3);

    // Вывод строк результатов
    System.out.println(city + " " + num + " " + reg);
}

// Явное закрытие курсора и подключения
answer.close();
dbconn.close();
```

Использованные методы просты и выполняют те же действия, которые мы уже видели во встроенном SQL и в C/C++ API. Объект ResultSet поддерживает курсор, указывающий текущее положение в результатах запроса. Его метод next() перемещает курсор строка за строкой по всему набору строк результатов запроса. Имеется явный метод get, который позволяет получать данные из столбцов каждой строки. Строгая типизация и защита памяти в Java делают такой подход необходимым, но он приводит к гораздо большим накладным расходам по сравнению с подходом C/C++, где выполняется привязка переменных и автоматическое присваивание им соответствующих значений при выборке очередной строки. Наконец, метод close() завершает обработку результатов запроса.

Приведенный пример, кроме того, показывает два альтернативных метода указания столбца, значение которого должно быть получено методом get. Вы можете указать имя столбца (используется для столбцов OFFICE и CITY) или его номер среди столбцов результатов запроса (используется для столбца REGION). JDBC использует для этого перегрузку методов get: одна версия метода принимает строку (имя столбца), а вторая — целое число (его номер).

Использование подготовленных инструкций в JDBC

Методы executeQuery() и executeUpdate() объекта Statement обеспечивают возможности динамического SQL. Они являются аналогами вызова SQLExecDirect() стандарта CLI. База данных на другом конце соединения JDBC не знает заранее, какой SQL-текст будет передан при вызове метода выполнения. Она должна проанализировать инструкцию на лету и определить, как именно следует ее выполнять. Подход динамического SQL делает эту часть интерфейса JDBC достаточно простой в применении, но создает высокие накладные расходы. В критичных по производительности приложениях с большим количеством транзакций лучше использовать альтернативный подход с подготовленными инструкциями.

Этот подход использует те же концепции, что и инструкции PREPARE/EXECUTE встроенного динамического SQL и функции SQLPrepare() и SQLExecute() стандарта CLI. Инструкция SQL, которая должна выполняться многократно (такая, как инструкция UPDATE, которая будет применяться ко множеству строк, или запрос, который будет выполняться в программе сотни раз), сначала *подготавливается* путем передачи ее СУБД для разбора и анализа. Позже эта инструкция может многократно выполняться с малыми накладными расходами. Вы можете изменять конкретные значения, используемые инструкцией при каждом выполнении, передавая новые значения параметров для каждого выполнения. Например, вы можете изменить значения, используемые каждой операцией UPDATE, или изменить значение, с которым выполняется сравнение в условии отбора в предложении WHERE запроса, использующего параметры.

Чтобы воспользоваться подготовленной инструкцией в JDBC, ваша программа вызывает вместо метода createStatement() метод prepareStatement(). В отличие от createStatement(), метод prepareStatement() получает аргумент, а именно — строку, содержащую SQL-инструкцию, которую следует подготовить. В строке инструкции параметры, которые будут переданы при ее выполнении, указываются в виде вопросительных знаков (?), служащих в качестве *маркеров параметров*. Параметр может применяться в инструкции везде, где можно использовать константу. Метод prepareStatement() возвращает объект PreparedStatement, который включает некоторые дополнительные методы, помимо предоставляемых объектом Statement. В табл. 19.18 показаны некоторые из этих дополнительных методов; почти все они служат для работы с параметрами.

Таблица 19.18. Дополнительные методы объекта PreparedStatement

Метод	Описание
setInt()	Устанавливает значение целочисленного параметра
setShort()	Устанавливает значение короткого целочисленного параметра
setLong()	Устанавливает значение длинного целочисленного параметра
setFloat()	Устанавливает значение параметра с плавающей точкой
setDouble()	Устанавливает значение параметра с плавающей точкой двойной точности
setString()	Устанавливает значение строкового параметра

Окончание табл. 19.18

Метод	Описание
setBoolean()	Устанавливает значение параметра типа BOOLEAN
setDate()	Устанавливает значение параметра типа DATE
setTime()	Устанавливает значение параметра типа TIME
setTimestamp()	Устанавливает значение параметра типа TIMESTAMP
setByte()	Устанавливает значение параметра типа BYTE
setBytes()	Устанавливает значение параметра типа BINARY или VARBINARY
setBigDecimal()	Устанавливает значение параметра типа DECIMAL или NUMERIC
setNull()	Устанавливает значение параметра, равное NULL
setObject()	Устанавливает значение параметра произвольного типа
clearParameters	Очищает все значения параметров
getParameterMetaData()	Возвращает объект ParameterMetaData для подготовленной инструкции (только в JDBC 3.0)

Дополнительные методы set() объекта PreparedStatement получают два аргумента. Один указывает номер параметра, для которого передается значение, а второй представляет само значение параметра. При использовании указанных методов типичная последовательность работы с подготавливаемыми инструкциями в JDBC выглядит примерно следующим образом.

1. Java-программа устанавливает соединение с СУБД обычным способом.
2. Программа вызывает метод prepareStatement(), передавая ему текст инструкции, которую следует подготовить и которая включает маркеры параметров. СУБД анализирует инструкцию и создает внутреннее, оптимизированное представление выполняемой инструкции.
3. Позже, когда приходит время выполнить подготовленную инструкцию, программа вызывает один из методов set из табл. 19.18 для каждого из параметров, передавая им необходимые значения.
4. Когда значения всех параметров установлены, программа вызывает метод executeQuery или executeUpdate для выполнения инструкции.
5. Программа повторяет шаги 3 и 4 вновь и вновь (обычно десятки или сотни раз или даже больше), изменяя значения параметров. Если некоторое значение параметра не изменяется от одного выполнения инструкции до другого, соответствующий метод set можно не вызывать.

Вот фрагмент программы, демонстрирующий описанную методику.

```
// Объекты, строки и переменные, используемые в фрагменте
Connection dbconn;          // Подключение к базе данных
String city;                // Город
String str1 = "UPDATE OFFICES SET REGION = ? WHERE MGR = ?";
String str2 = "SELECT CITY FROM OFFICES WHERE REGION = ?";
```

```
< Код, создающий подключение >

// Подготовка инструкции UPDATE
PreparedStatement pstmt1 = dbconn.prepareStatement(str1);

// Подготовка запроса
PreparedStatement pstmt2 = dbconn.prepareStatement(str2);

// Установка параметров инструкции UPDATE и ее выполнение
pstmt1.setString(1,"Central");
pstmt1.setInt(2,108);
pstmt1.executeUpdate();

// Сброс одного из параметров и повторное
// выполнение с сохранением
pstmt1.setInt(2,104);
pstmt1.executeUpdate();
dbconn.commit()

// Установка параметров запроса и его выполнение
pstmt2.setString(1,"Central");
ResultSet answer = pstmt2.executeQuery();

// Цикл по строкам объекта ResultSet
while (answer.next()) {
    // Получение каждого столбца данных
    city = answer.getString(1);
    // Вывод строки результатов
    System.out.println("Город центрального региона "
                       + city);
}
answer.close();

// Установка новых параметров запроса и его выполнение
pstmt2.setString(1,"Eastern");
ResultSet answer = pstmt2.executeQuery();

// Цикл по строкам объекта ResultSet
while (answer.next()) {
    // Получение каждого столбца данных
    city = answer.getString(1);
    // Вывод строки результатов
    System.out.println("Город восточного региона "
                       + city);
}
answer.close();

// Закрываем соединение
dbconn.close();
```

Использование вызываемых инструкций в JDBC

В нескольких последних разделах описана поддержка выполнения динамических инструкций SQL в JDBC (при помощи объекта `Statement`, созданного методом `createStatement()`) и подготовка инициализаций SQL (при помощи объекта `PreparedStatement`, созданного методом `prepareStatement()`). JDBC поддер-

живает также выполнение хранимых процедур и функций при помощи третьего типа объектов инструкций, `CallableStatement`, которые создаются методом `prepareCall()`.

Вот как Java-программа вызывает хранимую функцию или процедуру с использованием JDBC.

1. Java-программа вызывает метод `prepareCall()`, передавая ему SQL-инструкцию, вызывающую хранимую подпрограмму. Параметры для ее вызова указываются маркерами в строке инструкции, так же как и для подготавливаемой инструкции.
2. Этот метод возвращает объект `CallableStatement`.
3. Java-программа использует методы `set()` объекта `CallableStatement` для указания значений параметров для вызова процедуры или функции.
4. Java-программа использует другой метод объекта `CallableStatement` для указания типов данных значений, возвращаемых хранимой процедурой или функцией.
5. Java-программа вызывает один из методов `execute()` объекта `Callable Statement` для вызова хранимой процедуры.
6. Наконец, Java-программа вызывает один или несколько методов `get()` объекта `CallableStatement` для получения значений, возвращаемых хранимой процедурой (если таковые имеются), или возвращаемого значения хранимой функции.

Объект `CallableStatement` предоставляет все методы объекта `Prepared Statement`, перечисленные в табл. 19.16 и 19.18. Его дополнительные методы, используемые для регистрации типов данных выходных параметров и получения значений этих параметров после вызова, показаны в табл. 19.19.

Таблица 19.19. Дополнительные методы объекта `CallableStatement`

Метод	Описание
`registerOutParameter()`	Регистрирует тип данных выходного параметра
`getInt()`	Получает целочисленное возвращаемое значение
`getShort()`	Получает короткое целочисленное значение из определенного столбца
`getLong()`	Получает длинное целочисленное значение из определенного столбца
`getFloat()`	Получает значение с плавающей точкой из определенного столбца
`getDouble()`	Получает значение с плавающей точкой двойной точности из определенного столбца
`getString()`	Получает строковое значение из определенного столбца
`getBoolean()`	Получает логическое значение из определенного столбца
`getDate()`	Получает значение даты из определенного столбца

Окончание табл. 19.19

Метод	Описание
getTime()	Получает значение времени из определенного столбца
getTimestamp()	Получает значение метки времени из определенного столбца
getByte()	Получает значение байта из определенного столбца
getBytes()	Получает данные типа BINARY фиксированной или переменной длины
getBigDecimal()	Получает данные типа DECIMAL или NUMERIC
getObject()	Получает данные произвольного типа

Лучшим способом проиллюстрировать методику вызова хранимой процедуры или функции является небольшой пример. (В этом примере опущено тело хранимой процедуры, так как нас интересует, как ее вызвать, а не что она делает.) Предположим, что наша база данных содержит хранимую процедуру

```
CREATE PROCEDURE CHANGE_REGION
   (IN OFFICE INTEGER,
   OUT OLD_REG VARCHAR(10),
    IN NEW_REG VARCHAR(10));
```

которая изменяет регион офиса (согласно двум входным параметрам) и возвращает старое значение региона, и хранимую функцию

```
CREATE FUNCTION GET_REGION
   (IN OFFICE INTEGER)
      RETURNS VARCHAR(10);
```

которая возвращает регион по заданному номеру офиса. Приведенный фрагмент Java-программы показывает, как вызываются хранимые процедуры и функции с применением JDBC.

```
// Объекты, строки и переменные, используемые в фрагменте
Connection dbconn;            // Подключение к базе данных
String str1 = "{CALL CHANGE_REGION(?, ?, ?)}";
String str2 = "{? = CALL GET_REGION(?)}";
String oldreg;                // Старый регион
String ansreg;                // Текущий регион

< Код, создающий подключение >

// Подготовка двух инструкций
CallableStatement cstmt1 = dbconn.prepareCall( str1 );
CallableStatement cstmt2 = dbconn.prepareCall( str2 );

// Указание значений параметров и типов возвращаемых
// данных для вызова хранимой процедуры
cstmt1.setInt( 1, 12 );          // Офис 12 (Чикаго)
cstmt1.setString(3, "Central"); // Новый центральный регион
cstmt1.registerOutParameter(2,Types.VARCHAR);    // Возврат
                                              // значения varchar

// Вызов хранимой процедуры
```

```
cstmt1.executeUpdate();

oldreg = cstmt.getString(2); // Возвращаемый (второй)
                             // параметр — строка

// Указание значений параметров и типа возвращаемого
// значения для вызова хранимой функции
cstmt2.setInt( 1, 12 );          // Офис 12 (Чикаго)
cstmt2.registerOutParameter(1, Types.VARCHAR); // Возврат
                                       // значения varchar

// Вызов хранимой функции
cstmt2.executeUpdate();

ansreg = cstmt.getString(1); // Возвращаемое значение
                        // (первый параметр) — строка

// Закрываем соединение
dbconn.close();
```

Обратите внимание: строка вызова хранимой процедуры или функции заключена в фигурные скобки. Входные параметры, передаваемые в хранимую процедуру или функцию, обрабатываются точно так же, как и параметры для подготавливаемых инструкций. Выходные параметры хранимой процедуры требуют несколько иного механизма: метод `registerOutParameter()` указывает их типы данных, а вызовы методов `get()` возвращают их значения после завершения вызова. Все эти методы приведены в табл. 19.19. Параметры, являющиеся и входными, и выходными, требуют и передачи значения в вызов процедуры с использованием методов `set()`, и указания выходного типа данных при помощи метода `registerOutParameter()` с последующим получением данных с помощью методов `get()`.

В случае хранимых функций имеются только входные параметры, так что программа вновь использует методы `set()`. Возвращаемое значение указывается при помощи маркера параметра в строке подготавливаемой инструкции. Его тип данных регистрируется, а значение получается точно так же, как и для обычного выходного параметра.

Обработка ошибок в JDBC

Когда в процессе операции JDBC возникает ошибка, интерфейс JDBC генерирует исключение Java. Большинство ошибок выполнения инструкций SQL генерирует исключение `SQLException`. Ошибка может быть обработана при помощи стандартного механизма Java `try/catch`. Когда генерируется исключение `SQLException`, метод `catch()` вызывается с объектом `SQLException`, некоторые методы которого приведены в табл. 19.20.

Методы `SQLException` позволяют получить сообщение об ошибке, код ошибки `SQLSTATE`, а также код ошибки, специфичный для конкретной СУБД. Одна операция JDBC может создать несколько ошибок. В этом случае ошибки доступны программе последовательно. Вызов `getNextException()` при обработке первой ошибки вернет объект `SQLException` для второго исключения и так до тех пор, пока не останется необработанных ошибок.

Таблица 19.20. Методы `SQLException`

Метод	Описание
`getMessage()`	Получает сообщение об ошибке, описывающее исключение
`getSQLState()`	Получает значение `SQLSTATE` (пятисимвольная строка, описанная в главе 17, "Встроенный SQL")
`getErrorCode()`	Получает код ошибки, специфичный для конкретного драйвера или СУБД
`getNextException()`	Переходит к очередному исключению в последовательности

Курсоры произвольного доступа в JDBC

Так же как курсоры с произвольным доступом были внесены в стандарты ANSI/ISO SQL, они были добавлены в последних версиях спецификаций и в результирующие множества JDBC. Вы указываете о своем желании получить результаты запроса с произвольным доступом при помощи параметра метода `executeQuery()`. Если указать необходимость произвольного доступа к результатам, возвращаемый вызовом объект `ResultSet` будет предоставлять вам некоторые дополнительные методы для управления курсором. Наиболее важные из них перечислены в табл. 19.21.

Таблица 19.21. Дополнительные методы для работы с курсором объекта `ResultSet`

Метод	Описание
Перемещение курсора	
`previous()`	Перемещает курсор к предыдущей строке результатов запроса
`beforeFirst()`	Перемещает курсор перед началом результатов запроса
`first()`	Перемещает курсор к первой строке результатов запроса
`last()`	Перемещает курсор к последней строке результатов запроса
`afterLast()`	Перемещает курсор за последнюю строку результатов запроса
`absolute()`	Перемещает курсор к строке результатов запроса с указанным абсолютным номером
`relative()`	Перемещает курсор к строке результатов запроса с указанным относительным номером
Определение местоположения курсора	
`isFirst()`	Определяет, является ли текущая строка первой строкой результирующего множества
`isLast()`	Определяет, является ли текущая строка последней строкой результирующего множества
`isBeforeFirst()`	Определяет, позиционирован ли курсор перед началом результирующего множества
`isAfterLast()`	Определяет, позиционирован ли курсор за концом результирующего множества
`moveToInsertRow()`	Перемещает курсор к "пустой" строке для вставки новых данных
`moveToCurrentRow()`	Возвращает курсор к строке, которая была текущей перед вставкой

Окончание табл. 19.21

Метод	Описание
Обновление столбца текущей строки	
updateInt()	Обновляет целочисленное значение столбца
updateShort()	Обновляет короткое целочисленное значение столбца
updateLong()	Обновляет длинное целочисленное значение столбца
updateFloat()	Обновляет значение столбца с плавающей точкой
updateDouble()	Обновляет значение столбца с плавающей точкой двойной точности
updateString()	Обновляет строковое значение столбца
updateBoolean()	Обновляет логическое значение столбца
updateDate()	Обновляет значение столбца, представляющее собой дату
updateTime()	Обновляет значение столбца, представляющее собой время
updateTimestamp()	Обновляет значение столбца, представляющее собой временную метку
updateByte()	Обновляет байтовое значение столбца
updateBytes()	Обновляет значение столбца фиксированной или переменной длины
updateBigDecimal()	Обновляет значение столбца типа DECIMAL или NUMERIC
updateNull()	Обновляет значение столбца, присваивая ему значение NULL
updateObject()	Обновляет значение столбца произвольного типа

В дополнение к результирующим множествам с произвольным доступом, последние версии спецификаций JDBC содержат поддержку обновимых результирующих множеств. Эта возможность соответствует возможности UPDATE...WHERE CURRENT OF встраиваемого SQL. Она позволяет обновлять определенные столбцы строки в текущей позиции курсора, а также вносить в таблицу новые строки данных через результирующие множества.

Получение метаданных в JDBC

Интерфейс JDBC предоставляет объекты и методы для получения метаданных баз данных, результатов запросов и параметризованных инструкций. Объект JDBC Connection обеспечивает доступ к метаданным базы данных, которую он представляет. Вызов метода getMetaData() возвращает объект DatabaseMetaData, описанный в табл. 19.22. Каждый метод, приведенный в таблице, возвращает результирующее множество, содержащее информацию о типах объектов базы данных: таблиц, столбцов, первичных ключей и т.д. Это результирующее множество может быть обработано с применением обычных методов JDBC. Другие методы доступа предоставляют название базы данных, ее версию и другую подобную информацию.

Таблица 19.22. Методы объекта `DatabaseMetaData` для получения информации о базе данных

Метод	Описание
`getTables()`	Возвращает результирующее множество с информацией о таблицах базы данных
`getColumns()`	Возвращает результирующее множество с информацией об именах и типах данных столбцов данной таблицы
`getPrimaryKeys()`	Возвращает результирующее множество с информацией о первичном ключе данной таблицы
`getProcedures()`	Возвращает результирующее множество с информацией о хранимых процедурах
`getProcedureColumns()`	Возвращает результирующее множество с информацией о параметрах данной хранимой процедуры

Очень полезными могут оказаться и метаданные о результатах запроса. Объект `ResultSet` имеет метод `getMetaData()`, который можно вызвать для получения описания результатов запроса. Этот метод возвращает объект `ResultSetMetaData`, описанный в табл. 19.23. Его методы позволяют определить количество столбцов в результатах запроса, имя и тип данных каждого столбца по его номеру.

Таблица 19.23. Методы объекта `ResultSetMetaData`

Метод	Описание
`getColumnCount()`	Возвращает количество столбцов в результатах запроса
`getColumnName()`	Возвращает имя конкретного столбца в результатах запроса
`getColumnType()`	Возвращает тип данных конкретного столбца в результатах запроса

Наконец, полезны и метаданные о параметрах, используемых в подготовленных SQL-инструкциях или вызовах хранимых процедур. Метод `getParameterMetaData()`, который получает эту информацию, унаследован от объекта `CallableStatement`, который расширяет объект `PreparedStatement`. Метод возвращает объект `ParameterMetaData`, описанный в табл. 19.24. Вызовы методов этого объекта предоставляют информацию о количестве параметров инструкции, их типах данных, являются ли параметры входными или выходными и т.д.

Таблица 19.24. Методы объекта `ParameterMetaData`

Метод	Описание
`getParameterClassName()`	Возвращает имя класса (тип данных) определенного параметра
`getParameterCount()`	Возвращает количество параметров в инструкции
`getParameterMode()`	Возвращает режим работы параметра (`IN`, `OUT`, `INOUT`)
`getParameterType()`	Возвращает тип данных SQL определенного параметра
`getParameterTypeName()`	Возвращает тип данных СУБД определенного параметра
`getPrecision()`	Возвращает точность определенного параметра

Окончание табл. 19.24

Метод	Описание
`getScale()`	Возвращает масштаб определенного параметра
`isNullable()`	Определяет, может ли определенный параметр иметь значение NULL
`isSigned()`	Определяет, может ли определенный параметр быть знаковым числом

Расширенные возможности JDBC

JDBC 2.0 и 3.0 добавляют некоторые возможности, которые расширяют базовую функциональность JDBC для доступа к базам данных. *Источники данных* JDBC, введенные в JDBC 2.0, предоставляют высокоуровневый метод поиска доступных драйверов и баз данных и подключения к ним. Они скрывают детали подключения от прикладного программиста на Java. По сути, источник данных идентифицируется некоторым внешним каталогом, способным транслировать логические имена в конкретные детали подключения. Используя источник данных, прикладной программист может указать целевую базу данных по абстрактному имени, и каталог вместе с программным обеспечением JDBC сами отработают детали подключения.

Наборы строк (rowsets) JDBC представляют собой еще одну концепцию, развитую в новых версиях JDBC. Набор строк расширяет концепцию результирующего множества JDBC, которая представляет множество строк результатов запроса. Помимо самих результатов запроса, набор строк инкапсулирует информацию о базе данных, подключении к ней, имя пользователя и пароль, а также другие сведения. Набор строк хранит всю эту информацию независимо от состояния подключения к базе данных, т.е. он может продолжать существовать и в отключенном состоянии; более того — он может использоваться для повторного подключения к базе данных.

Наборы строк обладают рядом других характеристик и возможностей. Набор строк удовлетворяет требованиям компонента JavaBeans и при подключении к базе данных предоставляет способ сделать результирующее множество соответствующим спецификации Enterprise Java Bean (EJB). Наборы строк хранят табулированные результаты запросов, с которыми можно работать — получать, перемещаться по ним и даже обновлять — независимо от текущего состояния подключения. Если будет выполнено обновление при отключенной базе данных, то при последующем подключении будет выполнена ресинхронизация. Наконец, концепция набора строк необязательно должна быть привязана к SQL и реляционным базам данных. Данные в наборе строк концептуально могут поступать из любого табулированного источника данных, такого как электронные таблицы на персональном компьютере или даже таблица в текстовом документе. Полное рассмотрение наборов строк JDBC выходит за рамки данной книги; если вас интересует эта или иная возможность JDBC, обратитесь к документации по адресу `http://java.sun.com/javase/technologies/database/index.jsp`.

Резюме

Во многих реляционных СУБД для программного доступа к базе данных применяется интерфейс прикладного программирования (API).

- В зависимости от конкретной СУБД и ее истории, API может быть как альтернативой встраиваемому SQL, так и основным способом доступа прикладных программ к базе данных.
- При использовании API обработка запросов, передача параметров, компиляция и выполнение инструкций и другие подобные задачи осуществляются с помощью вызовов функций, благодаря чему программный SQL остается идентичным интерактивному SQL. В случае встроенного SQL эти задачи выполняются специальными инструкциями (`OPEN`, `FETCH`, `CLOSE`, `DESCRIBE`, `PREPARE`, `EXECUTE` и т.д.), которые имеются только в программном SQL.
- ODBC API компании Microsoft является широко распространенным программным интерфейсом, позволяющим приложению не зависеть от особенностей конкретной СУБД. Однако различия между существующими СУБД проявляются в том, что не все функции и возможности ODBC поддерживаются той или иной СУБД.
- Стандарт SQL/CLI был создан на основе ODBC API и совместим с ним на базовом уровне. Этот стандарт описывает набор функций, дополняющих встроенный SQL. Многие производители СУБД поддерживают его, поскольку давно поддерживают ODBC.
- В случае языка программирования Java стандартом де-факто является интерфейс JDBC, поддерживаемый всеми основными СУБД и входящий как API для работы с базами данных в стандарт Java2 Enterprise Edition (J2EE), реализованный всеми основными серверами приложений.
- Важной частью рынка остаются собственные API разных СУБД, в частности, Oracle OCI. Все они предлагают одни и те же базовые возможности, но их расширенные возможности существенно различаются, так же как и детали вызова функций и используемые структуры данных.
- Как правило, производители СУБД прилагают огромные усилия для повышения производительности их собственных API; ODBC и/или SQL/CLI поддерживаются, в основном, "для галочки". Поэтому приложения с высокими требованиями к производительности обычно используют собственные API СУБД (а значит, оказываются жестко привязаны к конкретной СУБД).

VI ЧАСТЬ

SQL сегодня и завтра

Влияние SQL продолжает расширяться вместе с появлением новых возможностей SQL и добавлением новых типов обрабатываемых данных. В главах с 20, "Хранимые процедуры SQL", по 27, "Будущее SQL", рассматриваются некоторые из таких расширяющихся областей SQL. В главе 20, "Хранимые процедуры SQL", описаны хранимые процедуры SQL, которые предоставляют СУБД дополнительные возможности для реализации бизнес-правил и взаимодействия с базой данных. В главе 21, "SQL и хранилища данных", описывается роль SQL в анализе данных и тенденции создания хранилищ данных на основе SQL. В главе 22, "SQL и серверы приложений", рассматривается роль SQL в создании интерактивных веб-сайтов и, в частности, связь с технологией серверов приложений. В главе 23, "Сети и распределенные базы данных", обсуждается применение SQL для создания распределенных баз данных. Взаимодействие SQL и объектно-ориентированных технологий, а также новое поколение объектно-реляционных баз данных являются предметом рассмотрения главы 24, "SQL и объекты". Глава 25, "SQL и XML", посвящена взаимоотношениям SQL и XML, включая рассмотрение развивающихся архитектур веб-служб на основе XML. В главе 26, "Специализированные базы данных", изучаются базы данных специального назначения, включая мобильные и встраиваемые. И, наконец, в главе 27, "Будущее SQL", обсуждаются основные тенденции в продолжающейся эволюции языка SQL и перспективы его развития в ближайшее десятилетие.

Глава 20
Хранимые процедуры SQL

Глава 21
SQL и хранилища данных

Глава 22
SQL и серверы приложений

Глава 23
Сети и распределенные базы данных

Глава 24
SQL и объекты

Глава 25
SQL API

Глава 26
Специализированные базы данных

Глава 27
Будущее SQL

20
ГЛАВА

Хранимые процедуры SQL

Одна из долгосрочных тенденций на рынке баз данных заключается во все возрастающей роли архитектур обработки данных. Дореляционные системы занимались, в основном, хранением и выборкой данных; за навигацию, сортировку и отбор данных и их обработку отвечали прикладные программы. С наступлением эпохи реляционных баз данных и SQL, СУБД приняли на себя и иные задачи. Поиск в базе данных и сортировка были встроены в язык SQL, и теперь эти действия (а также статистическая обработка данных) легли на плечи СУБД. Явная навигация по базе данных перестала быть необходимой. Последующие усовершенствования SQL, такие как первичные и внешние ключи и ограничения на значения, продолжили эту тенденцию, отобрав у прикладных программ заботу о проверке корректности данных и о целостности данных — все, что у них оставалось со времени ранних реализаций SQL. На каждом шаге этого пути СУБД несла все большую ответственность и обеспечивала все более централизованный контроль, снижая возможность повреждения данных из-за ошибки в прикладной программе.

Во многих отделах информационных технологий больших компаний и организаций эта тенденция параллельна организационным тенденциям. Корпоративные базы данных и содержащиеся в них данные начинают рассматриваться как важная корпоративная ценность, так что во многих отделах выделяются группы администраторов баз данных, отвечающие за поддержку базы данных, за определение (а в некоторых случаях и за обновление) содержащихся в ней данных и за обеспечение структурированного доступа к ней. Другие группы в отделе информационных технологий (или где-то в ином подразделении компании) могут разрабатывать прикладные программы, генерировать отчеты, запросы или иную логику доступа к базе данных. В большинстве организаций за обновление данных отвечают прикладные программы и сотрудники, которые их используют. Группа администрирования иногда отвечает за обновление ссылочных данных и помощь в выполнении таких задач, как, например, пакетная загрузка новых данных. Но безопасность базы данных, разрешенные виды доступа и, в общем случае, все, что касается функционирования базы данных, — все это епархия администраторов.

Частью указанной тенденции являются три важные функциональные возможности современных реляционных баз данных уровня предприятия — хранимые процедуры, функции и триггеры. *Хранимые процедуры* могут решать прикладные задачи непосредственно в самой базе данных, например хранимая процедура может реализовать прикладную логику принятия заказа от клиента или перевода денег из одного банка в другой. *Функции* представляют собой хранимые SQL-программы, которые для каждой строки данных возвращают только одно значение. В отличие от хранимых процедур, функции вызываются при обращениях к ним в SQL-инструкциях (которые могут располагаться почти везде, где может использоваться имя столбца). Это делает их идеальным средством для выполнения преобразований и вычислений над данными, выводимыми в результате запросов или используемыми в условиях отбора. Практически во всех реляционных СУБД имеется собственный набор функций общего назначения, так что добавленные пользователями базы данных функции часто называются *пользовательскими*. *Триггеры* используются для автоматического вызова хранимых процедур на основе некоторых возникающих в базе данных условий. Например, триггер может автоматически переводить деньги со сберегательного счета на чековый, когда последний оказывается исчерпан. В этой главе рассматриваются базовые концепции хранимых процедур, функций, триггеров и их реализация в некоторых популярных СУБД.

Хранимые процедуры в популярных СУБД за два последних десятилетия были существенно усовершенствованы. Полное рассмотрение программирования хранимых процедур, функций и триггеров выходит за рамки нашей книги, но рассмотренные здесь концепции и сравнения разных СУБД дадут вам понимание их базовых возможностей и заложат основу для применения их вами в своем программном обеспечении для работы с СУБД. Хранимые процедуры, функции и триггеры по сути превращают SQL в настоящий язык программирования, и в этой главе мы предполагаем, что с основными концепциями программирования вы уже знакомы.

Концепции хранимых процедур

В своей исходной форме SQL не был полноценным языком программирования. Он задумывался и создавался как язык, предназначенный для выполнения операций над базами данных — создания их структуры, ввода и обновления данных — и особенно для выполнения запросов. SQL может использоваться как интерактивный командный язык: пользователь по очереди вводит инструкции SQL с клавиатуры, а СУБД их выполняет. В этом случае последовательность операций над базой данных определяется ее пользователем. Инструкции SQL могут встраиваться в программы, написанные на других языках программирования, например на C или COBOL, и тогда последовательность операций над базой данных определяется приложением.

С появлением хранимых процедур язык SQL обогатился рядом дополнительных базовых возможностей, обычно связанных с языками программирования. Последовательности расширенных инструкций SQL группируются, образуя программы или процедуры SQL (для простоты мы объединяем процедуры, функции и триггеры под одним общим названием и говорим о процедурах SQL). Конкретные детали зависят от реализации языка, но в целом эти возможности можно описать так.

- **Условное выполнение**. Структура `IF...THEN...ELSE` позволяет SQL-процедуре проверить условие и, в зависимости от результата, выполнить различные действия.
- **Циклы**. Цикл `WHILE` или `FOR` либо иная подобная структура позволяет многократно выполнять последовательность инструкций SQL до тех пор, пока не выполнится заданное условие окончания цикла. В некоторые реализации языка SQL включены специальные циклы для прохода по всем строкам в таблице результатов запроса.
- **Блочная структура**. Последовательность инструкций SQL может быть сгруппирована в единый блок и использована в других управляющих конструкциях как единая инструкция.
- **Именованные переменные**. SQL-процедура может сохранить вычисленное, извлеченное из базы данных или полученное любым другим способом значение в переменной, а когда оно понадобится снова — извлечь его из этой переменной.
- **Именованные процедуры**. Последовательность инструкций SQL можно объединить в группу, дать ей имя и назначить формальные входные и выходные параметры, так что получится обычная подпрограмма или функция, какие используются в традиционных языках программирования. Определенную однажды таким образом процедуру можно вызывать по имени, передавая ей нужные значения в качестве входных параметров. Если она является функцией, возвращающей значение, то его можно использовать в выражениях.

Набор элементов, реализующих все эти возможности, составляет язык хранимых процедур (SPL — Stored Procedure Language).

Впервые механизм хранимых процедур был предложен компанией Sybase в ее популярном продукте Sybase SQL Server. (Функции и триггеры появились немного позже, так что и рассматривать их в данной главе мы станем немного позже.) В основном, энтузиазм по поводу их появления был связан с возможностью повышения производительности при работе в архитектуре "клиент/сервер". При отсутствии хранимых процедур каждая запрошенная прикладной программой (работающей на машине клиента) SQL-операция включала пересылку запроса по сети на сервер базы данных и ожидание ответа по той же сети. Если логическая транзакция требовала шесть SQL-операций, требовалось и шесть пересылок информации по сети между клиентом и сервером. С появлением хранимых процедур эта последовательность из шести операций может быть запрограммирована как процедура и сохранена в базе данных. Прикладная программа просто запросит выполнение сохраненной процедуры и подождет результатов. При этом шесть циклов передачи информации по сети превратятся в один — запрос выполнения хранимой процедуры и ответ от нее.

Хранимые процедуры доказали свою применимость в модели "клиент/сервер", и Sybase воспользовалась этим, заняв лидирующее положение в этой области. Но конкуренты не заставили себя долго ждать, и сегодня большинство СУБД уровня

предприятия предоставляет возможности работы с хранимыми процедурами. В корпоративных базах данных преимущества хранимых процедур не ограничиваются повышением производительности при работе в сети. Хранимые процедуры менее присущи иным типам специализированных СУБД, таким как хранилища данных или базы данных в оперативной памяти. Некоторые СУБД моделируют свои структуры SPL на основе конструкций языков программирования C или Pascal, другие пытаются соответствовать стилю инструкций DML и DDL. Oracle же, например, в качестве основы своего SPL (PL/SQL) принял язык программирования Ada, поскольку он был стандартом правительства США. В результате концепция хранимых процедур во всех диалектах SQL одна, но их синтаксис очень отличается.

Простейший пример

Проще всего объяснять основы построения хранимых процедур на примере. Представьте себе процесс добавления сведений о клиенте в нашу демонстрационную базу данных. Он состоит из такой последовательности действий.

1. Получаем идентификатор и имя клиента, лимит кредита и предполагаемый объем продаж, а также закрепленного за ним служащего (и его офис).
2. Добавляем в таблицу CUSTOMERS данные о новом клиенте.
3. Обновляем строку закрепленного за клиентом служащего в таблице SALESREPS, увеличивая его плановый объем продаж на заданную сумму.
4. Обновляем строку назначенного клиенту офиса в таблице OFFICES, увеличивая его плановый объем продаж на заданную сумму.
5. Сохраняем изменения в базе данных, если все они выполнены успешно.

Без использования хранимой процедуры для выполнения всех этих действий потребовалась бы приведенная ниже последовательность инструкций SQL (мы добавляем клиента XYZ Corporation с идентификатором 2137, лимитом кредита $30000 и планируемым на первый год объемом продаж $50000; этот клиент будет работать со служащим Полом Крузом, идентификатор которого 103, из чикагского офиса).

```
INSERT INTO CUSTOMERS (CUST_NUM, COMPANY,
                       CUST_REP, CREDIT_LIMIT)
     VALUES (2317, 'XYZ Corporation', 103, 30000.00);

     UPDATE SALESREPS
        SET QUOTA = QUOTA + 50000.00
      WHERE EMPL_NUM = 103;

     UPDATE OFFICES
        SET TARGET = TARGET + 50000.00
      WHERE CITY = 'Chicago';

     COMMIT;
```

Вся эта работа может выполняться одной заранее созданной хранимой процедурой. Эта процедура (на диалекте PL/SQL СУБД Oracle) представлена на рис. 20.1. Она называется ADD_CUST и принимает шесть параметров: идентификатор и имя клиента, лимит кредита и предполагаемый объем продаж, идентификатор закрепленного за клиентом служащего и название города, в котором расположен назначенный клиенту офис.

```
/* Процедура для добавления данных о новом клиенте     */
create procedure add_cust(
   c_name     in varchar2,  /* Имя клиента              */
   c_num      in integer,   /* Идентификатор клиента    */
   cred_lim   in number,    /* Лимит кредита            */
   tgt_sls    in number,    /* Объем продаж             */
   c_rep      in integer,   /* Идентификатор служащего */
   c_offc     in varchar2)  /* Город расположения офиса*/
as
begin
    /* Добавляем новую строку в таблицу CUSTOMERS      */
    insert into customers (cust_num, company,
                           cust_rep, credit_limit)
         values (c_num, c_name, c_rep, cred_lim);

    /* Обновляем запись в таблице SALESREPS            */
    update salesreps
       set quota = quota + tgt_sls
     where empl_num = c_rep;

    /* Обновляем запись в таблице OFFICES              */
    update offices
       set target = target + tgt_sls
     where city = c_offc;

    /* Завершаем транзакцию                            */
    commit;
end;
```

Рис. 20.1. Пример хранимой процедуры на диалекте PL/SQL

Поскольку процедура хранится прямо в базе данных, ее можно вызвать следующей простой инструкцией, в которой указаны значения всех шести параметров.

```
ADD_CUST('XYZ Corporation', 2317, 30000.00,
         50000.00, 103, 'Chicago');
```

СУБД выполнит указанную вами хранимую процедуру, по очереди выполняя все входящие в нее инструкции. Если процедура ADD_CUST успешно завершит свою работу, все изменения будут зафиксированы в базе данных. В противном случае СУБД вернет код ошибки и сообщение, указывающее, что именно пошло не так.

Использование хранимых процедур

Приведенная на рис. 20.1 процедура иллюстрирует метод использования нескольких базовых элементов, общих для всех диалектов языка хранимых процедур. Практически во всех диалектах определение хранимой процедуры создается с помощью инструкции CREATE PROCEDURE. Дополняющая ее инструкция DROP PROCEDURE

предназначена для удаления из базы данных процедуры, которая больше не нужна. Инструкция CREATE PROCEDURE задает следующие элементы:

- имя хранимой процедуры;
- количество и типы данных ее параметров;
- имена и типы данных всех локальных переменных, используемых процедурой;
- последовательность инструкций, которые выполняются при вызове процедуры.

В следующих разделах подробно описываются все эти элементы и рассказывается о специальных инструкциях SQL, управляющих ходом выполнения хранимых процедур.

Создание хранимой процедуры

Во многих распространенных диалектах SPL для создания хранимой процедуры используется инструкция CREATE PROCEDURE. Эта инструкция назначает новой процедуре имя, по которому в дальнейшем ее можно будет вызывать. Имя процедуры должно соответствовать общим правилам для идентификаторов SQL (процедура на рис. 20.1 названа ADD_CUST). Хранимая процедура может принимать нуль или более параметров (в нашем примере их шесть: C_NAME, C_NUM, CRED_LIM, TGT_SLS, C_REP и C_OFFC). Во всех распространенных диалектах SPL значения параметров указываются в виде разделенного запятыми списка, заключенного в скобки и следующего за именем вызываемой процедуры. Заголовок хранимой процедуры содержит имена параметров и типы их данных. Для параметров хранимых процедур могут использоваться те же типы данных, которые поддерживаются СУБД для столбцов таблиц.

На рис. 20.1 все параметры процедуры являются *входными*, о чем говорят ключевые слова IN, следующие за их именами в заголовке процедуры (сказанное относится к диалекту PL/SQL в Oracle). Когда вызывается процедура, переданные ей аргументы присваиваются ее параметрам и начинается выполнение тела процедуры. Имена параметров могут использоваться в теле процедуры (и, в частности, в составляющих ее стандартных инструкциях SQL) везде, где допускается использование констант. Встретив имя параметра, СУБД подставляет на его место текущее значение этого параметра. На рис. 20.1 параметры используются в инструкциях INSERT и UPDATE для вычисления значений столбцов и в условиях отбора.

В дополнение к входным параметрам, некоторые диалекты SPL поддерживают *выходные* параметры, с помощью которых хранимые процедуры могут возвращать значения, вычисленные в ходе выполнения процедуры. При интерактивном вызове хранимых процедур от выходных параметров мало пользы, а вот если одна хранимая процедура вызывает другую, выходные параметры позволяют им эффективно обмениваться информацией. Некоторые диалекты SPL поддерживают параметры, которые одновременно являются и входными, и выходными, т.е. их значения передаются хранимой процедуре, та их меняет и результирующие значения возвращаются вызывающей процедуре.

На рис. 20.2 приведен еще один вариант процедуры ADD_CUST, написанный на диалекте Sybase Transact-SQL. (Диалект Transact-SQL используется и в Microsoft SQL Server; он мало отличается от исходной версии этого диалекта из Sybase SQL Server, которая послужила основой для обеих линий продуктов — Microsoft и Sybase.) Обратите внимание на отличия Transact-SQL от диалекта Oracle:

- ключевое слово PROCEDURE может быть сокращено до PROC;
- список параметров процедуры не заключен в скобки, а просто следует за именем процедуры;
- все имена параметров начинаются с символа @, как в объявлении процедуры, так и в ее теле;
- отсутствует формальный маркер окончания инструкции — тело процедуры представляет собой единую инструкцию Transact-SQL. Если требуется несколько инструкций, для их группирования используется блочная структура Transact-SQL.

```
/* Процедура для добавления данных о новом клиенте      */
create proc add_cust
   @c_name     varchar(20),  /* Имя клиента             */
   @c_num      integer,      /* Идентификатор клиента   */
   @cred_lim   decimal(9,2), /* Лимит кредита           */
   @tgt_sls    decimal(9,2), /* Объем продаж            */
   @c_rep      integer,      /* Идентификатор служащего */
   @c_offc     varchar(15)   /* Город расположения офиса*/
as
begin
    /* Добавляем новую строку в таблицу CUSTOMERS       */
    insert into customers (cust_num, company, cust_rep,
                           credit_limit)
         values (@c_num, @c_name, @c_rep, @cred_lim)

    /* Обновляем запись в таблице SALESREPS             */
     update salesreps
        set quota = quota + @tgt_sls
      where empl_num = @c_rep

     /* Обновляем запись в таблице OFFICES              */
     update offices
        set target = target + @tgt_sls
      where city = @c_offc

     /* Завершаем транзакцию                            */
     commit trans
end
```

Рис. 20.2. Хранимая процедура на диалекте Transact-SQL

На рис. 20.3 приведен третий вариант процедуры ADD_CUST, написанный на диалекте Informix. Объявление процедуры и ее параметров здесь ближе к диалекту Oracle. В отличие от Transact-SQL, идентификаторы локальных переменных и параметров не содержат никаких специальных символов. Определение процедуры завершается декларацией END PROCEDURE — такой более четкий синтаксис способствует уменьшению количества ошибок.

```
/* Процедура для добавления данных о новом клиенте      */
create procedure add_cust(
   c_name     varchar(20),  /* Имя клиента              */
   c_num      integer,      /* Идентификатор клиента    */
   cred_lim   money(16, 2), /* Лимит кредита            */
   tgt_sls    money(16, 2), /* Объем продаж             */
   c_rep      integer,      /* Идентификатор служащего  */
   c_offc     varchar(15))  /* Город расположения офиса */

    /* Добавляем новую строку в таблицу CUSTOMERS       */
    insert into customers (cust_num, company, cust_rep,
                           credit_limit)
         values (c_num, c_name, c_rep, cred_lim);

    /* Обновляем запись в таблице SALESREPS             */
    update salesreps
       set quota = quota + tgt_sls
     where empl_num = c_rep;

     /* Обновляем запись в таблице OFFICES              */
     update offices
        set target = target + tgt_sls
      where city = c_offc;

     /* Завершаем транзакцию                            */
     commit transaction;
end procedure;
```

Рис. 20.3. Процедура ADD_CUST на диалекте Informix

Во всех диалектах, в которых используется инструкция CREATE PROCEDURE, хранимая процедура может быть удалена соответствующей инструкцией DROP PROCEDURE.

```
DROP PROCEDURE ADD_CUST;
```

Вызов хранимой процедуры

Хранимую процедуру можно вызывать по-разному: из приложения с помощью соответствующей инструкции SQL, из другой хранимой процедуры, а также в интерактивном режиме. Синтаксис вызова хранимых процедур зависит от используемого диалекта.

Разные диалекты SQL используют различный синтаксис вызова процедур. Вот пример вызова процедуры ADD_CUST на диалекте PL/SQL.

```
EXECUTE ADD_CUST('XYZ Corporation', 2317, 30000.00,
                 50000.00, 103, 'Chicago');
```

Передаваемые процедуре параметры указываются в том порядке, в каком они объявлены, в виде списка, заключенного в скобки. При вызове из другой хранимой процедуры или триггера ключевое слово EXECUTE может быть опущено, тогда вся инструкция немного упрощается.

```
ADD_CUST('XYZ Corporation', 2317, 30000.00,
         50000.00, 103, 'Chicago');
```

Процедура может вызываться и с указанием имен параметров; в этом случае параметры могут передаваться в процедуру в любом порядке.

```
EXECUTE ADD_CUST (c_name    = 'XYZ Corporation',
                  c_num     = 2137,
                  cred_lim  = 30000.00,
                  c_offc    = 'Chicago',
                  c_rep     = 103,
                  tgt_sales = 50000.00);
```

На диалекте Transact-SQL вызов этой же процедуры имеет такой вид.

```
EXECUTE ADD_CUST 'XYZ Corporation', 2317, 30000.00,
                 50000.00, 103, 'Chicago';
```

Скобки здесь не нужны — просто перечислите значения параметров через запятую после имени процедуры. Ключевое слово EXECUTE можно сократить до EXEC. Кроме того, задавая параметры, можно явно указать их имена, что позволяет перечислять параметры в любом порядке. Вот альтернативный вызов процедуры ADD_CUST на диалекте Transact-SQL, полностью эквивалентный предыдущему:

```
EXEC ADD_CUST @C_NAME = 'XYZ Corporation',
              @C_NUM = 2317,
           @CRED_LIM = 30000.00,
            @TGT_SLS = 50000.00,
              @C_REP = 103,
             @C_OFFC = 'Chicago';
```

На диалекте Informix та же инструкция будет иметь такой вид.

```
EXECUTE PROCEDURE ADD_CUST ('XYZ Corporation', 2317,
                            30000.00, 50000.00, 103,
                            'Chicago');
```

Здесь вновь параметры заданы в виде списка, разделенного запятыми и заключенного в скобки. Эта форма инструкции может применяться в любом контексте. Например, ее можно использовать для вызова хранимой процедуры из клиентского приложения. А при вызове из другой хранимой процедуры можно использовать еще и такой эквивалентный синтаксис.

```
CALL ADD_CUST ('XYZ Corporation', 2317, 30000.00,
               50000.00, 103, 'Chicago');
```

Переменные хранимых процедур

В хранимых процедурах удобно (а иногда и просто необходимо) объявлять переменные для хранения некоторых промежуточных значений. Эту возможность обеспечивают все диалекты SQL. Обычно переменные объявляются в начале тела процедуры, сразу за заголовком и перед последовательностью составляющих ее инструкций SQL. Для переменных можно использовать все те же типы данных, что и для столбцов таблиц (разумеется, поддерживаемые данной СУБД).

На рис. 20.4 приведен фрагмент несложной хранимой процедуры на диалекте Transact-SQL, которая вычисляет общую стоимость заказов указанного клиента и, в зависимости от того, превысит ли эта сумма $30000, сохраняет в переменной од-

но из двух сообщений. Обратите внимание: диалект Transact-SQL требует, чтобы все имена переменных, как и имена параметров, начинались с символа @. Инструкция DECLARE объявляет локальные переменные процедуры. В данном случае имеются две переменные: типа MONEY и типа VARCHAR.

```
/* Процедура проверки общей стоимости заказов клиента */
create proc chk_tot
   @c_num integer       /* Один входной параметр       */
as

    /* Объявляем две локальные переменные              */
    declare @tot_ord money, @msg_text varchar(30)

begin
    /* Вычисляем общую стоимость заказов клиента       */
    select @tot_ord = sum(amount)
      from orders
     where cust = @c_num

    /* В зависимости от величины суммы заносим
       в переменную соответствующее сообщение          */
    if tot_ord < 30000.00
       select @msg_text = "Большой объем заказов"
    else
       select @msg_text = "Малый объем заказов"

    /* Выполняем дальнейшую обработку сообщения        */
    . . .

end
```

Рис. 20.4. Использование локальных переменных в Transact-SQL

Инструкция SELECT, написанная на диалекте Transact-SQL, выполняет еще одну функцию — присваивает значения переменным. Вот простейший пример использования этой инструкции для занесения текста сообщения в переменную @MSG_TEXT.

```
SELECT @MSG_TEXT = "Большой объем заказов"
```

Занесение в переменную общей стоимости заказов в начале процедуры — более сложный пример. Здесь инструкция SELECT одновременно используется и для выполнения запроса, генерирующего присваиваемое переменной значение.

На рис. 20.5 показана версия той же хранимой процедуры в СУБД Informix. Она имеет несколько следующих отличий:

- локальные переменные объявляются с помощью инструкции DEFINE;
- имена переменных являются обычными идентификаторами SQL, они не начинаются со специального символа;
- для занесения результата запроса в локальную переменную используется специальная инструкция SELECT...INTO;
- чтобы просто присвоить переменной значение, можно воспользоваться инструкцией LET.

```
/* Процедура проверки общей стоимости заказов клиента */
create procedure chk_tot(c_num integer)
    /* Объявляем две локальные переменные              */
    define tot_ord  numeric(16, 2);
    define msg_text varchar(30);

    /* Вычисляем общую стоимость заказов клиента       */
    select sum(amount) into tot_ord
      from orders
     where cust = c_num;

    /* В зависимости от величины суммы заносим
       в переменную соответствующее сообщение          */
    if tot_ord < 30000.00
       let msg_text = "Большой объем заказов"
    else
       let msg_text = "Малый объем заказов"

    /* Выполняем дальнейшую обработку сообщения        */
    . . .

end procedure;
```

Рис. 20.5. Использование локальных переменных в Informix SPL

На рис. 20.6 показана версия той же хранимой процедуры на диалекте Oracle PL/SQL. И здесь есть несколько отличий от диалектов Transact-SQL и Informix:

- инструкция SELECT...INTO имеет тот же вид, что и в хранимых процедурах Informix, и применяется для занесения значений, возвращаемых однострочным запросом, в локальную переменную;
- синтаксис операторов присваивания взят из языка Pascal (:=). Эта пара символов используется в Oracle PL/SQL вместо инструкции LET.

```
/* Процедура проверки общей стоимости заказов клиента */
create procedure chk_tot(c_num in integer)
as
declare
    /* Объявляем две локальные переменные              */
    tot_ord  number(16, 2);
    msg_text varchar(30);

begin
    /* Вычисляем общую стоимость заказов клиента       */
    select sum(amount) into tot_ord
      from orders
     where cust = c_num;

    /* В зависимости от величины суммы, заносим
       в переменную соответствующее сообщение          */
    if tot_ord < 30000.00
       msg_text := 'Большой объем заказов'
    else
       msg_text := 'Малый объем заказов'

    /* Выполняем дальнейшую обработку сообщения        */
    . . .

end;
```

Рис. 20.6. Использование локальных переменных в Oracle PL/SQL

Локальные переменные в хранимых процедурах могут использоваться в качестве источника данных в выражениях SQL везде, где допускаются константы. При вычислении такого выражения имя переменной заменяется ее текущим значением. Кроме того, в инструкциях SQL локальные переменные можно использовать для хранения значений, полученных в результате вычисления выражения или выполнения запроса. Все эти применения переменных были показаны в примерах программ на рис. 20.4–20.6.

Блоки инструкций

Практически во всех хранимых процедурах, кроме самых простых, возникает необходимость объединения некоторой последовательности инструкций в группу, интерпретируемую как одна инструкция. Например, если в вашей процедуре используется конструкция IF...THEN...ELSE, то, скорее всего, вам понадобится возможность группировки инструкций, поскольку большинство диалектов SPL требует, чтобы каждая ветвь условной конструкции состояла только из одной инструкции. Поэтому, если необходимо выполнение в ветви нескольких инструкций, их следует объединить в один блок.

В Transact-SQL структура блока инструкций очень проста.

```
/* Блок инструкций Transact-SQL */
begin
    /* Здесь должна располагаться
       последовательность инструкций SQL */
    ...

end
```

Единственной задачей ключевых слов BEGIN...END является ограничение блока инструкций; они не влияют на область видимости локальных переменных или других объектов базы данных. Процедуры, циклы и другие конструкции Transact-SQL оперируют единственной инструкцией, и поэтому в них очень часто используются блоки, объединяющие несколько инструкций в одну.

В Informix блок не только определяет группу инструкций, но и (необязательно) объявляет локальные переменные и обработчики исключений для этой группы. Вот структура блока инструкций Informix.

```
/* Блок инструкций Informix SPL          */
/* Объявления локальных переменных блока */
define . . .

/* Определение обработчика исключений    */
on exception . . .

/* Последовательность инструкций SQL     */
begin . . .

end
```

Раздел объявления переменных не обязателен. Вы видели его пример в процедуре на рис. 20.5. Необязателен и раздел обработки исключений (о нем рассказывается далее в этой главе). У ключевых слов BEGIN...END та же роль, что и в Transact-SQL: они объединяют последовательность инструкций SQL в единый блок. Между ними не обя-

зательно должно быть несколько инструкций — может быть и одна инструкция, если, к примеру, вы хотите объявить переменные или определить специальную обработку исключений только для нее.

В Informix не требуется столь часто использовать блоки инструкций, как в Transact-SQL. И для циклов, и для условных конструкций предусмотрены маркеры завершения (IF...END IF, WHILE...END WHILE, FOR...END FOR), благодаря чему в них можно включать группы инструкций (завершая каждую из них точкой с запятой). Поэтому явное определение блока инструкций только ради их группировки требуется редко.

В Oracle PL/SQL возможности создания блоков инструкций те же, что и в Informix. Допускается выделять группу инструкций и определять для нее локальные переменные и обработку исключений.

```
/* Блок инструкций Oracle PL/SQL          */
/* Объявления локальных переменных блока */
declare . . .

/* Последовательность инструкций SQL      */
begin . . .

/* Определение обработчика исключений     */
exception . . .

end;
```

Все три раздела блока не обязательны. Часто встречаются простые блоки BEGIN...END или блоки DECLARE...BEGIN...END, состоящие из объявлений переменных и набора инструкций. Как и в Informix, в Oracle предусмотрены специальные маркеры конца цикла и условных конструкций, поэтому в таких конструкциях явное определение блоков инструкций с помощью ключевых слов BEGIN...END не требуется.

Функции

В дополнение к хранимым процедурам, многие диалекты SPL поддерживают *хранимые функции*. От процедуры функция отличается тем, что при каждом вызове возвращает одно значение (некоторые данные, объект, XML-документ), в то время как хранимая процедура может либо вернуть много разных значений, либо ни одного. Поддержка возвращаемых значений сильно варьируется от диалекта SPL к диалекту. Функции обычно используются в качестве выражений столбцов в инструкциях SELECT и таким образом вызываются по одному разу для каждой строки результирующего множества, выполняя вычисления, преобразования данных и другие действия для того, чтобы получить возвращаемое значение столбца. Далее приведен простой пример хранимой функции. Предположим, что вы хотите определить хранимую процедуру, которая для данного идентификатора клиента вычисляет общую стоимость его заказов. Если определить эту процедуру как функцию, то указанная стоимость может быть ее возвращаемым значением.

На рис. 20.7 показан пример такой функции для Oracle PL/SQL. Обратите внимание на предложение RETURN в ее объявлении, которое сообщает СУБД тип данных возвращаемого значения. В большинстве СУБД хранимую функцию можно вызвать в интерактивном режиме, и возвращенное ею значение будет выведено на экран. Ес-

ли же вызвать функцию из хранимой процедуры, то возвращенное ею значение можно использовать в дальнейших вычислениях или сохранить в переменной.

```
/* Возвращает общую стоимость заказов клиента        */
create function get_tot_ords(c_num in number)
         return number
as
/* Объявляем локальную переменную для хранения итога */
declare tot_ord number(16, 2);

begin
    /* Простой запрос, возвращающий итоговую сумму   */
    select sum(amount) into tot_ord
      from orders
     where cust = c_num;

    /* Возвращаем сумму в качестве значения функции  */
    return tot_ord;
end;
```

Рис. 20.7. Хранимая функция в Oracle PL/SQL

Многие диалекты SPL допускают использование хранимых функций в SQL-выражениях. В частности, это верно для диалекта Oracle PL/SQL, поэтому функцию, приведенную на рис. 20.7, можно, например, использовать в условии отбора.

```
SELECT COMPANY, NAME
  FROM CUSTOMERS, SALESREPS
 WHERE CUST_REP = EMPL_NUM
   AND GET_TOT_ORDS(CUST_NUM) > 10000.00;
```

Когда СУБД вычисляет условие отбора для каждой строки в таблице результатов запроса, она использует идентификатор клиента из этой строки в качестве аргумента функции GET_TOT_ORDS и проверяет, не превысило ли значение, возвращаемое этой функцией, $10000. Этот же запрос можно построить иначе, включив в предложение FROM еще и таблицу ORDERS и сгруппировав результат по клиентам и служащим. Многие СУБД выполняют запросы с группировкой более эффективно, чем приведенный выше, так как последний требует обработки таблицы заказов для каждого клиента.

На рис. 20.8 показана версия той же хранимой функции для Informix. Как видите, она мало чем отличается от версии для Oracle.

```
/* Возвращает общую стоимость заказов клиента        */
create function get_tot_ords(c_num integer)
         returning numeric(16, 2)

/* Объявляем локальную переменную для хранения итога */
define tot_ord numeric(16, 2);

begin
    /* Простой запрос, возвращающий итоговую сумму   */
    select sum(amount) into tot_ord
      from orders
     where cust = c_num;

    /* Возвращаем сумму в качестве значения функции  */
    return tot_ord;
end function;
```

Рис. 20.8. Хранимая функция в Informix SPL

Диалект Transact-SQL, используемый в Microsoft SQL Server и Sybase Adaptive Server Enterprise (ASE), поддерживает хранимые (пользовательские) функции, аналогичные показанным на рис. 20.7 и 20.8.

Возврат значений через параметры

Хранимая функция возвращает только одно значение. Однако некоторые диалекты SPL позволяют возвращать из процедуры более одного значения с помощью *выходных параметров*. Выходные параметры перечисляются в списке параметров процедуры так же, как и рассмотренные в предыдущих примерах входные параметры. Однако вместо того чтобы передавать данные *в* процедуру, они возвращают данные *из* хранимой процедуры.

На рис. 20.9 показана хранимая процедура PL/SQL, которая получает идентификатор клиента и возвращает его имя, имя закрепленного за ним служащего и название города, в котором расположен офис служащего. Процедура имеет четыре параметра. Первый параметр, CNUM, — входной; в нем процедуре передается идентификатор клиента. Остальные три — выходные; они используются для передачи запрошенных данных вызывающей процедуре. В этом простом примере инструкция SELECT...INTO помещает извлеченные из таблиц данные прямо в выходные параметры хранимой процедуры. В более сложных процедурах выходные значения могут формироваться в локальных переменных и затем помещаться в выходные параметры с помощью операторов присваивания PL/SQL.

```
/* Возвращает имя клиента, имя обслуживающего его
   служащего и город, в котором расположен офис служащего */
create procedure get_cust_info(c_num  in  integer,
                               c_name out varchar,
                               r_name out varchar,
                               c_offc out varchar)
as
begin
    /* Простой однострочный запрос, возвращающий
       интересующую нас информацию                          */
    select company, name, city
      into c_name, r_name, c_offc
      from customers, salesreps, offices
     where cust_num = c_num
       and empl_num = cust_rep
       and office = rep_office;
end;
```

Рис. 20.9. Хранимая процедура с выходными параметрами в Oracle PL/SQL

Чтобы процедура могла вызвать другую процедуру с выходными параметрами, она должна предоставить для них буферы, куда будут помещены возвращаемые значения, — это могут быть локальные переменные или собственные выходные параметры вызывающей процедуры. Вот анонимный (неименованный) блок Oracle PL/SQL, который вызывает процедуру GET_CUST_INFO, приведенную на рис. 20.9.

```
/* Получаем информацию о клиенте 2111 */
declare the_name varchar(20),
        the_rep  varchar(15),
```

```
        the_city varchar(15);
execute get_cust_info(2111, the_name, the_rep, the_city);
```

Конечно, вряд ли эта процедура будет вызываться с непосредственно заданным идентификатором клиента, но синтаксически такой вызов вполне допустим. Для трех остальных параметров в вызове процедуры указаны переменные, в которые должны быть помещены возвращаемые значения. А вот как *нельзя* вызывать процедуру GET_CUST_INFO, поскольку второй параметр является выходным и не может представлять собой строковый литерал.

```
/* Получаем информацию о клиенте 2111 */
execute get_cust_info(2111, 'XYZ Co', the_rep, the_city);
```

В дополнение к входным и выходным параметрам, Oracle поддерживает параметры, которые *одновременно* являются и входными, и выходными (INOUT). Они передаются хранимой процедуре по тем же правилам, что и обычные выходные параметры, но, кроме того, переданные в них значения используются вызываемой процедурой как входные данные.

На рис. 20.10 приведена версия процедуры GET_CUST_INFO, написанная на Transact-SQL. В заголовке этой процедуры выходные параметры объявлены несколько иначе, чем в примере для Oracle, и инструкция SELECT также имеет несколько иную форму, а имена переменных начинаются с "@". Во всем остальном структура процедуры такая же, как и в случае Oracle.

```
/* Возвращает информацию о клиенте по его номеру */
create procedure get_cust_info(
                 @c_num integer,
                 @c_name varchar(20) out,
                 @r_name varchar(15) out,
                 @c_offc varchar(15) out
as
begin
    /* Простой однострочный запрос, возвращающий
       интересующую нас информацию              */
    select @c_name = company,
           @r_name = name,
           @c_offc = city
      from customers, salesreps, offices
     where cust_num = @c_num
       and empl_num = cust_rep
       and office = rep_office
end
```

Рис. 20.10. Хранимая процедура с выходными параметрами в Transact-SQL

Когда эта процедура вызывается из другой процедуры Transact-SQL, второй, третий и четвертый параметры должны быть объявлены в *вызывающей* процедуре как выходные, как и в объявлении вызываемой процедуры. Вот как выглядит синтаксис вызова процедуры, приведенной на рис. 20.10, в Transact-SQL.

```
/* Получаем информацию о клиенте 2111 */
declare the_name varchar(20)
declare the_rep  varchar(15)
declare the_city varchar(15)
exec get_cust_info @c_num  = 2111,
```

```
        @c_name = the_name output,
        @r_name = the_rep  output,
        @c_offc = the_city output;
```

На рис. 20.11 показана версия этой же хранимой процедуры в Informix. В этой СУБД выбран другой способ возврата нескольких значений — не через выходные параметры, а через несколько возвращаемых значений функции. Таким образом, в Informix процедура GET_CUST_INFO превращается в функцию. Возвращаемые ею значения определены в предложении RETURNING в заголовке процедуры и возвращаются назад с помощью инструкции RETURN.

```
/* Возвращает информацию о клиенте по его номеру   */
create function get_cust_info(c_num integer)
      returning varchar(20), varchar(15), varchar(15)

    define c_name varchar(20);
    define r_name varchar(15);
    define c_offc varchar(15);

    /* Простой однострочный запрос, возвращающий
       интересующую нас информацию                 */
    select company, name, city
      into c_name, r_name, c_offc
      from customers, salesreps, offices
     where cust_num = c_num
       and empl_num = cust_rep
       and office   = rep_office;

    /* Возвращаем три значения                     */
    return c_name, r_name, c_offc;
end function;
```

Рис. 20.11. Хранимая функция, возвращающая несколько значений, в Informix SPL

Инструкция CALL в Informix, используемая для вызова подобных функций, также содержит специальное предложение RETURNING, предназначенное для приема возвращаемых значений.

```
/* Получаем информацию о клиенте 2111 */
define the_name varchar(20);
define the_rep  varchar(15);
define the_city varchar(15);
call   get_cust_info(2111)
       returning the_name, the_rep, the_city;
```

Как и в Transact-SQL, в Informix имеется версия инструкции CALL, позволяющая передавать параметры по именам.

```
call get_cust_info(c_num = 2111)
     returning the_name, the_rep, the_city;
```

Условное выполнение

Одним из базовых элементов хранимых процедур является конструкция IF...THEN...ELSE, используемая для организации ветвлений внутри процедуры. Давайте снова вернемся к рис. 20.1 с процедурой ADD_CUST, добавляющей в базу дан-

ных информацию о новом клиенте. Предположим, что правила добавления нового клиента изменились и теперь плановый объем продаж служащего не может быть увеличен более чем на некоторую определенную сумму. Если предполагаемая сумма заказов клиента на первый год составляет не более $20000, эта сумма будет добавлена к плану служащего, но если эта сумма больше $20000, к плану будут добавлены фиксированные $20000. Модифицированная версия процедуры приведена на рис. 20.12. Конструкция IF...THEN...ELSE работает точно так же, как и в обычных языках программирования.

```
/* Процедура для добавления данных о новом клиенте  */
create procedure add_cust (
   c_name   in varchar2, /* Имя клиента              */
   c_num    in number,   /* Идентификатор клиента    */
   cred_lim in number,   /* Лимит кредита            */
   tgt_sls  in number,   /* Объем продаж             */
   c_rep    in number,   /* Идентификатор служащего */
   c_offc   in varchar2) /* Город расположения офиса*/
as
begin
    /* Добавляем новую строку в таблицу CUSTOMERS   */
    insert into customers (cust_num, company,
                           cust_rep, credit_limit)
         values (c_num, c_name, c_rep, cred_lim);

    /* Обновляем запись в таблице SALESREPS         */
    if tgt_sls <= 20000.00
    then
        update salesreps
           set quota = quota + tgt_sls
         where empl_num = c_rep;
    else
        update salesreps
           set quota = quota + 20000.00
         where empl_num = c_rep;
    end if;


    /* Обновляем запись в таблице OFFICES           */
    update offices
       set target = target + tgt_sls
     where city = c_offc;

    /* Завершаем транзакцию                         */
    commit;
end;
```

Рис. 20.12. Условные конструкции в хранимых процедурах

Все диалекты SPL допускают создание вложенных инструкций IF для реализации более сложной логики. В некоторых диалектах даже имеются специальные разновидности условных конструкций, позволяющие организовывать множественное ветвление. Предположим, что, в зависимости от того, окажется ли предполагаемая общая стоимость заказов клиента меньше $20000, между $20000 и $50000 или больше $50000, процедура ADD_CUST должна выполнять три разных действия. В Oracle PL/SQL этот алгоритм можно реализовать так.

```
/* Определяем диапазон стоимости заказов    */
if tgt_sls < 20000.00
  then
    /* Обработка наименьшего объема заказов */
    . . .
elsif tgt_sls <= 50000.00
  then
    /* Обработка среднего объема заказов    */
    . . .
else
    /* Обработка максимального объема       */
    . . .
end if;
```

В диалекте Informix допускаются такие же конструкции, только ключевое слово ELSIF меняется на ELIF.

Циклы

Еще одним базовым элементом хранимых процедур практически во всех диалектах является конструкция для многократного выполнения группы инструкций. Проще говоря, это — цикл. Циклы могут быть разными: в зависимости от используемого вами диалекта SPL, могут поддерживаться циклы FOR в стиле Basic со счетчиком итераций (в которых значение целочисленной переменной уменьшается или увеличивается при каждом проходе цикла, пока не достигнет заданного предела) или циклы WHILE в стиле C, в которых условие продолжения цикла вычисляется в начале или конце группы составляющих его инструкций.

Предположим, что мы хотим циклически выполнять некоторую группу инструкций, изменяя значение переменной-счетчика с именем ITEM_NUM в пределах от 1 до 10. Вот как это делается в Oracle PL/SQL.

```
/* Обрабатываем каждый из 10 элементов */
for item_num in 1..10 loop
   /* Обрабатываем текущий элемент     */
   . . .

   /* Проверяем, не следует ли
      завершить цикл раньше            */
   exit when (item_num = special_item);
end loop;
```

Последовательность инструкций, составляющих тело цикла, должна выполниться 10 раз, и при каждом проходе цикла значение переменной ITEM_NUM увеличивается на 1. Предложение EXIT обеспечивает возможность выхода из цикла раньше, причем это можно сделать как по заданному условию, так и безо всякого условия.

А вот тот же пример, написанный на диалекте Informix (обратите внимание на синтаксические отличия и дополнительные возможности этого цикла).

```
/* Обрабатываем каждый из 10 элементов */
for item_num = 1 to 10 step 1
   /* Обрабатываем текущий элемент     */
   . . .
```

```
   /* Проверяем, не следует ли
      завершить цикл раньше            */
   if (item_num = special_item)
       then exit for;
end for;
```

Еще одной распространенной формой цикла является выполнение последовательности инструкций до тех пор, пока остается или пока не станет истинным заданное условие. Ниже дан пример бесконечного цикла в Oracle PL/SQL. Чтобы этот цикл все же когда-нибудь остановился, внутри его тела должна осуществляться проверка условия окончания цикла (в данном случае проверяется равенство двух переменных), и если это условие истинно, должна выполняться команда выхода из цикла.

```
/* Циклически обрабатываем некоторые данные */
loop
   /* Очередной шаг обработки               */
   . . .

   /* Проверяем, не следует ли
      завершить цикл раньше                 */
   exit when (test_value = exit_value);
end loop;
```

Однако чаще условие окончания цикла встраивается прямо в структуру цикла. Тело цикла многократно выполняется до тех пор, пока это условие истинно. Предположим, например, что вы хотите уменьшать плановые объемы продаж для каждого офиса до тех пор, пока их сумма по всей компании не станет меньше $2400000. План каждого офиса должен быть уменьшен на сумму, кратную $10000. Вот (не особенно эффективный) цикл из хранимой процедуры для решения этой задачи, написанный на Transact-SQL.

```
/* Уменьшаем плановые объемы продаж до тех пор,
   пока их сумма не станет меньше $2400000 */
while (select sum(target) from offices) < 2400000.00
   begin
      update offices
         set target = target - 10000.00
   end
```

Блок BEGIN...END в этом цикле не обязателен, поскольку его тело и так состоит из единственной инструкции, но большинство циклов WHILE в Transact-SQL его содержат. Если тело цикла содержит более одной инструкции, блок BEGIN...END обязателен.

А вот версия того же цикла для Oracle PL/SQL.

```
/* Уменьшаем плановые объемы продаж до тех пор,
   пока их сумма не станет меньше $2400000 */
select sum(target) into total_tgt from offices;
while (total_tgt < 2400000.00)
loop
   update offices
      set target = target - 10000.00;
```

```
   select sum(target) into total_tgt from offices;
end loop;
```

Как видите, вложенная в цикл WHILE инструкция SELECT диалекта Transact-SQL заменена инструкцией SELECT...INTO, в которой результат запроса присваивается локальной переменной. При каждом проходе цикла таблица OFFICES обновляется и общая сумма вычисляется заново.

Ниже показана третья версия этого же цикла — для СУБД Informix.

```
/* Уменьшаем плановые объемы продаж до тех пор,
   пока их сумма не станет меньше $2400000   */
select sum(target) into total_tgt from offices
while (total_tgt < 2400000.00);
   update offices
      set target = target - 10000.00;
   select sum(target) into total_tgt from offices;
end while;
```

В различных диалектах SPL используются и другие варианты создания циклов, но их возможности и синтаксис аналогичны описанным в этих трех примерах.

Другие управляющие конструкции

Некоторые диалекты SPL включают дополнительные управляющие конструкции. Например, в Informix инструкция EXIT прерывает нормальное выполнение цикла и передает управление инструкции, следующей непосредственно за циклом. Инструкция CONTINUE также прерывает нормальное выполнение цикла, но вызывает переход к следующей итерации цикла. У обеих инструкций имеется по три формы — для каждого из типов циклов, которые они могут прерывать.

```
exit for;
exit while;
exit foreach;
continue for;
continue while;
continue foreach;
```

В Transact-SQL единственная инструкция BREAK заменяет все три варианта инструкции EXIT; инструкция CONTINUE в этом диалекте тоже только одна. В Oracle инструкция EXIT выполняет ту же функцию, что и в Informix, а инструкция CONTINUE отсутствует.

Еще один способ изменения хода выполнения хранимых процедур — это безусловный переход к метке, выполняемый инструкцией GOTO. В большинстве диалектов метка представляет собой идентификатор, за которым следует двоеточие. Как правило, выход по метке за пределы цикла не допускается, как не допускается и переход внутрь цикла или условной конструкции. Следует помнить, что, как и в структурных языках программирования, использование инструкции GOTO не поощряется, поскольку она затрудняет понимание и отладку программ.

Циклы с курсорами

Одной из самых распространенных ситуаций, в которых требуется циклическое выполнение определенных действий, является построчная обработка результирующего множества, возвращенного некоторым запросом. Во всех основных диалектах SPL для этого предусмотрены специальные конструкции. Концептуально они подобны встраиваемым в клиентские приложения инструкциям DECLARE CURSOR, OPEN CURSOR, FETCH и CLOSE CURSOR встроенного SQL или соответствующим вызовам API-функций. Однако результат запроса в данном случае направляется не приложению, а хранимой процедуре, которая выполняется самой СУБД. Соответственно, результирующие данные оказываются не в переменных клиентского приложения, а в локальных переменных хранимой процедуры.

В качестве иллюстрации предположим, что нам нужно заполнить две таблицы данными из таблицы ORDERS. В первую из них, BIGORDERS, войдут имена клиентов и стоимости заказов на сумму более $10000, а во вторую, SMALLORDERS, — имена служащих и стоимости заказов на сумму менее $1000. Конечно, наиболее эффективным решением этой задачи было бы выполнение двух отдельных инструкций INSERT с подчиненными запросами на выборку, но мы поступим иначе.

1. Выполним запрос на выборку стоимости заказа, имени клиента и имени служащего для каждого заказа.
2. Извлечем стоимость заказа из очередной строки результирующего множества и проверим, попадает ли она в диапазон, соответствующий таблице BIGORDERS или SMALLORDERS.
3. Если обнаружено попадание в диапазон, вставим строку в соответствующую таблицу.
4. Повторим действия, указанные в пп. 2 и 3, для всех строк результирующего множества.
5. Сохраним изменения в базе данных.

На рис. 20.13 приведена хранимая процедура Oracle, выполняющая эти действия. Курсор, определяющий запрос, задан в разделе объявлений процедуры; мы назвали его O_CURSOR. В том же разделе объявлена *строковая переменная* CURS_ROW, используемая для доступа к строкам результирующего множества. Ее тип ROWTYPE (*тип строки*) подобен типу данных struct языка C; он представляет собой набор отдельных значений столбцов. Включив в объявление переменной CURS_ROW имя курсора, мы указали, что элементы структуры ROWTYPE будут иметь те же имена и типы, что и столбцы в таблице результирующего множества.

Созданный нами запрос обрабатывается в специальном цикле FOR. СУБД выполняет этот запрос (по сути, она выполняет эквивалент инструкции OPEN встроенного SQL) до того, как перейдет в тело цикла. Затем в цикле СУБД извлекает следующую запись из таблицы результатов запроса и помещает значения столбцов в переменную CURS_ROW, откуда хранимая процедура может их извлекать. Далее выполняются инструкции в теле цикла. Когда все строки будут обработаны, курсор автоматически закроется и работа хранимой процедуры продолжится с инструкции, следующей за циклом FOR.

```
create procedure sort_orders()
declare
    /* Курсор для запроса                          */
    cursor o_cursor is
    select amount, company, name
      from orders, customers, salesreps
     where cust = cust_num
       and rep = empl_num;

    /* Переменная для хранения получаемых строк */
    curs_row o_cursor%rowtype;

begin

   /* Цикл обработки всех строк в таблице
      результатов запроса                         */
   for curs_row in o_cursor
   loop

      /* Обработка малых заказов                  */
      if (curs_row.amount < 1000.00)
      then insert into smallorders
           values (curs_row.name, curs_row.amount);

      /* Обработка больших заказов                */
      elsif (curs_row.amount > 10000.00)
      then insert into bigorders
           values (curs_row.company, curs_row.amount);
      end if;
   end loop;

   commit;
end;
```

Рис. 20.13. Цикл FOR для работы с курсором в PL/SQL

На рис. 20.14 приведен эквивалент этой же процедуры на диалекте Informix. Здесь результаты запроса помещаются в обычные локальные переменные, а специальный тип данных для хранения всей строки курсора не используется. Инструкция FOREACH многофункциональна. Она определяет запрос, отмечает начало цикла (конец помечается объявлением END FOREACH), выполняет запрос, а также извлекает строки из таблицы результатов запроса, помещая значения столбцов в указанные локальные переменные. Обработка каждой записи происходит в теле цикла FOREACH. Когда все записи обработаны, курсор автоматически закрывается и работа хранимой процедуры продолжается с инструкции, следующей за циклом. Обратите внимание на то, что в этом примере курсору даже не присваивается отдельное имя, поскольку все операции по созданию и обработке курсора интегрированы в инструкцию FOREACH.

В Transact-SQL нет специального цикла FOR, предназначенного для обработки результатов запроса. Однако вы можете использовать инструкции DECLARE CURSOR, OPEN, FETCH и CLOSE, аналогичные тем, которые применяются для работы с курсорами во встроенном SQL. На рис. 20.15 приведена версия процедуры SORT_ORDERS этого диалекта. Обратите внимание на инструкции DECLARE CURSOR, OPEN, FETCH и CLOSE для работы с курсором. Управление циклом обеспечивает системная переменная @@SQLSTATUS, в Transact-SQL являющаяся аналогом переменной SQLSTATE встроенного SQL. Эта переменная получает значение 0, если инструкция FETCH успешно извлекла следующую запись, и ненулевое значение, когда все записи получены.

```
create procedure sort_orders()

    /* Локальные переменные для результатов запроса   */
    define ord_amt numeric(16,2); /* Стоимость заказа */
    define c_name  varchar(20);   /* Имя клиента      */
    define r_name  varchar(15);   /* Имя служащего    */

    /* Выполняем запрос и обрабатываем каждую строку
       в таблице результатов запроса                  */
   foreach select amount, company, name
             into ord_amt, c_name, r_name
             from orders, customers, salesreps
            where cust = cust_num
              and rep  = empl_num;
       begin

         /* Обработка малых заказов       */
         if (ord_amt < 1000.00)
         then insert into smallorders
                  values (r_name, ord_amt);

         /* Обработка больших заказов     */
         elif (ord_amt > 10000.00)
         then insert into bigorders
                   values (c_name, ord_amt);
         end if;
       end;
   end foreach;
end procedure;
```

Рис. 20.14. Цикл FOREACH для работы с курсором в Informix SPL

```
create procedure sort_orders()
as
/* Локальные переменные для хранения результатов     */
declare @ord_amt decimal(16,2); /* Стоимость заказа  */
declare @c_name  varchar(20);   /* Имя клиента       */
declare @r_name  varchar(15);   /* Имя служащего     */

/* Объявляем курсор для запроса                      */
declare o_curs cursor for
        select amount, company, name
          from orders, customers, salesreps
         where cust = cust_num
           and rep  = empl_num

begin

    /* Открываем курсор и извлекаем первую запись    */
    open o_curs
    fetch o_curs into @ord_amt, @c_name, @r_name

    /* Если строк больше нет, выходим из процедуры   */
    if (@@sqlstatus = 2)
    begin
        close o_curs
        return
    end
```

Рис. 20.15. Цикл WHILE для работы с курсором в Transact-SQL

```
        /* Цикл обработки всех строк таблицы результатов */
        while (@@sqlstatus = 0)
        begin

            /* Обработка малых заказов        */
            if (@ord_amt < 1000.00)
            then insert into smallorders
                     values (@r_name, @ord_amt)

            /* Обработка больших заказов      */
            else if (@ord_amt > 10000.00)
            then insert into bigorders
                     values (@c_name, @ord_amt)
        end

        /* Все записи обработаны; закрываем курсор     */
        close o_curs
end
```

Окончание рис. 20.15

Обработка ошибок

Когда приложение работает с базой данных с применением встроенного SQL или SQL API, оно само отвечает за обработку возникающих при этом ошибок. Коды ошибок возвращаются приложению сервером баз данных, а дополнительную информацию можно получить, вызвав дополнительные API-функции или обратившись к специальной структуре, содержащей диагностическую информацию. Если же операции над данными выполняются хранимой процедурой, то она сама должна обрабатывать ошибки.

В Transact-SQL информацию о происшедших ошибках можно получить из специальных системных переменных. Имеется огромное (более 100) количество глобальных системных переменных, хранящих информацию о состоянии сервера и транзакции, открытых подключениях и т.п. Однако для обработки ошибок чаще всего используются только две из них.

- **@@ERROR** — код ошибки, происшедшей при выполнении последней инструкции SQL.
- **@@SQLSTATUS** — состояние последней операции FETCH.

Признаком нормального завершения операции в обеих переменных является значение 0. Другие значения указывают на ошибки или нестандартные ситуации. В хранимых процедурах Transact-SQL глобальные переменные используются точно так же, как и локальные. В частности, их можно применять для организации циклов и ветвления.

В Oracle PL/SQL принят другой стиль обработки ошибок. Эта СУБД предоставляет программистам набор системных исключений — ошибок и предупреждений, которые могут возникать в ходе выполнения инструкций SQL. В хранимой процедуре (или блоке инструкций) может присутствовать раздел EXCEPTION, указывающий СУБД, как ей обрабатывать любые исключительные ситуации, возникающие в этой процедуре (блоке). Всего их в Oracle предусмотрено более десяти. Кроме того, можно создавать и собственные исключения.

В большинстве предыдущих примеров этой главы вообще не выполнялось никакой обработки ошибок. На рис. 20.16 представлена доработанная версия хранимой процедуры, которую мы привели на рис. 20.7. В ней обрабатывается особая ситуация, когда с заданным клиентом не связано ни одного заказа (т.е. когда запрос, вычисляющий стоимость заказов клиента, порождает исключение NO_DATA_FOUND). В ответ приложению, которое вызвало данную хранимую процедуру, возвращается код ошибки (уже не системный, а прикладной) и соответствующее сообщение. Все остальные исключительные ситуации обрабатываются в блоке WHEN OTHERS.

```
/* Возвращает общую стоимость заказов клиента          */
create function get_tot_ords (c_num in number)
         returns number
as

/* Объявляем локальную переменную для хранения итога  */
declare tot_ord number(16, 2);

begin
    /* Простой однострочный запрос, возвращающий
       единственную запись с итоговой суммой           */
    select sum(amount) into tot_ord
      from orders
     where cust = c_num;

    /* Возвращаем из функции полученную сумму          */
    return tot_ord;

exception
    /* Обработка ситуации, когда
       не найдено ни одного заказа                     */
    when no_data_found
    then raise_application_error(-20123,
                                 'Неверный номер клиента');
    /* Обработка остальных исключений                  */
    when others
    then raise_application_error (-20199,
                                  'Неизвестная ошибка');

end;
```

Рис. 20.16. Хранимая функция с обработчиком ошибок в Oracle PL/SQL

Аналогично обрабатываются исключения и в Informix. На рис. 20.17 показана версия хранимой функции для Informix. Инструкция ON EXCEPTION определяет последовательность инструкций, которые должны быть выполнены, если произойдет одно из указанных исключений (перечень кодов исключений приведен в скобках через запятую).

```
/* Возвращает общую стоимость заказов клиента          */
create function get_tot_ords (c_num in integer)
       returning numeric(16, 2)

/* Объявляем локальную переменную для хранения итога  */
define tot_ord  numeric(16, 2);
```

Рис. 20.17. Хранимая функция с обработчиком ошибок в Informix SPL

```
/* Определяем обработчик исключений -123 и -121        */
on exception in (-121, -123)
    /* Выполняем соответствующие исключениям действия */
    . . .
end exception;
on exception in (-121, -123)
    /* Обрабатываем все остальные исключения           */
    . . .
end exception;

begin
    . . .
end function;
```

Окончание рис. 20.17

Преимущества хранимых процедур

Перенос программного кода из клиентских приложений прямо в базу данных в виде хранимых процедур приносит много пользы, причем как администраторам базы данных, так и программистам и конечным пользователям. Вот основные преимущества этого подхода.

- **Производительность**. Многие популярные СУБД компилируют хранимые процедуры (либо автоматически, либо по запросу), создавая их внутреннее представление, которое выполняется гораздо быстрее, чем если бы вы динамически компилировали каждую составляющую их инструкцию SQL.
- **Повторное использование**. После того как хранимая процедура создана, ее можно вызывать из любых приложений, и не нужно снова и снова программировать одни и те же действия, благодаря чему экономится время программиста и уменьшается риск программных ошибок в клиентских приложениях.
- **Сокращение сетевого трафика**. В системах "клиент/сервер" с точки зрения нагрузки на сеть экономнее пересылать между парой компьютеров только запрос на выполнение хранимой процедуры и получать результаты ее работы, чем обрабатывать каждую инструкцию SQL в отдельности. Особенно это важно в тех случаях, когда сетевой трафик и так слишком высок или пропускная способность соединения низкая.
- **Безопасность**. В большинстве СУБД хранимые процедуры считаются защищаемыми объектами, и им назначаются отдельные привилегии. Пользователь, вызывающий хранимую процедуру, должен иметь право на ее выполнение, но не обязательно на доступ к таблицам, с которыми она работает (просматривает или модифицирует). Таким образом, администратор базы данных получает более широкие возможности в плане защиты данных и управления доступом пользователей к объектам базы данных.

- **Инкапсуляция**. Идея хранимых процедур соответствует одной из главных целей объектно-ориентированного программирования (ООП) — инкапсуляции данных, структур и кода в набор весьма ограниченных, четко определенных внешних интерфейсов. В терминах ООП хранимые процедуры можно называть методами, являющимися единственным средством для работы пользователей или внешних программ с объектами РСУБД. Если вы хотите строго придерживаться объектно-ориентированного подхода, то с помощью системы защиты СУБД нужно полностью запретить непосредственный доступ к данным через SQL и оставить для доступа к базе данных *только* хранимые процедуры. Однако столь суровые ограничения практикуются очень редко.
- **Простота доступа**. В больших базах данных уровня предприятия набор хранимых процедур может быть основным средством для доступа прикладных программ к базе данных. Хранимые процедуры образуют точно определенное множество транзакций и запросов, которые приложения могут выполнять над базой данных. Для большинства прикладных программистов вызов простой предопределенной функции, которая проверяет остаток на счету для указанного номера клиента или добавляет заказ с указанными идентификаторами клиента и товара и количеством последнего, оказывается существенно проще для понимания, чем соответствующие инструкции SQL.
- **Обеспечение бизнес-правил**. Возможности ветвления в хранимых процедурах очень часто используются для размещения бизнес-правил в базе данных. Например, хранимая процедура, используемая для добавления заказа к базе данных, может содержать логику проверки достаточности кредита у клиента, размещающего заказ, а также проверять наличие достаточного количества товара на складе и отвергать заказ, если хотя бы одно из этих условий не выполнено. В большой компании может быть несколько различных способов размещения заказов в базе данных: одна программа для продавцов, другая — для отдела телемаркета, третья — для заказов через веб-сайт и т.д. Каждая из них содержит свою процедуру принятия заказа; обычно эти процедуры написаны разными программистами в разное время. Но если все они используют одну и ту же хранимую процедуру для добавления заказа, то тем самым гарантируется единообразность применения бизнес-правил независимо от происхождения заказа.

Производительность хранимых процедур

Различные производители СУБД по-разному реализуют хранимые процедуры. В некоторых системах текст хранимой процедуры находится в базе данных в том виде, в каком он был написан программистом, и интерпретируется только тогда, когда процедура выполняется. Этот подход позволяет создавать гибкие языки программирования хранимых процедур, но сильно замедляет их выполнение. Ведь СУБД должна по ходу выполнения процедуры читать ее инструкции, осуществлять их синтаксический анализ и определять, что ей следует делать.

Именно по причине низкой производительности интерпретируемых процедур некоторые производители СУБД предпочитают компилировать их заранее, генерируя некоторый промежуточный код, который выполняется гораздо быстрее. Компиляция может происходить как автоматически, сразу после создания процедуры, так и по запросу пользователя. Однако у предварительной компиляции хранимых процедур есть свой недостаток: точный процесс выполнения процедуры фиксируется во время ее компиляции и уже не может быть изменен. Чем это плохо? Предположим, что после компиляции процедуры были определены дополнительные индексы для таблиц, с которыми она работает. Скомпилированные запросы этой процедуры были оптимизированы без учета этих индексов и будут выполняться медленнее, чем могли бы в случае, если их перекомпилировать.

Для решения этой проблемы некоторые СУБД автоматически помечают все скомпилированные процедуры, на которых могли отразиться изменения объектов базы данных, как нуждающиеся в повторной компиляции. Когда такая процедура в очередной раз вызывается, СУБД видит пометку и компилирует процедуру перед ее выполнением. Так сохраняется преимущество предварительной компиляции, и процедуры выполняются оптимальным образом. Но и у этого подхода остается один недостаток: непредсказуемые задержки в выполнении процедур, связанные с их динамической перекомпиляцией. Когда повторная компиляция не нужна, хранимая процедура может выполниться очень быстро; если же процедуру потребовалось перекомпилировать, перед ее выполнением может быть довольно ощутимая задержка — существенно большая, чем при использовании старой скомпилированной версии.

Чтобы выяснить, как компилируются хранимые процедуры конкретной СУБД, можно посмотреть, какие опции имеются для инструкций `CREATE PROCEDURE`, `EXECUTE PROCEDURE`, `ALTER PROCEDURE` и др.

Системные хранимые процедуры

Если СУБД поддерживает хранимые процедуры, то, скорее всего, в ней есть и некоторый набор готовых системных процедур, которые автоматизируют различные операции с базой данных и некоторые функции ее администрирования. Пионером создания системных хранимых процедур была Sybase, включившая их в свой продукт Sybase SQL Server. Сегодня сотни хранимых процедур Transact-SQL облегчают выполнение множества полезных функций, таких как, например, управление записями пользователей, заданиями, распределенными серверами, репликацией и т.д. Большинство системных процедур Transact-SQL названо в соответствии со следующими соглашениями:

- `sp_add_`*имя* — добавление нового объекта (пользователя, сервера, реплики и т.п.);
- `sp_drop_`*имя* — удаление существующего объекта;
- `sp_help_`*имя* — получение информации об объекте или объектах.

Например, процедура sp_helpuser возвращает информацию о пользователях текущей базы данных. В Microsoft SQL Server имена системных хранимых процедур Transact-SQL зачастую имеют между словами символы подчеркивания, помимо символа подчеркивания в префиксе sp_. Кстати, поскольку производитель выбрал префикс sp_ для того, чтобы указывать с его помощью системные хранимые процедуры, лучше избегать применения этого префикса в именах пользовательских процедур.

Oracle в качестве такого предопределенного префикса использует DBMS_. Большинство таких процедур собрано в пакеты в соответствии с их функциональностью. Например, пакет DBMS_LOB содержит подпрограммы общего назначения (хранимые процедуры и функции) для работы с большими объектами (LOB).

Внешние хранимые процедуры

Хотя на расширенных диалектах SQL большинства ведущих СУБД можно писать довольно мощные хранимые процедуры, их возможности все же ограничены. Одним из главных ограничений является отсутствие связи с внешним миром, т.е. доступа к функциям операционной системы и приложений, работающих в той же системе. Кроме того, расширенные диалекты SQL — это языки очень высокого уровня, не предназначенные для решения низкоуровневых задач, обычно решаемых на языках программирования типа C или C++. Для преодоления этих ограничений в большинстве СУБД можно обращаться к внешним хранимым процедурам.

Внешняя хранимая процедура — это процедура, написанная на одном из традиционных языков программирования (например, на C или Pascal) и скомпилированная вне СУБД. Для ее использования нужно предоставить СУБД объявление процедуры — ее имя, параметры и другую информацию, необходимую для ее вызова, наподобие соглашения о передаче параметров. После этого можно вызывать внешнюю процедуру точно так же, как и обычные хранимые процедуры базы данных, написанные на SQL. СУБД обрабатывает вызов и передает управление внешней процедуре, а по окончании ее работы снова получает управление и принимает возвращенные процедурой данные.

Microsoft SQL Server предоставляет программистам набор системных внешних процедур, обеспечивающих доступ к функциям операционной системы. Например, процедура xp_sendmail позволяет отправлять пользователям электронные почтовые сообщения с информацией о событиях, происходящих в базе данных.

```
xp_sendmail @RECIPIENTS = 'Joe', 'Sam',
            @MESSAGE = 'Отчет готов';
```

Другая процедура, xp_cmdshell, выполняет команды операционной системы, в которой работает SQL Server. Допускается также создание пользовательских внешних процедур, которые хранятся в динамически компонуемых библиотеках (DLL) и вызываются из хранимых процедур SQL Server.

Informix обеспечивает доступ к функциям операционной системы с помощью специальной инструкции SYSTEM. Кроме того, эта СУБД поддерживает пользовательские внешние процедуры, для объявления которых предназначена инструк-

ция CREATE PROCEDURE. Там, где обычно начинается тело хранимой процедуры Informix, помещается предложение EXTERNAL, в котором задается имя, местоположение и язык внешней процедуры. Объявленную таким образом процедуру можно вызывать как обычную хранимую процедуру Informix. Ту же возможность предоставляют и версии Oracle 8 и выше — в них внешние процедуры тоже объявляются с помощью инструкции CREATE PROCEDURE. Семейство продуктов DB2 компании IBM обеспечивает аналогичный набор возможностей.

Триггеры

Как говорилось в начале этой главы, *триггер* — это особая хранимая процедура, которая вызывается в ответ на модификацию содержимого базы данных. В отличие от хранимых процедур, созданных с помощью инструкции CREATE PROCEDURE, триггер нельзя выполнить с помощью инструкции CALL или EXECUTE. Каждый триггер связывается с определенной таблицей базы данных, и, когда данные в таблице изменяются (инструкцией INSERT, DELETE или UPDATE), он *запускается*, т.е. СУБД выполняет инструкции, составляющие его тело. Некоторые СУБД, например Oracle, позволяют связывать триггеры с системными событиями, такими как подключение пользователя к базе данных или закрытие базы данных.

Триггеры могут использоваться для автоматического обновления информации в базе данных. Предположим, вы хотите настроить базу данных таким образом, чтобы каждый раз, когда в таблицу служащих добавляется новая запись, план нового служащего добавлялся к плановому объему продаж офиса, в котором он работает. Вот триггер для Oracle PL/SQL, который решает эту задачу.

```
create or replace trigger upd_tgt
    /* Триггер, запускаемый при добавлении
       записи в таблицу SALESREPS */
    before insert on salesreps
    for each row
    begin
        if :new.quota is not null then
            update offices
               set target = target + new.quota;
        end if;
    end;
```

Для создания нового триггера, включаемого в базу данных, в большинстве ведущих СУБД (поддерживающих триггеры) используется инструкция CREATE TRIGGER. Она назначает триггеру имя (в нашем примере UPD_TGT), указывает, с какой таблицей его следует связать (SALESREPS) и в ответ на какие события он должен вызываться (в нашем случае таким событием является выдача инструкции INSERT). Далее следует тело триггера, которое, как и у обычных хранимых процедур, определяет последовательность инструкций, выполняемых при его вызове. В нашем примере для каждой записи, добавляемой в таблицу, должна быть выполнена указанная инструкция UPDATE, обновляющая таблицу OFFICES. Значение QUOTA добавляемой строки в теле триггера можно получить с помощью выражения :NEW.QUOTA.

Преимущества и недостатки триггеров

Триггеры, являясь составной частью определения базы данных, могут быть исключительно полезными. Они могут выполнять множество функций, включая следующие.

- **Контроль изменений**. Триггер может отслеживать и отменять определенные изменения, не разрешаемые в конкретной базе данных.
- **Каскадные операции**. Триггер может обнаруживать определенные операции (например, удаление сведений о клиенте или служащем) и автоматически вносить соответствующие изменения в другие таблицы базы данных (скажем, корректировать баланс счетов и объемы продаж).
- **Поддержка целостности**. Триггер может поддерживать более сложные связи между данными, чем те, которые могут быть выражены простыми ограничениями на значения столбцов и условиями ссылочной целостности. Для сохранения этих связей может потребоваться выполнение последовательности инструкций SQL, иногда даже с использованием конструкций `IF...THEN...ELSE`.
- **Вызов хранимых процедур**. В ответ на обновление базы данных триггер может вызвать одну или несколько хранимых процедур и даже выполнить какие-то действия вне базы данных, используя внешние процедуры.
- **Обнаружение системных событий**. В СУБД с поддержкой триггеров для системных событий триггер может отслеживать такие события, как, например, подключение к базе данных определенного пользователя.

В каждом из этих случаев триггер реализует набор бизнес-правил, налагаемых на содержимое базы данных. Благодаря тому что эти правила хранятся в базе данных централизованно (т.е. только в одном месте — в определении триггера) и триггер выполняется в СУБД автоматически, эти правила распространяются на все операции, выполняемые над базой данных ее собственными хранимыми процедурами и всеми внешними приложениями. Если их нужно изменить, достаточно сделать это в одном единственном месте.

Главным недостатком триггеров является их влияние на производительность операций с базой данных. Если триггер связан с некоторой таблицей, СУБД выполняет его каждый раз, когда эта таблица обновляется. И если от базы данных требуется очень быстрое добавление и модификация больших объемов данных, триггеры могут оказаться неприемлемым решением, тем более что перед проведением пакетных операций данные могут быть проверены заранее и наверняка будут удовлетворять всем требованиям целостности. По этой причине некоторые СУБД позволяют избирательно активировать и отключать триггеры по мере необходимости.

Триггеры в диалекте Transact-SQL

В Transact-SQL триггеры создаются инструкцией `CREATE TRIGGER` (как в диалекте Microsoft SQL Server, так и в диалекте Sybase Adaptive Server). Вот как выглядит рассмотренный ранее триггер `UPD_TGT` на диалекте Transact-SQL.

```
create trigger upd_tgt
   /* Триггер, активизирующийся при добавлении
      записи в таблицу SALESREPS */
   on salesreps
   for insert
   as
   if (@@rowcount = 1)
      begin
         update offices
            set target = target + inserted.quota
           from offices, inserted
          where offices.office = inserted.rep_office
      end
   else
       raiserror 23456;
```

Первое предложение определяет имя триггера (UPD_TGT). Второе (является обязательным) указывает, с какой таблицей он связан. Третье предложение (также обязательно) определяет операцию, в ответ на которую должен выполняться данный триггер. В данном случае мы определяем триггер, активизируемый при добавлении данных. Кроме инструкции INSERT, можно связывать триггеры с инструкциями UPDATE и DELETE, причем триггер может быть связан с одной, любыми двумя или всеми тремя инструкциями сразу — они перечисляются через запятую. Однако для любой из этих трех операций с таблицей может быть связан только один триггер. Тело триггера начинается с ключевого слова AS. Чтобы понять, что делает наш триггер, необходимо разобраться, как Transact-SQL обрабатывает строки целевой таблицы при ее модификации.

Специально для триггеров Transact-SQL определяет две виртуальные таблицы, структура которых идентична структуре таблицы, с которой связан триггер. Эти таблицы называются DELETED и INSERTED. Они заполняются строками из модифицируемой таблицы, причем их конкретное содержимое зависит от выполняемой операции.

- **DELETE** — все строки, удаленные из связанной таблицы, помещаются в таблицу DELETED; таблица INSERTED пуста.
- **INSERT** — все строки, добавленные в связанную таблицу, помещаются в таблицу INSERTED; таблица DELETED пуста.
- **UPDATE** — для каждой строки целевой таблицы, измененной инструкцией UPDATE, ее *исходная* версия помещается в таблицу DELETED, а *новая* версия — в таблицу INSERTED.

К этим двум виртуальным таблицам можно обращаться из тела триггера, и их данные можно использовать в триггере наряду с данными всех остальных таблиц. В нашем примере триггер проверяет, была ли в таблицу добавлена одна запись, для чего он обращается к системной переменной @@ROWCOUNT. Если это так, значение столбца QUOTA из виртуальной таблицы INSERTED прибавляется к значению столбца TARGET соответствующей строки таблицы OFFICES. Чтобы найти в таблице OFFICES соответствующую запись, мы соединяем эту таблицу с виртуальной таблицей INSERTED по идентификаторам офисов.

А вот еще один триггер, который решает проблему иного рода. Он отслеживает попытки удаления из базы данных записей о клиентах, для которых имеются записи о заказах. Обнаружив такую ситуацию, триггер отменяет всю транзакцию, включая инструкцию DELETE, в ответ на которую был вызван триггер.

```
create trigger chk_del_cust
   /* Триггер, активизирующийся при удалении
      записи из таблицы CUSTOMERS          */
   on customers
   for delete
   as
   /* Выясняем, имеются ли заказы у
      удаляемого клиента                   */
   if (select count(*)
         from orders, deleted
        where orders.cust = deleted.cust_num) > 0
      begin
         rollback transaction
         print "Удаление невозможно: имеются заказы"
         raiserror 31234
      end;
```

Для триггеров, связанных с инструкцией UPDATE, в Transact-SQL предусмотрена возможность выяснить, какие именно столбцы таблицы были изменены, и в ответ выполнить соответствующие действия. Для этого в триггерах может использоваться специальная форма инструкции IF — IF UPDATE. Следующий триггер активизируется в ответ на обновление записей в таблице SALESREPS и выполняет различные действия в зависимости от того, какой из столбцов изменен, QUOTA или SALES.

```
create trigger upd_reps
   /* Триггер, активизирующийся при обновлении
      таблицы SALESREPS                          */
   on salesreps
   for insert, update
   as
   if update(quota)
       /* Обрабатываем обновление столбца QUOTA */
       . . .
   if update(sales)
       /* Обрабатываем обновление столбца SALES */
       . . .
```

Триггеры в диалекте Informix

В Informix триггеры тоже создаются с помощью инструкции CREATE TRIGGER. Как и в Transact-SQL, эта инструкция определяет имя триггера, таблицу, с которой он связан, и действия, в ответ на которые он выполняется. Вот несколько примеров, иллюстрирующих ее синтаксис.

```
create trigger new_sls
  insert on salesreps . . .

create trigger del_cus_chk
  delete on customers . . .
```

```
create trigger ord_upd
  update on orders . . .

create trigger sls_upd
  update of quota, sales on salesreps . . .
```

Последний триггер активизируется только в ответ на обновление заданных столбцов таблицы SALESREPS.

Informix позволяет указать, когда именно должен вызываться создаваемый триггер.

- **BEFORE** — триггер вызывается перед выполнением любых изменений, когда ни одна строка связанной таблицы еще не модифицирована.
- **AFTER** — триггер вызывается после выполнения всех изменений, когда все строки связанной таблицы уже модифицированы.
- **FOR EACH ROW** — триггер вызывается для каждой модифицируемой строки, и ему доступны как старая, так и новая версия этой строки.

Для каждого из этих этапов в триггере может быть задана отдельная последовательность действий. Например, триггер может вызвать хранимую процедуру для вычисления общей стоимости заказов перед обновлением таблицы ORDERS, потом отреагировать на обновление каждой строки, а затем снова вычислить общую стоимость заказов уже после обновления всех требуемых записей. Вот определение триггера, который работает таким образом.

```
create trigger upd_ord
    update of amount on orders
    referencing old as pre new as post

    /* Вычисляем общую стоимость заказов перед обновлением */
    before (execute procedure add_orders()
              into old_total;)

    /* Отслеживаем увеличение и уменьшение заказов         */
     for each row
         when (post.amount < pre.amount)
                 /* Записываем в таблицу информацию
                    об уменьшенном заказе                   */
             (insert into ord_less
                 values (pre.cust,
                         pre.order_date,
                         pre.amount,
                         post.amount)
         when (post.amount > pre.amount);)
                 /* Записываем в таблицу информацию
                    об увеличенном заказе                   */
             (insert into ord_more
                 values (pre.cust,
                         pre.order_date,
                         pre.amount,
                         post.amount);)

    /* После обновления таблицы повторно вычисляем
       общую стоимость заказов                              */
```

```
after (execute procedure add_orders()
          into new_total;)
```

Предложение BEFORE в этом триггере указывает на то, что перед тем как начнется выполнение инструкции UPDATE, должна быть вызвана хранимая процедура ADD_ORDERS. Предполагается, что эта процедура вычисляет общую стоимость заказов, а результат заносится в переменную OLD_TOTAL. Подобным же образом предложение AFTER определяет, что та же хранимая процедура вызывается по завершении выполнения инструкции UPDATE и возвращаемое ею значение заносится в другую локальную переменную — NEW_TOTAL.

Предложение FOR EACH ROW определяет действие, которое должно быть выполнено для каждой обновленной строки таблицы. Таковым в нашем примере является вставка строки в одну из двух таблиц, в зависимости от того, был ли текущий обрабатываемый заказ уменьшен или увеличен. Эти таблицы содержат идентификаторы клиентов, даты заказов, а также их старые и новые суммы. Для получения данной информации триггеру нужен доступ к старым (до изменения) и новым (после изменения) значениям строки.

Предложение REFERENCING задает имена, которые будут использоваться для ссылки на эти две версии текущей строки. В нашем примере для ссылки на старые значения столбцов используется квалификатор PRE, а для ссылки на новые значения — квалификатор POST. Эти имена не являются предопределенными — вы можете назвать таблицы как вам удобно.

В отличие от других СУБД, Informix ограничивает набор действий, которые могут выполняться триггером. Допускаются только следующие действия:

- инструкция INSERT;
- инструкция DELETE;
- инструкция UPDATE;
- инструкция EXECUTE PROCEDURE.

Впрочем, последняя опция обеспечивает все же некоторую гибкость. Вызванная триггером процедура может выполнять практически любые действия, которые могли бы быть произведены самим триггером.

Триггеры в диалекте Oracle PL/SQL

В Oracle возможности создания триггеров шире, чем в Informix и Transact-SQL. В этой СУБД для создания триггеров также используется инструкция CREATE TRIGGER, но структура ее иная. Подобно Informix, она позволяет связать триггер с различными этапами операций обновления.

- **Триггер уровня инструкции** вызывается один раз для каждой инструкции модификации данных. Он может быть вызван до или после ее выполнения.
- **Триггер уровня строки** вызывается один раз для каждой модифицируемой записи. Он также может вызываться до или после модификации.

- **Замещающий триггер** выполняется вместо инструкции SQL. С помощью такого триггера можно отслеживать попытки приложения или пользователя обновить, добавить или удалить записи и вместо этих действий выполнять свои собственные. Можно определить триггер, который должен выполняться вместо либо некоторой инструкции, либо каждой попытки изменения строки таблицы.
- **Триггер системного события** выполняется при определенном системном событии, таком как подключение пользователя к базе данных, или при выключении базы данных.

Ниже даны определения триггеров в Oracle PL/SQL, которые выполняют ту же роль, что и сложный триггер для Informix из предыдущего раздела. В Oracle эта задача решается при помощи трех триггеров: два из них — это триггеры BEFORE и AFTER уровня инструкции, а третий выполняется для каждой обновляемой строки.

```
create trigger bef_upd_ord
   before update on orders
   begin
      /* Вычисляем общую стоимость заказов
         до обновления */
      old_total = add_orders();
   end;

create trigger aft_upd_ord
   after update on orders
   begin
      /* Вычисляем общую стоимость заказов
         после обновления */
      new_total = add_orders();
   end;

create trigger dur_upd_ord
   before update of amount on orders
   referencing old as pre new as post

   /* Отслеживаем увеличение и уменьшение
      отдельных заказов */
   for each row
      when (:post.amount != :pre.amount)
         begin
            if (:post.amount < :pre.amount)
            then
               /* Записываем в отдельную таблицу
                  информацию об уменьшенном заказе */
               insert into ord_less
                    values (:pre.cust,
                            :pre.order_date,
                            :pre.amount,
                            :post.amount);
            elsif (:post.amount > :pre.amount)
            then
               /* Записываем в отдельную таблицу
                  информацию об увеличенном заказе */
               insert into ord_more
```

```
                values (:pre.cust,
                        :pre.order_date,
                        :pre.amount,
                        :post.amount);
        end if;
    end;
```

Всего получается 14 различных типов триггеров Oracle. Двенадцать из них — это комбинации операций INSERT, UPDATE и DELETE со временем выполнения BEFORE или AFTER на уровне ROW или STATEMENT (3×2×2) плюс еще два триггера типа INSTEAD OF уровня ROW и STATEMENT. Однако на практике в реляционных базах данных Oracle триггеры типа INSTEAD OF применяются редко; они были введены в Oracle 8 для поддержки некоторых новейших объектно-ориентированных функций.

Дополнительные вопросы, связанные с использованием триггеров

В базах данных триггеры могут вызывать те же проблемы, что и инструкции UPDATE и DELETE. Например, триггер может вызвать серию каскадных операций. Предположим, что триггер активизируется в ответ на попытку пользователя обновить содержимое таблицы. В теле этого триггера выполняется инструкция UPDATE, обновляющая другую таблицу. Триггер этой второй таблицы может обновить еще одну таблицу и т.д. Ситуация ухудшается, когда один из этих триггеров пытается обновить исходную таблицу, обновление которой и вызвало всю серию операций. В этом случае получается бесконечный цикл активизации триггеров.

В различных СУБД эта проблема решается по-разному. В одних системах ограничивается набор действий, которые могут выполняться триггерами. В других предусмотрены встроенные функции, которые позволяют триггеру определить уровень вложенности, на котором он работает. В третьих имеются системные установки, определяющие, допустимо ли каскадирование триггеров. А есть системы, которые ограничивают уровень вложенности каскадно выполняемых триггеров.

Еще одна проблема триггеров состоит в том, что при пакетных операциях с базой данных, например при внесении в нее больших объемов информации, триггеры сильно замедляют работу базы данных. Поэтому некоторые СУБД позволяют избирательно отключать триггеры в таких ситуациях. В Oracle, например, предусмотрена такая форма инструкции ALTER TRIGGER.

```
ALTER TRIGGER BEF_UPD_RD DISABLE;
```

Аналогичную возможность обеспечивает инструкция CREATE TRIGGER в Informix.

Хранимые процедуры и стандарт SQL

Развитие технологии хранимых процедур, функций и триггеров в значительной степени связано с конкуренцией, существующей на рынке СУБД. Впервые хранимые процедуры и триггеры появились в СУБД Sybase SQL Server и быстро завоевали популярность, вследствие чего к средине 1990-х годов во многие системы уровня предприятия были добавлены собственные процедурные расширения SQL. И хотя первоначально хранимые процедуры не интересовали разработчиков

стандарта SQL, после публикации стандарта SQL2 в 1992 году им стали уделять самое пристальное внимание. К тому времени начались работы по созданию нового стандарта SQL3, но было принято решение отделить разработку стандарта для хранимых процедур от объектно-ориентированных расширений SQL3 и сосредоточиться на процедурных расширениях языка SQL.

Результатом этих усилий стала новая часть стандарта SQL, опубликованная в 1996 году как SQL/PSM (SQL/Persistent Stored Modules — постоянно хранимые модули SQL), международный стандарт ISO/IEC 9075-4. Он представляет собой набор дополнений, исправлений, новых и замененных параграфов стандарта SQL2 1992 года (ISO/IEC 9075:1992). Кроме дополнений к стандарту SQL, SQL/PSM представлял собой часть черновика планируемого стандарта SQL3. Разработка этого стандарта заняла больше времени, чем изначально планировалось, но в конечном счете SQL/PSM занял свое место в качестве части 4 стандарта SQL3, официально известного как SQL:1999. Стандарт SQL Call-Level Interface (CLI), описанный в главе 19, "SQL API", прошел аналогичный путь и стал частью 3 стандарта SQL. Когда стандарт SQL:1999 был опубликован, избранные фрагменты SQL/PSM, используемые в других частях стандарта, были перенесены в спецификацию SQL/Foundation (часть 1).

Стандарт SQL/PSM, опубликованный в 1996 году, касался только хранимых процедур и функций; спецификация триггеров *не* вошла в стандарт ISO SQL. Стандартизация триггеров рассматривалась в процессе разработки стандартов SQL2 и SQL/PSM, но было решено, что триггеры слишком сильно связаны с другими объектно-ориентированными расширениями, предлагаемыми для внесения в SQL3. В результате триггеры вошли в стандарт SQL:1999.

Публикация стандартов SQL/PSM и SQL:1999 отстала от первых коммерческих реализаций хранимых процедур и триггеров на много лет. К моменту принятия стандарта большинство производителей крупных СУБД давно ответили на энтузиазм пользователей и давление со стороны конкурентов введением в свои продукты хранимых процедур и триггеров. В отличие от некоторых других расширений SQL, где главную роль играли нововведения IBM, а реализация DB2 становилась стандартом де-факто, производители основных СУБД реализовывали хранимые процедуры и триггеры каждый по-своему и часто конкурировали между собой, внося уникальные возможности в свои реализации. В конечном итоге стандартизация хранимых процедур и триггеров ANSI/ISO мало влияет на сегодняшний рынок СУБД. Следует ожидать, что со временем реализации ANSI/ISO войдут в основные СУБД, но в качестве дополнений, а не замены их собственных реализаций хранимых процедур и триггеров.

Стандарт SQL/PSM для хранимых процедур

Возможности хранимых процедур, определенные стандартом SQL/PSM, сходны с теми, которые имеются в современных СУБД. Стандарт включает синтаксические конструкции для следующего:

- создания и именования процедур и функций, написанных на расширенном диалекте SQL;

- вызова ранее созданных процедур и функций;
- передачи параметров вызванной процедуре или функции и получения результатов ее работы;
- объявления и использования локальных переменных в процедурах и функциях;
- группировки инструкций SQL в блоки для совместного выполнения;
- условного выполнения инструкций SQL (`IF...THEN...ELSE`);
- циклического выполнения групп инструкций SQL.

Стандартом SQL/PSM определяются два типа подпрограмм, вызываемых посредством SQL. *Процедура SQL* — это подпрограмма, которая не возвращает значения. Она вызывается с помощью инструкции `CALL`.

```
CALL ADD_CUST ('XYZ Corporation', 2317, 30000.00,
               50000.00, 103, 'Chicago');
```

Как и в случае рассмотренных ранее в этой главе языков программирования хранимых процедур различных СУБД, хранимые процедуры SQL/PSM получают параметры в инструкции `CALL`. Хранимые процедуры SQL/PSM могут также возвращать данные назад вызывающим процедурам через выходные параметры (как и рассматриваемые ранее хранимые процедуры в различных СУБД). SQL/PSM, подобно языкам некоторых СУБД, поддерживает и комбинированные входные/выходные параметры.

Функция SQL возвращает значение. Она вызывается так же, как и встроенные функции SQL, используемые в выражениях.

```
SELECT COMPANY
  FROM CUSTOMERS
 WHERE GET_TOT_ORDS(CUST_NUM) > 10000.00;
```

SQL/PSM ограничивает функции SQL единственным возвращаемым значением посредством механизма вызова функции. И выходные, и входные/выходные параметры в функциях SQL запрещены.

Подпрограммы SQL являются объектами, входящими в структуру базы данных, описанную в стандарте SQL. Стандарт SQL/PSM допускает как создание подпрограмм в схеме базы данных (подпрограммы уровня схемы), где они существуют параллельно с таблицами, представлениями, утверждениями и другими объектами, так и создание их в модуле SQL, который представляет собой процедурную единицу, существующую еще со времени появления стандарта SQL1.

Создание подпрограммы

Следуя практике большинства ведущих СУБД, стандарт SQL/PSM определяет, что хранимые процедуры и функции должны создаваться с помощью инструкций `CREATE PROCEDURE` и `CREATE FUNCTION`. На рис. 20.18 и 20.19 изображен их синтаксис. В дополнение к показанным на рисунках возможностям, стандарт предусматривает также возможность объявления внешних хранимых процедур с указанием того, на каком языке они написаны, могут ли они извлекать и модифицировать данные, каковы соглашения об их вызове и т.д.

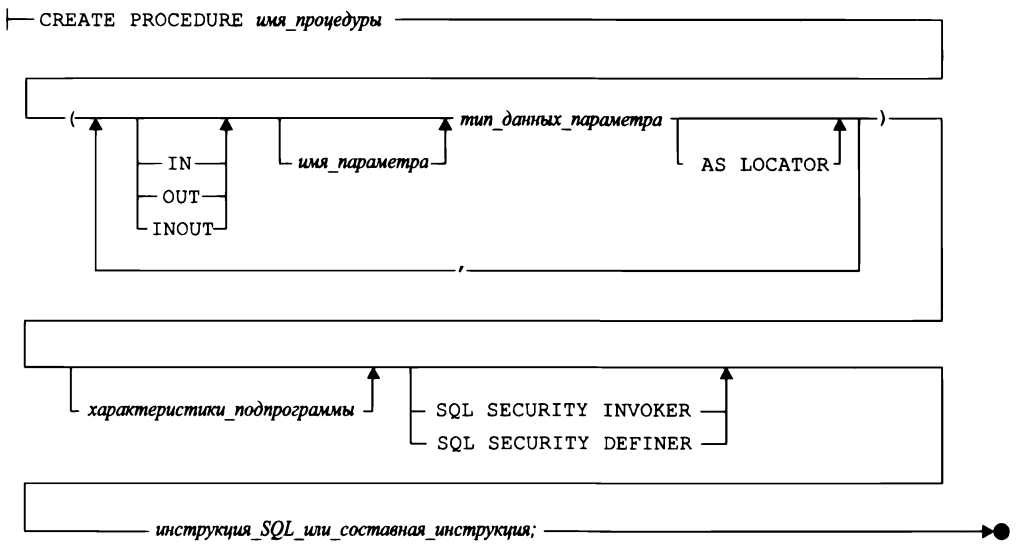


Рис. 20.18. Синтаксическая диаграмма инструкции SQL/PSM CREATE PROCEDURE

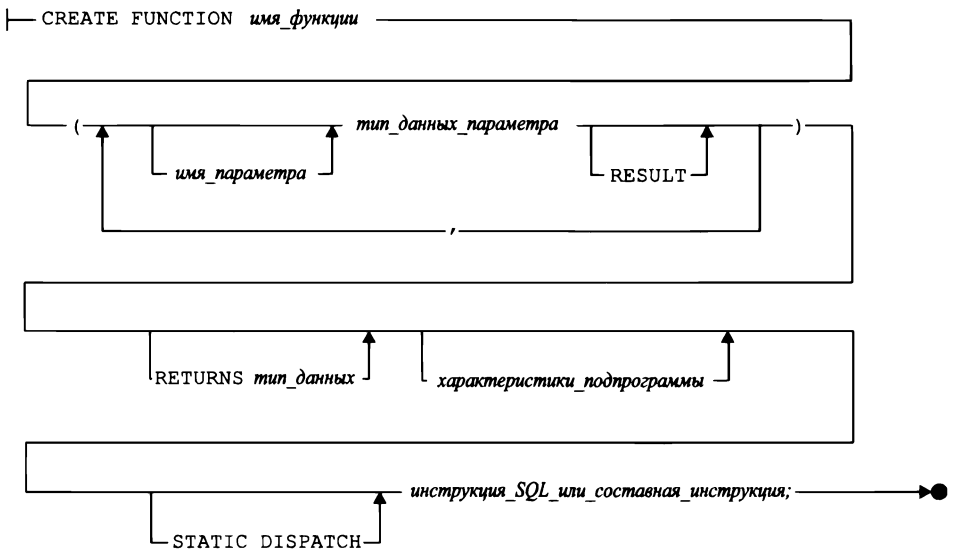


Рис. 20.19. Синтаксическая диаграмма инструкции SQL/PSM CREATE FUNCTION

Конструкции управления потоком

Стандарт SQL/PSM определяет классические программные управляющие конструкции, которые имеются в большинстве современных диалектов SPL. На рис. 20.20 представлен синтаксис циклических и условных конструкций.

Обратите внимание на то, что список инструкций SQL состоит из последовательности инструкций SQL, каждая из которых завершается точкой с запятой. Таким об-

разом, явное использование блоков инструкций в конструкциях IF...THEN...ELSE и LOOP при наличии последовательностей инструкций не требуется. Циклические структуры весьма гибкие: имеются циклы с проверкой условия в начале и конце цикла, а также бесконечные циклы, выход из которых может выполняться только по явной команде, имеющейся в теле цикла. Каждая управляющая конструкция завершается ключевым словом END со спецификатором, указывающим тип конструкции, что облегчает чтение и отладку программ.

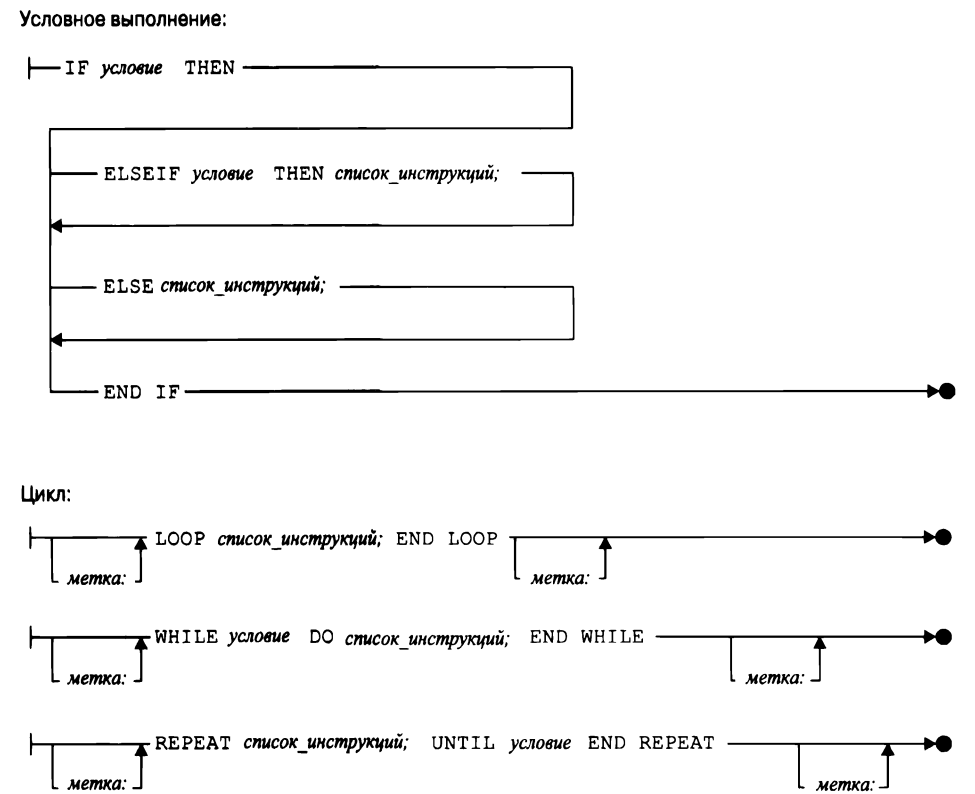


Рис. 20.20. Синтаксическая диаграмма управляющих конструкций в SQL/PSM

Операции с курсорами

Стандарт SQL/PSM расширяет возможности управления курсорами, предусматриваемые стандартом SQL2 для встроенного SQL. Инструкции DECLARE CURSOR, OPEN, FETCH и CLOSE сохранили свою роль и назначение, но для передачи параметров и получения возвращаемых данных в них используются не программные переменные, а параметры и переменные хранимых процедур.

Стандартом SQL/PSM вводится одна новая полезная конструкция — цикл обработки курсора (рис. 20.21). Как и аналогичные конструкции Oracle и Informix, о которых рассказывалось в этой главе, цикл объединяет определение курсора, инструкции OPEN, FETCH и CLOSE в единую структуру, содержащую код для обработки каждой записи.

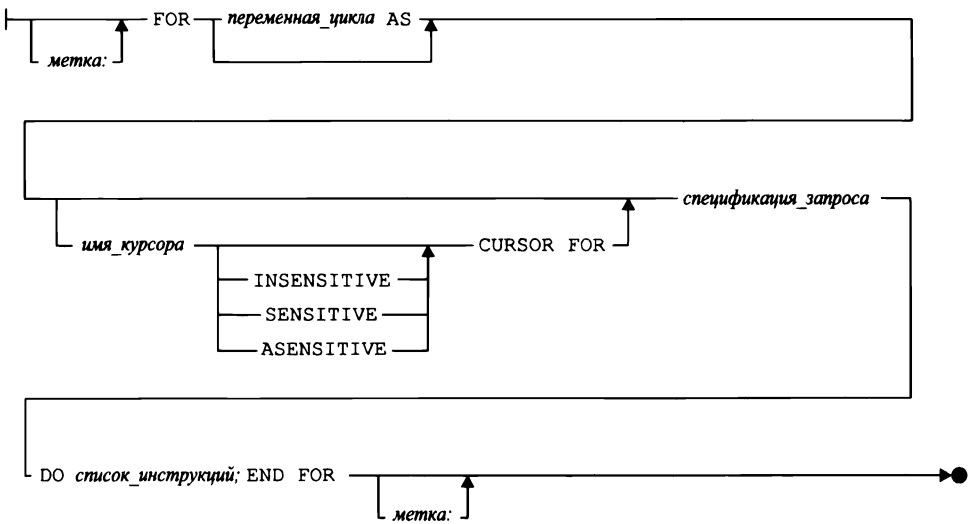


Рис. 20.21. Синтаксическая диаграмма цикла обработки курсора в SQL/PSM

Блочная структура

На рис. 20.22 изображен синтаксис блока инструкций, определенный стандартом SQL/PSM. Это довольно громоздкая конструкция, обеспечивающая следующие возможности.

- Возможность пометить блок при помощи метки инструкции.
- Объявление локальных переменных для использования в блоке.
- Объявления пользовательских локальных состояний ошибки.
- Объявления курсоров для запросов, выполняемых внутри блока.
- Объявления обработчиков состояний ошибки.
- Определение последовательности выполняемых инструкций SQL.

Эти возможности похожи на рассмотренные ранее возможности блоков в Oracle и Informix.

Локальные переменные в процедурах и функциях SQL/PSM (на самом деле, в блоках) объявляются с помощью инструкции DECLARE. Значения им присваиваются инструкцией SET. Функции возвращают значения, используя для этого инструкции RETURN. Вот пример блока инструкций, который может присутствовать в теле хранимой функции.

```
try_again:
   begin
      /* Объявление двух локальных переменных */
      declare msg_text varchar(40);
      declare tot_amt decimal(16, 2);

      /* Получение общей стоимости заказов     */
      set tot_amt = get_tot_ords();
      if (tot_amt > 0)
```

```
    then
       return (tot_amt);
    else
       return (0.00);
    end if
end try_again;
```

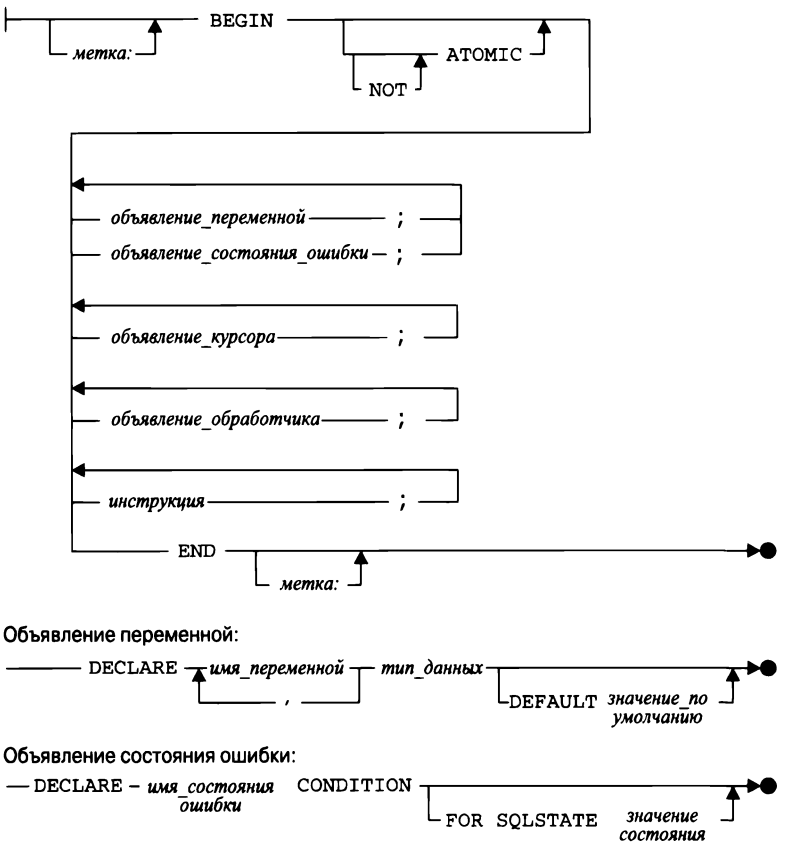


Рис. 20.22. Синтаксическая диаграмма блока инструкций SQL/PSM

Обработка ошибок

Блочные конструкции, определяемые стандартом SQL/PSM, обеспечивают довольно мощный механизм обработки ошибок. В стандарте описаны предопределенные состояния ошибок, которые могут быть обнаружены и обработаны.

- **SQLWARNING** — одно из предупреждений, определенных стандартом SQL.
- **NOT FOUND** — ситуация, когда инструкция FETCH достигла конца результирующего множества.
- **SQLSTATE** *значение* — проверка определенного кода ошибки SQLSTATE.
- **Пользовательское состояние** — именованное в хранимой процедуре состояние.

Состояния обычно определяются в терминах значений переменной SQLSTATE. Вместо использования числовых кодов, можно назначать состояниям имена. Кроме того, можно описывать собственные состояния, требующие специальной обработки.

```
declare bad_err condition for sqlstate '12345';
declare my_err condition;
```

После описания состояния его можно явно сгенерировать в хранимой процедуре с помощью инструкции SIGNAL.

```
signal bad_err;
signal sqlstate '12345';
```

Для обработки состояний ошибки, которые могут возникать в ходе выполнения хранимой процедуры, стандарт SQL/PSM позволяет создавать *обработчики*. В определении обработчика указывается перечень состояний, которые он должен обрабатывать, и действия, которые он должен предпринимать. Существует три типа обработчиков.

- **CONTINUE** — когда обработчик завершит свою работу, управление будет передано инструкции, следующей за той, при выполнении которой возникло состояние ошибки; таким образом, выполнение процедуры будет *продолжено* со следующей строки.
- **EXIT** — когда обработчик завершит свою работу, управление будет передано в конец блока инструкций, при выполнении которого возникло состояние ошибки; таким образом, будет осуществлен *выход* из текущего блока.
- **UNDO** — когда обработчик завершит свою работу, все изменения базы данных, сделанные в блоке инструкций, при выполнении которого возникла исключительная ситуация, будут отменены; результат будет таким же, как при отмене транзакции, начатой в начале блока.

Вот несколько примеров определений обработчиков.

```
/* Обработать предупреждение SQL и продолжить */
declare continue handler for sqlwarning
     call my_warn_routine();

/* Обработать ошибку и отменить изменения     */
declare undo handler for user_disaster
     begin
         /* Исправляем последствия ошибки     */
         . . .
     end;
```

Текст обработчика может быть довольно сложным, и в ходе его выполнения могут возникать новые ошибки. Чтобы избежать бесконечной обработки ошибок, сигнализация о состоянии ошибки на время обработки автоматически отключается. Однако стандарт позволяет перекрыть такое поведение при помощи инструкции RESIGNAL. Она работает в точности так же, как и инструкция SIGNAL, но используется исключительно в подпрограммах обработки ошибок.

Перегрузка имен подпрограмм

Стандарт SQL/PSM допускает перегрузку имен хранимых процедур и функций. Под этим термином подразумевается распространенная в объектно-ориентированном программировании возможность давать нескольким подпрограммам *одинаковые* имена, что повышает гибкость использования хранимых процедур. Подпрограммы с одинаковыми именами должны отличаться друг от друга количеством или типами принимаемых ими параметров. Например, можно определить три следующие хранимые функции.

```
create function combo(a, b)
    a integer;
    b integer;
    returns integer;
    as return (a+b);

create function combo(a, b, c)
    a integer;
    b integer;
    c integer;
    returns integer;
    as return (a+b+c);

create function combo(a, b)
    a varchar(255);
    b varchar(255);
    returns varchar(255);
    as return (a || b);
```

Первая функция COMBO складывает два целых числа, вторая — три, а третья функция комбинирует две символьные строки путем их конкатенации. Стандарт SQL/PSM позволяет назвать все три функции одним именем и хранить их в одной базе данных. Когда СУБД встречает вызов функции COMBO, она анализирует количество и типы переданных функции аргументов и определяет, какую версию этой функции ей нужно вызвать. Таким образом, механизм перегрузки позволяет программисту создать семейство подпрограмм с общим именем для выполнения одной и той же логической функции, но для данных различных типов. В терминах объектно-ориентированного программирования перегрузка иногда именуется *полиморфизмом*, что буквально означает, что одна и та же функция может принимать много различных форм.

Чтобы упростить управление семействами подпрограмм, имеющих одинаковые имена, стандарт SQL/PSM вводит концепцию *уникальных* имен. Уникальное имя — это второе имя подпрограммы, которое должно быть уникальным в схеме базы данных или в модуле. Оно однозначно идентифицирует подпрограмму и используется при удалении подпрограммы из базы данных и отражается в представлениях информационной схемы базы данных, описывающих хранимые процедуры. При вызове подпрограммы уникальное имя не используется, поскольку это противоречит основному назначению перегрузки. Поддержка уникальных имен или некоторого иного механизма является практическим требованием для любой системы с поддержкой перегрузки или полиморфизма объектов и обеспечивает возможность управления ими путем удаления или изменения их определений, поскольку система должна быть в состоянии выяснить, какой именно объект модифицируется.

Внешние хранимые процедуры

В основном, стандарт SQL/PSM посвящен расширениям языка SQL, которые используются для определения процедур и функций SQL. Заметим, однако, что метод вызова процедуры (инструкция CALL) или функции (обращение к функции по имени в инструкции SQL) применим не только к процедурам, определенным на языке SQL. На самом деле стандарт SQL/PSM предусматривает и работу с внешними хранимыми процедурами и функциями, написанными на других языках программирования, таких как C или Pascal. В этом случае для определения процедур и функций в СУБД используются такие же инструкции CREATE PROCEDURE и CREATE FUNCTION, как и для обычных хранимых процедур и функций. Эти инструкции указывают имена процедур и функций, а также принимаемые ими параметры. Специальное предложение инструкции CREATE определяет язык, на котором написана хранимая процедура или функция, с тем чтобы СУБД могла выполнить соответствующие преобразования типов данных и корректный вызов подпрограммы.

Дополнительные элементы SQL/PSM, связанные с хранимыми процедурами

Стандарт SQL/PSM рассматривает процедуры и функции как управляемые объекты базы данных, использующие расширения инструкций SQL для работы с другими объектами. Для удаления ненужных подпрограмм используется вариация инструкции DROP, а для изменения определения процедуры или функции — вариация инструкции ALTER. Точно так же стандартный механизм безопасности SQL расширяется дополнительными привилегиями. Привилегия EXECUTE дает пользователю возможность выполнять хранимую процедуру или функцию. Управление данной привилегией осуществляется так же, как и другими привилегиями базы данных, — при помощи инструкций GRANT и REVOKE.

В соответствии со стандартом SQL/PSM, хранимая процедура принадлежит схеме базы данных, а поскольку в одной базе данных может быть множество схем, то, чтобы однозначно идентифицировать процедуру при ее вызове, можно использовать квалифицированное имя с указанием конкретной схемы. Стандарт SQL/PSM предоставляет альтернативный метод поиска определения процедуры, для которой не задано имя схемы, с использованием новой концепции пути.

Этот метод заключается в следующем. С помощью инструкции PATH создается *путь*, представляющий собой последовательность имен схем, которые должны просматриваться в поисках хранимой процедуры. Значение по умолчанию для этого параметра может быть задано как часть заголовка схемы в инструкции CREATE SCHEMA. Кроме того, его можно динамически модифицировать в ходе сеанса SQL с помощью инструкции SET PATH.

Стандарт SQL/PSM позволяет автору хранимой процедуры или функции подсказать СУБД, каким образом повысить эффективность ее выполнения. Одним из примеров может служить определение хранимой процедуры как DETERMINISTIC или NOT DETERMINISTIC. Процедура, определенная как DETERMINISTIC, всегда возвращает одни и те же результаты при вызове с одними и теми же значениями параметров. Если СУБД обнаруживает многократные вызовы подпрограммы, описанной как DETERMINISTIC, она может сохранить копию возвращаемых результа-

тов. Позже, при очередном вызове, СУБД не потребуется реальное выполнение процедуры — она сможет просто вернуть те же результаты, что и в последний раз.

Еще одна подсказка заключается в указании, модифицирует ли хранимая процедура содержимое базы данных или ограничивается только его чтением. Это позволяет СУБД не только оптимизировать обращения к базе данных, но и налагать ограничения, связанные с безопасностью, на внешние процедуры из других источников. Другие подсказки говорят СУБД, должна ли функция вызываться, если один из ее параметров равен NULL, а также управляют тем, как СУБД выбирает для выполнения конкретную процедуру или функцию при применении перегрузки.

Стандарт SQL/PSM для триггеров

Триггеры должны были быть стандартизованы в рамках разработки стандарта SQL3, которая в конечном итоге привела к публикации стандарта ANSI/ISO SQL:1999. К тому времени многие коммерческие СУБД уже реализовали триггеры, так что стандарт обобщил возможности, доказавшие свою полезность на практике. Как и в коммерческих продуктах, стандартные триггеры ANSI/ISO определяются для одной конкретной таблицы. Стандарт позволяет определять триггеры только для таблиц, но не для представлений.

Механизмы работы триггеров в СУБД SQL Server, Oracle и Informix, показанные в примерах этой главы, помогут разобраться со стандартным механизмом ANSI/ISO. В стандарте нет каких-либо кардинальных отличий от уже описанных возможностей разных СУБД. Вот как выглядит попытка сравнения стандарта с конкретными реализациями.

- **Именование**. Стандарт рассматривает триггеры как именованные объекты базы данных.
- **Типы**. Стандарт предоставляет разработчикам триггеры INSERT, DELETE и UPDATE; триггеры UPDATE могут быть связаны с обновлением конкретного столбца или группы столбцов.
- **Время запуска**. Стандарт позволяет триггерам выполняться до инструкции обновления базы данных или после нее.
- **Операции уровня строки и инструкции**. Стандарт предоставляет разработчикам как триггеры уровня инструкции (выполняющиеся по одному разу для каждой инструкции обновления базы данных), так и триггеры уровня строки (выполняющиеся многократно, для каждой модифицируемой строки таблицы).
- **Псевдонимы**. Обращение к значениям "до" и "после" модифицированной строки осуществляется при помощи механизма псевдонимов, подобного псевдонимам таблиц в предложении FROM.

Для определения триггера используется инструкция CREATE TRIGGER, показанная на рис. 20.23. Отдельные части инструкции знакомы вам по примерам триггеров различных СУБД, с которыми вы встречались ранее в этой главе.

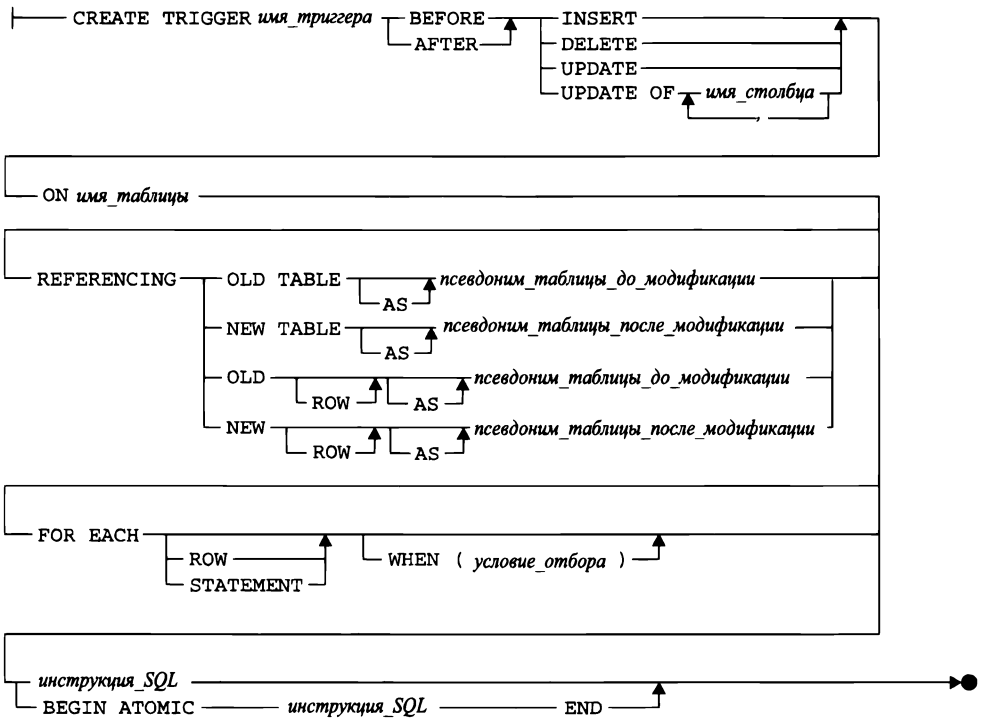


Рис. 20.23. Синтаксическая диаграмма стандартной инструкции CREATE TRIGGER

Одним очень полезным расширением, предоставленным стандартом, является предложение WHEN, которое может быть указано как часть выполняемого действия. Предложение WHEN (его использование не обязательно) работает наподобие предложения WHERE, определяя, должен ли вызываться триггер. Когда СУБД выполняет инструкцию некоторого типа, указанного в определении триггера, она проверяет условие отбора в предложении WHEN. Условие отбора имеет точно такой же вид, как и в предложении WHERE, и при вычислении дает значение либо TRUE, либо FALSE. Действие триггера выполняется только в том случае, когда условие отбора дает значение TRUE.

Для обеспечения безопасности стандарт SQL ввел новую привилегию — TRIGGER, которая может быть выдана определенному пользователю для работы с определенной таблицей. Пользователь, которому предоставлена данная привилегия, имеет право установить триггер для указанной таблицы (владелец таблицы всегда обладает таким правом).

Резюме

Хранимые процедуры и триггеры находят чрезвычайно широкое применение в реляционных базах данных и приложениях, связанных с обработкой транзакций.

- Хранимая процедура представляет собой набор взаимосвязанных операций, которые будут выполняться при вызове процедуры. Это повышает эффективность работы с базой данных и снижает вероятность ошибок.
- С целью поддержки хранимых процедур в SQL были введены расширения, придавшие этому языку черты, свойственные обычным языкам программирования. К расширениям относятся: возможность использования локальных переменных, организация ветвлений и циклов, а также специальные инструкции для построчной обработки результатов запросов.
- Хранимые функции являются специальным видом хранимых процедур, которые возвращают значения.
- Триггеры являются хранимыми процедурами, автоматически выполняемыми в ответ на попытки модификации таблиц, с которыми они связаны. Триггер может активизироваться операциями `INSERT`, `DELETE` и `UPDATE`, а в некоторых реализациях и системными событиями.
- В ведущих СУБД для поддержки хранимых процедур и триггеров используются различные диалекты SQL, достаточно сильно отличающиеся друг от друга.
- В настоящее время для хранимых процедур, функций и триггеров имеется международный стандарт. Однако разработан он не так давно и еще не полностью реализован в СУБД ведущих производителей.

21
ГЛАВА

SQL и хранилища данных

Одним из наиболее важных применений SQL и технологий реляционных баз данных на сегодняшний день является быстро растущая область *организации информационных хранилищ* и интеллектуальные ресурсы предприятий. Цель организации информационных хранилищ — использование накопленных данных для информационного обеспечения принятия решений. Растущая популярность Интернета и прямое взаимодействие с потребителями вызвали буквально взрывной рост количества данных о поведении покупателей (которое отслеживается по их перемещениям по страницам Интернета и действиям на этих страницах). Информационные хранилища рассматривают эти данные как ценнейший актив, который путем правильной обработки можно превратить в мощнейшее преимущество в конкурентной борьбе. Сопутствующий процесс *добычи информации* (data mining) включает анализ информации для получения сведений о поведении потребителей и взаимозависимостях различной информации о них. Реляционные SQL-базы данных являются ключевой технологией, лежащей в основе таких приложений.

Последние два десятилетия популярность приложений, связанных с деловым анализом, резко возросла. Исследования корпоративных информационных систем показывают, что в большинстве крупных компаний используются хранилища данных различного масштаба, а также те или иные программно-аналитические комплексы. В определенном смысле хранилища можно назвать реляционными базами данных, совершившими замкнутый цикл и вернувшимися к своим истокам. Когда реляционные СУБД только появились, существовавшие на то время СУБД (например, иерархическая СУБД IMS компании IBM) были ориентированы исключительно на обработку транзакций. Реляционные СУБД заняли пустовавшую нишу приложений, предназначенных для поддержки принятия решений. С ростом популярности таких приложений большинство производителей реляционных СУБД вступили также в борьбу за рынок приложений для обработки транзакций, и вскоре этот рынок также был ими завоеван. А когда на сцену вышли хранилища данных, ведущие производители реляционных СУБД снова переориентировали часть своих усилий на то, что раньше называлось поддержкой принятия решений, — только с новой терминологией и гораздо более мощными средствами, чем в свое время.

Концепции хранилищ данных

В основе технологии хранилищ данных лежит идея о том, что базы данных, ориентированные на *оперативную обработку транзакций* (Online Transaction Processing — OLTP), и базы данных, предназначенные для делового анализа, используются совершенно по-разному и служат разным целям. Первые — это средство производства, основа каждодневного функционирования предприятия. На производственном предприятии подобные базы данных поддерживают процессы принятия заказов клиентов, учета сырья, складского учета и оплаты продукции, т.е. выполняют главным образом учетные функции. С такими базами данных, как правило, работают клиентские приложения, используемые клерками, производственным персоналом, работниками складов и т.п. В противоположность этому базы данных *интеллектуальных ресурсов предприятия* (business intelligence, BI) (которые Кодд именовал ориентированными на *оперативную аналитическую обработку* (Online Analytical Processing, OLAP)) используются для принятия решений на основе сбора и анализа информации. Их главные пользователи — это менеджеры, служащие планового отдела и отдела маркетинга.

Ключевые отличия аналитических и OLTP-приложений, с точки зрения их взаимодействия с базами данных, перечислены в табл. 21.1. Чтобы лучше понять ее смысл, давайте рассмотрим процесс работы типичных приложений каждого типа. В качестве OLTP-примера возьмем приложение для обработки заказов клиентов. Его обращения к базе данных сводятся обычно к следующему:

- выборка строки из таблицы клиентов для проверки правильности идентификатора клиента;
- проверка лимита кредита клиента;
- выборка строки из таблицы товаров для проверки наличия заказанного товара;
- вставка новой строки в таблицу заказов с данными о новом заказе;
- обновление строки в таблице товаров для уменьшения количества имеющегося в наличии товара.

Таблица 21.1. Сравнение транзакционных и аналитических баз данных

Характеристика базы данных	База данных OLTP	База данных хранилища
Содержимое	Текущие данные	Данные, накопленные за долгий период времени
Структура данных	Структура таблиц соответствует структуре транзакций	Структура таблиц понятна и удобна для написания запросов
Типичный размер таблиц	От тысяч до нескольких миллионов строк	Миллионы и миллиарды строк
Схема доступа	Предопределена для каждого типа обрабатываемых транзакций	Произвольная; зависит от того, какая именно задача стоит перед пользователем в данный момент и какие сведения нужны для ее решения

Окончание табл. 21.1

Характеристика базы данных	База данных OLTP	База данных хранилища
Количество строк, к которым обращается один запрос	Десятки	От тысяч до миллионов
С чем работает приложение	С отдельными строками	С группами строк (итоговые запросы)
Интенсивность обращений к базе данных	Большое количество бизнес-транзакций в минуту или в секунду	Запросы обрабатываются минуты и даже часы
Тип доступа	Выборка, вставка и обновление	Практически 100%-ная выборка
Чем определяется производительность	Время выполнения транзакции	Время выполнения запроса

Рабочая нагрузка на базу данных в таком приложении — это небольшие по объему транзакции, состоящие из простых запросов на выборку, вставку и обновление отдельных строк. Вот примеры таких операций:

- получение цены товара;
- проверка количества имеющегося в наличии товара;
- удаление заказа;
- изменение адреса клиента;
- увеличение лимита кредита клиента.

В противоположность этому типичная транзакция бизнес-анализа (для примера возьмем формирование отчета о заказах) может потребовать выполнения таких операций:

- соединение информации из таблиц заказов, товаров и клиентов;
- вычисление общей стоимости заказов каждого товара каждым клиентом;
- вычисление общего количества единиц каждого заказанного товара;
- сортировка результирующей информации по клиентам.

Рабочая нагрузка на базу данных при получении этой информации представляет собой один долго выполняющийся запрос, требующий выборки большого количества данных. В нашем примере нужно извлечь информацию обо всех заказах, вычислить итоговые и средние значения и сформировать сводную таблицу. Таково большинство запросов из области делового анализа.

- В каких регионах в этом квартале продажа шла наиболее успешно?
- Насколько изменились объемы продаж каждого товара в последнем квартале по сравнению с аналогичным кварталом предыдущего года?
- Каковы тенденции изменения объемов продаж каждого товара?
- Какие клиенты покупают самые ходовые товары?
- Что общего у всех этих клиентов?

Как видите, рабочая нагрузка баз данных, используемых в аналитических и OLTP-приложениях, настолько различна, что очень трудно или даже невозможно подобрать одну СУБД, которая наилучшим образом удовлетворяла бы требованиям приложений обоих типов.

Компоненты хранилища данных

На рис. 21.1 изображена архитектура хранилища данных. Выделим три ее основных компонента.

- **Средства наполнения хранилища** — это набор программ, отвечающий за извлечение данных из корпоративных OLTP-систем (реляционных баз данных, унаследованных баз данных на мэйнфреймах и мини-компьютерах), их обработку и загрузку в хранилище; этот процесс обычно требует предварительной обработки извлекаемых данных, их фильтрации и переформатирования, причем записи загружаются в хранилище не по одной, а целыми пакетами.
- **База данных хранилища** — обычно это реляционная база данных, оптимизированная для хранения огромных объемов данных, их очень быстрой пакетной загрузки и выполнения сложных аналитических запросов.
- **Средства анализа данных** — это программный комплекс, выполняющий статистический и временной анализ, анализ типа "что если" и представление результатов в графической форме.

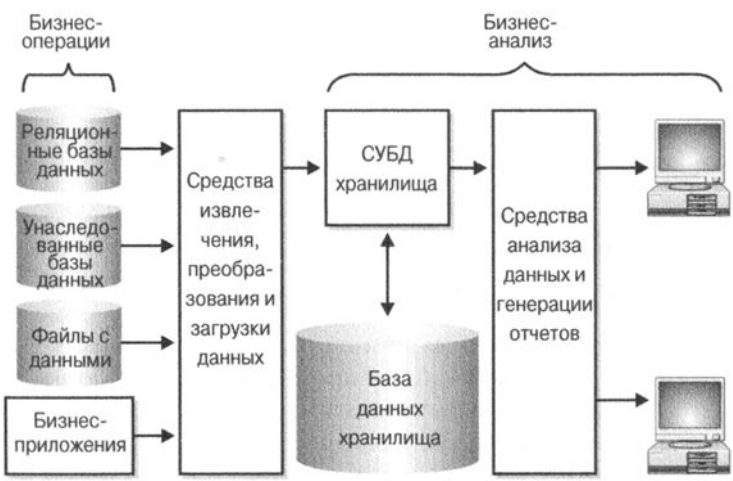


Рис. 21.1. Компоненты хранилища данных

Производители на рынке хранилищ данных сначала сосредоточивали свои усилия в одной из перечисленных областей. Некоторые создавали продукты для наполнения хранилищ, другие сосредоточивались на анализе данных. Кое-кто пытался охватить обе области, но в обеих областях продолжали работу независимые производители программного обеспечения, включая нескольких, доход которых достигал 100 миллионов долларов.

Вначале, когда рынок хранилищ данных только начинал свое развитие, были попытки разработок специализированных баз данных для хранилищ. Со временем этот вопрос заинтересовал и ведущих производителей СУБД. Некоторые из них разработали собственные специализированные базы данных, другие просто перекупили более мелкие компании с их разработками. Сегодня показанные на рис. 21.1 компоненты почти всегда представляют собой специализированные основанные на SQL СУБД, поставляемые одним из ведущих производителей баз данных уровня предприятия.

Эволюция хранилищ данных

Поначалу идея хранилища данных заключалась в создании огромного собрания всех данных предприятия, накопленных за весь период его работы. К такому хранилищу можно было бы адресовать практически любые возможные запросы, касающиеся истории деловой жизни компании. Многие компании пытались создать подобные хранилища, но мало у кого это получилось. На практике оказалось, что огромное хранилище масштаба предприятия создать не только трудно, но и дорого, и, что гораздо важнее, — его довольно неудобно использовать.

Поэтому со временем акцент сместился на хранилища данных для отдельных аспектов бизнеса; размещаемая в них информация носила конкретную практическую направленность и поддавалась более глубокому и эффективному анализу. Такие хранилища, меньшие, но все еще очень объемные, стали называть *центрами данных* (data marts). Именно управление распределенными центрами данных предприятия стало в последнее время основным объектом внимания производителей корпоративных СУБД. В частности, особое внимание уделяется выборке и форматированию данных в ситуациях, когда несколько центров данных извлекают информацию из одной и той же производственной базы данных. Это требует координации их действий, чтобы не получилось возврата к огромным централизованным хранилищам. Еще один подход заключается в том, чтобы оставить данные на месте, в базах данных OLTP, и формировать центры данных по запросам с помощью специализированного инструментария промежуточного уровня, который заставляет данные в нескольких базах данных иметь вид хранящихся в одной огромной интегрированной базе данных, которую можно рассматривать как виртуальное хранилище данных. В такой архитектуре, известной как информационная интеграция предприятия (enterprise information integration, EII), инструментарий промежуточного уровня тиражирует каждый запрос среди всех поддерживаемых физических баз данных и собирает результаты перед тем, как вернуть их пользователю, пославшему исходный запрос.

Хранилища и центры данных используются в самых разных отраслях. Но, пожалуй, наиболее широко (и агрессивно) они применяются в тех промышленно-финансовых сферах, где информация о тенденциях бизнеса служит основой для принятия решений, приводящих к значительной экономии средств или приносящих большую прибыль. Сюда входит следующее.

- **Крупные производства** — анализ тенденций в области сбыта, в частности сезонных колебаний объемов продаж, может помочь компании эф-

фективнее спланировать производство, разгрузить склады и сэкономить деньги для других целей.

- **Коммерция** — анализ эффективности мероприятий, направленных на увеличение сбыта (реклама, доставка товаров на дом и т.п.) и изучение демографических факторов, помогает выявить наиболее эффективные способы привлечения потенциальных покупателей и сэкономить на таких мероприятиях миллионы долларов.
- **Телекоммуникации** — анализ схемы звонков клиентов может помочь в создании более привлекательных комплексов цен и услуг и, возможно, привлечь новых клиентов из числа тех, которые пользуются услугами других компаний.
- **Авиакомпании** — анализ схемы перемещений пассажиров является основой планирования рейсов, тарифов и объемов перевозок с целью максимального увеличения прибылей компании.
- **Финансовые структуры** — анализ факторов, связанных с получением и погашением кредитов клиентами, и их сравнение с данными прошлых лет позволяют делать более точные заключения о кредитоспособности клиентов.

Архитектура баз данных для хранилищ

Структура базы данных для хранилища обычно разрабатывается таким образом, чтобы максимально облегчить анализ информации; ведь это основная функция хранилища. Данные должно быть удобно "раскладывать" по разным направлениям (называемым *измерениями*). Например, сегодня пользователь хочет просмотреть сводку продаж товаров по регионам, чтобы сравнить объемы продаж в разных областях страны. Завтра тому же пользователю понадобится картина изменения объемов продаж по регионам в течение определенного периода — ему нужно будет узнать, в каких регионах объемы продаж растут, а в каких, наоборот, сокращаются и какова динамика этих изменений. Структура базы данных должна обеспечивать проведение подобных типов анализа, позволяя выделять данные, соответствующие заданному набору измерений.

Кубы фактов

В большинстве случаев информация в базе данных хранилища может быть представлена в виде N-мерного куба (N-куба) фактов, отражающих деловую активность компании в течение определенного времени. Простейший трехмерный куб данных о продажах изображен на рис. 21.2. Каждая его ячейка представляет один факт — объем продаж в долларах. Вдоль одной грани куба (одного измерения) располагаются месяцы, в течение которых выполнялись отражаемые кубом продажи. Второе измерение составляют категории товаров, а третье — регионы продаж. В каждой ячейке содержится объем продаж для соответствующей комбинации значений по всем трем измерениям. Например, значение $50475 в левой верхней ячейке — это объем продаж *одежды* в *январе* в *восточном* регионе.

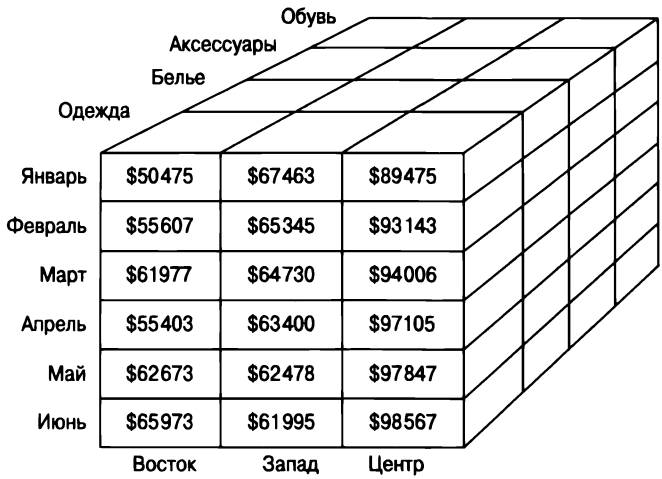


Рис. 21.2. Трехмерное представление данных о продажах

На рис. 21.2 показан трехмерный куб, но в реальных приложениях используются более сложные кубы с десятками и более измерений. Впрочем, хотя десятимерный гиперкуб трудно визуализировать (если вообще возможно), принципы его организации те же, что и у трехмерного. Каждое измерение представляет некоторую переменную величину, по которой ведется анализ. Каждой комбинации значений всех измерений соответствует одно значение некоторого параметра — факт, который обычно является зафиксированным в системе результатом деятельности компании.

Для иллюстрации структуры базы данных, обычно используемой в хранилищах данных, давайте разберем работу типичного аналитического приложения на простом примере. Возьмем коммерческую компанию, которая продает несколько категорий товаров разных производителей и имеет несколько сотен клиентов в разных регионах страны. Руководству компании требуется провести анализ данных о динамике продаж по пяти измерениям, анализ должен осветить имеющиеся тенденции и помочь выявить возможности увеличения сбыта. Вот эти пять измерений, определяющих пятимерный гиперкуб данных, фактами которого являются объемы продаж.

- **Категория**. Категория продаваемого товара (обувь, аксессуары, белье, одежда и т.д.). В хранилище содержатся данные о примерно двух десятках категорий товаров.
- **Производитель**. Производитель конкретного товара. Компания может торговать товарами 50 разных производителей.
- **Клиент**. Покупатель товаров. У компании несколько сотен клиентов. Крупнейшие из них заказывают товары в центральном офисе и обслуживаются одними и теми же служащими. Другие покупают товары в локальных отделениях компании и обслуживаются их сотрудниками.
- **Регион**. Регион страны, в котором продаются товары. Одни клиенты компании всегда покупают товар в одном и том же регионе, другие могут делать это в нескольких регионах.

- **Месяц**. Месяц, в котором были проданы товары. Для сравнительного анализа руководство компании решило ограничиться хранением данных за последние 36 месяцев (3 года).

Каждое из этих пяти измерений относительно независимо от остальных. Объемы продаж по конкретному клиенту могут быть рассредоточены по нескольким регионам. Товары конкретной категории могут поставляться одним или несколькими производителями. Фактом в каждой ячейке пятимерного куба является объем продаж для конкретной комбинации значений пяти измерений. Всего куб содержит более 35 миллионов ячеек (24 категории × 50 производителей × 300 клиентов × × 3 региона × 36 месяцев).

Схема звезды

Для большинства хранилищ данных самым эффективным способом моделирования N-мерного куба фактов является *схема звезды*. На рис. 21.3 показано, как выглядит такая схема для хранилища данных коммерческой компании, описанного в предыдущем разделе. Каждое измерение куба представлено таблицей его значений. На рис. 21.3 таких таблиц пять: `CATEGORIES` (категории), `SUPPLIERS` (производители), `CUSTOMERS` (клиенты), `REGIONS` (регионы) и `MONTHS` (месяцы). Для каждого значения измерения в таблице имеется отдельная строка. Например, в таблице `MONTHS` 36 строк, по одной для каждого месяца всего периода, за который анализируются данные о продажах. А в таблице `REGIONS` три региона представлены тремя строками.

Таблицы измерений в этой схеме часто содержат столбцы с описательной текстовой информацией или другими атрибутами, связанными с конкретным измерением (такими, как имя контактного лица из фирмы-производителя, адрес и номер телефона клиента или сроки контракта). Эти столбцы могут выводиться в отчетах, генерируемых аналитическим приложением. У таблицы измерения всегда имеется первичный ключ, индексирующий значения этого измерения. В таблице измерений всегда есть один или несколько столбцов, содержащих естественные идентификаторы измерений, такие как код региона, месяц и год или размер одежды. Однако эти естественные идентификаторы редко используются в качестве первичных ключей таблиц измерений из-за того, что со временем естественные идентификаторы могут изменяться. Это явление известно как *медленно изменяющиеся измерения*. Чтобы избежать изменений значений первичных ключей, обычно в их качестве в таблицах фактов используются числа, не имеющие реального смысла и известные как *суррогатные ключи*. Суррогатные ключи упрощают внешние ключи, поскольку они состоят из одного столбца. Без такого суррогатного ключа в таблице измерения `MONTHS` ее внешний ключ в таблице фактов состоял бы из двух столбцов — месяца и года.

В хранилище на рис. 21.3 суррогатные ключи используются во всех таблицах. Однако следует обратить внимание на то, что в таблицы измерений включены и естественные идентификаторы — например, имена регионов в таблице `REGIONS` ("Запад", "Восток" и т.д.) или названия категорий в таблице `CATEGORIES` ("Одежда", "Обувь" и т.д.).

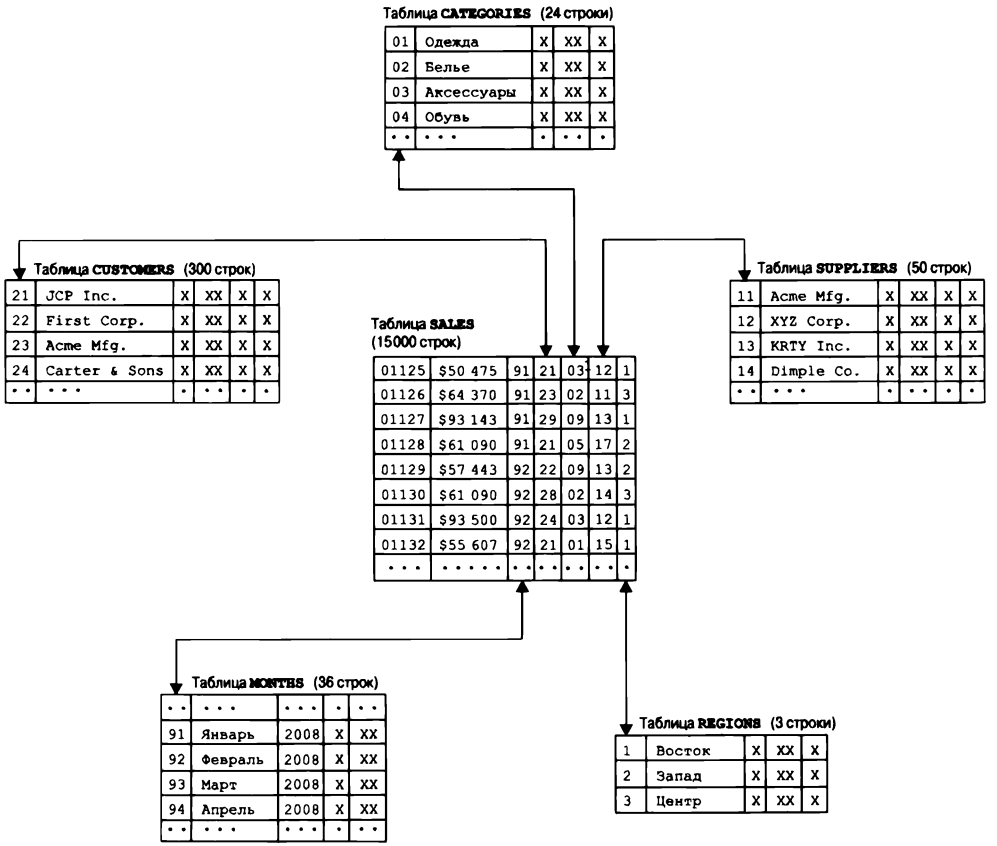


Рис. 21.3. Схема звезды для хранилища данных коммерческой компании

Самой большой таблицей в базе данных является *таблица фактов*. Она — центр схемы. На рис. 21.3 это таблица продаж SALES. Таблица фактов содержит столбец со значениями, которые находятся в ячейках N-мерного куба (см. рис. 21.2). Кроме него, в таблице фактов имеются столбцы с внешними ключами для всех таблиц измерений. Внешние ключи связывают строки с соответствующими строками таблиц измерений. В нашем примере таких внешних ключей пять. В данной структуре каждая строка представляет данные одной ячейки N-куба. Однако простой куб в состоянии показать только три измерения. Чтобы справиться с дополнительными измерениями, следует визуализировать дополнительные кубы. N-куб на рис. 21.2 представляет измерения REGIONS, CATEGORIES и MONTHS. Чтобы представить измерение SUPPLIERS, N-куб следует повторить 50 раз, по одному разу для каждого возможного производителя. Далее, для представления измерения CUSTOMERS требуется эти 50 кубов продублировать по 300 раз — по одному разу для каждого возможного клиента (итого, 15000 кубов). К счастью, некоторые многомерные СУБД в состоянии представлять кубы для анализа без физического хранения каждого куба.

В таблице фактов обычно немного столбцов, но зато очень много строк — сотни тысяч и даже миллионы; это вполне обычный размер хранилищ данных в промышленных и коммерческих компаниях. Столбец фактов почти всегда содержит

числовые значения, например денежные суммы, количество проданного товара и т.п. Подавляющее большинство отчетов, генерируемых аналитическими приложениями, содержит только итоговые данные — суммы, средние, максимальные и минимальные значения, процентные отношения, — получаемые путем арифметических операций над числовыми значениями из столбца фактов.

Причина, по которой схема на рис. 21.3 названа звездой, достаточно очевидна. Концы звезды образуются таблицами измерений, а их связи с таблицей фактов, расположенной в центре, образуют лучи. При такой структуре базы данных большинство запросов из области делового анализа объединяет центральную таблицу фактов с одной или несколькими таблицами измерений. Вот несколько примеров.

Показать итоговые объемы продаж одежды за январь 2008 года по регионам.

```
SELECT SUM(SALES_AMOUNT), REGION
  FROM SALES, REGIONS
 WHERE MONTH = 'Январь'
   AND YEAR  = 2008
   AND PROD_TYPE = 'Одежда'
   AND SALES.REGION = REGIONS.REGION
 ORDER BY REGION;
```

Показать средние объемы продаж товаров каждого производителя по клиентам и по месяцам.

```
SELECT AVG(SALES_AMOUNT), CUST_NAME, SUPP_NAME, MONTH, YEAR
  FROM SALES, CUSTOMERS, SUPPLIERS
 WHERE SALES.CUST_CODE = CUSTOMERS.CUST_CODE
       SALES.SUPP_CODE = SUPPLIERS.SUPP_CODE
 GROUP BY CUST_NAME, SUPP_NAME, MONTH, YEAR
 ORDER BY CUST_NAME, SUPP_NAME, MONTH, YEAR;
```

Многоуровневые измерения

В схеме звезды, изображенной на рис. 21.3, каждое измерение имеет только один уровень. На практике же довольно распространена более сложная структура базы данных, в которой измерения могут иметь по нескольку уровней.

- Данные о продажах могут накапливаться отдельно для каждого офиса. Офисы располагаются в районах, а районы в регионах.
- Данные о продажах накапливаются по месяцам, но анализировать их полезно не только по месяцам, но и по кварталам.
- Данные о продажах накапливаются по отдельным продуктам, а каждый продукт ассоциирован с конкретным производителем.

Подобные многоуровневые измерения усложняют звездообразную схему и на практике возникающую проблему решают по-разному.

- **Дополнительные данные в таблицах измерений**. Таблица измерения REGIONS, связанная с географическим местоположением, может содержать информацию об отдельных офисах и при этом иметь столбцы, указывающие, к какому району и региону относится каждый офис. Агреги-

рованные данные, касающиеся районов и регионов, могут быть получены с помощью итоговых запросов, объединяющих таблицу фактов с таблицей измерения и группирующих данные по столбцам района и региона. Этот подход концептуально довольно прост, но не всегда позволяет получить нужные данные за один раз: чаще всего требуется проводить вычисления в несколько этапов, запрос за запросом. Из-за этого на формирование итоговых данных может уходить слишком много времени.

- **Многоуровневые таблицы измерений**. Таблица географического измерения может быть расширена, чтобы в ней были отдельные строки для офисов, районов и регионов. Строки, содержащие суммарные данные для измерений более высокого уровня, добавляются в таблицу фактов при ее обновлении. Это решает проблему производительности запросов, поскольку промежуточные итоговые данные уже вычислены. Однако формулировать запросы становится сложнее, поскольку данные о любой сделке дублируются и присутствуют в таблице фактов в трех строках: для офиса, района и региона. Поэтому вычисление любых статистических данных требует большой аккуратности. Обычно при такой схеме базы данных в таблицу фактов включают столбец уровня, указывающий, к какому уровню принадлежит стоящее в строке значение, и каждый запрос, вычисляющий итоги и другие статистические данные, должен содержать условие для отбора записей только одного конкретного уровня.
- **Предварительно вычисленные итоги в таблицах измерений**. Вместо усложнения таблицы фактов, можно помещать предварительно вычисленные итоговые данные в таблицы измерений. Например, итоговый объем продаж по району может храниться в строке этого района в таблице географического измерения. Это решает проблему дублирования фактов, присущую предыдущему решению, однако годится только для очень простых случаев. Готовые итоговые суммы по районам не помогут получить сводки продаж отдельных товаров по районам или проследить изменение динамики продаж в районах по месяцам, а бесконечное увеличение сложности таблицы географического измерения — не выход.
- **Таблицы фактов для разных уровней**. Вместо усложнения единой таблицы фактов, этот подход предполагает создание нескольких таблиц фактов для разных уровней группировки данных. Для поддержки межуровневых запросов (например, о продажах по районам и месяцам) нужна специальная таблица фактов, в которой данные просуммированы для этих уровней измерений. Результирующая схема таблиц измерений и фактов имеет множество перекрестных связей и напоминает снежинку, поэтому схемы баз данных этого типа обычно так и называют — *схема снежинки*. Этот подход решает проблему производительности и устраняет дублирование данных в таблицах фактов, но может существенно усложнить структуру баз данных хранилищ, поскольку для каждого типа возможных запросов нужна своя таблица фактов. Кроме того, многие популярные инструменты анализа данных не умеют работать с такими схемами.

На практике поиск оптимальной схемы и архитектуры конкретного хранилища представляет собой сложное комплексное решение, учитывающее особенности фактов и измерений, типичные запросы и многое другое. Многие компании прибегают для этого к услугам сторонних консультантов.

Расширения SQL для хранилищ данных

Теоретически реляционная база данных звездообразной структуры обеспечивает хороший фундамент для выполнения запросов из области делового анализа. Возможность легко связывать представленную в базе данных информацию на основе значений данных как нельзя лучше подходит для произвольных, неструктурированных запросов, свойственных аналитическим приложениям. Однако между типичными аналитическими запросами и возможностями базового языка SQL имеются серьезные нестыковки, некоторые из которых перечислены ниже.

- **Сортировка данных**. Многие аналитические запросы явно или неявно требуют предварительной сортировки данных. Вам наверняка не раз встречались такие запросы, как "первые десять процентов", "первая десятка", "самые низкопроизводительные" и т.п. Однако ориентированный на работу со множествами язык SQL оперирует неотсортированными наборами записей. Единственным средством сортировки данных в нем является предложение `ORDER BY` в инструкции `SELECT`, причем сортировка выполняется в самом конце процесса, когда данные уже отобраны и обработаны.
- **Хронологические последовательности**. Многие запросы к хранилищам данных предназначены для анализа изменения некоторых показателей во времени: они сравнивают результаты этого года с результатами прошлого, результаты этого месяца с результатами того же месяца в прошлом году, показывают динамику роста годовых показателей в течение ряда лет и т.п. Однако очень трудно, а иногда и просто невозможно получить сравнительные данные за разные периоды времени в одной строке, возвращаемой стандартной инструкцией SQL. В общем случае это зависит от структуры базы данных.
- **Сравнение с итоговыми данными**. Многие аналитические запросы сравнивают значения отдельных элементов (например, объемы продаж отдельных офисов) с итоговыми данными (например, объемами продаж по регионам). Такой запрос трудно выразить на стандартном диалекте SQL. А если вам нужен сводный отчет классического формата — с детальными данными, промежуточными и общими итогами, то его с помощью SQL вообще нельзя сгенерировать, поскольку структура всех строк в таблице результатов запроса должна быть одинаковой.

Пытаясь решить все эти проблемы, производители СУБД для хранилищ данных обычно расширяют в своих продуктах возможности языка SQL. Например, в СУБД компании Red Brick (позже купленная компанией Informix, в свою очередь, купленной IBM), которая была одним из пионеров рынка хранилищ данных, в язык RISQL (Red Brick Intelligent SQL) включены следующие расширения:

- **диапазоны** — позволяют формулировать запросы вида "отобрать первые десять записей";
- **перемещение итогов и средних** — используется для хронологического анализа, требующего предварительной обработки исходных данных;
- **расчет текущих итогов и средних** — позволяет получать данные по отдельным месяцам плюс годовой итог на текущую дату и выполнять другие подобные запросы;
- **сравнительные коэффициенты** — позволяют создавать запросы, выражающие отношение отдельных значений к общим и промежуточным итогам без использования сложных подчиненных запросов;
- **декодирование** — упрощает замену кодов из таблиц измерений понятными именами (например, замену кодов товаров их названиями);
- **промежуточные итоги** — позволяют получать результаты запросов, в которых объединена детализированная и итоговая информация, причем с несколькими уровнями итогов.

Другие разработчики СУБД для хранилищ данных используют аналогичные расширения SQL в своих реализациях или встраивают такие возможности в свои инструменты для анализа. Как и в случае с другими расширениями SQL, их концептуальные возможности, предлагаемые различными разработчиками, сходны, а реализация зачастую совершенно разная.

Производительность хранилищ данных

Производительность хранилища данных — это одна из его важнейших характеристик. Если запросы, связанные с деловым анализом, выполняются слишком долго, пользователи не станут использовать хранилище для получения ответов на вопросы, возникающие в ходе принятия решений. Если данные слишком долго загружаются в хранилище, информационная служба предприятия откажется от их частого обновления, и отсутствие актуальной информации снизит полезность хранилища. Поэтому важнейшей задачей планирования хранилища является достижение оптимального соотношения между быстротой выполнения запросов и быстротой загрузки свежих данных.

Скорость загрузки данных

Загрузка в хранилище очередной порции данных — дело не быстрое. Этот процесс может занимать часы, а для больших хранилищ — даже дни. Обычно он состоит из следующих операций.

- **Извлечение данных**. Как правило, загружаемые в хранилище данные поступают из нескольких различных источников. Среди них могут быть реляционные базы данных OLTP-приложений.

- **Очистка данных**. Поступающие данные часто бывают "грязными" в том смысле, что содержат значительные ошибки. В частности, старые OLTP-системы могут не выполнять строгого контроля целостности, допуская, например, ввод записей с неверными идентификаторами клиентов или товаров. Поэтому в процессе наполнения хранилища обычно производится проверка целостности данных.
- **Перекрестная проверка данных**. Во многих компаниях для поддержки тех или иных деловых операций используются OLTP-приложения, написанные в разное время и не интегрированные в единое целое. Изменение данных в одной системе (например, добавление новых товаров в приложении для обработки заказов) может не отражаться автоматически в других приложениях (например, в системе складского учета) или отражаться, но с большими задержками. Когда данные из таких разобщенных систем поступают в хранилище, их необходимо проверять на согласованность.
- **Переформатирование данных**. Формат поступающих данных может сильно отличаться от формата, используемого в базе данных хранилища. Например, если текстовые данные переносятся с мэйнфреймов, их нужно транслировать из кодировки EBCDIC в ASCII. Еще один распространенный источник расхождений — форматы дат и времени. Кроме этих очевидных отличий, могут быть и более серьезные: информация из одной таблицы OLTP-источника может разделяться на части, помещаемые в несколько таблиц хранилища, и, наоборот, информация из нескольких таблиц источника может "сливаться" для помещения в одну таблицу хранилища.
- **Вставка/обновление данных**. После предварительной обработки выполняется помещение данных в хранилище. Обычно это специализированная операция, выполняемая в пакетном режиме без транзакций, но зато с использованием специальных средств восстановления в случае сбоев. Ее скорость должна быть очень высокой — обычно до сотен мегабайтов в час.
- **Создание/обновление индексов**. Специализированные индексы, используемые программным обеспечением хранилища, должны быть модифицированы в соответствии с обновленным содержимым базы данных. Как и в случае вставки/обновления данных, эта операция может потребовать особой обработки. В одних ситуациях быстрее и проще создать весь индекс заново, чем модифицировать его по мере поступления строк, в других ситуациях более приемлем второй вариант.

Все эти задачи обычно выполняются в пакетном режиме специализированными утилитами, отвечающими за наполнение хранилища. Когда идет загрузка, выполнение пользовательских запросов к базе данных запрещается, чтобы работа могла выполняться с максимальной скоростью. Несмотря на предельную оптимизацию всех операций, время загрузки информации в хранилище по мере увеличения объема его базы данных постепенно растет. Поэтому планирование соотношения скорости загрузки и скорости выполнения запросов должно проводиться на перспективу. Хранилища с большим количеством индексов или заранее вычисляемыми итоговыми значениями могут обеспечивать очень высокую производитель-

ность запросов, но загрузка данных в них требует слишком много времени. В хранилища более простой структуры данные загружаются быстрее, но запросы некоторых типов могут выполняться недопустимо долго. Поэтому перед администратором хранилища стоит задача нахождения оптимального баланса двух составляющих производительности.

Производительность запросов

Разработчики СУБД для хранилищ данных вкладывают огромные средства в оптимизацию своих продуктов. И результаты их усилий по-настоящему впечатляют. Однако наряду с производительностью растут объемы и сложность хранилищ, поэтому конечный пользователь зачастую может вообще не заметить никаких изменений.

Для повышения производительности запросов, связанных с деловым анализом, используется несколько технологий.

- **Специализированные схемы индексации**. Типичные аналитические запросы извлекают из хранилища некоторое подмножество данных, отбираемых по указанным значениям одного или нескольких измерений. Например, для сравнения результатов за текущий и предыдущий месяцы требуется информация только о двух месяцах из 36 (в нашей учебной базе данных). Специализированные схемы индексации разрабатываются таким образом, чтобы обеспечить сверхбыструю выборку соответствующих строк из таблицы фактов и объединение их со строками таблиц измерений. В некоторых из этих схем используются технологии побитовой обработки, когда за каждым из возможных значений измерения (или комбинации измерений) закрепляется отдельный разряд в значении индекса. Строки, соответствующие условию отбора, можно очень быстро идентифицировать с помощью операций побитовой арифметики, поскольку побитовое сравнение выполняется быстрее, чем сравнение чисел.
- **Технологии параллельной обработки**. Запросы, связанные с деловым анализом, нередко разделяются на части, которые могут выполняться параллельно, что сокращает общее время выполнения запроса. Например, если требуется соединить четыре таблицы базы данных, можно воспользоваться преимуществами двухпроцессорной системы, соединяя две таблицы в одном процессе, а две другие — в другом. Затем результаты этих промежуточных операций объединяются в общий набор записей. Аналогичным образом может быть распределена между процессорами и обработка данных одной таблицы: например, заказы за первое полугодие могут обрабатываться в одном процессе, а заказы за второе полугодие — в другом. Мультипроцессорные системы используются хранилищами данных не так, как в OLTP-системах. Оперативная обработка транзакций — это обычно множество одновременно выполняющихся небольших запросов, которые можно динамически распределять между процессорами, чтобы увеличить общую пропускную способность системы, в то время как бизнес-анализ — это последовательно поступающие

запросы на обработку огромного количества данных, и выполнение каждого такого запроса может быть ускорено за счет распределения между процессорами его составляющих.

- **Специализированные алгоритмы оптимизации**. Получив сложный запрос, включающий условия отбора и объединения таблиц, СУБД должна найти для его выполнения оптимальную последовательность действий. При оптимизации OLTP-приложений стараются как можно раньше использовать связи между первичными и внешними ключами, поскольку это позволяет значительно сократить объемы промежуточных наборов записей. При оптимизации хранилища может быть принято совершенно иное решение на основании накопленной в процессе наполнения хранилища информации о распределении значений в базе данных.
- **Разбиение таблиц и индексов**. Разбиением называется невидимое для пользователя деление таблицы или индекса на отдельно хранящиеся и управляемые части. Одно из преимуществ разбиения состоит в возможности резервного копирования, восстановления, добавления и удаления частей без прерывания операций над другими частями. Кроме того, может наблюдаться большой выигрыш в производительности, если СУБД сможет разбивать глобальные запросы на запросы к каждой части, которые смогут выполняться параллельно, а также исключать работу с частями, в которых для данного запроса нет никаких данных.
- **Тесная интеграция программного и аппаратного обеспечения**. Требования производительности к современным хранилищам данных приводят к необходимости более тесной интеграции аппаратных и программных платформ, на которых они работают. Эта интеграция требует, чтобы аппаратные средства оптимально соответствовали возможностям программного обеспечения, перенося, например, решение ряда задач, таких как фильтрация или агрегирование данных, на программное обеспечение аппаратных устройств хранения данных (примером может служить Exadata от Oracle).

Как за ускорение загрузки данных, так и за повышение скорости выполнения запросов отвечает администратор базы данных. Он должен отслеживать появление новых, более производительных, версий используемой СУБД и искать возможности повышения производительности системы на аппаратном уровне, например, за счет использования более быстрых процессоров или увеличения их количества.

Резюме

Хранилища данных представляют собой одну из самых быстрорастущих частей рынка реляционных СУБД, и к ним предъявляется ряд специальных требований.

- Базы данных хранилищ должны иметь специальную структуру и должны быть оптимизированы для выполнения типичных аналитических запросов, поскольку их рабочая нагрузка на базу данных существенно отличается от нагрузки, создаваемой OLTP-системами.

- Для наполнения хранилищ требуются специализированные утилиты, обеспечивающие высокую скорость загрузки данных.
- Для эффективного использования информации из хранилищ требуется применение специализированных структур баз данных (например, схемы звезды), обеспечивающих выполнение различных видов делового анализа при высокой производительности.
- При работе с хранилищами часто используются расширения SQL, позволяющие анализировать скрытые тенденции роста, группировать данные по диапазонам, сравнивать данные за разные периоды времени и многое другое.
- Проектирование большого хранилища данных требует специальных знаний и умения найти оптимальное соотношение между быстротой загрузки данных в хранилище и скоростью выполнения аналитических запросов.

22
ГЛАВА

SQL и серверы приложений

Серверы приложений представляют собой одну из основных новых компьютерных технологий, вызванную к жизни ростом Интернета. Серверы приложений образуют ключевой уровень архитектуры большинства коммерческих веб-сайтов. Как следует из их названия, серверы приложений предоставляют платформу для работы прикладной логики, которая определяет взаимодействие с пользователем на веб-сайте. Но серверы приложений выполняют и другую важную роль — действуя как посредники между компонентами веб-сайта со стороны Интернета (веб-серверы и инструменты управления содержимым сайта) и компонентами со стороны информационных технологий, такими как унаследованные корпоративные приложения и базы данных. В этой роли серверы приложений должны тесно сотрудничать с программным обеспечением СУБД, а языком такого сотрудничества является SQL. В этой главе исследуется роль SQL в многоуровневой архитектуре веб-сайта с применением серверов приложений.

SQL и веб-сайты: ранние реализации

Серверы приложений не всегда играли ведущую роль в архитектурах веб-сайтов. Ранние веб-сайты были направлены, в первую очередь, на донесение своего содержимого пользователям в виде статических веб-страниц. Содержимое веб-сайта было структурировано в виде ряда предопределенных веб-страниц, хранящихся в виде файлов. Веб-сервер принимал запрос от пользовательского браузера (в виде сообщений HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста)), находил запрошенную страницу (или страницы) и пересылал ее с использованием того же HTTP назад браузеру для вывода на экран. Содержимое веб-страниц хранилось в формате HTML (HyperText Markup Language, язык разметки гипертекста). HTML-код конкретной страницы содержал текст и графику, выводимые на экран, а также ссылки, обеспечивающие переход от одних страниц к другим.

Прошло совсем немного времени, и требования к распространяемой через World Wide Web информации "перешагнули" статические возможности предопределен-

ных веб-страниц. Компании стали использовать веб-сайты для связи со своими клиентами, и им требовалась поддержка таких возможностей, как поиск определенного товара или принятие заказов от клиентов. Первым шагом в этом направлении стала обработка информации веб-сервером, как показано на рис. 22.1. Веб-серверы стали принимать не только запросы статических страниц, но и запросы на выполнение *сценариев*, т.е. последовательностей инструкций, предназначенных для определения того, какая информация должна быть представлена на экране клиента.

Рис. 22.1. Динамические веб-страницы при отсутствии серверов приложений

Сценарии веб-серверов часто писались на специализированных языках сценариев, таких как Perl и PHP. В простейшем виде сценарий мог выполнять очень простые вычисления (такие, как получение текущих даты и времени от операционной системы) и выводить результат как часть веб-страницы. В более сложном виде сценарий мог получать введенную пользователем на странице с формой информацию, выполнять запрос к базе данных с ее использованием и возвращать результаты этого запроса. Поскольку результат работы сценария мог меняться от одного вызова к другому, получающиеся в результате веб-страницы стали *динамическими*: их содержимое могло меняться от просмотра к просмотру, в зависимости от конкретных результатов выполнения сценария.

Языки сценариев обеспечивали самую первую возможность связи между веб-сайтами и SQL-базами данных. Сценарий мог, например, передать SQL-запрос СУБД и получить результаты для вывода их в виде веб-страницы. Однако при таком решении возникала масса других проблем. Большинство языков сценариев интерпретируемые, и выполнение сложного сценария могло потребовать много процессорного времени. Сценарии работали как отдельные процессы систем на базе UNIX или Windows, что приводило к высоким накладным расходам при ежесекундном запуске десятков или сотен сценариев. Эти и другие ограничения решения с применением сценариев привели к разработке альтернативного подхода и появлению серверов приложений как части архитектуры веб-сайта.

Серверы приложений и трехуровневые архитектуры веб-сайтов

Логичной эволюцией сценариев веб-сервера стало выделение отдельной роли *сервера приложений*, что привело к трехуровневой архитектуре, показанной на рис. 22.2. Обратите внимание на то, что многие профессионалы рассматривают веб-сервер и сервер приложений как разные уровни и считают эту архитектуру четырехуровневой, или, более обобщенно, N-уровневой. Веб-сервер отвечает за поиск и обслуживание статических веб-страниц и статических частей веб-страниц. Когда требуется определить, какая информация будет выведена пользователю или когда следует обработать полученные от пользователя данные, веб-сервер вызывает отдельный сервер приложений для выполнения необходимых действий. В случае небольшого веб-сайта сервер приложений представляет собой отдельный процесс на том же физическом сервере, где работает веб-сервер. В более общем случае, на больших веб-сайтах, веб-сервер и сервер приложений работают на двух разных компьютерах, обычно соединенных высокоскоростной локальной сетью. Независимо от конфигурации, веб-сервер передает запросы в виде сообщений серверу приложений и получает ответы в виде HTML-содержимого, которое будет выводиться на странице.

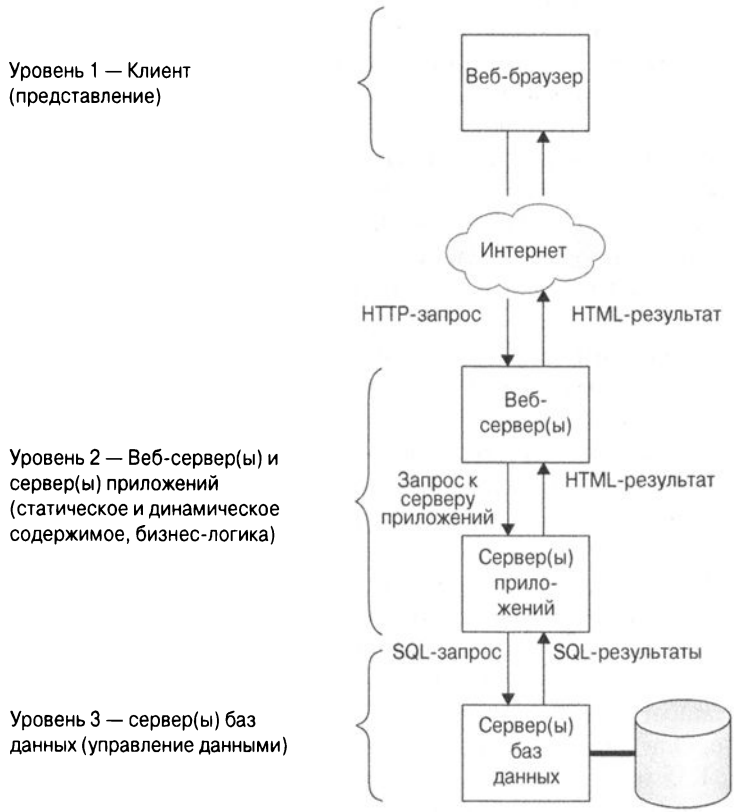


Рис. 22.2. Трехуровневая архитектура с применением сервера приложений

В первое время предлагалось большое количество разнообразных продуктов для серверов приложений. Некоторые серверы требовали, чтобы приложения были написаны на языках программирования C или C++, другие требовали использования Java. Интерфейс между сервером приложений и веб-сервером определялся интерфейсом прикладного программирования (API) двух ведущих производителей веб-серверов — Netscape и Microsoft. Но все прочие аспекты — от языка программирования до поддержки служб, предоставляемых сервером приложений и API для доступа к базам данных, — стандартизованы не были.

Разработка фирмой Sun Microsystem продукта Enterprise Java Beans (EJBs) и основанная на нем спецификация Java2 Enterprise Edition (J2EE) послужили началом стандартизации серверов приложений. Класс *java bean* представляет собой класс Java, следующий стандарту Sun Java Beans Standard, предоставляющему инфраструктуру создания объектов для использования в графическом инструментарии. EJBs основан на растущей популярности Java в качестве языка программирования. Спецификации Sun, ведущего разработчика серверов и лидера в области интернет-технологий, стали общепризнанными. Эти спецификации включают также стандартизованный API для доступа к базам данных — одной из наиболее важных функций серверов приложений — а именно: соответствующий стандарту Java Database Connectivity (JDBC).

Вскоре серверы приложений на основе спецификации J2EE вырвались в лидеры рынка. Производители, которые придерживались других подходов, дополняли свои продукты поддержкой Java и в конечном итоге переходили на стратегию J2EE. Еще через некоторое время на рынке серверов приложений началось укрупнение. Компания Sun приобрела NetDynamics, разработчика одного из первых серверов приложений J2EE. BEA Systems, ведущий производитель программного обеспечения среднего уровня для обработки транзакций, приобрела WebLogic, еще одного пионера рынка серверов приложений. (В 2007 году BEA была приобретена компанией Oracle.) Netscape, традиционно работавшая в области веб-серверов, дополнила линию своих продуктов покупкой Kiva, еще одного лидера рынка в области серверов приложений.

Позже, когда AOL приобрела Netscape, а затем организовала совместное предприятие с Sun, оба эти J2EE-сервера приложений перешли под управление Sun, в конечном итоге превратившись в сервер приложений Sun iPlanet (который позже стал сервером приложений SunONE). Hewlett-Packard последовала общему примеру, приобретя еще одного разработчика серверов приложений Bluestone. IBM же оказалась белой вороной и создала собственный сервер приложений под названием WebSphere. Oracle также пошел по пути разработки собственного продукта — Oracle Application Server, хотя со временем многие программные продукты Oracle были заменены приобретенными компонентами сторонних производителей.

После нескольких лет агрессивной конкурентной борьбы, спецификация J2EE продолжала эволюционировать и включала расширенные возможности доступа серверов приложений к базам данных. WebLogic компании BEA и WebSphere компании IBM заняли ведущее место, практически поделив между собой рынок. Продукты от Sun, Oracle и десятка более мелких производителей поделили оставшуюся часть рынка. Каждый мало-мальски значительный серверный продукт удовлетворял требованиям спецификации J2EE и обеспечивал возможность обращения к базам данных на основе JDBC.

Доступ серверов приложений к базам данных

Концентрация рынка серверов приложений вокруг спецификации J2EE привела к эффективной стандартизации, по крайней мере на время, *внешнего* интерфейса между сервером приложений и СУБД на базе JDBC. Концептуально сервер приложений может автоматически обращаться к любой базе данных, предоставляющей JDBC-совместимый API, что тем самым делает его независимым от конкретной СУБД. На практике небольшие различия между СУБД в таких областях, как, например, диалект SQL или именование, требуют определенной настройки кода и тестирования. Однако эти отличия невелики, и подгонка под конкретную базу данных относительно проста.

Настройка управления данными со стороны прикладного кода, работающего на сервере приложений, оказывается более сложной задачей. В то время как сервер приложений обеспечивает единообразность управления данными, он делает это в нескольких различных архитектурах, использующих разные типы EJBs спецификации J2EE. Разработчик приложения должен делать выбор среди этих подходов, а в ряде случаев — и комбинировать их, чтобы добиться соответствия приложения поставленным требованиям. Вот некоторые из решений, которые должны быть приняты.

- Будет ли логика приложения обращаться к базе данных непосредственно из сессии или содержимое базы данных будет представлено как entity beans с инкапсулированной в них логикой? (Что такое system beans и entity beans, будет рассказано немного позже.)
- Если используется непосредственный доступ из сессии, может ли session bean быть без запоминания состояния (что упрощает его кодирование и управление им со стороны сервера приложений) или логика обращения к базе данных требует, чтобы bean был с запоминанием состояния, сохраняющий контекст между вызовами?
- Если для представления содержимого базы данных используются entity beans, может ли приложение для управления взаимодействием с базой данных полагаться на их хранение сервером приложений или логика приложения требует, чтобы entity bean обеспечивал собственную логику обращения к базе данных?
- Если для моделирования содержимого базы данных используются entity beans, должны ли они взаимнооднозначно соответствовать таблицам базы данных или лучше, если они будут представлять более высокоуровневые объектно-ориентированные представления данных, с данными внутри каждого bean, полученными из нескольких таблиц базы данных?

Компромиссы, представленные приведенными вопросами проектирования, хорошо демонстрируют проблемы соответствия SQL и технологий реляционных баз данных требованиям веб и архитектуры без запоминания состояния, а также требованиям серверов приложений и объектно-ориентированного программирования. В нескольких следующих разделах описываются основы EJBs и варианты решений проблем обращения в различных поддерживаемых архитектурах доступа к данным.

Типы EJB

В сервере приложений, совместимом с J2EE, пользовательский прикладной код Java, который реализует некоторую бизнес-логику, упакован и выполняется как набор EJBs. Объект EJB имеет хорошо определенное множество внешних интерфейсов (методов), которые он должен предоставлять, и написан с точным множеством открытых методов класса, которые определяют внешний интерфейс объекта. Вся работа выполняется в bean-объекте, и любые закрытые переменные, которые поддерживаются им для внутреннего пользования, могут быть инкапсулированы и скрыты от других объектов и разработчиков, которым не надо знать детали внутренней реализации; более того, код, который они пишут, не должен никоим образом зависеть от них.

Объекты EJBs работают на сервере приложений в *контейнере*, который предоставляет для beans как среду времени выполнения, так и службы. Последние простираются от общих служб, таких как управление памятью или диспетчеризация работы beans, до специфичных служб типа доступа к сети или к базе данных (через JDBC). Контейнер предоставляет также услуги по хранению состояния beans между их активациями. (*Сохраняемость* (persistence) — это свойство объектно-ориентированного программирования, которое обеспечивает сохранение данных для последующего использования. Обычным средством для этого являются базы данных.)

С точки зрения управления данными, имеются два основных типа объектов EJBs, графически проиллюстрированных на рис. 22.3.

- **Session beans** представляют сессии отдельного пользователя с сервером приложений. Концептуально имеется взаимнооднозначное соответствие между каждым session bean и текущим пользователем. На рисунке пользователи Мери, Джо и Сэм представлены каждый своим собственным session bean. Если внутри такого bean имеются внутренние переменные экземпляра, то эти переменные представляют текущее состояние, связанное с данным пользователем в течение этой конкретной сессии.
- **Entity beans** представляют бизнес-объекты и логически соответствуют индивидуальным строкам таблицы базы данных. Например, для entity beans, представляющих офисы, имеется взаимнооднозначное соответствие между каждым entity bean и конкретным офисом, который, кроме того, представлен в нашей учебной базе данных одной строкой таблицы `OFFICES`. Если в bean имеются внутренние переменные экземпляра, то их значения представляют значения столбцов в соответствующей строке таблицы `OFFICES`. Это состояние не зависит ни от какой конкретной сессии пользователя.

Beans любого типа могут обращаться к базе данных, но обычно делают это разными способами.

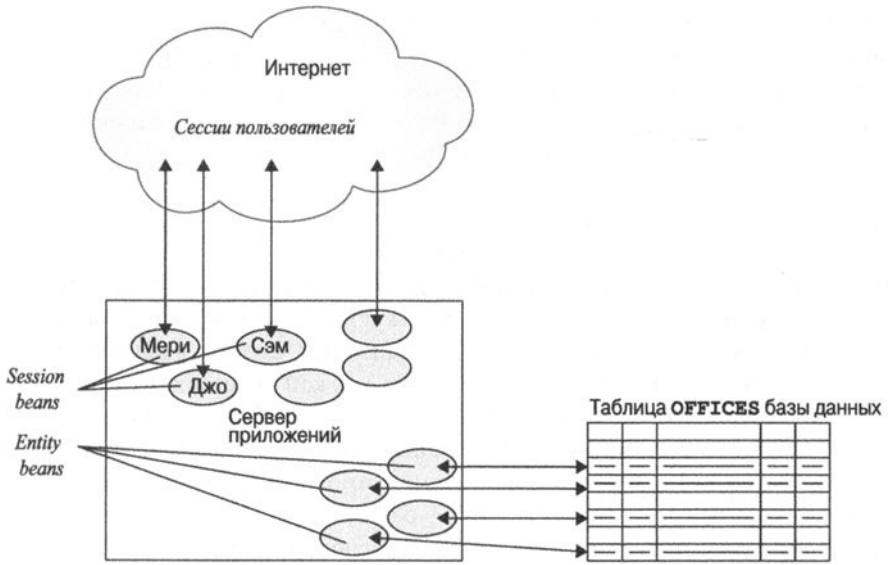


Рис. 22.3. Типы EJBs

Доступ к базе данных со стороны session bean

Обычно session bean обращается к базе данных при помощи последовательности из одного или нескольких вызовов JDBC от имени пользователя, которого представляет данный bean. Сервер приложений разделяет session beans на две категории, в зависимости от того, как они работают с состояниями.

- **Session bean без состояния**. Этот тип не хранит никакой информации о состоянии между вызовами методов. Свои действия он выполняет от имени одного пользователя и по одному запросу за раз. Каждый запрос, который выполняет bean, не зависит от предыдущих запросов. При таком ограничении каждый вызов bean должен передавать ему (в виде параметров) всю необходимую для выполнения запроса информацию.
- **Session bean с состоянием**. Этот тип поддерживает хранение информации о состоянии между вызовами. Bean должен "помнить" информацию из предыдущих вызовов (собственное состояние) для того, чтобы корректно выполнять последующие вызовы. Для хранения этой информации он использует закрытые переменные экземпляра.

В двух следующих разделах приведены примеры задач, которые проще всего решить при помощи описанных типов session bean. Вы указываете, должен ли session bean хранить состояние или нет, в специальном *дескрипторе применения* (deployment descriptor), который содержит информацию, передаваемую серверу приложений, на котором будет работать этот bean.

Сервер приложений на занятом веб-сайте может легко иметь больше session beans и прочих объектов EJBs, чем позволяет оперативная память. В таком случае сервер приложений поддерживает активными в оперативной памяти только огра-

ниченное количество экземпляров session bean. Если связанный с неактивной в настоящий момент сессией пользователь становится активен (т.е. следует обработать очередной его щелчок на странице веб-сайта), то сервер приложений выбирает другой экземпляр того же класса bean и *пассивизирует* его, т.е. сохраняет значения его переменных экземпляра, а затем использует данный bean для обслуживания активированной пользовательской сессии.

То, является ли session bean поддерживающим состояния или нет, существенно влияет на его активизацию и пассивизацию. Поскольку session bean без хранения состояния не требует сохранения информации между вызовами методов, сервер приложений не должен сохранять соответствующие переменные экземпляра при пассивизации и восстанавливать их при активизации. Однако в случае session bean с сохранением состояния сервер приложений должен копировать переменные экземпляра в постоянную память (в дисковый файл или базу данных) при пассивизации и восстанавливать их при активизации. Таким образом, session beans с сохранением состояния приводят к снижению производительности и пропускной способности сервера приложений. Beans без сохранения состояния предпочтительны в смысле производительности, но многие приложения сложно или даже невозможно реализовать без использования beans с сохранением состояния.

Использование JDBC из session bean без сохранения состояний

На рис. 22.4 показан простой пример приложения, которое легко реализовать при помощи обращений к базе данных session beans без сохранения состояния. Страница не веб-сайте выводит текущую цену товаров компании. Эта страница не может быть статической, поскольку цена может вырасти каждую минуту. Так что когда браузер пользователя запрашивает эту страницу, веб-сервер передает запрос серверу приложений, который вызывает метод session bean. Этот session bean может использовать JDBC для того, чтобы послать SQL-инструкцию `SELECT` базе данных и получить текущие цены на товары. Session bean переформатирует полученный результат и отправляет его веб-серверу в виде фрагмента веб-страницы, которую тот вернет пользователю.

Session beans без сохранения состояния могут выполнять и более сложные функции. Предположим, что у той же компании на том же веб-сайте имеется страница, на которой пользователь может заказать каталог продукции, заполнив небольшую форму. Когда форма заполнена и пользователь щелкает на кнопке ее отправки, браузер посылает данные из формы на веб-сервер, который передает запрос серверу приложений. В этот раз вызывается другой метод session bean, который получает в качестве параметров информацию из заполненной формы. Session bean может использовать JDBC для пересылки SQL-запроса `INSERT` в таблицу базы данных, хранящую заказы каталогов, полученные через веб-сайт.

В каждом из приведенных примеров вся необходимая для работы session bean информация передается методу в качестве параметров. Когда работа выполнена, хранить эту информацию больше не требуется. При очередном вызове вновь будет передана вся необходимая информация, так что хранить какие-то данные между вызовами не требуется. Что еще более важно, действия базы данных при каждом вызове никак не зависят от всех других вызовов. Ни одна транзакция базы данных не требует нескольких вызовов метода.

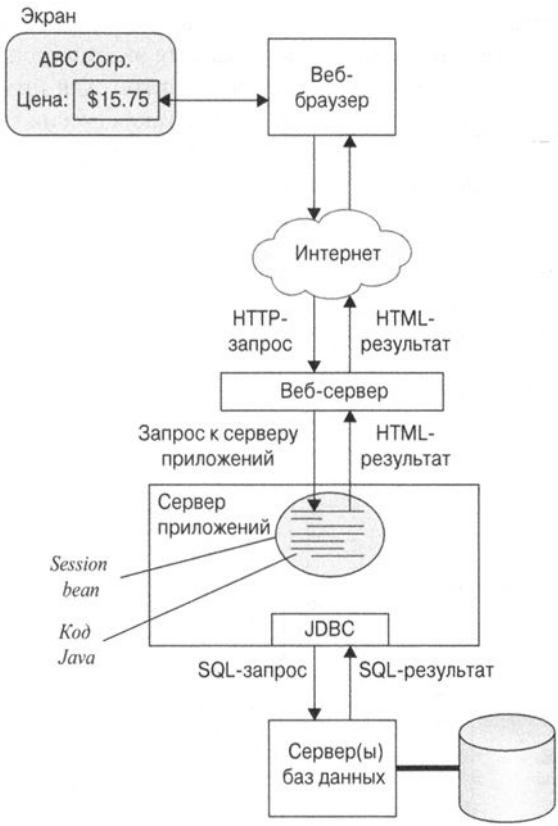


Рис. 22.4. Вызовы базы данных из session bean без сохранения состояний

Использование JDBC из session bean с сохранением состояния

Для многих задач ограничения, накладываемые session beans без сохранения состояний, оказываются чрезмерными. Рассмотрим более сложную форму, охватывающую четыре страницы. Когда пользователь заполняет каждую страницу и отправляет ее на сайт, session bean должен накапливать информацию и хранить ее до тех пор, пока не будут заполнены все четыре страницы и все данные не будут готовы для передачи в базу данных. Необходимость хранения информации между вызовами метода требует применения session bean с сохранением состояний.

Еще один пример, в котором требуется session bean с сохранением состояний, представляет собой коммерческий веб-сайт, где пользователь может выполнять покупки, складывая покупаемые товары в виртуальную "корзину", которая затем будет оплачена. После 40–50 щелчков на страницах сайта пользователь соберет в корзине пять-шесть товаров. Если после этого пользователь щелкнет на кнопке показа содержимого корзины, то проще всего будет удовлетворить его любопытство, если хранить это содержимое как состояние session bean.

В обоих примерах session bean должен поддерживать непрерывную связь с базой данных для эффективного выполнения стоящей перед ним задачи. На рис. 22.5 по-

казана соответствующая схема работы, отличающаяся от приведенной на рис. 22.4. Даже если bean можно реализовать без переменных экземпляра (например, путем хранения всей информации о состоянии в базе данных), для поддержания доступа к базе данных требуется одна непрерывная сессия. Такая сессия поддерживается посредством API со стороны клиента СУБД, а сам API должен хранить информацию о состоянии сессии между вызовами метода bean.

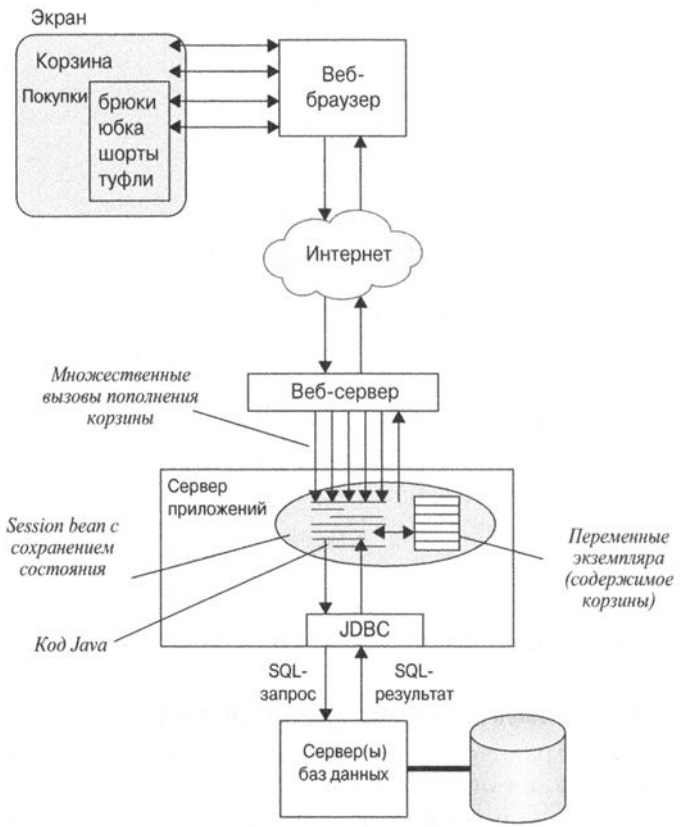


Рис. 22.5. Вызовы базы данных из session bean с сохранением состояний

Доступ к базе данных со стороны entity bean

Можно реализовать законченное, интеллектуальное приложение для веб-сайта с использованием session beans на сервере приложений на основе J2EE. Однако программирование приложений с применением session beans обычно приводит к коду в большей степени процедурному, чем объектно-ориентированному. Объектно-ориентированная философия состоит в наличии классов объектов (в данном случае, классов EJB), представляющих сущности реального мира, такие как покупатели или офисы, а также экземпляров объектов, представляющих отдельных покупателей или офисы. Но session beans не представляют никакие из указанных сущностей; они призваны представлять текущие активные пользовательские сес-

сии. Когда взаимодействие с базой данных осуществляется непосредственно при помощи session beans, представление сущностей реального мира остается в основном на долю базы данных; соответствующие объекты при этом отсутствуют.

Entity beans предоставляют объекты для сущностей реального мира и строк представляющей их таблицы реляционной базы данных. Классы entity bean олицетворяют покупателей и офисы; отдельные экземпляры представляют конкретных покупателей и офисы. Другие объекты (такие, как session beans) на сервере приложений могут взаимодействовать с покупателями и офисами с использованием объектно-ориентированных технологий, вызывая методы представляющих их entity beans.

Для поддержки этой объектно-ориентированной модели требуется очень тесное сотрудничество между представлениями сущностей при помощи entity bean и их представлениями в базе данных. Если session bean вызывает метод entity bean заказчика, который изменяет лимит кредита последнего, то это изменение должно быть отражено в базе данных, с тем чтобы приложение обработки заказов, использующее эту базу данных, работало с новым значением лимита кредита. Аналогично, если приложение управления складом добавляет некоторое количество товара в базу данных, соответствующий entity bean на сервере приложений также должен быть обновлен.

Так же как при необходимости сервер приложений пассивизирует и активизирует session beans, в случае большой загрузки пассивизируются и активизируются и entity beans. Перед тем как сервер приложений пассивизирует entity bean, состояние последнего должно быть сохранено в базе данных. Когда же сервер приложений реактивизирует entity bean, его переменные экземпляра должны получить те же значения, что были перед пассивизацией, для чего их следует загрузить из базы данных. Класс entity bean определяет методы обратного вызова, которые entity bean должен предоставить для реализации описанной синхронизации.

Имеется точное соответствие между действиями entity beans и действиями базы данных, показанное в табл. 22.1. Спецификация J2EE предоставляет два различных способа управления этой координацией.

- **Хранение на уровне beans**. Каждый entity bean сам ответственен за синхронизацию с базой данных. Прикладной программист, разрабатывающий entity bean и его методы, должен при необходимости использовать JDBC для чтения и записи информации в базу данных. Контейнер сервера приложений уведомляет bean при выполнении действий, требующих взаимодействия с базой данных.
- **Хранение на уровне контейнера**. За синхронизацию с базой данных отвечает контейнер EJB на сервере приложений. Контейнер отслеживает взаимодействия с entity bean и автоматически использует JDBC для чтения и записи информации в базу данных и обновления (при необходимости) переменных экземпляра bean. Прикладной программист, разрабатывающий entity bean и его методы, должен сосредоточиться на бизнес-логике bean и считать, что его переменные экземпляра точно представляют состояние информации в базе данных.

Таблица 22.1. Соответствие действий базы данных и EJB

Инструкция базы данных	Метод EJB	Действие EJB/базы данных
INSERT	ejbCreate(), ejbPostCreate()	Создает новый экземпляр entity bean; начальное состояние определяется параметрами вызова create(). В базу данных должна быть внесена новая строка с этими значениями
SELECT	ejbLoad()	Загружает значения переменных экземпляра, считывая их из базы данных
UPDATE	ejbStore()	Сохраняет значения переменных экземпляра в базе данных
DELETE	ejbRemove()	Удаляет экземпляр entity bean; соответствующая строка в базе данных также должна быть удалена

Обратите внимание на то, что entity beans могут быть только с сохранением состояния. Различия между рассматриваемыми типами beans по сути являются не отличиями между beans с сохранением состояния и без него, а отличиями в том, *кто* отвечает за поддержание корректности состояния. В следующих двух разделах будут рассмотрены практические вопросы, связанные с каждым типом entity beans и принятием компромиссных решений об их использовании.

Хранение на уровне контейнера

Дескриптор применения entity bean указывает, что хранение состояния должно выполняться на уровне контейнера. Дескриптор также определяет отображение переменных экземпляра bean на столбцы в базе данных. Кроме того, дескриптор определяет первичный ключ, уникальным образом идентифицирующий bean и соответствующую строку базы данных. Значение первичного ключа используется в операциях, которые сохраняют и получают значения переменных из базы данных.

В случае хранения на уровне контейнера за поддержание синхронизации между entity bean и строкой базы данных отвечает EJB-контейнер. Он использует вызовы JDBC для сохранения значений переменных экземпляра в базе данных, для восстановления этих значений, для вставки новой строки в базу данных и удаления имеющейся. Перед тем как сохранять значения в базе данных, контейнер вызывает метод обратного вызова bean ejbStore() для уведомления bean о том, что он должен привести свои внутренние переменные в согласованное состояние. Аналогично после загрузки значений из базы данных контейнер вызывает метод обратного вызова bean ejbLoad(), позволяя выполнить необходимые действия (например, вычислить значения, которые не сохраняются, а вычисляются на основе значений других переменных). Точно так же вызываются еще несколько методов обратного вызова — ejbRemove() перед тем, как контейнер удалит строку из базы данных, и ejbCreate() и ejbPostCreate() при вставке новой строки. У многих entity beans эти методы обратного вызова пустые, поскольку все операции с базой данных выполняет контейнер.

Хранение на уровне bean

Если дескриптор применения entity bean указывает, что хранение выполняется на уровне bean, то контейнер полагает, что каждый entity bean самостоятельно взаимодействует с базой данных. Когда создается новый entity bean, контейнер вызывает его методы `ejbCreate()` и `ejbPostCreate()`. Bean отвечает за выполнение соответствующей инструкции базы данных `INSERT`. Аналогично при удалении entity bean контейнер вызывает его метод `ejbRemove()`. Bean сам отвечает за выполнение соответствующей инструкции базы данных `DELETE`, так что после возврата из метода `ejbRemove()` контейнер может свободно удалять bean и повторно использовать выделенную ему память.

Загрузка bean точно так же осуществляется путем вызова контейнером метода `ejbLoad()`, а сохранения — путем вызова `ejbStore()`. Кроме того, контейнер уведомляет bean о пассивизации и активизации. Конечно, ничто не ограничивает взаимодействие entity bean с базой данных только этими методами. Если bean требуется доступ к базе данных в процессе выполнения некоторого из его методов, bean может выполнять любые необходимые для этого вызовы JDBC. Вызовы JDBC в методах обратного доступа предназначаются только для решения задачи сохранения состояния.

Выбор уровня хранения

Вы, конечно, можете спросить — зачем вообще нужно хранение на уровне bean, если хранение на уровне контейнера полностью устраняет необходимость заботиться о синхронизации с базой данных? Ответ заключается в наличии ряда ограничений хранения на уровне контейнера.

- **Несколько баз данных**. У большинства серверов приложений entity beans должны отображаться на один сервер баз данных. Если данные entity bean поступают от нескольких баз данных, то единственным способом синхронизации является хранение на уровне bean.
- **Несколько таблиц**. Хранение на уровне контейнера хорошо работает тогда, когда все переменные экземпляра entity bean хранятся в одной строке одной таблицы — когда имеется взаимно однозначное соответствие между экземплярами bean и строками таблицы. Если entity bean требуется моделировать более сложный объект, такой как заголовок заказа и отдельные его строки, информация о которых хранится в разных связанных таблицах, обычно требуется хранение на уровне bean, так как только код bean в состоянии обеспечить интеллектуальную связь объекта с информацией в базе данных.
- **Оптимизация производительности**. В случае хранения на уровне контейнера последний должен исходить из принципа "все или ничего" для выбора хранимых переменных экземпляра. Каждый раз при сохранении или загрузке следует обрабатывать все переменные. Во многих приложениях, однако, entity bean может, в зависимости от текущего конкретного состояния, определить, какие именно переменные следует сохранить, — и это могут быть далеко не все переменные экземпляра. Если entity bean

содержит большое количество информации, то разница в производительности может оказаться весьма значительной.

- **Оптимизация базы данных**. Если методы entity bean, реализующие его бизнес-логику, требуют большого количества обращений к базе данных (запросов и обновлений), то некоторые из операций, которые при схеме хранения на уровне контейнера выполняет последний, могут оказаться избыточными. При хранении на уровне bean последний может оказаться в состоянии определить, какие именно операции с базой данных совершенно необходимы для синхронизации и когда база данных находится в актуальном состоянии.

На практике эти ограничения часто препятствуют применению хранения на уровне контейнера. Поэтому недаром новейшие усовершенствования спецификации EJB направлены именно на их преодоление. Тем не менее хранение на уровне bean остается очень важной технологией в современных серверах приложений.

Усовершенствования EJB 2.0

Опубликованная в апреле 2001 года спецификация EJB 2.0 представляет собой существенно переработанную версию EJB. Многие усовершенствования в EJB 2.0 оказались несовместимы с соответствующими возможностями EJB 1.x. Чтобы избежать неработоспособности совместимых с EJB 1.x beans, EJB 2.0 предоставляет дополнительные возможности в соответствующих областях, которые обеспечивают параллельное существование beans версий EJB 1.x и EJB 2.0. Полное описание отличий EJB 1.x от EJB 2.0 выходит за рамки данной книги. Однако некоторые из отличий объясняются сложностями применения хранения на уровне контейнера в спецификации EJB 1.x, и эти изменения непосредственно влияют на работу с базой данных в EJB.

Одна из трудностей при работе с EJB 1.x уже упоминалась ранее — это трудность моделирования сложных объектов, которые получают свои данные из нескольких таблиц базы данных или содержат нереляционные структуры типа массивов и иерархических данных. В EJB 1.x можно моделировать сложный объект как семейство взаимосвязанных entity beans, каждый из которых связан с одной таблицей. Такой подход позволяет использовать хранение на уровне контейнера, но связи между частями комплексного объекта при этом должны реализовываться в коде приложения. В идеале внутренние детали таких сложных объектов должны быть скрыты от прикладного кода. В качестве альтернативы можно моделировать сложный объект как единый entity bean, данные внутренних переменных которого получаются из нескольких связанных таблиц. Так достигается желаемая прозрачность прикладного кода, но хранение на уровне контейнера при получении данных из нескольких таблиц при этом затруднено.

EJB 2.0 решает этот вопрос при помощи абстрактных *методов доступа* (accessor methods), которые используются для установки и получения значения каждой хранимой переменной экземпляра в entity bean. Контейнер поддерживает память для переменных и их значений. Bean явно вызывает метод доступа `get()` для получения значения переменной экземпляра и метод доступа `set()` для его уста-

новки. Аналогично имеются абстрактные методы доступа `get()` и `set()` для каждого *отношения*, связывающего строки базы данных, которые поставляют информацию для entity bean. Отношения "многие-ко-многим" легко обрабатываются путем отображения их на переменные коллекций Java.

Эти новые возможности обеспечивают контейнер полной информацией о всех переменных экземпляра, используемых bean. Последний может представлять сложный объект, который получает данные из нескольких таблиц базы данных, скрывая детали от кода приложения. Но теперь хранение на уровне контейнера становится возможным благодаря тому, что контейнер "знает" все о разных частях объекта и об отношениях между частями.

Еще одна проблема спецификации EJB 1.x заключается в том, что хотя взаимодействия с базой данных и стандартизованы, *методы поиска*, которые используются для выявления активных entity beans, таковыми не являются. Методы поиска реализуют такие возможности, как поиск определенного entity bean по первичному ключу или поиск множества beans, отвечающих определенному критерию. Без такой стандартизации невозможно обеспечить переносимость между разными серверами приложений.

EJB 2.0 преодолевает ограничения, связанные с поиском, при помощи абстрактных *методов отбора* (select methods), которые выполняют поиск entity beans. Эти методы используют введенный в EJB 2.0 язык запросов (Query Language, EJBQL). Хотя этот язык и основан на SQL, он включает такие нереляционные конструкции, как, например, выражения путей.

Наконец, EJB 2.0 спроектирован для работы со стандартом SQL и его абстрактными типами данных. Поддержка этих типов несколько упрощает взаимодействие между entity beans и базой данных, поддерживающей абстрактные типы (в настоящее время их поддерживают только некоторые СУБД).

Усовершенствования EJB 3.0

Спецификация EJB 3.0, черновой вариант которой был опубликован в 2004 году, а окончательный — в 2006 году, заставляет контейнер выполнять большее количество работы, тем самым упрощая программирование приложений. Она снижает количество кода, который должен создать разработчик, в том числе устраняет необходимость в методах обратного вызова и упрощает программирование entity bean.

EJB 3.0 упрощает интерфейс прикладного программирования разными путями.

- В качестве альтернативы сложным дескрипторам применения можно использовать аннотации с метаданными. Последние могут использоваться для определения типов beans, настроек транзакций и безопасности, отображения объектов и т.п.
- Вместо среды EJB и ссылок на ресурсы можно применять Dependency инжекции зависимостей.
- Разработчики beans могут объявлять любой метод как метод обратного вызова, что делает излишним использование специальных методов обратного вызова.

- Методы перехвата могут быть определены в session beans (с сохранением состояния или без такового) или в beans, управляемых сообщениями. *Метод перехвата* представляет собой метод, который перехватывает вызов бизнес-метода. Вместо определения метода перехвата, в классе bean может использоваться класс-перехватчик.
- Клиенты могут непосредственно вызывать метод EJB без создания экземпляра bean.

Session beans также усовершенствованы и упрощены в нескольких направлениях.

- Session beans EJB 3.0 стали проще, так как они представляют собой чистые классы Java, которые не должны реализовывать интерфейсы session bean. Session bean может иметь удаленный, локальный или и локальный, и удаленный интерфейсы.
- Для определения bean или интерфейса и свойств времени выполнения session beans могут использоваться аннотации с метаданными.
- Для session beans (с сохранением состояния или без такового) поддерживаются методы-слушатели обратного вызова.
- Для инжекции объектов среды, ресурсов или контекста EJB разработчики могут использовать метаданные или дескрипторы применения.
- Методы или классы перехвата поддерживаются как для session beans с сохранением состояния, так и без него.

Управляемые сообщениями beans усовершенствованы и упрощены в нескольких направлениях.

- Они могут реализовывать интерфейс `MessageListener` вместо интерфейса `MessageDriven`.
- Метаданные упрощают спецификацию bean или интерфейса и свойств времени выполнения.
- Поддерживаются методы-слушатели обратного вызова.
- Вместо дескрипторов применения можно использовать зависимости.
- Могут использоваться методы или классы перехвата.

Entity beans и API хранения могут быть усовершенствованы и упрощены несколькими способами. Во-первых, EJB 3.0 стандартизирует модель хранения POJO (Plain Old Java Object, старые простые объекты Java) и упрощает entity beans, которые становятся конкретными классами Java, не требующими никаких интерфейсов. Классы entity bean непосредственно поддерживают полиморфизм и наследование. Во-вторых, EJB 3.0 включает новый EntityManager API для создания, поиска, удаления и обновления объектов. В-третьих, вместо дескрипторов применения можно использовать аннотации, что существенно упрощает разработку объектов. В-четвертых, существенно усилены возможности языка запросов. В-пятых, поддержка методов-слушателей обратного вызова обеспечивается применением методов, указанных в аннотациях либо дескрипторах применения. Наконец, процессор хранения в EJB 3.0 может использоваться и вне контейнера.

Разработка приложений с открытым кодом

Развитие Интернета привело к появлению множества новинок в области серверов приложений, распространения приложений, а также их кооперации с SQL-базами данных. В этих областях активно работают и разработчики программного обеспечения с открытым кодом. Например, наиболее широко используемым веб-сервером с открытым кодом является Apache. Для создания веб-приложений разработчики часто используют комбинацию асинхронных сценариев JavaScript и XML, известную как Ajax (это слово можно рассматривать и как аббревиатуру AJAX — Asynchronous JavaScript and XML, асинхронный JavaScript и XML). Вместо JavaScript могут использоваться и другие языки сценариев, такие как PHP; применение XML не является совершенно необходимым с технической точки зрения, так что аббревиатура AJAX менее предпочтительна, чем просто имя Ajax[1]. Хотя сама аббревиатура появилась только в 2005 году, асинхронная загрузка веб-содержимого уходит корнями в средину 1990-х годов.

Веб-приложения, использующие Ajax, могут получать данные асинхронно, при помощи фонового процесса, не влияющего на вид и поведение существующей веб-страницы. Данные получаются при помощи объекта `XMLHttpRequest` или (в браузерах, не поддерживающих объекты) с использованием удаленных сценариев. Базы данных, используемые вместе с Ajax для сохранения и загрузки долговременно хранимых данных, практически всегда представляют собой реляционные продукты с открытым кодом, такие как MySQL.

Еще одной популярной платформой для распространения информации и приложений посредством веб-страниц является LAMP (Linux, Apache, MySQL и PHP/Python/Perl). Фактически, большая часть явления, известного как "Web 2.0", построена на платформе LAMP, которая обеспечивает дешевую инфраструктуру на базе SQL для веб-приложений. В настоящее время ведущую роль в Интернете играет именно LAMP и ее вариации — например, с возможными новыми компонентами (в качестве примера можно привести LAMR — LAMP с заменой языка сценариев на Ruby on Rails (ROR)). С ростом популярности служб на базе Интернета, таких как Infrastructure as a Service (IaaS), Platform-as-a-Service (PaaS) и Software as a Service (SaaS), технологии серверов приложений LAMP продолжают развиваться и эволюционировать. Все эти службы в своей массе опираются на LAMP и ее непосредственных преемников.

Серверы приложений и кеширование

Узким местом веб-сайтов большого объема, ограничивающим их производительность, может стать обращение к базам данных. Структура EJB по мере усложнения бизнес-логики приводит к росту обращений к базе данных для поддержки синхронизации между entity bean и базой данных. Если веб-сайт реализует высокую степень персонализации (т.е. большая часть страниц генерируется динамически на основе информации о конкретном пользователе, запросившем их), то ко-

---

[1] *Имеется в виду Аякс — греческий герой времен Троянской войны.* — Примеч. пер.

личество обращений к базе данных растет с еще большей скоростью. В предельном случае каждый щелчок пользователя на странице такого сайта может требовать обращения к базе данных за информацией о пользователе. Наконец, работа пользователей осуществляется в режиме реального времени, так что главную роль играет пиковая, а не средняя нагрузка на сайт. Среднее время обработки каждого щелчка пользователя менее важно, чем время обработки при пиковой нагрузке — время, из-за которого раздраженные пользователи будут называть сайт "тормознутым" или как-то похоже (и не менее неприятно).

В веб уже имеется эффективное решение проблемы с пиковой нагрузкой — кеширование веб-страниц и горизонтальное масштабирование. При кешировании копии страниц с большим количеством обращений рассылаются по сети и дублируются. В результате общая пропускная способность для данных веб-страниц повышается, а сетевой траффик, связанный с ними, снижается. При горизонтальном масштабировании содержимое веб-сайта воспроизводится на двух или большем количестве серверов (до десятков, а в тяжелых случаях — и до сотен серверов), полная мощность которых в плане обработки страниц оказывается существенно выше, чем у одного сервера.

Аналогичное кеширование и горизонтальное масштабирование используются и для повышения пропускной способности серверов приложений. Большинство нынешних коммерческих серверов приложений реализует *кеширование beans*, при котором в памяти сервера приложений содержатся копии наиболее часто используемых entity beans. Кроме того, серверы приложений часто используются в банках или кластерах, когда каждый сервер приложений обеспечивает одинаковую бизнес-логику и одинаковые возможности обработки приложений. Многие коммерческие серверы приложений используют горизонтальное масштабирование в пределах одного сервера, что обеспечивает эффективное использование симметричной многопроцессорности (SMP). Так, для восьмипроцессорного сервера совершенно типична ситуация, когда параллельно работают восемь независимых копий серверного программного обеспечения. На рис. 22.6 показана конфигурация типичного сервера приложений, состоящего их трех четырехпроцессорных серверов.

К сожалению, горизонтальное масштабирование и кеширование противоречат друг другу при работе с данными с сохранением состояния, такими как хранящиеся в entity bean или базе данных. Без специальной логики синхронизации кеша, обновления bean в кеше одного сервера не будут автоматически появляться во всех кешах, что потенциально может привести к некорректным и несогласованным результатам. Рассмотрим, например, что произойдет с данными о доступности товара на складе, если три или четыре различных кеша содержат копии entity bean для одного и того же товара и бизнес-логика сервера приложений обновляет эти значения. Кеши очень быстро станут содержать различные доступные количества товара, причем ни одно из них не будет точным. К сожалению, логика синхронизации кешей, необходимая для обнаружения и предупреждения такого рода ситуаций, сопровождается большими накладными расходами. Абсолютная синхронизация требует применения двухфазного протокола принятия изменений (который будет рассматриваться в главе 23, "Сети и распределенные базы данных").

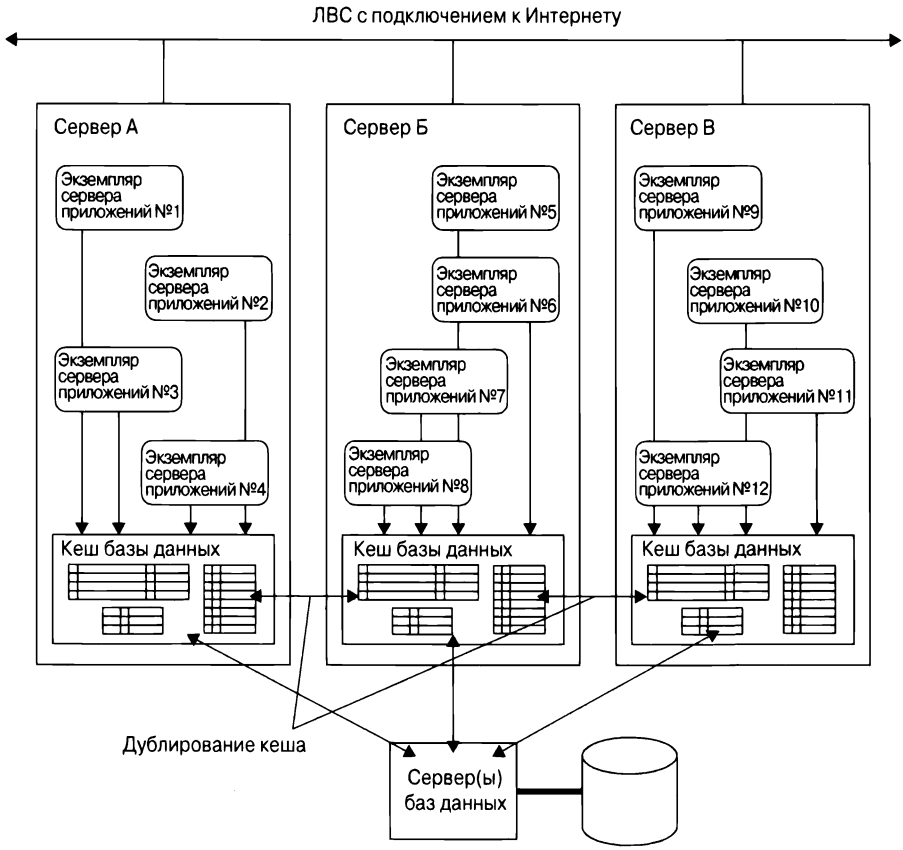


Рис. 22.6. Серверы приложений и кеширование EJB

Кеши баз данных могут решать проблему множественных кешей beans в пределах одного SMP-сервера, как показано на рис. 22.7. При кешировании на уровне базы данных, вместо кеширования на уровне beans, один кеш базы данных может обеспечить согласованность всех экземпляров серверов приложений на одном сервере. Однако при этом все равно останется проблема синхронизации между различными физическими серверами. Если отношение количества чтений базы данных к количеству обновлений базы данных велико (как, например, в случае высоко персонализированного веб-сайта), то накладные расходы на синхронизацию кешей могут остаться относительно малыми, а выигрыш от горизонтального масштабирования — значительным.

Oracle использовала кеширование базы данных в своем Oracle Application Server и пыталась использовать это кеширование как конкурентное преимущество. IBM тут же объявила о планах добавить интегрированное кешировании базы данных в своей СУБД DB2, но пока что это еще не сделано. Имеется ряд продуктов сторонних производителей, призванных работать в качестве кешей баз данных для серверов приложений, в том числе для некоторых объектно-ориентированных баз данных. Однако вопрос о влиянии этой технологии на рынок серверов приложений пока остается открытым.

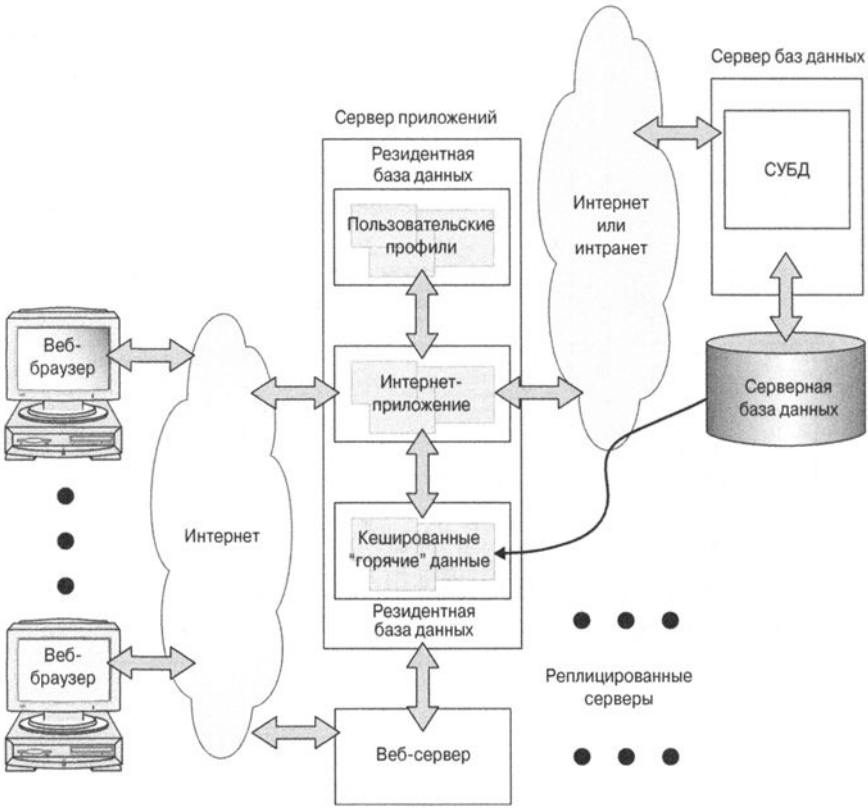


Рис. 22.7. Серверы приложений и кеширование базы данных

Резюме

В этой главе описываются серверы приложений и роль, которую они играют в связке веб с корпоративными системами, в том числе с корпоративными базами данных.

- Популярные серверы приложений реализуют спецификацию J2EE, которая стандартизирует обращения к базам данных посредством JDBC API.
- Бизнес-логика в сервере приложений реализуется при помощи EJBs, а именно — при помощи session beans или entity beans.
- Session beans представляют собой воплощение пользовательских сессий. Они могут непосредственно обращаться к базам данных с применением вызовов JDBC.
- Session beans без сохранения состояний поддерживают очень простые транзакции — при каждом вызове выполняется одно обращение к базе данных.

- Beans с сохранением состояний поддерживают транзакции, охватывающие несколько вызовов, но их логика должна обеспечивать сохранение состояния при пассивизации и его восстановление при активизации.
- Entity beans представляют собой воплощение объектов реального мира и соответствуют строкам таблиц базы данных. Они могут быть только с сохранением состояния.
- Entity beans могут использовать хранение на уровне контейнера, когда сервер приложений автоматически обеспечивает синхронизацию между entity bean и базой данных.
- При хранении на уровне beans последние могут брать ответственность за синхронизацию с базой данных на себя.
- В области интегрирования баз данных с веб-сайтами продолжается рост популярности таких архитектур с открытым кодом, как Ajax, LAMP и LAMR.

23
ГЛАВА

Сети и распределенные базы данных

За несколько последних десятилетий компьютерные сети радикально преобразовали корпоративные вычислительные системы. В большинстве компаний каждый персональный компьютер подключен к локальной вычислительной сети (ЛВС). Мощные серверы рабочих групп, подключенные к ЛВС, отвечают вычислительным нуждам конкретных отделов. Корпоративные сети соединяют ЛВС одного здания (или комплекса зданий типа университетского городка) и подключают их к региональным или корпоративным центрам данных. Дополнительные связи служат для объединения корпоративных центров по всему миру. Интернет представляет собой сеть сетей, связывая компании одну с другой и с отдельными пользователями. В этой сетевой среде прикладные программы могут работать на компьютерах любого уровня и в любом месте.

В этой новой сетевой среде компьютерные данные уже не размещаются на отдельной машине под управлением одной СУБД. Вместо этого данные в организации разбросаны среди множества различных систем, каждая со своим администратором баз данных. Зачастую это различные компьютерные системы и системы управления базами данных разных производителей. Когда компании пытаются объединить свои системы обработки данных через Интернет, проблема становится еще более сложной. Даже если в компании действует единый стандарт на СУБД и структуры базы данных, то это не значит, что он будет распространен и на поставщиков или клиентов, с которыми будут строиться внешние связи для создания единой электронной системы управления бизнесом.

Описанные тенденции привели к тому, что в компьютерной промышленности внимание специалистов по обработке данных сосредоточилось на проблемах управления базами данных в сетевой среде. В настоящей главе рассматриваются задачи, возникающие при работе с распределенными данными, различные архитектурные решения, позволяющие решать эти задачи, и некоторые конкретные продукты, предлагаемые производителями СУБД для этой цели.

Проблемы управления распределенными данными

На момент появления в 1970-х годах реляционных баз данных и языка SQL практически все коммерческие системы обработки данных имели централизованную архитектуру. Корпоративные данные хранились на массивных дисковых накопителях, подключенных к центральной ЭВМ. Приложения, связанные с обработкой транзакций и генерацией отчетов, выполнялись на центральной машине и там же обращались к данным. Большая часть операций осуществлялась в пакетном режиме. Пользователи получали доступ к системе с помощью простейших "неинтеллектуальных" алфавитно-цифровых терминалов, не имеющих собственных вычислительных мощностей. Центральный компьютер отвечал за форматирование данных, выдаваемых на экран терминала, и прием данных с терминала.

В такой среде роль реляционных баз данных и языка SQL была четко обозначена. СУБД отвечала за прием, хранение и извлечение информации на основе запросов, формулируемых с помощью SQL. Правила бизнес-логики определялись *вне* базы данных в бизнес-приложениях, разрабатываемых сотрудниками информационной службы предприятия. СУБД и все программы работали на том же центральном компьютере, где хранились данные, поэтому производительность системы не зависела от внешних факторов, в частности от сетевого трафика и сбоев внешних устройств.

Современные коммерческие информационные системы мало напоминают централизованные архитектуры 70-х годов. На рис. 23.1 схематически изображена часть компьютерной сети, которую можно встретить на производственном предприятии или фирме, специализирующейся на оптовой торговле. Данные хранятся в разнообразных вычислительных системах, объединенных в сеть.

- **Мэйнфреймы (большие ЭВМ)**. Основные приложения по обработке данных этого предприятия, например программы бухгалтерского учета и начисления зарплаты, выполняются на мэйнфрейме производства компании IBM. Старые приложения, разработанные и эксплуатируемые в течение последних 20–30 лет, по-прежнему хранят информацию в иерархических базах данных IMS. Компания планирует со временем перенести эти базы данных в СУБД DB2; именно для этой СУБД разрабатываются все новые приложения. Заметим, что со временем различия между мэйнфреймами и серверами стираются — по мере того как серверы становятся все больше, а мэйнфреймы — все меньше.
- **Рабочие станции и серверы Unix**. Инженерные службы компании используют для своих нужд рабочие станции и серверы Unix (производства Sun Microsystems). Результаты технических испытаний и спецификации хранятся в базе данных Oracle. Компания использует базы данных Oracle для управления заказами и учета складских запасов, а также в шести дистрибьюторских центрах компании, где работают серверы под управлением Linux. Их применение оказалось достаточно успешным для того, чтобы принять решение о дополнительном развертывании серверов Linux в ближайшем будущем.

- **Серверы ЛВС**. Во всех отделах компании организованы локальные сети, с помощью которых осуществляется совместный доступ к файлам и принтерам. В некоторых отделах ведутся собственные базы данных. Например, отдел кадров приобрел программное обеспечения учета кадров и для хранения своих данных использует СУБД SQL Server на платформе Windows 2008. В плановом отделе используется собственная программа планирования, взаимодействующая с СУБД Informix Universal Server.
- **Персональные компьютеры**. Все служащие компании имеют в своем распоряжении персональные компьютеры. Многие начальники ведут личные базы данных, пользуясь для этого электронными таблицами Excel, Microsoft Access или одной из персональных СУБД типа Oracle Personal Edition.
- **Переносные компьютеры**. Компания недавно приобрела пакет автоматизации торговых операций и снабдила каждого торгового менеджера переносным компьютером, на котором выполняются презентации, обрабатывается почта, а также хранится локальная упрощенная копия базы данных (управляемая СУБД SQL Anywhere компании Sybase) с последней информацией о ценах на продукцию компании и о наличии товаров на складах. В базе данных также накапливается информация о заказах, принятых менеджером. В конце рабочего дня через Интернет менеджер подключается к сети компании, и ночью компьютер передает сведения о принятых заказах и обновляет локальную копию базы данных.
- **Наладонные устройства**. Менеджеры компании широко используют персональные устройства с возможностью подключения к Интернету (смартфоны). В дополнение к персональному календарю и адресной книге, приложения на смартфоне могут использовать беспроводные подключения для проверки цен или ввода заказов. Беспроводная сеть может использоваться и для оповещения пользователей через их смартфоны о важных изменениях в базе данных, — например, о повышении цен или о недостатке товара на складе.
- **Подключения к Интернету**. У компании есть свой веб-сервер, на котором клиенты, дилеры и дистрибьюторы могут получить последнюю информацию о продуктах и услугах, предлагаемых компанией. Поначалу это был простой информационный сервер, но недавно конкуренты стали принимать заказы через Интернет, поэтому первоочередной задачей компании является оснащение своего сервера средствами электронной коммерции.

В связи с тем что данные распределены по различным системам, нетрудно представить себе запросы, связанные с обращением сразу к нескольким базам данных:

- инженерам необходимо объединять результаты лабораторных испытаний (хранящиеся на рабочих станциях) с производственными планами-прогнозами (хранящимися на мэйнфрейме в центральной базе данных) для выбора одной из трех альтернативных технологий;

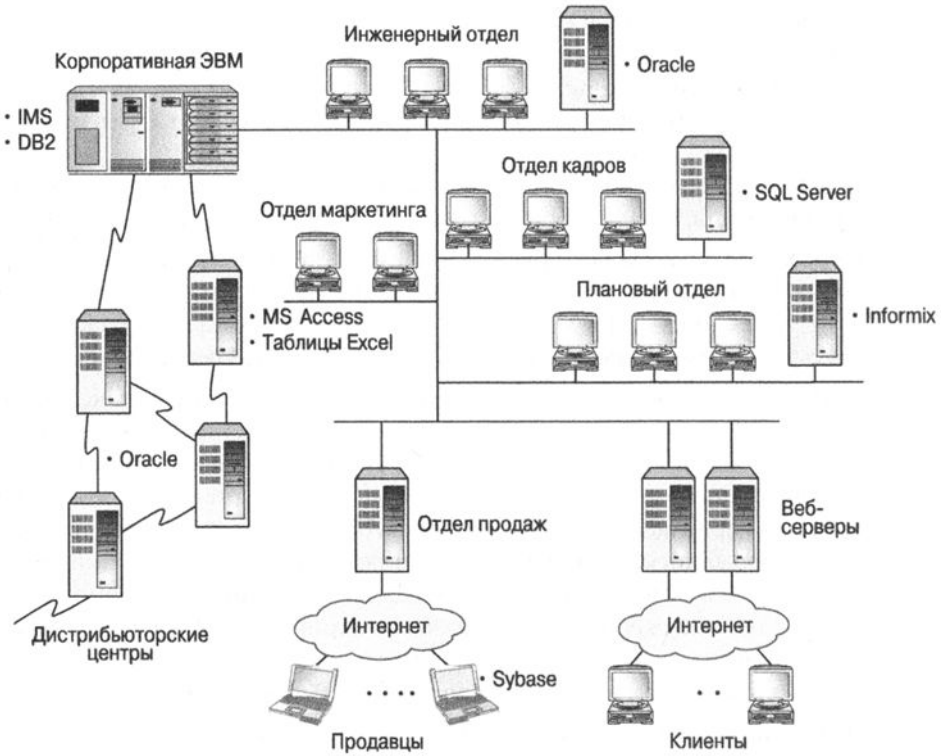


Рис. 23.1. Применение СУБД в типичной корпоративной сети

- служащим планового отдела необходимо связывать финансовые планы-прогнозы (хранящиеся в базе данных Informix) с финансовыми отчетами за прошедшие периоды (хранящимися в центральной базе данных);
- начальнику отдела сбыта необходимо знать запасы каждого товара на каждом оптовом складе (данные хранятся на шести серверах Linux), чтобы планировать их поставки;
- ежедневно из центральной базы данных необходимо рассылать информацию о текущих ценах в базы данных дистрибьюторских центров, а также в локальные базы данных, находящиеся на портативных компьютерах продавцов;
- данные о принятых заказах должны ежедневно поступать с портативных компьютеров продавцов в центральную базу данных и рассылаться в базы данных дистрибьюторских центров; суммарная накопленная информация о заказах должна поступать из дистрибьюторских центров в центральную базу данных, чтобы на основании полученных сведений можно было корректировать производственные планы;
- продавцы могут принимать заказы от клиентов и планировать даты поставки товаров, пользующихся наибольшей популярностью, на основании информации, содержащейся в их локальных базах данных, не подоз-

ревая, что другие менеджеры могут в это же время принимать заказы на те же товары; следовательно, все принимаемые заказы должны быть согласованы с текущими складскими запасами, а в случае необходимости — должны быть определены приоритеты, и если сроки поставок изменяются, клиентов следует уведомить об этом;

- начальники отделов со своих персональных компьютеров могут выполнять запросы на выборку информации из различных баз данных, доступных для совместного использования.

Ведущие поставщики СУБД предлагают множество продуктов, позволяющих эффективно управлять распределенными данными и решать описанные выше задачи. Кроме того, вопросы управления распределенными данными были предметом интенсивных исследований в университетской и корпоративной среде, было написано множество статей теоретического характера, описывающих круг возникающих проблем и способы их решения. В результате удалось достигнуть общего соглашения относительно "идеальных" характеристик системы управления распределенными базами данных.

- **Прозрачность местоположения**. Пользователь не должен беспокоиться о том, где физически располагаются данные. СУБД должна представлять все данные так, как если бы они были локальными, и отвечать за сохранение этой иллюзии.
- **Гетерогенные системы**. СУБД должна работать с данными, которые хранятся в различных системах с разной архитектурой и производительностью, включая персональные компьютеры, рабочие станции, серверы ЛВС, мини-компьютеры и мэйнфреймы.
- **Сетевая прозрачность**. За исключением различий в производительности, СУБД должна работать одинаково в различных сетях, от высокоскоростных ЛВС до низкоскоростных телефонных линий.
- **Распределенные запросы**. Пользователь должен иметь возможность объединять данные из любых таблиц (распределенной) базы данных, даже если эти таблицы физически расположены в различных системах.
- **Распределенные обновления**. Пользователь должен иметь возможность изменять данные в любой таблице, на доступ к которой у него есть необходимые привилегии, независимо от того, находится ли эта таблица в локальной или удаленной системе.
- **Распределенные транзакции**. СУБД должна выполнять транзакции (используя инструкции COMMIT и ROLLBACK), выходящие за границы одной вычислительной системы, и поддерживать целостность (распределенной) базы данных даже при возникновении отказов как в отдельных системах, так и в сети в целом.
- **Безопасность**. СУБД должна обеспечивать защиту всей (распределенной) базы данных от несанкционированного доступа.
- **Универсальный доступ**. СУБД должна обеспечивать единую методику доступа ко всем данным предприятия.

Ни одна из существующих в настоящее время распределенных СУБД по своим возможностям не соответствует этому идеалу. Имеются препятствия, из-за которых с трудом реализуются даже простые формы управления распределенными базами данных. К ним относятся следующие.

- **Производительность**. В централизованной базе данных время доступа к данным составляет несколько миллисекунд, а скорость их передачи — несколько миллионов символов в секунду. Даже в высокоскоростной локальной сети время доступа увеличивается до сотых и десятых долей секунды, а скорость передачи данных падает до 100000 символов в секунду или даже еще ниже. Время доступа к данным по телефонной линии может занимать секунды или минуты, а максимальная пропускная способность уменьшается до нескольких тысяч символов в секунду. Эта огромная разница в быстродействии может резко замедлять доступ к удаленным данным.
- **Целостность**. Чтобы при выполнении распределенных транзакций соблюдался принцип "все или ничего", необходимо активное взаимодействие двух или более независимых СУБД, работающих в различных вычислительных системах. При этом должны использоваться специальные протоколы двухфазного завершения транзакций, которые приводят к повышению сетевого трафика и длительной блокировке тех частей баз данных, которые участвуют в распределенной транзакции.
- **Статический SQL**. Встроенная статическая инструкция SQL компилируется и сохраняется в базе данных в виде плана выполнения. Когда запрос объединяет данные из двух или более баз данных, в какой из них следует хранить план выполнения? Может быть, необходимо иметь два согласованных плана или более? А если изменяется структура одной базы данных, то как можно изменить план выполнения в другой базе данных? Применение динамического SQL в сетевой среде для решения этой проблемы практически всегда ведет к неприемлемому снижению производительности приложений из-за повышения сетевого трафика и многочисленных задержек.
- **Оптимизация**. Когда доступ к данным осуществляется по сети, обычные правила оптимизации инструкций SQL применять нельзя. Например, полное сканирование локальной таблицы может оказаться оптимальнее, чем поиск по индексу в удаленной таблице. Программа оптимизации должна знать параметры сети и, в частности, ее быстродействие. Если говорить в общем, роль оптимизации становится более важной, а ее осуществление более трудным.
- **Совместимость данных**. В различных вычислительных системах существуют разные типы данных, и даже когда в двух системах присутствуют данные одного и того же типа, они могут иметь разные форматы. Например, Windows PC и Apple Mac хранят 16-разрядные целые числа поразному. Для представления символов в мэйнфреймах компании IBM используется кодировка EBCDIC, а в персональных компьютерах и сер-

верах на базе Unix и Linux — кодировка ASCII. В распределенной СУБД эти различия должны быть незаметны.

- **Системные каталоги**. Во время работы СУБД часто обращается к своим системным каталогам. Где в распределенной базе данных следует хранить каталог? Если он будет храниться в одной системе, то удаленный доступ к каталогу будет медленным, что может парализовать работу СУБД. Если расположить его в нескольких различных системах, то изменения в каталоге придется распространять по сети и синхронизировать.
- **Оборудование от разных поставщиков**. Вряд ли управление всеми данными на предприятии будет осуществляться с помощью СУБД одного типа; как правило, в распределенной базе данных используется несколько СУБД, что требует активной совместной работы СУБД, поставляемых конкурирующими компаниями. Но такое маловероятно.
- **Распределенные взаимоблокировки**. Когда в двух системах одновременно выполняются транзакции, которые пытаются осуществить доступ к заблокированным данным в другой системе, в распределенной базе данных может возникнуть взаимоблокировка, хотя в каждой из двух систем ее не будет. СУБД должна обеспечивать обнаружение глобальных взаимоблокировок в распределенной базе данных. Это требует координации усилий сетевых приложений и обычно ведет к резкому снижению производительности.
- **Восстановление**. Если в одной из систем, входящих в распределенную базу данных, произойдет сбой, то администратор этой системы должен иметь возможность запустить процедуру восстановления независимо от других вычислительных систем в сети, и восстановленная система должна быть синхронизирована с другими системами.

Практические подходы к управлению распределенными базами данных

Из-за указанных препятствий поставщики СУБД вводят управление распределенными данными последовательно, шаг за шагом. Они вынуждены сосредоточивать усилия на определенных видах сетевого доступа к базе данных, распределения данных и управления ими, которые наилучшим образом подходят для определенного сценария. Например, сначала в СУБД может появиться модуль, позволяющий извлекать набор записей из главной базы данных и рассылать его по сети в подчиненные базы данных. Затем этот модуль может быть усовершенствован и позволит отслеживать изменения в главной базе данных, происшедшие с момента последней выборки, чтобы можно было загружать только изменения.

В следующих версиях СУБД модуль уже будет располагать средствами автоматического управления всем процессом, предоставляя при этом графический пользовательский интерфейс для указания извлекаемых данных и сценарии для автоматизации периодического процесса извлечения данных. Точно так же СУБД мо-

жет первоначально позволять пользователю одной системы делать запросы к базе данных, размещенной в другой системе. В новой версии СУБД будет разрешено включать запрос к удаленной базе данных в качестве подчиненного в запрос к локальным таблицам. В последующих версиях возможности управления распределенными данными будут расширяться, приближаясь к поставленной цели — обеспечению универсального и прозрачного доступа к данным.

Доступ к удаленным базам данных

Одним из простейших подходов к управлению данными, хранящимися в разных местах, является удаленное обращение к данным. В этом случае пользователю одной базы данных предоставляется возможность подключаться по сети к другой базе данных и извлекать из нее информацию. В простейшем случае это означает выполнение одиночного запроса к удаленной базе данных (рис. 23.2). Кроме запросов на выборку, это могут быть запросы на добавление, обновление и удаление. Такая схема работы часто применяется, когда локальная база данных является периферийной (например, эта база данных расположена в региональном офисе или дистрибьюторском центре), а удаленная — центральной корпоративной базой данных.

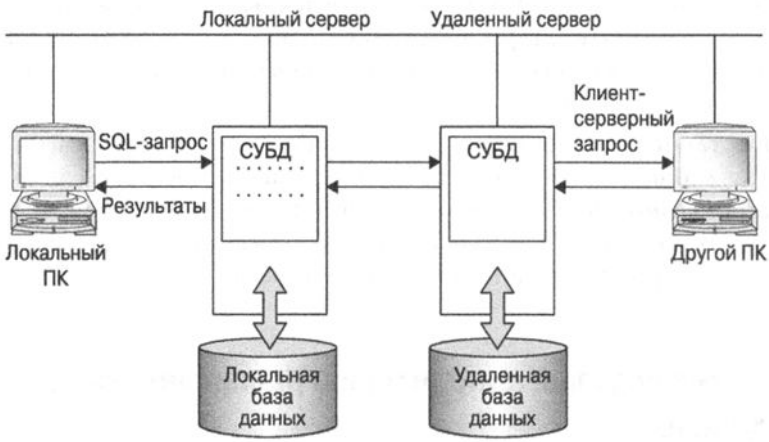


Рис. 23.2. Доступ к удаленной базе данных

В дополнение к запросам к удаленным данным, на рис. 23.2 показан клиент-серверный запрос к удаленной базе данных от пользователя другого ПК. Заметим, что, с точки зрения удаленной базы данных, разница между обработкой запроса от обычного клиентского ПК и запроса, полученного по специальной технологии удаленного доступа, очень невелика. В обоих случаях запрос передается по сети, удаленная СУБД определяет, имеет ли пользователь нужные привилегии, и выполняет запрос. Информация о том, чем завершилось выполнение запроса, передается обратно по сети.

А вот локальная СУБД, участвующая в процессе, изображенном на рис. 23.2, должна выполнять совершенно иные действия, чем при выполнении обычного локального запроса.

- Должна определить, к какой удаленной базе данных обращается пользователь и как найти ее в сети.
- Должна установить соединение с этой базой данных для выполнения удаленного запроса.
- Должна определить, как локальная схема привилегий и аутентификации пользователя соотносится с аналогичной схемой удаленной СУБД. Это значит, что ей нужно выяснить, достаточно ли просто предать удаленной СУБД имя пользователя и пароль, с которыми он подключился к локальной базе данных, или для удаленного доступа нужны другие имя пользователя и пароль, и если да, то должны ли они быть определены автоматически или введены пользователем.

По сути, локальная СУБД становится агентом пользователя. Она организует стандартное клиент-серверное подключение к удаленной СУБД, в котором сама выступает в качестве клиента.

Несколько ведущих производителей СУБД уровня предприятия предлагают различные реализации описанной схемы удаленного доступа. Они отличаются деталями организации работы пользователя и администратора базы данных. В одних случаях СУБД поддерживает специальные расширения языка SQL, в других же механизм установки соединения практически не связан с SQL.

Например, Sybase Adaptive Server Enterprise (ASE) позволяет очень просто получить доступ к удаленной базе данных. Подключившись к локальному серверу Sybase, пользователь может направить ему инструкцию `CONNECT TO`, в которой должен указать известное локальному серверу имя удаленного сервера. Например, если имеется удаленный сервер с именем `CENTRALHOST`, то инструкция

```
CONNECT TO CENTRALHOST
```

делает его текущим сервером сеанса. После выполнения этой инструкции локальный сервер входит в режим сквозной передачи и направляет все последующие инструкции удаленному серверу. Теперь пользователь может работать с удаленной базой данных через установленное соединение с помощью обычных инструкций SQL.

Получить имена и продажи всех продавцов, выполнивших план продаж.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS
 WHERE SALES > QUOTA;
```

Когда работа с удаленным сервером окончена, нужно всего лишь выполнить инструкцию `DISCONNECT`. При этом режим сквозной передачи будет отменен, а локальный сервер снова станет текущим. За исключением пары инструкций `CONNECT/DISCONNECT`, весь механизм управления удаленным сервером реализован вне языка SQL. Администратор базы данных предоставляет локальному серверу всю необходимую информацию о наличии удаленных серверов, их местоположении и именах посредством системных хранимых процедур `spaddserver()` и `spdropserver()`. По умолчанию для доступа к удаленному серверу используются текущие имя и пароль пользователя, но в качестве альтернативы администратор базы данных может задать для удаленного сервера специальные имя и па-

роль пользователя, опять же посредством системных хранимых процедур. Sybase предлагает и другие, более сложные возможности управления распределенными базами данных, но описанный базовый механизм обладает неоценимым преимуществом — предельной простотой.

В Oracle используется иная схема организации доступа к удаленным базам данных: необходимо, чтобы как на локальном, так и на удаленном сервере, помимо СУБД, было установлено сетевое программное обеспечение SQL*Net. Администратор базы данных отвечает за создание одного или нескольких именованных *каналов связи* (между локальной и удаленными базами данных). Каждый канал определяет следующее:

- местоположение удаленного компьютера в сети;
- используемый коммуникационный протокол;
- имя базы данных Oracle на удаленном сервере;
- имя и пароль пользователя для подключения к удаленной базе данных.

Весь доступ к удаленным базам данных осуществляется через каналы связи и ограничивается привилегиями, предоставленными пользователю в удаленной системе. Можно сказать, что определение канала связи содержит ответы на такие вопросы: "какая база данных", "как с ней взаимодействовать" и "каковы привилегии пользователя". Администратор назначает каждому каналу собственное имя. Каналы связи могут быть *закрытыми* (созданными для конкретного пользователя локальной системы) или *открытыми* (доступными для многих пользователей локальной системы).

Для доступа к удаленной базе данных пользователь локальной системы применяет стандартные инструкции SQL, только к именам удаленных таблиц и представлений добавляется символ '@', за которым следует имя канала связи. Предположим, например, что вы работаете с учебной базой данных через канал с именем CENTRALHOST. Следующая инструкция возвращает информацию из таблицы SALESREPS.

Получить имена и объемы продаж всех служащих, опережающих план.

```
SELECT NAME, QUOTA, SALES
  FROM SALESREPS@CENTRALHOST
 WHERE SALES > QUOTA;
```

При работе с удаленными данными поддерживаются практически все базовые локальные возможности Oracle. Единственным обязательным требованием является то, что имя каждого объекта удаленной базы данных должно сопровождаться именем канала связи. Вот пример соединения двух таблиц, выполняемого на удаленном сервере Oracle.

Получить имена всех служащих, опережающих план, и узнать, в каких офисах они работают.

```
SELECT NAME, CITY, QUOTA, SALES
  FROM SALESREPS@CENTRALHOST, OFFICES@CENTRALHOST
 WHERE SALES > QUOTA
   AND REP_OFFICE = OFFICE;
```

Informix Universal Server располагает возможностями, подобными возможностям Oracle, но использует иной механизм идентификации удаленных баз данных и немного другой расширенный синтаксис SQL. В архитектуре Informix различаются удаленный *сервер* базы данных и удаленная *база данных*, управляемая этим сервером. Так гораздо удобнее работать с несколькими базами данных, расположенными на одном удаленном сервере. Предположим, что копия учебной базы данных называется SAMPLE и располагается на удаленном сервере с названием CENTRALHOST. В этом случае эквивалентом приведенных выше примеров для Oracle и Sybase будет следующий запрос.

Получить имена и объемы продаж всех служащих, опережающих план.

```
SELECT NAME, QUOTA, SALES
  FROM SAMPLE@CENTRALHOST:SALESREPS
 WHERE SALES > QUOTA;
```

Имя базы данных указывается перед именем таблицы и отделяется от него двоеточием. Если база данных является удаленной, за ее именем следует еще символ @ и имя сервера.

Прозрачность доступа к удаленным данным

Достаточно написать лишь несколько запросов, чтобы почувствовать, насколько это неудобно и утомительно — при работе с удаленными базами данных имена таблиц и представлений уточнять именем базы данных, канала связи, сервера и т.п. Например, если в двух таблицах удаленной базы данных имеются столбцы с одинаковыми именами, в любом запросе, извлекающем данные из обеих таблиц, должны использоваться префиксы имен столбцов — имена таблиц, в которых, в свою очередь, должны быть префиксы для удаленного доступа. Вот полное имя столбца NAME в удаленной таблице SALESREPS, принадлежащей пользователю JOE и хранящейся в удаленной базе данных SAMPLE на удаленном сервере CENTRALHOST в СУБД Informix.

```
SAMPLE@CENTRALHOST.JOE.SALESREPS.NAME
```

Ссылка на один-единственный столбец заняла полстроки! Поэтому неудивительно, что в инструкциях SQL для доступа к удаленным базам данных очень часто используются псевдонимы таблиц. Синонимы и псевдонимы (о которых рассказывалось в главе 16, "Системный каталог") помогают обеспечить более прозрачный доступ к удаленной базе данных. Вот пример синонима, который может быть определен пользователем или администратором базы данных.

```
CREATE SYNONYM REMOTE_REPS FOR SAMPLE@CENTRALHOST.JOE.SALESREPS;
```

Эквивалентное определение синонима в Oracle выглядит так.

```
CREATE SYNONYM REMOTE_REPS FOR JOE.SALESREPS@CENTRALHOST;
```

Определив такой синоним, можно сократить приведенное выше имя столбца.

```
REMOTE_REPS.NAME
```

Теперь любой запрос, в котором имеются ссылки на таблицу REMOTE_REPS и ее столбцы, на самом деле будет направлен удаленной базе данных, но этот факт для пользователя незаметен. Большинство коммерческих СУБД поддерживает как открытые синонимы (доступные всем пользователям), так и закрытые (созданные для конкретного пользователя или группы пользователей). При этом синонимы могут стать дополнительной частью механизма защиты и предоставляться только тем пользователям, которым разрешен доступ к удаленным таблицам.

Некоторые производители СУБД идут еще дальше и позволяют в локальной базе данных создавать представления на основе удаленных таблиц. Вот пример инструкции Oracle, создающей представление EAST_REPS.

Создание представления на основе двух удаленных таблиц.

```
CREATE VIEW EAST_REPS AS
     SELECT EMPL_NUM, NAME, AGE, CITY
       FROM SALESREPS@CENTRALHOST, OFFICES@CENTRALHOST
      WHERE REP_OFFICE = OFFICE
        AND REP_OFFICE BETWEEN 11 AND 19;
```

Если в базе данных определено такое представление, пользователь может адресовать к нему запросы, не заботясь о том, куда на самом деле они будут направляться, — ему не нужно думать ни о каналах связи, ни об именах удаленных таблиц. Представление не только обеспечивает прозрачный доступ к удаленным данным, но и скрывает от пользователя операцию соединения таблиц OFFICES и SALESREPS.

Прозрачный доступ к удаленным данным, обеспечиваемый представлениями и синонимами, обычно считается очень полезной характеристикой СУБД. Однако у него есть один недостаток. Поскольку факт удаленного доступа скрыт от пользователя, пользователь не предполагает наличие накладных расходов, связанных с таким доступом. В результате пользователь или программист может непреднамеренно направить удаленному серверу запрос, возвращающий очень большое количество записей, и резко повысить сетевой трафик. Создавая синонимы и представления, администратор базы данных должен учитывать такую возможность.

С прозрачностью удаленного доступа связан и еще один важный вопрос: если удаленные таблицы кажутся локальными, может ли пользователь формулировать запросы, в которых локальные и удаленные таблицы используются *совместно*? Можно ли, например, соединять таблицы, хранящиеся на разных серверах, и связывать информацию из удаленной и локальной баз данных? Еще более серьезный вопрос возникает при анализе транзакций. Если СУБД допускает прозрачный доступ к удаленной базе данных, может ли пользователь обновить строку локальной базы данных *и* добавить строку в удаленную базу данных, а затем отменить всю транзакцию? Поскольку удаленные ресурсы представляются пользователю локальными, ответ *должен* быть: "Конечно, ведь локальные и удаленные таблицы должны представляться пользователю единой локальной базой данных".

На деле же поддержка таких распределенных запросов и транзакций добавляет к реализации удаленного доступа новый уровень сложности, а потенциально еще и невероятно увеличивает нагрузку на сеть. Поэтому, хотя некоторые коммерческие СУБД и поддерживают распределенные транзакции, на практике они используются редко. Эти возможности и связанные с ними издержки более подробно

описываются далее в настоящей главе. А в следующих разделах мы поговорим об альтернативной технологии, используемой гораздо чаще, — дублировании данных, или *репликации баз данных*.

Дублирование таблиц

Доступ к удаленным базам данных из локальных СУБД очень удобен при выполнении небольших запросов и нерегулярном доступе к данным. Если же приложению требуется интенсивный и частый доступ к удаленной базе данных, тогда коммуникационные издержки описанной ранее схемы работы могут оказаться неприемлемыми. Когда интенсивность и частота операций удаленного доступа превышает определенный предел, часто более эффективным оказывается использование локальной *копии* удаленных данных. Многие производители СУБД предоставляют специальные средства для упрощения процесса извлечения и распространения данных. В простейшем случае содержимое таблицы извлекается из главной базы данных, пересылается по сети другой системе и помещается в соответствующую таблицу-реплику в подчиненной базе данных (рис. 23.3). Эта процедура обычно выполняется периодически во время наименьшей загрузки системы.

Рис. 23.3. Репликация между главной и подчиненными базами данных

Такой подход очень удобен в тех случаях, когда данные в реплицируемых таблицах изменяются редко или изменения выполняются в пакетном режиме. Предположим, например, что несколько таблиц нашей учебной базы данных, расположенные в главной системе, должны быть реплицированы в локальные базы данных. Содержимое таблицы `OFFICES` практически никогда не меняется. Поэтому она будет прекрасным кандидатом для репликации в рабочие базы данных дистрибьюторских центров и торговых менеджеров. После того как локальные таблицы-реплики соответствующим образом созданы и заполнены данными из главной таблицы, их можно обновлять раз в месяц или не обновлять вовсе, пока не будет открыт какой-нибудь новый офис.

Еще одним хорошим кандидатом на репликацию является таблица PRODUCTS. Хотя цены товаров меняются чаще, чем информация об офисах, однако в большинстве компаний их изменения проводятся комплексно, раз в неделю или раз в день. В этот естественный цикл легко вписать и рассылку новых цен по дистрибьюторским центрам. Чтобы всегда содержать самые последние данные, таблицы с ценами в локальных базах данных дистрибьюторов вовсе не должны быть жестко связаны с главной таблицей. Ежедневного или еженедельного обновления их данных, синхронно с пакетным обновлением цен в главной таблице, вполне достаточно.

Описанную стратегию репликации можно реализовать вообще без поддержки со стороны СУБД. Вы можете написать приложение со встроенным SQL, которое будет работать на мэйнфрейме и извлекать данные о ценах на товары из базы данных, помещая их в файл. Еще одна программа может пересылать этот файл в дистрибьюторские центры, где третья программа будет считывать его содержимое и генерировать соответствующие инструкции DROP TABLE, CREATE TABLE и INSERT для заполнения таблиц-реплик.

Первым шагом в направлении автоматизации этой стратегии стала разработка высокоскоростных программ для извлечения и загрузки данных. В этих утилитах, предлагаемых многими производителями современных СУБД, обычно используются специализированные низкоуровневые технологии доступа к базам данных, поэтому выборка и загрузка данных в них выполняется гораздо быстрее, чем при использовании обычных инструкций SELECT и INSERT. Позднее стали появляться аналогичные программные продукты независимых производителей. Эта категория программного обеспечения, названная "извлечение, преобразование и загрузка" (extract, transform and load (ETL)), занимается связью различных систем баз данных и форматов файлов. Инструментарий ETL обычно предлагает графический пользовательский интерфейс для указания извлекаемых данных, набор инструментов для переформатирования данных, средства передачи данных по сети и утилиты для сохранения данных в целевой системе. В комплект входят также средства для управления процессом и его мониторинга.

Имеются две дополнительные категории программного обеспечения для интеграции — это интеграция приложений масштаба предприятия (enterprise application integration, EAI) и интеграция информации масштаба предприятия (enterprise information integration, EII). Эти категории в определенной мере перекрываются с категорией ETL, так что наилучшим способом отличать их является рассмотрение предназначения данных. Как уже упоминалось, целью ETL является база данных. С другой стороны, EAI обеспечивает получение данных из одного приложения и передачу их в другое, обычно с применением некоторой технологии передачи сообщений. Целью EAI является приложение. Программное обеспечение EII предназначено для доставки данных из разных источников пользователю, который и является целью EII. Некоторые инструменты EII позволяют представлять пользователю файлы разных форматов так, как если бы это были реляционные базы данных, так что конечный пользователь может работать с ними посредством SQL. Несмотря на наличие общего в ETL, EAI и EII, именно ETL обеспечивает возможность загрузки (или перегрузки) таблиц баз данных по расписанию.

Репликация таблиц

Некоторые производители СУБД пошли по пути включения функций репликации прямо в СУБД. Например, Oracle предлагает материализованные представления (которые до версии Oracle8i именовались снимками (snapshot)) для автоматического создания локальной копии удаленной таблицы. *Материализованное представление* сохраняет строки, определенные запросом из его определения. В простейшей форме локальная таблица представляет собой предназначенную только для чтения реплику удаленной главной таблицы, которая загружается при определении представления. Однако материализованные представления могут быть определены таким образом, чтобы СУБД Oracle автоматически периодически обновляла их. Вот инструкция Oracle, создающая локальную копию данных о ценах товаров (предполагается, что в главной базе данных имеется таблица PRODUCTS — такая же, как в нашей учебной базе данных).

Создать локальную реплику с информацией о ценах из удаленной таблицы PRODUCTS.

```
CREATE MATERIALIZED VIEW PRODPRICE
            AS SELECT MFR_ID, PRODUCT_ID, PRICE
                 FROM PRODUCTS@REMOTE_LINK;
```

Эта инструкция создает локальную таблицу с именем PRODPRICE. Таблица состоит из трех столбцов, заданных в инструкции SELECT, извлекающей данные из удаленной (главной) базы данных. Символ '@', за которым следует имя REMOTE_LINK, говорит Oracle, что таблица PRODUCTS, данные которой должны быть реплицированы в локальную таблицу, является удаленной и доступна через канал связи REMOTE_LINK. Этот канал заранее создается и настраивается администратором базы данных. Напоследок инструкция CREATE MATERIALIZED VIEW заполняет материализованное представление PRODPRICE данными из таблицы PRODUCTS и физически сохраняет его в локальной базе данных.

При использовании материализованных представлений этого типа пользователям не разрешается изменять содержащиеся в них данные с помощью инструкций INSERT, UPDATE или DELETE. Все изменения вносятся только в главную (удаленную) таблицу и автоматически переносятся в таблицы-реплики (материализованные представления) средствами самой СУБД Oracle. При желании администратор базы данных может обновить материализованное представление вручную при помощи хранимой процедуры DBMS_MVIEW.REFRESH, поставляемой Oracle. Инструкция CREATE MATERIALIZED VIEW включает специальное предложение REFRESH, предназначенное для автоматического обновления реплик. Вот пара примеров его использования.

Создать локальную реплику с информацией о ценах из удаленной таблицы PRODUCTS; обновлять данные раз в неделю путем полной перезагрузки.

```
CREATE MATERIALIZED VIEW PRODPRICE
   REFRESH COMPLETE START WITH SYSDATE NEXT SYSDATE+7
         AS SELECT MFR_ID, PRODUCT_ID, PRICE
              FROM PRODUCTS@REMOTE_LINK;
```

Создать локальную реплику с информацией о ценах из удаленной таблицы PRODUCTS; обновлять данные каждый день, пересылая из главной таблицы только изменения.

```
CREATE MATERIALIZED VIEW PRODPRICE
       REFRESH FAST START WITH SYSDATE NEXT SYSDATE+1
          AS SELECT MFR_ID, PRODUCT_ID, PRICE
               FROM PRODUCTS@REMOTE_LINK;
```

В последнем примере ключевое слово FAST указывает, что при обновлении материализованного представления из главной таблицы передаются только измененные данные. Для реализации этой возможности Oracle ведет в удаленной системе журнал изменений, который обновляется каждый раз, когда обновление таблицы PRODUCTS должно быть отражено в материализованном представлении. Информация из этого журнала используется, когда приходит время обновления материализованного представления.

Для приложений, подобных нашему, где изменения цен товаров затрагивают лишь небольшую часть таблицы PRODUCTS, эта стратегия довольно эффективна. Издержки, связанные с ведением журнала, весьма незначительны по сравнению с уменьшением сетевого трафика, обусловленным передачей по сети только измененных данных. В других приложениях, где в промежутках между обновлением материализованного представления изменяется гораздо большая часть строк главной таблицы, имеет смысл отказаться от ведения журнала изменений.

По умолчанию Oracle идентифицирует строки (чтобы определить, какие из них изменены) на основе первичного ключа. Если первичный ключ не является частью реплицированных данных, Oracle использует внутренние идентификаторы строк для представления этих строк в материализованном представлении.

Инструкция SELECT, определяющая материализованное представление, может иметь любую допустимую форму, в том числе содержать условие отбора.

Создать локальную реплику с информацией о ценах из удаленной таблицы PRODUCTS; обновлять данные каждый день, пересылая из главной таблицы только изменения.

```
CREATE MATERIALIZED VIEW PRODPRICE
           REFRESH FAST START WITH SYSDATE NEXT SYSDATE+1
        AS SELECT MFR_ID, PRODUCT_ID, PRICE
             FROM PRODUCTS@REMOTE_LINK
            WHERE PRICE > 1000.00;
```

Учтите, что наличие условия отбора не влияет на журнал изменений, поскольку СУБД Oracle должна фиксировать в нем все изменения таблицы PRODUCTS — ведь на основе этого журнала могут обновляться несколько материализованных представлений. Возможно также создание материализованного представления на основе соединения нескольких главных таблиц удаленной базы данных.

Создать локальную реплику с данными о служащих, обновляемую еженедельно.

```
CREATE MATERIALIZED VIEW SALESTEAM
           REFRESH FAST START WITH SYSDATE NEXT SYSDATE+7
        AS SELECT NAME, QUOTA, SALES, CITY
             FROM SALESREPS@REMOTE, OFFICES@REMOTE
            WHERE REP_OFFICE = OFFICE;
```

Можно еще усложнить запрос путем группирования.

Создать локальную реплику с итоговыми суммами заказов клиентов, обновляемую ежедневно.

```
CREATE MATERIALIZED VIEW CUSTORD
           REFRESH FAST START WITH SYSDATE NEXT SYSDATE+1
         AS SELECT COMPANY, SUM(AMOUNT)
              FROM CUSTOMERS@REMOTE, ORDERS@REMOTE
             WHERE CUST = CUST_NUM;
```

Конечно, с введением каждого нового уровня сложности увеличиваются и издержки, связанные с процессом репликации. Однако общие принципы остаются теми же. Вместо того чтобы пересылать по сети каждый запрос к удаленной базе данных, приложение, установленное в периферийной системе, работает с локальными материализованными представлениями нужных ему данных, и эти представления периодически обновляются, причем частота обновления отражает динамику изменения информации в главной базе данных. Таким образом, приложения могут длительное время вообще не работать с сетью, а стратегия обновления материализованных представлений вырабатывается таким образом, чтобы соотношение издержек на ведение журнала и передачу новых данных по сети было оптимальным. Для тех ситуаций, когда периферийное приложение настолько интенсивно работает с базой данных, что вызываемая им нагрузка на сеть и базу данных превышает издержки, связанные с ведением журналов и обновлением материализованных представлений, репликация является самым эффективным способом повышения производительности всего программного комплекса.

Двунаправленная репликация

В простейших реализациях (см. рис. 23.3) таблица связана с каждой своей репликой строгим соотношением "главная-подчиненная". Центральная (главная) таблица содержит реальные, актуальные, данные. Это всегда самая последняя информация, и она должна обновляться приложениями только в главной таблице. Копии периодически обновляются в пакетном режиме самой СУБД. В промежутках между обновлениями их информация может оказаться несколько устаревшей, но если база данных сконфигурирована таким образом, значит, это приемлемая плата за преимущество использования локальных копий данных. Приложениям не разрешается обновлять данные, содержащиеся в копиях реплицированной таблицы. В случае подобной попытки СУБД генерирует сообщение об ошибке.

По умолчанию инструкция Oracle `CREATE MATERIALIZED VIEW` создает этот тип подчиненной реплики таблицы. Более совершенные функции, такие как множественные обновимые реплики одной и той же главной таблицы, требуют использования Oracle Advanced Replication, выходящей за рамки данной книги.

В схеме репликации Microsoft SQL Server иерархическая связь реплик является неявной. SQL Server определяет главную таблицу как издателя данных, а подчиненные таблицы — как их подписчиков. В создаваемой по умолчанию конфигурации существует один обновляемый издатель и несколько подписчиков, данные которых доступны только для выборки. Развивая эту аналогию, SQL Server поддер-

живает два вида обновлений: *подписка* (когда издатель сам отправляет обновленные данные подписчикам) и *запрос* (когда вся ответственность за получение обновленных данных лежит на подписчиках).

Однако существуют такие типы приложений, для которых технология табличной репликации очень удобна, но иерархическое отношение между репликами к ним неприменимо. Например, в приложениях, от которых требуется очень высокая степень надежности, часто поддерживаются две идентичные копии данных в двух компьютерных системах. Если одна система выходит из строя, для продолжения работы используется вторая. Другим примером может быть интернет-приложение с большим количеством пользователей, выполняющее очень интенсивный обмен с базой данных. Для обеспечения приемлемой эффективности работы пользователей такого приложения его рабочая нагрузка может быть распределена между несколькими компьютерными системами с отдельными синхронизируемыми копиями данных. Или еще один пример. В торговой компании может существовать одна центральная таблица клиентов и сотни ее реплик в портативных компьютерах торговых менеджеров. При этом все менеджеры должны иметь возможность вводить в свои реплики информацию о новых клиентах или изменять данные о старых клиентах. Для всех этих (и многих других) типов приложений наиболее эффективной является схема, при которой *все* реплики допускают модификацию своего содержимого (рис. 23.4).

Рис. 23.4. Схема двунаправленной репликации

В связи с поддержкой множественных реплицированных таблиц, все копии которых должны принимать изменения, возникает ряд проблем, связанных с целостностью данных. Что произойдет, если одна и та же строка таблицы будет обновлена в нескольких репликах? Когда СУБД будет синхронизировать эти реплики, какой из двух вариантов строки она должна будет принять? Может быть, следует отклонить оба изменения или попробовать их объединить? А что если из одной реплики строка будет удалена, а в другой модифицирована?

В СУБД, поддерживающих двунаправленную репликацию, все эти вопросы решаются путем определения набора правил для разрешения конфликтов, и эти правила автоматически используются подсистемой репликации. Например, правило может говорить о том, что в случае конфликта между обновлениями в центральной базе данных и базе данных на портативном компьютере всегда побеждает обновление в центральной таблице. Или же правило может устанавливать приоритет самого последнего обновления. В дополнение ко встроенным правилам, определяемым самой СУБД, подсистема репликации может поддерживать возможность разрешения конфликтов с помощью пользовательских подпрограмм (например, хранимых процедур). Такая подпрограмма может решить, какая реплика победит, а какая проиграет, учитывая особенности конкретных операций над данными.

Затраты на репликацию

На практике любая стратегия репликации подразумевает определенный компромисс между актуальностью данных и производительностью системы. Причем учитываются, как правило, не только чисто технические аспекты, но и особенности конкретной ситуации, в которой применяется репликация. Для примера рассмотрим процедуру обработки заказов в учебной базе данных и предположим, что эта процедура выполняется на пяти вычислительных системах, расположенных в разных географических точках. При поступлении заказа производится обращение к таблице PRODUCTS для проверки того, имеется ли в наличии достаточное количество товара. В этой таблице хранятся данные о наличии товаров на складах компании во всем мире.

Предположим, что компания проводит такую политику: служащий, принимающий заказ, должен безусловно гарантировать клиенту, что товар будет доставлен ему в течение 24 часов с момента приема заказа. В этом случае таблица PRODUCTS должна содержать данные, действительные буквально на текущую минуту и отражающие размещение заказов, принятых всего лишь несколько секунд назад. Существует только два возможных способа, позволяющих обеспечить это условие. Первый способ — совместное использование одной центральной таблицы PRODUCTS всеми пользователями в пяти региональных центрах. Альтернативный способ — наличие зеркальной копии таблицы PRODUCTS в каждом из пяти центров. Второе решение вряд ли практично, так как частые обновления таблицы PRODUCTS, выполняемые для поддержания идеальной синхронизации при каждом приеме заказа, вызовут резкое возрастание сетевого трафика.

А теперь предположим, что компания решила сделать свою политику менее строгой, полагая, что она по-прежнему сможет обеспечить адекватный уровень обслуживания клиентов. Например, если заказ не может быть выполнен немедленно, компания обещает уведомить клиента об этом в течение 24 часов и дает ему возможность отменить заказ. В этом случае отличным решением будет реплицированная таблица PRODUCTS. Ночью все изменения, сделанные в таблице PRODUCTS, могут вноситься в реплицированные копии во всех пяти региональных центрах. При приеме заказов в течение дня наличие товара проверяется по *локальной копии* таблицы PRODUCTS, и только эта локальная копия обновляется. Это предотвращает прием заказов, для которых в начале рабочего дня отсутствовало

достаточное количество товара, но не предотвращает прием в разных местах заказов, для которых суммарное количество заказанного товара превысит наличные запасы. Следующей ночью, когда услуги связи дешевле, чем днем, заказы из каждого регионального центра передаются в главный офис, где происходит их обработка и обновление центральной таблицы PRODUCTS. Заказы, которые не могут быть удовлетворены из наличных запасов, отмечаются, и создается соответствующий отчет. Когда обработка завершается, обновленная таблица PRODUCTS вместе с "отчетом о неудовлетворенных заказах" передается обратно во все пять региональных центров для использования в течение следующего рабочего дня.

Какая же из описанных двух схем является более корректной? Как показывает этот пример, вопрос не столько в архитектуре базы данных, сколько в особенностях ведения бизнеса. На самом деле и то и другое тесно взаимосвязано. Вот почему выбор той или иной схемы репликации неизбежно приводит к упрощению одних операций и усложнению других.

Типичные схемы репликации

Во многих случаях можно выбрать такую схему репликации, при которой конфликты между обновленными репликами будут устранены или, по крайней мере, сведены к минимуму. Если конфликт все же возник, истиной в последней инстанции будут правила разрешения конфликтов, применяемые в СУБД. В следующих разделах рассматривается несколько типичных сценариев репликации таблиц и принципы разрешения возникающих в них конфликтов.

Горизонтальные подмножества таблиц

Один из эффективных способов распространения таблицы по сети заключается в следующем: разделить таблицу горизонтально, размещая полученные блоки записей в разные системы. На рис. 23.5 изображен простой пример горизонтального разделения таблицы. Здесь компания имеет три дистрибьюторских центра, в каждом из которых предлагаются свои группы товаров, выполняется собственная обработка заказов и имеется своя база данных товарных запасов.

Для реализации этой схемы центральная таблица PRODUCTS разделяется горизонтально на три части, и в нее добавляется столбец LOCATION, указывающий местоположение товара. Строки таблицы, описывающие товары конкретного дистрибьюторского центра, передаются в региональные базы данных, управляемые центральной СУБД.

Большинство обновлений таблицы PRODUCTS выполняется в дистрибьюторских центрах в процессе обработки заказов. Поскольку реплики таблицы PRODUCTS в каждом центре независимы друг от друга (одна строка таблицы может входить только в одну реплику), конфликты обновлений невозможны. Все изменения реплик могут периодически пересылаться в центральную базу данных, чтобы в ней была представлена актуальная информация.

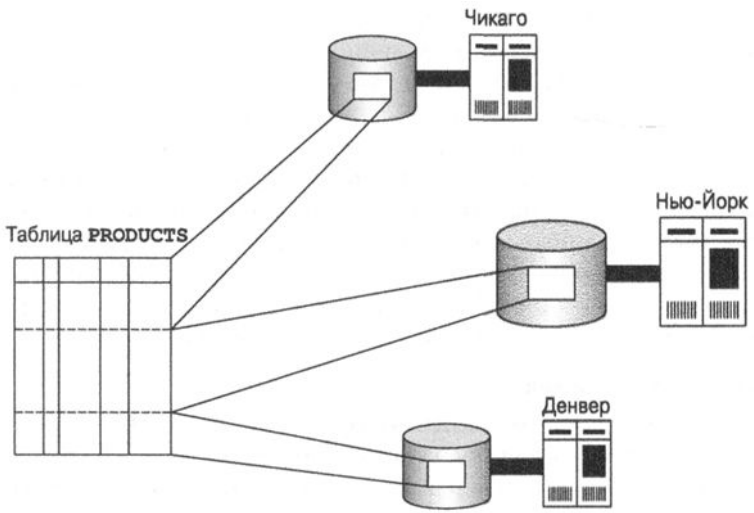


Рис. 23.5. Горизонтальная репликация

Вертикальные подмножества таблиц

Еще один способ распределения таблицы по сети состоит в вертикальном разделении таблицы и размещении ее столбцов в разных системах. На рис. 23.6 приведен простой пример вертикального разделения таблицы. Таблица SALESREPS была расширена новыми столбцами с информацией о служащих (номер телефона, семейное положение и т.д.) и теперь используется как отделом обработки заказов, так и отделом кадров, каждый из которых имеет свою собственную базу данных. Работа каждого отдела сфокусирована, по большей части, на одном или двух столбцах таблицы, однако имеется немало запросов и отчетов, в которых используются столбцы, относящиеся как к отделу кадров, так и к отделу обработки заказов.

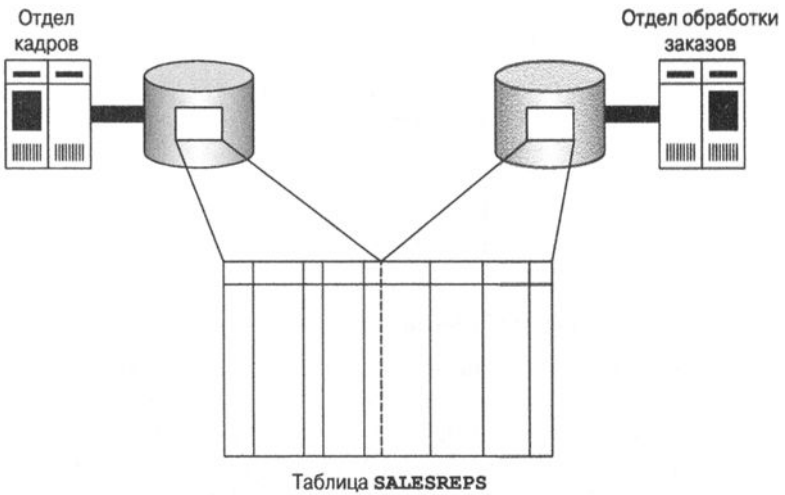


Рис. 23.6. Вертикальная репликация

Для реализации этой схемы таблица SALESREPS целиком реплицируется в обе системы, но логически считается разделенной вертикально на две части. Столбцы таблицы, содержащие личные данные (NAME, AGE, HIRE_DATE, PHONE, MARRIED), принадлежат базе данных отдела кадров. Любые конфликты, связанные с обновлением этих столбцов, решаются в пользу отдела кадров. Остальные столбцы (EMPL_NUM, QUOTA, SALES, REP_OFFICE) принадлежат базе данных отдела обработки заказов. В его пользу решаются конфликты в отношении этих столбцов. Поскольку в обеих базах данных имеется полная копия таблицы, можно создавать по ней отчеты и обращаться к ней с запросами, причем вся обработка будет производиться локально. Только при обновлениях задействуется механизм репликации, возникает сетевой трафик и возможно возникновение конфликтов.

Зеркальная репликация таблиц

Когда репликация необходима для обеспечения высокой отказоустойчивости системы (чтобы система могла продолжать свою работу в случае аппаратных сбоев или сбоев в базе данных), создается зеркальная копия всей таблицы (рис. 23.7). В простейшей реализации одна база данных становится активной, а другая — резервной. Все запросы направляются к активной базе данных (система А), которая посылает реплики всех изменений резервной базе данных (система Б). В случае сбоя запросы перенаправляются в резервную систему, которая содержит свежие данные. Недостатком такого подхода является то, что при нормальной работе без сбоев ресурсы резервной системы тратятся впустую. Эту систему необходимо обслуживать и оплачивать, хотя она не участвует в обработке данных.

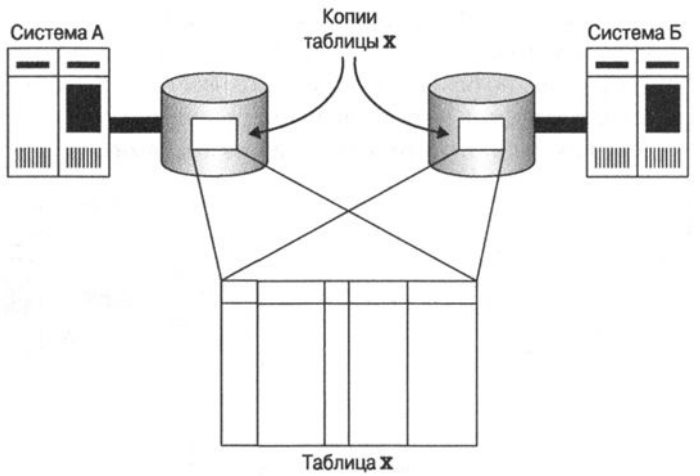


Рис. 23.7. Зеркальная репликация

Из-за указанного недостатка, системы с высокой отказоустойчивостью обычно проектируют так, чтобы сбалансировать возникающую в них нагрузку (рис. 23.8). В такой конфигурации некоторая промежуточная программа перехватывает запросы к СУБД и распределяет их между двумя или более системами. При нормальной работе ни одна из систем не простаивает. Более того, легко можно повысить производительность обработки запросов, добавив другие системы с копиями реплицированной таблицы.

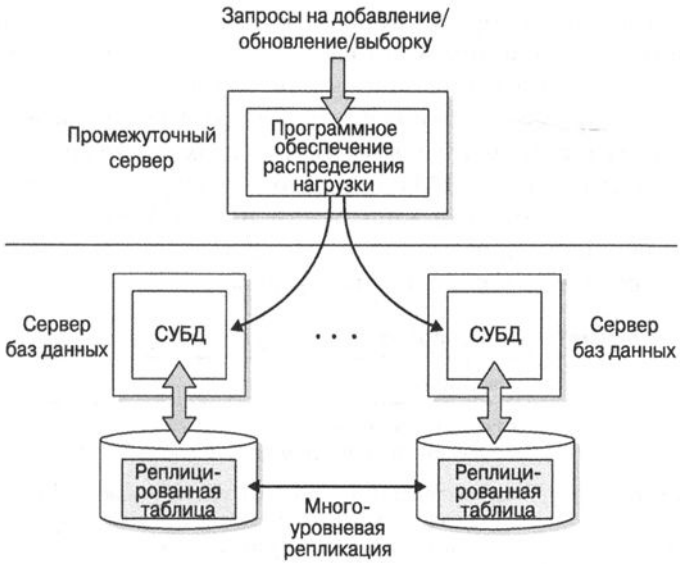


Рис. 23.8. Репликация с балансировкой нагрузки

Этот подход оказывается весьма эффективным, если соотношение запросов на выборку к запросам на изменение очень высоко (например, 95% составляют запросы на выборку и 5% — запросы на добавление, удаление или обновление). Если же процент запросов на изменение выше определенного предела, то издержки, связанные с репликацией и разрешением конфликтов, могут свести на нет эффективность всей системы в целом.

Повысить эффективность в такой ситуации можно, если распределять запросы на изменение, руководствуясь определенными правилами. Например, если реплицируется таблица клиентов, то первичным ключом в ней может быть имя клиента. Тогда программу распределения нагрузки можно написать так, что запросы на изменение данных о клиентах, чьи имена начинаются с букв от *А* до *М*, будут направляться одной системе, а запросы на изменение данных о клиентах, чьи имена начинаются с букв от *Н* до *Я*, — другой. Это уменьшает вероятность возникновения конфликтов. Поскольку в данной схеме таблица по-прежнему целиком реплицируется во все системы, запросы на выборку можно произвольным образом распределять между системами для выравнивания нагрузки.

Доступ к распределенным базам данных

В течение нескольких последних лет теоретические изыскания в области доступа к распределенным базам данных медленно, но уверенно пробивают дорогу в коммерческие продукты. Сегодня многие базы данных уровня предприятия предлагают как минимум некоторый уровень прозрачности доступа к распределенным базам данных. Как указывалось ранее, затраты на обеспечение доступа к распределенным базам данных могут быть достаточно существенны. Два очень схожих запроса могут требовать совершенно разного сетевого трафика и различ-

ных накладных расходов. Скорость выполнения одного и того же запроса методом грубой силы и оптимизированным методом также может быть различной в зависимости от качества оптимизации в конкретной СУБД.

Из-за указанных трудностей все поставщики СУБД внедряют в свои продукты поддержку доступа к распределенным базам данных постепенно, шаг за шагом. Несколько лет назад компания IBM объявила о своих планах постепенного обеспечения доступа к распределенным данным в своих СУБД. Четыре стадии, предложенные компанией IBM и приведенные в табл. 23.1, являются прекрасной основой для понимания принципов управления распределенными данными.

Таблица 23.1. Четырехстадийный подход к доступу к распределенным базам данных

Стадия	Описание
Удаленный запрос	Каждая инструкция SQL обращается к одной удаленной базе данных; каждая инструкция представляет собой транзакцию
Удаленная транзакция	Каждая инструкция SQL обращается к одной удаленной базе данных; транзакция, состоящая из нескольких инструкций, также может обращаться только к одной базе данных
Распределенная транзакция	Каждая инструкция транзакции обращается к одной удаленной базе данных, но транзакция в целом может обращаться к нескольким базам данных
Распределенный запрос	Инструкция SQL может обращаться к нескольким удаленным базам данных; транзакция, состоящая из нескольких инструкций, также может обращаться к нескольким базам данных

Схема компании IBM представляет простую модель формулировки задачи доступа к распределенным данным: пользователю, работающему с одной вычислительной системой, требуется получить доступ к данным, хранящимся в других вычислительных системах (одной или нескольких). Сложность распределенного доступа увеличивается на каждой последующей стадии. Таким образом, возможности СУБД можно охарактеризовать достигнутой стадией. Кроме того, на каждой стадии можно различать доступ только для выборки данных (посредством инструкции `SELECT`) и доступ для изменения данных (посредством инструкций `INSERT`, `DELETE` и `UPDATE`). Часто в СУБД для некоторого уровня сначала вводится возможность только выборки данных и лишь впоследствии появляется поддержка как выборки, так и изменения данных.

Удаленные запросы

Первым уровнем доступа к распределенным данным, по определению IBM, является *удаленный запрос*, схема выполнения которого изображена на рис. 23.9. На этом уровне пользователь может выполнить инструкцию SQL, которая извлекает или изменяет информацию в одной удаленной базе данных. Каждая отдельная инструкция SQL является транзакцией, что аналогично режиму автозавершения, часто используемому при работе с базой данных в интерактивном режиме. Пользователь может выполнять последовательно несколько инструкций SQL, обращающихся к разным базам данных, но СУБД не поддерживает транзакции, состоящие из нескольких инструкций.

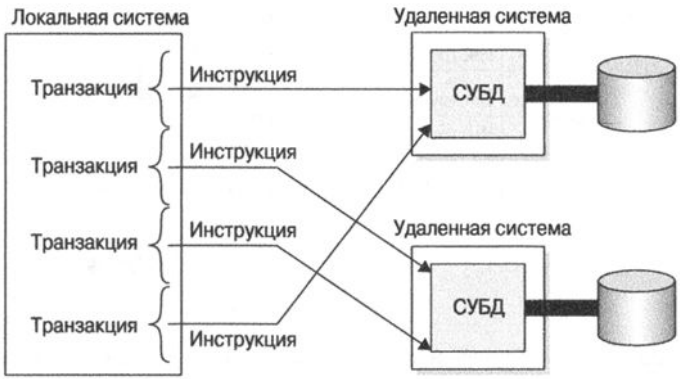


Рис. 23.9. Доступ к распределенным данным: удаленные запросы

Удаленные запросы полезны в тех случаях, когда пользователю необходимо получить информацию из корпоративной базы данных. Как правило, все запрашиваемые сведения находятся в одной базе данных, например в базе заказов или базе производственных данных. Выполняя удаленный запрос, программа, работающая на персональном компьютере, извлекает удаленные данные для последующей их обработки на своем локальном компьютере — в электронных таблицах, графических программах или настольных издательских системах.

Однако возможностей удаленного запроса недостаточно для большинства приложений, работающих в режиме транзакций. Рассмотрим приложение, обеспечивающее ввод заказов, которое находится на персональном компьютере и работает с корпоративной базой данных. Чтобы обработать новый заказ, приложение должно проверить наличие нужного количества товара на складе, добавить заказ в базу данных, уменьшить количество товара на складе и откорректировать сумму заказов для клиента вместе с объемом продаж. Для этого потребуется с полдесятка различных инструкций SQL. Как объяснялось в главе 11, "Целостность данных", если не выполнить эти инструкции в виде одной транзакции, то согласованность базы данных может быть нарушена. Однако на уровне удаленных запросов транзакции, состоящие из нескольких инструкций, выполняться не могут, и, следовательно, данное приложение работать не будет.

Удаленные транзакции

Вторым уровнем доступа к распределенным данным, по определению компании IBM, является *удаленная транзакция* ("удаленная единица работы", выражаясь языком IBM), схематически изображенная на рис. 23.10. На этом уровне поддерживаются транзакции, состоящие из нескольких инструкций SQL. Пользователь может передать в СУБД последовательность инструкций SQL, осуществляющих выборку или обновление информации в удаленной базе данных, а затем выполнить или отменить всю последовательность как одну транзакцию. СУБД гарантирует, что вся транзакция будет выполнена как единое целое — либо успешно, либо нет, как это происходит в локальной базе данных. Все инструкции SQL, составляющие транзакцию, должны быть адресованы одной удаленной базе данных.

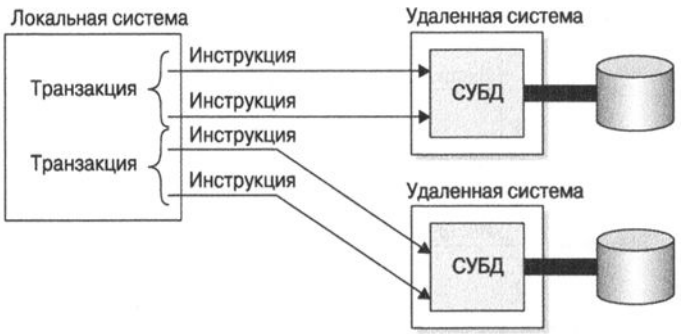


Рис. 23.10. Доступ к распределенным данным: удаленные транзакции

Удаленные транзакции открывают дорогу для приложений, работающих с распределенными данными в режиме транзакций. Например, на данном уровне доступа программа ввода заказов, рассмотренная в качестве примера в предыдущем разделе, при обработке нового заказа может выполнить последовательность запросов на выборку и обновление информации из базы данных складских запасов, а также на добавление информации. Программа завершает эту последовательность команд, образующих транзакцию, инструкцией `COMMIT` или `ROLLBACK`.

Как правило, для выполнения удаленных транзакций необходимо наличие СУБД (или, по крайней мере, программы, способной выполнять транзакции) на персональном компьютере, а также в той системе, где находится база данных. Логика выполнения транзакций в СУБД должна распространяться на всю сеть, чтобы ответ на вопрос, была выполнена транзакция или нет, всегда был однозначным. Однако реальную ответственность за сохранение целостности базы данных несет удаленная СУБД.

Удаленные транзакции часто поддерживаются специальными шлюзовыми программами, связывающими СУБД разных типов. В частности, большинство поставщиков корпоративных СУБД (Sybase, Oracle, Informix) предоставляют шлюзы к DB2 для своих СУБД на базе Unix или Linux. Некоторые шлюзовые программы не только обеспечивают выполнение удаленных транзакций, но и дают пользователю возможность объединять в одном запросе таблицы из локальной базы данных с таблицами из удаленной базы данных, управляемой СУБД другого типа. Однако эти программы не обеспечивают (а без поддержки удаленной СУБД и не могут обеспечить) выполнение транзакций, необходимых для реализации более высоких уровней доступа к распределенным данным. Шлюзовая программа может поддерживать по отдельности целостность локальной и удаленной баз данных, но не может исключить ситуацию, когда в одной базе данных транзакция будет завершена, а в другой — отменена.

Распределенные транзакции

Третьим уровнем доступа к распределенным данным, по определению IBM, является *распределенная транзакция* ("распределенная единица работы", выражаясь языком IBM), схематически изображенная на рис. 23.11. На этом уровне каждая отдельная инструкция SQL по-прежнему обращается с запросом к одной базе данных в одной удаленной системе. Однако транзакция, представляющая собой по-

следовательность инструкций SQL, может обращаться к двум или более базам данных, расположенным в различных системах. Когда транзакция выполняется (или отменяется), СУБД гарантирует, что все части транзакции во всех участвующих системах будут завершены (отменены), т.е. СУБД гарантирует, что не будет частичной транзакции, когда транзакция завершается в одной системе и отменяется в другой.

Рис. 23.11. Доступ к распределенным данным: распределенные транзакции

Уровню распределенных транзакций соответствуют только очень сложные приложения, работающие в режиме транзакций. Например, в корпоративной сети, представленной на рис. 23.1, программа обработки заказов может с персонального компьютера запросить информацию из базы складских запасов, расположенной на двух или трех серверах дистрибьюторских центров, чтобы проверить наличие требуемых товаров, а затем обновить базы данных и включить товары в заказ клиента. СУБД следит за тем, чтобы другие параллельно обрабатываемые заказы не мешали сеансу удаленного доступа первой транзакции.

Реализовать распределенные транзакции гораздо труднее, чем запросы, соответствующие первым двум уровням доступа к распределенным данным. Невозможно обеспечить выполнение распределенной транзакции без активного взаимодействия отдельных СУБД, участвующих в транзакции. По этой причине поддержка распределенных транзакций, как правило, реализуется только в однородной среде, которую составляют СУБД одного производителя (например, все СУБД только Oracle или только Sybase). Это взаимодействие обычно осуществляется при помощи специального протокола *двухфазного завершения транзакции*, более подробно рассматриваемого далее в настоящей главе.

Распределенные запросы

Последним уровнем доступа к распределенным данным в модели IBM является распределенный запрос, схема выполнения которого изображена на рис. 23.12. На этом уровне одна инструкция SQL может обращаться к таблицам из двух или более баз данных, расположенных в различных системах. СУБД отвечает за автоматическое выполнение инструкции по сети. Последовательность инструкций, представляющих собой распределенный запрос, может быть объединена в транзакцию. Как и на предыдущем уровне, СУБД должна гарантировать целостность распределенной транзакции во всех участвующих системах.

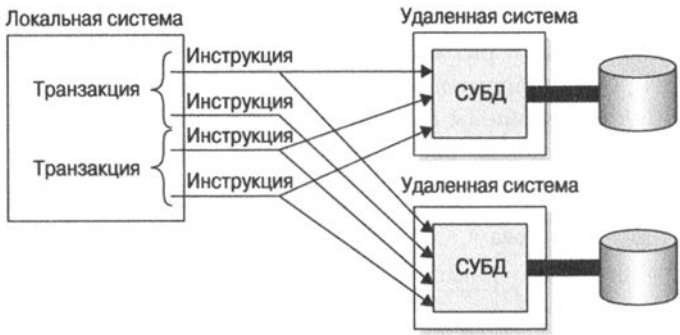


Рис. 23.12. Доступ к распределенным данным: распределенные запросы

На этом уровне от программной логики СУБД, отвечающей за выполнение транзакций, не требуется ничего нового, так как СУБД должна поддерживать транзакции, выходящие за рамки одной системы, уже на предыдущем уровне распределенных транзакций. Однако на уровне распределенных запросов предъявляются значительные требования к подпрограммам оптимизации инструкций. Теперь оптимизирующий модуль при оценке альтернативных способов выполнения инструкции SQL должен учитывать скорость передачи данных в сети. Если локальной СУБД приходится многократно обращаться к удаленной таблице (например, при создании объединения), то, возможно, быстрее будет скопировать часть таблицы по сети посредством одной операции пакетной передачи данных, чем многократно извлекать по сети отдельные строки.

Оптимизирующий модуль должен также решить, какая СУБД будет управлять выполнением инструкции. Если большая часть таблиц находится в удаленной системе, то, возможно, было бы лучше, чтобы инструкцию выполняла удаленная СУБД этой системы. Однако если удаленная система сильно загружена, то такое решение может оказаться неудачным. Таким образом, задача оптимизирующего модуля становится и более сложной, и более важной.

В конечном счете, на уровне распределенных запросов главная цель — сделать так, чтобы вся распределенная база данных выглядела для пользователя как одна большая база данных. В идеальном случае пользователь имел бы полный доступ к любой таблице распределенной базы данных и мог бы выполнять транзакции SQL, ничего не зная о физическом местоположении данных. К сожалению, этот идеальный сценарий в реальных сетях оказывается нежизнеспособным. Число таблиц распределенной базы данных в сети любого размера быстро стало бы очень большим, и пользователи обнаружили бы, что невозможно найти интересующие их данные. К тому же пришлось бы координировать идентификаторы пользователей во всех базах данных на предприятии, чтобы каждый идентификатор пользователя был уникальным во всех базах данных. Администратору базы данных также было бы очень трудно выполнять свою работу.

Поэтому на практике распределенные запросы следует внедрять избирательно. Администраторы баз данных должны решать, какие удаленные таблицы будут доступны для локальных пользователей, а какие останутся скрытыми от них. Совместно работающие СУБД должны перемещать идентификаторы и пароли поль-

зователей из одной системы в другую, причем каждая база данных должна иметь возможность автономно обеспечивать защиту при удаленном доступе к данным. Распределенные запросы, которые потребляли бы слишком много сетевых ресурсов или ресурсов СУБД, должны обнаруживаться и предотвращаться прежде, чем они повлияют на общую производительность СУБД.

В настоящее время распределенные запросы из-за своей сложности не поддерживаются полностью ни в одной коммерческой СУБД, и пройдет еще некоторое время, прежде чем станет доступной большая часть возможностей, присущих рассматриваемому уровню доступа к распределенным данным. Важным шагом на пути к этому явилась стандартизация протокола распределенных транзакций. Протокол XA, изначально разработанный для координации взаимодействия программ, осуществляющих мониторинг транзакций, теперь активно применяется для управления распределенными транзакциями. Java-версия (Java Transaction Protocol, JTP) обеспечивает интерфейс распределенных транзакций для приложений и серверов приложений на основе Java. В настоящее время большинство коммерческих СУБД разработано для применения в сетевом окружении, поддерживающем интерфейсы XA и JTA.

Протокол двухфазного завершения транзакций*

Если распределенная СУБД поддерживает распределенные транзакции, то она должна соблюдать правило "все или ничего", обязательное для SQL-транзакций. Пользователь распределенной СУБД ожидает, что завершенная транзакция будет завершена во всех системах, где располагаются данные, и отмененная транзакция также будет отменена во всех системах. Кроме того, сбои в сети или в одной из систем должны приводить к тому, что СУБД прекратит выполнение транзакции и отменит ее, а не оставит транзакцию в частично завершенном состоянии.

Все коммерческие СУБД, которые поддерживают или планируют поддерживать распределенные транзакции, используют для этого специальный метод, называемый *двухфазным завершением*. Чтобы применять распределенные транзакции, вам не требуется понимать работу этого метода. Суть его в том и состоит, чтобы поддерживать распределенные транзакции без участия пользователя. Однако знание механизма двухфазного завершения помогает более эффективно планировать работу с базой данных.

Чтобы понять, почему необходим специальный механизм двухфазного завершения, рассмотрим базу данных, представленную на рис. 23.13. Пользователь, находящийся в системе А, обновил одну таблицу в системе Б и одну таблицу в системе В и хотел бы теперь завершить транзакцию. Предположим, что СУБД в системе А пытается завершить транзакцию, просто посылая инструкцию `COMMIT` в систему Б и систему В, а затем ожидая подтверждающих ответов. Такой метод работает до тех пор, пока обе системы Б и В успешно выполняют свои части транзакции. Но что произойдет, если, например, неисправность диска или тупиковая ситуация не позволят системе В выполнить запрашиваемые действия? Система Б завершит свою часть транзакции и пошлет в систему А подтверждение, система В, вследствие ошибки, отменит свою часть транзакции и пошлет в систему А сообщение об

ошибке, а пользователь останется с частично завершенной и частично отмененной транзакцией. Обратите внимание на то, что теперь система А не может "передумать" и дать системе Б команду отменить транзакцию. Транзакция в системе Б завершилась, и другие пользователи, опираясь на ее результаты, возможно, уже модифицировали какие-то данные.

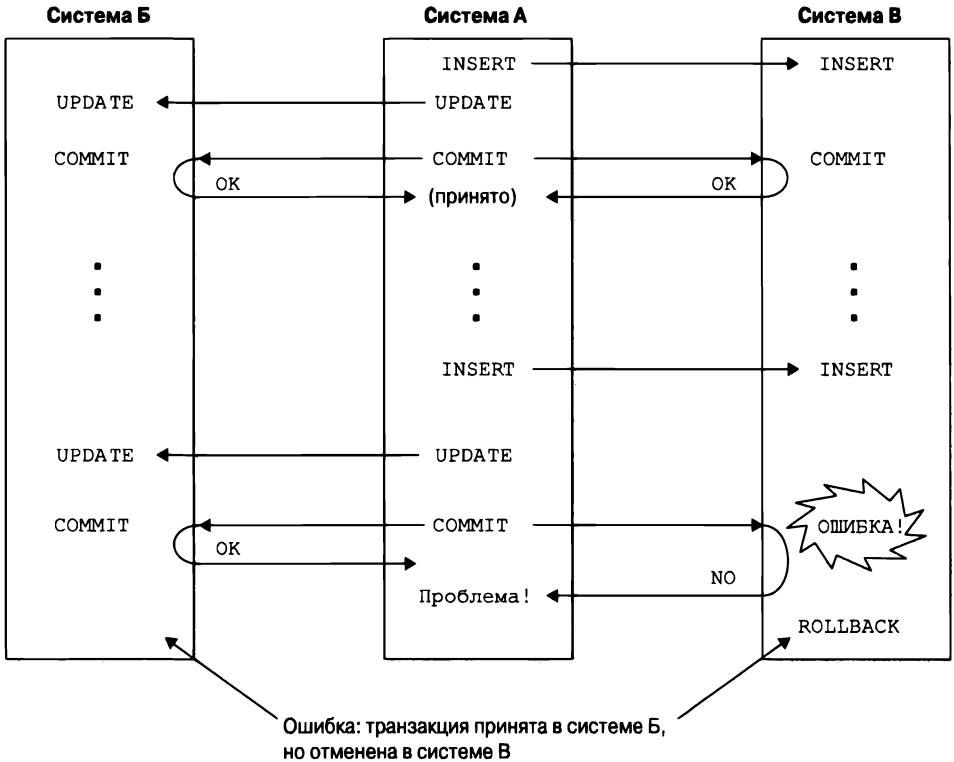


Рис. 23.13. Проблемы при широковещательной схеме принятия транзакций

Метод двухфазного завершения позволяет избежать проблем, возникающих при использовании обычного механизма, проиллюстрированного на рис. 23.13. На рис. 23.14 изображена схема работы метода двухфазного завершения.

1. Программа в системе А выдает инструкцию `COMMIT`, чтобы завершить текущую (распределенную) транзакцию, которая обновила таблицы в системах Б и В. Система А действует как *координатор* процесса завершения, согласовывая свои действия с системами Б и В.
2. Система А посылает сообщение `GET READY` в системы Б и В и делает отметку об этом в своем журнале транзакций.
3. Когда СУБД в системе Б или В получает сообщение `GET READY`, она должна приготовиться *либо* к завершению, *либо* к отмене текущей транзакции. Если СУБД может перейти в состояние "готова завершить", она отвечает `YES` системе А и отмечает этот факт в своем локальном журнале транзакций; если она не может перейти в это состояние, то отвечает `NO`.

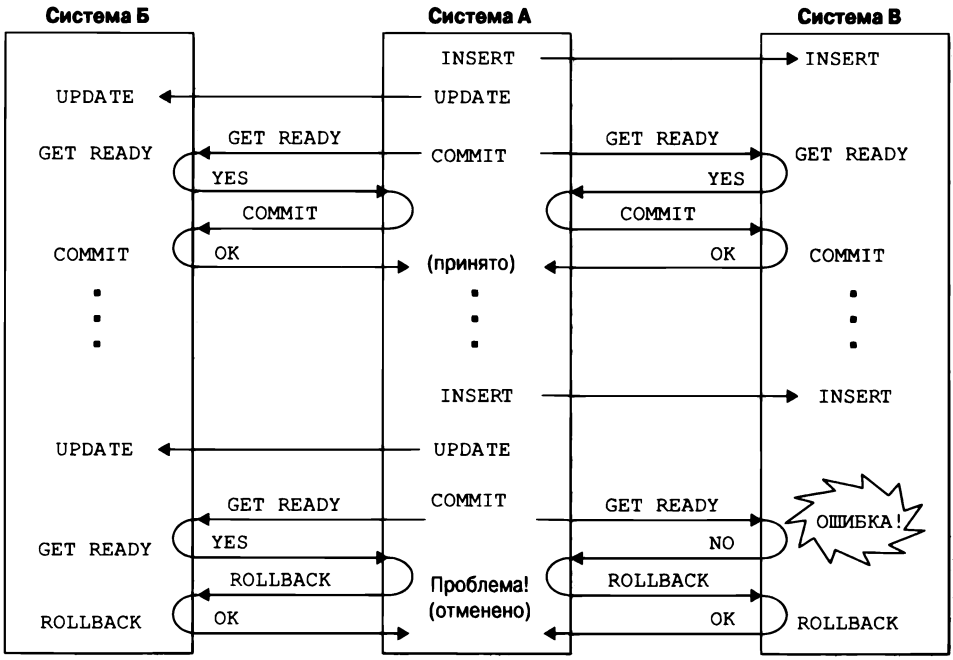


Рис. 23.14. Протокол двухфазного завершения транзакций

4. Система А ожидает ответов на свое сообщение GET READY. Если все ответы будут YES, система А посылает инструкцию COMMIT в системы Б и В, делая отметку об этом в своем журнале транзакций. Если хотя бы один из ответов будет NO или если все ответы не будут получены в течение определенного промежутка времени, то система А посылает в обе другие системы инструкцию ROLLBACK и делает отметку об этом в своем журнале транзакций.
5. Когда СУБД в системе Б или В получает инструкцию COMMIT или ROLLBACK, она *обязана* выполнить полученную команду. Если СУБД ответила YES на сообщение GET READY, полученное на шаге 3, она потеряла возможность самостоятельно решать судьбу транзакции. СУБД завершает или отменяет свою часть транзакции в зависимости от полученной команды, записывает сообщение COMMIT или ROLLBACK в свой журнал транзакций и возвращает сообщение OK в систему А.
6. Когда система А получит все сообщения OK, она будет знать, что транзакция либо завершена, либо отменена, и возвратит в программу соответствующее значение переменной SQLCODE.

Данный метод защищает распределенную транзакцию от любой одиночной ошибки в системе Б, системе В или сети. Два приведенных ниже примера иллюстрируют, как с помощью этого метода осуществляется восстановление после сбоя.

- Предположим, что в системе В происходит ошибка до отправки сообщения YES на шаге 3. Система А не получит сообщение YES и пошлет инструкцию ROLLBACK, вынуждая систему Б отменить транзакцию. Про-

грамма восстановления в системе В не найдет сообщение YES или COMMIT в локальном журнале транзакций и в процессе восстановления отменит транзакцию в системе В. В итоге все части транзакции будут отменены.

- Предположим, что в системе В происходит ошибка после отправки сообщения YES на шаге 3. Система А решит, завершать или отменять транзакцию, в зависимости от ответа системы Б. Программа восстановления в системе В найдет в локальном журнале транзакций сообщение YES, но не найдет сообщение COMMIT или ROLLBACK, обозначающее конец транзакции. Программа восстановления спросит у координатора (системы А), как закончилась транзакция, и выполнит соответствующие действия. Обратите внимание: система А должна сохранять запись о своем решении завершить или отменить транзакцию до тех пор, пока не получит окончательное подтверждение OK от всех участников; это необходимо для того, чтобы в случае ошибки она могла передать эту информацию в программу восстановления.

Метод двухфазного завершения гарантирует целостность распределенных транзакций, но при его реализации значительно возрастает сетевой трафик. Если транзакция охватывает n систем, то для успешного завершения транзакции координатор должен послать и получить $4n$ сообщений. Причем эти сообщения будут отправлены в дополнение к тем сообщениям, посредством которых осуществляется передача инструкций SQL и результатов запросов между системами. К сожалению, если к распределенной транзакции предъявляется требование обеспечения целостности базы данных в случае системных ошибок, то избежать подобного потока сообщений невозможно.

В связи с тем что распределенные транзакции вызывают избыточный сетевой трафик, они могут оказать серьезное отрицательное влияние на производительность базы данных. По этой причине необходимо проектировать распределенные базы данных таким образом, чтобы данные, к которым часто обращаются (или, по крайней мере, которые часто обновляются), находились в локальной системе или в одной удаленной системе. Транзакции, осуществляющие обновления в двух или более удаленных системах, должны по возможности выполняться редко.

Сетевые приложения и архитектура баз данных

Нововведения в области компьютерных сетей в последние десятилетия тесно связаны с новшествами в архитектурах реляционных баз данных и SQL. Мощные мини-компьютеры (например, семейства VAX компании Digital), соединенные по сети с мэйнфреймами, были первой популярной платформой реляционных СУБД. Они обеспечивали процесс принятия решений на основе данных, загруженных с мэйнфреймов. Кроме того, они поддерживали локальные OLTP-приложения, служащие для ввода деловой информации и загрузки ее в базы данных корпоративных приложений, выполнявшихся на мэйнфреймах.

Появление производительных серверов на базе UNIX (например, производства компании Sun) вызвало новую волну расширения и усовершенствования различ-

ных СУБД, которая привела к появлению архитектуры "клиент/сервер", доминировавшей в сфере обработки корпоративных данных в 1990-х годах. Позже рост корпоративных сетей и приложений (таких, как Enterprise Resource Planning) потребовал нового уровня масштабирования и распределенности баз данных. Сегодня взрывной рост популярности Интернета и продуктов с открытым кодом является движущей силой другой волны инноваций, требующей от баз данных невиданной ранее интенсивности транзакций, из-за чего в последнее время активно развиваются технологии кеширования баз данных и размещения их в оперативной памяти.

Приложения "клиент/сервер" и архитектура баз данных

Когда базы данных на основе SQL впервые появились в мини-компьютерных системах, архитектура баз данных и их приложений была очень простой — вся работа, от вывода на экран до вычислений, обработки данных и обращения к базе данных, выполнялась на центральном процессоре мини-компьютера. С появлением мощных персональных компьютеров и серверных платформ эта архитектура по ряду причин претерпела серьезнейшие изменения.

Графический пользовательский интерфейс популярных офисных программ (электронных таблиц, текстовых процессоров и т.п.) привел к появлению нового стандарта простоты использования приложений, и компании захотели, чтобы он был распространен также на корпоративные приложения, работающие с базами данных. Наличие графического интерфейса требует интенсивного использования процессора и высокой пропускной способности канала между процессором и видеопамятью, содержащей экранное изображение. Хотя некоторые протоколы и предусматривают работу графического интерфейса по сети (протокол X-windows), лучшим местом для размещения программного кода, реализующего графический интерфейс приложения, является персональный компьютер.

Отнюдь не последним фактором, определяющим выбор архитектуры, являются экономические соображения. В пересчете на вычислительную мощность персональные компьютерные системы гораздо дешевле мини-компьютеров и серверов UNIX. Если перенести большую часть обработки данных бизнес-приложений на относительно недорогие ПК, общая стоимость программно-аппаратного обеспечения предприятия может быть значительно снижена. Это важный аргумент в пользу переноса на персональные компьютеры программного кода не только уровня визуального представления, но и уровня бизнес-логики.

Под влиянием этих и других факторов и получила столь широкое распространение архитектура "клиент/сервер" (рис. 23.15). Многие современные приложения разработаны именно для нее. Ключевую роль во взаимодействии между клиентской и серверной подсистемами играет SQL. Запросы от прикладной логики (на ПК) отправляются СУБД (на сервере) в виде SQL-инструкций. Ответы СУБД отправляются обратно по сети в виде кодов завершения или таблиц результатов запроса.

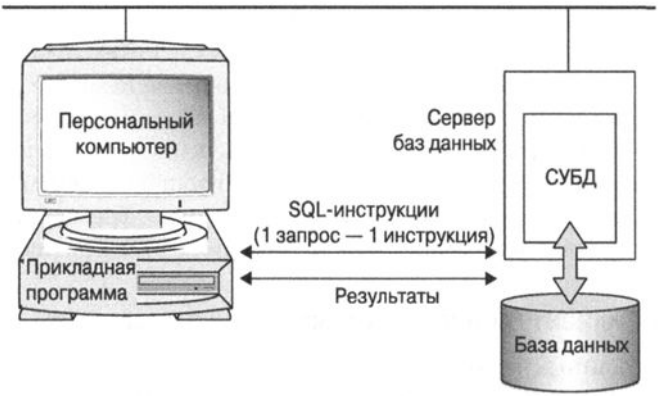


Рис. 23.15. Архитектура "клиент/сервер"

Приложения "клиент/сервер" с хранимыми процедурами

Когда приложение распределено между двумя или более компьютерными системами, как на рис. 23.15, одним из главных вопросов его функционирования является взаимодействие между частями этого приложения. Каждая операция, относящаяся к удаленной системе, вызывает сетевой трафик, а сетевые соединения, как известно, — это всегда самая медленная часть всей системы как в отношении пропускной способности, так и в отношении времени ожидания ответа. В архитектуре, изображенной на рис. 23.15, каждое обращение к базе данных (т.е. каждая инструкция SQL) требует передачи по сети как минимум одной пары сообщений (запрос-ответ).

В ходе одной транзакции типичного OLTP-приложения обычно выполняется по меньшей мере десяток инструкций SQL. Например, для принятия заказа на один товар в нашей учебной базе данных приложение должно выполнить следующее:

- получить идентификатор клиента по его имени (одиночная инструкция `SELECT`);
- получить лимит кредита этого клиента для проверки его платежеспособности (одиночная инструкция `SELECT`);
- получить информацию о товаре, такую как цена и количество на складе (одиночная инструкция `SELECT`);
- добавить в таблицу `ORDERS` строку нового заказа (инструкция `INSERT`);
- обновить информацию о товаре, чтобы отразить уменьшение его количества на складе (инструкция `UPDATE`);
- обновить информацию о клиенте, чтобы отразить уменьшение лимита его кредита (инструкция `UPDATE`);
- завершить всю транзакцию (инструкция `COMMIT`).

В результате получается семь пар сообщений, пересылаемых между приложением и базой данных. В реальном приложении количество обращений к базе данных за одну транзакцию может быть в два или три раза большим. А с увеличением

объема транзакций пропорционально растет и сетевой трафик, достигая очень высокой интенсивности.

Использование хранимых процедур позволяет несколько модифицировать эту архитектуру, значительно снизив генерируемый приложением сетевой трафик. Схема работы такой системы изображена на рис. 23.16. Хранимая процедура хранится прямо в базе данных и инкапсулирует последовательность действий и логику принятия решений, необходимую для выполнения всех операций, составляющих одну транзакцию.

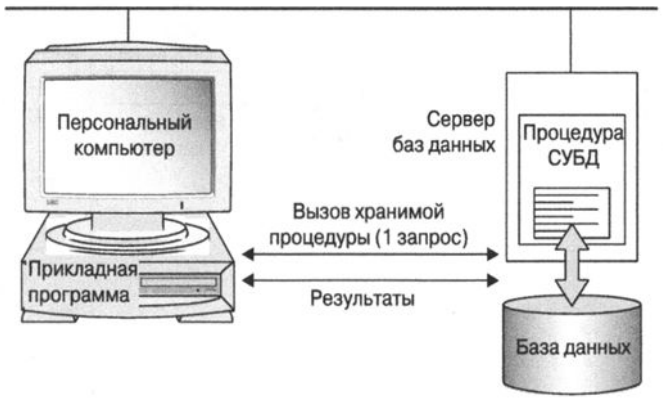


Рис. 23.16. Архитектура "клиент/сервер" с хранимыми процедурами

Таким образом, часть деловой логики приложения переносится на сервер базы данных и больше не требует сетевого обмена информацией. Вместо пересылки в СУБД отдельных инструкций SQL приложение просто вызывает хранимую процедуру, передавая ей в нашем случае имя клиента, название товара и требуемое количество. Если все проходит нормально, хранимая процедура возвращает код успешного завершения. Если же возникают проблемы (например, на складе нет нужного товара или кредит клиента исчерпан), тогда приложение получает код ошибки и соответствующее сообщение. В результате весь сетевой трафик транзакции сводится к одной паре сообщений.

У хранимых процедур есть и ряд других преимуществ, но сокращение сетевого трафика — главное из них. Именно это явилось главной причиной продвижения на рынке СУБД SQL Server и помогло укрепить ее позиции как специализированной СУБД для высокопроизводительных OLTP-приложений. Хранимые процедуры настолько популярны, что сейчас их поддерживают все ведущие СУБД масштаба предприятия.

Корпоративные приложения и кеширование данных

Сегодня практически все коммерческие приложения, предназначенные для автоматизации деятельности предприятия, используют SQL и реляционные базы данных. В качестве примеров можно назвать программное обеспечение для планового отдела, отдела кадров, бухгалтерии таких фирм, как SAP, Infor Global Solutions (бывш. BAAN), Oracle (скупившая PeopleSoft и Siebel Systems), Sage

Group, Microsoft, IBM, i2 Technologies и др. Эти крупномасштабные приложения обычно работают на мощных серверах UNIX и создают очень высокую рабочую нагрузку на СУБД. Для изоляции работы приложений и СУБД и переноса основной нагрузки по обработке данных на приложения используется трехуровневая архитектура, схематически представленная на рис. 23.17.

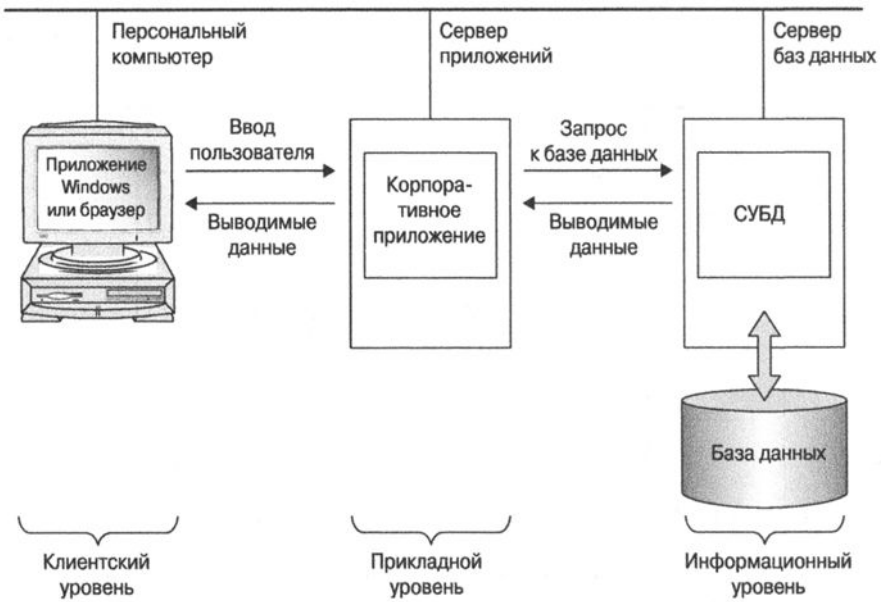


Рис. 23.17. Типичная трехуровневая архитектура крупных корпоративных приложений

Даже при использовании хранимых процедур сетевая нагрузка и нагрузка на СУБД, создаваемая некоторыми подобными приложениями, может превысить возможности полосы пропускания сети и максимальную интенсивность транзакций, поддерживаемую СУБД. В качестве примера можно рассмотреть приложение для планирования поставок, помогающее предприятию-производителю планировать заказы материалов, сырья и комплектующих. Для выполнения своей задачи такое приложение должно проанализировать *каждый* имеющийся заказ на продукцию и подготовить перечень компонентов, необходимых для производства каждого продукта. Сложные продукты могут состоять из тысяч компонентов, а эти компоненты в свою очередь также могут собираться из десятков и сотен отдельных элементов.

Если составлять план поставок с использованием обычных технологий программирования, приложение должно запросить из базы данных информацию о составе каждого продукта (и каждой его комплектующей), входящего в каждый заказ клиента, и затем сформировать итоговые данные о количестве материалов и комплектующих, необходимых для удовлетворения этих заказов. Если предприятие получает сотни и тысячи заказов, для их обработки могут потребоваться часы. Хуже того, приложение может просто не укладываться в то время, которое выделено ему для выполнения задания (обычно это ночные часы, когда рабочая нагрузка на базу данных минимальна).

Чтобы добиться приемлемой производительности, во всех крупномасштабных приложениях, выполняющих интенсивную обработку данных, используются специальные технологии кеширования, когда данные с помощью специального промежуточного программного обеспечения извлекаются из серверной базы данных и перемещаются "ближе" к приложению. В большинстве случаев технологии кеширования относительно примитивны. Например, перечень материалов может быть прочитан однажды и помещен в таблицу, находящуюся в оперативной памяти компьютера, на котором работает приложение. Избавившись от однотипных, многократно повторяющихся запросов о составе продукции, программа может резко повысить свою производительность.

С недавнего времени разработчики приложений стали использовать более сложные технологии кеширования. Например, приложение может реплицировать наиболее интенсивно используемые данные в таблицы локальной базы данных. Еще более мощной альтернативой является загрузка частей базы данных в оперативную память — она применяется в тех случаях, когда обрабатываются данные относительно небольшого объема (десятки или сотни мегабайтов). С появлением 64-разрядных операционных систем и неуклонным снижением цен на оперативную память практикуется кеширование все больших объемов данных (десятков или сотен гигабайтов).

С ростом требований к деловым приложениям сложные технологии кеширования и репликации приобретают все большее значение. Ведущие компании-производители уже ориентируются на планирование в реальном времени, когда поступающие заказы клиентов обрабатываются мгновенно и тут же отражаются в планах поставок материалов. Реализация этих требований неизбежно означает рост интенсивности и сложности взаимодействия приложений с базами данных.

Управление базами данных в Интернете

Интернет-приложения также могут вести интенсивный обмен информацией с базами данных, и из-за низкой пропускной способности каналов связи, по сравнению с локальными сетями, они еще больше нуждаются в применении высокоэффективных технологий кеширования и репликации. Например, финансовые компании борются за привлечение клиентов для интерактивных биржевых сделок, предлагая все более эффективные возможности в плане анализа активности рынка ценных бумаг. Системы управления данными, необходимые для поддержки таких приложений, требуют сбора данных в реальном масштабе времени (нужно гарантировать достоверность информации о ценах, спросе и предложении) и пиковой нагрузки на базы данных в десятки тысяч транзакций в секунду. Подобные требования предъявляются и к приложениям, осуществляющим управление веб-сайтами и их мониторинг с интенсивным доступом. Тенденция к персонализации веб-сайтов, когда программное обеспечение на лету определяет, какие рекламные баннеры следует вывести на запрошенной пользователем странице, какие предложить товары и тому подобное, является еще одной причиной повышения нагрузки на базы данных.

В настоящее время уже имеется эффективная технология решения подобных проблем — кеширование веб-сайтов. Копии наиболее популярных веб-страниц распределяются по сети и реплицируются. В результате сеть может обслужить большее количество запросов веб-страниц, а объем сетевого трафика, связанного

с предоставлением самых популярных страниц, уменьшается. Подобная архитектура завоевывает популярность и для обеспечения интенсивного доступа к базам данных через Интернет (рис. 23.18). В этом случае интернет-сервер кеширует часто используемые данные, такие как последние новости или финансовая информация, в резидентной сверхвысокопроизводительной базе данных типа Oracle TimesTen или MySQL Cluster Server. Кроме того, сервер хранит в оперативной памяти информацию о пользователях и обращается к ней, чтобы настроить веб-сайт в соответствии с требованиями пользователей.

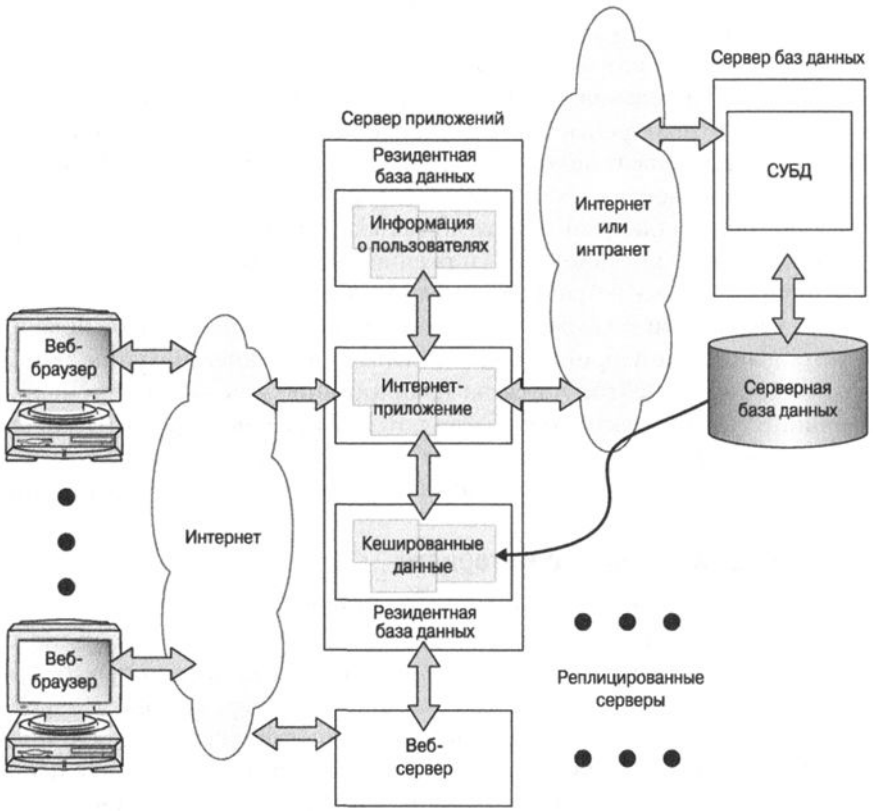


Рис. 23.18. Высокопроизводительное управление данными

Как следует из рис. 23.18, для управления данными в высокопроизводительных программных комплексах используются те же методы, что и для управления веб-сайтами, обслуживающими очень большое количество посетителей. Однако наличие такого звена, как база данных, и необходимость поддерживать ее целостность делают всю систему гораздо более сложной. И все же базовые технологии остаются теми же — репликация, резидентные базы данных и отказоустойчивые архитектуры. С ростом интернет-трафика и усилением персонализации требования к системе только растут, что приводит к более совершенным архитектурам сетевых баз данных.

Резюме

В этой главе были рассмотрены средства управления распределенными данными, имеющиеся в различных СУБД, а также описаны компромиссы, к которым приходится прибегать при осуществлении доступа к распределенным данным.

- Распределенная база данных находится, как правило, на нескольких вычислительных системах, объединенных с помощью сети. В каждой системе имеется собственная копия СУБД, автономно обеспечивающая доступ к локальным данным. Чтобы предоставить пользователю удаленный доступ к данным, эти копии СУБД при необходимости работают совместно.
- Идеальной распределенной базой данных является та, в которой пользователь не замечает, что данные распределены; все они предоставляются ему так, как если бы они находились в локальной системе.
- Поскольку такую идеальную распределенную базу данных создать очень трудно, элементы распределенных баз данных вводятся в коммерческие СУБД поэтапно.
- Непосредственный доступ к удаленным базам данных уместен в тех случаях, когда он составляет лишь малую часть работы приложения. Тогда издержки, связанные с передачей запросов и данных по сети, не столь значительны по сравнению с возможностью оставить данные там, где они хранятся, и не подключать дополнительные механизмы для их репликации.
- Репликация баз данных является прекрасным решением для приложений, интенсивно работающих с данными, расположенными в разных местах. Эта технология позволяет перенести данные ближе к пользователю, но за счет дополнительных затрат на синхронизацию и неполной актуальности данных.
- Издержки, связанные с доступом к удаленным данным и репликацией, — это вопрос не только технологический. Выбирая стратегию доступа к базам данных, следует учитывать реальные требования бизнеса.
- Современные тенденции к широкому внедрению распределенных приложений масштаба предприятия, интернет-приложений, хранилищ данных и других крупномасштабных приложений и технологий ведут к неуклонному усложнению методов распределенной обработки данных. Для достижения приемлемой производительности их многоуровневые архитектуры требуют использования интеллектуальных стратегий кеширования и репликации.

24
ГЛАВА

SQL и объекты

Единственная серьезная угроза господству языка SQL и реляционных баз данных в последние годы исходит от не менее важной тенденции — растущей популярности объектно-ориентированных технологий. Объектно-ориентированные языки программирования (такие, как C++ и Java), объектно-ориентированные средства создания приложений и сетевого программирования (например, брокеры объектных запросов или более поздние веб-службы) стали базовыми технологиями разработки современного программного обеспечения. Свою первоначальную популярность объектные технологии завоевали тем, что позволяли создавать приложения для персональных компьютеров с графическим пользовательским интерфейсом. Однако их влияние неуклонно расширяется, и сегодня на их основе разрабатываются и, что еще более важно, связываются между собой масштабные сетевые приложения для крупных корпораций.

В начале 90-х годов образовалась группа компаний, взявшаяся внедрять объектно-ориентированные принципы в системы управления базами данных. Эти компании верили, что их объектно-ориентированные базы данных (ООБД) столь же уверенно вытеснят устаревшие реляционные базы данных, как те в свое время вытеснили более ранние модели. Однако их прогнозы не оправдались. Реляционные технологии и SQL пока лишь укрепляют свое влияние. Более того, в ответ на вызов новых конкурентов производители реляционных СУБД перешили к наступательной тактике, активно встраивая объектные технологии в свои СУБД и создавая нечто вроде гибридных объектно-реляционных моделей. В этой главе рассказывается о том, что все эти нововведения приносят в SQL и какие объектно-ориентированные расширения предлагаются ведущими производителями современных СУБД.

Объектно-ориентированные базы данных

В последние несколько лет большинство научных исследований в области реляционных баз данных было сконцентрировано на новых, постреляционных моделях

представления данных. Как уже говорилось, реляционная модель имела явные преимущества перед более ранними, иерархической и сетевой, моделями. Цель исследований заключалась в разработке новых моделей представления данных, свободных от некоторых недостатков реляционной модели. Большинство исследований по новым моделям данных было сфокусировано на объединении принципов объектно-ориентированного программирования с традиционными характеристиками баз данных, такими как постоянные дисковые хранилища и управление транзакциями.

В дополнение к академическим исследованиям, в первой половине 1990-х годов были сделаны немалые капиталовложения в группу компаний, целью которых было построение технологий управления данными нового поколения. Обычно эти компании начинали с объектных структур данных, используемых в объектно-ориентированном программировании для работы с данными в оперативной памяти, и распространяли их на информацию на диске и на многопользовательский доступ. Ранние коммерческие продукты включают Gemstone (Servio Logic, позже переименована в Gemstone Systems), Gbase (Graphael) и Vbase (Ontologic). В средине 1990-х годов появились ITASCA (Itasca Systems), Jasmine (Fujitsu), Objectivity/DB (Objectivity, Inc.), ObjectStore (Progress Software, приобретена eXcelon, ранее Object Design), Matisse (Matisse Software), O_2 (O_2 Technology, приобретена Informix, в свою очередь приобретена IBM), ONTOS (Ontos, Inc., бывш. Ontologic), POET (Poet Software, ныне FastObjects фирмы Versant), Versant Object Database (Versant Corporation) и VOSS (Logic Arts). Энтузиасты этих объектно-ориентированных баз данных твердо верили в победу над реляционными базами данных и доминирование на рынке к концу десятилетия. Хотя эти надежды и не оправдались, производители объектно-ориентированных баз данных серьезно повлияли на своих реляционных конкурентов.

Характеристики объектно-ориентированной базы данных

В отличие от реляционной модели данных, для которой в 1970 году доктор Кодд в своей статье дал четкое математическое определение, в отношении объектно-ориентированной базы данных такой ясности нет. Когда употребляют этот термин, обычно имеют в виду базу данных, организованную на основе принципов, характерных для объектно-ориентированного программирования.

- **Объекты**. В ООБД любая сущность является объектом и обрабатывается как объект. Табличная, в виде строк и столбцов, организация реляционной базы данных заменяется ее организацией в виде *коллекции объектов*. В общем случае коллекция объектов сама является объектом и обрабатывается так же, как и другие объекты.
- **Классы**. Характерное для реляционных баз данных понятие атомарного типа данных заменяется в ООБД иерархическими понятиями *класс* и *подкласс*. Например, `VEHICLES` (транспортные средства) может быть классом объектов, а отдельным представителем (экземпляром) этого класса может быть автомобиль, велосипед, поезд или лодка. Класс `VEHICLES` может включать в себя подклассы, например, `CARS` (автомобили) и `BOATS` (лодки), являющие собой конкретные виды транспортных средств. Аналогично

класс CARS может включать подкласс CONVERTIBLES (автомобили с откидным верхом) и т.д.

- **Наследование**. Объекты *наследуют* характеристики своего класса и всех классов более высокого уровня иерархии, к которой они принадлежат. Например, одной из характеристик транспортного средства может быть "количество пассажиров". Все члены классов CARS, BOATS и CONVERTIBLE также имеют этот атрибут, так как они являются подклассами класса VEHICLES. Класс CARS может также иметь атрибут "количество дверей", и класс CONVERTIBLES унаследует этот атрибут. Однако класс BOATS не будет его наследовать.
- **Атрибуты**. Характеристики объекта моделируются его атрибутами. Примерами атрибутов объекта, представляющего транспортное средство, могут быть цвет, количество дверей и название модели. Связь атрибутов с определяемыми ими объектами можно сравнить со связью столбцов и строк таблицы.
- **Сообщения и методы**. Объекты взаимодействуют друг с другом посредством *сообщений*. Когда объект получает сообщение, он отвечает на него, выполняя *метод* — подпрограмму, хранимую внутри объекта, которая определяет способ обработки сообщения. Таким образом, поведение объекта описывается набором его методов. Обычно объект имеет много общих методов с другими объектами своего класса.
- **Инкапсуляция**. Внутренняя структура и данные объектов скрыты от внешнего мира за набором строго определенных интерфейсов. Получать информацию об объекте и взаимодействовать с ним можно только посредством его интерфейсных методов, функции и поведение которых четко специфицированы. Это делает объект более предсказуемым и ограничивает возможности случайного разрушения хранящихся в нем данных.
- **Идентификаторы объектов**. Объекты различают с помощью уникальных идентификаторов, обычно реализованных в виде абстрактных указателей, называемых *дескрипторами*. Дескрипторы часто используются для представления связей между объектами: объект указывает на другой объект с помощью дескриптора, который он хранит в одном из своих атрибутов.

Благодаря этим принципам объектно-ориентированные базы данных хорошо подходят для приложений со сложными типами данных, например для программ компьютерного моделирования или программ разработки составных документов, объединяющих текст, графику и электронные таблицы. Такие базы данных естественным образом отражают иерархию разнородных данных. Например, весь документ может быть представлен как один объект, состоящий из меньших объектов (глав), которые, в свою очередь, состоят из еще более мелких объектов (абзацев, рисунков и т.д.). Иерархия классов дает возможность отслеживать тип каждого объекта в документе (абзацы, диаграммы, иллюстрации, заголовки, колонтитулы и т.д.).

Наконец, механизм сообщений естественным образом обеспечивает поддержку графического интерфейса пользователя. Приложение может послать сообщение "нарисуй себя" любой части документа, давая таким образом команду объекту

отобразить себя на экране. Если пользователь изменяет размер окна, отображающего документ, приложение может отреагировать на это, посылая сообщение "измени свой размер" каждой части документа, и т.д. Каждый объект документа сам отвечает за свое отображение на экране, поэтому в документ можно легко добавлять новые объекты.

"Плюсы" и "минусы" объектно-ориентированных баз данных

Появление на рынке объектно-ориентированных баз данных вызвало волнение и споры в среде специалистов по базам данных. Главным аргументом сторонников ООБД было то, что новая технология позволяет добиться соответствия между программной моделью данных и структурой базы данных. Они утверждали, что в области обработки данных жесткая, фиксированная, структура реляционных таблиц, состоящих из строк и столбцов, является пережитком эры перфокарт с их фиксированными полями данных и ориентацией на записи. Для адекватного моделирования сущностей реального мира требуется другая, более гибкая, модель, построенная на основе классов объектов, которые могут быть сходны между собой (т.е. иметь общие атрибуты) и в то же время отличаться друг от друга.

Еще одним серьезным недостатком реляционной модели сторонники объектно-ориентированных баз данных считают то, что многотабличные соединения, являющиеся ее неотъемлемой частью, настолько снижают производительность баз данных, что делают реляционную технологию неприемлемой для разработки современных приложений, нуждающихся во все более высокопроизводительных платформах. Наконец, поскольку объекты прочно утвердились в современных программах в качестве структур для хранения данных в оперативной памяти, сторонники ООБД утверждают, что нет ничего естественнее и удобнее, чем распространить эту модель и на постоянное хранение данных. Это позволит самым прозрачным и оптимальным образом переносить данные из памяти на диск, где они будут в том же виде доступны всем пользователям и программам.

Оппоненты объектно-ориентированного подхода непреклонны в своем убеждении, что в переходе на объектную архитектуру баз данных нет никакой необходимости и этот переход не принесет никаких существенных преимуществ. Они утверждают, что дескрипторы ООБД есть не что иное, как встроенные указатели, использовавшиеся в дореляционных (иерархических и сетевых) базах данных, только по-другому названные. Кроме того, указывается, что, как и эти ранние технологии построения баз данных, объектно-ориентированный подход не имеет той мощной математической основы, на которой построена реляционная модель. Следствием этого недостатка является отсутствие стандартов для построения объектно-ориентированных баз данных и стандартизированного языка запросов, подобного SQL, что препятствует разработке инструментальных средств и приложений, независимых от СУБД.

В ответ на претензии к производительности реляционных баз данных их защитники указывают, что на их основе созданы некоторые из наиболее высокопроизводительных приложений масштаба предприятия. Кроме того, они проводят четкое разграничение между реляционной моделью данных и ее реализацией, в которой для повышения производительности вполне могут использоваться

встроенные указатели. И наконец, они утверждают, что все проблемы несоответствия между объектно-ориентированным программированием и реляционными базами данных могут быть разрешены с помощью таких технологий, как JDBC и другие объектно-реляционные интерфейсы.

Влияние объектных технологий на рынок баз данных

Настоящие объектно-ориентированные базы данных все же завоевали некоторую часть рынка — в приложениях с очень сложными моделями данных, а также там, где модели классов и наследования очень близки к их прототипам из реального мира. Но несмотря на это компании-производители таких СУБД испытывают большие трудности с продвижением своих продуктов на рынок и еще очень далеки от коммерческого успеха. Для большинства из них годовой доход в 100 миллионов долларов пока еще недосягаем, а многие вообще нерентабельны и по нескольку раз сменяли руководство. В то же время производители реляционных баз данных продолжают уверенно идти в гору. У крупнейших из них годовой доход достигает сотен миллионов и даже миллиардов долларов. Так что реляционные технологии со всей очевидностью продолжают доминировать на рынке баз данных.

Не удивительно, что два лагеря производителей баз данных оказывают друг на друга сильнейшее влияние. Поскольку рынок медленно и неохотно принимает объектные технологии, производители ООБД исследуют факторы, которым реляционные базы данных два десятилетия тому назад были обязаны своим успехом. Были сформированы группы стандартизации объектных технологий, например Object Data Management Group (ODMG). Некоторые производители снабдили свои СУБД реляционными драйверами со стандартными интерфейсами, такими как ODBC и SQL, которые обеспечивают доступ к ООБД из реляционных баз данных. Другие производители рассчитывают на поддержку со стороны международных стандартов и работают над включением объектно-ориентированных возможностей в стандарт SQL. Таким образом, наметилась тенденция сосуществования объектных и реляционных технологий.

Но нельзя отрицать и то, что объектные технологии также оказали существенное влияние на разработчиков реляционных СУБД. Некоторые элементы баз данных, начавшие свое существование как реляционные (например, хранимые процедуры), сейчас рекламируются как предоставляющие объектно-ориентированные возможности (например, инкапсуляцию). Производители постепенно добавляют в свои реляционные базы данных некоторые объектно-ориентированные элементы, например абстрактные типы данных. В результате получаются объектно-реляционные гибриды, т.е. СУБД, обладающие и теми, и другими возможностями. Расширяясь до невероятных пределов, реляционная модель включает в себя такие немыслимые ранее структуры, как таблицы в таблицах, моделирующие отношения между классами объектов.

Один из ведущих производителей реляционных СУБД, компания Informix Software (приобретенная IBM) купила комплекс объектно-ориентированных технологий, приобретя компанию Illustra Software. Объектно-реляционная СУБД Illustra была основана на проекте Postgres, созданном в Калифорнийском университете в Беркли и наследовавшем первую реляционную СУБД, созданную в этом

университете, — Ingres. Получившийся в результате продукт называется Informix Universal Server. Еще один гигант индустрии баз данных, Oracle Corporation, развивает собственную СУБД, вводя в нее объектно-ориентированные технологии. Итогом нескольких лет интенсивных усилий компании в этом направлении стала СУБД Oracle8, появившаяся в 1997 году, а последующие версии только усилили объектную ориентированность этой СУБД.

Объектно-ориентированные базы данных существенно повлияли и на сам стандарт SQL. Наиболее важным изменением в стандарте SQL3 (формально известном как SQL:1999) было добавление объектных возможностей. Новые объектно-ориентированные возможности практически удвоили количество страниц спецификации SQL. Приобретение и разработка объектно-реляционных баз данных ведущими производителями, а также формальное принятие объектных расширений SQL сигнализируют о растущей взаимосвязи SQL с миром объектных технологий.

Объектно-реляционные базы данных

Практически все объектно-реляционные базы данных (ОРБД) вначале были просто реляционными, а впоследствии вобрали в себя ряд объектно-ориентированных функций. Для ведущих производителей СУБД, чьи продукты масштаба предприятия развивались в течение минимум полутора десятков лет, этот путь оказался самым удобным и простым. Не стоит и говорить о том, что это гораздо дешевле, чем начинать с нуля, а кроме того, такая стратегия позволяет использовать огромную базу инсталлированных реляционных систем, предоставив их пользователям возможность плавного перехода на новые технологии, что выгодно как пользователям, так и производителям.

Вот каковы основные объектные расширения, уже имеющиеся в современных ОРБД.

- **Большие объекты данных**. Традиционные типы данных реляционных СУБД невелики по размеру — это целые числа, даты, короткие символьные строки. Большие объекты могут хранить документы, аудио- и видеоклипы, веб-страницы и другие мультимедийные данные.
- **Структурированные/абстрактные типы данных**. Реляционные типы данных атомарны и неделимы. Структурированные типы данных позволяют объединять отдельные элементы данных в высокоуровневые структуры, интерпретируемые как самостоятельные сущности.
- **Пользовательские типы данных**. Реляционные СУБД обычно содержат довольно ограниченный набор встроенных типов данных, тогда как одним из главных преимуществ объектно-ориентированных систем считается предоставляемая пользователю возможность создавать собственные типы данных.
- **Таблицы в таблицах**. Столбцы реляционных баз данных содержат отдельные значения, тогда как в ОРБД столбцы могут содержать такие

сложные элементы данных, как структуры или даже целые таблицы. Благодаря этому таблицы могут представлять иерархии объектов.

- **Последовательности, множества и массивы**. В традиционных реляционных базах данных наборы однотипных данных представляются в виде строк отдельной таблицы, связанных с таблицей-владельцем посредством внешнего ключа. В ОРБД допускается непосредственное хранение коллекций данных (совокупностей однотипных элементов — семейств, множеств, массивов) в одном столбце.
- **Хранимые процедуры**. Для записи, модификации и выборки данных традиционные реляционные СУБД поддерживают специальные интерфейсы, ориентированные на обработку наборов записей. Объектно-реляционные СУБД поддерживают процедурные интерфейсы, в частности хранимые процедуры, обеспечивающие возможность инкапсуляции данных и процедур их обработки внутри базы данных и строгое определение набора методов для взаимодействия с нею извне.
- **Дескрипторы и идентификаторы объектов**. Чисто реляционная СУБД требует, чтобы в качестве однозначных идентификаторов записей выступали сами данные (первичные ключи). Объектно-реляционные базы данных обеспечивают встроенную поддержку идентификаторов записей или других уникальных идентификаторов объектов.

Поддержка больших объектов

Реляционные СУБД традиционно ориентировались на обработку деловых данных. Единицами хранимой и обрабатываемой информации в них были элементы данных, представляющие денежные суммы, имена, адреса, дату/время и т.п. Эти типы данных относительно просты, и для их хранения требуется не много места, от нескольких байтов для целых чисел, представляющих, например, количество заказанных единиц товара или количество товара на складе, до нескольких десятков байтов для таких символьных данных, как имя клиента, адрес служащего или описание товара. Реляционные СУБД оптимизированы для управления записями, содержащими до нескольких десятков столбцов подобных типов. Технологии, используемые ими для управления дисковой памятью и индексации данных, разработаны с расчетом на то, что записи имеют размеры от нескольких сотен до нескольких тысяч байтов. Программы, которые сохраняют и извлекают подобные данные, легко могут держать в памяти десятки и сотни таких записей, пользуясь буферами разумного размера. Технологии построчной обработки результатов реляционных запросов прекрасно справляются со своей задачей.

Однако многие современные типы данных обладают совершенно иными характеристиками. Для хранения графического изображения, имеющего высокое разрешение, могут потребоваться сотни и тысячи байтов. Документы текстовых процессоров, такие как контракты или текст этой книги, могут занимать еще больше места. В качестве примеров можно также привести HTML-текст, определяющий содержимое веб-страницы, и файл PostScript с параметрами изображения, подготовленного для вывода на печать. Даже относительно короткая звукозапись высо-

кого качества может занимать миллионы байтов, а для хранения видеоклипов могут требоваться сотни мегабайтов и даже гигабайты памяти. При этом значение мультимедийных приложений все больше возрастает; пользователи хотят хранить мультимедийную информацию в базах данных и управлять ею наравне с остальными данными. Поэтому именно возможность эффективного управления большими объектами (large objects — LOB) была одним из первых рекламируемых преимуществ объектно-ориентированных баз данных.

Большие объекты в реляционной модели

Вначале для поддержки больших объектов реляционные базы данных просто пользовались возможностями операционной системы и ее файловой подсистемы. Каждый большой двоичный объект хранился в отдельном файле, а имя этого файла помещалось в таблицу в виде символьного значения, интерпретируемого как указатель на файл. Когда приложению требовалось получить содержимое большого двоичного объекта, связанного с одной из строк таблицы, оно считывало имя файла и извлекало из этого файла двоичные данные. За чтение и запись файла полностью отвечало приложение. Этот подход работал, но был чреват ошибками и требовал от программиста знания интерфейсов *как* самой СУБД, *так и* файловой системы. Недостаточность интеграции между содержимым объекта типа LOB и базой данных была совершенно очевидна. Например, нельзя было попросить СУБД сравнить два больших двоичных объекта, чтобы выяснить, идентично ли их содержимое; СУБД не обеспечивала для таких данных даже элементарных возможностей текстового поиска.

Сегодня большинство ведущих СУБД масштаба предприятия обеспечивает непосредственную поддержку одного или нескольких типов больших объектов. Вы можете определить столбец для хранения такого объекта и включать его в SQL-запросы. Конечно, на операции с большими объектами все же накладываются некоторые ограничения — например, они не могут использоваться для соединения таблиц или в предложении группировки `GROUP BY`.

Sybase поддерживает два типа больших объектов. В столбцах типа `TEXT` может храниться до двух миллиардов байтов текста переменной длины. Имеются некоторые средства поиска содержимого таких столбцов (например, оператор `LIKE`). Второй тип, `IMAGE`, может вмещать до двух миллиардов байтов произвольных двоичных данных переменной длины. Microsoft SQL Server, в дополнение к этим двум типам, поддерживает тип `NTEXT`, позволяющий хранить до миллиарда символов любых национальных языков в двухбайтовой кодировке.

СУБД DB2 компании IBM поддерживает сходный набор типов данных. В ее столбцах типа `CLOB` (character large object) может храниться до двух миллиардов байтов текста. Тип данных `DBCLOB` (double-byte character large object) позволяет хранить до миллиарда символов в двухбайтовой кодировке. А тип `BLOB` (binary large object) служит для хранения до двух миллиардов байтов двоичных данных.

СУБД Oracle исторически предоставляла два типа больших объектов. Тип данных `LONG` вмещал до двух миллиардов байтов текстовых данных, а `LONG RAW` — столько же двоичных данных. Использование любого из этих типов данных огра-

ничивалось одним столбцом таблицы. С выходом Oracle8 поддержка больших объектов была существенно расширена:

- тип BLOB позволяет хранить до 8 Тбайт двоичных данных;
- тип CLOB позволяет хранить до 8 Тбайт однобайтовых символьных данных;
- тип NCLOB позволяет хранить многобайтовые символьные данные как BLOB;
- тип BFILE служит для хранения двоичных данных во внешних (по отношению к базе данных) файлах.

Типы данных BLOB, CLOB и NCLOB интегрированы в Oracle и могут использоваться, например, в транзакциях. Что касается данных типа BFILE, то управление ими осуществляется посредством хранящихся в базе данных указателей на внешние файлы операционной системы. В транзакциях они использоваться не могут. Для управления данными типа BLOB, CLOB и NCLOB из хранимых процедур используется набор специальных функций PL/SQL, о которых рассказывается в следующем разделе.

Похожий набор типов данных поддерживается и в Informix Universal Server. В этой СУБД большие объекты подразделяются на простые и интеллектуальные:

- тип BYTE служит для хранения простых больших объектов с двоичными данными;
- тип TEXT служит для хранения простых больших объектов с текстовыми данными;
- тип BLOB служит для хранения интеллектуальных больших объектов с двоичными данными;
- тип CLOB служит для хранения интеллектуальных больших объектов с текстовыми данными.

В простых объектах может храниться до 2 Гбайт данных. Весь объект должен извлекаться и сохраняться приложением как единое целое и может также перемещаться между базой данных и внешним файлом. В интеллектуальных объектах может храниться до 4 Тбайт данных, и для их обработки применяются специальные функции Informix, позволяющие оперировать данными по частям и получать к ним произвольный доступ наподобие доступа к файлам. Кроме того, для этих объектов Informix использует специальные средства протоколирования, управления транзакциями и поддержания целостности данных.

Специализированная обработка больших объектов

Поскольку большие объекты, по сравнению с обычными элементами данных, обрабатываемыми в СУБД, могут быть очень велики, с их использованием связан ряд проблем.

- **Хранение и оптимизация данных**. Хранение больших объектов прямо в таблицах вместе с другими данными нарушает всю схему оптимизации, основанную на использовании страниц данных, размер которых соответствует размеру дисковых страниц. По этой причине данные LOB всегда

хранятся в отдельных областях дисковой памяти. Большинство ведущих СУБД, поддерживающих большие объекты, предлагают специальные возможности для их хранения, включая именованные области хранения, которые задаются при создании столбца LOB.

- **Запись больших объектов в базу данных**. Поскольку размер больших объектов может достигать десятков и сотен мегабайтов, программы не могут хранить их в своих буферах целиком. Они обрабатывают данные LOB по частям (например, отдельные страницы текстового документа или отдельные кадры видеоклипа). Однако встроенный SQL и традиционные программные интерфейсы SQL осуществляют обработку наборов записей, занося в таблицу с помощью инструкций `INSERT` и `UPDATE` все поля записи за один раз. Для записи данных в столбец LOB по частям используются специальные технологии, позволяющие многократно вызывать одну и ту же API-функцию для одного столбца.
- **Извлечение больших объектов из базы данных**. Здесь проблема та же, что и в случае с их записью, только наоборот. Встроенный SQL и традиционные программные интерфейсы SQL поддерживают только инструкции `SELECT` и `FETCH`, извлекающие данные из всех полей записи за один раз. Однако поскольку объект LOB может иметь размер в десятки и сотни мегабайтов, большинство программ не могут сохранить его в своем буфере целиком. Поэтому разработаны специальные технологии для извлечения таких объектов по частям, чтобы приложение могло их обрабатывать.
- **Протоколирование транзакций**. Для поддержки транзакций большинство СУБД ведет специальные журналы, куда записываются копии данных до и после модификации. Однако из-за размера объектов LOB их запись в журнал транзакций может сильно тормозить всю работу. По этой причине многие СУБД не протоколируют изменения больших объектов или же позволяют включать и отключать режим протоколирования.

В некоторых СУБД этот вопрос решается с помощью дополнительных API-функций, специально предназначенных для управления большими объектами. Они предоставляют приложениям произвольный доступ к отдельным сегментам этих объектов. Например, в Oracle8 можно извлекать и записывать фрагменты объектов LOB в хранимых процедурах, написанных на PL/SQL. Подобные возможности предоставляют и другие объектно-реляционные СУБД, например Informix Universal Server.

Когда хранимая процедура Oracle извлекает из таблицы объект LOB, Oracle на самом деле возвращает не его содержимое, а *локатор* LOB (в объектной терминологии — *дескриптор*). Локатор используется совместно с набором из тридцати пяти специальных функций, с помощью которых хранимая процедура `DBMS_LOB` может манипулировать данными, хранящимися в столбце LOB. Вот краткое описание некоторых из этих функций.

- DBMS_LOB.READ(*локатор,длина,смещение,буфер*) — извлекает в буфер PL/SQL указанное количество байтов/символов из идентифицируемого локатором большого объекта, начиная с заданного смещения.
- DBMS_LOB.WRITE(*локатор,длина,смещение,буфер*) — записывает находящееся в буфере PL/SQL указанное количество байтов/символов в идентифицируемый локатором большой объект, начиная с заданного смещения.
- DBMS_LOB.APPEND(*локатор1,локатор2*) — добавляет все содержимое большого объекта, идентифицируемого вторым локатором, в конец большого объекта, идентифицируемого первым локатором.
- DBMS_LOB.ERASE(*локатор,длина,смещение*) — удаляет указанное количество байтов/символов из большого объекта, идентифицируемого локатором, начиная с заданного смещения; для символьных объектов на место удаленных данных записываются пробелы, а для двоичных объектов — нули.
- DBMS_LOB.COPY(*локатор1,локатор2,длина,смещение1,смещение2*) — копирует указанное количество байтов/символов большого объекта, идентифицируемого вторым локатором, начиная со второго смещения, в большой объект, идентифицируемый первым локатором, начиная с первого смещения.
- DBMS_LOB.TRIM(*локатор,длина*) — усекает большой объект, идентифицируемый локатором, до заданной длины (в байтах/символах).
- DBMS_LOB.SUBSTR(*локатор,длина,смещение*) — возвращает в виде текстовой строки указанное количество байтов/символов из большого объекта, идентифицируемого локатором, начиная с заданного смещения.
- DBMS_LOB.GETLENGTH(*локатор*) — возвращает в виде целого числа длину (в байтах/символах) большого объекта, идентифицируемого локатором.
- DBMS_LOB.COMPARE(*локатор1,локатор2,длина,смещение1,смещение2*) — сравнивает указанное количество байтов/символов большого объекта, идентифицируемого первым локатором, начиная с первого смещения, и большого объекта, идентифицируемого вторым локатором, начиная со второго смещения; возвращает нуль, если заданные фрагменты объектов одинаковы, и ненулевое значение в противном случае.
- DBMS_LOB.INSTR(*локатор,шаблон,смещение,i*) — возвращает в виде целого значения позицию *i*-го вхождения шаблона в большой объект, идентифицируемый локатором, начиная с заданного смещения.

Oracle накладывает еще одно ограничение на обновление и модификацию больших объектов, выполняемые посредством описанных функций. Поскольку большие объекты слишком громоздки, чтобы участвовать в транзакциях, Oracle обычно не блокирует их при выборке строки таблицы приложением или хранимой процедурой PL/SQL. Чтобы обновить большой объект, его сначала нужно яв-

но заблокировать. Для этого в инструкцию SELECT, возвращающую локатор LOB, включается предложение FOR UPDATE. Вот фрагмент хранимой процедуры PL/SQL, которая извлекает из таблицы большой объект, содержащий текстовый документ, и обновляет 100 символов в средине этого документа.

```
declare
    lob     CLOB;
    textbuf varchar(255);

begin
   /* Помещаем в буфер текст, подлежащий вставке */
   ...

   /* Получаем локатор большого объекта и блокируем
      этот объект для обновления */
   select document_lob into lob
     from documents
    where document_id = '34218'
      for update;

   /* Записываем в объект новые 100 байт текста  */
   dbms_lob.write(lob, 100, 500, textbuf);

   commit;
end;
```

Абстрактные (структурированные) типы данных

Традиционная реляционная база данных оперирует простыми, неделимыми, атомарными значениями данных. Если элемент данных, такой как адрес, может быть разделен на составляющие — улица, дом, город, штат, почтовый индекс, — тогда у проектировщика базы данных есть выбор. Он может разделить весь адрес на четыре компонента и хранить их в отдельных столбцах. Или же он может интерпретировать весь адрес как единое целое, и в этом случае компоненты адреса нельзя будет обрабатывать по отдельности. Промежуточного варианта, позволяющего интерпретировать адрес как одно целое и в то же время иметь доступ к его отдельным элементам, реляционная модель не предусматривает.

Однако такой способ работы с данными предусмотрен во многих языках программирования (включая и такие не объектно-ориентированные языки, как C и Pascal). Эти языки поддерживают составные типы данных, называемые структурами. Структура объединяет обычные переменные и вложенные структуры, доступ к которым может осуществляться по отдельности. При этом вся структура может обрабатываться как одно целое, если это оказывается более удобным. Структурированные, или составные, типы данных в ОРБД предоставляют аналогичные возможности в контексте СУБД.

В Informix Universal Server поддерживаются абстрактные типы данных (АТД) посредством концепции *типов строчных данных*, или *записей*. Запись можно рассматривать как структурированный набор элементов, называемых *полями*. Ниже приведена инструкция CREATE TABLE, создающая таблицу PERSONNEL, в которой для хранения имени и адреса используются вложенные записи.

```
CREATE TABLE PERSONNEL (
    EMPL_NUM INTEGER,
        NAME ROW(
           F_NAME VARCHAR(15),
           M_INIT CHAR(1),
           L_NAME VARCHAR(20))
     ADDRESS ROW(
      STREET VARCHAR(35),
        CITY VARCHAR(15),
       STATE CHAR(2),
    POSTCODE ROW(
        MAIN INTEGER,
         SFX INTEGER)));
```

В этой таблице три столбца. Первый, EMPL_NUM, содержит целые числа. Следующие два, NAME и ADDRESS, содержат записи, на что указывает ключевое слово ROW, за которым в скобках следует список полей. Записи в столбце NAME состоят из трех полей, а в столбце ADDRESS — из четырех. Последнее поле столбца ADDRESS (POSTCODE) само является записью, состоящей из двух полей. Таким образом, в нашем простом примере создается двухуровневая иерархия, однако глубина иерархии полей может быть (и часто бывает) большей.

Для доступа к отдельным полям столбца используется запись с точками, такая же как при записи полных имен столбцов. Добавление точки *после* имени столбца позволяет указывать имена отдельных полей столбца. Вот пример инструкции SELECT, извлекающей идентификаторы, имена и фамилии служащих с заданным почтовым индексом.

```
SELECT EMPL_NUM, NAME.F_NAME, NAME.L_NAME
  FROM PERSONNEL
 WHERE ADDRESS.POSTCODE.MAIN = '12345';
```

Предположим, что в базе данных имеется еще одна таблица, MANAGERS, в которой есть столбец NAME точно такой же структуры, как и в таблице PERSONNEL. Тогда следующий запрос возвращает идентификаторы служащих, которые являются менеджерами.

```
SELECT EMPL_NUM
  FROM PERSONNEL, MANAGERS
 WHERE PERSONNEL.NAME = MANAGERS.NAME;
```

В первом из этих двух запросов из столбца NAME извлекаются значения отдельных полей. Второй запрос демонстрирует ситуацию, когда удобнее оперировать этим же столбцом *в целом*, как записью (всеми тремя полями), — мы использовали этот способ для сравнения данных. Очевидно, что гораздо удобнее попросить СУБД сравнить два столбца целиком, вместо того чтобы сравнивать все их поля по отдельности. Эти примеры демонстрируют преимущества типа ROW, обеспечивающего доступ к полям на любом уровне иерархии.

При добавлении данных в таблицу столбцы типа ROW требуют специальной обработки. Поскольку в таблице PERSONNEL три столбца, в предложении VALUES предназначенной для нее инструкции INSERT должно быть три элемента. Для столбцов типа ROW используется специальный конструктор записи, который группирует отдельные значения полей, формируя из них корректную запись. Вот

пример инструкции INSERT для таблицы PERSONNEL, иллюстрирующий применение конструктора.

```
INSERT INTO PERSONNEL
     VALUES (1234,
             ROW('John', 'J', 'Jones'),
             ROW('197 Rose St.', 'Chicago', 'IL',
                 ROW(12345, 6789)));
```

Определение абстрактных типов данных

Использование данных типа ROW в таблицах Informix имеет один недостаток: структура записи определяется для каждого столбца в отдельности. Если же в двух таблицах понадобятся столбцы одинакового типа, придется определять его дважды. Это противоречит одному из ключевых принципов объектно-ориентированного проектирования — повторному использованию определений объектов. Вместо того чтобы определять структуру каждого конкретного объекта (в данном случае столбца каждой из двух таблиц), разработчик базы данных должен иметь возможность сначала сформировать тип записи, а потом использовать его при последующем создании столбцов. СУБД Informix Universal Server обеспечивает такую возможность, позволяя создавать *именованные записи*. (Записи, о которых рассказывалось в предыдущем разделе, были *безымянными*.)

Именованная запись создается с помощью инструкции CREATE ROW TYPE.

```
CREATE ROW TYPE NAME_TYPE (
    F_NAME VARCHAR(15),
    M_INIT CHAR(1),
    L_NAME VARCHAR(20));

CREATE ROW TYPE POST_TYPE (
      MAIN INTEGER,
       SFX INTEGER);

CREATE ROW TYPE ADDR_TYPE (
    STREET VARCHAR(35),
      CITY VARCHAR(15),
     STATE CHAR(2),
  POSTCODE POST_TYPE);
```

Обратите внимание на то, что в определении именованной записи может использоваться другой, созданный ранее, тип записи. Например, последнее из полей записи ADDR_TYPE имеет тип POST_TYPE. Теперь все три созданных нами типа могут использоваться не только в определении таблицы PERSONNEL, но и, самое главное, в определении любых других таблиц, где нужны столбцы для хранения адресов, имен и почтовых индексов. Абстрактные типы данных очень помогают унифицировать структуру информации в ОРБД. Вот как выглядит новое определение таблицы PERSONNEL с использованием созданных выше абстрактных типов данных.

```
CREATE TABLE PERSONNEL (
    EMPL_NUM INTEGER,
        NAME NAME_TYPE,
     ADDRESS ADDR_TYPE);
```

На рис. 24.1 приведен пример такой таблицы, отражающий иерархическую структуру ее столбцов.

Таблица PERSONNEL

	NAME			ADDRESS				
							POSTCODE	
EMPL_NUM	F_NAME	M_INIT	L_NAME	STREET	CITY	STATE	MAIN	SFX
1234	Sue	J.	Marsh	1803 Main St.	Alamo	NJ	31948	4567
1374	Sam	F.	Wilson	564 Birch Rd.	Marion	KY	82942	3524
1421	Joe	P.	Jones	13 High St.	Delano	NM	13527	2394
1432	Rob	G.	Mason	9123 Plain Av.	Franklin	PA	83624	2643
			•					
			•					
			•					

Рис. 24.1. Использование абстрактных типов данных в таблице PERSONNEL

Oracle также поддерживает абстрактные типы данных, но использует несколько иной синтаксис. Вот как выглядит в Oracle инструкция CREATE TYPE, создающая такие же типы данных, как и в предыдущем примере.

```
CREATE TYPE NAME_TYPE AS OBJECT (
     F_NAME VARCHAR2(15),
     M_INIT CHAR(1),
     L_NAME VARCHAR2(20));

CREATE TYPE POST_TYPE AS OBJECT (
       MAIN NUMBER,
        SFX NUMBER);

CREATE TYPE ADDR_TYPE AS OBJECT (
     STREET VARCHAR2(35),
       CITY VARCHAR2(15),
      STATE CHAR(2),
   POSTCODE POST_TYPE);
```

В Oracle абстрактные типы данных называются объектами. Фактически, определенный таким образом тип данных функционирует подобно классу объектов, если придерживаться традиционной объектно-ориентированной терминологии. Распространяя эту терминологию и на состав объектов, Oracle называет отдельные компоненты абстрактных типов данных (соответствующие *полям* в Informix) *атрибутами*. Например, у типа данных ADDR_TYPE четыре атрибута. Последний из них, POSTCODE, сам является абстрактным типом данных.

И в Oracle, и в Informix для доступа к отдельным элементам абстрактных типов данных применяется запись с точками. Например, поле MAIN почтового индекса в таблице PERSONNEL идентифицируется следующим образом.

```
PERSONNEL.ADDRESS.POSTCODE.MAIN
```

Если бы таблица принадлежала другому пользователю, например Сэму, полное имя атрибута было бы еще длиннее.

```
SAM.PERSONNEL.ADDRESS.POSTCODE.MAIN
```

Informix допускает еще более широкое использование абстрактных типов данных — они могут служить шаблонами не только для отдельных столбцов, но и для целых таблиц. Например, создав следующий абстрактный тип данных

```
CREATE ROW TYPE PERS_TYPE (
  EMPL_NUM INTEGER,
      NAME NAME_TYPE,
   ADDRESS ADDR_TYPE);
```

можно определить таблицу PERSONNEL следующим образом.

```
CREATE TABLE PERSONNEL
     OF TYPE PERS_TYPE;
```

Столбцы этой таблицы будут точно такими же, что и в рассматривавшихся ранее примерах, но теперь PERSONNEL — *типизированная таблица*. Типизированные таблицы помогают формализовать структуру объектов базы данных. Для каждого класса объектов определяется собственный тип записи, и все таблицы, которые содержат объекты данного класса, определяются посредством этого типа. Кроме того, типизированные таблицы являются ключевым элементом используемой в Informix концепции наследования, о которой рассказывается далее в настоящей главе.

Использование абстрактных типов данных

К сожалению, синтаксис обновления и добавления данных структурированных типов достаточно сложен. С неименованными записями в Informix Universal Server особых сложностей не возникает. Значения, записываемые в столбец типа ROW, должны просто иметь соответствующее количество полей правильных типов. Для их формирования используется конструктор записи, который объединяет отдельные поля в единую запись.

К именованным типам данных предъявляются более строгие требования. Данные, которые записываются в столбец абстрактного типа данных, должны иметь в точности тот же тип. Для этого в инструкции INSERT или UPDATE необходимо выполнить явное приведение типа сконструированной записи к типу данных столбца.

```
INSERT INTO PERSONNEL
     VALUES (1234,
             ROW('John', 'J', 'Jones')::NAME_TYPE,
             ROW('197 Rose St.', 'Chicago', 'IL',
                 ROW(12345, 6789)));
```

Первый оператор :: приводит созданную конструктором запись (состоящую из трех полей) к типу NAME_TYPE, чтобы ее можно было присвоить столбцу NAME. Остальные два оператора делают то же самое для формирования значения столбца ADDRESS.

В Oracle принят немного иной подход к конструированию составных элементов данных и их вставке в столбцы АТД. Когда вы создаете АТД (с помощью инструкции CREATE TYPE), Oracle автоматически определяет *метод-конструктор* для это-

го типа. Конструктор можно представить себе как функцию, которая принимает в качестве аргументов отдельные значения атрибутов, объединяет их и возвращает значение соответствующего абстрактного типа. Именованные конструкторы используются в предложении VALUES инструкции INSERT. Вот как это выглядит на примере таблицы PERSONNEL.

```
INSERT INTO PERSONNEL
     VALUES (1234,
             NAME_TYPE('John', 'J', 'Jones'),
             ADDR_TYPE('197 Rose St.', 'Chicago', 'IL',
                       POST_TYPE(12345, 6789)));
```

Конструкторы NAME_TYPE, ADDR_TYPE и POST_TYPE выполняют ту же функцию, что и конструктор ROW в Informix, плюс осуществляют приведение типов, необходимое для занесения данных в соответствующие столбцы.

Наследование

Поддержка абстрактных типов данных составляет основу объектно-ориентированных расширений реляционной модели. Еще одной важной составляющей объектной модели является *наследование*. Путем наследования новый класс объектов может быть определен как конкретная разновидность существующего класса и наследовать его атрибуты и поведение.

На рис. 24.2 представлен пример того, как применяется наследование при построении модели, описывающей иерархию служащих компании. Все служащие являются членами класса PERSONNEL и имеют стандартные атрибуты этого класса: идентификатор, имя и адрес. Некоторые служащие являются коммерческими представителями и имеют дополнительные атрибуты: плановый объем продаж и идентификатор менеджера по продажам. Другие служащие являются инженерами и у них иной набор атрибутов: ученая степень, текущий проект, над которым они работают, и т.п. Для каждой из этих категорий служащих определен собственный класс, который является подклассом PERSONNEL. Подкласс наследует все характеристики родительского класса (для инженеров и коммерческих представителей должны храниться обычные сведения, связанные с любым служащим). В то же время подкласс может иметь и дополнительные атрибуты, уникальные для его категории объектов. На рис. 24.2 иерархия классов служащих имеет три уровня: инженеры в ней подразделяются на технологов, разработчиков и руководителей.

Рис. 24.2. Иерархия служащих компании

Механизм наследования в Informix Universal Server обеспечивает очень простой способ определения абстрактных типов, соответствующих иерархии служащих, представленной на рис. 24.2. Предположим, что в базе данных Informix уже создан тип PERS_TYPE, определение которого мы приводили ранее в этой главе, и на его основе создана типизированная таблица PERSONNEL. Тогда, используя механизм наследования, можно создать АТД для других классов нашей иерархии.

```
CREATE ROW TYPE SALES_TYPE (
   SLS_MGR INTEGER,          /* Идентификатор менеджера */
    SALARY DECIMAL(9,2),     /* Годовой оклад           */
     QUOTA DECIMAL(9,2))
     UNDER PERS_TYPE;

CREATE ROW TYPE ENGR_TYPE (
    SALARY DECIMAL(9,2),     /* Годовой оклад           */
 YRS_EXPER INTEGER)          /* Стаж                    */
     UNDER PERS_TYPE;

CREATE ROW TYPE MGR_TYPE (
     BONUS DECIMAL(9,2))     /* Годовой бонус           */
     UNDER ENGR_TYPE;

CREATE ROW TYPE TECH_TYPE (
 WAGE_RATE DECIMAL(5,2))     /* Почасовая ставка        */
     UNDER ENGR_TYPE;
```

АТД, который мы определили для технологов (TECH_TYPE), является подтипом (подклассом) АТД для инженеров (ENGR_TYPE), поэтому он наследует все поля этого АТД *плюс* все поля АТД для служащих (PERS_TYPE) *плюс* имеет собственные дополнительные поля. В общем случае *подтипом* некоторого АТД называется тип данных, который расположен ниже в иерархии и наследует все его поля. В определении подтипа должно присутствовать предложение UNDER с указанием наследуемого типа. И наоборот, тип более высокого уровня является *надтипом* для типов более низкого уровня, определенных под ним.

Теперь, определив иерархию типов данных, можно легко создать на их основе типизированные таблицы Informix.

```
CREATE TABLE ENGINEERS
     OF TYPE ENGR_TYPE;
CREATE TABLE TECHNICIANS
     OF TYPE TECH_TYPE;
CREATE TABLE MANAGERS
     OF TYPE MGR_TYPE;
CREATE TABLE REPS
     OF TYPE SALES_TYPE;
```

Как видите, с использованием такой иерархии вся сложность модели данных переносится на определения абстрактных типов, а определения таблиц становятся предельно простыми. Разумеется, все остальные характеристики таблиц могут (и должны) быть заданы в определениях таблиц. Например, таблица REPS содержит столбец, который является внешним ключом для таблицы PERSONNEL, поэтому в ее определение должно быть включено предложение FOREIGN KEY.

```
CREATE TABLE REPS
     OF TYPE SALES_TYPE
 FOREIGN KEY (SLS_MGR)
  REFERENCES PERSONNEL(EMPL_NUM);
```

Хотя наследование создает связь между структурами таблиц, основанных на одном АТД, однако в отношении самих данных таблицы остаются независимыми. Строка, добавленная в таблицу TECHNICIANS, не появится автоматически ни в таблице ENGINEERS, ни в таблице PERSONNEL. Каждая из этих таблиц содержит собственные данные, которые добавляются в нее отдельно. Тем не менее в Informix можно связать и данные таблиц, для чего в этой СУБД определен еще один тип наследования — *табличное наследование*, превращающее таблицы в нечто очень близкое к классам объектов.

Табличное наследование: реализация классов

В Informix Universal Server имеется интересная возможность, отдаляющая структуру таблиц базы данных от традиционной реляционной модели и приближающая ее к концепции класса объектов, — это *табличное наследование*. Посредством данной методики можно создать иерархию типизированных таблиц (классов), подобную изображенной на рис. 24.3. Хотя эти таблицы по-прежнему базируются на созданной нами иерархии АТД, они сами создают параллельную иерархию.

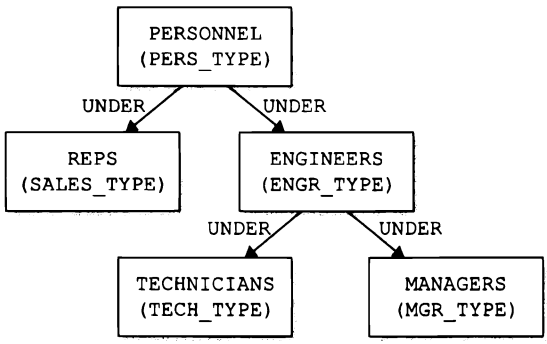


Рис. 24.3. Иерархия таблиц Informix с табличным наследованием

А вот набор инструкций CREATE TABLE, создающих эту иерархию.

```
CREATE TABLE ENGINEERS
     OF TYPE ENGR_TYPE
       UNDER PERSONNEL;

CREATE TABLE TECHNICIANS
     OF TYPE TECH_TYPE
       UNDER ENGINEERS;

CREATE TABLE MANAGERS
     OF TYPE MGR_TYPE
       UNDER ENGINEERS;

CREATE TABLE REPS
     OF TYPE SALES_TYPE
       UNDER PERSONNEL;
```

Когда таблица создается подобным образом, она наследует не только набор столбцов исходной таблицы, но и ряд других ее характеристик: первичный и внешние ключи, условия ссылочной целостности и ограничения на значения, триггеры, индексы, области хранения данных и другие специфические для Informix характеристики. При этом наследуемые характеристики в инструкции CREATE TABLE можно заменять новыми определениями.

Иерархическая связь таблиц оказывает влияние на то, как Informix Universal Server интерпретирует их строки. Иерархически связанные таблицы образуют коллекцию вложенных *множеств* записей (рис. 24.4). Когда строка добавляется в иерархию таблиц, она по-прежнему добавляется в конкретную таблицу. Например, Джо Джонс (Joe Jones) является технологом (запись о нем входит в таблицу TECHNICIANS), тогда как Сэм Уилсон (Sam Wilson) — разработчик (таблица ENGINEERS), а Сью Марш (Sue Marsh) — рядовая служащая (таблица PERSONNEL). А вот SQL-запросы теперь действуют совершенно иначе. Когда вы выполняете запрос к одной таблице иерархии, он возвращает строки не только из этой таблицы, но и из *всех* ее подтаблиц. Например, следующий запрос

```
SELECT *
  FROM PERSONNEL;
```

возвращает строки не только из таблицы PERSONNEL, но и из таблиц ENGINEERS, TECHNICIANS, MANAGERS и REPS. Аналогичным образом запрос

```
SELECT *
  FROM ENGINEERS;
```

вернет строки из таблиц ENGINEERS, TECHNICIANS и MANAGERS. СУБД интерпретирует таблицы как вложенные наборы записей, и запрос к любой из них возвращает все записи, входящие в соответствующий ей набор. Если же вам нужны только записи из таблицы верхнего уровня, включите в запрос ключевое слово ONLY.

```
SELECT *
  FROM ONLY(ENGINEERS);
```

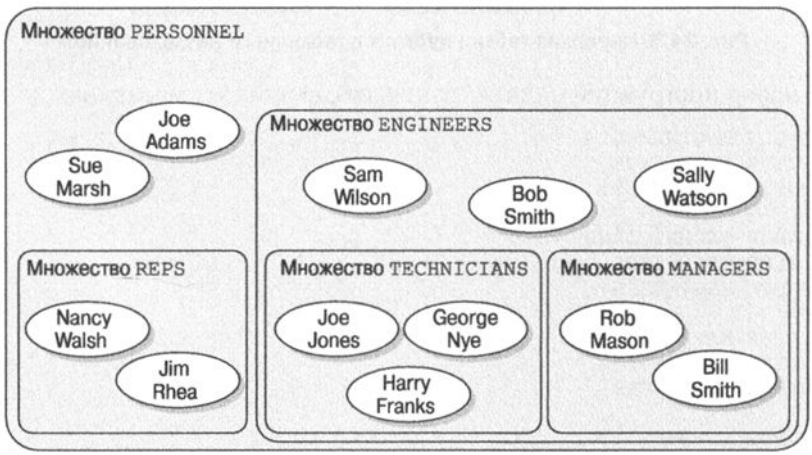


Рис. 24.4. Вложенные таблицы в иерархии наследования

Ту же логику СУБД применяет и в отношении запросов на удаление. Например, следующая инструкция DELETE

```
DELETE FROM PERSONNEL
 WHERE EMPL_NUM = 1234;
```

успешно удаляет запись о служащем с идентификатором 1234 независимо от того, в какой таблице иерархии она находится на самом деле. СУБД интерпретирует эту инструкцию так: "Удалить из набора записей PERSONNEL все строки, соответствующие заданному условию". Если же вы хотите удалить только те строки, которые хранятся в конкретной таблице иерархии, например в таблице ENGINEERS, но не в ее подчиненных таблицах, тогда, как и в случае с запросами на выборку, вам нужно воспользоваться ключевым словом ONLY.

```
DELETE FROM ONLY(ENGINEERS)
 WHERE EMPL_NUM = 1234;
```

Та же логика применяется и к инструкциям UPDATE. Например, следующая инструкция изменяет фамилию служащего независимо от того, в какой таблице иерархии хранится его запись.

```
UPDATE PERSONNEL
   SET L_NAME = 'Harrison'
 WHERE EMPL_NUM = 1234;
```

Если же нужно ограничить область действия инструкции UPDATE одной таблицей, то, как и в предыдущих примерах, можно воспользоваться ключевым словом ONLY.

Конечно, в инструкциях, действующих на определенном уровне иерархии, могут использоваться только те столбцы, которые определены на этом уровне. Например, следующая инструкция будет неправильной.

```
DELETE FROM PERSONNEL
 WHERE SALARY < 20000.00;
```

Дело в том, что в таблице PERSONNEL, расположенной на верхнем уровне иерархии, нет столбца SALARY. Этот столбец определен для некоторых ее подтаблиц. А вот такая инструкция вполне допустима.

```
DELETE FROM MANAGERS
 WHERE SALARY < 20000.00;
```

На этом уровне иерархии столбец SALARY определен.

Объектно-реляционные технологии в Informix Universal Server выходят далеко за рамки обычных реляционных баз данных. Фанаты реляционной архитектуры считают операции, выполняемые в приведенных выше примерах, опасными и нелогичными. "Почему, — спрашивают они, — добавление строки в одну таблицу вызывает ее неожиданное появление в двух других таблицах? Почему после выполнения запроса на удаление, условию которого не соответствует ни одна строка указанной в нем таблицы, вдруг исчезают строки из других таблиц?" Ответ прост: иерархия таблиц не является строго реляционным набором независимых таблиц, она ведет себя совершенно иначе и обладает рядом характеристик иерархии классов. Хорошо это или плохо — зависит от вашей точки зрения. Тем, кто хочет работать с ОРБД, нужно принять новый взгляд на вещи и понять, что они имеют дело

с иначе построенной моделью мира, — поведение ее объектов нельзя оценивать с точки зрения логики реляционной модели. И первое время нужно будет внимательно следить за тем, чтобы не делать реляционных предположений о результатах выполнения инструкций SQL.

Множества, массивы и коллекции

В реляционной базе данных таблица является *единственной* информационной структурой для представления множества объектов. Например, множество инженеров в нашей базе данных представлено строками в таблице ENGINEERS. Предположим, что каждый инженер имеет ряд ученых степеней (бакалавр естественных наук Массачусетсского технологического института, доктор философии Университета штата Мичиган и т.д.), информация о которых хранится в базе данных. Количество степеней у каждого инженера свое — их может не быть вовсе, а может быть и с полдюжины. В строго реляционной базе данных существует только один "правильный" способ добавления этой информации в модель данных: должна быть создана новая таблица DEGREES (рис. 24.5). Каждая строка этой таблицы представляет одну ученую степень одного инженера. Первый столбец каждой строки таблицы DEGREES содержит идентификатор инженера, степень которого описывается этой строкой, и служит внешним ключом к таблице ENGINEERS, образуя между таблицами отношение "предок-потомок". Остальные столбцы описывают саму степень.

Таблица **ENGINEERS**

	NAME			ADDRESS						
							POSTCODE			
EMPL NUM	F NAME	M INIT	L NAME	STREET	CITY	STATE	MAIN	SFX	SALARY	YRS EXPER
1234	Bob	J.	Smith	956 Elm Rd.	Forest	NY	38294	4567	$45,000	6
1374	Sam	F.	Wilson	564 Birch Rd.	Marion	KY	82942	3524	$30,000	12
1421	Sally	P.	Watson	87 Dry Lane	Mt Erie	DL	73853	2394	$34,500	9
1532			▪							
			▪							
			▪							

Таблица **DEGREES**

EMPL_NUM	DEGREE	SCHOOL
1245	BS	Michigan
1245	MS	Purdue
1374	BS	Lehigh
1439	BS	MIT
1436	BS	MIT
1439	MBA	Stanford
	▪	
	▪	
	▪	

Внешний ключ

Рис. 24.5. Реляционная модель информации об инженерах и их ученых степенях

Реляционную структуру таблиц, изображенную на рис. 24.5, вы видели в этой книге уже множество раз — именно так с самого начала строились все реляционные базы данных. Однако у этой схемы есть определенные недостатки. Во-первых, в построенной на ее основе базе данных обычно оказывается очень много таблиц и межтабличных связей на основе внешних ключей, в результате чего ее структура

становится громоздкой и сложной для понимания. Во-вторых, многие типичные запросы требуют соединения трех, четырех и более таблиц. В-третьих, в большинстве СУБД соединения реализованы таким образом, что чем больше таблиц объединяется в запросе, тем медленнее он выполняется.

Для объектной модели, связывающей данные об инженерах и их ученых степенях, структура таблиц, представленная на рис. 24.5, совершенно не подходит. С точки зрения этой модели, ученые степени не являются отдельными объектами, а потому им не нужна собственная таблица. Степени являются *атрибутами* имеющего их инженера. Тот факт, что их количество переменно, для объектной модели не составляет проблемы, поскольку переменное количество однородных элементов всегда можно представить в виде массива внутри объекта, хранящего данные об инженере.

Объектно-реляционные базы данных поддерживают множества, массивы и другие типы коллекций. Их можно использовать для определения столбцов таблиц. Такой столбец будет содержать не одно значение в каждой ячейке, а набор значений. С помощью специальных расширений синтаксиса SQL пользователь или хранимая процедура могут манипулировать коллекцией данных как единым целым или обращаться к ее отдельным элементам.

Определение коллекций

Informix Universal Server поддерживает коллекции атрибутов при помощи *типов данных коллекций*. Поддерживаются три различных типа данных.

- **Список** — это упорядоченный набор значений одного типа. В списке есть первый, последний и *n*-й элемент. Элементы списка не обязательно должны быть уникальными. Например, список имен сотрудников, нанятых в текущем году, может быть таким (в порядке приема на работу): `{'Jim', 'Mary', 'Sam', 'Jim', 'John'}`.
- **Мультимножество** — это неупорядоченный набор значений одного типа. В нем нет ни первого, ни последнего, ни *n*-го элемента — все они равнозначны. Элементы мультимножества не обязательно должны быть уникальными. Например, список имен сотрудников будет мультимножеством, если вас не интересует порядок их найма: `{'Jim', 'Sam', 'John', 'Mary', 'Jim'}`.
- **Множество** — это неупорядоченный набор уникальных значений одного типа. В нем тоже нет ни первого, ни последнего, ни *n*-го элемента, но все элементы множества обязательно должны иметь уникальные значения. Например, приведенный выше список имен сотрудников не может считаться множеством, а вот набор их фамилий может — `{'Johnson', 'Samuels', 'Wright', 'Jones', 'Smith'}`.

Чтобы проиллюстрировать концепцию коллекций, расширим таблицы нашей ОРБД следующим образом.

- Таблица REPS будет включать плановые объемы продаж на первый, второй, третий и четвертый кварталы. Квартальные планы логичнее всего представить в виде списка, так как они имеют естественный порядок (от первого до четвертого), один и тот же тип данных (денежный) и их значения не обязательно должны быть уникальны (т.е. планы первого и второго кварталов вполне могут совпадать).
- Таблица ENGINEERS будет включать информацию об ученых степенях инженеров. Эта информация будет состоять из двух компонентов: собственно степени (BS — бакалавр естественных наук, PhD — доктор философии и т.п.) и организации, в которой она получена. Данные будут храниться в виде мультимножества, поскольку инженер может иметь две одинаковые степени, например бакалавра естественных наук и бакалавра бизнеса, полученные в одном и том же учебном заведении.
- Таблица TECHNICIANS будет включать информацию о проектах, в которых участвует каждый технолог. Технолог может участвовать и в нескольких проектах, но у каждого проекта будет уникальное название. Поэтому данные о проектах будут храниться в виде множества.

Вот инструкции ALTER TABLE, которые вносят в структуру таблиц нашей базы данных описанные изменения.

```
ALTER TABLE REPS            /* Четыре квартальных плана */
        ADD QTR_TGT LIST(DECIMAL(9,2));

ALTER TABLE TECHNICIANS     /* Проекты                  */
        ADD PROJECTS SET(VARCHAR(15));

ALTER TABLE ENGINEERS       /* Ученые степени           */
        ADD DEGREES MULTISET(ROW(
     DEGREE VARCHAR(3),
     SCHOOL VARCHAR(15)));
```

Полученные таблицы представлены на рис. 24.6; структуру можно назвать "строки в строках". В случае таблицы ENGINEERS, ее структуру более точно можно описать как "таблица в таблице". Как видите, коллекции вносят серьезные изменения в реляционную модель, основанную на хранении строк и столбцов с атомарными элементами. С появлением коллекций, таблицы как бы растягиваются, чтобы вместить дополнительные данные.

Informix Universal Server допускает самое широкое использование коллекций. Коллекция может быть полем записи, а сами записи могут быть элементами коллекции. При желании можно даже определить коллекцию в коллекции. Например, проекты в нашем примере могут состоять из подчиненных проектов, участие в которых тоже можно отслеживать в базе данных. С добавлением каждого нового уровня сложности усложняется как SQL, так и язык хранимых процедур, которым нужны средства для манипулирования новыми типами данных.

Таблица REPS

	NAME			ADDRESS								
							POSTCODE					
EMPL_NUM	F_NAME	M_INIT	L_NAME	STREET	CITY	STATE	MAIN	SFX	SLS_MGR	SALARY	QUOTA	QTR_TGT
4267	Nancy	Q.	Walsh	•••	•••	•••	•••	•••	2598	$35000	$750000	$160000 $190000 $210000 $190000
4316	Jim	F.	Rea	•••	•••	•••	•••	•••	2598	$32000	$690000	$120000 $165000 190000 $215000
			•									

Таблица TECHNICIANS

	NAME			ADDRESS						
							POSTCODE			
EMPL_NUM	F_NAME	M_INIT	L_NAME	STREET	CITY	STATE	MAIN	SFX	WAGE_RATE	PROJECT
1421	Joe	P.	Jones	•••	•••	•••	•••	•••	$16.75	bingo at las checkmate
1537	Harry	E.	Franks	•••	•••	•••	•••	•••	$20.50	at las
1618	George	W.	Nye	•••	•••	•••	•••	•••	$19.75	gonzo bingo
			•							

Таблица ENGINEERS

	NAME			ADDRESS							DEGREES	
							POSTCODE					
EMPL_NUM	F_NAME	M_INIT	L_NAME	STREET	CITY	STATE	MAIN	SFX	SALARY	YRS_EXPER	DEGREE	SCHOOL
1234	Bob	J.	Smith	•••	•••	•••	•••	•••	$45000	6	BS MS	Michigan Purdue
1374	Sam	F.	Wilson	•••	•••	•••	•••	•••	$30000	12	BS	Lebigh
1439	Sally	P.	Watson	•••	•••	•••	•••	•••	$34500	9	BS BS MBA	MIT MIT Stanford
				•								

Рис. 24.6. Таблицы со столбцами, хранящими коллекции

Oracle также поддерживает коллекции посредством двух объектно-реляционных расширений.

- **Массив переменного размера** — это упорядоченный набор значений одного типа. Элементы массива не обязательно должны быть уникальными. При создании столбца этого типа следует задать максимальное количество элементов массива. Для доступа к отдельным элементам массивов в Oracle поддерживается расширенный синтаксис SQL.
- **Вложенная таблица** — это самая настоящая таблица в таблице. Столбец данного типа содержит ячейки, которые сами являются таблицами. Oracle хранит содержимое вложенной таблицы отдельно от главной таблицы, но поддерживает расширенный синтаксис SQL, позволяющий обращаться к вложенной таблице как к столбцу главной таблицы. В отличие от массива переменной длины, вложенная таблица может содержать любое количество строк.

Столбец таблицы может быть объявлен как VARRAY (массив переменной длины) или TABLE OF (вложенная таблица). Вот несколько инструкций CREATE TYPE и CREATE TABLE, в которых для создания таблиц, аналогичных представленным на рис. 24.6, используются два новых типа данных.

```
CREATE TYPE TGT_ARRAY AS
  VARRAY(4) OF NUMBER(9,2);

CREATE TABLE REPS (
    EMPL_NUM NUMBER,
        NAME NAME_TYPE,
     ADDRESS ADDR_TYPE,
     SLS_MGR NUMBER,       /* Менеджер          */
      SALARY NUMBER(9,2),  /* Оклад             */
       QUOTA NUMBER(9,2),  /* План продаж       */
     QTR_TGT TGT_ARRAY);   /* Квартальные планы */

CREATE TYPE DEGR_TYPE AS OBJECT (
     DEGREE VARCHAR2(3),
     SCHOOL VARCHAR2(15));

CREATE TYPE DEGR_TABLE AS
      TABLE OF DEGR_TYPE;

CREATE TABLE ENGINEERS (
    EMPL_NUM NUMBER,
        NAME NAME_TYPE,
     ADDRESS ADDR_TYPE,
      SALARY NUMBER (9,2), /* Оклад */
   YRS_EXPER NUMBER,       /* Стаж  */
     DEGREES DEGR_TABLE)
NESTED TABLE DEGREES STORE AS ENGINEERS_DEGREES;
```

Плановые объемы продаж по кварталам проще всего представить в виде массива переменной длины. Поскольку кварталов ровно четыре, максимальный размер массива известен заранее. В этом примере массив содержит простые значения, но в общем случае он может хранить данные абстрактного (структурированного) типа.

Информация об ученых степенях представлена в виде вложенной таблицы. Конечно, можно было бы заранее установить максимальное количество строк и использовать массив структур, но в общем случае, когда число строк спрогнозировать трудно, удобнее воспользоваться вложенной таблицей. Мы создали для нее АТД с двумя атрибутами. Каждая строка вложенной таблицы будет содержать информацию об ученой степени и учебном заведении, в котором она была получена.

Коллекции и запросы на выборку

Наличие столбцов, содержащих коллекции, усложняет формулирование запросов к таблицам и обработку их результатов. Объектно-реляционные СУБД обычно поддерживают набор специальных расширений SQL, позволяющих выполнять простейшие запросы, включающие коллекции данных. В случае более сложных запросов обычно приходится писать хранимые процедуры с циклами для обработки элементов коллекций.

Informix трактует коллекции как наборы значений, подобные тем, что возвращаются подчиненными запросами. Для проверки вхождения некоторого значения в коллекцию используется оператор IN. Вот пример запроса, возвращающего список технологов, работающих над проектом "bingo".

```
SELECT EMPL_NUM, NAME
  FROM TECHNICIANS
 WHERE 'bingo' IN (PROJECTS);
```

Имя столбца, содержащего коллекцию (в данном случае столбца PROJECTS), указывается в скобках после оператора IN. В интерактивном режиме можно включать такие столбцы (типа SET, LIST или MULTISET) в предложение SELECT — выполнив запрос, Informix выведет все содержимое этих столбцов на экран. Для обработки результатов подобных запросов в приложениях (использующих встроенный SQL или библиотеку API-функций) нужно использовать специальные API-функции или расширения Informix SPL.

В Oracle имеются дополнительные возможности обработки вложенных таблиц в SQL-запросах. В новейших версиях имеется специальная функция TABLE, которая превращает таблицу со столбцом, содержащим вложенные таблицы, в одну большую таблицу, содержащую по одной строке для каждой строки каждой вложенной таблицы (в более старых версиях эта функциональность обеспечивалась ключевым словом THE (ныне объявленным нежелательным для использования), с небольшими отличиями в синтаксисе). Вот пример запроса, выводящего список учебных заведений, в которых заданный инженер получил свои ученые степени.

```
SELECT T2.SCHOOL
  FROM ENGINEERS T1, TABLE(T1.DEGREES) T2
 WHERE EMPL_NUM = 1234;
```

В предложении FROM сначала указывается таблица ENGINEERS, которая содержит вложенную таблицу DEGREES и получает псевдоним T1. Затем функция TABLE использует псевдоним T1 для получения полностью квалифицированного имени вложенной таблицы DEGREES. Можно использовать и действительное имя таблицы, такое как ENGINEERS.DEGREES. Функция превращает вложенную таблицу

в "плоскую", создавая по строке для каждой вложенной в основную таблицу строки, что очень похоже на процесс соединения отдельной таблицы DEGREES с основной таблицей. В этом примере предложение SELECT оказывается очень простым — оно просто выбирает один столбец из вложенной таблицы.

Возможность делать таблицы "плоскими" и обрабатывать их так, как если бы они были результатом соединения двух отдельных таблиц, обладает огромным потенциалом. Фактически большинство запросов к таблице с вложенными таблицами может быть непосредственно выражено на SQL без использования хранимых процедур. Однако логика таких запросов и принципы их построения достаточно сложны, что видно даже на таком простом примере.

Работа с коллекциями данных

Для добавления новых строк в таблицы со столбцами, содержащими коллекции, в Informix используются три конструктора: конструктор множества (SET), конструктор мультимножества (MULTISET) и конструктор списка (LIST). Они преобразуют наборы отдельных значений в коллекции соответствующего типа. Вот пара инструкций INSERT, добавляющих строки в таблицы, представленные на рис. 24.6.

```
INSERT INTO TECHNICIANS
     VALUES (1279,
             ROW('Sam', 'R', 'Jones'),
             ROW('164 Elm St.', 'Highland', 'IL',
                 ROW(12345, 6789)),
             SET{'atlas', 'checkmate', 'bingo'});

INSERT INTO ENGINEERS
     VALUES (1281,
             ROW('Jeff', 'R', 'Ames'),
             ROW('1648 Green St.', 'Elgin', 'IL',
                 ROW(12345, 6789)),
             MULTISET{ROW('BS', 'Michigan'),
                      ROW('BS', 'Michigan'),
                      ROW('PhD','Stanford')});
```

Первая из двух инструкций добавляет в таблицу TECHNICIANS одну строку с трехэлементным множеством в столбце PROJECTS. Вторая инструкция добавляет строку в таблицу ENGINEERS с трехэлементным мультимножеством в столбце DEGREES. Поскольку члены этого мультимножества сами являются записями, для их создания используется конструктор записи ROW.

В Oracle выбран другой подход к построению коллекций, предназначенных для вставки в таблицы. Обсуждая абстрактные типы данных Oracle, мы говорили о том, что с каждым АТД автоматически связывается *метод-конструктор*. Эта концепция распространяется также на массивы переменной длины и вложенные таблицы. Для каждого из этих типов данных автоматически создается конструктор, который можно использовать для вставки значений этих типов в таблицы. Вот как следует применять конструкторы в инструкции INSERT.

```
INSERT INTO REPS (EMPL_NUM, NAME, ADDRESS, QTR_TGT)
     VALUES (109,
             NAME_TYPE('Mary', 'X', 'Jones'),
```

```
            ADDR_TYPE('164 Elm St.', 'Highland', 'IL',
                      POST_TYPE(12345, 6789)),
            TGT_ARRAY(5000, 5000, 8000, 12000));

INSERT INTO ENGINEERS (EMPL_NUM, NAME, ADDRESS, DEGREES)
     VALUES (1281,
            NAME_TYPE('Jeff', 'R', 'Ames'),
            ADDR_TYPE('1648 Green St.', 'Elgin','IL',
                      POST_TYPE(12345, 6789)),
            DEGR_TABLE(DEGR_TYPE('BS', 'Michigan'),
                       DEGR_TYPE('BS', 'Michigan'),
                       DEGR_TYPE('PhD', 'Stanford')));
```

Коллекции и хранимые процедуры

Для хранимых процедур, работающих с таблицами, коллекции представляют ряд новых проблем. Для их решения и в Oracle, и в Informix язык хранимых процедур дополнен специальными возможностями. В Informix для работы с коллекциями вводятся специальные переменные коллекций. Вот фрагмент хранимой процедуры, обрабатывающей столбец PROJECTS из таблицы TECHNICIANS.

```
define proj_coll collection;  /* Коллекция проектов  */
define a_project varchar(15); /* Отдельный проект    */
define proj_cnt  integer;     /* Количество проектов */
define empl_name name_type;   /* Буфер для имени     */

/* Выясняем, в скольких проектах участвует технолог  */
select cardinality(projects) into proj_cnt
  from technicians
 where empl_num = 1234;

/* Если проектов слишком много, отказываемся добавить
   еще один                                          */
if (proj_cnt > 6) then ...

/* Извлекаем запись о технологе, содержащую
   коллекцию проектов                                */
select name, projects into empl_name, proj_coll
  from technicians
 where empl_num = 1234;

/* Добавляем в список проектов данного технолога
   проект 'gonzo'                                    */
insert into table(proj_coll)
     values ('gonzo');

/* Просматриваем список проектов                     */
foreach proj_cursor for
 select * into a_project
   from table(proj_coll)
     if (a_project = 'atlas' then
        begin
          update table(proj_coll) (project)
             set project = 'bingo'
           where current of proj_cursor;
          exit foreach;
```

```
        end;
    end if;
end foreach;

/* Обновляем запись в базе данных модифицированным
   списком проектов                                    */
update technicians
   set project = proj_coll
 where empl_num = 1234;
```

Этот пример демонстрирует несколько аспектов обработки коллекций в хранимых процедурах Informix. Прежде всего, коллекция извлекается из базы данных и помещается в переменную специального типа. Для хранения коллекций можно использовать и переменные, явно объявленные как SET (а также MULTISET или LIST). Хранящаяся в переменной коллекция может обрабатываться как обыкновенная таблица. Например, для добавления в коллекцию нового проекта достаточно выполнить инструкцию INSERT, а для поиска и изменения существующего проекта — позиционную инструкцию UPDATE. Обратите внимание на то, что в цикле FOREACH каждый элемент коллекции по очереди извлекается в переменную, чтобы хранимая процедура могла его обработать. Наконец, содержимое переменной может использоваться для обновления столбца, содержащего коллекцию.

В Oracle для обработки массивов переменной длины применяется схожий подход. Отдельные элементы массива абстрактного типа данных доступны через их индексы. Типичная процедура доступа к элементам массива переменной длины в Oracle такова.

1. Извлекаем из таблицы строку, содержащую массив переменной длины, и помещаем ее в локальную переменную, тип которой соответствует структуре строки таблицы или набора извлекаемых из нее столбцов.
2. Выполняем цикл FOR с переменной-счетчиком *n*, значение которой изменяется от единицы до числа, соответствующего количеству элементов массива. Чтобы узнать размер массива, нужно обратиться к специальному атрибуту столбца, в котором он содержится, — COUNT.
3. В цикле FOR для обращения к конкретному элементу массива указывается имя переменной массива и индекс этого элемента.

Аналогичным образом можно обрабатывать и вложенные таблицы, хотя обычно это делается проще. Как правило, вначале таблицу делают "плоской" с помощью SQL-запроса с функцией TABLE, а потом обрабатывают результаты в цикле FOR. Обработка результирующей "плоской" таблицы тоже может быть непростым делом. В частности, хранимая процедура может проверять, поступила ли очередная обрабатываемая ею запись из той же строки главной таблицы, что и предыдущая, и, если да, выполнять некоторую специальную обработку, например подсчет промежуточных итогов. Таким образом, обработка массивов переменной длины и вложенных таблиц в хранимых процедурах Oracle напоминает типичные вложенные циклы в программах формирования отчетов на языке COBOL тридцатилетней давности.

Как видите, обработка коллекций и их отдельных элементов требует довольно сложного программирования — как правило, здесь не обойтись SQL-запросами, пусть даже и с использованием специальных расширений SQL. Именно поэтому объектно-реляционные СУБД часто критикуют за уход от простоты реляционной модели и возврат к явной навигации по базе данных, которая использовалась для доступа к дореляционным базам данных. Приведенные в этой главе примеры показывают, что в этих утверждениях есть по меньшей мере некоторая доля правды.

Пользовательские типы данных

В объектно-реляционных СУБД обычно предусмотрен механизм определения пользовательских типов данных. Например, приложению для работы с географической информацией может потребоваться тип данных LOCATION, состоящий из значений широты и долготы, каждое из которых, в свою очередь, состоит из градусов, минут и секунд. Для эффективной обработки данных этого типа приложению может потребоваться определить специальные функции, например функцию DISTANCE(X,Y), вычисляющую расстояние между двумя географическими точками. Для типа LOCATION должны быть также переопределены некоторые встроенные операции, такие, например, как проверка на равенство (=).

Одним из способов поддержки пользовательских типов данных в Informix Universal Server является ключевое слово OPAQUE. Для СУБД тип данных, созданный как OPAQUE, непрозрачен. Она может хранить и извлекать данные этого типа, но что они собой представляют и как обрабатываются, она не знает. Говоря объектно-ориентированным языком, данные типа OPAQUE полностью инкапсулированы. Пользователь должен явно (во внешних процедурах, написанных на С или другом языке программирования) определить структуру данных для этого типа, описать операции, которые могут над ним выполняться (например, операцию сравнения двух значений), и функции для преобразования этого типа данных между внешним и внутренним представлениями. Таким образом, тип данных OPAQUE реализует низкоуровневые возможности расширения базовых возможностей СУБД за счет дополнительных типов данных, которые выглядят и интерпретируются как встроенные.

Более широкие возможности определения пользовательских типов данных в Informix обеспечивает ключевое слово DISTINCT. Определение типов данных как DISTINCT позволяет сделать их несовместимыми, даже если они созданы на базе одного и того же встроенного типа данных Informix. Например, названия городов и компаний в базе данных логически являются данными разных типов. Названия города и компании нельзя сравнивать между собой, но если определить содержащие их столбцы как VARCHAR(20), СУБД позволит это сделать (однако вряд ли вам надо сравнивать между собой название города и название компании).

Для обеспечения более строгой целостности базы данных можно назначить каждому из этих элементов данных собственный тип, имеющий категорию DISTINCT.

```
CREATE DISTINCT TYPE CITY_TYPE AS VARCHAR(20);
CREATE DISTINCT TYPE CO_NAME_TYPE AS VARCHAR(20);
```

Теперь при создании таблиц можно использовать для столбцов, содержащих названия компаний и городов, типы данных CITY_TYPE и CO_NAME_TYPE. Если попытаться сравнить столбцы этих двух типов, СУБД выдаст сообщение об ошибке. Их можно сравнивать, но только после явного приведения к какому-то одному типу. Таким образом, ключевое слово DISTINCT помогает поддерживать целостность базы данных и предотвращать ошибки в программах и запросах, которые с ней работают.

Хотя Oracle и не поддерживает типы данных DISTINCT, для достижения того же эффекта можно воспользоваться пользовательскими типами данных, состоящими из отдельных столбцов.

```
CREATE TYPE CITY_TYPE    AS OBJECT (COL VARCHAR2(20));
CREATE TYPE CO_NAME_TYPE AS OBJECT (COL VARCHAR2(20));
```

Методы и хранимые процедуры

В объектно-ориентированных языках программирования объекты инкапсулируют данные и код их обработки; детали организации данных и программные инструкции для манипулирования ими скрыты от пользователя. Взаимодействовать с объектом можно только посредством его *методов* — специальных общедоступных подпрограмм, принадлежащих объекту (или, более точно, классу объектов). Например, один метод объекта, представляющего клиента компании, может возвращать предел его кредита. Другой метод может изменять этот предел. Само же значение предела инкапсулировано в объекте — оно скрыто от пользователя, который не знает, в каком виде хранятся данные, и не может получить к ним непосредственный доступ.

Данные в таблице классической реляционной базы данных не инкапсулированы. И данные, и их структура доступны внешним пользователям. Фактически одним из главных преимуществ реляционной модели является именно то, что с помощью языка SQL пользователь может формулировать произвольные запросы к базе данных. А если вспомнить о наличии общедоступного системного каталога реляционной базы данных, то контраст с объектно-ориентированной архитектурой становится еще более очевидным. База данных с системным каталогом предоставляет пользователю или приложению полную информацию о своей структуре, так что даже приложение, не знающее ее структуры заранее, с помощью SQL-запросов может легко определить, что в ней находится и в каком виде.

Можно ли приблизить реляционную архитектуру к объектной в отношении инкапсуляции? Оказывается, да, и довольно простым способом. Хранимые процедуры совместно со схемой безопасности позволяют имитировать инкапсуляцию данных и методов. Теоретически всем пользователям реляционной базы данных можно предоставить лишь разрешения на выполнение ограниченного набора хранимых процедур и никаких разрешений на непосредственный доступ к таблицам. Тогда способ работы пользователей с базой данных будет близок к работе с настоящим объектом: только регламентированные действия и никакого доступа к его данным и коду. Однако на практике разработчикам приложений, как правило, требуется возможность самим писать хранимые процедуры. Кроме того, программам формирования запросов и генераторам отчетов разрешается получать доступ к структуре базы данных.

Oracle формализует соответствие между методами объектов и хранимыми процедурами, позволяя явно определять хранимые процедуры как *функции-члены* АТД. Созданная таким образом функция может использоваться в запросах, обрабатывающих объекты АТД, как если бы она была встроенной функцией СУБД, предназначенной для работы с этим типом данных. Ниже приведено новое определение АТД ADDR_TYPE, используемого для хранения адреса, с относительно простой функцией-членом GET_FULL_POST(). Эта функция получает почтовый индекс, представляющий собой запись с двумя полями (пять цифр главного почтового индекса и четыре — дополнительного), и возвращает этот же индекс в виде одного числа из девяти цифр.

```
CREATE TYPE ADDR_TYPE AS OBJECT (
     STREET VARCHAR(35),
       CITY VARCHAR(15),
      STATE CHAR(2),
   POSTCODE POST_TYPE,
     MEMBER FUNCTION GET_FULL_POST(POSTCODE IN POST_TYPE)
     RETURN NUMBER,
     PRAGMA RESTRICT_REFERENCES(GET_FULL_POST, WNDS));

CREATE TYPE BODY ADDR_TYPE AS
     MEMBER FUNCTION GET_FULL_POST(POSTCODE POST_TYPE)
     RETURN NUMBER IS
      BEGIN
            RETURN((POSTCODE.MAIN * 10000) + POSTCODE.SFX);
      END;
END;
/
```

Функция-член объявляется в инструкции CREATE TYPE, создающей АТД, вслед за строками, описывающими элементы данных. Дополнительное предложение PRAGMA сообщает Oracle, что функция не модифицирует содержимое базы данных; это является обязательным требованием для функций, используемых в запросах на выборку. Отдельная инструкция CREATE TYPE BODY определяет тело функции. После нескольких начальных предложений в ней следует определение функции точно такого же формата, как и в стандартных инструкциях CREATE PROCEDURE и CREATE FUNCTION. Создав функцию-член, ее можно использовать в запросах на выборку, как в следующем примере, возвращающем список служащих, почтовый индекс которых равен 12345-6789.

```
ADDR_TYPE.GET_FULL_POST(12345,6789);
```

Informix Universal Server не имеет, как Oracle, специального механизма для превращения хранимых процедур в объектно-ориентированные методы. Зато он допускает использование значений типа ROW (соответствующих объектным типам данных Oracle) в качестве параметров хранимых функций. При вызове функции ей передается запись соответствующего типа (аналогичного АТД POSTCODE из предыдущего примера для Oracle), и функция может выполнить над ней любые необходимые действия. Например, можно создать хранимую функцию GET_FULL_POST() с единственным параметром типа POST_TYPE. После этого можно будет послать базе данных запрос вида

```
SELECT EMPL_NUM
  FROM PERSONNEL
 WHERE GET_FULL_POST(ADDRESS.POSTCODE) = 123456789;
```

Еще одной мощной возможностью объектно-ориентированного программирования является *перегрузка* методов, т.е. создание нескольких версий одного и того же метода для обработки различных типов данных. В иерархии классов часто требуется определить метод, который выполнял бы одинаковые или очень похожие операции над различными классами. Например, можно создать метод GET_TGT_WAGES(), который будет возвращать суммарную годовую заработную плату для любого объекта класса ENGINEERS и всех его подклассов. Этот метод, реализованный в виде хранимой функции, должен возвращать годовую заработную плату любого служащего, для которого он будет вызван. Выполняемые им вычисления будут зависеть от типа ("класса") конкретного служащего.

- Для технолога суммарная годовая заработная плата состоит из почасовой ставки, умноженной на 52 (число недель в году) и на 40 (количество рабочих часов в неделе).
- Для руководителя суммарная годовая заработная плата состоит из годового оклада и бонуса.
- Для всех остальных инженеров суммарная годовая заработная плата равна их годовому окладу.

Чтобы реализовать этот алгоритм, следует для каждого класса инженеров создать *отдельную* функцию GET_TGT_WAGES(). Каждая функция получает в качестве параметра объект (строку таблицы TECHNICIANS, ENGINEERS или MANAGERS) и возвращает вычисленную сумму. Имена всех функций идентичны, из-за чего их и называют перегруженными, — одно имя на самом деле связано с несколькими хранимыми функциями. При вызове функции СУБД определяет тип переданного ей аргумента (т.е. класс объекта) и решает, какую из реальных функций ей следует выполнить.

В Informix Universal Server возможность перегрузки хранимых процедур обеспечивается без каких-либо дополнительных объектно-ориентированных расширений. Эта СУБД позволяет создавать хранимые процедуры с одинаковыми именами, при условии что у них не будут совпадать количество и типы аргументов. Так, например, можно создать следующие три функции:

```
/* Вычисляет годовую зарплату технолога      */
CREATE FUNCTION GET_TGT_WAGES(PERSON TECH_TYPE)
    RETURNS DECIMAL(9,2)
        AS RETURN (PERSON.WAGE_RATE * 40 * 52)
END FUNCTION;

/* Вычисляет годовую зарплату руководителя  */
CREATE FUNCTION GET_TGT_WAGES(PERSON MGR_TYPE)
    RETURNS DECIMAL(9,2)
        AS RETURN (PERSON.SALARY + PERSON.BONUS)
END FUNCTION;

/* Вычисляет годовую зарплату инженера       */
CREATE FUNCTION GET_TGT_WAGES(PERSON ENGR_TYPE)
    RETURNS DECIMAL(9,2)
        AS RETURN (PERSON.SALARY)
END FUNCTION;
```

После выполнения этих инструкций можно вызвать функцию GET_TGT_WAGES() и передать ей строку из таблицы TECHNICIANS, ENGINEERS или MANAGERS. СУБД автоматически определит, какую из трех функций ей следует выполнить, и вернет правильный результат.

Метод *замещения* процедур в Informix Universal Server делает хранимые процедуры еще более гибким средством работы с типизированными таблицами. Если вы вызовете хранимую процедуру с аргументом типа ROW и передадите ей строку из типизированной таблицы, Informix сначала посмотрит, есть ли в базе данных хранимая процедура с указанным именем и в точности совпадающим типом аргумента. Например, если вызвать хранимую процедуру GET_LNAME() для извлечения фамилии из записи типа TECH_TYPE (вероятно, из таблицы TECHNICIANS), Informix проверит, есть ли в базе данных процедура GET_LNAME() для обработки значения типа TECH_TYPE. Если такой процедуры не окажется, Informix не станет генерировать сообщение об ошибке. Вместо этого она просмотрит иерархию типов в поиске одноименной процедуры, определенной для родительского типа TECH_TYPE. Если обнаружится процедура GET_LNAME() для типа данных ENGR_TYPE, Informix выполнит ее, чтобы получить требуемую информацию. В противном случае поиск будет продолжен вверх по иерархии — может быть, имеется функция GET_LNAME() для типа данных PERS_TYPE. Таким образом, замещение процедур означает, что вы можете определять хранимые процедуры для типа данных, расположенного на самом верхнем уровне иерархии, и эти процедуры будут использоваться для обработки всех его подтипов. Говоря объектно-ориентированным языком, методы класса наследуются всеми его подклассами.

Поддержка объектов в стандарте SQL

Как упоминалось в начале этой главы, наиболее существенным расширением стандарта SQL:1999 была поддержка объектно-реляционных возможностей. Новые инструкции, предложения и выражения были добавлены в спецификацию SQL в следующих областях:

- пользовательские типы данных;
- составные (абстрактные) типы данных;
- массивы;
- перегруженные (полиморфные) хранимые процедуры;
- конструкторы строк и таблиц, поддерживающие абстрактные типы;
- выражения для значений строк и таблиц с поддержкой абстрактных типов.

Объектные расширения стандарта SQL не соответствуют в точности основным коммерческим объектно-реляционным СУБД, но лежащие в их основе концепции те же, что и проиллюстрированные выше для конкретных продуктов. Похоже, что в этой области SQL будет наблюдаться следование стандарту. Постепенно, за несколько выпусков новых версий, основные СУБД начнут, в дополнение к собственному, поддерживать и стандартный синтаксис SQL. Так что в ближайшие годы объектно-реляционные возможности на практике будут представлять собой причудливую смесь стандарта с дополнениями от конкретных производителей.

Резюме

Похоже, что объектно-ориентированные базы данных будут играть все возрастающую роль в специализированных сегментах рынка, таких как инженерное проектирование, обработка сложных составных документов и графические пользовательские интерфейсы. Однако пока они не получили широкого распространения в качестве платформы приложений для обработки деловых данных. Вместо этого на рынок вышли гибридные объектно-реляционные продукты, предлагаемые некоторыми ведущими производителями СУБД.

- В объектно-реляционных СУБД и SQL, и язык хранимых процедур значительно расширены новыми объектно-ориентированными инструкциями и новым синтаксисом.
- Самыми распространенными объектно-реляционными нововведениями являются абстрактные/структурированные типы данных, таблицы в таблицах и явно поддерживаемые дескрипторы объектов. Они значительно расширяют реляционную модель, но усложняют выполнение запросов к базе данных.
- Каждый производитель СУБД предлагает свои объектно-реляционные расширения. Между ними много концептуальных различий, а сходные возможности реализованы зачастую по-разному.
- Объектно-реляционные возможности особенно полезны для сложных моделей данных — они позволяют упростить общую структуру базы данных за счет усложнения отдельных таблиц или объектов.
- Объектно-реляционным расширениям уделяется особое внимание со стороны стандарта SQL, и очевидно, что в будущем они будут включены во многие реляционные СУБД.

25
ГЛАВА

SQL и XML

XML (Extensible Markup Language — расширяемый язык разметки) представляет собой одну из наиболее важных новых технологий, возникшую в результате эволюции Интернета и Веб. XML является стандартным языком для представления *структурированных данных*. В свою очередь, SQL — стандартный язык для определения, доступа и обновления структурированных данных, хранящихся в реляционных базах данных. Наличие связи между XML и SQL представляется совершенно очевидным. Естественно встает вопрос о том, *какова* эта связь и конфликтуют ли эти две технологии или, напротив, дополняют друг друга? Ответ — понемногу и того, и другого. В этой главе вы найдете обзор азов XML и узнаете об эволюции взаимоотношений XML и SQL, а также об интеграции XML в основные SQL-продукты.[1]

Что такое XML

Как следует из имени XML, он представляет собой *язык разметки*. У него много общего с его знаменитым родственником — языком разметки гипертекста (HyperText Markup Language, HTML), который приобрел популярность как базовая веб-технология. *Разметка* документа — методика столь же древняя, как и книгопечатание. При подготовке к печати сложного документа, такого как книга, газета или журнал, приходится думать о двух логически связанных частях. Это *содержимое* документа, которое обычно состоит из текста и графики и определяет его смысл, и *структура* документа (заголовки, подзаголовки, абзацы, подписи), которая вместе с форматированием (шрифты, отступы, схемы страниц) помогает организовать содержимое и представить его читателю в удобном и понятном виде. С самых первых дней книгопечатания редакторы использовали определенные

[1] *При желании более детально ознакомиться с XML обратитесь, например, к книге Д. Хантер, Д. Рафтер и др.* XML. Базовый курс, 4-е издание. — *М.: ООО "И.Д. Вильямс", 2009.* — Примеч. ред.

символы разметки и форматирования, которые определяли структуру и форматирование содержимого документа при печати.

При появлении на сцене компьютеризированных издательских систем команды разметки в содержимом документа превратились в инструкции для издательских программ. У каждого типа издательского программного обеспечения имеется свой набор команд разметки, что затрудняет переносимость документов между системами. Как способ стандартизации языков разметки был разработан и принят в качестве стандарта ISO стандартный обобщенный язык разметки (Standard Generalized Markup Language, SGML). Говоря более строго, SGML представляет собой *метаязык* для определения конкретных языков разметки. Его изобретатели понимали, что ни один язык разметки не в состоянии удовлетворить всем возможным требованиям, но все языки разметки имеют общие элементы. Можно создать семейство тесно связанных языков разметки, стандартизируя эти общие элементы. HTML представляет собой один из таких языков разметки, разработанный специально для использования гипертекста при связывании документов. XML — еще один такой язык, ориентированный на строгую типизацию и четкое структурирование содержимого документа. Общие SGML-корни делают HTML и XML родственными языками и являются причиной их подобия.

И HTML, и XML являются рекомендациями Консорциума W3C (World Wide Web Consortium, W3C), определяемыми спецификациями, которые разработаны, обсуждены и опубликованы W3C. W3C — независимый, некоммерческий консорциум, целью которого является разработка и распространение стандартов для работы в Интернете и Веб. Рекомендации W3C имеют статус "официально одобренных"; это означает, что W3C поддерживает и рекомендует их применение. Благодаря этому HTML и XML являются промышленными стандартами, не зависящими от конкретных производителей.

HTML был первым языком на основе SGML, получившим всемирное распространение. Содержимое огромного количества веб-страниц практически каждого веб-сайта в Интернете представляют собой HTML-документы. В HTML-документе специальные элементы разметки, которые называются дескрипторами, указывают графические элементы (например, такие как кнопки), выводимые веб-браузерами. Дескрипторы также описывают гипертекстовые ссылки на другие документы, которые браузер должен загружать при щелчке на кнопке. Другие дескрипторы идентифицируют графические элементы, которые должны быть вставлены в HTML-текст при его выводе.

Стремительное развитие Веб в 1990-х годах привело к тому, что HTML был быстро адаптирован для вывода гораздо более богатого содержимого форматированных веб-страниц. Вскоре были созданы дескрипторы HTML для управления форматированием веб-страниц, использования курсивного или полужирного шрифта, центрования и отступов и т.п. В некоторых случаях эти дескрипторы были уникальны для конкретного веб-браузера, такого как Netscape или Microsoft Internet Explorer. Со временем основным назначением разметки на HTML-страницах стало форматирование и представление информации. Преимуществом этого подхода стало то, что страницы стали выводиться одинаково на разных уст-

ройствах и разными браузерами, а недостатком — то, что логическая структура содержимого веб-страницы теряется в деталях форматирования и представления.

Важной исходной целью SGML было то, чтобы заданный логический элемент, такой как заголовок страницы или раздела, мог быть единообразно идентифицирован в сотнях документов (например, в сотнях страниц веб-сайта). После этого обеспечить единообразность вывода можно просто директивой наподобие "выводи все заголовки разделов синим полужирным шрифтом Times New Roman 16-го размера". Однако, к сожалению, авторы веб-страниц проявляют тенденцию к явной разметке каждого элемента, такого как заголовок подраздела, с применением детальных инструкций форматирования. Это быстро приводит к несогласованности, и, что еще хуже, изменение форматирования требует редактирования сотен отдельных страниц — вместо одного изменения, которое будет воспринято всеми страницами.

Одной из основных движущих сил развития XML было восстановление подхода к разметке на уровне логики, а не на уровне внешнего представления. XML реализует гораздо более строгие правила, связанные со структурой документа, чем HTML. Большинство его компонентов и возможностей непосредственно связано с представлением логической структуры документа. Имеются сопутствующие стандарты для определения типов документов, такие как XML Schema, которые позволяют еще более усилить эту направленность XML.

Азы XML

Чтобы разобраться в том, как взаимодействуют XML и SQL, необходимы базовые знания о том, что такое XML и как его использовать. Если вы уже знакомы с XML или используете его, то можете смело пропустить этот раздел и переходить к следующему. Если же XML вам незнаком, то в этом разделе вы найдете простое введение в этот язык, основанное на конкретных примерах XML-документов.

На рис. 25.1 показано типичное XML-представление текстового документа, а именно — отрывка второй части оригинального издания данной книги. Здесь вы не найдете примера работы с данными или с SQL, но зато здесь проиллюстрированы ключевые концепции XML. Каждый *элемент* XML-документа представлен соответствующим элементом XML с простой структурой, показанной на рис. 25.2. Элемент идентифицируется *открывающим дескриптором*, который содержит имя типа элемента, заключенное в угловые скобки (<>).

На рис. 25.1 абзацы идентифицируются открывающим дескриптором `<para>`, а заголовки — открывающим дескриптором `<header>`. Конец каждого элемента указывается *закрывающим дескриптором*, который также состоит из имени типа элемента, которому предшествует косая черта (/), и также помещается в угловые скобки. На рис. 25.1 абзацы завершаются дескриптором `</para>`, а заголовки — дескриптором `</header>`. Между открывающим и закрывающим дескрипторами находится *содержимое* элемента. На рис. 25.1 содержимое в основном представляет собой текст. Текст может быть помещен в одинарные или двойные кавычки — лишь бы в начале и конце текста использовались одинаковые кавычки.

```
<?xml version="1.0"?>
<bookPart partNum="2" title="RetrievingData">
  <para>Queries are ... handle complex queries.</para>
  <chapter chapNum="5" revStatus="final">
    <title>SQL Basics</title>
    <para>This chapter ... in this chapter.&quot;</para>
    <section>
      <header hdrLevel="1">Statements</header>
      <para>The main body of...in Figure5-1.</para>
      <para>Every SQL statement...or expressions.</para>
      <figure figNum="5-1"></figure>
      <table tabNum="5-1"></table>
      <para>The ANSI/ISO SQL...InTable 5-3.</para>
      <table tabNum="5-2"></table>
      <table tabNum="5-3"></table>
      <para>Throughout this book...in lowercase.</para>
      <figure figNum="5-2"></figure>
      <para>Variable items...is UNDERLINED.</para>
    </section>
    <section>
      <header hdrLevel="1">Names</header>
      <para>The objects in a...entry forms (Ingres).</para>
      <para>The original...special characters.</para>
      <section>
        <header hdrLevel="2">Table Names</header>
        <para>When you specify...or designer.</para>
        <para>In a larger...table name</para>
              ... etc ...
      </section>
    </section>
  </chapter>
  <chapter chapNum="6">
    <title>Simple Queries</title>
    <para>In many ways...in the database.</para>
    <section>
              ... etc ...
    </section>
  </chapter>
              ... etc ...
</bookPart>
```

Рис. 25.1. Пример XML-документа

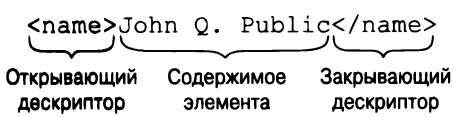


Рис. 25.2. Анатомия XML-элемента

На рис. 25.1 показана иерархия элементов, типичная для большинства XML-документов. На верхнем уровне находится элемент `bookPart`. Его содержимым является не текст, а другие элементы — последовательность элементов `chapter`. Каждый элемент `chapter` содержит элемент `title`, возможно, с некоторыми предваряющими элементами `para`, а затем идет ряд элементов `section`. Каждый элемент `section` содержит элемент `header` и один или несколько элементов `para`, с возможным включением некоторого количества элементов `figure` и `table`. Содержимым каждого элемента `para` может быть только текст.

В дополнение к иерархии элементов, на рис. 25.1 приведены примеры *атрибутов*, еще одной фундаментальной структуры XML. Атрибут связан с конкретным элементом XML и описывает некоторые его характеристики. Каждый атрибут имеет имя и значение. На рис. 25.1 элемент `chapter` имеет атрибут `chapNum`, значением которого является номер главы, связанный с определенным содержимым. Элемент `chapter` имеет другой атрибут — `revStatus`, значение которого указывает, является ли текст черновым вариантом, исправленным или окончательным. Отдельные элементы `<header>` на рис. 25.1 также имеют атрибут `hdrLevel`, указывающий, принадлежит ли данный заголовок к верхнему уровню (уровень 1) или нижнему (уровни 2 или 3).

Первая строка XML-документа на рис. 25.1 указывает, что перед вами документ XML 1.0. Каждая прочая часть документа описывает структуру элемента, его содержимое или атрибуты элементов. XML-документы могут быть и значительно более сложными, но эти фундаментальные компоненты являются важной частью взаимодействия XML с базами данных. Обратите внимание на чувствительность имен элементов и атрибутов к регистру. Элемент с именем `bookPart` и именем `bookpart` — это разные элементы. В этом существенное отличие от соглашения SQL для имен таблиц и столбцов, которые обычно не чувствительны к регистру.

На рис. 25.1 не показана одна дополнительная возможность XML, весьма полезная на практике. Если у элементов нет содержимого, а есть только атрибуты, то конец элемента может быть указан в той же паре угловых скобок, что и его начало, — при помощи косой черты непосредственно перед закрывающей угловой скобкой. При использовании этого соглашения элемент

```
<figure figNum="5-1"></figure>
```

с рис. 25.1 может быть записан как

```
<figure figNum="5-1"/>
```

Спецификация XML определяет ряд правил, которым должны следовать все XML-документы. Она указывает, что элементы в XML-документе должны быть строго вложенными один в другой. Закрывающий дескриптор более низкоуровневого элемента должен располагаться до закрывающего дескриптора элемента более высокого уровня, содержащего указанный низкоуровневый элемент. Стандарт также указывает, что атрибут должен иметь уникальное для данного элемента имя; не допускается наличие у одного элемента двух атрибутов с одним и тем же именем. XML-документы, соответствующие всем описанным правилам, называются *корректными* (well-formed) XML-документами.

XML для данных

Хотя своими корнями XML уходит в документы и их обработку, он вполне применим для представления структурированных данных, с которыми обычно имеют дело приложения для обработки данных. На рис. 25.3 показан типичный XML-документ — очень упрощенный заказ. Он существенно отличается от документа на рис. 25.1, но его ключевые компоненты — те же. Вместо `chapter` элемен-

том верхнего уровня является purchaseOrder. Его содержимым, как и содержимым элемента chapter, являются подэлементы — customerNumber, orderNumber, orderDate и orderItem. Элемент orderItem, в свою очередь, состоит из других подэлементов. На рис. 25.3 показаны также атрибуты элемента terms, которые представляют некоторые бизнес-термины, связанные с заказом. Атрибут ship определяет способ доставки заказа. Атрибут bill связан с кредитом.

```
<?xml version="1.0"?>
<purchaseOrder>
  <customerNumber>2117</customerNumber>
  <orderNumber>112961</orderNumber>
  <orderDate>2007-12-17</orderDate>
  <repNumber>106</repNumber>
  <terms ship="ground" bill="Net30"></terms>
  <orderItem>
    <mfr>REI</mfr>
    <product>2A44L</product>
    <qty>7</qty>
    <amount>31500.00</amount>
  </orderItem>
</purchaseOrder>
```

Рис. 25.3. Простой XML-документ, представляющий заказ

Должно быть очевидно, что простой XML-документ на рис. 25.3 тесно связан с таблицей ORDERS нашей учебной базы данных. Вы можете сравнить его со структурой таблицы ORDERS, показанной в приложении А, "Учебная база данных" (А.5). Элементы нижнего уровня в документе, в основном, соответствуют отдельным столбцам таблицы ORDERS, исключая элемент terms. Элемент верхнего уровня документа представляет строку таблицы. Преобразование группы документов, наподобие показанного на рис. 25.3, во множество строк таблицы ORDERS выполняется достаточно просто, так что оно может быть легко автоматизировано простой компьютерной программой.

В отличие от таблицы ORDERS, XML-документ содержит один промежуточный уровень иерархии, группирующий информацию о заказываемом продукте — идентификатор производителя, идентификатор продукта, количество и общий объем заказа. В реальном заказе указанная группа данных может повторяться несколько раз, образуя в заказе несколько строк. XML-документ можно легко расширить для поддержки этой структуры, добавляя второй или третий элемент orderItem после первого. Учебная база данных так легко расширена быть не может. Для поддержки заказов с несколькими строками таблицу ORDERS, вероятно, следует разбить на две таблицы, в одной из которых хранится информация из заголовка (номер заказа, дата, идентификатор клиента и т.д.), а во второй — отдельные строки заказа.

XML и SQL

Происхождение из SGML дает XML несколько уникальных и полезных характеристик, имеющих аналоги в языке SQL.

- **Описательный подход**. XML говорит о том, чем является каждый элемент документа, а не о том, как его обрабатывать. Вы можете вспомнить, что это же является характеристикой SQL, который сосредоточивает свое внимание на том, какие данные запрошены, а не на том, как их получить.
- **Составные блоки**. Документы XML построены из небольшого количества базовых блоков, включающих две фундаментальные концепции, *элементы* и *атрибуты*. Имеются определенные (хотя и не совершенные) параллели между элементами XML и таблицами SQL и между атрибутами XML и столбцами SQL.
- **Типы документов**. XML определяет и проверяет документы на согласованность с определенными типами, которые соответствуют документам реального мира — таким, например, как заказ или отпускная записка. И здесь вновь имеется параллель с SQL, где таблицы представляют различные сущности реального мира.

Хотя между XML и SQL имеется много сходства, они достаточно сильно отличаются друг от друга.

- **Ориентация на документы или данные**. Базовые концепции XML связаны с типичными структурами документов. XML "текстоцентричен" и разграничивает содержимое (элементы документа) и его характеристики (атрибуты). Базовые концепции SQL происходят из типичных структур записей для обработки данных. SQL ориентирован на данные, обладает широким диапазоном типов данных (в их бинарных представлениях), а его структуры (таблиц и столбцов) ориентированы на содержимое (данные). Несоответствие фундаментальных моделей XML и SQL может привести к определенным конфликтам или сложному выбору при их совместном использовании.
- **Иерархическая и табличная структуры**. Естественной структурой XML является иерархическая, отражающая иерархию элементов в большинстве типов документов. (Например, книга содержит главы, главы состоят из разделов, разделы включают заголовок, абзацы и рисунки.) Эти структуры гибки и изменчивы. Один раздел может содержать пять абзацев и один рисунок, другой — три абзаца и два рисунка, третий — шесть абзацев и ни одного рисунка. Структуры же SQL табличные, а не иерархичные, и отражают типичные для приложений по обработке данных записи. Структуры SQL достаточно жесткие. Каждая строка таблицы содержит одни и те же столбцы, в одном и том же порядке. Каждый столбец в каждой строке содержит данные одного и того же типа. Здесь нет необязательных столбцов — каждый столбец должен присутствовать в каждой строке. Эти отличия могут привести к конфликтам при совместном использовании XML и SQL.
- **Объекты и операции**. Основное предназначение XML — *представление* объектов. Если взять значимую часть XML-кода и спросить "что она представляет", то ответом будет объект — например, абзац, заказ, адрес

клиента. Цели SQL более широкие, но, в основном, они сходятся к *управлению* объектами. Если взять значимую часть SQL-кода и спросить "что она представляет", то ответом, как правило, будет *операция* над объектом — создание, удаление, поиск одного или нескольких объектов или обновление содержимого объекта. Эти отличия делают два указанных языка дополняющими друг друга в их предназначении и использовании.

Элементы и атрибуты

Реляционная модель предлагает только один способ представления значений данных в базе данных — как значений отдельных столбцов в отдельных строках таблицы. Модель XML-документа предлагает два способа представления данных.

- **Элементы**. Элемент XML-документа имеет содержимое, и это содержимое может включать значения данных в форме текста этого элемента. При таком представлении значения данных являются фундаментальной частью иерархии XML-документа; иерархия строится из элементов. Кстати, иерархическая древовидная структура приводит к тому, что практики, говоря о коллекции связанных элементов XML-документов, называют ее *лесом*. Часто элемент, содержащий значение данных, является листом в дереве XML-документа; такой элемент является дочерним по отношению к элементу более высокого уровня, но сам дочерних элементов не имеет. Это почти всегда так для элементов, представляющих данные из реляционной базы данных. Однако XML поддерживает *смешанные* элементы, которые содержат объединение текста (содержимого) и других подэлементов.
- **Атрибуты**. Элемент в XML-документе может иметь один или несколько именованных атрибутов, а каждый атрибут имеет текстовое значение. Атрибуты присоединяются к элементу в XML-иерархии, но не к содержимому элемента. Имена различных атрибутов элемента должны быть различными, так что двух атрибутов с одним именем быть не может. Кроме того, XML рассматривает порядок атрибутов элемента как не имеющий значения; они могут появляться в любом порядке. Это отличается от рассмотрения элементов в XML, которые имеют определенную позицию в документе и где различия между первым, вторым и третьим дочерними элементами элемента более высокого уровня существенны.

Наличие двух различных способов представления данных в XML означает, что у вас есть два разных корректных способа для выражения содержимого реляционной базы данных в виде XML. Вот две строки данных

```
ORDER_NUM MFR PRODUCT QTY AMOUNT
--------- --- ------- ---- ----------
   112963 ACI   41004   28  $3,276.00
   112983 ACI   41004    3    $702.00
```

которые могут быть представлены в виде следующего XML-документа, в котором для представления значений столбцов использованы элементы.

```
<?xml version="1.0"?>
<queryResults>
  <row>
    <orderNum>112963</orderNum>
    <mfr>ACI</mfr>
    <product>41004</product>
    <qty>28</qty>
    <amount>3276.00</amount>
  </row>
  <row>
    <orderNum>112983</orderNum>
    <mfr>ACI</mfr>
    <product>41004</product>
    <qty>3</qty>
    <amount>702.00</amount>
  </row>
</queryResults>
```

А вот представление тех же данных XML-документом с применением атрибутов.

```
<?xml version="1.0"?>
<queryResults>
  <row orderNum="112963"
       mfr="ACI"
       product="41004"
       qty="28"
       amount="3276.00">
  </row>
  <row orderNum="112983"
       mfr="ACI"
       product="41004"
       qty="3"
       amount="702.00">
  </row>
</queryResults>
```

Как и следовало ожидать, у каждого из методов имеются свои приверженцы со своей аргументацией. Сторонники подхода с применением элементов приводят следующие аргументы.

- Элементы представляют собой более фундаментальные объекты модели XML по сравнению с атрибутами; они выполняют роль носителей содержимого во всех языках разметки (HTML, XML, SGML и т.д.), а содержимое базы данных (значения столбцов) должно быть представлено в XML в качестве содержимого.
- Порядок элементов имеет значение, а в некоторых случаях имеет значение и порядок данных в СУБД (например, при идентификации столбца по номеру в запросе или при использовании номера столбца для получения результатов запроса с использованием API).
- Элементы обеспечивают единообразный способ представления данных столбцов, независимо от того, имеет ли он простой, атомарный тип данных (целое число, строка) или более сложные, составные, пользовательские типы данных, поддерживаемые объектно-реляционными расширениями стандарта SQL. Атрибуты такую возможность не предоставляют. (Значения атрибутов атомарны.)

Сторонники подхода с применением атрибутов приводят следующие аргументы.

- Атрибуты представляют собой фундаментальное соответствие столбцам реляционной модели. Отдельные строки представляют сущности, так что они должны отображаться на элементы. Значения столбцов описывают атрибуты сущности (строк*и*), в которой они находятся; соответственно, в XML они должны быть представлены значениями атрибутов.
- Ограничение на уникальность имен атрибутов в пределах элемента соответствует требованию уникальности имен столбцов в пределах таблицы. Неупорядоченная природа атрибутов соответствует неупорядоченной природе столбцов фундаментальной реляционной модели. (Позиции столбцов используются только как сокращения, для удобства, но не являются важными для отношений между столбцами.)
- Представление с применением атрибутов более компактное, поскольку имена столбцов появляются в XML только один раз, а не два раза — в открывающем и закрывающем дескрипторах. Это практическое преимущество при хранении и передаче XML.

В сегодняшних продуктах XML и SQL можно обнаружить оба подхода. Конкретный выбор зависит от предпочтений автора документа и удобства организации использования XML с SQL. Кроме того, диктовать применение того или иного стиля могут промышленные стандарты обмена информацией с использованием XML.

Использование XML с базами данных

Быстро растущая популярность XML заставляет производителей баз данных внедрять поддержку XML в своих продуктах. Виды этой поддержки различны, но обычно попадают в одну из приведенных ниже категорий.

- **Вывод XML**. XML-документ может легко представить данные в одной или нескольких строках результатов запроса. При такой поддержке СУБД в ответ на SQL-запрос генерирует XML-документ вместо обычных результатов запроса в виде обычных строк/столбцов. Стандарт SQL определяет ряд SQL-функций, которые могут использоваться для преобразования данных, полученных из реляционных таблиц, в XML.
- **Ввод XML**. XML-документ может легко представить данные, вставляемые в одну или несколько новых строк таблицы. Он может также представлять данные для обновления строки таблицы или идентификации удаляемой строки. При такой поддержке СУБД принимает XML-документ в качестве входных данных вместо запроса SQL.
- **Обмен данными XML**. XML представляет собой естественный способ выражения данных, которыми обмениваются СУБД или серверы баз данных. Данные из исходной базы данных трансформируются в XML-документ и передаются целевой базе данных, которая преобразует их в свой формат. Тот же стиль обмена данными полезен для перемещения

данных между реляционными базами данных и приложениями, не являющимися СУБД, такими как программы учета ресурсов предприятия, интеграции приложений предприятия и т.п.

- **Хранение XML**. Реляционная база данных может легко принимать XML-документы (которые представляют собой строки текстовых символов) как части строк переменной длины (VARCHAR) или символьных больших объектов (CLOB). На базовом уровне поддержки XML весь документ становится содержимым одного столбца в одной строке базы данных. Немного более мощная поддержка XML возможна, если СУБД позволяет объявлять столбец с использованием явного типа данных XML. Хотя стандарт ANSI/ISO SQL содержит спецификацию типа данных XML (XML), пока что его не реализует ни один производитель. Однако Oracle, DB2 UDB и SQL Server поддерживают собственные XML-типы в своих частных реализациях.
- **Интеграция данных XML**. Более серьезный уровень интеграции XML возможен в том случае, когда СУБД в состоянии анализировать XML-документ, разбирать его на составные элементы и сохранять отдельные элементы в отдельных столбцах. После этого можно использовать обычный SQL для поиска информации в столбцах, тем самым выполняя поиск в XML-документе. В ответ на запрос СУБД может генерировать XML-документ из сохраненных составных элементов.

Вывод XML

Одна из наиболее простых комбинаций XML и баз данных заключается в применении XML в качестве формата для вывода результатов SQL-запроса. Результаты запроса имеют структурированный табличный формат, который легко транслировать в XML-представление. Рассмотрим простой запрос к учебной базе данных.

```
SELECT ORDER_NUM, MFR, PRODUCT, QTY, AMOUNT
  FROM ORDERS
 WHERE CUST = 2103;

ORDER_NUM MFR PRODUCT  QTY     AMOUNT
--------- --- ------- ---- ----------
   112963 ACI   41004   28  $3,276.00
   112983 ACI   41004    3    $702.00
   113027 ACI   41002   54  $4,104.00
   112987 ACI   4100Y   11 $27,500.00
```

Если запросить у СУБД вывод результатов запроса в формате XML, то вывод может иметь следующий вид.

```
SELECT ORDER_NUM, MFR, PRODUCT, QTY, AMOUNT
  FROM ORDERS
 WHERE CUST = 2103;

<?xml version="1.0"?>
<queryResults>
    <row>
        <order_num>112963</order_num>
        <mfr>ACI</mfr>
```

```
        <product>41004</product>
        <qty>28</qty>
        <amount>3276.00</amount>
    </row>
    <row>
        <order_num>112983</order_num>
        <mfr>ACI</mfr>
        <product>41004</product>
        <qty>3</qty>
        <amount>702.00</amount>
    </row>
    <row>
        <order_num>113027</order_num>
        <mfr>ACI</mfr>
        <product>41002</product>
        <qty>54</qty>
        <amount>4104.00</amount>
    </row>
    <row>
        <order_num>112987</order_num>
        <mfr>ACI</mfr>
        <product>4100Y</product>
        <qty>11</qty>
        <amount>27500.00</amount>
    </row>
</queryResults>
```

Это типичный результат для некоторых популярных СУБД с поддержкой вывода в формате XML. Результаты запроса представляют собой корректный самодостаточный XML-документ. Если передать полученные результаты анализатору XML (они описываются позже в данной главе), то он интерпретирует их как содержащие

- один корневой элемент — `queryResults`;
- четыре подэлемента `row`, являющихся дочерними по отношению к корневому;
- по пять подэлементов у каждого элемента `row`, и в этом случае у каждого элемента `row` имеются все пять подэлементов, в одном и том же порядке.

Предыдущий пример XML-документа, генерируемого непосредственно из базы данных, идеален. На самом же деле поддержка XML у разных реализаций существенно отличается. Например, Microsoft участвовала в разработке спецификации SQL/XML в стандарте ANSI/ISO SQL; однако позже ею было принято решение не реализовывать стандарт, а разработать собственное решение. В SQL Server для вывода результатов в формате XML используется предложение `FOR XML`. Хотя генерируемые результаты не включают корректный XML-документ, они представляют собой корректные XML-элементы, которые легко инкорпорируются в XML-документы. Вот как работает предыдущий пример в SQL Server.

```
SELECT ORDER_NUM, MFR, PRODUCT, QTY, AMOUNT
  FROM ORDERS
 WHERE CUST = 2103
   FOR XML AUTO;
```

```
<ORDERS ORDER_NUM="112963" MFR="ACI"
        PRODUCT="41004"    QTY="28"
        AMOUNT="3276.0000"  />
<ORDERS ORDER_NUM="112983" MFR="ACI"
        PRODUCT="41004"    QTY="6"
        AMOUNT="702.0000"   />
<ORDERS ORDER_NUM="112987" MFR="ACI"
        PRODUCT="4100Y"    QTY="11"
        AMOUNT="27500.0000" />
<ORDERS ORDER_NUM="113027" MFR="ACI"
        PRODUCT="41002"    QTY="54"
        AMOUNT="4104.0000"  />
```

Возможность получения результата в формате XML может быть значительным преимуществом. Для выполнения дальнейшей обработки полученный вывод можно непосредственно передать в программу, принимающую в качестве входной информации XML-документы. Вывод может быть передан по сети другой системе, а в силу XML-формата его элементы самоописываемы, т.е. *каждая* система или приложение будут интерпретировать результаты запроса одинаково — как четыре строки с пятью элементами каждая. Поскольку вывод представляет собой чисто текстовый формат, он не может быть некорректно интерпретирован из-за отличий в представлении бинарных данных в системе-отправителе и системе-получателе. Наконец, если XML передается при помощи протокола HTTP с использованием стандарта SOAP (Simple Object Access Protocol — простой протокол обращения к объекту), то XML-сообщение может легко проходить через корпоративные брандмауэры и связывать отправляющее приложение в одной компании с приложением-получателем в другой компании.

Вывод в формате XML не лишен и недостатков. Одним из них является большой объем выводимых данных, который превышает объем данных в табличном виде примерно раза в четыре. Если этот вывод должен быть сохранен на диск, потребуется в четыре раза больше дискового пространства. При пересылке в другую систему потребуется в четыре раза больше времени либо, чтобы время осталось неизменным, повышение пропускной способности сети в четыре раза. Это не очень серьезные проблемы для небольших данных, как в рассмотренных примерах, но в случае результатов из тысяч и десятков тысяч строк они могут стать очень существенными, особенно в случае крупных предприятий, когда эти результаты надо будет умножить на сотни работающих приложений.

Простой XML-вывод также теряет некоторую информацию о данных. Символ доллара в случае XML-вывода исчезает, так что из самого содержимого невозможно определить, представляют ли данные денежные суммы и, если да, в какой именно валюте. Возможности XML Schema предоставляют способ получения этой информации, но только ценой еще большего увеличения размера результатов запроса.

Рассмотрим также вопрос стандартизации определения данных. Сам по себе XML является стандартом, но две компании не смогут работать с заказами друг друга в XML-формате до тех пор, пока дескрипторы в соответствующих XML-документах не будут иметь одни и те же имена и определения. Например, если первая компания использует для идентификатора заказываемого товара дескриптор `<productcode>`, а вторая — `<SKU>` (от stock keeping unit), то эти заказы не мо-

гут быть обработаны одним и тем же способом. Одним из решений может служить появление в области XML-документации специализированных стандартов кодирования, таких как HR-XML, разработанный HR-XML Consortium для кадровых служб. Такие стандарты представляют собой аналог стандартов корпоративного обмена данными (Enterprise Data Interchange, EDI), разработанных в последние пару десятилетий.

Функции SQL/XML

Стандарт SQL определяет ряд функций, которые могут использоваться для преобразования данных столбца в XML-элементы. Функция SQL/XML представляет собой просто функцию, которая возвращает значение в XML-виде. Например, запрос может выбирать не-XML-данные (т.е. данные отличных от XML типов), но затем можно форматировать результаты запроса в виде XML-документа для вывода на веб-странице или для передачи другому приложению. Основные функции SQL/XML показаны в табл. 25.1.

Таблица 25.1. Функции SQL/XML

Функция	Возвращаемое значение
XMLAGG	Единое XML-значение, содержащее XML-лес, образованный коллекцией строк, каждая из которых содержит единственное XML-значение
XMLATTRIBUTE	Атрибут в виде name=value в XMLELEMENT
XMLCOMMENT	Комментарий XML
XMLCONCAT	Связанный список XML-значений, создающий единое значение, содержащее XML-лес
XMLDOCUMENT	XML-значение, содержащее единый узел документа
XMLELEMENT	XML-элемент, который может быть дочерним по отношению к узлу документа, с именем, переданным в качестве параметра
XMLFOREST	XML-элемент, содержащий последовательность XML-элементов, полученную из столбцов таблицы, с использованием в качестве имен элементов имена соответствующих столбцов
XMLPARSE	XML-значение, образованное путем анализа переданной строки без проверки ее корректности
XMLPI	XML-значение, содержащее команду обработки XML
XMLQUERY	Результат выражения XQuery (XQuery — подъязык, используемый для выполнения поиска в тексте XML, хранящемся в базе данных; будет рассмотрен позже в данной главе)
XMLTEXT	XML-значение, содержащее единственный текстовый узел XML, который может быть дочерним по отношению к узлу документа
XMLVALIDATE	XML-последовательность, являющаяся результатом проверки корректности XML-значения

Имеются и другие функции помимо перечисленных в табл. 25.1. Функции SQL/XML могут комбинироваться для создания очень сложных запросов. Доступность конкретных функций варьируется в зависимости от конкретной реализации SQL. Вот простые примеры, которые должны пояснить применение функций XMLELEMENT и XMLFOREST на практике.

```
SELECT XMLELEMENT("OrderNumber", ORDER_NUM)
  FROM ORDERS
 WHERE ORDER_NUM=112963;

<OrderNumber>112963</OrderNumber>


SELECT XMLFOREST(ORDER_NUM AS "OrderNumber",
                 MFR, PRODUCT, QTY, AMOUNT)
  FROM ORDERS
 WHERE ORDER_NUM=112963;

<OrderNumber>112963</OrderNumber>
<MFR>ACI</MFR>
<PRODUCT>41004</PRODUCT>
<QTY>28</QTY>
<AMOUNT>3276</AMOUNT>
```

Обратите внимание на то, что имена XML-элементов берутся из имен столбцов и переводятся в верхний регистр, как это принято в SQL. Однако, используя псевдонимы столбцов, как это было сделано для столбца ORDER_NUM, можно изменить имена столбцов на другие, которые вам нужны.

Ввод XML

Так же как XML может применяться для представления строки результатов запроса, являющейся выводом базы данных, его можно применять и для представления строки данных, вносимой в базу данных. Для обработки XML-данных СУБД должна проанализировать XML-документ, содержащий вносимые данные, и идентифицировать отдельные элементы данных (представленные элементами или атрибутами). СУБД должна сопоставить имена элементов или атрибутов (обычно с применением имен столбцов) или преобразовать их (с применением схемы конкретной СУБД) в имена столбцов целевой таблицы, которая будет получать новые данные. Концептуально простая инструкция INSERT

```
INSERT INTO OFFICES (OFFICE, CITY, REGION, SALES)
     VALUES (23,'San Francisco','Western',0.00);
```

может быть легко транслирована в эквивалентную гибридную SQL/XML-инструкцию наподобие следующей.

```
INSERT WITH <?xml version="1.0"?>
       INTO OFFICES (OFFICE, CITY, REGION, SALES)
     VALUES <row>
              <office>23</office>
              <city>San Francisco</city>
              <region>Western</region>
              <sales>0.00</sales>
            </row>
```

Обновление базы данных обрабатывается аналогично. Простая инструкция UPDATE

```
UPDATE OFFICES
   SET TARGET = 200000.00,
```

```
      MGR    = 108
 WHERE OFFICE = 23;
```

может быть легко транслирована в эквивалентную гибридную SQL/XML-инструкцию наподобие следующей.

```
UPDATE WITH <?xml version="1.0"?> OFFICES
 WHERE OFFICE = 23
                <update_info>
                  <values>
                    <target>200000.00</target>
                    <mgr>108</mgr>
                  </values>
                  <where>office = 23</where>
                </update_info>
```

Инструкция DELETE требует только наличия предложения WHERE, с применением указанных выше соглашений.

Хотя некоторые производители SQL СУБД и добавили возможность обработки XML-операций INSERT, UPDATE и DELETE с использованием указанного подхода, методы представления имен таблиц и столбцов, а также значений данных в XML-тексте, как и отображение их на соответствующие структуры базы данных, остаются зависящими от конкретной СУБД. Хотя стандарт ANSI/ISO SQL и включает спецификацию применения инструкций INSERT, UPDATE и DELETE и столбцов с типом данных XML, а также предложения WITH, которое поддерживает использование XML в инструкциях SQL, стандарт для гибридного SQL/XML (пока что) отсутствует.

Хотя представление входных и обновляемых значений в виде небольших XML-документов концептуально простое и понятное, при этом встают некоторые важные вопросы функционирования СУБД. Например, список столбцов в инструкции SQL INSERT является избыточным, если XML-документ, содержащий вставляемые значения данных, включает имена столбцов как имена элементов или атрибутов. Почему бы просто не опустить список столбцов и не позволить XML-документам определять, какие столбцы должны быть вставлены? В случае интерактивного SQL это не составляет проблемы, но вряд ли формат XML будет использоваться в интерактивном режиме. В случае программного SQL проблема заключается в том, что XML-документ и содержащиеся в нем значения данных передаются СУБД во время работы программы. Если имена столбцов (или даже имя таблицы) также передаются только в XML-документе, то СУБД до момента выполнения не знает, какие таблицы и столбцы будут задействованы. В этой ситуации СУБД вынуждена использовать динамический SQL, описанный в главе 18, "Динамический SQL", со всеми его накладными расходами и сниженной производительностью.

Подобные проблемы возникают и у предложения WHERE в инструкциях UPDATE или DELETE, а также предложения SET инструкции UPDATE. Чтобы получить производительность и эффективность статического SQL, СУБД должна знать заранее (во время компиляции), какие будут задействованы условия отбора и какие столбцы будут обновляться. Один подход к этой проблеме заключается в применении параметризованного вида этих инструкций. Вот все тот же пример UPDATE, использующий упомянутый подход.

```
UPDATE WITH <?xml version="1.0"?> OFFICES
   SET TARGET = ?, MGR = ?
 WHERE OFFICE = ?
       <update_info>
         <param>200000.00</param>
         <param>108</param>
         <param>23</param>
       </update_info>
```

В этом случае XML-текст и SQL-текст оказываются разделены. SQL-текст самодостаточен и может быть обработан в процессе компиляции. XML-текст также самодостаточен, и СУБД может сопоставить его значения параметров с параметрами, требующимися SQL-инструкции, во время работы. Этот пример следует обычному стилю SQL указания параметров по их позициям, но XML-документ в результате теряет свое свойство самоописания. В зависимости от конкретной СУБД, возможно использование в XML-документе именованных элементов и сопоставление их с именованными параметрами инструкции во время работы программы.

Обмен XML-данными

В простейшем виде СУБД может поддерживать обмен XML-данными, просто выводя результаты запроса в виде XML-документа и передавая его в качестве данных операции INSERT. Однако при этом от пользователя или программиста требуется аккуратная разработка формата генерируемых результатов запроса исходной базой данных, чтобы он соответствовал ожидаемому инструкцией INSERT в целевой базе данных. Обмен XML-данными более полезен, если СУБД обладает явной его поддержкой.

Некоторые коммерческие СУБД предлагают возможность выполнить быстрый экспорт таблицы (или, в более сложном случае, результатов запроса) во внешний файл, форматированный как XML-документ. Ей сопутствует возможность быстрого импорта из такого файла в таблицу СУБД. В такой схеме файл XML-документа становится стандартным способом представления содержимого таблицы для обмена информацией.

Заметим, что при наличии такой возможности импорта/экспорта таблицы с использованием XML ее применение не ограничивается обменом информацией между базами данных. Источником XML-документа может быть и некоторое приложение уровня предприятия, наподобие системы управления поставками (Supply Chain Management, SCM). Точно так же приложение может быть и приемником такого файла. Кроме того, многие приложения уровня предприятия в настоящее время поддерживают работу с XML-документами. Они могут выполнять дальнейшую обработку и интеграцию данных, например устранение дублей или слияние нескольких входных файлов.

Хранение и интеграция XML-данных

Ввод, вывод и обмен XML-данными предлагают очень эффективный способ интеграции существующих реляционных баз данных с растущим миром XML. При таком подходе XML используется во внешнем мире для представления структуриро-

ванных данных, но в самой базе данных данные остаются в виде табличной, бинарной, структуры строк и столбцов. С распространением XML-документов естественным следующим шагом станет хранение в базе данных самих XML-документов.

Простое хранение XML в виде больших объектов

Любая SQL СУБД, которая поддерживает большие объекты, автоматически включает базовую поддержку хранения и выборки XML-документов. Раздел главы 24, "SQL и объекты", посвященный большим объектам, описывает, как некоторые коммерческие базы данных хранят и извлекают большие текстовые документы с применением типов данных больших объектов (CLOB). Многие коммерческие продукты поддерживают хранение в виде данных CLOB документов размером до 4 Гбайт, чего достаточно для подавляющего большинства XML-документов. Как уже упоминалось, Oracle, DB2 UDB и SQL Server поддерживают собственные типы данных XML, являющиеся альтернативным способом хранения данных XML.

Чтобы хранить XML-документы с использованием типа данных CLOB, обычно надо определить таблицу, которая содержит один столбец CLOB для текста документа и несколько вспомогательных столбцов (с данными стандартных типов), в которых содержатся атрибуты, идентифицирующие документ. Например, если таблица предназначена для хранения заказов, то можно определить вспомогательные столбцы для хранения идентификатора покупателя, даты заказа и его номера (с использованием типов данных INTEGER, VARCHAR или DATE), в дополнение к столбцу CLOB для хранения самого XML-документа. При этом в таблице можно выполнять поиск заказов на основе идентификаторов клиентов, дат заказов или их номеров и использовать методы работы с данными CLOB (описанными в главе 24, "SQL и объекты") для выборки или сохранения XML-документа.

Преимуществом такого подхода является относительная простота его реализации. Кроме того, при этом происходит четкое разделение операций SQL (таких, как обработка запросов) и операций XML. Недостаток же заключается в слабом уровне интеграции XML/СУБД. В простейших реализациях хранимый XML-документ совершенно непрозрачен для СУБД, которая ничего не знает о его содержимом. Вы не можете выполнить эффективный поиск документа на основе значений одного из его атрибутов или элементов, если только этот атрибут или элемент не извлечен из документа и не представлен отдельным столбцом таблицы. Но если вы заранее знаете, какие типы поиска будут выполняться, то это не такое уж сильное ограничение.

Некоторые объектно-реляционные базы данных предоставляют более усовершенствованные возможности поиска среди CLOB, расширяя предложение WHERE возможностью полнотекстового поиска. Эти продукты позволяют выполнять поиск в столбцах CLOB как в тексте, с использованием возможностей поиска, обычно предоставляемых текстовыми редакторами. Таким образом, вы получаете, хотя и ограниченную, возможность поиска XML-документов, хранящихся в столбцах CLOB. При таком поиске вы можете, например, обнаружить все заказы, в которых имеется фраза "Type 4 Widgets". Однако будет трудно или попросту невозможно выполнить поиск тех XML-документов, в которых "Type 4 Widgets" находится в конкретном элементе. Поскольку программное обеспечение поиска ничего не знает о структуре XML-документов, оно вернет и те документы, в которых подстрока "Type 4 Widgets" находится в комментарии или ином элементе.

Большие объекты и анализаторы

При обмене между базами данных или сохранении в файле или столбце СУБД CLOB, XML-документы всегда имеют текстовый вид. Это делает содержимое высокопереносимым, но затрудняет его обработку компьютерными программами. XML-*анализатор* представляет собой часть программного обеспечения, которая преобразует XML-документы из текстового вида в более удобное для программы внутреннее представление. Любая SQL СУБД с поддержкой XML включает XML-анализатор в качестве части своего программного обеспечения, для собственной обработки XML-данных. Если СУБД поддерживает CLOB, то дальнейшая интеграция с XML связана с применением XML-анализатора для работы с содержимым CLOB-столбцов.

Есть два популярных типа XML-анализаторов, которые поддерживают два стиля обработки.

- **Объектная модель документа**. Анализаторы, использующие объектную модель документа (Document Object Model, DOM), преобразуют XML-документ в иерархическую древовидную структуру в оперативной памяти компьютера. Затем программа может осуществлять вызовы DOM API для обхода дерева, перемещения вверх и вниз по иерархии элементов. DOM API делает структуру элементов XML-документа легко доступной для программистов и упрощает произвольный доступ к частям документа.
- **Простой API для XML**. Анализаторы на основе простого API для XML (Simple API for XML, SAX) преобразуют XML-документ в ряд *обратных вызовов* программы, которые информируют программу о каждой встречающейся части XML-документа. Программа может быть структурирована таким образом, чтобы выполнять определенные действия, когда встречается начало нового раздела документа или определенный атрибут. SAX API навязывает последовательный стиль обработки. API с применением обратных вызовов хорошо соответствует структуре программы, управляемой событиями.

Анализатор любого типа проверяет корректность XML-документа, а также может проверить его соответствие схеме (о схемах будет рассказано немного позже). DOM-анализатор практичен для достаточно небольших XML-документов; он требует удвоенного количества памяти по сравнению с документом, поскольку строит в памяти второе, древовидное, представление всего документа. В случае очень больших документов SAX-анализатор облегчает обработку документа, выполняя ее небольшими дискретными частями. Однако тот факт, что весь документ одновременно недоступен, при необходимости обработки разных частей документа не в последовательном порядке может потребовать от программы несколько проходов по нему.

XML-маршалинг

Хранение XML-документов в виде больших объектов базы данных для некоторых типов интеграции SQL/XML является превосходным решением. Если XML-документы представляют собой, например, текстовые бизнес-документы или они являются текстовыми компонентами веб-страниц, то СУБД требуется очень не-

многое для "понимания" содержимого таких документов. Вероятно, каждый документ можно идентифицировать одним или несколькими ключевыми словами или атрибутами, которые легко выделить и сохранить как обычные столбцы для обеспечения поиска.

Если обрабатываемые XML-документы в действительности представляют собой обрабатываемые записи с данными, то простая интеграция с применением больших объектов может быть слишком примитивной. По всей вероятности, в этом случае потребуется доступ и обработка отдельных элементов, а также поиск на основе их содержимого и атрибутов. СУБД предоставляет такие возможности для своих "родных" данных в строках и столбцах. Почему бы в таком случае СУБД не выполнять разложение входного XML-документа, преобразование содержимого его элементов и значений атрибутов в соответствующее множество строк и столбцов внутренних данных для последующей обработки? Что касается работы с внешними программами, то мы уже видели, как такой подход может работать при преобразовании табличных результатов запроса в XML-документ. Тот же метод можно использовать и для восстановления XML-документа, если он вновь потребуется в текстовом виде.

Преобразование XML-документов, которые представляют собой отличное представление данных для внешних приложений, во внутреннее представление данных и из него, полезно не только при работе баз данных. Та же проблема возникает, например, при обработке XML в Java, когда крайне желательно преобразовать XML-документ в набор экземпляров Java-классов для внутренней обработки и обратно в XML-документ. Процесс декомпозиции XML-документа на элементы и атрибуты в некотором внутреннем, бинарном представлении называется в литературе, посвященной XML, *демаршалингом* (unmarshaling). И наоборот, процесс сборки представления в виде набора отдельных элементов и атрибутов в полный текстовый XML-документ называется *маршалингом* (marshaling).

В случае простых XML-документов маршалинг и демаршалинг также просты, и коммерческие СУБД начинают их поддерживать. Рассмотрим еще раз простой заказ, представленный на рис. 25.3. Его элементы взаимно однозначно отображаются на столбцы таблицы ORDERS. В простейшем случае имена элементов (или атрибутов) идентичны именам соответствующих столбцов. СУБД может получить входной XML-документ наподобие показанного и автоматически превратить его элементы (или атрибуты, в зависимости от используемого стиля документа) в значения столбцов, используя имена элементов (или атрибутов) для управления процессом преобразования. Восстановление XML-документа из строк таблицы также не представляет никакой проблемы.

СУБД потребуется выполнить немного больше работы, если имена элементов в XML-документе не совпадают с именами столбцов. В этом случае требуется указать некоторое отображение между именами элементов (или атрибутов) и именами столбцов. Такое отображение несложно разместить в системном каталоге СУБД.

Многие реальные XML-документы не отображаются в одну строку таблицы. На рис. 25.4 показано простое расширение XML-документа заказа с рис. 25.3, в котором поддерживается типичная для реальных заказов возможность включения в один заказ разных товаров. Как должен выполняться демаршалинг такого XML-документа в учебную базу данных? Одно из решений состоит в том, чтобы сделать

каждую строку заказа отдельной строкой таблицы ORDERS (проигнорируем пока что тот факт, что каждая строка таблицы ORDERS должна содержать уникальный номер заказа, который является первичным ключом). В результате получится определенное дублирование данных, поскольку в нескольких строках будут одни и те же номер и дата заказа, идентификатор клиента и идентификатор продавца. Кроме того, это решение затруднит маршалинг данных — СУБД должна знать, что все строки с одним и тем же номером заказа следует собрать в один XML-документ с несколькими строками заказов. Понятно, что маршалинг/демаршалинг даже такого простого документа требует более сложного отображения.

```
<?xml version="1.0"?>
<purchaseOrder>
    <customerNumber>2117</customerNumber>
    <orderNumber>112961</orderNumber>
    <orderDate>2007-12-17</orderDate>
    <repNumber>106</repNumber>
    <terms ship="ground" bill="Net30"></terms>
    <orderItem>
        <mfr>REI</mfr>
        <product>2A44L</product>
        <qty>7</qty>
        <amount>31500.00</amount>
    </orderItem>
    <orderItem>
        <mfr>ACI</mfr>
        <product>41003</product>
        <qty>10</qty>
        <amount>6520.00</amount>
    </orderItem>
</purchaseOrder>
```

Рис. 25.4. Немного расширенный XML-заказ

Многострочный заказ всего лишь слегка усложняет маршалинг и демаршалинг XML-документов. Более общая ситуация показана на рис. 25.5, где СУБД должна выполнить демаршалинг XML-документа в несколько строк нескольких несвязанных таблиц. Для маршалинга документа СУБД должна изучить взаимоотношения между таблицами, чтобы найти связанные строки и восстановить XML-иерархию. Причина такой сложности в несоответствии между естественной иерархической структурой XML и плоской, нормализованной табличной структурой реляционной базы данных.

Маршалинг и демаршалинг одновременно и упрощаются, и усложняются при поддержке базой данных объектно-реляционных расширений, таких как структурированные типы данных. Преобразование в и из XML может быть проще благодаря тому, что отдельные столбцы таблицы теперь могут иметь собственную иерархическую структуру. Высокоуровневый элемент XML (такой, как адрес, состоящий из улицы, города, страны и почтового индекса) может быть отображен в соответствующий столбец с абстрактным типом данных ADDRESS, со своей собственной внутренней иерархией. Однако преобразование в и из XML теперь требует большего количества решений при проектировании базы данных, поиска компромисса между простотой маршалинга/демаршалинга и гибкостью табличного подхода.

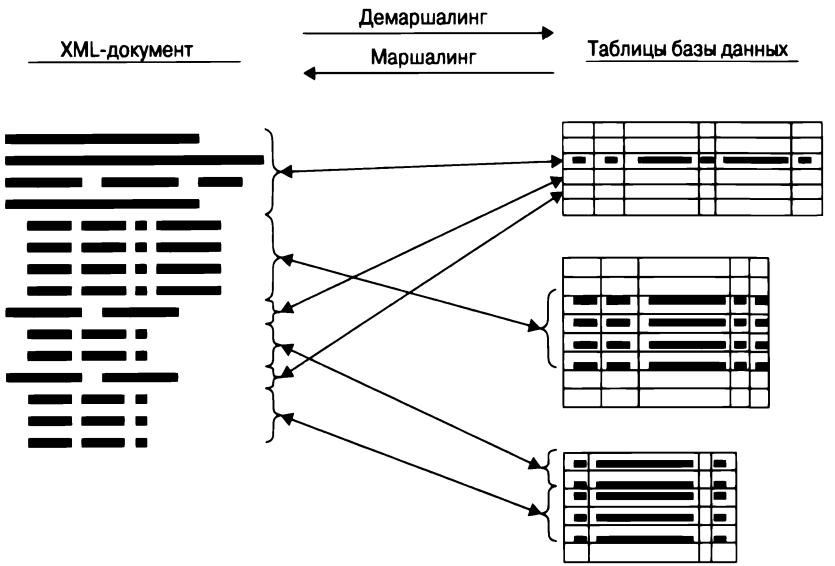


Рис. 25.5. Маршалинг и демаршалинг XML

Некоторые коммерческие продукты начинают предлагать возможности маршалинга/демаршалинга либо анонсируют планы по их реализации в будущих версиях. Накладные расходы, связанные с таким преобразованием, могут существенно влиять на производительность, так что вопрос о популярности этих возможностей при практическом использовании остается открытым. Однако когда приложение работает с внешними данными в виде XML, преобразование данных между XML и SQL в некоторый момент обязано быть выполненным, так что преобразование в самой СУБД может оказаться наиболее эффективным решением.

XML и метаданные

Одним из наиболее мощных качеств реляционной модели является ее очень строгая поддержка типов данных и структур данных, реализуемая определениями таблиц, столбцов, первичных и внешних ключей и ограничений. Кроме того, как показано в главе 16, "Системный каталог", системный каталог реляционной базы данных содержит *метаданные*, или "данные о данных", в базе данных. Запрашивая системный каталог, можно узнать структуру базы данных, включая типы данных столбцов, какие столбцы образуют ее таблицы, и межтабличные взаимоотношения.

XML-документы, напротив, сами по себе содержат только очень ограниченные метаданные. Они навязывают иерархическую структуру элементов своим данным, но единственными данными о структуре являются имена элементов и атрибутов. XML-документ может быть корректен и иметь совершенно неверную структуру. Например, ничто не мешает корректному XML-документу иметь именованный элемент с текстовыми данными в одном экземпляре и с подэлементами в другом или именованный атрибут с целочисленным значением в одном элементе и значением даты в другом. Понятно, что документ с такой структурой, при всей его кор-

ректности, не представляет данные, которые легко трансформировать в реляционную базу данных. При применении XML для обработки данных требуется более строгая поддержка типов данных и структур.

Стандарты XML и программы за время недолгого существования XML решали этот вопрос несколькими способами.

- **Определение типа документа**. Часть исходной спецификации XML 1.0, определение типа документа (Document Type Definitions, DTD) предоставляет способ определения структуры документа. Анализаторы XML могут изучить XML-документ в контексте DTD и определить, является ли он *корректным* (т.е. отвечает ли он ограничениям DTD).
- **XML-Data**. Предложенная консорциуму W3C в 1998 году, спецификация XML-Data была ранней попыткой разрешить некоторые сложности в схеме DTD. Она никогда не получала одобрения W3C, но многие ее идеи получили развитие в спецификации XML Schema. Microsoft приняла собственную версию XML-Data под названием XML-Data Reduced (XDR) и реализовала ее как часть своего сервера интеграции BizTalk и браузера Internet Explorer 5.0.
- **XML Schema**. Отдельная спецификация, которая стала рекомендацией W3C в мае 2001 года, XML Schema была основана на идеях XML-Data. XML Schema обеспечивает более строгую поддержку типов данных и обладает тем преимуществом, что определение схемы (метаданные документа) само по себе является XML-документом, так же как метаданные в реляционной базе данных имеют структуру таблицы.
- **Стандарты промышленных групп**. Как упоминалось ранее, различные промышленные группы объединялись для определения стандартов XML для разных типов документов, используемых для обмена данными в их отрасли. Например, финансовые фирмы работали над стандартами для описания финансовых инструментов и рыночных данных. Фирмы-производители работали над стандартами для описания заказов, их подтверждений и т.п. Такие стандарты для документов определенных типов обычно основаны на обобщенных стандартах, таких как DTD и XML Schema.

Метаданные XML и стандарты типов документов быстро эволюционируют. У W3C имеется часто обновляемый веб-сайт `http://www.w3.org`, который предоставляет доступ к различным стандартам, связанным с XML, и информации об их состоянии. Информацию о промышленных стандартах можно найти по адресу `http://www.xml.org`, где находится веб-сайт организации по продвижению стандартов для структурированной информации (Organization for the Advancement of Structured Information Systems, OASIS). Этот сайт содержит классифицированный по отраслям реестр стандартов, связанных с XML.

DTD

Самая ранняя попытка стандартизации метаданных XML — возможность определения типа документа (Document Type Definition, DTD) в исходной спецификации XML 1.0. DTD используются для определения формы и структуры определен-

ного типа документа (такого, как заказ товара или перевод денег). На рис. 25.6 показано DTD, которое можно использовать для простого заказа на рис. 25.5. Этот DTD демонстрирует только часть возможностей определения типа документа, но все основные компоненты DTD в нем присутствуют.

```
<!ELEMENT purchaseOrder (customerNumber, orderNumber,
                         orderDate, terms, orderItem*)>
<!ELEMENT customerNumber (#PCDATA)>
<!ELEMENT orderDate (#PCDATA)>
<!ELEMENT repNumber (#PCDATA)>
<!ELEMENT terms EMPTY>
<!ATTLIST terms
    ship CDATA
    bill CDATA#REQUIRED>
<!ELEMENT orderItem (mfr, product, qty, amount)>
<!ELEMENT mfr (#PCDATA)>
<!ELEMENT product (#PCDATA)>
<!ELEMENT qty (#PCDATA)>
<!ELEMENT amount (#PCDATA)>
```

Рис. 25.6. DTD для простого заказа

Записи `!ELEMENT` в DTD определяют иерархию элементов. DTD определяет различные типы элементов.

- **Текстовый элемент**. Элемент, содержащий только текстовую строку, которая может представлять значение данных из одного столбца таблицы базы данных.
- **Элемент с элементами**. Содержимым элемента являются другие элементы (подэлементы); данный элемент является родительским в локальной иерархии дочерних и родительских элементов. Этот тип элемента может использоваться для представления строки таблицы; подэлементы при этом представляют столбцы.
- **Элемент со смешанным содержимым**. Такой элемент может содержать смесь текстового содержимого и подэлементов. Этот тип обычно бесполезен для представления содержимого баз данных, поскольку смесь подэлементов и данных естественным образом в табличной структуре не представима.
- **Пустой элемент**. Такой элемент не имеет содержимого — ни подэлементов, ни текста, — но может иметь атрибуты. Элемент этого типа может представлять строку таблицы, если его атрибуты используются для представления значений отдельных столбцов.
- **Элемент с произвольным содержимым**. На содержимое этого элемента не накладываются никакие ограничения. Элемент может быть пустым, содержать подэлементы и/или текст. Подобно элементам со смешанным содержимым, этот тип обычно бесполезен для представления содержимого баз данных.

В DTD заказа на рис. 25.6, элемент верхнего уровня `purchaseOrder` и элемент `orderItem` являются элементами с элементами. Их объявления перечисляют эле-

менты, которые могут в них содержаться. Элементы `customerNumber` и `orderDate` являются текстовыми элементами, на что указывают определения `#PCDATA`. Элемент `terms` пустой; он может иметь только атрибуты. Оба атрибута могут иметь символьные значения (на что указывает тип `CDATA`); один атрибут обязателен, второй — нет. Заметим, что в этом DTD стили "данные как элементы" (для информации о покупателе) и "данные как атрибуты" (для заказа) комбинируются только в иллюстративных целях. На практике для упрощения обработки следует выбрать тот или иной стиль представления данных и последовательно его придерживаться.

DTD критичны для того, чтобы XML имел практическую ценность для представления структурированных документов для обмена данными. Они позволяют определить важнейшие элементы транзакционного документа, такого как заказ товара. При наличии DTD для такого документа легко проверить его правильность независимо от его происхождения. Любой XML-анализатор, как на основе DOM API, так и на основе SAX API, в состоянии выполнить проверку XML-документа на соответствие указанному DTD. Кроме того, можно явно объявить DTD, которому должен соответствовать XML-документ, в самом документе.

У DTD имеются и определенные недостатки. В них не хватает той строгости типизации данных, которая присуща реляционной базе данных. Например, нет никакого способа указать, что элемент должен содержать целое число или дату. У DTD также нет поддержки пользовательских типов или структур поддокументов. Например, элемент `orderItem` на рис. 25.6 может использоваться не только в заказе, но и в документе изменения заказа, отмены заказа и т.п. Было бы удобно объявить подструктуру `orderItem` один раз, дать ей имя, а затем обращаться к ней в определениях других документов. К сожалению, DTD не предоставляет такой возможности.

DTD также несколько ограничены в смысле типов структур содержимого, которые они допускают, хотя на практике их обычно достаточно для поддержки транзакционных документов, требующихся для гибридных приложений. Наконец, выражения, которые DTD применяют для определения структуры документа, представляют собой расширенную форму Бэкуса-Наура (BNF) (примером может служить звездочка после объявления `orderItem` в списке элементов `purchaseOrder` на рис. 25.6, которая означает "этот элемент может повторяться нуль или несколько раз").

Хотя эта форма и знакома студентам-информатикам, изучающим языки программирования, но о ней не знают те, кто знаком с XML по разметке документов HTML. Все эти недостатки проявились сразу же после принятия XML 1.0, и тут же началась работа над более мощным языком метаданных XML. Эта работа привела к появлению спецификации XML Schema, описанной в следующем разделе.

XML Schema

XML Schema 1.0 получила статус официальной рекомендации W3C в мае 2001 года, и ее быстро стали поддерживать многие коммерческие XML-продукты. DTD продолжают поддерживаться для обеспечения обратной совместимости, но XML Schema обладает некоторыми важными преимуществами и решает большинство проблем DTD. На рис. 25.7 приведена схема документа для заказа с рис. 25.4, в этот раз определенная с использованием XML Schema. Полезно сравнить объявле-

ние XML Schema на рис. 25.7 с объявлением DTD на рис. 25.6. Даже этот простой пример показывает строгую типизацию в XML Schema; элементы и атрибуты имеют типы данных, которые очень похожи на типы данных SQL. Кроме того, схема на рис. 25.7 также представляет собой XML-документ, так что она более понятна для тех, кто знаком с азами XML, чем DTD на рис. 25.6.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="purchaseOrder" type="POType"/>
  <complexType name="POType">
    <sequence>
      <element name="customerNumber" type="integer"/>
      <element name="orderNumber" type="integer"/>
      <element name="orderDate" type="date"/>
      <element name="repNumber" type="integer" length="3"/>
      <element name="terms">
        <attribute name="ship" type="string"/>
        <attribute name="bill" type="string"/>
      </element>
      <element name="orderItem" minOccurs="0"
                                maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="mfr" type="string" length="3"/>
            <element name="product" type="string"/>
            <element name="qty" type="integer"/>
            <element name="amount"  type="decimal"
                                fractionDigits="2"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</schema>
```

Рис. 25.7. XML Schema для простого заказа

Типы данных в XML Schema

С точки зрения базы данных, поддержка типов данных и структур данных в XML Schema является одним из ее важных преимуществ. XML Schema определяет более 20 встроенных типов данных, которые соответствуют типам данных SQL. В табл. 25.2 перечислены наиболее важные встроенные типы данных XML Schema.

Как и стандарт SQL, XML Schema поддерживает пользовательские типы данных, порожденные из встроенных типов данных или других пользовательских типов. Вы можете указать производный тип данных как ограничение на другой тип XML. Например, вот определение порожденного типа `repNumType`, который ограничивает допустимые номера служащих значениями от 101 до 199.

```
<simpleType name="repNumType">
  <restriction base="integer">
    <minInclusive value="101" />
    <maxExclusive value="200" />
  </restriction>
</simpleType>
```

Таблица 25.2. Встроенные типы данных XML Schema

Типы данных XML Schema	Описание
Числовые данные	
`Integer`	Целое число
`PositiveInteger`	Только положительные целые числа
`NegativeInteger`	Только отрицательные целые числа
`NonNegativeInteger`	Только нуль или положительные целые числа
`NonPositiveInteger`	Только нуль или отрицательные целые числа
`Int`	32-битовое знаковое целое число
`UnsignedInt`	32-битовое беззнаковое целое число
`Long`	64-битовое знаковое целое число
`UnsignedLong`	64-битовое беззнаковое целое число
`Short`	16-битовое знаковое целое число
`UnsignedShort`	16-битовое беззнаковое целое число
`Decimal`	Числа с возможными десятичными знаками
`Float`	Числа с плавающей точкой стандартной точности
`Double`	Числа с плавающей точкой двойной точности
Символьные данные	
`String`	Символьная строка переменной длины
`NormalizedString`	Строка с преобразованными в пробелы символами новой строки, возврата каретки и табуляции
`Token`	Строки, обработанные как `NormalizedString`, а также с удаленными ведущими и завершающими пробелами и свернутыми к одному пробелу последовательностями пробелов
Дата и время	
`Time`	Время дня (часы/минуты/секунды/тысячные доли)
`DateTime`	День и время (эквивалент SQL `TIMESTAMP`)
`Duration`	Интервал времени (эквивалент SQL `INTERVAL`)
`Date`	Только год/месяц/день
`Gmonth`	Григорианский месяц (от 1 до 12)
`Gyear`	Григорианский год (от 0000 до 9999)
`Gday`	Григорианский день (от 1 до 31)
`GmonthDay`	Григорианский месяц/день
Прочие данные	
`Boolean`	Значения `TRUE/FALSE`
`Byte`	Знаковый байт
`UnsignedByte`	Беззнаковый байт
`base64Binary`	Бинарные данные, выраженные в виде `base64`
`HexBinary`	Бинарные данные, выраженные в виде шестнадцатеричных чисел
`AnyURI`	URI, такой как `http://www.w3.org`
`Language`	Корректный язык XML (английский, французский и т.д.)

При таком определении типа данных можно объявить сущности или атрибуты в схеме как имеющие тип данных repNumType, и это ограничение будет автоматически реализовано. XML Schema предоставляет богатый набор характеристик типов данных (именуемых *аспектами*) для создания ограничений. Сюда входят размер данных (для строк и бинарных данных), включающие и исключающие диапазоны данных, количество цифр и десятичных знаков (для числовых данных), а также явное перечисление допустимых значений. Имеется даже встроенная возможность проверки соответствия шаблонам с применением синтаксиса регулярных выражений, подобно используемой в языке сценариев Perl.

XML Schema также дает возможность определять сложные типы данных, являющиеся определяемыми пользователем структурами. Например, вот как выглядит определение сложного типа custAddrType, который состоит из знакомых подэлементов.

```
<complexType name="custAddrType">
  <sequence>
    <element name="street"   type="string" />
    <element name="city"     type="string" />
    <element name="state"    type="string" />
    <element name="postCode" type="integer" />
  </sequence>
</complexType>
```

Можно также создать пользовательский тип данных, который представляет собой список элементов другого типа. Например, вот определение сложного типа repListType, который является списком номеров служащих.

```
<simpleType name="repListType">
    <list itemType="repNumType" />
</simpleType>
```

XML Schema позволяет перегрузить пользовательский тип данных, разрешая ему принимать один из нескольких различных типов данных в зависимости от конкретных нужд. Например, в предыдущем определении custAddrType почтовый индекс определен как целое число. Это сработает для пятизначных индексов на Украине (за исключением того, что не будет сохранен ведущий нуль), но в других странах (например, в Канаде) шестисимвольный индекс включает буквы и цифры. Можно объединить эти индексы в один более универсальный индекс, который может быть любым из указанных.

```
<simpleType name="ua5Type">
    <restriction base="integer">
        <totalDigits value=5 />
    </restriction>
</simpleType>
<simpleType name="ca6Type">
    <restriction base="string">
        <length value=6 />
    </restriction>
</simpleType>
<simpleType name="intlPostType">
    <union memberTypes="ua5Type ca6Type" />
</simpleType>
```

Имея определения пользовательских типов данных, можно легко определить большие и более сложные структуры. Например, вот часть заказа с рис. 25.7, расширенная таким образом, чтобы включать адреса для отправки счета и товара и допускать наличие списка торговых представителей.

```
<complexType name="purchaseOrderType">
    ... другие элементы объявлений ...
    <element name="billAddr" type="custAddrType" />
    <element name="shipAddr" type="custAddrType" />
    <element name="repNums" type="repListType" />
    ... другие элементы объявлений ...
```

Элементы и атрибуты в XML Schema

XML Schema предоставляет богатый словарь для указания корректной структуры типа документа и допустимых элементов и атрибутов, составляющих его. XML Schema поддерживает те же базовые типы элементов, что и определенные в модели DTD.

- **Простое содержимое**. Элемент содержит только текстовое содержимое (хотя, как пояснялось в предыдущем разделе, текст может быть ограничен данными определенного типа, например датой или числовыми значениями). Содержимое этого типа определяется с использованием элемента `simpleContent`.
- **Элементное содержимое**. Данный элемент может содержать только подэлементы. Содержимое этого типа определяется с использованием элемента `complexType`.
- **Смешанное содержимое**. Данный элемент содержит смесь подэлементов и собственного текстового содержимого. В отличие от смешанной модели DTD, XML Schema требует, чтобы последовательность элементов и текстового содержимого была строго определена, и корректный документ должен отвечать определенной последовательности. Содержимое данного типа определяется с использованием атрибута `mixed` элемента `complexType`.
- **Пустое содержимое**. Элемент содержит только атрибуты, и никакого текстового содержимого. XML Schema рассматривает такой элемент как частный случай элементного содержимого без объявленных элементов.
- **Произвольное содержимое**. Элемент содержит любое сочетание содержимого и подэлементов в любом порядке. Содержимое этого типа определяется с использованием типа данных XML Schema `anyType` в качестве типа данных элемента.

Эти базовые типы элементов могут встречаться в объявлениях элементов схемы. Кроме того, можно указать, что элемент может встречаться в документе многократно, а также (необязательно) указать минимальное и максимальное количество вхождений элемента в документ. XML Schema поддерживает для элементов значения в стиле SQL `NULL` для указания того, что содержимое элемента неизвестно. По терминологии XML, это значения `nil`, но концепция их та же, что и у значений `NULL` в

SQL. Эта возможность упрощает отображение данных между элементами XML-документа и столбцами базы данных, которые могут содержать значения NULL.

XML Schema позволяет вам определить логическую группу элементов, которые обычно используются совместно, и дать этой группе имя. Последующие объявления элементов могут включать всю именованную группу элементов как одно целое. Сгруппированные элементы обеспечивают дополнительную гибкость структуры элементов. Группа может определять *последовательность* элементов, которые должны присутствовать все вместе в указанном порядке. Можно указать и возможность *выбора* элементов, когда в документ должен входить только один из множества определенных типов элементов.

XML Schema предоставляет подобное управление и атрибутами. Вы можете указать отдельный атрибут как обязательный или необязательный, определить значение атрибута по умолчанию, которое должно использоваться, если значение атрибута в документе явно не указано, или фиксированное значение атрибута, которое заставляет атрибут *всегда* иметь это значение в экземпляре документа. Группы атрибутов позволяют определять и именовать набор атрибутов, которые обычно употребляются вместе. Вся группа атрибутов может быть объявлена для элемента в схеме просто при помощи указания ее имени.

Наконец, XML Schema поддерживает пространства имен XML, которые применяются для хранения различных *словарей* XML и управления ими, т.е. различными коллекциями определений типов данных и объявлений структур данных, используемых для различных целей. В большой организации полезно определить стандартизованные XML-представления для распространенных базовых бизнес-объектов, таких как адреса, идентификаторы товаров, идентификаторы клиентов и т.п., и собрать их в едином репозитории. Объявления XML более высокого уровня для документов, таких как заказы, отпускные записки, счета и т.п., также полезны, но, как правило, должны быть собраны вместе в группы в соответствии с их совместным использованием.

Пространства имен XML поддерживают такие возможности, позволяя собирать связанные определения и объявления XML, сохранять их в файле и идентифицировать по имени. XML Schema для нового типа документа может затем брать определения фундаментальных данных и структур из одного или нескольких пространств имен, указывая их в заголовке схемы. Стандартный словарь XML и многие встроенные типы данных определены в пространстве имен, поддерживаемом на веб-сайте W3C. Исходный файл пространства имен XML идентифицируется при помощи URL.

Если объявление XML Schema включает определения более чем из одного пространства имен XML, возможен конфликт имен. Одно и то же имя может быть легко выбрано разработчиками двух разных пространств имен для представления совершенно разных структур или типов данных XML. Для устранения этой потенциальной неоднозначности определения типов данных и структур XML могут быть указаны с применением *квалифицированных* имен (с использованием метода, похожего на использование квалифицированных имен столбцов в SQL). Каждое пространство имен, указанное в заголовке схемы, может получить собственный *префикс*, который затем используется для квалифицированного обращения к элементам в этом про-

странстве имен. Для простоты префиксы в примерах схем в этой главе были опущены. Вот более типичный заголовок схемы и отрывок из тела схемы с использованием префиксов и квалификации для ссылок на основное пространство имен XML Schema (поддерживаемое W3C) и корпоративное пространство имен.

```
<schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:corp="http://www.company.com/schemas/purchase">
<complexType name="purchaseOrderType">
    ... другие элементы объявлений ...
    <element name="orderDate" type="xsd:date" />
    <element name="billAddr"  type="corp:custAddrType" />
    <element name="shipAddr"  type="corp:custAddrType" />
    <element name="repNums"   type="corp:repListType"
                              nillable="true" />
    ... другие элементы объявлений ...
```

В этом примере корпоративное пространство имен XML идентифицируется префиксом `corp`, а основное пространство имен XML Schema — префиксом `xsd`. Все ссылки на типы данных квалифицированы одним из этих префиксов, так что в результате они не являются неоднозначными. Поскольку квалифицированные ссылки могут оказаться весьма громоздкими, для минимизации количества используемых префиксов можно указать пространство имен по умолчанию. Полная система именования XML Schema существенно более сложна, чем здесь описано, но и из этого описания ясно, что XML Schema поддерживает создание очень сложных спецификаций типов документов большими группами разработчиков.

Как и в случае DTD, основное преимущество XML Schema заключается в возможности создания хорошо определенных типов документов, на соответствие которым можно проверять конкретные их экземпляры. Все популярные XML-анализаторы, реализующие как SAX API, так и DOM API, обеспечивают проверку корректности документов на основе XML Schema. Вы можете включить схему, которой должен соответствовать XML-документ, в сам документ, но анализаторы в состоянии выполнить проверку любого XML-документа на соответствие произвольной схеме.

XML и запросы

SQL предоставляет возможность выполнения запросов для поиска, преобразования и выборки структурированных данных из реляционных баз данных, так что вполне естественно встает вопрос об аналогичном средстве для поиска, преобразования и выборки структурированных данных из XML-документов. Первые усилия в этом направлении — определить средство для запросов и преобразования данных — привели к появлению двух спецификаций: языка преобразования расширяемых листов стилей (Extensible Stylesheet Language Transformation, XSLT) и языка XML Path (XPath). Как и у самого XML, корни этих спецификаций находятся в обработке документов.

XSLT ориентирован на преобразование XML-документа, как показано на рис. 25.8. Лист стилей определяет выполняемое преобразование, отбирая преобразуемые элементы XML-документа и указывая, как они должны модифицироваться и комбинироваться с другими элементами для получения выходного XML-

документа. Одно из распространенных применений XSLT — преобразование единой обобщенной версии веб-страницы в различные версии для разных размеров экранов и устройств вывода.

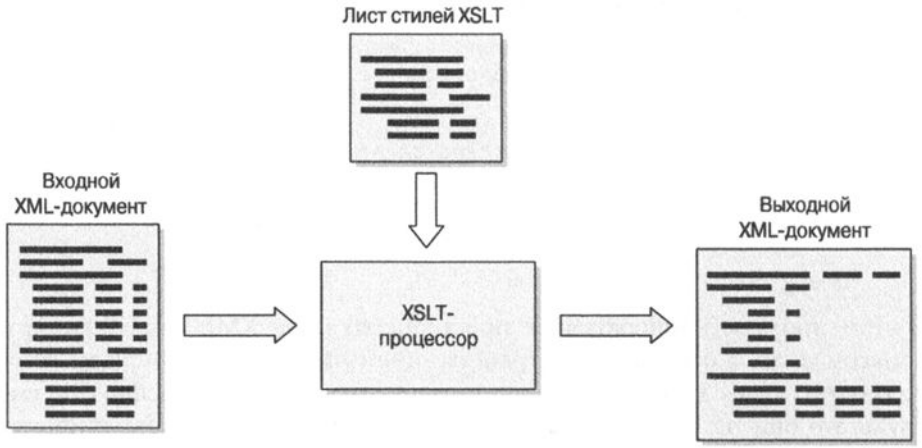


Рис. 25.8. Преобразование XML-документа при помощи XSLT

При использовании языка XSLT часто требуется отобрать для преобразования отдельные элементы или группы элементов или перемещаться по иерархии элементов для получения данных из родительских и дочерних элементов. XPath изначально появился как часть языка XSLT для отбора элементов и навигации. Однако быстро выяснилось, что XPath подходит и для других приложений, и спецификация была разделена. В самом начале существования XML XPath де-факто представлял собой средство запросов для XML-документов.

Позже выяснилось, что XPath оказался не совсем полноценным языком, так что была сформирована рабочая группа W3C, задачей которой была разработка средства запросов под рабочим названием XML Query, или XQuery. После нескольких черновых версий рабочая группа XSL (ответственная за XSLT и XPath) и рабочая группа XQuery объединили свои усилия. В январе 2007 года были опубликованы стандарты XQuery 1.0 и XPath 2.0. Эти два языка тесно связаны и там, где это возможно, используют общие синтаксис и семантику.

Полное описание XQuery и XPath выходит за рамки данной книги. Однако краткий обзор концепций XQuery и несколько примеров проиллюстрируют его отношения с SQL.

Концепции XQuery

В основе SQL лежит табличная модель данных; модель же данных в основе XQuery — древовидная иерархия узлов, представляющая XML-документ. В действительности XQuery использует более тонкую структуру, чем иерархия элементов XML-документов и XML Schema. Узлы XQuery соотносятся с запросами баз данных.

- **Узел элемента**. Этот тип узла XQuery представляет сам элемент.
- **Текстовый узел**. Этот тип узла представляет содержимое элемента. Он является дочерним по отношению к узлу соответствующего элемента.

- **Узел атрибута**. Этот тип узла представляет атрибут и значение атрибута элемента. Он является дочерним по отношению к узлу соответствующего элемента.
- **Узел документа**. Это специализированный узел элемента, который представляет верхний, или *корневой*, уровень документа.

Для навигации по дереву и идентификации одного или нескольких объектов для обработки XQuery использует *выражение пути*. Во многих смыслах выражение пути играет в XQuery ту же роль, что в SQL, — выражение запроса. Выражение пути идентифицирует отдельный узел в дереве объектов XQuery путем указания последовательности шагов перемещения по иерархии, необходимых для достижения узла. Выражения пути XQuery делятся на два типа.

- **Выражения пути от корня**. Такие выражения начинаются от вершины (корня) дерева объектов и идут вниз по иерархии, пока не достигнут указанного элемента. В документе книги, представленном на рис. 25.1, выражение пути `/bookPart/chapter/section/para` приводит к отдельному абзацу в разделе главы.
- **Относительное выражение пути**. Такое выражение начинается с текущего узла дерева (узла, обрабатываемого в настоящий момент) и идет вниз и/или вверх по иерархии, пока не достигнет узла назначения. В документе книги, представленном на рис. 25.1, относительное выражение пути `section/para` ведет вниз к определенному абзацу, если текущий узел является узлом `chapter`.

Шаги в пути могут определять перемещение вниз по дереву к дочерним узлам, представляющим подэлементы, содержимое элементов или атрибуты элементов. Эти шаги могут определять и движение вверх, к родительскому узлу. На каждом шаге можно указать условие *проверки узла*, которое должно быть удовлетворено для того, чтобы продолжалось перемещение по пути к целевому элементу. В табл. 25.3 приведены некоторые типичные выражения пути и перемещения, которые они определяют.

Подобно SQL, XQuery — язык, ориентированный на множества. Он оптимизирован для работы с *последовательностями* XQuery, которые представляют собой упорядоченные коллекции из нуля или большего количества объектов. Эти объекты могут быть элементами, атрибутами или значениями данных. Операции XQuery, в основном, получают в качестве входных данных последовательности и на выходе дают другие последовательности. Простой атомарный объект данных обычно рассматривается как последовательность из одного объекта.

XQuery также напоминает SQL тем, что является строго типизированным языком. Рабочая черновая спецификация XQuery эволюционировала в сторону поддержки XQuery тех же типов данных, что и у XML Schema (которые были описаны ранее, в разделе "XML Schema"). Подобно SQL, XQuery предоставляет конструкторы для создания значений сложных данных.

Таблица 25.3. Некоторые типичные выражения пути

Выражение пути	Навигация
section/para	Перемещение вниз к дочернему элементу section, а от него — к его дочернему элементу para
/bookPart/chapter/section	Начиная с вершины иерархии, переход вниз через bookPart, затем через его дочерний узел chapter к дочернему узлу последнего section
..	Перемещение вверх от текущего узла к его родительскому узлу
../chapter	Перемещение вверх от текущего узла к его родительскому узлу, а затем вниз к дочернему узлу chapter
.//para	Выбор любого дочернего узла para, который находится в иерархии где угодно ниже текущего узла
@@hdrLevel	Выбор атрибута hdrLevel текущего узла
/header@hdrLevel	Выбор атрибута hdrLevel дочернего узла header
para[3]	Выбор третьего дочернего элемента типа para
*	Выбор всех дочерних элементов текущего узла
*/para	Выбор всех "внучатых" элементов para текущего узла
chapter[@revStatus="draft"]	Выбор всех дочерних элементов chapter текущего узла, у которых имеется атрибут status со значением draft

XQuery существенно отличается от SQL, будучи языком, ориентированным на выражения, а не на инструкции. Кстати говоря, в XQuery все является выражениями, вычисление которых дает значение. Выражения пути представляют собой один тип выражений XQuery, которые генерируют в качестве результата последовательности узлов. Другие выражения могут объединять литеральные значения, вызовы функций, арифметические и логические выражения, а также заключенные в скобки их комбинации, образующие выражения произвольной сложности. Выражения могут также комбинировать последовательности узлов, используя такие операторы, как объединение или пересечение множеств (такие же, как и соответствующие операции с множествами в языке SQL).

Именованные переменные в XQuery указываются при помощи ведущего символа доллара ($) в имени. Например, $orderNum, $currentOffice и $c представляют собой корректные имена переменных XQuery. Переменные могут свободно использоваться в выражениях XQuery, объединяясь с литералами и другими переменными и значениями узлов. Переменные получают новые значения при помощи вызовов функций и путем присваивания в выражениях for или let.

Обработка запросов в XQuery

Выражения пути XQuery обеспечивают XML-эквивалент простой SQL-инструкции SELECT с предложением WHERE. Предположим, что коллекция XML-документов содержит XML-эквивалент содержимого учебной базы данных; при этом документы верхнего уровня носят имена таблиц учебной базы данных, а структуры отдельных

строк — имена, представляющие собой единственное число этих имен (так, документ OFFICES содержит отдельные элементы OFFICE, которые представляют строки таблицы OFFICES). Вот несколько запросов и соответствующие им выражения пути.

Найти офисы под управлением сотрудника с идентификатором 108.

```
/offices/office[mgr=108]
```

Найти все офисы с продажами, превышающими план.

```
/offices/office[sales > target]
```

Найти все заказы на товары производителя ACI с суммами свыше $30 000.

```
/orders/order[mfr = 'ACI' and amount > 30000.00]
```

Поскольку учебная база данных представляет собой неглубокую табличную структуру, XML-иерархия имеет глубину, равную всего лишь трем. Для иллюстрации возможностей запросов в более иерархичных документах XML рассмотрим еще раз книгу на рис. 25.1. Вот несколько запросов и соответствующих выражений пути.

Найти все составные части глав со статусом черновика.

```
/bookPart/chapter[revStatus="draft"]/*
```

Получить третий абзац второй главы части 2.

```
/bookPart[@partNum="2"]/chapter[2]/para[3]
```

Эти выражения не дают нам возможности управления результатами запроса, которую предоставляет инструкция SELECT. Здесь нет и эквивалента курсорам SQL для построчной обработки результатов. Вместо этого XQuery предоставляет выражения For/Let/Where/Returns (выражения FLWR). Лучше всего проиллюстрировать эти возможности на конкретных примерах. Рассмотрим еще раз структуру XML-документов для учебной базы данных. Приведенный запрос реализует двухтабличное соединение и генерирует три указанных столбца результатов запроса.

Перечислить все имена продавцов, даты заказов и их суммы, для заказов менее чем на $5000. Отсортировать результаты по суммам заказов.

```
<smallOrders> {
  for $o in document("orders.xml")//orders[amount < 5000.00],
      $r in document("salesreps.xml")//salesreps[empl_num=$o/rep]
  return
    <smallOrder> {
      $r/name,
      $o/order_date,
      $o/amount
    }
    </smallOrder>
  sortby(amount)
  }
</smallOrders>
```

На внешнем уровне содержимое элемента smallOrders определяется выражением XQuery, заключенным во внешние фигурные скобки. Выражение for использует две переменные для итерации по двум документам, соответствующим

таблицам ORDERS и SALESREPS. Эти две переменные эффективно реализуют соединение двух таблиц (документов). Предикаты (аргументы поиска) в конце каждой из двух строк после ключевого слова for соответствуют предложению SQL WHERE. Предикат в первой строке ограничивает запрос только теми заказами, сумма которых меньше $5000. Предикат во второй строке реализует соединение, используя переменную $o для связи строк в таблице (документе) SALESREPS со строками таблицы (документа) ORDERS.

Часть return выражения for указывает, какие элементы должны быть возвращены в качестве результата вычисления выражения. Она соответствует списку инструкции SELECT в запросе SQL. Возвращаемое значение представляет собой XML-последовательность элементов smallOrder, где каждый элемент берется из соответствующего элемента исходных таблиц (документов). Здесь переменные используются для того, чтобы квалифицировать путь к элементу, значение которого будет возвращено. Наконец, часть sortby выражения функционирует в точности так же, как и соответствующее предложение ORDER BY в запросе SQL.

В этом примере не проиллюстрированы некоторые дополнительные возможности. В итерации for для получения значений дополнительных переменных (которые могут потребоваться предикатам или другим выражениям) можно использовать выражение let. Условное выполнение поддерживается при помощи конструкции if...then...else. Статистические функции позволяют группировать запросы XQuery, как это делается в итоговых запросах SQL, описываемых в главе 8, "Итоговые запросы". Все это делает запросы XQuery сравнимыми по гибкости с запросами SQL. Однако, как видно из приведенного примера, стиль выражений в этих языках существенно отличается, отражая ориентацию языка XQuery как на выражения, так и на навигацию.

Базы данных на основе XML

С распространением XML были созданы несколько компаний, попытавшихся создать коммерческие XML-базы данных. Обычно такие базы данных хранят свои данные в виде XML-документов. Содержимое базы данных может храниться как в "родном" формате, в виде XML-текста, так и в некотором ином виде, например в виде, используемом XML-анализатором на базе DOM для представления документа в памяти. Большинство современных XML-баз данных поддерживает XPath как фундаментальную возможность, при этом многие из них добавляют к XPath собственные расширения, чтобы получить более полный язык запросов. Кроме того, в таких базах данных часто реализуется поддержка XQuery — либо вместо XPath, либо в качестве второго, дополнительного, языка запросов.

Производители XML-баз данных обычно приводят те же аргументы в пользу своих разработок, что и производители объектно-ориентированных баз данных десятилетием ранее. Чем больше данных в мире оказывается представлено в виде XML, тем больший интерес представляют базы данных, основанные на той же модели данных. Выбор XML-документов в качестве внутреннего формата снижает накладные расходы на маршалинг и демаршалинг, обеспечивая сохранение доступности отдельных элементов и атрибутов и навигацию в иерархической струк-

туре. Наконец, они говорят о том, что постоянно растет количество пользователей, знакомых с HTML и XML, так что XML-базы данных будут доступны для большего количества пользователей, чем реляционные базы данных на основе SQL.

Пока еще рано судить об успехе XML-баз данных на рынке. Похоже, что XML-базы данных могут оказаться хорошим выбором для работы с данными, например, на веб-сайтах, где требуется хранение XML-документов, доступ к ним и их преобразование. Однако история объектно-ориентированных баз данных говорит о том, что основного успеха добьются реляционные базы данных, в которые будет добавлена поддержка наиболее важных возможностей новой модели данных, причем темпы эволюции реляционных баз данных будут достаточно быстрыми, чтобы позволить им продолжать доминировать на рынке. Создается впечатление, что реляционные базы данных будут и дальше доминировать в области обработки данных. Однако со временем будет расти степень интеграции в эти продукты, и со временем реляционные базы данных будут предлагать все больше и больше XML-ориентированных возможностей.

Резюме

В этой главе рассматривались взаимоотношения XML и SQL, а также XML-документов и реляционных баз данных.

- Происхождение XML связано с потребностями печати и издательской деятельности; изначально этот язык создавался как метод определения содержимого документов.
- Ориентация XML на работу с документами приводит к естественному иерархическому рассмотрению данных. Несоответствие иерархий XML и таблиц SQL — одна из наибольших трудностей при интеграции этих технологий.
- XML-документы включают иерархию элементов. Элементы могут включать содержимое, иметь именованные атрибуты, а также другие дочерние элементы.
- Интеграция XML с реляционными базами данных может принимать различные формы — генерация XML-вывода, входные данные в XML-формате, обмен данными в формате XML, а также хранение и выборка XML-данных в базе данных.
- XML Schema и более старый стандарт XML DTD определяют структуры документов определенных типов. Они полезны для приведения содержимого документов к стандартному виду, приемлемому для приложений, работающих с данными.
- XQuery — язык запросов для XML-документов. В нем есть определенные параллели с SQL, но его направленность на выражения и навигацию приводит к существенному отличию его стиля от стиля SQL.
- Делаются попытки разработки XML-баз данных с XQuery в качестве языка запросов. Однако на это ведущие производители DBMS небезуспешно отвечают включением поддержки XML в свои реляционные базы данных.

26
ГЛАВА

Специализированные базы данных

На сегодняшнем рынке баз данных доминируют большие системы управления базами данных уровня предприятия. Флагманские продукты Oracle, IBM, Microsoft, Sybase и других производителей представляют собой сложное программное обеспечение, которое стремится удовлетворить все возможные потребности, следуя принципу "один для всех". Большие компании могут использовать последние версии баз данных Oracle или IBM DB2 для обработки транзакций со своего веб-сайта, в качестве хранилища данных, для анализа бизнес-данных, управления финансами и для поддержки баз данных отделов. Все эти приложения представляют различные задачи и различные нагрузки, но базы данных уровня предприятия достаточно мощные и гибкие, чтобы удовлетворить их все.

Однако у некоторых приложений требования настолько специализированы или настолько жесткие, что обычная база данных уровня предприятия не в состоянии им соответствовать. Для таких приложений требуются свои, специализированные, базы данных. И еще одним доказательством мощи SQL служит то, что даже такие узкоспециализированные базы данных сегодня используют этот язык. В данной главе рассмотрим четыре ниши специализированных баз данных и возможности SQL, добавленные для того, чтобы соответствовать их требованиям.

Низкие задержки и базы данных в памяти

Некоторые важные приложения баз данных в области телекоммуникаций и финансовых служб требуют очень быстрый ответ от базы данных, зачастую в течение микросекунд. Например, у каждой сети мобильной связи имеется база данных, которая отслеживает текущее местоположение каждого мобильного телефона и то, какой базовой станцией он обслуживается. Такая база данных должна быть всегда актуальна, с тем чтобы в любой момент можно было определить, куда сле-

дует направить входящий звонок для любого подключенного к сети мобильного телефона. Если вызов уже сделан, а пользователь перемещается, требования к базе данных становятся еще выше, так как этот вызов должен передаваться от одной базовой станции к другой. В сети из десятков миллионов мобильных телефонов каждое отдельное обращение к базе данных должно быть очень кратким, иначе база данных станет узким местом при обработке вызовов.

Требования к базе данных становятся еще жестче в случае пользователя с предоплатой. Когда такой пользователь пытается воспользоваться телефоном — делает звонок, отправляет текстовое сообщение или выходит в Интернет — сеть должна быстро определить, разрешено ли ему выполнять такое действие. За кулисами программное обеспечение должно определить, кто пытается воспользоваться сервисом, выяснить его тарифный пакет, определить стоимость услуги (которая может меняться в зависимости от дня недели, времени суток или даже от местоположения клиента и прочих факторов), просмотреть текущий счет клиента, выяснить, достаточно ли у него денег, чтобы позволить ему запрошенные действия, и списать с его счета плату за соединение. Такая последовательность действий может потребовать десятков обращений к базе данных и обновления финансовой базы данных оператора — и все это за время, пока пользователь ожидает предоставления запрошенной услуги. Зачастую у мобильного оператора десятки миллионов клиентов, большинство которых используют предоплату, так что обращения к базе данных должны быть очень, очень быстрыми.

Биржевые приложения выдвигают похожие требования. Поток данных, таких как запросы на покупку или продажу, может превышать 50000 сообщений в секунду, и это число за год обычно удваивается. Соответствующие приложения, отслеживающие текущее состояние рынка, должны быть в состоянии обработать каждое сообщение (обычно требующее обновления строки базы данных) менее чем за 20 микросекунд, иначе они просто не смогут справляться с создавшейся очередью входных сообщений. Кроме того, возможность быстрой реакции на изменяющиеся условия рынка — одно из важнейших конкурентных преимуществ, так что уменьшение времени обработки запроса даже на несколько микросекунд может стоить миллионов долларов в год.

Требования приложений наподобие описанных в настоящее время выходят за рамки основного направления развития баз данных. В типичной архитектуре "клиент/сервер" задержки, связанные с сетью, зачастую измеряются миллисекундами, так что даже бесконечно быстрая база данных будет давать слишком большие задержки. Если убрать задержки, связанные с сетью, останется доступ к физическому диску с базой данных, который также может занимать десятки миллисекунд. Если убрать и эти задержки, то сложность программного обеспечения базы данных даже в простейшей базе данных может потребовать для обработки запроса десятков миллисекунд процессорного времени.

Чтобы соответствовать столь экстремальным требованиям, разработчики исторически создают собственные доморощенные базы данных под требования каждого приложения. Каждая уважающая себя фирма на Уолл-стрит имеет собственный информационный отдел, который разрабатывает свои системы для проведения торгов. В конце 1990-х годов для решения подобных задач появился новый вид баз данных — *базы данных в памяти* на базе SQL.

Анатомия баз данных в памяти

Резидентные базы данных, полностью размещающиеся в памяти (in-memory database), используя традиционную архитектуру баз данных уровня предприятия, радикально изменяют некоторые базовые принципы. В обычной базе данных активные данные хранятся на дисках, и в любой момент времени в оперативной памяти компьютера находится только малая часть данных. В базе данных в памяти *все* данные хранятся в оперативной памяти (откуда и происходит название таких баз данных). Каждое обращение или изменение может быть удовлетворено обращением к оперативной памяти; обращаться к дискам за данными не приходится никогда. База данных может поддерживать журнал транзакций на диске или хранить там копию базы данных для восстановления после ошибок (большинство коммерческих продуктов именно так и поступает), но обращение к диску не входит в действия по выполнению операций с данными.

Это простое изменение "дискоцентричности" на "памятицентричность" имеет важные следствия для способа работы СУБД.

- В обычной базе данных данные в памяти представляют собой копию "реальных" данных на диске, и эта копия становится неактуальной при обновлении данных другими процессорами или системой. СУБД должна принимать меры для гарантии обновления данных в памяти и обновлять их с диска в случае неактуальности. В случае базы данных в памяти в оперативной памяти компьютера находится активная копия данных, которая всегда актуальна.
- В обычной базе данных размещение данных на диске кардинально снижает производительность. Администратор базы данных должен принимать меры для настройки и оптимизации устройств долговременного хранения, с тем чтобы данные, к которым наиболее часто обращаются, требовали для выборки и обновления минимального количества операций дискового ввода-вывода, которые гораздо медленнее соответствующих операций с памятью. В базах данных в памяти конфигурация устройств хранения данных неважна — скорость обращения к разным местам оперативной памяти в большинстве случаев одинакова.
- В силу важности конфигурации физического диска в обычной базе данных она постоянно занимается перераспределением данных, разделяя данные в одном дисковом блоке на несколько блоков и обновляя структуры внутренних данных, отслеживающие эти перемещения. Базы данных в памяти не требуют никаких перемещений данных в памяти.
- Поскольку физическое размещение строки базы данных на диске может изменяться, обычная база данных отслеживает строки с помощью виртуальных адресов или идентификаторов строк. Всякий раз при физическом обращении к строке база данных выполняет внутренний поиск ее реального физического расположения.
- В обычных базах данных администраторы и пользователи рассматривают отказоустойчивость как данность, поскольку обычные СУБД разработаны так, чтобы при записи обновлений сначала выполнять запись в журнальный файл, а журналы обычно имеют зеркальные копии на несколь-

ких физических устройствах. Но в случае баз данных в памяти это не так, поскольку случайное отключение питания или сбой сервера легко приводят к потере изменений, сделанных со времени последней записи на диск. Отказоустойчивость в базах данных в памяти может быть достигнута при помощи таких методов, как избыточное копирование, системы бесперебойного питания и энергонезависимая память, но они требуют усилий по планированию и проектированию.

Практическое влияние этих отличий можно проиллюстрировать, рассмотрев простую операцию базы данных — поиск строки, на которую указывает внешний ключ. В учебной базе данных это может быть переход от определенной строки таблицы ORDERS к соответствующей строке таблицы PRODUCTS для заказанного товара. В базе данных в памяти это очень простая операция. Внешний ключ в такой базе данных представляет собой обычный указатель на место в памяти, где хранится соответствующая строка PRODUCTS. Переход по такому указателю требует буквально нескольких процессорных команд, а необходимое для их выполнения время измеряется микросекундами или даже наносекундами.

В обычной базе данных ситуация гораздо сложнее. Внешний ключ, вероятно, представлен значениями данных производителя и идентификатора товара. База данных должна использовать свою схему индексации для обнаружения соответствующей строки PRODUCTS. В больших базах данных с десятками тысяч товаров индекс может иметь несколько уровней. Если дисковый блок с индексом верхнего уровня находится в оперативной памяти, СУБД может сразу начать перемещение вниз по уровням индекса. Если нет — сначала требуется обнаружить блок данных на диске и загрузить его в память. То же самое повторяется на каждом уровне индекса, причем для поиска конкретной записи в блоке требуется проведение вычислений. Наконец, поиск завершен, и виртуальный идентификатор искомой строки найден.

Теперь тот же процесс повторяется — на этот раз для обнаружения самой строки с данными. СУБД транслирует виртуальный идентификатор строки в физическое местоположение — вновь при помощи интенсивных вычислений. Просматриваются текущие блоки данных в буферах памяти СУБД для выяснения, не находятся ли требуемые данные уже в оперативной памяти. Если да, СУБД должна проверить актуальность этих данных — возможно, содержимое этого блока уже не актуально из-за обновлений, внесенных другой программой. Если данные в блоке актуальны, с ними можно работать. Если данные не актуальны или их нет в оперативной памяти — требуется считать содержимое блока с диска. Однако сначала СУБД должна выбрать блок в памяти, который будет заменен новым блоком, для чего требуется проведение новых вычислений в соответствии с принятым в базе данных алгоритмом. Если этот выбранный блок был обновлен, его сначала требуется записать на диск. Только после этого можно загружать новый блок.

Но и это еще не все. Такой двух- или четырехкилобайтовый блок, скорее всего, содержит несколько строк таблицы PRODUCTS, так что следует вычислить точное местоположение строки. После этого СУБД может выбрать строку и скопировать ее в другое место в памяти, где с ней можно будет работать.

Даже если традиционной СУБД и не потребуется выполнять никакие описанные дисковые операции, *возможность* дискового ввода-вывода требует выполнения в лучшем случае сотен тысяч команд процессора. Такое отличие в сложности при-

водит к разительной разнице в задержке между базами данных в памяти и традиционными базами данных даже в том случае, когда последним не требуется обращаться к диску.

Реализация баз данных в памяти

Первые коммерческие базы данных в памяти появились в конце 1990-х годов после десятилетия теоретических изысканий. Одним из преимуществ этих продуктов является возможность применения того же стандартного SQL, что и в традиционных базах данных уровня предприятия. Программисты, которые уже знакомы с разработкой приложений для DB2 или Oracle, могут легко перенести свои проекты для баз данных в памяти. Это преимущество и гибкость реляционных баз данных на основе SQL, по сравнению с жесткостью ранних патентованных баз данных в памяти, сделало их популярным выбором для новых телекоммуникационных и финансовых приложений в начале следующего десятилетия.

В средине десятилетия популярность баз данных в памяти привлекла внимание крупных производителей. В 2005 году компания Oracle приобрела одного из лидеров рынка — компанию TimesTen. В ответ подразделение баз данных IBM через полтора года приобрело другого игрока на том же поле — Solid Data Systems. Но хотя ведущие производители баз данных в памяти и оказались скуплены на корню, эти базы данных остаются отдельной нишей рынка.

Кеширование с базами данных в памяти

Во многих приложениях, требующих низкого времени задержки, характерного для баз данных в памяти, используемые ими данные связаны с другими данными, находящимися в гораздо большей обычной базе данных уровня предприятия. Например, в приложениях мобильной связи сетевая информация реального времени о мобильных телефонах, их расположении и текущих операциях связана с данными о клиентах, их тарифных планах и т.п. Для получения полного представления о клиенте в реальном времени информация в базе данных в памяти и информация в базе данных уровня предприятия должны быть связаны.

В последнее время базы данных в памяти эволюционировали так, чтобы отвечать указанным требованиям в качестве высокопроизводительного кеша обычной базы данных уровня предприятия. Такая кеширующая архитектура показана на рис. 26.1. Запросы реального времени к базе данных удовлетворяются кешем в памяти, в то время как традиционная база данных работает с информацией на диске. И Oracle, и IBM предлагают такие кеширующие конфигурации после приобретений производителей баз данных в памяти.

Кеширование при помощи базы данных в памяти, предлагаемое в настоящее время, обеспечивает согласованный SQL-доступ ко всем базам данных. Кеш обеспечивает определенный уровень прозрачности для SQL-программиста, которому нет необходимости знать, в какой именно системе расположены данные. Однако на сегодняшний день такие кеши не являются прозрачными для администратора баз данных. Администратор должен тщательно выбирать таблицы, которые будут

кешированы в памяти, чтобы запросы реального времени могли быть удовлетворены без обращения к основной базе данных. Такие решения требуют тщательного сопоставления роста производительности за счет кеширования и накладных расходов на синхронизацию баз данных для поддержания целостности данных. Похоже, что такие двухуровневые архитектуры приобретут популярность в области поддержки крупных веб-сайтов и для решения других интернет-задач. В связи с этим важную роль играют повышение уровня прозрачности и большая "автоматическая интеллектуальность" кеширования.

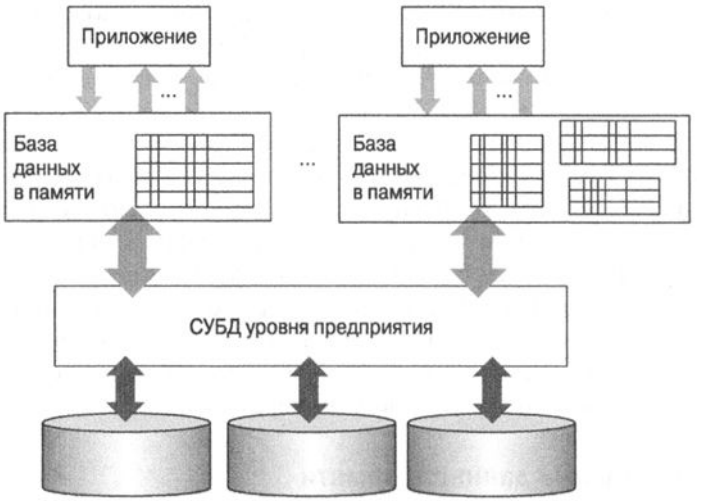


Рис. 26.1. Кеширование базой данных в памяти

Сложные базы данных для обработки событий и потоковые базы данных

Некоторые важные финансовые приложения и системы реального времени включают обработку потока данных из некоторого источника. Например, в биржевых приложениях различные фондовые биржи передают непрерывный поток котировок (предложений о продаже или покупке ценных бумаг) и завершенных транзакций, которые должны быть получены и проанализированы в автоматическом режиме. В военном приложении реального времени различными радарами и сенсорами генерируется поток данных о перемещениях войск и возникающих угрозах, который также должен быть получен и проанализирован для своевременного принятия необходимых решений. В этих примерах и подобных им отдельные сообщения в потоке данных представляют события реального мира, такие как котировки акций или перемещения войск. По этой причине такие приложения часто именуют приложениями обработки потока или *обработки событий*.

Архитектура обычной базы данных уровня предприятия работает с приложениями обработки событий, сначала получая поток события и сохраняя его на диске как последовательность строк таблицы (или таблиц) базы данных, как показано

на рис. 26.2. После того как данные сохранены, требуемый анализ может быть выполнен при помощи одного или нескольких запросов. Если приложение требует постоянного, непрерывного, анализа данных по мере их поступления, оно будет повторять запросы снова и снова, возможно, каждые несколько секунд. Каждый последующий запрос будет использовать вновь добавленные в таблицу данные и отражать их в результатах запроса. Если приложение должно работать только с самыми свежими данными, то может быть добавлен еще один процесс, удаляющий из таблицы устаревшие данные.

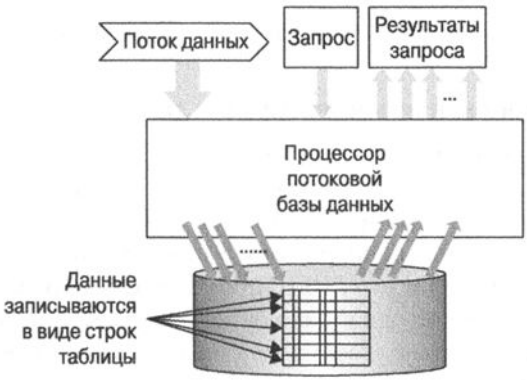


Рис. 26.2. Обработка потока данных обычной базой данных

Как показано на рисунке, обычные базы данных предназначены для запросов данных "в состоянии покоя", т.е. данных в том виде, в котором они имеются в памяти компьютера. В приложениях для обработки событий зачастую реальной целью является анализ и резюмирование данных "в движении", т.е. потока данных, поступающих по сети. Зачастую приложение интересуют не отдельные события в потоке, а их вклад в среднее или сумму в реальном времени или когда они отклоняются от некоторого нормального значения. Для таких приложений сохранение данных на диске — бессмысленное занятие. В некоторых ситуациях дисковые операции оказываются узким местом, ограничивающим скорость обработки поступающих событий. В результате события могут быть пропущены, что может вылиться в проигранную битву или, что еще страшнее, — в упущенную прибыль.

В конце 1990-х–начале 2000-х годов начались исследования задачи обработки событий и построения прототипов различных баз данных для непосредственной обработки данных "в движении". Такие потоковые базы данных рассматривают данные событий на лету, без сохранения на диск, как показано на рис. 26.3. Запросы выполняются непрерывно, генерируя равномерный поток данных об изменениях среднего значения или суммы или отмечая отдельные события как аномальные и требующие последующего изучения. В некоторых из этих систем поток данных может комбинироваться с данными таблиц обычной базы данных, которая может содержать, например, референтные значения для сравнения с данными из потока. В результате получается гибридная база данных, объединяющая обработку событий и обычную базу данных.

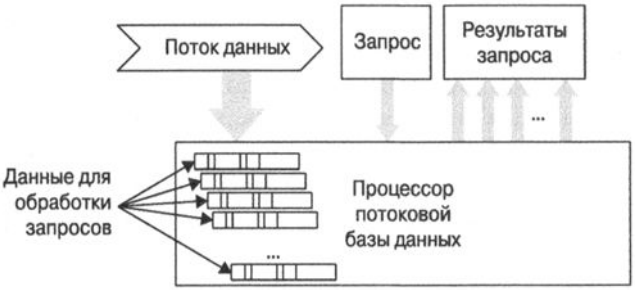


Рис. 26.3. Обработка потока данных потоковой базой данных

Непрерывные запросы в потоковых базах данных

Концепция *непрерывного запроса* является фундаментальной для операций потоковых баз данных. В отличие от запроса традиционной базы данных, который получает и вычисляет определенные данные на основе сохраненных на диске данных за определенное время, непрерывный запрос выполняется многократно, по мере поступления данных в потоке. Большинство баз данных предлагает возможность указать либо *временные* окна, либо окна *записей*. Кроме того, можно выбрать *скользящее* или *прыгающее* окно. Работа непрерывных запросов и окон проиллюстрирована на рис. 26.4.

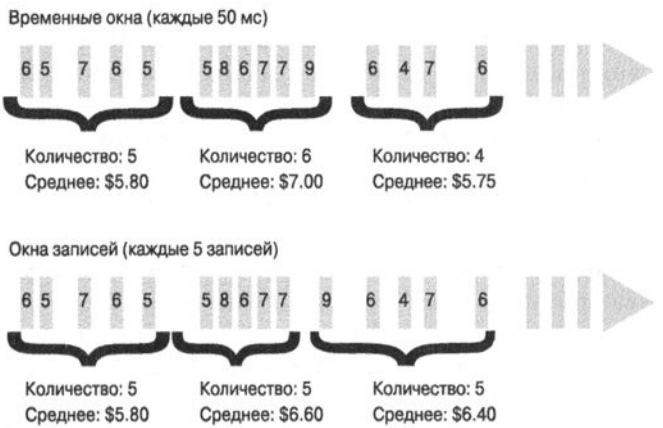


Рис. 26.4. Окна непрерывных запросов

При использовании временных окон от потоковой базы данных можно, например, потребовать вычислять среднюю цену с пятиминутным интервалом. Последовательность значений запросов во времени представляет состояние фондовой биржи в процессе торгов. Можно воспользоваться окнами записей, когда база данных вычисляет среднюю цену для каждой группы из 100 сделок. И вновь последовательность значений запросов представляет состояние фондовой биржи в процессе торгов, но в этот раз не во времени, а по сделкам. Во всех случаях результаты запроса чаще всего представляют собой статистические вычисления числовых данных (суммы, средние, максимум, минимум, стандартное отклонение и т.п.) в

пределах окна, возможно, сгруппированные. Бывает полезно комбинировать эти динамически вычисляемые значения со статической информацией из традиционной базы данных.

Реализации потоковых баз данных

Некоторые из ранних работ в области потоковых баз данных были проведены университетскими исследователями из МТИ, университетов Брауна и Брандейса, возглавляемыми доктором Майклом Стоунбрейкером (Michael Stonebreaker), основоположником технологий реляционных баз данных. В 2003 году Стоунбрейкер и некоторые из его сотрудников основали компанию StreamBase Systems.

В Калифорнийском университете в Беркли также велись работы над потоковыми базами данных, как в свое время там же велись работы над ранними технологиями баз данных. Здесь исследования возглавил доктор Майкл Франклин (Michael Franklin). Его исследования базировались на базе данных с открытым кодом Postgres (по иронии судьбы в свое время разработанной одной из предыдущих команд Стоунбрейкера). В 2005 году исследователи из Беркли тоже основали свою компанию, занимающуюся потоковыми базами данных, — Truviso.

В Великобритании десятилетие исследований в Кембриджском университете привело к основанию в 1999 году компании Apama, которая сосредоточила основные усилия на уровне приложений. Их основой в плане обработки событий стали два других продукта, ориентированные на обобщенную обработку сложных событий (Complex Event Processing, CEP) и на мониторинг бизнес-активности (Business Activity Monitoring, BAM). В 2005 году компания Progress Software приобрела Apama в качестве дополнения к своей объектной базе данных ObjectStore. Двумя другими игроками на рынке были Coral8, основанная в 2003 году в Монтейн Вью, Калифорния, и Aleri, слившиеся в 2009 году.

Все участники рынка, в первую очередь, ориентировались на приложения для рынка ценных бумаг, поскольку тут очевидны требования, которые могут быть удовлетворены потоковыми базами данных, а кроме того, именно тут можно получить максимальную прибыль от применения CEP-приложений. Все они добились определенного успеха в веб-бизнесе и телекоммуникациях, что рассматривается как большой потенциал для завоевания рынка, но пока что применение такого программного обеспечения за пределами Уолл-стрит и правительственных организаций весьма ограничено.

Компоненты потоковых баз данных

Рынок потоковых баз данных развился до той точки, когда все производители предлагают сходные наборы возможностей. На рис. 26.5 показаны основные элементы потоковых баз данных, обычно включающие следующее.

- **Процессор обработки событий** принимает входные сообщения от потока данных или из сети и выполняет непрерывные запросы к получаемым данным. Процессор может включать возможность комбинации потока данных с данными из статических таблиц или из статических данных, управляемых традиционной СУБД уровня предприятия.

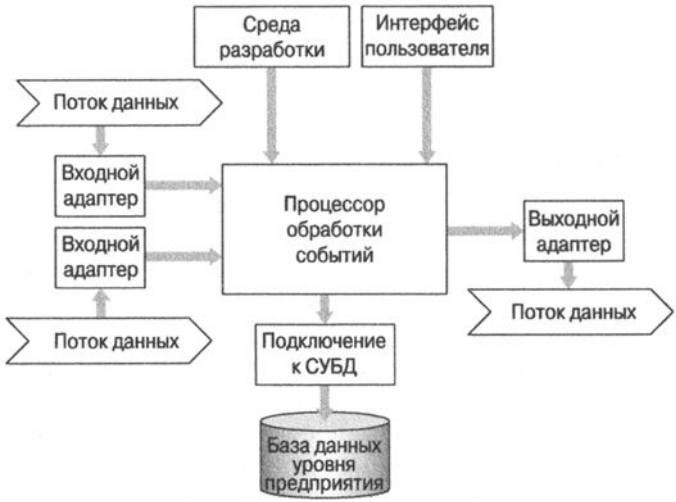


Рис. 26.5. Элементы потоковой базы данных

- Набор **входных адаптеров** принимает входные сообщения от потоков данных и передает их процессору обработки событий в стандартном формате. Обычно эти адаптеры поддерживают Java Messaging Service (JMS) API, популярную службу сообщений Rendezvous компании Tibco, различные потоки от финансовых рынков и другие системы сообщений.
- Набор **выходных адаптеров** принимает сообщения, прошедшие через процессор обработки событий или генерируемые им, и преобразует их для передачи распространенным системам сообщений, таким как JMS-совместимые системы или Rendezvous.
- **Подключение к СУБД** позволяет процессору обработки событий соединять данные из потока с данными из традиционных баз данных. Все основные продукты поддерживают обращение к данным посредством JDBC, а некоторые обеспечивают поддержку частных API конкретных баз данных уровня предприятия.
- **Среда разработки** позволяет программистам создавать непрерывные запросы, а также тестировать и развертывать их в качестве производственных систем. Некоторые производители предлагают графические среды разработки для определения фильтрации данных, их слияния, соединения и группирования. Другие предоставляют текстовый язык на базе SQL для создания запросов. Среда разработки может также поддерживать *язык моделирования* для определения форматов различных потоков данных и того, как данные должны обрабатываться процессором.
- **Интерфейс пользователя** обеспечивает графическое представление результатов запросов, выводит средние значения, итоговые величины, исключения и тому подобное в реальном времени, по ходу обработки потоковых данных процессором.

Встраиваемые базы данных

Некоторые базы данных совершенно невидимы конечному пользователю, так как они глубоко спрятаны в машинах или устройствах и используются для поддержки их операций или управления ими. Например, оборудование, управляющее производственным процессом или автоматизированной погрузкой, может содержать базу данных, которая помогает управлять сборочным конвейером или погрузчиком. Сетевой элемент, такой как роутер, коммуникатор или автоответчик, могут содержать базу данных, которая хранит параметры настройки, собирает статистику производительности или определяет действия в ответ на команды пользователя. Система управления автомобилем может содержать базу данных с информацией о ваших любимых радиостанциях или собирать информацию о функционировании двигателя. В каждом из описанных случаев работа базы данных является основой для функционирования устройства, но при этом никакие запросы к базе данных пользователю не видны, а работой базы данных не управляет никакой администратор.

Еще десять лет назад встраиваемые базы данных для поддержки приложений наподобие описанных всегда представляли собой "самоделки", отвечающие конкретным нуждам приложения. Встраиваемые базы данных практически всегда работают в условиях ограниченных ресурсов, с небольшим количеством оперативной памяти и небольшим дисковым пространством (или вовсе без него). Тогда просто не было никакой возможности строить такую систему на базе SQL — для этого просто не было ресурсов, а обобщенность базы данных на основе SQL существенно превосходила требования приложения. Однако со временем стоимость памяти, дискового пространства и процессорных мощностей постоянно падала, а сложность и интеллектуальность автоматизированных систем, сетевого оборудования и т.п. существенно выросла. Эти две тенденции привели к появлению коммерческих встраиваемых SQL-баз данных.

Характеристики встраиваемых баз данных

Коммерческие встраиваемые SQL-базы данных обычно обладают одними и теми же базовыми характеристиками, определяемыми требованиями обслуживаемых ими приложений. Сюда входит следующее.

- **Малое потребление памяти**. В то время как базы данных уровня предприятия требуют для работы десятков гигабайт (а то и больше) памяти, встраиваемые базы данных могут обходиться несколькими сотнями килобайт.
- **Отсутствие администрирования**. В то время как база данных уровня предприятия требует наличия отдельной должности администратора, который конфигурирует базу, управляет ею и настраивает ее, встраиваемая база данных управляется использующим ее приложением. Должность администратора отсутствует, а при включении продукта с такой базой данных в работу обычно требуется минимальное конфигурирование (либо не требуется и оно).
- **Поддержка нетрадиционного хранения**. Данные, которыми управляет встраиваемая база данных, могут храниться в оперативной памяти,

в энергонезависимой памяти, на USB-диске или еще где-то, помимо традиционных дисков, обычно используемых серверами баз данных.

- **Ограниченная поддержка SQL**. При существенно сниженном количестве памяти поддержка SQL обычно ограничена базовыми манипуляциями данными и запросами. Приложению вряд ли потребуются расширения хранилища данных, экзотические типы данных, интеграция с XML или аудит базы данных.
- **Статическая схема базы данных**. Структура базы данных разрабатывается для конкретного приложения и может быть определена еще во время проектирования продукта. Потребность в динамическом добавлении, удалении или изменении определений столбцов или таблиц крайне мала.
- **Однопользовательская работа.** Обычно базу данных использует одно приложение (или небольшая группа приложений), так что нет необходимости в сложной многопользовательской работе с базой данных.

Реализации встраиваемых баз данных

В 1990-х годах наблюдалось распространение встраиваемых баз данных, определявшееся продуктами, в которых требовалось встроенное управление данными, а также повышением мощности процессоров и увеличением количества оперативной памяти. Некоторые из таких баз данных оказались результатом университетских исследований, такие как встраиваемая база данных SleepyCat на базе BerkeleyDB. В Канаде пионером в разработках встраиваемых баз данных была Empress Software. Еще одной ранней разработкой была база данных Raima, впоследствии приобретенная компанией Birdstep. Разработки компании Progress Software включают встраиваемую базу данных OpenEdge. Компания Encirq разработала одну из встраиваемых баз данных с минимальным потреблением памяти.

В области баз данных с открытым кодом также появились свои разработки. Одним из наиболее популярных на сегодня продуктов является база данных SQLite, реализованная как продукт с открытым кодом. База данных с открытым кодом MySQL, хотя она и больше других встраиваемых баз данных, играет важную роль в этой области рынка. Одной из наиболее привлекательных черт баз данных с открытым кодом является то, что разработчик, при желании, может настроить процессор базы данных для своих нужд, отбросив все лишнее и сократив тем самым потребляемые базой данных ресурсы.

Мобильные базы данных

Еще одним, четвертым, типом специализированных баз данных являются базы данных, разработанные специально для нужд переносных устройств, таких как наладонники, органайзеры, смартфоны, лэптопы и т.п. Поскольку эти устройства работают на аккумуляторах, их ресурсы обычно существенно более скромны, чем у серверов, на которых работают базы данных уровня предприятия. Обычно они менее мощные и обладают более медленными процессорами, весьма ограниченным количеством памяти — как оперативной, так и дисковой (причем последней может не быть вообще). За последнее десятилетие резко выросла популярность лэптопов и ноутбуков,

появились совершенно новые категории устройств. Все это привело к разработкам мобильных SQL-баз данных для поддержки таких устройств и их приложений.

Роли мобильных баз данных

Мобильные базы данных обычно играют одну или несколько четко определенных ролей в поддерживаемых устройствах.

- **Поддержка работы устройства**. База данных может хранить данные конфигурации или настройки текущего пользователя. Такая роль обычно скрыта от пользователя и использует мобильную базу данных как встраиваемую.
- **Поддержка встроенных приложений**. В переносных компьютерах персональный календарь или рабочий дневник могут храниться в локальной базе данных, что облегчает поиск в них. В смартфоне список контактов и их адресов, номеров телефонов и тому подобного также может храниться в базе данных. Такая база данных может быть скрыта от пользователя, но видима приложениям.
- **Поддержка мобильных приложений**. Приложения уровня предприятия могут использовать переносные компьютеры в качестве устройств для накопления или анализа данных, в которых пользователь вводит данные с применением удобного пользовательского интерфейса.
- **Работа с локальными базами данных**. Иногда пользователю переносного компьютера может потребоваться персональная база данных для хранения информации или ее анализа.
- **Доступ к базам данных уровня предприятия**. Мобильные устройства могут служить порталами для доступа пользователя к данным, хранящимся в базе данных предприятия. Программное обеспечение локальной базы данных принимает запросы на обращение к данным, передает их базе данных предприятия, получает результаты и передает их назад локальному программному обеспечению мобильного устройства.

Синхронизация с базой данных предприятия — ключевая характеристика ведущих мобильных баз данных. Когда сеть недоступна, мобильная база данных поддерживает автономное функционирование приложений устройства. Когда же сеть становится доступна, мобильная база данных выполняет интеллектуальную синхронизацию, загружая изменения, сделанные в локальной базе данных, получая изменения из базы данных предприятия и разрешая конфликты между ними. Кроме того, когда сеть становится доступна, мобильная база данных может выступать в роли интеллектуального кеша, храня локальную копию данных, к которым часто выполняется обращение, чтобы избежать излишней нагрузки на сеть.

Реализации мобильных баз данных

Наибольшим успехом на рынке мобильных баз данных пользуется SQLAnywhere компании Sybase, в первую очередь благодаря ее распространенности на лэптопах. Этот продукт предлагает удачный баланс между полноценной поддержкой SQL и от-

носительно скромными требованиями к ресурсам. Он обеспечивает синхронизацию данных с базой данных предприятия посредством API, являющихся промышленным стандартом (ODBC и JDBC), так что возможна его интеграция практически со всеми коммерческими базами данных.

Sybase агрессивно предлагает SQLAnywhere разработчикам приложений для мобильных и промышленных приложений в качестве мобильной базы данных, которая легко может быть встроена в мобильные версии их продуктов или распространяться вместе с ними. Даже при отсутствии встраивания мобильные приложения стремятся поддерживать SQLAnywhere из-за популярности этой базы данных. Этот продукт является частью всеобщей политики ориентации Sybase на мобильные приложения.

Oracle в этой области предлагает базу данных Oracle Lite, которая поставляется как дополнение к базе данных уровня приложения той же компании. Oracle Lite работает на лэптопах и других избранных мобильных устройствах. Сервер синхронизации от Oracle обеспечивает связь Oracle Lite с базами данных Oracle и другими базами данных уровня предприятия при помощи ODBC/JDBC. Многие годы Oracle не обращала особого внимания на Oracle Lite, и в результате последняя потеряла свои позиции на рынке, уступив SQLAnywhere. Однако в последнее время компания Oracle изменила свою политику и начала пропагандировать Oracle Lite разработчикам приложений и крупным клиентам, которым требуются мобильные решения. Этот продукт служит также встраиваемой базой данных, поддерживая такие устройства, как разное промышленное оборудование и торговые автоматы.

Резюме

Хотя на рынке доминируют базы данных уровня предприятия, свои важные ниши имеются и у специализированных баз данных.

- Базы данных в памяти обслуживают приложения, которым требуются очень малые задержки в обработке запросов, например, в области телекоммуникаций или биржевых торгов.
- Встраиваемые базы данных обслуживают приложения, которые работают в условиях ограниченных ресурсов — например, в автомобильных системах, недорогом сетевом оборудовании и т.п.
- Потоковые базы данных обслуживают приложения, имеющие дело с большим потоком данных, который должен непрерывно обрабатываться и анализироваться, например, с информацией от радаров или биржевой информацией.
- Мобильные базы данных обеспечивают потребности органайзеров, наладонников, смартфонов и прочих переносных устройств, обеспечивая как работу локальной базы данных, так и синхронизацию с базой данных предприятия.
- Могут возникнуть новые области, в которых потребуются свои средства управления данными. Язык SQL обладает достаточной универсальностью и гибкостью, чтобы приспособиться к любым новым требованиям и остаться стандартным языком для любых специализированных баз данных.

27
ГЛАВА

Будущее SQL

Язык SQL и основанные на нем реляционные базы данных находятся среди важнейших технологий современной информационной индустрии. Со времени своего первого появления, около тридцати лет назад, SQL прошел большой путь и стал *стандартным* языком для работы с базами данных. В первое десятилетие своей истории, благодаря поддержке со стороны компании IBM, производителей СУБД и международных органов стандартизации, он стал стандартом управления данными в корпоративной среде. Во втором десятилетии влияние SQL расширилось на рынок персональных компьютеров, а также на новые сегменты рынка, такие как хранилища данных. Третье десятилетие SQL сделало его основой управления данными в интернет-вычислениях и создало новые многомиллиардные ниши рынка с его участием типа интеллектуальных ресурсов предприятия. Приведенные ниже факты ясно свидетельствуют о большом значении SQL.

- Компания Oracle, второй по величине производитель программного обеспечения, была основана на волне популярности реляционной модели управления данными, и SQL поддерживается во всех ее флагманских продуктах и приложениях уровня предприятия.
- Компания IBM, крупнейший производитель на компьютерном рынке, предлагает реляционную СУБД DB2 для всех линий своих продуктов, а также для конкурирующих платформ.
- Компания Microsoft, крупнейший производитель программного обеспечения, рассматривает свою СУБД SQL Server в качестве центрального звена в стратегии завоевания рынка, а также в качестве средства управления данными для всех своих служб и приложений.
- Компания Sun Microsystems увидела в SQL достаточную ценность для того, чтобы купить MySQL AG, тем самым приобретя наиболее популярную в мире РСУБД с открытым кодом MySQL, которая обслуживает огромное количество веб-сайтов в Интернете.

- Потенциальный конкурент SQL и реляционной модели, язык XML, был внедрен во многие SQL-продукты для управления данными, что только увеличило доминирование SQL на рынке.
- Большинство прикладных корпоративных систем (ПО планирования и учета, финансовые, маркетинговые и торговые приложения) основано на реляционных базах данных.
- SQL является стандартом для специализированных баз данных для мобильных устройств и лэптопов, а также для встраиваемых приложений в области телекоммуникаций и производственных систем.
- Доступ к базам данных на основе SQL представляет собой неотъемлемую часть серверов приложений в Интернете, а SQL-базы данных используются всеми крупными сайтами электронной коммерции, от Amazon до eBay.

В настоящей главе описываются современные тенденции развития баз данных и анализируется влияние этих тенденций как на язык SQL, так и на весь компьютерный рынок в ближайшие годы.

Тенденции на рынке баз данных

Современный рынок СУБД измеряется десятками миллиардов долларов. Это во многом достаточно устоявшийся рынок, ведущими поставщиками на котором являются Oracle, IBM и Microsoft и рост которого обычно выражается однозначным (изредка — небольшим двузначным) числом процентов в год. Однако это вовсе не означает застоя — в новые проекты вкладываются огромные деньги, исчисляемые сотнями миллионов долларов в год, мелкие компании, пытающиеся выйти на рынок, поглощаются крупными акулами. В крупных компаниях тысячи разработчиков трудятся над расширением и усовершенствованием имеющихся продуктов. Наблюдается рост специализированных категорий баз данных — встраиваемых, потоковых и многих других. Основными тенденциями современного рынка являются, с одной стороны, насыщение и консолидация рынка СУБД, а с другой, — появление разнообразных технологических решений и новых приложений.

Насыщение рынка корпоративных баз данных

Реляционная технология составляет ядро корпоративных систем обработки данных, и реляционные базы данных используются практически во всех крупных компаниях. Поскольку корпоративные базы данных чрезвычайно важны и многие из них эксплуатируются уже в течение многих лет, большинство компаний выбрали какую-то одну СУБД в качестве собственного корпоративного стандарта. После того как такой стандарт формально установился в компании, переход на другую СУБД становится проблематичным. Хотя конкурирующие СУБД могут работать лучше в некоторых ситуациях или предлагать новые полезные возможности, достаточно заявления поставщика "стандартной" СУБД о том, что эти возможности будут включены в следующую версию продукта, и потери клиентов не произойдет.

Все это привело к усилению позиций ведущих поставщиков СУБД. Наличие служб прямой доставки и технической поддержки, а также многолетние соглашения о сотрудничестве теперь играют такую же, если не более важную, роль, что и технологические преимущества. В связи с этим лидеры рынка СУБД стремятся наращивать свой бизнес в рамках инсталлированной базы собственных продуктов, а не пытаться привлечь клиентов своих конкурентов.

Важным фактором является тенденция к консолидации рынка. В общих чертах этот процесс можно описать следующим образом. Первопроходцами новой технологии становятся, как правило, начинающие компании, которые растут за счет продажи своих продуктов приверженцам всего нового. Последние помогают сформировать технологию и определить ниши рынка, в которых она может приносить прибыль. Через несколько лет, когда преимущества новой технологии становятся очевидными, компании-новаторы поглощаются крупными поставщиками СУБД, которые распространяют технологию на инсталлированную базу своих продуктов. В первой половине 90-х годов описанную схему можно было наблюдать в отношении поставщиков СУБД и разработчиков различных утилит и инструментов для баз данных. Во второй половине 90-х годов поглощаться начали разработчики специализированных баз данных, таких как хранилища данных или объектно-реляционные базы данных. В средине 2000-х годов Oracle и IBM приобрели производителей баз данных в памяти, а Sun приобрела MySQL AG, крупнейшего производителя базы данных с открытым кодом. Следующей мишенью должны стать специализированные хранилища данных.

Сегментация рынка СУБД

Несмотря на насыщение отдельных сегментов рынка (особенно в сфере корпоративных систем), постоянно формируются его новые сегменты и ниши. Традиционно рынок делился на базы данных для мэйнфремов (где доминировала IBM), баз данных для информационных центров (Oracle) и баз данных для рабочих групп (Microsoft). Сегодня рынок более многолик и точнее сегментирован с учетом сферы применения СУБД и ее специализированных возможностей. Новые и наиболее быстро развивающиеся сегменты рынка включают следующее:

- хранилища данных, ориентированные на управление терабайтами информации;
- интеллектуальные ресурсы предприятий и базы данных для оперативного анализа (OLAP), предназначенные для выполнения сложного анализа данных на предмет выявления скрытых тенденций развития;
- мобильные базы данных, которые предназначены для "мобильных" сотрудников, таких как продавцы на местах, консультанты и т.п. Такие мобильные базы данных могут связываться с центральным сервером для синхронизации с центральной базой данных;
- встраиваемые базы данных, работающие в закрытых системах, таких как производственное оборудование, системы управления автомобилей или сетевое оборудование. Такие базы данных нетребовательны к ресурсам и требуют минимального администрирования (или обходятся без него);

- базы данных в памяти и кеши баз данных, разработанные для приложений со сверхвысокими требованиями ко времени отклика;
- базы данных для истории посещений и записи активности миллионов пользователей веб-сайтов, позволяющие оптимизировать веб-сайты на основе анализа реального поведения пользователей;
- кластеризованные базы данных, использующие преимущества параллельной работы мощных дешевых серверов для повышения масштабируемости и надежности;
- потоковые базы данных, работающие с передаваемыми по сети динамическими данными, такими как сетевой трафик, биржевые котировки или банковские транзакции.

Пакеты корпоративных приложений

В начале эры SQL большинство корпоративных приложений для повседневных нужд разрабатывалось собственными силами информационных служб предприятий. Сегодня большинство компаний отказались от стратегии *"сделай"* в пользу стратегии *"купи"* в отношении большинства корпоративных приложений, таких как системы планирования ресурсов предприятий (Enterprise Resource Planning — ERP), системы управления поставками (Supply Chain Management — SCM), системы управления человеческими ресурсами (Human Resource Management — HRM), системы автоматизации торговых операций (Sales Force Automation — SFA), системы обслуживания клиентов и др. Все эти системы в настоящее время поставляются в виде комплексных программных пакетов, обеспеченных обслуживанием, технической поддержкой и консультациями. И во всех них используются реляционные базы данных.

Появление комплексных корпоративных систем оказало существенное влияние на развитие рынка баз данных. Большинство подобных систем поддерживало СУБД лишь двух или трех ведущих поставщиков. Это привело к еще большему усилению позиций ведущих СУБД и усложнило жизнь начинающим компаниям. Другим следствием стало снижение цен на СУБД, которые теперь стали чаще рассматриваться как составная часть более крупного программного пакета, а не как отдельный компонент. Например, если клиент планирует установить в своей компании некоторое программное обеспечение, скажем, для выполнения структурного анализа, то он вынужден приобретать и базу данных, причем она должна быть из ограниченного списка тех баз данных, которые поддерживаются приобретаемым программным обеспечением. В результате доминирующие позиции главных производителей баз данных на рынке только укрепляются, а возможность выхода на рынок для новичков становится все более призрачной. В результате складывается тенденция лидирования на рынке больших корпоративных баз данных IBM и Oracle и сдерживания принятия SQL Server, Sybase и MySQL корпоративными центрами данных. Сами СУБД все больше рассматриваются не как отдельное стратегическое решение, а как компонент приложения, приобретаемого для решения тех или иных корпоративных задач.

В ответ на эти тенденции и насыщение рынка Oracle стремится занять позиции лидера не только в области разработчика баз данных, но и производителя корпо-

ративных приложений. Сначала компания Oracle разрабатывала собственные корпоративные приложения, которые имели на рынке ограниченный успех. Изменив стратегию, в последнее десятилетие Oracle строит свой бизнес на затратах десятков миллиардов долларов на приобретение компаний по разработке программного обеспечения (и их клиентов). IBM проявляет такую же активность, но сторонится многих важных категорий программного обеспечения.

Взаимосвязи между корпоративными приложениями и корпоративными базами данных все еще присутствуют на рынке. Oracle рекламирует преимущества принципа "все в одном" для всех видов программного обеспечения, постоянно обещая (но далеко не всегда выполняя обещанное) более тесную интеграцию между корпоративными приложениями в широком диапазоне. Программное обеспечение, приобретаемое Oracle, имеет тенденцию со временем поддерживать работу только с базами данных Oracle, тем самым уменьшая возможность выбора клиентами средств управления данными. IBM же приняла консультационно-сервисный подход, в первую очередь предлагая клиентам помощь и консультации в выборе лучших из доступных продуктов. Злые языки, впрочем, утверждают, что таковыми с точки зрения IBM всегда оказываются именно ее аппаратные и программные продукты (включая базу данных DB2). SAP занимает доминирующие позиции по количеству установленных корпоративных приложений, заполняя свой портфель предложений небольшими точно выверенными покупками. Однако у SAP нет собственной базы данных уровня предприятия, так что приложения этой компании приносят Oracle и IBM сотни миллионов долларов прибыли. Будущее рынка корпоративных приложений пока что остается неясным, но уже понятно, что он будет разделен основными производителями баз данных.

Программное обеспечение в виде служб

На фоне распространения пакетов корпоративных приложений возникла еще одна тенденция — распространение корпоративных приложений через Интернет с доступом к ним при помощи веб-браузера. При использовании такой модели — программного обеспечения в виде службы (Software-as-a-Service, SaaS) — корпоративный отдел информационных технологий не инсталлирует корпоративные приложения и не работает с ними в собственном центре данных. Вместо этого корпоративные приложения работают на серверах их разработчика или на серверах, управляемых им. Потенциальная выгода предприятия от такого решения заключается в сокращении штата отдела информационных технологий, снижении стоимости, быстром доступе к новым, усовершенствованным, версиям программного обеспечения, которое прозрачно для пользователей инсталлируется через Интернет. Пионером в этой области стала компания Salesforce.com, которая начинала с приложений для автоматизации процесса продаж и чей годовой доход вырос до миллиарда долларов. Со временем и другие производители воспользовались этой моделью для распространения офисного программного обеспечения, а также приложений ERP, CRM, HR.

Модель SaaS потенциально в состоянии серьезно повлиять на рынок корпоративных баз данных. При работе с такого рода приложениями конкретная база данных в его основе оказывается невидимой для клиента. Это может быть Oracle,

IBM DB2, SQL Server, MySQL или какая-то иная база данных — до тех пор, пока все корректно работает, пользователя этот вопрос не должен интересовать. Так что распространение модели SaaS грозит тем, что покупка баз данных перестанет быть задачей информационных отделов корпораций и сконцентрируется в руках производителей SaaS-приложений.

На момент написания книги SaaS оставалась небольшой частью корпоративных информационных технологий. Она пока что не распространилась далее определенных видов приложений типа приложений для автоматизации процесса продаж, но уже доказала свою популярность среди мелкого и среднего бизнеса, которому не по карману собственные информационные службы. Экономические выгоды модели SaaS неоспоримы, и компания Salesforce.com храбро заявила, что ее стратегия состоит в том, чтобы покончить с корпоративными приложениями. Если SaaS продолжит свое распространение и начнет проникать в финансовые системы и ERP, производителям основных баз данных придется прикладывать усилия для защиты уже установленных баз и своего места под рыночным солнцем.

Повышение производительности аппаратного обеспечения

Одной из главных причин роста популярности SQL было резкое увеличение производительности реляционных СУБД. Частично этот рост произошел благодаря улучшению технологии разработки баз данных и оптимизации запросов. Однако, в основном, увеличение производительности СУБД явилось результатом общего повышения быстродействия аппаратных средств и изменений в программном обеспечении, которые смогли в полной мере использовать аппаратные новшества. Повышение производительности мэйнфреймов сопровождалось аналогичными процессами на рынке серверов Unix и Windows, где вычислительная мощность удваивается каждый год.

Самые разительные перемены происходят с системами симметричной многопроцессорной обработки (Symmetric Multiprocessing — SMP), в которых рабочая нагрузка распределяется между двумя, четырьмя, восемью и более процессорами, работающими параллельно. Многопроцессорная архитектура особенно подходит для OLTP-приложений, где нагрузкой является большое число маленьких, одновременно выполняемых транзакций в базе данных. Традиционные поставщики OLTP-систем, например компания Tandem, всегда применяли многопроцессорную архитектуру, а в крупнейших системах на базе мэйнфреймов она используется уже больше десяти лет. В начале 90-х годов SMP-системы активно вторглись на рынок серверов Unix, а в текущем десятилетии — стали нормой для серверов на базе персональных компьютеров.

Сегодня даже настольные системы на базе процессоров Intel и AMD оснащены двух- или четырехъядерными процессорами, мощность которых близка к мощности двух или четырех отдельных процессоров. Блейд-серверы развили это направление, сделав высокоэкономичными серверы с десятками процессоров. Принятие на вооружение 64-битовой аппаратной архитектуры и соответствующих операционных систем позволяет поднять объем памяти серверов до десятков и даже сотен гигабайт. Серверы, конкурирующие по мощности с мэйнфреймами, в настоящее время стоят до 100000 долл.

Многоядерные и многопроцессорные системы дают преимущества и для систем поддержки принятия решений и приложений для анализа данных. Производители СУБД ведут интенсивные исследования в области распараллеливания обработки запросов, когда один сложный SQL-запрос разбивается на параллельно выполняемые фрагменты кода. При использовании таких технологий запрос, который требовал двух часов работы в однопроцессорной системе, может быть выполнен буквально за несколько минут. Однако еще более часто увеличение мощности процессоров используется не просто для ускорения вычислений, а для выполнения более сложного и интеллектуального анализа, который был недоступен ранее.

Сегодня пока что нет никаких признаков остановки роста производительности баз данных, достигаемого, в первую очередь, за счет оптимизации СУБД для работы в многопроцессорных и многоядерных системах. В прошлом такой рост достигался, в первую очередь, за счет повышения производительности аппаратного обеспечения. В будущем, вероятно, необходимость повышения производительности баз данных будет оказывать еще большее влияние на развитие нового аппаратного обеспечения, процессоров и серверов.

Специализированные серверы баз данных

История SQL и реляционных баз данных — это еще и история серверов баз данных. Для построения высокоэффективных систем производители комбинировали высокопроизводительные процессоры, быстрые диски, предустановленное программное обеспечение, чтобы в результате получить сервер, который достаточно просто включить в сеть для работы. Производители серверов баз данных обычно доказывают, что при помощи специально спроектированной системы можно достичь существенно более высокой производительности базы данных, чем при применении компьютерной системы общего назначения. В некоторых случаях такие системы включают специализированные интегральные микросхемы (application-specific integrated circuits, ASIC), которые реализуют некоторую логику СУБД на аппаратном уровне, достигая максимально возможной производительности. Специализированные системы от таких компаний, как Teradata, Sharebase (бывш. Britton-Lee) и Netezza, нашли применение в приложениях, включающих сложные запросы к очень большим базам данных (в настоящее время на рынке осталась только компания Teradata).

Идея специализированного сервера баз данных была вновь рассмотрена в конце 1990-х годов компанией Oracle Corporation и ее исполнительным директором Ларри Эллисоном (Larry Ellison). Эллисон утверждал, что эра Интернета доказала успех продуктов "все в одном", таких как сетевое оборудование или кеширующие веб-серверы. Oracle анонсировала сотрудничество с рядом производителей серверного аппаратного обеспечения по созданию специализированного сервера для баз данных Oracle. Однако все их усилия практически не отразились на состоянии рынка, так что постепенно эта работа сошла на нет.

Не так давно к этой идее вновь вернулись некоторые вновь созданные компании — в форме кеширующих серверов баз данных, которые располагаются в сети между приложением и корпоративной базой данных. В такой конфигурации критична абсолютная прозрачность кеша, и появление исключительно популярной

базы данных с открытым кодом MySQL помогло обеспечить эту прозрачность. Многие серверы баз данных работают под управлением MySQL, так что приложение, обращающееся к базе данных, не в состоянии выяснить, работает ли оно со специализированным сервером или с MySQL, запущенной на обычном компьютере. Oracle также возобновила свою работу над специализированным сервером, анонсировав высокопроизводительную базу данных Oracle на аппаратном обеспечении от Hewlett-Packard.

Стандартизация SQL

Принятие официального стандарта ANSI/ISO для языка SQL было одним из главных факторов, благодаря которым SQL в 1980-е годы стал стандартным языком реляционных баз данных. Соответствие стандарту ANSI/ISO стало отправной точкой для оценки реляционных СУБД, поэтому каждый поставщик утверждал, что его программный продукт совместим со стандартом ANSI/ISO. В течение последующих 20 лет все популярные СУБД в действительности стали поддерживать стандарт SQL, по крайней мере, широко употребляемые его части. Другие части, такие как язык модулей, чаще игнорировались, чем реализовывались. Так постепенно популярные СУБД пришли к поддержке базовых возможностей SQL, так что ядро языка SQL в разных СУБД стало практически одинаковым.

Как уже говорилось в главе 3, "Перспективы SQL", исходный стандарт SQL был довольно ограниченным, в нем имелось много пробелов, а также немало областей, оставленных на усмотрение разработчиков СУБД. Несколько лет комитет по стандартизации работал над стандартом SQL2, который устранял эти недостатки и расширял язык SQL. В отличие от первого стандарта, в котором описывались элементы языка, уже имевшиеся в большинстве СУБД, стандарт SQL2 на момент своей публикации в 1992 году был попыткой опередить разработчиков СУБД, а не следовать за ними. В этом стандарте описывались функции и особенности языка, которые еще не использовались широко в современных СУБД. Некоторые из этих возможностей, такие как расширенные возможности соединения и широкое использование подзапросов, сегодня широко реализованы; другие же так и остаются не принятыми большинством производителей.

То же самое произошло и при принятии последующих версий стандарта SQL как в 1999, 2003 годах, так и в последние годы. Стандарт существенно вырос в размерах (более чем в три раза со времени SQL2) и был разделен на десяток частей. В новых разделах, таких как встраивание XML, продолжается состязание между разработками производителей СУБД, которые надеются получить таким образом конкурентные преимущества, и стандартом, обеспечивающим переносимость.

Вероятное будущее развитие стандарта SQL представляется экстраполяцией пути, по которому он следовал последние годы. Ядро языка останется высоко стандартизированным. Все больше возможностей постепенно станут частью этого ядра и будут определены как дополнения к языку (либо для них будут разработаны собственные стандарты). Производители СУБД продолжат добавлять к своим реализациям SQL новые возможности в попытках выделить свои разработки и ответить на новые требования, выдвигаемые рынком; со временем те новые возможности, которые станут наиболее популярны, будут внесены в стандарт.

SQL в следующем десятилетии

Вряд ли сегодня кто-нибудь возьмется предсказать развитие рынка баз данных и SQL в следующие пять–десять лет. Каждая новая технология за последние три десятилетия оказывала существенное влияние на управление данными и SQL. Одним из примеров может быть появление персональных компьютеров и вызванная ими эра "клиент/сервер". Позже появление Интернета и браузерной архитектуры привело к новым методам управления данными — веб-службам. Если заглянуть в будущее, то можно предположить, что Интернет станет поистине вездесущим, связывающим буквально все электронные устройства. Пожалуй, грядущее развитие Интернета окажет на архитектуры управления данными большее влияние, чем само его появление. Тем не менее вполне можно сделать несколько предсказаний о грядущих тенденциях в эволюции управления базами данных (они будут рассмотрены в последнем разделе данной главы).

Распределенные базы данных

В наши дни все большее распространение получают корпоративные приложения и даже еще более масштабные системы. Но одной централизованной базе данных трудно поддерживать десятки крупных приложений и тысячи параллельно работающих пользователей. Поэтому массивные корпоративные базы данных становятся все более распределенными; они разделяются на меньшие базы данных, обслуживающие отдельные приложения и информационные потребности корпорации. Чтобы соответствовать растущим требованиям корпоративных и интернет-приложений, данные должны быть распределенными, но в то же время СУБД должна обеспечивать их целостность и координирование, чтобы гарантировать надежность и эффективность деловых операций.

Еще одной тенденцией, требующей отказа от централизованной архитектуры баз данных, является продолжающееся распространение портативных персональных компьютеров и других мобильных информационных устройств. Эти устройства по своей природе предназначены для работы в распределенных сетях, причем в режиме *непостоянного подключения* — иногда они работают автономно, иногда подключенными к сети, и это подключение может выполняться посредством как проводной, так и беспроводной связи. Базы данных, составляющие ядро мобильных приложений, должны быть способны эффективно работать в таком непостоянном режиме.

Все эти тенденции требуют распределения, интеграции и синхронизации баз данных, а также разработки эффективных технологий управления распределенными данными. Модель "все в одном" больше не подходит для распределенных сред, работающих по принципу "всегда и везде". Вместо этого одни транзакции требуют абсолютной синхронизации с центральной управляющей базой данных, тогда как другим необходима поддержка долго выполняемых операций, когда синхронизация может произойти по прошествии часов и даже дней. Без эффективных специализированных технологий и программных продуктов построение таких распределенных систем становится кошмаром для администраторов баз данных. Разработка подобных технологий и будет одной из важнейших задач производителей СУБД в следующем десятилетии и одним из главных источников их будущих доходов.

Массивные хранилища данных для оптимизации бизнеса

Последние несколько лет показали, что компании, интенсивно использующие технологии баз данных, получают огромные преимущества по сравнению со своими более инертными конкурентами. Например, компания Wal-Mart обязана своим успехом именно использованию информационных технологий для каждодневного учета складских и коммерческих операций. Это позволило компании минимизировать объемы складских запасов и эффективнее организовать взаимодействие с клиентами. Технологии накопления и последующего анализа данных позволяют компаниям выявлять скрытые тенденции и взаимосвязи — стоит только вспомнить легендарное открытие одного розничного торговца, обнаружившего, что продажа детских пеленок поздней ночью напрямую зависит от количества проданного накануне пива.

Поэтому очевидно, что компании будут продолжать накапливать всевозможную информацию о клиентах, продажах, складских операциях, ценах и других бизнес-факторах. Интернет создает новые возможности для сбора этого вида информации. Действия клиента на веб-сайте компании предоставляют информацию о его желаниях, потребностях и поведении. Информация о взаимодействии клиентов с активно посещаемым сайтом компании ежедневно дает гигабайты данных. Базы данных для управления таким огромным количеством данных должны быть способны быстро импортировать новые данные в огромном количестве и быстро выделять их подмножества для анализа. Они должны быть достаточно масштабируемыми, чтобы справиться с увеличением потока данных на порядки в течение года. Еще одна новая тенденция заключается в хранении таких баз данных в Интернете с использованием хранилищ и вычислительных мощностей компаний наподобие Amazon или Google. Чтобы справляться с огромными объемами данных за разумную цену, хранилища данных должны основываться на массовом применении дешевого широко распространенного аппаратного обеспечения.

Сверхпроизводительные базы данных

Внедрение интернет-ориентированных архитектур предъявляет ко всей инфраструктуре обработки данных предприятия невиданно высокие требования, ведь рабочая нагрузка на СУБД, генерируемая интернет-приложениями, просто не сравнима с той обычной нагрузкой, которую создавали корпоративные приложения еще несколько лет назад. Когда СУБД поддерживала приложения внутреннего пользования, с которыми одновременно работало несколько десятков сотрудников, проблемы производительности могли, конечно, вызывать раздражение у этих сотрудников, но это никак не сказывалось на клиентах компании. С появлением компьютерных центров технической поддержки качество обслуживания клиентов стало зависеть от эффективного управления данными, но приложения по-прежнему могли поддерживать максимум несколько сотен одновременно работающих пользователей (сотрудников, отвечающих на телефонные звонки клиентов).

С появлением Интернета связь между клиентом и базой данных компании становится и вовсе непосредственной. Недостаточная производительность системы проявляется в увеличении времени ответа на запрос клиента, а недоступность ба-

зы данных — в потере заказов. Более того, базы данных больше не защищены от резких повышений интенсивности транзакций. Если финансовая компания предлагает услуги по интерактивному заключению сделок и выполнению операций с ценными бумагами, она должна быть готова к пиковым нагрузкам в дни сильных колебаний на фондовой бирже, которые могут в десятки и сотни раз превышать обычную каждодневную нагрузку системы. Подобным же образом компания, занимающаяся электронной коммерцией, должна быть готова к сезонным изменениям активности покупателей, и под Новый год ее система должна так же хорошо справляться с нагрузкой, как и в средине марта.

Благодаря электронной коммерции и возможности доступа к информации через Интернет в реальном масштабе времени большинство популярных интернет-служб уже сейчас работает на порядок производительнее, чем самые быстрые реляционные СУБД. Чтобы справиться со столь высокими требованиями к производительности, компании активно внедряют распределенные архитектуры и технологии репликации баз данных. Они переносят активные данные поближе к пользователю, интенсивно используя кеширование всего, что только может помочь ускорить взаимодействие. Для обеспечения высокоскоростного доступа к базам данных их содержимое помещают прямо в оперативную память. В первую очередь все это относится к большим веб-сайтам с огромным количеством информации, но по мере того, как кеширование веб-страниц становится все более распространенным и превращается в основу для высокопроизводительной работы, точно так же и кеширование активных данных становится господствующей архитектурой управления растущими объемами данных в Интернете.

Одной из технологий, которая потенциально может справиться с требованиями к производительности, является применение недорогих твердотельных запоминающих устройств, способных заменить дисковые устройства, которые в настоящее время являются основным средством хранения информации. Хотя диски из года в год становятся все дешевле, их производительность растет не так быстро, поскольку зависит от электромеханических факторов. Твердотельная память дешевеет точно так же, как и дисковая, и широко применяется во множестве устройств, таких как MP3-плееры, смартфоны и др. Мы вступаем в эпоху твердотельных устройств для баз данных, в которых не будет ни одной движущейся части, — и эпоха эта начнется уже в ближайшие три–пять лет.

Интеграция Интернета и сетевых служб

В эпоху Интернета управление данными все в большей степени превращается в сетевую службу, которая должна быть тесно интегрирована с другими службами, такими как служба сообщений, транзакций или управление сетью. В некоторых из этих областей давно имеются стандарты, такие, например, как стандарт XA для управления распределенными транзакциями. В других областях стандарты находятся на завершающей стадии разработки, такие как стандарт SOAP для пересылки XML-данных посредством интернет-протокола HTTP и стандарт UDDI (Universal Description, Discovery, and Integration — универсальная система предметного описания и интеграции) для поиска служб в распределенном сетевом окружении. Пока что остается открытым вопрос создания эффективных стандартов

для управления распределенными сетями служб и для определения и поддержки уровней служб в этих сетях.

Многоуровневая архитектура также ставит новые вопросы о ролях, которые должны играть менеджер базы данных и другие компоненты информационных систем. Например, при рассмотрении сетевых транзакций с точки зрения распределенных баз данных решением может являться протокол двухфазного принятия транзакции, реализованный производителем СУБД. Когда сетевые транзакции включают комбинацию устаревших приложений, обновлений реляционных баз данных и коммуникации между приложениями, проблема управления транзакциями выходит за рамки базы данных и требует применения некоторых внешних механизмов.

Подобные проблемы сопровождают и появление серверов приложений на базе Java в качестве платформы промежуточного уровня для бизнес-логики. До эпохи Интернета бизнес-логика встраивалась в базы данных при помощи хранимых процедур. Позже языком хранимых процедур (вместо ранних фирменных языков различных производителей) стал язык программирования Java. В настоящее время серверы приложений образуют альтернативную платформу для бизнес-логики на языке программирования Java — также внешнюю по отношению к базе данных.

Похоже, что бизнес-логика и далее будет располагаться в базах данных или на серверах приложений, в зависимости от организационных, а не технических соображений. Группа управления данными для обеспечения высокоструктурированного обращения к данным будет применять хранимые процедуры и логику в базе данных. Группа интеграции приложений, для того чтобы объединить информацию из базы данных с данными, поступающими из других производственных систем, будет использовать код Java, работающий на сервере приложений. С приобретением Oracle компании BEA все основные производители баз данных теперь создают собственные серверы приложений, так что со временем базы данных и серверы приложений могут оказаться очень тесно интегрированы. Еще одной тенденцией является рост применения веб-платформы с открытым кодом LAMP (Linux, Apache, MySQL и PHP/Python/Perl).

Встраиваемые базы данных

Реляционные технологии управления данными распространились на многие сферы компьютерной индустрии, от маленьких наладонных устройств до больших мэйнфреймов. Практически все корпоративные приложения основаны на базах данных, служащих для хранения информации и управления ею. Еще большее количество приложений пользуется малыми базами данных. Например, специализированной формой баз данных являются службы каталогов, лежащие в основе нового поколения сетевых коммуникационных служб. Высокопроизводительные базы данных малого размера стали составной частью телекоммуникационных сетей, находя применение в сотовой связи, сложных схемах оплаты, интеллектуальных службах сообщений и др.

Встраиваемые базы данных традиционно реализовывались с помощью специально написанного программного кода, интегрированного в приложение. Благодаря максимальной специализации кода этот подход обеспечивал самую высокую производительность, но достигалось это за счет создания мало приспособленной

для модификации и с трудом управляемой архитектуры. Со снижением цен на оперативную память и высокопроизводительные процессоры стало возможным встраивание в такие приложения малых реляционных СУБД.

Преимущества поддержки стандарта встраиваемыми базами данных весьма существенны. Без серьезного ухудшения производительности приложения можно сделать более модульными, прозрачно вносить изменения в структуру базы данных и надстраивать новые модули поверх существующей базы данных. Благодаря этим преимуществам приложения со встраиваемыми базами данных представляют собой новую потенциально растущую область применения SQL и технологий реляционных баз данных. Как и во многих других областях информатики, окончательным триумфом SQL-баз данных может быть их растворение в структуре других программных продуктов и служб — при их отсутствии в качестве автономных приложений они все равно будут жизненно необходимы для программ и служб, их содержащих.

Интеграция с объектно-ориентированными технологиями

В будущем развитии реляционных баз данных труднее всего спрогнозировать то, как будет вестись их интеграция с объектно-ориентированными технологиями. Центр тяжести технологий разработки приложений со всей очевидностью сместился в сторону объектно-ориентированных инструментальных средств и методик. Растет популярность языков C++ и Java, которые применяются не только в клиентских приложениях, но и для реализации деловой логики на сервере. В веб-разработке доминируют объектно-ориентированные языки сценариев, такие как PHP и Perl. Однако базовые принципы табличной организации данных в реляционной модели уходят корнями в эру языка COBOL с его записями и полями и не имеют ничего общего с объектами.

Производители объектно-ориентированных СУБД решили проблему несоответствия между объектной и реляционной архитектурами радикально: они вообще отказались от реляционной модели в пользу строго объектной организации баз данных. Однако отсутствие стандартов, сложность освоения, отсутствие простых средств выполнения запросов и другие недостатки таких СУБД пока не позволили им добиться значительного успеха на рынке. Производители реляционных СУБД ответили на вызов тем, что интегрировали в реляционную модель объектные технологии, в частности, поддержку XML. Уже выполнена большая работа в этом направлении, но можно держать пари, что реляционная и объектная технологии будут интегрированы еще теснее. На этом пути можно отметить следующие тенденции.

- Продолжение роста популярности интерфейсов реляционных СУБД на основе языка Java, таких как JDBC и встраиваемый SQL для Java, а также интерфейсов объектно-ориентированных языков сценариев.
- Java становится стандартным языком хранимых процедур для реализации бизнес-логики в РСУБД. Постепенно в новых приложениях Java может вытеснить процедурные языки производителей, такие как Oracle PL/SQL.
- Растет поддержка разными СУБД сложных абстрактных типов данных, реализующих такие объектно-ориентированные возможности, как ин-

капсуляция и наследование. XML обеспечивает хранение структурированных нереляционных данных, которое будет дополнено возможностью хранения потоковых данных, таких как музыка и видео.

- Растет важность интерфейсов, ориентированных на обработку сообщений, включая триггеры, генерирующие внешние сообщения для взаимодействия с другими приложениями.
- Размываются границы между системами управления содержимым, используемыми для работы с документами, и системами управления реляционными базами данных — так же как XML стирает отличие между "документом" и "записью структурированных данных".

Эволюция РСУБД в последние пару десятилетий следует единому шаблону: на сцене появляется новая технология (такая, как объекты, XML или хранилища данных), которая вызывает массу энтузиазма и волну новых компаний. Проходит несколько лет, в течение которых новые игроки держатся на гребне волны и снимают сливки. В это же время производители РСУБД находят пути интеграции новой технологии в свои продукты — пусть вначале и на очень слабом уровне. Постепенно результат интеграции становится достаточно хорош для того, чтобы все основные производители баз данных приняли на вооружение новую технологию. Нет оснований считать, что в ближайшем будущем этот шаблон претерпит какие-то резкие изменения.

Горизонтально масштабируемые базы данных

С развитием Интернета многие стандартные элементы его архитектуры сталкиваются с необходимостью обработки очень большого количества "транзакций". Веб-серверы должны уметь обрабатывать миллионы запросов в час. Серверы приложений должны выполнять бизнес-логику для десятков тысяч транзакций. Примерно такого же масштаба задачи стоят и перед сетевым оборудованием — роутерами, коммуникаторами и т.п.

В каждом из описанных случаев масштабирование горизонтально — один веб-сервер превращается в два, три, десяток, а потенциально — в сотни серверов, разделяющих нагрузку на уровне веб-сервера. Эти серверы используют или одно общее, или реплицированные хранилища данных, причем каждый сервер в состоянии выдать любую запрошенную страницу. Чтобы такая схема работала, сервер должен представлять собой систему без сохранения состояния. Насколько это возможно, любая информация, которая должна храниться между отдельными запросами, должна передаваться обратно клиенту, а он должен передавать ее серверу при следующем запросе. Любая информация о состоянии, которая не может быть обработана указанным образом, передается "в обратном направлении" — на сервер базы данных, лежащий в основе описанной архитектуры.

Для многих крупных веб-сайтов или веб-служб такая база данных становится узким местом. К сожалению, горизонтальное масштабирование, работающее для веб-серверов и серверов приложений, в случае серверов баз данных приводит к проблемам. В силу самой их природы базы данных не могут быть без запоминания состояния. Все отдельные элементы транзакции должны быть одновременно

либо приняты, либо отвергнуты, а параллельно выполняющиеся транзакции не должны влиять друг на друга. Для них нет "обратного направления", куда можно передать информацию о состоянии — эта информация должна поддерживаться самой базой данных.

Так мы пришли к новой важной задаче архитектуры баз данных — разработка системы, которую можно эффективно горизонтально масштабировать. За последние годы были достигнуты значительные успехи в создании баз данных, обладающих возможностью кластеризации или применения грид-технологии. Эти новые архитектуры должны позволять распределить обработку данных среди десятков (или большего количества) серверов, но информация при этом должна оставаться согласованной во всех отношениях (см. главу 12, "Обработка транзакций"). Частично решение достигается путем интеллектуального разделения различных типов данных. Ссылки и неизменяемые данные (или изменяемые редко, такие как контактная информация клиентов) могут быть горизонтально реплицированы, так как поддержка их согласованности на десятках серверов не требует особых усилий. Изменяющиеся данные хранятся в одной системе кластера, и все запросы к таким данным автоматически перенаправляются в эту систему.

Резюме

SQL продолжает играть важную роль в компьютерной индустрии, и важность этой роли пока что не уменьшается.

- SQL-базы данных являются флагманским программным продуктом для трех крупнейших производителей программного обеспечения в мире — Microsoft, Oracle и IBM.
- SQL-базы данных в состоянии работать в любых компьютерных системах, от мэйнфреймов и серверов баз данных до наладонных устройств.
- Все основные корпоративные приложения, используемые в больших организациях, опираются при работе с данными на SQL-базы данных уровня предприятия.
- SQL-базы данных успешно отвечают на требования объектной модели путем соответствующих расширений SQL в объектно-реляционных базах данных.
- SQL-базы данных успешно отвечают на требования, выдвигаемые интернет-архитектурами, путем поддержки работы с XML и тесной интеграции с серверами приложений.
- Одной из приоритетных задач, стоящих перед современным рынком баз данных, является обеспечение возможности масштабирования баз данных и использование для обработки данных вычислительных кластеров.

VII ЧАСТЬ

Приложения

Приложение А
Учебная база данных

Приложение Б
Производители СУБД

Приложение В
Синтаксис SQL

ПРИЛОЖЕНИЕ А

Учебная база данных

Большинство примеров в данной книге основано на учебной базе данных, описанной в данном приложении. Учебная база данных содержит данные, поддерживающие простое приложение обработки заказов для маленькой компании. Она состоит из пяти таблиц.

- PRODUCTS (товары) — содержит по одной строке для каждого наименования товара, продаваемого компанией.
- OFFICES (офисы) — содержит по одной строке для каждого из пяти офисов компании.
- SALESREPS (служащие) — содержит по одной строке для каждого из десяти служащих компании.
- CUSTOMERS (клиенты) — содержит по одной строке для каждого из клиентов компании.
- ORDERS (заказы) — содержит по одной строке для каждого из заказов, сделанных клиентом. Для простоты считается, что один заказ может содержать только один товар.

На рис. А.1 в графическом виде представлены пять перечисленных таблиц, столбцы, которые в них содержатся, и отношения "предок-потомок", существующие между ними. Первичные ключи таблиц выделены серым цветом. Эти таблицы могут быть созданы при помощи приведенных ниже инструкций CREATE TABLE и ALTER TABLE.

Среди различных SQL-баз данных наблюдаются определенные отличия в поддержке типов данных и синтаксиса. Поэтому приведенные ниже инструкции могут потребовать внесения изменений, чтобы удовлетворять требованиям используемой вами базы данных. Чтобы облегчить вашу задачу, на сайте издательства имеются сценарии, адаптированные для баз данных DB2, MySQL, Oracle и SQL Server. В них также содержатся инструкции INSERT для загрузки данных в таблицы базы данных. Чтобы получить эти сценарии, выполните следующие шаги.

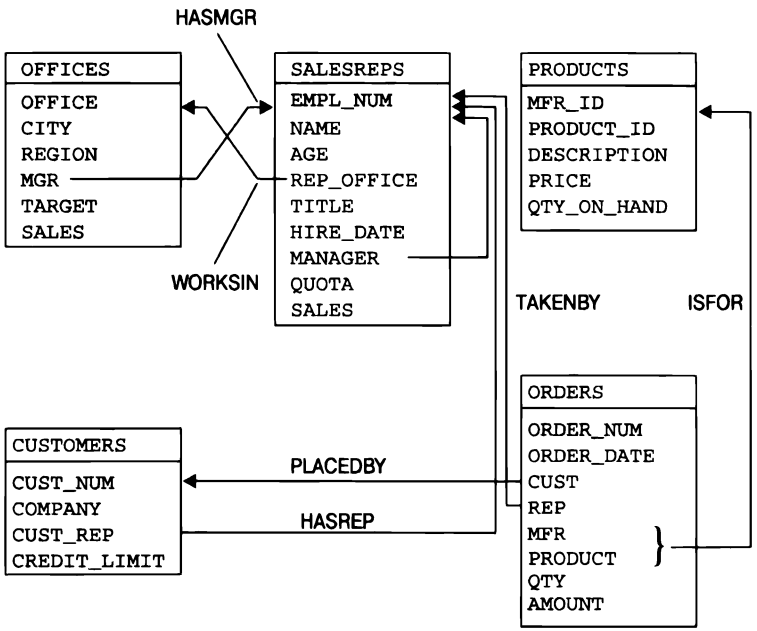


Рис. А.1. Структура учебной базы данных

1. Откройте ваш веб-браузер и перейдите по адресу `http://www.mhprofessional.com/computingdownload`.
2. На полосе в верхней части страницы щелкните на пункте COMPUTING.
3. В левой части щелкните на ссылке Downloads Section.
4. Прокрутите страницу вниз, до названия данной книги.
5. Выберите интересующие вас файлы и загрузите их на ваш компьютер. Если вашей СУБД нет в списке, рекомендуем начать с файлов для MySQL (как базы данных, наиболее соответствующей текущему стандарту SQL).
6. Обратитесь к документации по вашей базе данных, чтобы узнать, каким образом воспользоваться загруженными сценариями.

```
CREATE TABLE PRODUCTS
     (MFR_ID CHAR(3) NOT NULL,
  PRODUCT_ID CHAR(5) NOT NULL,
 DESCRIPTION VARCHAR(20) NOT NULL,
       PRICE DECIMAL(9,2) NOT NULL,
 QTY_ON_HAND INTEGER NOT NULL,
 PRIMARY KEY (MFR_ID, PRODUCT_ID));

CREATE TABLE OFFICES
     (OFFICE INTEGER NOT NULL,
        CITY VARCHAR(15) NOT NULL,
      REGION VARCHAR(10) NOT NULL,
         MGR INTEGER, TARGET DECIMAL(9,2),
       SALES DECIMAL(9,2) NOT NULL,
     PRIMARY KEY (OFFICE),
     FOREIGN KEY HASMGR (MGR)
```

```
  REFERENCES SALESREPS
   ON DELETE SET NULL);

CREATE TABLE SALESREPS
   (EMPL_NUM INTEGER NOT NULL,
        NAME VARCHAR(15) NOT NULL,
         AGE INTEGER,
  REP_OFFICE INTEGER,
       TITLE VARCHAR(10),
   HIRE_DATE DATE NOT NULL,
     MANAGER INTEGER,
       QUOTA DECIMAL(9,2),
       SALES DECIMAL(9,2) NOT NULL,
     PRIMARY KEY (EMPL_NUM),
     FOREIGN KEY (MANAGER)
  REFERENCES SALESREPS
   ON DELETE SET NULL,
 FOREIGN KEY WORKSIN (REP_OFFICE)
  REFERENCES OFFICES
   ON DELETE SET NULL);

CREATE TABLE CUSTOMERS
   (CUST_NUM INTEGER NOT NULL,
     COMPANY VARCHAR(20) NOT NULL,
    CUST_REP INTEGER,
CREDIT_LIMIT DECIMAL(9,2),
 PRIMARY KEY (CUST_NUM),
 FOREIGN KEY HASREP (CUST_REP)
  REFERENCES SALESREPS
   ON DELETE SET NULL);

CREATE TABLE ORDERS
  (ORDER_NUM INTEGER NOT NULL,
  ORDER_DATE DATE NOT NULL,
        CUST INTEGER NOT NULL,
         REP INTEGER,
         MFR CHAR(3) NOT NULL,
     PRODUCT CHAR(5) NOT NULL,
         QTY INTEGER NOT NULL,
      AMOUNT DECIMAL(9,2) NOT NULL,
 PRIMARY KEY (ORDER_NUM),
 FOREIGN KEY PLACEDBY (CUST)
  REFERENCES CUSTOMERS
   ON DELETE CASCADE,
 FOREIGN KEY TAKENBY (REP)
  REFERENCES SALESREPS
   ON DELETE SET NULL,
 FOREIGN KEY ISFOR (MFR, PRODUCT)
  REFERENCES PRODUCTS
   ON DELETE RESTRICT);

ALTER TABLE OFFICES
        ADD CONSTRAINT HASMGR
    FOREIGN KEY(MGR)
 REFERENCES SALESREPS(EMPL_NUM)
  ON DELETE SET NULL;
```

На рис. А.2–А.6 приведено содержимое каждой из пяти таблиц учебной базы данных. Таблицы результатов почти всех запросов, встречающихся в книге, основаны на данных, представленных на этих рисунках.

CUST_NUM	COMPANY	CUST_REP	CREDIT_LIMIT
2111	JCP Inc.	103	$50000.00
2102	First Corp.	101	$65000.00
2103	Acme Mfg.	105	$50000.00
2123	Carter & Sons	102	$40000.00
2107	Ace International	110	$35000.00
2115	Smithson Corp.	101	$20000.00
2101	Jones Mfg.	106	$65000.00
2112	Zetacorp	108	$50000.00
2121	QMA Assoc.	103	$45000.00
2114	Orion Corp.	102	$20000.00
2124	Peter Brothers	107	$40000.00
2108	Holm & Landis	109	$55000.00
2117	J.P. Sinclair	106	$35000.00
2122	Three-Way Lines	105	$30000.00
2120	Rico Enterprises	102	$50000.00
2106	Fred Lewis Corp.	102	$65000.00
2119	Solomon Inc.	109	$25000.00
2118	Midwest Systems	108	$60000.00
2113	Ian & Schmidt	104	$20000.00
2109	Chen Associates	103	$25000.00
2105	AAA Investments	101	$45000.00

Рис. А.2. Таблица CUSTOMERS

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA	SALES
105	Bill Adams	37	13	Sales Rep	2006-02-12	104	$350000.00	$367911.00
109	Mary Jones	31	11	Sales Rep	2007-10-12	106	$300000.00	$392725.00
102	Sue Smith	48	21	Sales Rep	2004-12-10	108	$350000.00	$474050.00
106	Sam Clark	52	11	VP Sales	2006-06-14	NULL	$275000.00	$299912.00
104	Bob Smith	33	12	Sales Mgr	2005-05-19	106	$200000.00	$142594.00
101	Dan Roberts	45	12	Sales Rep	2004-10-20	104	$300000.00	$305673.00
110	Tom Snyder	41	NULL	Sales Rep	2008-01-13	101	NULL	$75985.00
108	Larry Fitch	62	21	Sales Mgr	2007-10-12	106	$35000000	$361865.00
103	Paul Cruz	29	12	Sales Rep	2005-03-01	104	$275000.00	$286775.00
107	Nancy Angelli	49	22	Sales Rep	2006-11-14	108	$300000.00	$186042.00

Рис. А.3. Таблица SALESREPS

ORDER_NUM	ORDER_DATE	CUST	REP	MFR	PRODUCT	QTY	AMOUNT
112961	2007-12-17	2117	106	REI	2A44L	7	$31500.00
113012	2008-01-11	2111	105	ACI	41003	35	$3745.00
112989	2008-01-03	2101	106	FEA	114X	6	$1458.00
113051	2008-02-10	2118	108	QSA	Xk47	2	$1420.00
112968	2007-10-12	2102	101	ACI	41004	34	$3978.00
113036	2008-01-30	2107	110	ACI	4100Z	9	$22500.00
113045	2008-02-02	2112	108	REI	2A44R	10	$45000.00
112963	2007-12-17	2103	105	ACI	41004	28	$3276.00
113013	2008-01-14	2118	108	BIC	41003	1	$652.00
113058	2008-02-23	2108	109	FEA	112	10	$1480.00
112997	2008-01-08	2124	107	BIC	41003	1	$652.00
112983	2007-12-27	2103	105	ACI	41004	6	$702.00
113024	2008-01-20	2114	108	QSA	Xk47	20	$7100.00
113062	2008-02-24	2124	107	FEA	114	10	$2430.00
112979	2007-10-12	2114	102	ACI	4100Z	6	$15000.00
113027	2008-01-22	2103	105	ACI	41002	54	$4104.00
113007	2008-01-08	2112	108	IMM	773C	3	$2925.00
113069	2008-03-02	2109	107	IMM	775C	22	$31350.00
113034	2008-01-29	2107	110	REI	2A45C	8	$632.00
112992	2007-11-04	2118	108	ACI	41002	10	$760.00
112975	2007-10-12	2111	103	REI	2A44G	6	$2100.00
113055	2008-02-15	2108	101	ACI	4100X	6	$150.00
113048	2008-02-10	2120	102	IMM	779C	2	$3750.00
112993	2007-01-04	2106	102	REI	2A45C	24	$1896.00
113065	2008-02-27	2106	102	QSA	Xk47	6	$2130.00
113003	2008-01-25	2108	109	IMM	779C	3	$5625.00
113049	2008-02-10	2118	108	QSA	Xk47	2	$776.00
112987	2007-12-31	2103	105	ACI	4100Y	11	$27500.00
113057	2008-02-18	2111	103	ACI	4100X	24	$600.00
113042	2008-02-02	2113	101	REI	2A44R	5	$22500.00

Рис. А.4. Таблица ORDERS

OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Denver	Western	108	$300000.00	$186042.00
11	New York	Eastern	106	$575000.00	$692637.00
12	Chicago	Eastern	104	$800000.00	$735042.00
13	Atlanta	Eastern	105	$350000.00	$367911.00
21	Los Angeles	Western	108	$725000.00	$835915.00

Рис. А.5. Таблица OFFICES

MFR_ID	PRODUCT_ID	DESCRIPTION	PRICE	QTY_ON_HAND
REI	2A45C	Ratchet Link	$2750.00	210
ACI	4100Y	Widget Remover	$79.00	25
QSA	Xk47	Reducer	$355.00	38
BIC	41672	Plate	$180.00	0
IMM	779C	900-lb Brace	$1875.00	9
ACI	41003	Size 3 Widget	$107.00	207
ACI	41004	Size 4 Widget	$117.00	139
BIC	41003	Handle	$652.00	3
IMM	887P	Brace Pin	$250.00	24
QSA	Xk48	Reducer	$134.00	203
REI	2A44L	Left Hinge	$4500.00	12
FEA	112	Housing	$148.00	115
IMM	887H	Brace Holder	$54.00	223
BIC	41089	Retainer	$225.00	78
ACI	41001	Size 1 Widget	$55.00	277
IMM	775C	500-lb Brace	$1425.00	5
ACI	4100Z	Widget Installer	$2500.00	28
QSA	XK48A	Reducer	$177.00	37
ACI	41002	Size 2 Widget	$76.00	167
REI	2A44R	Right Hinge	$4500.00	12
IMM	773C	300-lb Brace	$975.00	28
ACI	4100X	Widget Adjuster	$25.00	37
FEA	114	Motor Mount	$243.00	15
IMM	887X	Brace Retainer	$475.00	32
REI	2A44G	Hinge Pin	$350.00	14

Рис. А.6. Таблица PRODUCTS

Реальная база данных для работы с заказами, вероятно, будет содержать несколько десятков таблиц, а таблицы будут включать такие дополнительные столбцы, как адреса для отправки счетов и товара, а также транзакции, такие как возврат товара или вычисление налогов. Конечно же, количество строк в реальной базе данных будет во много раз превосходить количество строк в учебной базе данных. Однако приведенной здесь учебной базы данных вполне достаточно, чтобы проиллюстрировать основные возможности SQL, а небольшое количество строк упрощает отслеживание того, как получаются окончательные результаты запросов к базе данных.

ПРИЛОЖЕНИЕ Б

Производители СУБД

В этом приложении приведена краткая информация об основных производителях баз данных, а также проекты с открытым кодом, лидирующие на рынке корпоративных баз данных или в других сегментах рынка.

- Aster Data (nCluster)
- CodeGear (Interbase)
- dataBased Intelligence (dBASE Plus)
- Encirq (Device SQL)
- EnterpriseDB (Postgres Plus)
- Firebird (Firebird)
- Greenplum (Greenplum)
- Hewlett-Packard (NonStop SQL, HP Oracle Database Machine)
- HSQLDB (HSQLDB)
- IBM (DB2 editions, Informix, SolidDB)
- Ingres Corporation (Ingres)
- Intersystems (Cachй)
- Matisse Software, Inc. (Matisse)
- Microsoft (SQL Server)
- Mimer Information Technology (Mimer)
- Netezza (Netezza)
- Oracle Corporation (Oracle editions, Rdb, TimesTen, SleepyCat)
- ParAccel Inc. (ParAccel)
- Postgres (PostgreSQL)
- Streambase Systems (Streambase)
- Sun Microsystems (MySQL)

- Sybase (Adaptive Server Enterprise, SQL Anywhere)
- Teradata Corporation (Teradata)
- Truviso, Inc. (Truviso)
- Vertica Systems (Vertica)
- Xeround (Xeround Intelligent Data Grid)

Aster Data (nCluster)

Aster Data — компания, ориентированная на "передовые хранилища данных". Ее усилия сосредоточены на создании нового поколения приложений для анализа данных, таких как поток информации о действиях клиентов очень больших веб-сайтов, распознавания в реальном времени спама и фишинга, выполнение сложного финансового анализа. Традиционные хранилища данных для решения таких задач оказываются неадекватны, поэтому компания предлагает собственную базу данных nCluster, основанную на многоуровневой кластерной архитектуре.

CodeGear (Interbase)

CodeGear — подразделение Embarcadero Technologies, которое в настоящее время отвечает за СУБД Interbase. Interbase хорошо подходит для приложений со встраиваемыми базами данных, когда вся система должна поместиться в несколько десятков мегабайт дискового пространства. Система спроектирована для работы без администратора базы данных. У данного продукта длинная и занятная история. Изначально он был разработан как база данных для рабочей станции Apollo в средине 1980-х годов. Права на Interbase были проданы Ashton-Tate, ведущему производителю баз данных того времени для персональных компьютеров. В 1991 году Ashton-Tate была куплена компанией Borland, и Interbase стал продуктом Borland. Примерно через 10 лет, в 2000-х годах, компания Borland приняла решение об изменении лицензии Interbase и сделала его доступным в качестве продукта с открытым кодом. Версия продукта с открытым кодом осталась активна в виде проекта Firebird; коммерческая же версия доступна у компании CodeGear/Embarcadero, продолжающей разработку.

dataBased Intelligence (dBASE Plus)

В настоящее время базой данных dBASE (одной из первых баз данных для персональных компьютеров, появившейся более 25 лет назад) занимается компания dataBased Intelligence (dBI). Изначально dBASE разрабатывалась компанией Ashton-Tate, была лидером рынка баз данных для персональных компьютеров, контролируя до 70% этого рынка. Со временем на рынок вышли более мощные базы данных, составив конкуренцию dBASE. Наиболее известная из них — SQL Server, образованная из базы данных фирмы Sybase. Одновременно стали появляться конкурирующие "клоны" dBASE, включая появившуюся в 1987 году Foxbase.

В 1991 году Borland приобрела Ashton-Tate, в первую очередь из-за большого количества установленных баз dBASE. Ответственность за разработку dBASE в 1999 году перешла к dBI, которая приступила к разработке нового продукта — dBASE Plus.

Encirq (DeviceSQL)

DeviceSQL компании Encirq представляет собой SQL-базу данных, предназначенную для встраивания. При малых требованиях к ресурсам и малом размере, DeviceSQL обеспечивает потребности в управлении данными для таких устройств, как принтеры, сканеры, факсы, автомобильные системы, и там, где требуется гибкость SQL при существенных ограничениях на количество памяти (минимальная конфигурация требует всего лишь несколько десятков килобайт памяти).

Позже компания использовала свои разработки на рынке систем для сложной обработки событий (Complex Event Processing, CEP). Здесь данная компания конкурирует с некоторыми вновь созданными компаниями по производству потоковых баз данных.

EnterpriseDB (Postgres Plus)

EnterpriseDB предлагает коммерческую версию базы данных с открытым кодом Postgres, оптимизированную для большого количества транзакций и высокой масштабируемости. Postgres Plus добавляет к ядру Postgres тонкую настройку, инструментарий для мониторинга, а также такие возможности, как уровень кеширования в памяти и шифрование. Еще один продукт, Postgres Plus Advanced Server, добавляет к Postgres совместимость с Oracle, включая синтаксис SQL в стиле Oracle, типы данных, совместимые с Oracle, поддержку языка Oracle PL/SQL и Oracle Call Interface (OCI).

Firebird (Firebird)

Firebird представляет собой SQL-базу данных с открытым кодом, предназначенную для встраиваемых приложений. Она обладает относительно богатой реализацией SQL (включая ACID-транзакции, ссылочную целостность, хранимые процедуры, триггеры и многое другое) в очень небольшом размере. Проект Firebird начался в 2000 году на основе базы данных с открытым кодом Borland Interbase. С того времени он был существенно переработан, причем разработка продолжается и в настоящее время.

Greenplum

Greenplum — фирма-новичок с венчурным капиталом, нацеленная на рынок хранилищ данных. Ее база данных Greenplum основана на технологии Postgres с открытым кодом, который компания оптимизирует и настраивает для поддержки больших хранилищ данных. Программное обеспечение Greeenplum распределяет содержимое хранилища среди большого количества дешевых серверных систем. Затем оно распараллеливает запросы среди серверов, чтобы обеспечить быстрый отклик независимо от размера базы данных.

Hewlett-Packard Company

Hewlett-Packard (HP) — один из наиболее крупных производителей компьютерных систем; управление базами данных исторически играет важную роль в линии продуктов HP. В 1970-х годах компания была первой вышедшей на рынок баз данных для мини-компьютеров с СУБД Image/1000, которая работала на мини-компьютерах HP 1000 для инженерных и научных расчетов, и Image/3000 — для

мини-компьютера HP 3000 для коммерческих приложений. СУБД Image базировалась на сетевой модели данных. В средине 1980-х годов HP вывела на рынок обновленную СУБД Allbase, которая обеспечивала реляционный SQL-интерфейс и была обратно совместима с базами данных Image. Важным рынком HP являлись мини-компьютеры, а позже — UNIX-серверы, которые стали важной платформой для независимых разработчиков СУБД, таких как Ingres, Informix, Oracle и Sybase. В конечном итоге компания забросила свой проект Allbase, переключившись на сотрудничество с независимыми производителями. HP вернулась на рынок баз данных через "черный ход", приобретя в 2001 году компанию Compaq, которая в свое время приобрела Tandem Computers. Позже HP вышла на рынок хранилищ данных путем партнерства с Oracle Corporation. Эти две компании объединили свои усилия в разработке HP Oracle Exadata Database Machine, компьютера, в котором программное обеспечение баз данных Oracle было тесно интегрировано с аппаратным обеспечением HP. Этот компьютер продается как устройство для хранилищ данных фирмой Oracle.

HSQLDB (HSQLDB)

HSQL Database Engine (HSQLDB) представляет собой реляционную базу данных с открытым кодом, написанную на Java. Она имеет малые размеры (до 100 Кбайт при встраивании в аплеты) с достаточно богатым синтаксисом SQL и возможностью выбора встраиваемой или серверной архитектуры. HSQLDB поддерживает таблицы на диске и в памяти. Продукт происходит от Java-базы данных Hypersonic и входит в комплект офисного программного обеспечения OpenOffice.org.

IBM Corporation (DB2, Informix)

IBM — наибольший в мире производитель в области вычислительной техники, и программное обеспечение обеспечивает большую и важную часть его доходов. Базы данных были важным источником доходов IBM в течение нескольких десятилетий и все еще составляют львиную долю ее программного обеспечения. IBM все еще поддерживает два старых нереляционных продукта — IMS и DL/1 (Data Language/1), но сейчас среди ее разработок доминируют реляционные SQL-продукты.

- **DB2 для z/OS** представляет собой флагманскую реляционную базу данных. Это большой, сложный программный продукт, работающий на мэйнфреймах IBM под управлением операционной системы z (преемника OS/390 и MVS). Этот продукт предлагает широчайший набор возможностей среди всех баз данных IBM, включая высокоэффективную обработку транзакций, хранилища данных и интегрированную поддержку XML.
- **DB2 для Linux, UNIX и Windows** — СУБД для серверных систем от уровня отдела до больших кластеров UNIX-серверов для больших корпоративных центров данных. Так же как и ее двойник для мэйнфреймов, она построена на реляционной модели и поддерживает XML-данные. В свое время эта СУБД носила название DB2 Universal Database (UDB) для того, чтобы отличать ее от версии DB2 для мэйнфреймов, построенной на другой версии кода. Имеется несколько разных версий данного программного обеспечения — Express Edition, Workgroup Edition и Enterprise Edition.

- **DB2 Everywhere** расширяет семейство DB2, поддерживая мобильные устройства и встраиваемые приложения. Эта редакция DB2 поддерживает также возможность синхронизации при непостоянном соединении. DB2 Everywhere конкурирует с СУБД Sybase SQL Anywhere, доминирующей в мобильном сегменте рынка.
- **Informix** приобретена IBM в 2001 году. Одна из наиболее ранних UNIX-баз данных, Informix переделана для поддержки SQL в 1985 году и стала одной из первых баз данных, которые смогли полностью использовать возможности SMP UNIX-серверов. В 1995 году компания Informix приобрела компанию Illustra — пионера в области объектно-реляционных баз данных. После приобретения Illustra проблемы управления привели компанию к застою в конце 1990-х годов, после чего ее приобрела IBM — в первую очередь из-за ее более чем 100000 клиентов. IBM поддерживает Informix в качестве OLTP-базы данных для UNIX.
- **SolidDB** — реляционная база данных в памяти, которую IBM приобрела в 2008 году. На момент написания этой книги IBM предлагает данный продукт в двух редакциях: IBM SolidDB представляет собой автономную базу данных в памяти с очень малым временем задержки для приложений реального времени и IBM SolidDB Universal Cache — для применения в качестве кеша, повышающего производительность дисковых РСУБД.

Перечисленные базы данных IBM являются частью более широкого семейства программного обеспечения IBM.

Ingres Corporation (Ingres)

База данных Ingres происходит от одного из самых ранних прототипов реляционных баз данных, созданного в Калифорнийском университете в Беркли. Исходный текст Berkeley Ingres был легко доступен за небольшую цену, и несколько студентов и профессоров организовали компанию Relational Technology, Inc. (позже переименованную в Ingres Corporation), чтобы продолжить разработку коммерческой версии. В начале и средине 1980-х годов СУБД Ingres и ее язык QUEL были главными конкурентами SQL, и в особенности сильная борьба на рынке СУБД среднего уровня развернулась между Ingres и Oracle. Когда SQL стал стандартным языком баз данных, Relational Technology адаптировал Ingres для поддержки и QUEL, и SQL. Впервые база данных Ingres была реализована для мини-компьютеров Digital и имела успех в академическом обществе, но со временем центр тяжести переместился на платформы UNIX, а позже Linux, где и остается в настоящее время.

Компания была приобретена в начале 1990-х годов компанией ASK, создателем программного обеспечения производственного назначения. Четырьмя годами позже компания ASK/Ingres была, в свою очередь, приобретена фирмой Computer Associates (CA). CA продолжала совершенствовать продукт в течение следующего десятилетия, но в конце концов в 2004 году решила превратить ее в базу данных с открытым кодом, так что в настоящее время это мощная, устоявшаяся база данных, доступная по очень скромной цене. CA образовала новую компанию, Ingres Corporation, которая обеспечивает поддержку и обслуживание базы данных с открытым кодом Ingres.

Intersystems (Caché)

Intersystems распространяет свою базу данных Caché как "самую быструю в мире объектную базу данных". Происхождение как компании, так и ее продукта относится к 1960-м годам (еще до появления реляционных баз данных!), и связано с операционной системой и языком программирования MUMPS (Massachusetts General Hospital Utility Multi-Programming System), разработанными в Massachusetts General Hospital. Эта система была создана для приложений, ориентированных на работу с базами данных, и была популярна в течение двух последующих десятилетий среди медицинских и финансовых учреждений. В конечном итоге компания скупила многих своих конкурентов и объединила их продукты под одной торговой маркой Caché в конце 1990-х годов. Само собой, за прошедшее время код претерпел существенные изменения и усовершенствования.

Matisse Software, Inc. (Matisse)

Matisse описывается как "постобъектно-реляционная" база данных, продолжающая эволюцию технологий баз данных после реляционной, объектно-ориентированной и объектно-реляционной фаз. База данных поддерживает расширенный язык SQL, разработанный для расширения модели данных за пределы таблиц и включения "семантических сетей" данных. Продукт предлагает также возможности распределенных баз данных для конфигураций ячеечных серверов.

Microsoft Corporation

Microsoft — крупнейший в мире производитель программного обеспечения для персональных компьютеров. Он доминирует в ряде секторов рынка для персональных компьютеров и рабочих групп, включая офисные приложения (MS Office и его электронные таблицы, текстовые редакторы и прочие компоненты), почтовые приложения (Microsoft Exchange server), службы каталогов (Microsoft Active Directory), а также программное обеспечение для рабочих групп (Microsoft SharePoint) и разные версии операционной системы Windows, под управлением которой работает все это программное обеспечение. Microsoft SQL Server — важнейший компонент линии программного обеспечения для рабочих групп. Он служит как в качестве автономной реляционной базы данных, так и как встраиваемый компонент для других серверных систем.

До 1987 года среди продуктов Microsoft не было СУБД. Но в 1987 году IBM анонсировала свою операционную систему OS/2 Extended Edition с интегрированной СУБД и обменом данными. В 1988 году Microsoft ответила базой данных SQL Server, версии СУБД от Sybase, перенесенной под OS/2. Вскоре Microsoft забросила OS/2 и занялась собственной операционной системой Windows, но при этом не остановила работы над SQL Server, перенеся его серверную версию под Windows. В то время на рынке баз данных для персональных компьютеров доминировала компания Ashton-Tate со своей базой данных dBASE, и Microsoft подстраховалась, приобретя в начале 1990-х годов Foxbase, которая предлагала свой клон dBASE. Однако продукт Foxbase был быстро заменен Microsoft Access, собственным продуктом с графическим интерфейсом и подключением через ODBC к "настоящим" базам данных, включая SQL Server.

Приверженность Microsoft к SQL Server с годами росла, и сегодня он входит в десятку наиболее важных продуктов Microsoft. Несмотря на конкуренцию со стороны Oracle, IBM и других, Microsoft SQL Server успешно использует факт принадлежности операционной системы Windows компании Microsoft и представляет собой наиболее популярную базу данных для Windows.

Microsoft постепенно делает SQL Server все более сложной средой для центров данных. В SQL Server 2008 добавлены средства шифрования, аудита и управления, а также поддержка пространственных данных, расширенная поддержка XML и пользовательских типов данных. Все эти возможности обеспечивают рост популярности SQL Server, которая ограничена только фактом жесткой привязки к серверным платформам Windows.

Mimer Information Technology (Mimer)

База данных Mimer (названная так в честь скандинавского бога мудрости) была создана более 20 лет назад в университете Упсала в Швеции. Университетский проект превратился в одноименную компанию, разрабатывающую SQL-базы данных для встраиваемых и мобильных приложений. Этот продукт доступен в трех вариантах. Mimer SQL Embedded оптимизирован по размеру (несколько сотен килобайт) и не требует поддержки. Mimer SQL Mobile оптимизирован для мобильных телефонов и других наладонных устройств, требующих операций с локальной базой данных и подключения к корпоративной базе данных. Mimer SQL Enterprise работает в качестве центральной базы данных для поддержки версий Embedded или Mobile или в качестве автономной корпоративной базы данных. Компания гордится совместимостью со стандартами.

Netezza Corporation (Netezza)

Netezza предлагает базу данных для хранилищ данных и приложений интеллектуальных ресурсов предприятия. Компания заявляет об упрощении хранения данных путем создания специализированных устройств, представляющих собой тесно интегрированные и тонко настроенные комбинации аппаратного и программного обеспечения. В них используются параллельные вычисления и распределение данных по многим индивидуальным элементам обработки, каждый из которых включает диск, микропроцессор и специализированные микросхемы Netezza, спроектированные для оптимизации и разгрузки обработки запросов к дисковым потокам данных. Поскольку многие запросы в заявленной области применения требуют последовательного сканирования целых таблиц, компания утверждает, что такой подход может превосходить по производительности обычные базы данных, в которых данные для обработки должны загружаться с диска в оперативную память.

Oracle Corporation

Oracle Corporation была первым производителем СУБД, разработавшим коммерческий SQL-продукт, опередив IBM почти на два года. В 1980-е годы компания Oracle стала крупнейшим независимым производителем СУБД. Руководимая основателем и генеральным директором Ларри Эллисоном (Larry Ellison), компания в настоящее время является наиболее успешной и крупной программной компанией в мире, уступая пальму первенства по доходам только Microsoft.

СУБД Oracle изначально разработана для мини-компьютеров Digital, но продажи Oracle существенно выросли с ростом популярности систем на базе UNIX (а позже — на базе Linux), которые принесли компании многомиллиардные доходы. СУБД Oracle изначально основывалась на прототипе IBM System/R и была совместима с SQL-продуктами IBM. По мере роста доли рынка Oracle разработала собственные расширения наподобие языка программирования PL/SQL. Во время "войны баз данных" в 1980-е годы Oracle агрессивно рекламировала высокую OLTP-производительность своей СУБД. Компания всегда совмещала отличные технологии с агрессивными продавцами и продуманной маркетинговой политикой.

В настоящее время линия продуктов Oracle включает корпоративные продукты, такие как серверы приложений и межплатформенное программное обеспечение, финансовые приложения, программное обеспечение для телекоммуникаций и многое другое. Однако в центре внимания компании остаются технологии баз данных, и основная доля рынка Oracle — это базы данных уровня предприятия. Компания является также лидером в области хранилищ данных, встраиваемых баз данных, XML-интеграции и во многих других нишах рынка; она даже рискует вкладывать средства в аппаратное обеспечение, продавая устройства для хранилищ данных на аппаратной основе от Hewlett-Packard.

Флагманская база данных Oracle прошла через несколько основных версий — Oracle7, Oracle8i и 9i (i означает "Интернет") и Oracle 10g и 11g (g означает "грид"). Это исключительно большое и сложное программное обеспечение, вокруг которого выросла целая подиндустрия — администрирование базы данных Oracle, настройка производительности базы данных Oracle, обучение и консультации Oracle. Большая часть технологий баз данных Oracle разработана самой компанией (в отличие от приложений и межплатформенного программного обеспечения Oracle, где компания приобрела ряд таких крупных и успешных компаний, как PeopleSoft, Siebel, Hyperion и BEA Systems). Компания сделала несколько важных приобретений и в области баз данных. Это база данных для мини-компьютеров Rdb/VMS, база данных в памяти TimesTen и встраиваемая база данных SleepyCat.

Rdb/VMS, приобретенная Oracle в 1994 году, была пионерской разработкой Digital Equipment Corporation для своего 32-битового мини-компьютера VAX/VMS. Эта база данных позже была перенесена в операционную систему OpenVMS (гибрид UNIX/VMS) и работала на высокопроизводительных процессорах Digital Alpha. В конечном счете все производители мини-компьютеров приняли решение прекратить разработки своих баз данных и либо просто завершили их, либо продали. Digital не был исключением, и Oracle приобрел как базу клиентов Rdb, так и команду ее разработчиков.

TimesTen, приобретенная Oracle в 2005 году, была компанией с венчурным капиталом, которая коммерциализировала технологию баз данных в памяти, разработанную HP Labs в начале 1990-х годов. В отличие от обычных баз данных, использующих для хранения данных диски, база данных в памяти оптимизирована для хранения данных в памяти и обеспечения сверхвысокой производительности. TimesTen сосредоточивала свои усилия на высокопроизводительных приложениях для телекоммуникаций и финансового сектора, а кроме того, разработала версию своего программного обеспечения, работающую в качестве кеширующего уско-

рителя для Oracle. После приобретения компании, Oracle продолжает выпускать Oracle TimesTen как автономную базу данных в памяти, а также ведет работы по более тесной интеграции ее в качестве кеша для своей флагманской СУБД Oracle.

В 2006 году Oracle приобрела SleepyCat Software и ее коммерческую версию базы данных с открытым кодом BerkeleyDB. BerkeleyDB представляет собой популярную базу данных с открытым кодом, популярность которой основана на исторической роли Калифорнийского университета в Беркли в качестве генератора новых идей в области баз данных. Oracle продолжает выпуск продукта с открытым кодом, который является ключевой частью инициатив Oracle на рынке встраиваемых баз данных.

ParAccel Inc. (ParAccel)

ParAccel является компанией с венчурным капиталом, нацеленной на рынок хранилищ данных и интеллектуальных ресурсов предприятия. Главный инженер компании Барри Зан (Barry Zane) ранее занимал ту же должность в Netezza, очень успешной компании по производству устройств для хранилищ данных. В отличие от Netezza, ParAccel занимается только программным обеспечением, используя архитектуру вычислений с высокой степенью параллелизма для кластеров недорогих высокопроизводительных серверов. Блестящей карьеров в области баз данных обладает и вице-президент ParAccel Брюс Скотт (Bruce Scott), который был одним из четырех сооснователей Oracle Corporation и работал над несколькими первыми версиями базы данных Oracle.

ParAccel делает упор на архитектурный подход для достижения масштабируемости массивных хранилищ данных. Хранение баз данных в ориентированном на столбцы (а не на строки) виде минимизирует количество сканируемых данных при вычислениях статистических функций. Сжатие минимизирует количество записываемых на диск данных. Поддержка операций в памяти позволяет добиться для небольших наборов данных очень высокой производительности.

PostgreSQL (PostgreSQL)

PostgreSQL — одна из наиболее популярных реализаций SQL-базы данных с открытым кодом, которая послужила базой для нескольких коммерческих продуктов. Исходная реализация Postgres была выполнена в Калифорнийском университете в Беркли и была задумана как преемник более раннего проекта Berkeley Ingres. Одной из основных ее целей была поддержка большого количества типов данных и взаимосвязей. В начале 1990-х годов вышло несколько версий продукта с открытым кодом под весьма либеральной лицензией BSD (Berkeley Software Distribution). Поддержка SQL была добавлена в средине 1990-х годов, и название Postgres было заменено на PostgreSQL, чтобы отразить это нововведение. Разработка Postgres вскоре перешла к сообществу разработчиков PostgreSQL, которое остается активным и сегодня.

Одной из наиболее ранних коммерциализаций PostgreSQL была объектно-реляционная база данных Illustra Information Technologies, которая затем была приобретена Informix в 1997 году, а ее возможности были интегрированы в базу данных Informix. PostgreSQL была также использована Sun Microsystems и стала продаваться в качестве части операционной системы Solaris. Позже EnterpriseDB

коммерциализировала и расширила PostgreSQL для приложений обработки транзакций, Greenplum — для хранилищ данных, а Truviso — для управления потоковой базой данных. Код PostgreSQL остается популярной основой для баз данных в различных системах и приложениях.

Streambase Systems (Streambase)

Компания Streambase была основана Майклом Стоунбрейкером (Michael Stonebreaker), бывшим профессором в Беркли, который также принимал участие в таких базах данных, как Ingres и Illustra. Как ясно из названия, фирма сосредоточивает усилия на применении управления данными на базе SQL к очень большим "потокам" данных, обычно поступающим по сети.

Целевыми приложениями данной базы данных является обслуживание финансовой сферы, включая обработку и анализ рыночных данных в реальном времени и обработку заказов. Сложная обработка событий важна также для ряда правительственных приложений, в которых большие объемы сетевой информации должны анализироваться на предмет выявления шаблонов и отклонений.

Sun Microsystems (MySQL)

MySQL — пожалуй, наиболее популярная SQL-база данных с более чем 10 миллионами инсталляций. Ее популярность особенно сильно выросла с развитием Интернета, поскольку именно эта база данных стала выбором множества веб-сайтов, где требовалось управление данными. Эта роль отражена в аббревиатуре "LAMP", которая означает типичный веб-сайт, работающий под операционной системой Linux, на базе веб-сервера Apache с базой данных MySQL и связывающим все это воедино языком программирования PHP/Python/Perl. Эта модель используется некоторыми очень мощными сайтами, такими как Facebook, Wikipedia, YouTube и частично — системой Google.

Изначально MySQL разрабатывалась и распространялась одноименной компанией MySQL AB из Швеции. Базовая версия базы данных доступна бесплатно для исследовательских целей и персонального использования в соответствии с лицензией GNU Public License (GPL). Коммерческое применение MySQL требует лицензионной платы, но очень небольшой, что способствует широкому распространению базы данных.

Одной из уникальных особенностей MySQL является четкое разделение высокоуровневых функций базы данных (включая работу с языком SQL) и низкоуровневых операций хранения данных, управляющих физическим сохранением и выборкой информации. Со временем было разработано несколько вариантов механизма хранения, оптимизированных для разных целей. Большие и более сложные механизмы обеспечивают дисковое хранение и поддержку ACID-транзакций для многопользовательских приложений. Меньший по размеру механизм обеспечивает хранение на диске при неполной поддержке транзакций, что больше подходит для однопользовательских приложений и для разработки программного обеспечения. Один из наиболее популярных ранних механизмов хранения — InnoDB — был приобретен в 2006 году компанией Oracle (в то время ходили слухи, что Oracle сделала это, чтобы воспрепятствовать популярности MySQL, однако Oracle продолжила выпуск версии InnoDB для поддержки MySQL).

В начале 2008 года Sun Microsystems заявила о покупке MySQL и ее использовании вместе с языком программирования Sun Java для развития программного обеспечения с открытым кодом. Это, с одной стороны, сделало MySQL частью гораздо большей организации с огромными ресурсами, но, с другой, — связало ее по рукам и ногам, что привело к тому, что в начале 2009 года компанию покинули два ее основателя.

Sybase, Inc.

История Sybase началась в средине 1980-х годов, примерно через десять лет после появления первых реляционных баз данных. Основатели компании и многие из ее ранних сотрудников были ранее сотрудниками других производителей СУБД, и для них Sybase представляла собой вторую или третью разрабатываемую РСУБД. Sybase позиционировала свой продукт как "РСУБД для онлайн-приложений". Ее основные возможности включают следующее.

- Архитектура "клиент/сервер", где клиент работает на рабочей станции Sun или VAX или персональном компьютере, а сервер — на машине VAX/VMS или Sun.
- Многопоточный сервер с собственным управлением задачами и вводом-выводом для достижения максимальной эффективности.
- Программное API, вместо используемого большинством производителей встроенного SQL-интерфейса.
- Хранимые процедуры, триггеры и диалект Transact-SQL, который расширяет SQL до полноценного серверного языка программирования.

Все это сделало Sybase наиболее продвинутой для своего времени базой данных. Технологическое лидерство, агрессивный маркетинг, большой капитал — все это привлекло к Sybase внимание и способствовало успеху. В последующем СУБД Sybase, переименованная в SQL Server, была перенесена под OS/2 и продавалась Microsoft производителям компьютеров (вместе с OS/2). Далее пути SQL Server и Sybase разошлись. Со временем Sybase расширила свою СУБД, приобретая средства разработки и другие программные продукты. Между третьей и четвертой редакциями СУБД Sybase была признана продуктом уровня предприятия.

В настоящее время флагманская СУБД Sybase (переименованная в Adaptive Server Enterprise) представляет собой полнофункциональную базу данных уровня предприятия для серверов UNIX. У компании большое количество клиентов, в основном, в финансовой и технологической сферах, а на рынке она занимает третье место после Oracle и IBM. Два дополняющих продукта обеспечивают расширенные возможности реплицирования (Replication Server), для связи распределенных баз данных, и интеграции (Open Server), для соединения разнородных сред баз данных. Особенного успеха Sybase добилась на рынке мобильных и встраиваемых баз данных, где ее SQL Anywhere является лидером.

Teradata Corporation (Teradata)

Teradata — один из основных игроков на рынке хранилищ данных, известный поддержкой крупнейших хранилищ данных в мире. Среди клиентов компании — некоторые из крупнейших торговых организаций, телекоммуникационные ком-

пании, банки и авиалинии. Репутация компании выросла после того, как она была выбрана для управления хранилищами данных розничных продаж корпорации Wal-Mart. Устройства для хранилищ данных Teradata объединяют специализированное аппаратное обеспечение и программное обеспечение с высокой степенью распараллеливания.

Teradata выросла из исследовательского проекта Калифорнийского технологического института в конце 1970-х годов, сосредоточившись на архитектурах с высокой степенью параллельности и их применении в управлении данными. В конце 1980-х годов компания могла похвастаться терабайтовыми хранилищами данных, и с этого времени хранилища данных стали рассматриваться как отдельный сегмент рынка баз данных. В конце 1991 года Teradata была приобретена NCR Corporation, которая в то время была дочерним предприятием телекоммуникационного гиганта AT&T.

Полтора десятилетия спустя Teradata стала доминировать на рынке хранилищ данных. В это время NCR отделилась от AT&T. Teradata вышла из состава NCR в конце 2007 года и теперь представляет собой автономную компанию, занимающуюся исключительно хранилищами данных. В настоящее время компания штурмует петабайтовый (один петабайт равен 1024 терабайт) барьер.

Truviso, Inc. (Truviso)

Truviso — компания с венчурным капиталом, которая сосредоточила свои усилия на потоковых базах данных. Подобно многим другим фирмам-новичкам, она основана профессором из Беркли Майклом Франклином (Michael Franklin) и базируется на потоковой базе данных Berkeley Telegraph, построенной на другой базе данных из Беркли — Postgres. Компания занимается приложениями для анализа данных, использующих подход "непрерывного анализа". В отличие от традиционного подхода, когда данные сначала накапливаются, а затем анализируются, этот подход анализирует данные постоянно, по мере их поступления. Основной продукт Truviso, TruCQ, использует SQL и дополняется TruLink, который связывает его с популярными источниками данных, и TruView, обеспечивающим визуализацию данных.

Vertica Systems (Vertica)

Vertica — еще одна фирма, организованная профессором из Беркли Майклом Стоунбрейкером (Michael Stonebreaker), который за последние 30 лет участвовал в ряде других работ над базами данных, а в настоящее время является адъюнкт-профессором Массачусетсского технологического института. База данных Vertica использует архитектуру, которая быстро стала популярной, объединяя постолбчатую организацию данных и их сжатие для снижения количества записей на диск. База данных использует стандартные интерфейсы ODBC и JDBC и предназначена для хранения больших объемов данных — от сотен гигабайт до сотен терабайт.

Vertica предлагает свою базу данных в трех разных вариантах: Vertica Database — программный продукт для работы в грид-системах на базе серверов Linux; Vertica Analytic Database Appliance — комбинирует базу данных Vertica с Red Hat Linux и ячеечным сервером HP; и, наконец, Vertica Analytic Database for the Cloud — работает на Amazon EC2 Elastic Compute Cloud, используя подход программного обеспечения в виде службы (SaaS).

Xeround (Xeround Intelligent Data Grid)

Израильская компания Xeround предлагает свой продукт Xeround Intelligent Data Grid (IDG), предназначенный для приложений оперативной обработки транзакций. В нем используется горизонтальное распределение данных в пределах сети процессоров, основанное на быстрой трансляции первичных ключей базы данных в сетевые адреса серверов. Это обеспечивает малые задержки при обращении к данным и высокую степень масштабируемости. В дополнение к SQL API, IDG предлагает XQuery и собственные API. Репликация в пределах грида обеспечивает избыточность и высокую доступность данных.

ПРИЛОЖЕНИЕ B

Синтаксис SQL

В стандарте ANSI/ISO синтаксис языка SQL описывается с помощью формальной BNF-нотации (форма Бэкуса-Наура). К сожалению, по определенным причинам стандарт является трудным для чтения и восприятия. Во-первых, в стандарте язык описывается по принципу "снизу вверх", а не "сверху вниз", что не позволяет получить общее представление об инструкциях SQL. Во-вторых, в стандарте употребляются не всегда понятные термины (например, *предикат* или *табличное выражение*). И наконец, описание синтаксиса в стандарте имеет очень много уровней и затрагивает даже самые мелкие детали языка, скрывая тем самым общую, относительно простую его структуру. В данном приложении представлены упрощенные синтаксические диаграммы инструкций SQL в том виде, в каком они реализованы в большинстве СУБД. В частности, эти упрощения включают следующее.

- Описаны те части языка, поддержка которых требуется от СУБД для совместимости со стандартом SQL на уровне Entry Level, плюс те элементы уровней Intermediate Level и Full Level, которые реализованы в большинстве СУБД.
- Не описан модульный язык, поскольку во всех существующих на сегодняшний день СУБД он заменен встроенным SQL или API-функциями.
- Компоненты языка названы своими обычными именами, а не техническими терминами, используемыми в стандарте.

Для BNF-нотации в данном приложении приняты следующие соглашения.

- Все ключевые слова набраны `ПРОПИСНЫМИ БУКВАМИ МОНОШИРИННЫМ ШРИФТОМ`.
- Элементы синтаксиса набраны *`курсивом`*.
- Выражение вида *список_элементов* означает *элемент* или список *элементов*, разделенных запятыми.

- Символ вертикальной черты (|) означает возможность выбора между несколькими вариантами.
- Квадратные скобки ([]) представляют необязательные элементы.
- Фигурные скобки ({}) указывают на необходимость выбора между обязательными элементами.

Инструкции DDL

Эти инструкции определяют структуру базы данных, включая ее таблицы, представления и объекты, специфичные для той или иной СУБД.

```
CREATE TABLE таблица (список_элементов_определения_таблицы)

DROP TABLE таблица [параметры_удаления]

CREATE VIEW представление [(список_столбцов)]
    AS спецификация_запроса
 [WITH CHECK OPTION]

DROP VIEW представление [параметры_удаления]

CREATE тип_объекта имя_объекта [спецификация_объекта]

DROP тип_объекта [параметры_удаления]

ALTER тип_объекта действие_при_изменении
```

Ключевые слова, используемые для задания объектов базы данных (*тип_объекта*), зависят от конкретной СУБД. К типичным объектам относятся таблицы (TABLE), представления (VIEW), схемы (SCHEMA), хранимые процедуры (PROCEDURE) и функции (FUNCTION), триггеры (TRIGGER), домены (DOMAIN), индексы (INDEX), а также именованные области хранения, поддерживаемые самой СУБД. Синтаксис задания этих объектов также специфичен для конкретной СУБД.

Элементы языка, используемые в инструкциях CREATE, DROP и ALTER, приведены в следующей таблице.

Элемент	Синтаксис
элемент_определения_таблицы	*определение_столбца* | *ограничение_таблицы*
определение_столбца	*столбец тип_данных* [DEFAULT {*литерал* | USER | NULL}] [*список_ограничений_столбца*]
ограничение_столбца	[CONSTRAINT *имя_ограничения*] {NOT NULL | *уникальность* | *ссылка_внешнего_ключа* | *условие_на_значение*} [*проверка_ограничения*]
ограничение_таблицы	[CONSTRAINT *имя_ограничения*] {*уникальность* | *ограничение_внешнего_ключа* | *условие_на_значение*} [*проверка_ограничения*]

Элемент	Синтаксис
уникальность	UNIQUE *(список_столбцов)* | PRIMARY KEY *(список_столбцов)*
ограничение_внешнего_ключа	FOREIGN KEY *(список_столбцов) ссылка_внешнего_ключа*
ссылка_внешнего_ключа	REFERENCES *таблица* [*(список_столбцов)*] [MATCH FULL | PARTIAL] [ON DELETE *действие_при_удалении*]
действие_при_удалении	CASCADE | SET NULL | SET DEFAULT | NOT NULL
условие_на_значение	CHECK *(условие_отбора)*
проверка_ограничения	[INITIALLY IMMEDIATE | INITIALLY DEFERRED] [[NOT] DEFERRABLE]
параметры_удаления	CASCADE | RESTRICT

Инструкции управления доступом

Доступом к объектам и службам базы данных управляют следующие инструкции.

```
GRANT {ALL PRIVILEGES | список_привилегий }
   ON {таблица | тип_объекта имя_объекта }
   TO {PUBLIC | список_пользователей }
[WITH GRANT OPTION]

REVOKE {ALL PRIVILEGES | список_привилегий }
    ON {таблица | тип_объекта имя_объекта }
  FROM {PUBLIC | список_пользователей }
 [WITH GRANT OPTION]
```

Ключевые слова для указания типов объектов базы данных (*тип_объекта*) зависят от конкретной СУБД. Типичными объектами при назначении привилегий являются TABLE, VIEW, SCHEMA, PROCEDURE, FUNCTION, TRIGGER, DOMAIN, INDEX и именованные области хранения, поддерживаемые СУБД. Синтаксис SQL, используемый для указания этих объектов, специфичен для поддерживающих их СУБД. Конкретные поддерживаемые привилегии также зависят от конкретных СУБД и объектов.

Элемент языка и синтаксис в инструкциях GRANT и REVOKE приведены в следующей таблице.

Элемент языка	Синтаксис
привилегия	SELECT | DELETE | UPDATE[(*список_столбцов*)] | INSERT[(*список_столбцов*)] | EXECUTE

Основные инструкции DML

"Одиночная" инструкция SELECT возвращает одну запись и помещает ее компоненты в базовые переменные (во встроенном SQL) или в переменные хранимой процедуры.

```
SELECT [ALL | DISTINCT]{список_возвращаемых_элементов | *}
  INTO список_базовых_переменных
  FROM список_ссылок_на_таблицы
[WHERE условие_отбора]
```

"Интерактивная" инструкция SELECT возвращает набор записей произвольного размера (допустима только в интерактивном режиме; во встроенном SQL и в хранимых процедурах необходимо использовать специальные инструкции, оперирующие курсорами).

```
 SELECT [ALL | DISTINCT]{список_возвращаемых_элементов | *}
   INTO список_переменных
   FROM список_ссылок_на_таблицы
 [WHERE условие_отбора]
 [GROUP BY список_ссылок_на_столбцы]
[HAVING условие_отбора]
 [ORDER BY список_сортируемых_элементов]
```

Следующие инструкции модифицируют информацию в базе данных.

```
INSERT INTO таблица [(список_столбцов)]
    {VALUES (список_вставляемых_элементов) | выражение_запроса}

DELETE FROM таблица [WHERE условие_отбора]

UPDATE таблица
   SET список_выражений_присваивания [WHERE условие_отбора]
```

Инструкции обработки транзакций

Эти инструкции сигнализируют об окончании транзакции SQL.

```
COMMIT [WORK]

ROLLBACK [WORK]
```

Инструкции для работы с курсорами

Эти инструкции программного SQL служат для выборки данных и для их позиционного обновления.

```
DECLARE курсор [SCROLL] CURSOR FOR выражение_запроса
 [ORDER BY список_сортируемых_элементов]
 [FOR {READ ONLY | UPDATE [OF список_столбцов]}]

OPEN курсор
```

```
CLOSE курсор

FETCH [[направление_выборки] FROM] курсор
 INTO список_базовых_переменных

DELETE FROM таблица
 WHERE CURRENT OF курсор

UPDATE таблица
   SET список_выражений_присваивания
 WHERE CURRENT OF курсор
```

Необязательное *направление_выборки* задается следующим образом.

```
NEXT | PRIOR | FIRST | LAST |
      ABSOLUTE номер_записи | RELATIVE номер_записи
```

Номер записи может быть задан в виде базовой переменной или литерала.

Выражения запросов

Стандарт SQL предоставляет богатый набор выражений для определений запросов, от простейших до сложных выражений запросов с использованием операций реляционных баз данных для объединения результатов простых запросов.

Базовая спецификация запроса имеет следующий вид.

```
 SELECT [ALL | DISTINCT]{список_возвращаемых_элементов | *}
   FROM список_ссылок_на_таблицы
  [WHERE условие_отбора]
  [GROUP BY список_ссылок_на_столбцы]
[HAVING условие_отбора]
```

Ссылки на таблицы в предложении FROM могут быть такими:

- *простая ссылка на таблицу*, содержащая (возможно, квалифицированное) имя таблицы;
- *возвращенная ссылка на таблицу*, содержащая подчиненный запрос (см. ниже), результатом которого является таблица (не все СУБД допускают использование подчиненных запросов в предложении FROM);
- *ссылка на соединенную таблицу* (см. ниже), которая объединяет данные из двух и более таблиц с применением реляционных операторов OUTER JOIN, INNER JOIN или иных операторов соединения. Не все СУБД допускают объединение таблиц в предложении FROM.

В соответствии с синтаксисом SQL, объединение таблиц задается следующим образом (на практике в различных СУБД поддерживаются разные типы объединений и допускаются отличия от приведенного ниже синтаксиса).

Элемент	Синтаксис
соединенная_таблица	*внутреннее_соединение* | *внешнее_соединение* | *расширенное_соединение* | *перекрестное_соединение* | (*соединенная_таблица*)
внутреннее_соединение	*ссылка_на_таблицу* [NATURAL] [INNER] JOIN *ссылка_на_таблицу* | *ссылка_на_таблицу* [INNER] JOIN *ссылка_на_таблицу* [*спецификация_соединения*]
внешнее_соединение	*ссылка_на_таблицу* [NATURAL] [LEFT|RIGHT|FULL] OUTER JOIN *ссылка_на_таблицу* | *ссылка_на_таблицу* [LEFT|RIGHT|FULL] OUTER JOIN *ссылка_на_таблицу* [*спецификация_соединения*]
расширенное_соединение	*ссылка_на_таблицу* UNION JOIN *ссылка_на_таблицу*
перекрестное_соединение	*ссылка_на_таблицу* CROSS JOIN *ссылка_на_таблицу*
спецификация_соединения	ON *условие_отбора* | USING (*список_столбцов*)

Стандарт SQL разрешает комбинировать базовые спецификации запроса с помощью операций UNION, EXCEPT и INTERSECT, создавая выражение запроса. Заключенное в скобки, выражение запроса становится подчиненным запросом, который может присутствовать в различных местах инструкции SELECT (например, в определенных условиях отбора в предложении WHERE).

Эти операции поддерживаются не во всех СУБД. Упрощенный их синтаксис (без учета приоритета операций) описан ниже.

Элемент	Синтаксис
выражение_запроса	*таблица* | *соединенная_таблица* | *выражение_сложения* | *выражение_вычитания* | *выражение_пересечения* | (*выражение_запроса*)
выражение_сложения	*выражение_запроса* UNION [ALL] [*спецификация_соответствия*] *выражение_запроса*
выражение_вычитания	*выражение_запроса* EXCEPT [ALL] [*спецификация_соответствия*] *выражение_запроса*
выражение_пересечения	*выражение_запроса* INTERSECT [ALL] [*спецификация_соответствия*] *выражение_запроса*
спецификация_соответствия	CORRESPONDING [BY (*список_столбцов*)]
подзапрос	(*выражение_запроса*)

Условия отбора

Эти выражения используются при выборке строк из таблиц базы данных.

Элемент языка	Синтаксис
условие_отбора	*отбираемый_элемент* | *отбираемый_элемент* {AND | OR} *отбираемый_элемент*
отбираемый_элемент	[NOT] {*проверка_отбора* | (*условие_отбора*)}
проверка_отбора	*проверка_сравнения* | *проверка_вхождения* | *проверка_подобия* | *проверка_на_null* | *проверка_на_принадлежность* | *многократная_проверка* | *проверка_на_существование*
проверка_сравнения	*выражение* {= | <<>> | << | <<= | >> | >>=} {*выражение* | *подзапрос*}
проверка_вхождения	*выражение* [NOT] BETWEEN *выражение* AND *выражение*
проверка_подобия	*ссылка_на_столбец* [NOT] LIKE *значение* [ESCAPE *значение*]
проверка_на_null	*ссылка_на_столбец* IS [NOT] NULL
проверка_на_принадлежность	*выражение* [NOT] IN {*список_значений* | *подзапрос*}
многократная_проверка	*выражение* {= | <<>> | << | <<= | >> | >>= } [ALL | ANY | SOME] *подзапрос*
проверка_на_существование	EXISTS *подзапрос*

Выражения

Эти выражения используются в списках возвращаемых элементов и условиях отбора.

Элемент языка	Синтаксис
выражение	*простое_выражение* | *простое_выражение* {+ | - | * | /} *простое_выражение*
простое_выражение	[+ | -] [*значение* | *ссылка_на_столбец* | *функция* | (*выражение*)]
значение	*литерал* | USER | *базовая_переменная* | *переменная_хранимой_процедуры*
базовая_переменная	*переменная* [[INDICATOR] *переменная*]
функция	COUNT(*) | *distinct_функция* | *all_функция*
distinct_функция	{AVG | MAX | MIN | SUM | COUNT} (DISTINCT *ссылка_на_столбец*)
all_функция	{AVG | MAX | MIN | SUM | COUNT} ([ALL] *выражение*)

Элементы инструкций

Эти элементы могут присутствовать в различных инструкциях SQL.

Элемент языка	Синтаксис
выражение_присваивания	*столбец* = {*выражение* | NULL | DEFAULT}
сортируемый_элемент	{*ссылка_на_столбец* | *целое_число*} [ASC | DESC]
вставляемый_элемент	{*значение* | NULL}
возвращаемый_элемент	*выражение*
ссылка_на_таблицу	*таблица* [*псевдоним*]
ссылка_на_столбец	[{*таблица* | *псевдоним*}.] *столбец*

Простые элементы

Эти элементы являются именами и константами, присутствующими в инструкциях SQL.

Элемент языка	Синтаксис
таблица	Имя таблицы
столбец	Имя столбца
пользователь	Имя пользователя базы данных
переменная	Имя базовой переменной или переменной хранимой процедуры
литерал	Числовая или строковая константа, заключенная в кавычки
целое_число	Целочисленная константа
тип_данных	Тип данных SQL
псевдоним	Идентификатор SQL
курсор	Имя курсора (идентификатор SQL)

Предметный указатель

@

@@ERROR, 715
@@SQLSTATUS, 715

A

ACID-тест, 334
Ajax, 775
ALL, 246
ALTER TABLE, 386
ALTER TRIGGER, 728
AND [NO] CHAIN, 339
ANY, 243
AVG, 208

B

BEGIN DECLARE SECTION, 519
BEGIN TRANSACTION, 336
BETWEEN, 136

C

CASCADE, 307; 309
CASE, 261
CAST, 259
CHECK, 321
CLI, 60; 620
  Атрибуты, 648
  Динамические запросы, 638
  И курсоры, 637
  Информация о реализации, 649
  Обработка запросов, 633
  Обработка инструкций, 628
  Обработка ошибок, 646
  Описатель, 645
  Сеанс подключения, 627
  Среда SQL, 626
  Структуры, 624
  Транзакция, 632
CLOSE, 536; 573
COALESCE, 263
CODASYL, 81
COMMENT ON, 477
COMMIT, 336
COUNT, 210
COUNT(*), 211
CREATE ALIAS, 392
CREATE ASSERTION, 390
CREATE DATABASE, 373
CREATE DOMAIN, 390
CREATE FUNCTION, 730
CREATE INDEX, 395
CREATE PROCEDURE, 695; 730
CREATE SCHEMA, 410
CREATE TABLE, 375
CREATE TRIGGER, 721; 722
CREATE TYPE, 835
CREATE VIEW, 419
CROSS JOIN, 194
CURRENT DATE, 280

D

DB2, 54
dblib, 596–619
DEALLOCATE PREPARE, 580
DECLARE CURSOR, 533; 569
  FOR UPDATE, 542
DECLARE TABLE, 509
DEFERRABLE, 323
DELETE, 286
DESCRIBE, 563; 567
DISTINCT, 130; 216
DOM, 875
DROP ALIAS, 392
DROP INDEX, 397
DROP PROCEDURE, 695
DROP TABLE, 385
DROP VIEW, 432

E

END DECLARE SECTION, 519
Enterprise Java Beans, 762
EXCEPT, 271
EXCEPTION, 715
EXEC SQL, 506
EXECUTE, 552; 555; 698
EXECUTE IMMEDIATE, 549; 579
EXISTS, 240

F

FETCH, 535; 572
FOR UPDATE, 542
FOREIGN KEY, 379
FROM, 123

G

GET DIAGNOSTICS, 513
GRANT, 439; 450
GROUP BY, 216

H

HAVING, 253
HTML, 858

I

IBM, 53; 936
IMS, 80
IN, 138; 239
Informix, 474; 476; 791; 836
Ingres, 53; 360; 383
INNER, 185
INSERT
  Многострочная, 282
  Однострочная, 278
INTERSECT, 271
IS, 146
IS NULL, 142

J

J2EE, 762
Java2 Enterprise Edition, 762
JDBC, 497; 665
  Архитектура, 667
    второго типа, 669
    первого типа, 668
    третьего типа, 669
    четвертого типа, 670
  Источник данных, 687
  Курсор произвольного доступа, 684
  Метаданные, 685
  Мост JDBC/ODBC, 668
  Набор строк, 687
  Обработка
    инструкций, 673
    ошибок, 683
  Объекты, 671
  Подготовленные инструкции, 678
JOIN, 163; 271

L

LAMP, 775
LIKE, 140
LOCK TABLE, 357

M

MATCH FULL, 318
MATCH PARTIAL, 319
MAX, 209
Microsoft, 938
MIN, 209
MySQL, 222; 260; 357; 374; 385; 400; 462; 942

N

NATURAL JOIN, 165
NOT DEFERRABLE, 323
NOT FOUND, 526
NOT NULL, 297
NULL, 135; 302
NULLIF, 263

O

OCI, 658
  Большие объекты, 664
  Дескрипторы, 659
  И транзакция, 663
  Обработка ошибок, 664
    результатов, 663
  Описатель, 663
  Определение, 663
  Подключение, 661
  Привязка, 663
  Функции, 659
ODBC, 60; 496; 619
  Архитектура, 652
  Асинхронное выполнение, 656
  Блочный курсор, 657
  Группировка подключений, 655
  Диспетчер драйверов, 651
  Драйвер, 651
  Закладка, 657
  И системные каталоги, 653
  Массив параметров, 657
  Пакетное выполнение инструкций, 656
  Подключение, 655
  Расширенные возможности, 654
  Смещение привязки, 656
  Структура, 651
OLAP, 742
OLTP, 742
ON EXCEPTION, 716
OPEN, 534; 570
Oracle, 53; 190; 199; 222; 260; 336; 357; 373; 384; 385; 399; 447; 462; 470; 476; 574; 658; 790; 939
  Формат SQLDA, 576
Oracle Application Server, 762
ORDER BY, 146
OS/2, 69; 938
OUTER, 184

P

PL/SQL, 694; 726
PREPARE, 552; 554
PRIMARY KEY, 378

R

RELEASE SAVEPOINT, 336
RESTRICT, 307; 309
REVOKE, 439; 454
ROLLBACK, 336

S

SaaS, 913
SAVE TRANSACTION, 336
SAVEPOINT, 336; 338
SAX, 875
SELECT, 120
  DISTINCT, 130
  FROM, 123; 177
  GROUP BY, 216
  HAVING, 225
  ORDER BY, 146
  WHERE, 131
  Список выбора, 123
SET CONSTRAINTS, 336
SET DEFAULT, 308; 309
SET NULL, 307; 309
SET TRANSACTION, 336
SGML, 858
SMP, 914
SPL, См. Язык хранимых процедур
SQL
  IS, 146
  IS NULL, 142
  JOIN, 163
  NULL, 118
  UNION, 150
  Безопасность, 437–462
  Вставка данных, 278
  Встроенная функция, 116
  Встроенный, 496
  Выборка данных, 42
  Выражения, 115
    Табличные, 267
  Динамический, 547
  Добавление данных, 45; 278

Инструкция, 97
DELETE, 286
UPDATE, 290
SELECT, 120
Ключевые слова, 99
Константы, 112
Область данных, 555
дескриптора, 556
Обновление данных, 46; 290
Каскадное, 311
Переносимость, 61
Препроцессор, 502
Программный, 495
Свойства, 34
Создание базы данных, 47
Сравнение строк, 267
Среда, 408
Стандарты, 56
Статический, 497; 545
Типы данных, 106
Удаление всех строк, 288
данных, 46; 286
Каскадное, 311
SQL Access Group, 59
SQL Server, 190; 222; 374; 385; 398; 447; 596
Функции API, 596
SQL/CLI, 620
SQL/PSM, 729
Обработка ошибок, 734
Обработчик состояния ошибки, 735
SQLCODE, 510; 526
SQLDA, 555; 567
В Oracle, 576
SQLSTATE, 512; 526
SQL-инжекция, 551
START TRANSACTION, 336
SUM, 207

T

TABLE OF, 846
TO SAVEPOINT, 339
Transact-SQL, 697; 722

U

UNION, 150; 271
UPDATE, 290
USING, 165

V

VARRAY, 846

W

WHENEVER, 515
WHERE, 131
WHERE CURRENT OF, 612
WITH CHECK OPTION, 431
WITH CUBE, 222
WITH ROLLUP, 222
WORK, 339

X

XML, 857; 859–61
DOM, 875
SAX, 875
XQuery, 888
XSLT, 887
Анализатор, 875
Атрибут, 864
И метаданные, 878
Определение типа документа, 879–881
Словарь, 886
Структура, 863
Элемент, 864
XML Schema, 879; 881–887
Типы данных, 882

А

Агрегирующая функция, 205
Архитектура базы данных, 400–406
"клиент/сервер", 64
Аутентификация, 440

Б

База данных, 30; 374
IMS, 80
Архитектура, 400–406
В памяти, 897
Встраиваемая, 905
Иерархическая, 79
Мобильная, 906
Объектно-ориентированная, 822
Объектно-реляционная, 826
Потоковая, 900
Производители, 933
Распределенная, 781–818; 917
Режим автосохранения, 341
Резидентная, 897
Реляционная, 50; 83
Репликация, 793
Создание, 371–412
Структура, 400
Схема, 406; 409
Базовая переменная, 517
Бизнес-правило, 325; 718
Блокировка, 350
Настройка, 361
Явная, 356
Блочный курсор, 657
Большой объект, 828

В

Веб-сайт, 759
Веб-служба, 75
Виртуальная таблица, 433
Внешнее соединение, 183
Внешний ключ, 89; 160
Внешняя ссылка, 236
Внутренний запрос, 232
Встроенная функция, 116
Встроенный SQL, 495
SQLCODE, 510
SQLSTATE, 512
Автоматическая перекомпоновка, 505
Базовая переменная, 517
Выборка данных, 523
Выполнение, 504
Защита данных, 504
Курсор, 530
Область коммуникации SQL, 510
Обработка ошибок, 509

Объявление таблицы, 509
Переменная-индикатор, 522
Структура данных, 528
Выборка данных, 42
Во встроенном SQL, 523
Выражения, 115
Табличные, 267

Г

Группа пользователей, 442
Группировка подключений, 655

Д

Декартово произведение, 180
Декларативный язык, 31
Демаршалинг, 876
Дескриптор, 625; 823
Применения, 765
Добавление данных, 45
Домен, 260; 296; 300; 319; 377; 390; 490
Драйвер, 619

Е

Естественное соединение, 165

Ж

Журнал транзакций, 341

З

Закладка, 657
Запись, 832
Запрос, 31
Внутренний, 232
Выражение, 271
Вычисляемые столбцы, 126
Использование NULL, 135
Исходные таблицы, 123
Многотабличный, 172
Непрерывный, 902
Объединение результатов, 150
Вложенное, 153
Повторяющиеся строки, 152
Сортировка, 153
Подчиненный, См. Подзапрос
Простой, 126
Результирующий набор, 123
С группировкой, 216
Сортировка результатов, 146
Спецификация, 269
Удаленный, 804

И

Идентификатор
Авторизации, 440
Пользователя, 439
Изменение таблицы, 386
Имена, 104
Квалифицированные, 105
Уникальные, 736
Индекс, 393
Инкапсуляция, 823; 852
Инструкция
DELETE, 286
Поисковая, 287
Предложение WHERE, 286
С подзапросом, 288
INSERT
Многострочная, 282
Однострочная, 278
SELECT, 120
DISTINCT, 130
Предложение FROM, 123; 177
GROUP BY, 216
HAVING, 225
ORDER BY, 146
WHERE, 131
Список выбора, 123
UPDATE, 290
Поисковая, 292
С подзапросом, 293
Интернет, 33
Интерфейс вызовов Oracle, См. OCI
Информационная схема, 485

К

Канал связи, 790
Каскадные удаления и обновления, 311
Каталог, 409
Системный, 463
Кеширование данных, 815
Класс, 822
Ключ
Внешний, 89
Первичный, 87
Суррогатный, 748
Ключевые слова SQL, 99
Кодда правила, 90
Команда, 99
Константы, 112
Конструктор, 836
Строки, 264
Таблицы, 267
Коррелированные подзапросы, 250
Куб фактов, 746
Курсор, 530
Блочный, 657
В JDBC, 684
В хранимой процедуре, 712
Выборка, 535
Закрытие, 536
И транзакции, 542
Объявление, 533
Открытие, 534
С произвольным доступом, 536
Удаление и обновление данных, 538
Курсоры
В CLI, 637

Л

Левое внешнее соединение, 186; 196
Литерал, 112
Локатор, 830

М

Маркер параметра, 551; 678
Маршалинг, 876
Материализованное представление, 433; 795
Метаданные, 685
Метод, 852
  Перегрузка, 854
Модель данных
  Иерархическая, 79
  Объектная, 822
  Реляционная, 77; 83
  Сетевая, 81
Модель транзакций, 336
Модуль, 442

Н

Набор строк, 687
Наследование, 823; 837
  Табличное, 839

О

Обновление данных, 46
  Каскадное, 311
  Позиционное, 539
Обработка ошибок, 734
Обработчик состояния ошибки, 735
Объединение результатов запросов, 150
  Вложенное, 153
  Повторяющиеся строки, 152
  Сортировка, 153
Объектная модель, 55
Оперативная обработка транзакций, См. OLTP
Описатель, 645; 663
Определение столбцов, 375
Отложенная передача параметров, 632
Отложенная проверка ограничений, 322
Отношение, 87

П

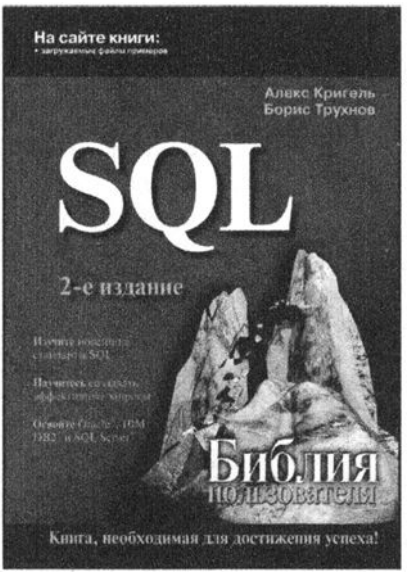
Пакет инструкций, 600
Параллельные транзакции, 348
Первичный ключ, 87; 160; 302; 378
Перегрузка имен, 736
Перекрестное соединение, 194; 195
Переменная-индикатор, 522
Персистентность, 334
План выполнения, 498
Подзапрос, 231
  В инструкции
    DELETE, 288
    UPDATE, 293
  В предложении
    FROM, 273
    HAVING, 253
  Вложенный, 249
  И соединение таблиц, 247
  Коррелированные подзапросы, 250
  Многократное сравнение, 243
  Табличный, 268
  Условия отбора, 236
Подкласс, 822
Подстановочные знаки, 140
Подчиненный запрос, См. Подзапрос
Полиморфизм, 736
Полное внешнее соединение, 185; 196
Правила Кодда, 90
  удаления и обновления, 307; 388
Правое внешнее соединение, 187; 196
Представление, 415
  Вертикальное, 421
  Горизонтальное, 420
  Материализованное, 433; 795
  Обновимость, 428
  С проверкой, 431
  Сгруппированное, 423
  Смешанное, 423
  Соединенное, 425
  Создание, 419
  Удаление, 432
Преобразоване типов данных, 259
Препроцессор, 502
Привилегия, 444
  ALTER, 447
  DELETE, 444
  EXECUTE, 447; 505
  INDEX, 447
  INSERT, 444
  REFERENCES, 446
  SELECT, 444
  UPDATE, 444
  USAGE, 446
  Владения, 446
  Отмена, 454
  Передача, 452
  Предоставление, 450
Привязка столбцов, 606
Примечание, 477
Производительность, 178
Псевдоним, 176; 177; 392
Путь, 737

Р

Размер диагностики, 338
Разметка, 857
Распределенная транзакция, 806
Расширенное соединение, 194; 197
Результирующий набор, 123
Реляционная модель данных, 50; 77; 83; 184
Реляционные базы данных, 30
Репликация, 793
  Двунаправленная, 797
  Зеркальная, 802
  Схема, 800
  Таблиц, 795
Роль, 443; 460

SQL. БИБЛИЯ ПОЛЬЗОВАТЕЛЯ
2-е издание

Алекс Кригель, Борис Трухнов



www.dialektika.com

Эта книга уникальна тем, что в каждой главе проводится сравнение реализаций стандарта SQL в трех ведущих СУБД: Oracle 11g, IBM DB2 9.5 и Microsoft SQL Server 2008. В результате получился исчерпывающий и практичный справочник для пользователей баз данных, от новичков до профессионалов. Книга удобно сочетает теорию с практикой, содержит описание новых технологий и позволит разобраться с многочисленными нюансами стандарта SQL и его реализаций.

- Изучите основы SQL и реляционных баз данных.
- Освойте работу с таблицами, представлениями, последовательностями и прочими объектами баз данных.
- Научитесь применять транзакции и блокировки в многопользовательской среде.
- Познакомьтесь с функциями, предложенными стандартом SQL и тремя ведущими производителями СУБД.
- Узнайте, как получать доступ к метаданным и реализовывать средства защиты базы данных.
- Изучите дополнительные темы: интеграция SQL в XML, бизнес-аналитика OLAP и многое другое.

ISBN 978-5-8459-1546-7

в продаже

XML
БАЗОВЫЙ КУРС
4-Е ИЗДАНИЕ

Дэвид Хантер и др.

www.dialektika.com

Эта книга представляет собой отличное пособие по XML для начинающих программистов. В нем рассказывается о том, что такое XML, как и где его следует использовать, и его место среди прочих информационных технологий. На фундаменте предыдущего издания авторы построили новую книгу, добавив в нее последние достижения XML-технологий, такие как XQuery, RSS и Atom, AJAX. В книге представлены самые последние спецификации, а также передовой опыт в решении встающих перед программистами в области XML задач. Четвертое издание книги поможет вам быстро пройти путь от начинающего программиста XML к профессионалу, свободно ориентирующемуся в самых современных технологиях и работающему с ними. Вы ознакомитесь с современным положением дел в области XML и баз данных, узнаете, как запрашивать XML-информацию, получать данные, создавать новые документы, и многое другое.

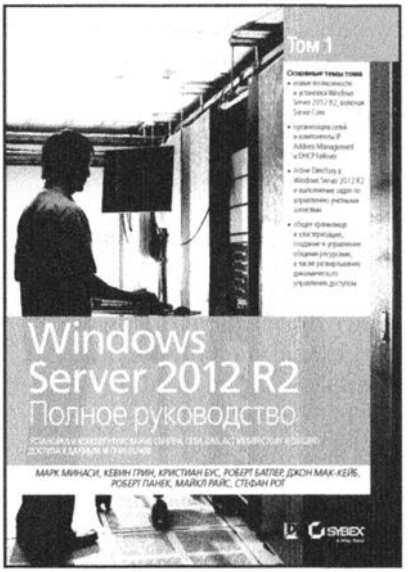
ISBN 978-5-8459-1533-7

в продаже

WINDOWS SERVER 2012 R2 ПОЛНОЕ РУКОВОДСТВО. ТОМ 1

УСТАНОВКА И КОНФИГУРИРОВАНИЕ СЕРВЕРА, СЕТИ, DNS, ACTIVE DIRECTORY И ОБЩЕГО ДОСТУПА К ДАННЫМ И ПРИНТЕРАМ

Марк Минаси и др.



www.dialektika.com

Попробуйте новый гипервизор Hyper-V, найдите новые и более простые способы дистанционного подключения к офису, а также изучите Storage Spaces — это всего лишь несколько компонентов Windows Server 2012 R2, которые подробно рассматриваются в данном обновленном издании книги признанного авторитета в области Windows Марка Минаси и команды экспертов по Windows Server, возглавляемой Кевином Грином. Основные темы книги: установка или модернизация и последующее управление сервером с Windows Server 2012 R2; настройка объединения сетевых интерфейсных плат Microsoft NIC Teaming 2012 и работа с PowerShell; установка операционной системы через графический пользовательский интерфейс или обновленную версию Server Core 2012; миграция, слияние и модификация Active Directory; управление адресным пространством с помощью IPAM; новые общие хранилища, пространства хранения и улучшенные инструменты для работы с ними; управление доступом к общим файлам — новый и усовершенствованный подход; использование и администрирование Remote Desktop, Virtual Desktop и Hyper-V.

ISBN 978-5-8459-1935-9 **в продаже**

Описание последних технологий и стандартов СУБД

Полное руководство по SQL

Материал от трех ведущих экспертов охватывает все аспекты SQL. Пересмотренное с учетом последних версий РСУБД, это руководство поясняет, как создавать, наполнять и администрировать высокопроизводительные базы данных и разрабатывать мощные и надежные приложения с использованием SQL.

Эта книга расскажет вам, как работать с командами и инструкциями SQL, создавать и настраивать реляционные базы данных, загружать и модифицировать объекты баз данных, выполнять мощные запросы, повышать производительность и выстраивать систему безопасности. Вы узнаете, как использовать инструкции DDL и применять API, интегрировать XML и сценарии Java, использовать объекты SQL, создавать веб-серверы, работать с удаленным доступом и выполнять распределенные транзакции. В этой книге вы найдете такие сведения, как описания работы с базами данных в памяти, потоковыми и встраиваемыми базами данных, базами данных для мобильных устройств, и многое другое.

- **Построение SQL-реляционных баз данных и приложений**
- **Создание, загрузка и модификация объектов баз данных с применением SQL**
- **Построение и выполнение простых, многотабличных и суммирующих запросов**
- **Реализация системы безопасности с использованием аутентификации, привилегий, ролей и представлений**
- **Оптимизация, резервное копирование, восстановление и репликация баз данных**
- **Работа с хранимыми процедурами, функциями, расширениями, триггерами и объектами**
- **Расширенная функциональность с применением API, динамического и встраиваемого SQL**
- **Описание таких вопросов, как транзакции, механизмы блокировок, материализованные представления и протокол двухфазного завершения транзакции**
- **Последние тенденции рынка и будущее SQL**

Джеймс Р. Грофф является исполнительным директором компании PBworks. Ранее Грофф занимал ту же должность в компании TimesTen, возглавляя ее в течение восьми лет до покупки компании в 2005 году корпорацией Oracle.

Пол Н. Вайнберг — старший вице-президент компании SAP. Вайнберг — эксперт в области SQL; его компания A2i, Inc. была приобретена SAP в 2004 году. Является соавтором Дж. Гроффа по предыдущим изданиям этой книги.

Эндрю Дж. Оппель — ведущий специалист по моделированию данных в Blue Shield. Кроме того, он уже более 20 лет преподает теорию баз данных в Калифорнийском университете в Беркли.

Категория: базы данных
Предмет рассмотрения: SQL
Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-907114-26-5

18095
9 785907 114265

ДИАЛЕКТИКА
http://www.williamspublishing.com

McGraw Hill
MHPROFESSIONAL.COM