Пол Нильсен — архитектор данных, разработчик, преподаватель. Весь опыт его работы воплощен в этой книге, и ее великолепных примерах. Если вы хотите изучить SQL Server 2005, то эта книга для вас.

Уэйн Снайдер, член совета директоров PASS

Пол Нильсен

MVP Microsoft® Most Valuable Professional

SQL Server™ 2005



Откройте для себя расширенные средства администрирования и программирования

Создавайте хранимые процедуры и повышайте производительность

Узнайте о новинках пакетов обновлений SP1 и SP2

Библия пользователя

Книга, необходимая для достижения успеха!

SQL Server™ 2005 Bible

Paul Nielsen



Wiley Publishing, Inc.

SQL Server™ 2005

Библия пользователя

Пол Нильсен

Москва • Санкт-Петербург • Киев
2008

ББК 32.973.26-018.2.75
Н66
УДК 681.3.07

Компьютерное издательство "Диалектика"

Главный редактор *С.Н. Тригуб*

Зав. редакцией *В.Р. Гинзбург*

Перевод с английского и редакция *С.А. Храмова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Нильсен, Пол.

Н66 Microsoft SQL Server 2005. Библия пользователя. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 1232 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1314-2 (рус.)

В этой книге содержится полное описание СУБД SQL Server 2005 с учетом дополнений и улучшений, привнесенных пакетами обновлений SP1 и SP2. Вы узнаете о ключевых принципах информационной архитектуры, заложенных в основу СУБД, об установке, обслуживании и администрировании сервера. В книге описаны языки запросов T-SQL и MDX, позволяющие осуществлять доступ к оперативным и аналитическим данным, а также управлять ими. Вы ознакомитесь с расширенными средствами бизнес-аналитики, реализованными в SQL Server 2005, освоите стратегии и средства защиты данных и самого сервера.

В книге изучаются вопросы измерения и настройки производительности SQL Server, даются практические советы по созданию планов обслуживания сервера и резервирования данных. Рассмотрены все службы, входящие в состав SQL Server 2005, а также средства интеграции с языками программирования среды .NET Framework, такими как C# и VB.NET, и другими СУБД.

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 Wiley Publishing, Inc.

Copyright © 2008 by Dialektika Computer Publishing.
Original English language edition Copyright © 2007 by Wiley Publishing, Inc.
All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Wiley Publishing, Inc.

ISBN 978-5-8459-1314-2 (рус.)

ISBN 0-7645-4256-7 (англ.)

© Компьютерное изд-во "Диалектика", 2008
перевод, оформление, макетирование
© Wiley Publishing, Inc., 2007

Оглавление

Введение 36

ЧАСТЬ I. ОСНОВЫ 41

ГЛАВА 1. Принципы информационной архитектуры 42

ГЛАВА 2. Моделирование реляционных баз данных 65

ГЛАВА 3. Архитектура SQL Server 2005 87

ГЛАВА 4. Установка SQL Server 2005 110

ГЛАВА 5. Подключение клиентского программного обеспечения 132

ГЛАВА 6. Использование утилиты Management Studio 139

ЧАСТЬ II. МАНИПУЛИРОВАНИЕ ДАННЫМИ С ПОМОЩЬЮ ИНСТРУКЦИИ SELECT 157

ГЛАВА 7. Основы выполнения запросов 158

ГЛАВА 8. Использование выражений и скалярных функций 182

ГЛАВА 9. Объединение данных 208

ГЛАВА 10. Включение данных с помощью подзапросов и CTE 233

ГЛАВА 11. Консолидация данных 253

ГЛАВА 12. Навигация по иерархическим данным 274

ГЛАВА 13. Использование полнотекстового поиска 287

ГЛАВА 14. Создание представлений 307

ГЛАВА 15. Работа с распределенными запросами 322

ГЛАВА 16. Модификация данных 343

ЧАСТЬ III. СРЕДА РАЗРАБОТКИ SQL SERVER 377

ГЛАВА 17. Реализация физической схемы базы данных 378

ГЛАВА 18. Программирование на языке Transact-SQL 417

ГЛАВА 19. Выполнение массовых операций 444

ГЛАВА 20. Курсор 449

ГЛАВА 21. Создание хранимых процедур 465

ГЛАВА 22. Создание пользовательских функций 484

ГЛАВА 23. Реализация триггеров 495

ГЛАВА 24. Расширенные технологии T-SQL 509

ГЛАВА 25. Расширяемость с помощью уровня абстракции данных 544

ГЛАВА 26. Программирование для SQL Server Everywhere 553
ГЛАВА 27. Программирование сборок CLR в SQL Server 589
ГЛАВА 28. Создание запросов в брокере служб 614
ГЛАВА 29. Поддержка пользовательских типов данных 619
ГЛАВА 30. Программирование в ADO.NET 2.0 638
ГЛАВА 31. Использование XML, XPath и XQuery 672
ГЛАВА 32. Создание хранилищ данных SOA с помощью Web-служб 683
ГЛАВА 33. InfoPath и SQL Server 2005 689

ЧАСТЬ IV. УПРАВЛЕНИЕ ДАННЫМИ НА УРОВНЕ ПРЕДПРИЯТИЯ 697
ГЛАВА 34. Конфигурирование SQL Server 698
ГЛАВА 35. Перенос баз данных 733
ГЛАВА 36. Планирование восстановления 742
ГЛАВА 37. Обслуживание базы данных 767
ГЛАВА 38. Автоматизация обслуживания баз данных с помощью SQL Server Agent 787
ГЛАВА 39. Репликация данных 804
ГЛАВА 40. Защита баз данных 854
ГЛАВА 41. Администрирование SQL Server Express 886

ЧАСТЬ V. БИЗНЕС-ЛОГИКА 895
ГЛАВА 42. ETL в службе интеграции 896
ГЛАВА 43. Бизнес-логика в службе анализа 936
ГЛАВА 44. Раскрытие данных в службе анализа 979
ГЛАВА 45. Программирование запросов MDX 997
ГЛАВА 46. Создание отчетов в службе отчетности 1013
ГЛАВА 47. Администрирование отчетов в службе отчетности 1043
ГЛАВА 48. Анализ данных в Excel и Data Analyzer 1059

ЧАСТЬ VI. СТРАТЕГИИ ОПТИМИЗАЦИИ 1073
ГЛАВА 49. Измерение производительности 1074
ГЛАВА 50. Анализ запросов и настройка индексов 1091
ГЛАВА 51. Управление транзакциями и блокировкой 1114
ГЛАВА 52. Обеспечение высокой доступности 1148
ГЛАВА 53. Масштабирование особо крупных баз данных 1161
ГЛАВА 54. Разработка высокопроизводительных поставщиков доступа к данным 1181

ЧАСТЬ VII. ПРИЛОЖЕНИЯ 1203
ПРИЛОЖЕНИЕ А. Спецификации SQL Server 2005 1204
ПРИЛОЖЕНИЕ Б. Учебные базы данных 1210
Предметный указатель 1217

Содержание

Об авторе 32
Соавторы 33

Введение 36

Для кого предназначена книга 36
Структура книги 37
Часть I. Основы 37
Часть II. Манипулирование данными с помощью инструкции SELECT 37
Часть III. Среда разработки SQL Server 37
Часть IV. Управление данными на уровне предприятия 38
Часть V. Бизнес-логика 38
Часть VI. Стратегии оптимизации 38
Часть VII. Приложения 38
Как использовать данную книгу 38
Принятые соглашения 38
Пиктограммы 39
Что можно найти на Web-сайте книги 40
Ждем ваших отзывов! 40

ЧАСТЬ I. ОСНОВЫ 41

ГЛАВА 1. Принципы информационной архитектуры 42

Простота или сложность 43
Сложность 43
Простота 44
Правило полезности 45
Удобство модели 45
Конфигурации хранилищ данных 46
Стили проектирования главных хранилищ данных 47
Целостность данных 51
Целостность сущностей 51
Целостность домена 51
Ссылочная целостность 51
Определенная пользователем целостность 52
Целостность транзакций 52
Ошибки транзакций 53
Уровни изоляции 54
Пустые значения 55

Производительность 56
Модель 56
Пакетная обработка 56
Индексация 56
Разделы 57
Кэширование 57
Доступность 57
Избыточность 58
Восстановление 58
Масштабируемость 59
Уровень абстракции 59
Обобщение 59
Безопасность 60
Ограничение доступа 60
Информация о владельцах 60
Журнал аудита 61
Теория оптимизации и SQL Server 61
Модель схемы 61
Запросы 62
Индексация 63
Конкуренция 63
Расширенная масштабируемость 63
Резюме 64
ГЛАВА 2. Моделирование реляционных баз данных 65
Моделирование реальности 66
Видимые сущности 67
Каждая строка — это остров 67
Первичные ключи 68
Таблицы, строки и столбцы 68
Идентификация множества сущностей 69
Множество объектов 69
Отношения между объектами 69
Организация и группировка объектов 70
Целостность значений 70
Сложные объекты 70
Реляционные шаблоны 70
Вторичные сущности и внешние ключи 71
Строгость отношения 71
Обязательность отношений 72
Диаграмма модели данных 73
Отношения “один ко многим” 73
Отношение “один к одному” 74
Отношения между подтипом и супертипом 74
Отношение “многие ко многим” 75
Сущности категорий 76
Возвратные отношения 77
Нормализация 78
Принципы проектирования сущностей и атрибутов 79
Нормальные формы 79

Простота и нормализация 80
Первая нормальная форма (1НФ) 80
Вторая нормальная форма (2НФ) 81
Третья нормальная форма (3НФ) 82
Нормальная форма Бойса–Кодда (BNCF) 83
Четвертая нормальная форма (4НФ) 84
Пятая нормальная форма (5НФ) 84
Реляционная алгебра 85
Резюме 86

ГЛАВА 3. Архитектура SQL Server 2005 87

Архитектуры доступа к данным 88
Модель баз данных "клиент/сервер" 88
Многоуровневая архитектура 91
Архитектура, ориентированная на службы 92
Службы SQL Server 92
Реляционное ядро 92
Transact-SQL 94
Visual Studio и CLR 95
Брокер служб 96
Служба репликаций 96
Полнотекстовый поиск 96
Служба уведомлений 97
Объекты управления сервером (SMO) 97
SQL Server Agent 97
Координатор распределенных транзакций 97
SQL Mail 98
Службы бизнес-аналитики 98
Служба интеграции 98
Служба отчетности 99
Служба анализа 100
Различные редакции SQL Server 2005 100
Enterprise (Developer) Edition 101
Standard Edition 101
Workgroup Edition 102
Express Edition 102
Everywhere Edition 103
Утилиты и компоненты SQL Server 103
SQL Server Management Studio 103
SQL Server Configuration Manager 104
Surface Area Configuration 104
Business Intelligence Development Studio 104
Интегрированная справка SQL 105
SQL Profiler 106
Performance Monitor 106
Database Tuning Advisor 106
Утилиты командной строки: SQLCmd и BulkCopy 106
Пакет дополнительных функций SQL Server 2005 106
AdventureWorks 107
Обзор метаданных 107

Системные базы данных 108
Представления метаданных 108
Резюме 109

ГЛАВА 4. Установка SQL Server 2005 110

Планирование установки 110
Операционная система 110
Вопросы безопасности 112
Режим аутентификации 113
Экземпляры SQL Server 114
Рекомендации относительно аппаратной части 115
Выделенный сервер 115
Потребности в памяти 115
Использование множества процессоров 116
Дисковая подсистема 116
Дисковые RAID-массивы 117
Производительность сети 117
Установка пакета 118
Сопровождаемая установка 118
Несопровождаемая установка 119
Удаленная установка 120
Установка на кластер 120
Установка множества экземпляров 120
Тестирование установки 121
Обновление предыдущих версий 121
Утилита SQL Server 2005 Upgrade Advisor 121
Обновление версии SQL Server 2000 122
Миграция в SQL Server 2005 124
Утилита Migration Assistant 124
Преобразование схемы 124
Миграция данных 125
Преобразование бизнес-логики 125
Проверка корректности и интеграция 126
Конфигурирование рабочего пространства SQL Server 127
Утилита Surface Area Configuration 127
Утилиты командной строки 130
Удаление SQL Server 130
Резюме 131

ГЛАВА 5. Подключение клиентского программного обеспечения 132

Разрешение подключений к серверу 132
Утилита Server Configuration Manager 133
Подключения SQL Native Client (SNAC) 134
Функции SQL Server Native Client 134
Системные требования 134
Зеркальное отображение баз данных 135
Асинхронные операции 135
Множества активизированных результирующих наборов данных (MARS) 136
Типы данных XML 136
Типы, определяемые пользователем 136
Особо крупные типы данных 136

Замена устаревших паролей 137
Уровень изоляции SNAPSHOT 137
Резюме 138

ГЛАВА 6. Использование утилиты Management Studio 139
Организация интерфейса 140
Размещение окон 141
Контекстное меню 143
Страница Summary 143
Окно Registered Servers 143
Окно Object Explorer 144
Навигация по дереву 144
Фильтрация объектов в окне Object Explorer 146
Конструктор таблиц 147
Создание диаграмм баз данных 148
Конструктор запросов 149
Нововведения, связанные с выходом пакета обновлений SP2 150
Отчеты Management Studio 151
Использование редактора запросов 152
Подключение к серверу 152
Открытие файла .sql 152
Выполнение пакетов SQL 152
"Горячие" клавиши и закладки 153
Просмотр плана выполнения запроса 154
Окно Solution Explorer 155
Введение в шаблоны 155
Резюме 156

ЧАСТЬ II. МАНИПУЛИРОВАНИЕ ДАННЫМИ С ПОМОЩЬЮ ИНСТРУКЦИИ SELECT 157

ГЛАВА 7. Основы выполнения запросов 158
Основы создания запроса 159
Синтаксическая организация инструкции запроса 159
Графическое представление инструкции запроса 159
Логическая структура запроса 160
Физическая структура запроса 161
Предложение FROM для выбора источников данных 161
Возможные источники данных 162
Именованные диапазоны 163
Имя таблицы 163
Четырехкомпонентные имена таблиц 163
Условия WHERE 164
Использование условия BETWEEN 165
Использование условия IN 167
Использование условия LIKE 168
Множественные условия WHERE 170
SELECT...WHERE 171
Упорядочение результирующего набора данных 172
Определение порядка сортировки с помощью имен столбцов 173
Определение порядка сортировки с помощью выражений 173

Определение порядка сортировки с помощью псевдонимов столбцов 174
Определение порядка сортировки с помощью порядковых номеров столбцов 175
Упорядочение и порядок сопоставления 175
SELECT DISTINCT 176
Ранжирование 178
TOP 178
Параметр WITH TIES 180
Резюме 181

ГЛАВА 8. Использование выражений и скалярных функций 182
Создание выражений 183
Операторы 184
Бинарные операторы 184
Оператор CASE 186
Простая форма оператора CASE 187
Булева форма оператора CASE 187
Работа с пустыми значениями 188
Проверка на пустые значения 189
Обработка пустых значений 190
Скалярные функции 193
Информационные функции 194
Функции работы с датой и временем 195
Строковые функции 197
Функции системы Soundex 200
Функции преобразования данных 203
Информация о среде сервера 206
Резюме 207

ГЛАВА 9. Объединение данных 208
Использование объединений 209
Внутренние объединения 211
Создание внутреннего объединения в коде SQL 212
Количество возвращаемых строк 212
Объединения ANSI SQL-89 213
Объединение множества таблиц 214
Внешние объединения 216
Внешние объединения и необязательные внешние ключи 218
Полные внешние объединения 219
Помещение во внешние объединения условий 221
Собственные объединения 222
Перекрестные (неограниченные) объединения 224
Экзотические объединения 225
Тета-объединения (Θ-объединения) 225
Объединения с множеством условий 226
Неключевые объединения 226
Разность множеств 227
Использование слияний 230
Слияние пересечения 231
Слияние разности/Except 231
Резюме 232

ГЛАВА 10. Включение данных с помощью подзапросов и CTE 233
Методы и расположение 233
Простые подзапросы 234
Общие табличные выражения 236
Использование скалярных подзапросов 237
Использование подзапросов в качестве списков 238
Использование подзапросов в качестве таблиц 242
Коррелированные подзапросы 244
Реляционное деление 248
Реляционное деление с остатком 249
Точное реляционное деление 251
Резюме 252
ГЛАВА 11. Консолидация данных 253
Простая консолидация 253
Основные итоговые функции 254
Основы статистики 256
Группировка в результирующем наборе данных 257
Простая группировка 259
Решение проблем в запросах консолидации данных 259
Генерирование итогов 264
Запросы сведения 264
Кубические запросы 265
Вычисления итогов 265
Создание перекрестных запросов 267
Перекрестные запросы с фиксированным столбцом 268
Динамические перекрестные запросы 271
Резюме 273
ГЛАВА 12. Навигация по иерархическим данным 274
Шаблон смежных списков 275
Основные шаблоны смежных списков 275
Вариации смежных списков 276
Навигация по смежному списку 279
Использование стандартной инструкции `SELECT` 279
Использование рекурсивного курсора 280
Использование пакетных решений 282
Использование пользовательских функций 283
Использование рекурсивных общих табличных выражений 285
Резюме 286
ГЛАВА 13. Использование полнотекстового поиска 287
Конфигурирование каталогов полнотекстового поиска 288
Создание каталога с помощью мастера 289
Создание каталога на языке T-SQL 290
Помещение данных в полнотекстовый индекс 291
Обслуживание каталога в Management Studio 292
Обслуживание каталога в программном коде T-SQL 293
Файлы шумов 294
Поиск слов 295
Функция `contains` 295

Функция ContainsTable 295
Расширенные параметры поиска 297
Поиск нескольких слов 297
Поиск с использованием символов макроподстановки 298
Поиск фраз 299
Поиск близких слов 299
Поиск словоформ 300
Поиск синонимов 300
Поиск с использованием веса слов 301
Нечеткий поиск 303
Параметр FREETEXT 303
Параметр FREETEXTTABLE 303
Индексация двоичных объектов 304
Вопросы производительности 305
Резюме 306

ГЛАВА 14. Создание представлений 307

Зачем использовать представления 307
Работа с представлениями 309
Создание представлений в Management Studio 309
Создание представлений с помощью кода DDL 311
Предложение ORDER BY и представления 312
Ограничения в представлениях 313
Выполнение представлений 313
Защита представлений 313
Защита данных 313
Защита представлений 315
Обновление информации с помощью представлений 317
Вложенные представления 318
Использование синонимов 321
Резюме 321

ГЛАВА 15. Работа с распределенными запросами 322

Основные концепции распределенных запросов 322
Доступ к базе данных локального сервера 324
Подключение к внешним источникам данных 325
Подключение в Management Studio 325
Подключение с помощью T-SQL 327
Подключение к источникам данных, отличным от SQL Server 330
Проектирование распределенных запросов 333
Распределенные запросы и Management Studio 333
Распределенные представления 334
Локальные распределенные запросы 334
Сквозные распределенные запросы 337
Распределенные транзакции 339
Координатор распределенных транзакций 340
Создание распределенных транзакций 340
Мониторинг распределенных транзакций 341
Резюме 342

ГЛАВА 16. Модификация данных 343
Вставка данных 345
Вставка одной строки значений 345
Вставка результирующего набора данных инструкции SELECT 347
Вставка результирующего набора данных из хранимой процедуры 348
Создание строки со значениями по умолчанию 350
Создание таблицы в процессе вставки данных 350
Обновление данных 353
Обновление одной таблицы 353
Выполнение глобального поиска и замены 354
Ссылка на множество таблиц при обновлении данных 355
Удаление данных 358
Ссылка при удалении на множество таблиц 358
Каскадные удаления 359
Альтернативы физическому удалению данных 361
Возвращение модифицированных данных 362
Возвращение данных из операции вставки 362
Возвращение данных из операции обновления 362
Возвращение данных из инструкции удаления 363
Возвращение данных в переменной @Table 363
Потенциальные препятствия на пути модификации данных 364
Проблема типа и длины данных 364
Проблемы первичного ключа 365
Проблемы внешних ключей 368
Проблемы уникальных индексов 370
Проблемы пустых значений и значений по умолчанию 370
Проблемы ограничений проверки 371
Проблемы триггеров INSTEAD OF 371
Проблемы триггеров AFTER 372
Вычисляемые столбцы 373
Проблемы необновляемых представлений 374
Проблемы представлений с параметром проверки 374
Проблемы системы безопасности 375
Резюме 376

ЧАСТЬ III. СРЕДА РАЗРАБОТКИ SQL SERVER 377

ГЛАВА 17. Реализация физической схемы базы данных 378
Проектирование физической схемы базы данных 379
Варианты проектирования физической схемы 379
Корректировка модели данных 380
Вопросы производительности 380
Вопросы масштабируемости 380
Ответственный подход к денормализации 381
Создание базы данных 382
Команда DDL CREATE 382
Концепции файлов базы данных 384
Автоматизация роста размера файла 385
Использование множества файлов 386
Планирование нескольких файловых групп 388

Создание таблиц 390
Создание таблиц в Management Studio 390
Работа со сценариями SQL 392
Схемы 392
Имена таблиц и столбцов 393
Файловые группы 394
Создание первичных ключей 395
Первичные ключи 395
Создание внешних ключей 399
Создание пользовательских столбцов данных 404
Типы данных столбцов 404
Вычисляемые столбцы 407
Ограничения и значения столбцов по умолчанию 408
Каталог данных 412
Триггеры DDL 413
Создание и изменение триггеров DDL 414
Функция `EventData()` 415
Включение и отключение триггеров DDL 416
Резюме 416

ГЛАВА 18. Программирование на языке Transact-SQL 417

Основы Transact-SQL 418
Пакеты T-SQL 418
Форматирование в T-SQL 419
Отладка T-SQL 420
Переменные 421
Значения по умолчанию и область определения переменных 421
Использование команд `SET` и `SELECT` 422
Условный отбор 423
Использование переменных в запросах SQL 423
Переменные с множественным присвоением 424
Управление выполнением процедур 424
Оператор `if` 425
`While` 426
`Goto` 427
Изучение SQL Server программным путем 427
`sp_help` 427
Глобальные переменные 428
Временные таблицы и табличные переменные 429
Локальные временные таблицы 430
Глобальные временные таблицы 430
Табличные переменные 431
Динамический SQL 431
Выполнение инструкций динамического SQL 432
`sp_executeSQL` 432
Создание динамического кода SQL 433
Обработка ошибок 434
`Try...Catch` 435
Старая глобальная переменная `@@Error` 436
Глобальная переменная `@@RowCount` 437

Raiserror 438
Блок Catch 442
Фатальные ошибки T-SQL 442
Резюме 443

ГЛАВА 19. Выполнение массовых операций 444

Команда bulk insert 445
Параметры команды bulk insert 446
Утилита BCP 447
Резюме 448

ГЛАВА 20. Курсор 449

Все о курсорах 450
Пять этапов жизни курсора 450
Управление курсором 451
Обновление курсора 452
Область определения курсора 452
Курсоры и транзакции 453
Стратегии курсора 453
Сложные логические решения 454
Программирование логики 455
Курсор SQL-92 с хранимой процедурой 456
Курсор прямого доступа с хранимой процедурой 457
Курсор прямого действия и пользовательская функция 458
Курсор обновления с хранимой процедурой 458
Запрос обновления с пользовательской функцией 459
Использование множества запросов 459
Запросы с выражением CASE 461
Анализ производительности 461
Пример денормализации списка 462
Резюме 464

ГЛАВА 21. Создание хранимых процедур 465

Управление хранимыми процедурами 466
Инструкции CREATE, ALTER и DROP 466
Возвращение набора записей 467
Компиляция хранимых процедур 467
Шифрование хранимых процедур 468
Системные хранимые процедуры 468
Передача данных в хранимые процедуры 469
Входные параметры 469
Значения параметров, заданные по умолчанию 470
Получение данных из хранимой процедуры 471
Выходные параметры 471
Использование команды RETURN 472
Маршруты и область определения возвращаемых данных 473
Использование хранимых процедур в запросах 474
Выполнение удаленных хранимых процедур 475
Завершенная хранимая процедура 476
Хранимая процедура pGetPrice 476
Хранимая процедура pOrder_AddNew 478

Хранимая процедура pOrder_AddItem 480
Добавление заказа 482
Резюме 483
ГЛАВА 22. Создание пользовательских функций 484
Скалярные функции 485
Создание скалярных функций 486
Вызов скалярных функций 488
Создание функций со связанной схемой 488
Внедренные табличные функции 488
Создание внедренной табличной функции 489
Вызов внедренной табличной функции 489
Использование параметров 490
Коррелированные пользовательские функции 491
Табличные функции с множеством инструкций 492
Создание табличных функций с множеством инструкций 492
Вызов функции 493
Резюме 494
ГЛАВА 23. Реализация триггеров 495
Основы триггеров 495
Порядок выполнения транзакций 496
Создание триггеров 497
Триггеры AFTER 498
Триггеры INSTEAD OF 499
Ограничения триггеров 500
Отключение триггеров 500
Создание списка триггеров 500
Триггеры и безопасность 501
Работа с транзакциями 501
Определение состава обновленных столбцов 501
Логические таблицы Inserted и Deleted 503
Создание триггеров, работающих со множеством строк 504
Взаимодействие триггеров 505
Организация триггеров 505
Вложенные триггеры 506
Рекурсивные триггеры 506
Триггеры INSTEAD OF и AFTER 508
Множество триггеров AFTER 508
Резюме 508
ГЛАВА 24. Расширенные технологии T-SQL 509
Проверка сложных правил бизнес-логики 510
Поддержка сложной ссылочной целостности 511
Обеспечение защиты данных на уровне строк 513
Таблица безопасности 514
Хранимая процедура проверки полномочий 520
Функция проверки полномочий 521
Использование учетной записи NT 522
Триггер проверки полномочий 523
Аудит изменений данных 524

Журнал аудита 524
Фиксированный триггер журнала аудита 525
Выполнение отката операций с помощью журнала аудита 528
Сложности аудита 529
Динамические триггеры и процедуры журнала аудита 530
Обработка транзакций консолидации 535
Триггер таблицы складских операций 536
Триггер складской таблицы 537
Логическое удаление данных 539
Триггеры логического удаления 539
Восстановление логически удаленных строк 541
Фильтрация логически удаленных строк 541
Каскадное логическое удаление 541
Маркировка неактивности 542
Архивирование данных 542
Резюме 543

ГЛАВА 25. Расширяемость с помощью уровня абстракции данных 544

Хранимая процедура `AddNew` 545
Хранимая процедура `Fetch` 547
Хранимая процедура `Update` 548
Обновление с условием `RowVersion` 548
Минимальное обновление 549
Хранимая процедура `Delete` 551
Резюме 552

ГЛАВА 26. Программирование для SQL Server Everywhere 553

Обзор SQL Server 2005 Everywhere Edition 554
История 554
Концепции 555
Что нового в SQL Server 2005 Everywhere 558
Основы SQL Everywhere 560
Установка SQL Everywhere 560
Query Analyzer 3.0 563
Создание базы данных SQL Everywhere 564
Модернизация существующей базы данных SQL CE 2.0 575
Синхронизация данных 576
Удаленный доступ к данным 577
Репликация слияния 578
Web-службы 579
Упаковка и развертывание 581
Программное создание базы данных 581
Развертывание исходной базы данных вместе с мобильным приложением 581
Динамическое создание базы данных с помощью репликации слияния 581
Прочие подходы 582
Вопросы безопасности 582
Защита паролем 582
Шифрование 582
Безопасная синхронизация данных 583
Настройка, обслуживание и администрирование 583
Измерение производительности выполнения запросов и ее повышение 583

Обслуживание SQL Everywhere 584
Восстановление поврежденной базы данных SQL Everywhere 586
Поддержание производительности репликации слияния 587
Дополнительная информация 588
Резюме 588

ГЛАВА 27. Программирование сборок CLR в SQL Server 589
Беглое знакомство со средой .NET Framework 590
Сборки 591
Домены приложений 592
Обзор типов CLR в SQL Server 595
Атрибуты типов .NET интеграции CLR 595
Общие характеристики типов CLR в SQL Server 597
Типы данных 597
Методы .NET, поддерживающие интеграцию CLR 600
Инструкции CLR DDL языка T-SQL и представления каталогов 602
Сборки 602
Объекты базы данных 603
Создание типов данных в Visual Studio 2005 603
Создание проекта CLR 604
Программирование хранимой процедуры CLR 607
Функции CLR 609
Что использовать: CLR или T-SQL 611
T-SQL еще рано сбрасывать со счетов 612
Резюме 613

ГЛАВА 28. Создание запросов в брокере служб 614
Конфигурирование очереди сообщений 615
Работа с диалогами 615
Отправка сообщения в очередь 616
Получение сообщений 616
Мониторинг брокера служб 618
Резюме 618

ГЛАВА 29. Поддержка пользовательских типов данных 619
Создание пользовательских типов интеграции CLR 621
Удовлетворение требований 622
Программирование пользовательских типов CLR в Visual Studio 624
Тестирование и отладка пользовательского типа 631
Вопросы производительности 631
Развертывание пользовательских типов интеграции CLR 633
Строго именованные сборки и глобальный кэш сборки 633
Создание строго именованных сборок .NET 634
Обслуживание пользовательских типов 636
Резюме 636

ГЛАВА 30. Программирование в ADO.NET 2.0 638
Обзор ADO.NET 639
ADO 640
Объектная модель ADO 644
ADO.NET 652
ADO.NET в Visual Studio 2005 664

Server Explorer 664
Отладка ADO.NET 665
Трассировка приложения 665
Основы создания приложений 666
Подключение к SQL Server 667
Адаптеры данных 667
Объект чтения данных и наборы записей 668
Потоки 669
Асинхронное выполнение 669
Использование одного значения из базы данных 669
Модификация данных 670
Связывание с элементами управления 671
Резюме 671

ГЛАВА 31. Использование XML, XPath и XQuery 672

Тип данных XML 673
Распределение и преобразование 673
Ограничения типа XML 674
Коллекции схем XML 674
Индексы XML 675
Выполнение запросов к данным XML 675
Xpath 676
Запросы FLWOR 676
Слияние XQuery с инструкцией `SELECT` 677
Декомпозиция данных XML в SQL Server 678
Чтение данных XML в SQL Server 678
Создание документов XML в SQL Server 2005 680
Резюме 682

ГЛАВА 32. Создание хранилищ данных SOA с помощью Web-служб 683

Прослушивание HTTP 684
Процесс `HTTP.sys` 685
Неявные концевые точки 685
Явные концевые точки 686
WSDL 686
Защита концевых точек 687
Резюме 687

ГЛАВА 33. InfoPath и SQL Server 2005 689

Обзор InfoPath 2003 689
Автоматическая проверка данных 690
Заполнение форм в автономном режиме 690
Условное форматирование 691
Вопросы защиты в InfoPath 2003 691
Объектная модель InfoPath 692
Сценарии и программный код .NET 692
Примечание 692
Создание шаблона формы 692
Прочие функции InfoPath 695
Резюме 696

ЧАСТЬ IV. УПРАВЛЕНИЕ ДАННЫМИ НА УРОВНЕ ПРЕДПРИЯТИЯ 697

ГЛАВА 34. Конфигурирование SQL Server 698

Установка параметров 698
Конфигурирование сервера 699
Конфигурирование базы данных 701
Конфигурирование подключения 702
Параметры конфигурации 703
Отображение расширенных свойств 703
Параметры конфигурации запуска и останова сервера 705
Параметры конфигурации памяти 706
Параметры конфигурации процессора 712
Параметры конфигурирования системы безопасности 715
Параметры конфигурации подключения 717
Параметры конфигурации сервера 720
Параметры конфигурации индекса 722
Конфигурирование автоматических настроек баз данных 723
Параметры конфигурации курсора 724
Параметры конфигурации SQL ANSI 725
Параметры конфигурации триггеров 728
Параметры конфигурации состояния базы данных 729
Параметры конфигурации восстановления 730
Резюме 732

ГЛАВА 35. Перенос баз данных 733

Мастер копирования баз данных 733
Использование сценария SQL 737
Отключение и подключение 739
Резюме 741

ГЛАВА 36. Планирование восстановления 742

Концепции восстановления 743
Модели восстановления 744
Простая модель восстановления 745
Полная модель восстановления 746
Модель с неполным протоколированием 747
Установка модели восстановления 748
Изменение модели восстановления 748
Резервирование базы данных 749
Устройства резервирования 749
Хранение и ротация резервных копий 750
Выполнение резервного копирования в Management Studio 750
Резервирование базы данных в программном коде 751
Программная проверка резервной копии 754
Работа с журналом транзакций 754
Внутренний мир журнала транзакций 754
Резервирование журнала транзакций 755
Сжатие журнала транзакций 756
Журнал транзакций и простая модель восстановления 757
Операции восстановления 757
Идентификация проблемы 757

Последовательности восстановления 758
Воссоздание базы данных в Management Studio 759
Воссоздание базы данных программным путем 761
Восстановление системных баз данных 764
Резервное копирование базы данных `master` 764
Восстановление базы данных `master` 764
Системная база данных `msdb` 765
Полное воссоздание сервера баз данных 765
Резюме 766

ГЛАВА 37. Обслуживание базы данных 767

Команды утилиты DBCC 767
Проверка целостности базы данных 768
Обслуживание индексов 772
Размер файла базы данных 776
Дополнительные команды утилиты DBCC 779
Управление обслуживанием базы данных 780
Планирование обслуживания базы данных 780
Мастер плана обслуживания 781
Обслуживание базы данных из командной строки 785
Мониторинг обслуживания базы данных 786
Резюме 786

ГЛАВА 38. Автоматизация обслуживания баз данных с помощью SQL Server Agent 787

Настройка SQL Server Agent 787
Концепции предупреждений, операторов и заданий 791
Управление операторами 791
Управление предупреждениями 792
Создание сообщений об ошибках, определенных пользователем 793
Создание предупреждения 794
Управление заданиями 796
Создание категории заданий 798
Создание определения задания 799
Настройка действий, выполняемых в задании 800
Конфигурирование графика выполнения задания 802
Обработка сообщений об успехе и неудаче операции 802
Резюме 803

ГЛАВА 39. Репликация данных 804

Зачем реплицировать данные 804
Поддержка восстановления данных при аппаратных и программных сбоях 804
Требования приложений 806
Выигрыш в производительности 807
Распространение данных 807
Сравнительная характеристика различных методов распределения данных 807
Модель репликации от Microsoft 810
Издатель 810
Подписчик 810
Распространитель 811
Модель с централизованным издателем 811

Модель с централизованным подписчиком 811
Модель с переизданиями 811
Одноранговая модель 811
Статья 812
Принудительная подписка 813
Подписка по запросу 813
Типы репликаций 813
Репликация снимков баз данных 814
Репликация снимков баз данных с непосредственным обновлением 815
Репликация снимков базы данных с очередью обновлений 815
Репликация снимков базы данных с непосредственным обновлением и очередью восстановления 816
Репликация транзакций 816
Агент снимков 816
Агент чтения журнала 816
Агент распространения 817
Одноранговая репликация 817
Двусторонняя репликация транзакций 818
Репликация транзакций с непосредственным обновлением 818
Репликация транзакций с очередью обновлений 818
Репликация транзакций с непосредственным обновлением и очередью восстановления 819
Репликация транзакций через Интернет 819
Репликация слияния 819
Репликация слияния и подписчики SQL CE и SQL Mobile 822
Репликация слияния через Интернет 822
Нововведения в репликациях SQL Server 2005 822
Продолжение репликаций снимков базы данных 822
Публикации Oracle 822
Повышенная безопасность 822
Одноранговая модель репликации 823
Репликация инструкций DDL 823
Репликация полнотекстовых индексов 823
Разрешение анонимных подписок на все публикации 824
Логические записи в репликации слияния 824
Предварительно вычисленные разделы 824
Обновление уникальных ключей 824
Пользовательская обработка конфликтов с помощью RMO 824
Многочисленные улучшения производительности 825
Маркеры трассировки 825
Распараллеливание транзакций 825
Загрузка только статей 825
Монитор репликаций 826
Репликации слияния по протоколу HTTPS 827
Повышение производительности и масштабируемости репликаций слияния 827
SQL RMO 827
Упрощенные мастера 827
Инициализация подписчика 828
Конфигурирование репликаций 828
Использование локального распространителя 829

Использование удаленного распространителя 829
Создание публикаций репликации снимков базы данных 830
Создание публикаций репликации транзакций 832
Создание публикаций двусторонней репликации транзакций 836
Создание публикаций Oracle 837
Создание публикаций одноранговой репликации 838
Создание публикаций репликации слияния 840
Создание подписок 841
Создание подписок Web-синхронизации 844
Мониторинг решений репликации 848
Профили агентов 848
Все подписки 849
Предупреждения и агенты 849
Изменение параметров агента репликации 850
Маркеры трассировки 851
Производительность репликации 851
Разрешение проблем репликации 852
Пакеты обновлений 852
Резюме 853

ГЛАВА 40. Защита баз данных 854

Концепции защиты 855
Система безопасности уровня сервера 855
Система безопасности уровня базы данных 856
Права собственности на объект 857
Система безопасности Windows 857
Система безопасности Windows 857
Регистрационная запись SQL Server 858
Безопасность сервера 858
Режимы аутентификации в SQL Server 858
Аутентификация Windows 859
Регистрационные записи SQL Server 863
Серверные роли 865
Безопасность базы данных 866
Гостевые учетные записи 866
Предоставление доступа к базе данных 867
Фиксированные роли базы данных 868
Разрешения защищаемых объектов 870
Роли приложений 870
Безопасность объектов 871
Разрешения уровня объекта 871
Стандартные роли базы данных 873
Безопасность объектов и Management Studio 875
Пример простой модели защиты 876
Уровень C2 системы безопасности 877
Представления и безопасность 878
Криптография 879
Введение в криптографию 879
Криптографическая иерархия SQL Server 880
Шифрование парафразой 880
Шифрование с помощью симметричного ключа 882

Как избежать "инъекций" SQL 884
Прикрепление вредоносного кода 884
Прикрепление `OR 1=1` 884
Пароль? Какой пароль? 885
Защита от "инъекций" кода SQL 885
Резюме 885

ГЛАВА 41. Администрирование SQL Server Express 886
Установка SQL Server Express 886
Использование параметров командной строки для фоновой установки SSE 888
Использование файлов INI при установке SSE 891
Версия Management Studio для SSE 893
Резюме 893

ЧАСТЬ V. БИЗНЕС-ЛОГИКА 895

ГЛАВА 42. ETL в службе интеграции 896
Среда проектирования 897
Вкладка `Connection Managers` 898
Переменные 899
Элементы конфигурирования 901
Обработчики ошибок 905
Выполнение пакета в среде разработки 905
Элементы пакетов службы интеграции 906
Диспетчеры подключений 906
Элементы потока управления 910
Компоненты потока данных 917
Обслуживаемые и управляемые пакеты 928
Протоколирование 929
Конфигурация пакета 930
Перезапуск из контрольной точки 931
Развертывание пакетов 932
Установка пакетов 932
Выполнение пакетов 932
Изменения в службе интеграции, связанные с выходом пакетов обновлений 933
Резюме 935

ГЛАВА 43. Бизнес-логика в службе анализа 936
Хранилища данных 937
Схема "звезда" 937
Единообразие 938
Загрузка данных 939
Знакомство со службой анализа 941
Архитектура службы анализа 941
Унифицированная модель измерений 942
Сервер 942
Клиент 943
Создание базы данных 944
Утилита Business Intelligence Development Studio 944
Источники данных 944
Представление источника данных 945

Создание куба 949
Измерения 950
Конструктор измерений 950
Изменение данных в измерениях 956
За пределами обычных измерений 958
Тонкая настройка измерений 961
Кубы 962
Представление Cube Structure 963
Представление Dimension Usage 965
Представление KPI 967
Представление Actions 968
Представление Partitions 968
Проекции 971
Хранилища данных 971
Уведомления SQL Server 973
Уведомления, инициируемые клиентом 973
Уведомления опроса 973
Целостность данных 974
Обработка пустых значений 974
Параметр UnknownMember 975
Конфигурирование ошибок 976
Пакеты обновлений 977
Резюме 977

ГЛАВА 44. Раскрытие данных в службе анализа 979

Процесс раскрытия данных 980
Моделирование в службе анализа 981
Алгоритмы 988
Алгоритм дерева решений 989
Линейная регрессия 990
Кластеризация 990
Последовательная кластеризация 991
Нейронные сети 992
Логистическая регрессия 992
Наивный Байесовский алгоритм 993
Ассоциативные правила 993
Временны*е* ряды 994
Интеграция OLAP 995
Резюме 996

ГЛАВА 45. Программирование запросов MDX 997

Основы запросов `SELECT` 998
Адресация в кубе 998
Структура измерения 999
Базовая инструкция `SELECT` 1000
Расширенные запросы `SELECT` 1005
Подкубы 1005
Предложение `WITH` 1005
Параметры измерений 1009
Сценарии MDX 1010
Вычисляемые члены и именованные множества 1011

Добавление бизнес-аналитики 1011
Резюме 1012

ГЛАВА 46. Создание отчетов в службе отчетности 1013

Что такое отчет 1014
Язык определения отчетов (RDL) 1014
Источники данных 1014
Наборы данных службы отчетности 1016
Параметры запроса и отчета 1016
Содержимое и композиция отчета 1017
Процесс создания отчета 1018
Создание проекта службы отчетности в Visual Studio 2005 1018
Создание отчета 1019
Использование мастера для создания отчета 1019
Создание отчета с нуля 1020
Работа с данными 1022
Работа с SQL в конструкторе отчетов 1022
Использование параметров запроса для отбора и фильтрации данных 1023
Добавление в набор данных вычисляемых полей 1026
Работа с источниками данных XML 1027
Работа с выражениями 1029
Проектирование композиции отчета 1031
Основные элементы композиции отчета 1031
Использование страниц параметров таблицы и матрицы 1035
Группировка и сортировка данных в элементах таблицы и матрицы 1035
Иллюстрирование данных с помощью диаграмм 1040
Пакеты обновлений и служба отчетности 1041
Резюме 1042

ГЛАВА 47. Администрирование отчетов в службе отчетности 1043

Развертывание отчетов службы отчетности 1043
Развертывание отчетов с помощью Visual Studio 2005 1043
Развертывание отчетов с помощью диспетчера отчетов 1046
Развертывание отчетов программным путем с использованием Web-службы Reporting Services 1047
Конфигурирование службы отчетности в диспетчере отчетов 1047
Конфигурирование настроек сайта службы отчетности 1047
Администрирование системы безопасности 1049
Работа со связанными отчетами 1052
Создание связанных отчетов 1053
Вооружаемся подписками 1053
Создание подписки, управляемой данными 1053
Резюме 1058

ГЛАВА 48. Анализ данных в Excel и Data Analyzer 1059

Сводные таблицы Excel 1060
Подключение к многомерным источникам данных 1061
Подключение к реляционным источникам данных 1063
Проектирование сводных таблиц 1065
Проектирование сводных диаграмм 1067
Диапазоны данных Excel 1068

Анализатор данных 1069
Создание представления 1070
Форматирование представления 1070
Резюме 1071

ЧАСТЬ VI. СТРАТЕГИИ ОПТИМИЗАЦИИ 1073

ГЛАВА 49. Измерение производительности 1074
Измерение точности 1074
Использование монитора производительности 1075
Монитор системы 1076
Протоколы счетчиков производительности 1079
Использование SQL Server Profiler 1079
Определение новой трассировки 1081
Отбор событий 1082
Фильтрация событий 1082
Организация столбцов 1083
Использование трассировки 1083
Интеграция данных монитора производительности 1083
Использование SQL Trace 1085
Использование Transact-SQL 1085
Использование динамических представлений управления 1085
Использование функции `GetDate()` 1086
Использование статистики 1086
Ключевой индикатор производительности базы данных 1087
Периодическое тестирование производительности 1088
Сбор данных производительности 1088
Тестирование влияния масштабирования на производительность 1089
Резюме 1090

ГЛАВА 50. Анализ запросов и настройка индексов 1091
Глобальный подход к настройке индексов 1091
Индексация 1092
Основы индексации 1092
Создание индексов 1094
Параметры индексов 1098
Создание базовых индексов 1101
Анализ запросов 1101
Просмотр плана выполнения запроса 1101
Использование параметра `Showplan` 1102
Интерпретация плана выполнения запроса 1103
Настройка индексов 1105
Отсутствие индексов 1105
Поиск закладки 1106
Оптимизируемый аргумент поиска 1108
Избирательность индексов 1108
Повторное использование планов выполнения запросов 1109
Углубленная стратегия индексирования 1111
Использование Database Engine Tuning Advisor 1112
Резюме 1113

ГЛАВА 51. Управление транзакциями и блокировкой 1114
Основы транзакций 1115
Целостность транзакций 1116
Свойства ACID 1116
Сбои транзакций 1117
Уровни изоляции 1121
Архитектура журнала транзакций 1123
Последовательность работы с журналом транзакций 1123
Восстановление журнала транзакций 1127
Концепция блокировок в SQL Server 1127
Гранулярность блокировок 1128
Режимы блокировок 1128
Продолжительность блокировки 1131
Мониторинг блокировок 1131
Использование утилиты Profiler 1133
Управление блокировками в SQL Server 1134
Установка уровня изоляции подключения 1134
Использование изоляции уровня снимков базы данных 1134
Использование параметров блокировки 1136
Ограничения блокировок уровня индексов 1137
Управление временем ожидания блокировок 1138
Оценка производительности конкуренции в базе данных 1138
Блокировки приложения 1139
Взаимоблокировки 1140
Создание взаимоблокировки 1140
Автоматическое выявление взаимоблокировок 1142
Обработка взаимоблокировок 1143
Минимизация взаимоблокировок 1143
Проектирование блокировок в приложениях 1143
Реализация оптимистической блокировки 1144
Потерянные обновления 1144
Стратегии производительности транзакций 1146
Резюме 1147

ГЛАВА 52. Обеспечение высокой доступности 1148
Тестирование доступности 1149
"Горячая" замена 1149
Доставка журнала 1150
Доставка учетных записей пользователей 1154
Возвращение к исходному первичному серверу 1154
Резервные серверы и кластеризация 1154
Установка резервного сервера баз данных 1155
Конфигурирование 1155
Зеркальное отображение баз данных 1157
Предварительные требования 1157
Конфигурирование 1158
Архитектура среды 1158
Резюме 1159

ГЛАВА 53. Масштабирование особо крупных баз данных 1161
Теория оптимизации и масштабируемость 1162
Масштабирование платформы 1162

Масштабирование решений 1165
Разделение таблиц и индексов 1165
Создание функции разделения 1167
Создание схем разделения 1168
Создание разделенной таблицы 1168
Выполнение запросов к разделенным таблицам 1171
Изменение разделенных таблиц 1171
Переключение таблиц 1173
Подвижные разделы 1175
Индексация разделенных таблиц 1176
Удаление разделения 1176
Работа с индексированными представлениями 1176
Индексированные представления и запросы 1178
Обновление индексированных представлений 1179
Резюме 1180

ГЛАВА 54. Разработка высокопроизводительных поставщиков доступа к данным 1181

Концепции доступа к данным 1182
В чем ценность хорошего доступа к данным 1182
Определение требований 1182
Объекты доступа к данным 1183
Как работают сценарии DAO 1184
Достоинства модели DAO 1186
Фабрики 1187
Достоинства фабричной модели 1190
Недостатки фабричной модели 1190
Поставщики данных 1191
Как работает поставщик данных 1193
Достоинства шаблона поставщика данных 1201
Недостатки шаблона поставщика данных 1201
Резюме 1202

ЧАСТЬ VII. ПРИЛОЖЕНИЯ 1203

ПРИЛОЖЕНИЕ А. Спецификации SQL Server 2005 1204

ПРИЛОЖЕНИЕ Б. Учебные базы данных 1210

Файлы учебных баз данных 1212
Cape Hatteras Adventures версии 2 1212
Требования приложения 1212
Модель базы данных 1213
OBX Kites 1214
Требования к приложению 1214
Модель базы данных 1214
База данных Family 1214
Требования к приложению 1215
Модель базы данных 1215
База данных Aesop's Fables 1215
Требования к приложению 1216
Модель базы данных 1216

Предметный указатель 1217

Об авторе

Пол Нильсен — известный разработчик баз данных, обладатель статуса Microsoft SQL Server MVP, автор и преподаватель, специализирующийся на архитектуре баз данных и технологиях Microsoft SQL Server. Пол занимался базами данных с 1982 года, работал архитектором корпоративных баз данных в компании Comparisson International, преподавателем SQL Server в организации Learning Tree, техническим редактором в журнале, посвященном базам данных, и даже служил на подводной лодке в качестве специалиста по системам обработки данных.

Пол входит в совет директоров PASS (Professional Association for SQL Server), курирует ряд форумов, посвященных SQL Server, и организовывает специализированные семинары для повышения квалификации профессионалов. В свободное время он увлекается подводной охотой, играет на гитаре и катается на мотоцикле в горах Колорадо. Он регулярно читает лекции, посвященные разработке приложений для SQL Server и проектированию баз данных, ведет спецкурс по оптимизации и расширенным технологиям разработки приложений.

Более полную информацию об авторе можно найти на сайте `www.SQLServerBible.com`. Если у вас возникнут вопросы к автору или вы захотите поделиться с ним своим мнением, посылайте сообщения по адресу `pauln@SQLServerBible.com`.

Соавторы

Иларий Коттер (Hilary Cotter) — разработчик Microsoft SQL Server и автор глав 13 и 39, посвященных полнотекстовому поиску и репликации данных. Он директор подразделений Text Mining и Database Strategy в компании RelevantNoise, занимающейся индексацией блогов для бизнес-решений. Иларий закончил машиностроительный факультет в университете Торонто, позже изучал экономику в университете Калгари и компьютерные науки в Калифорнийском университете в Беркли. Он и его прекрасная жена Мириам — родители пяти милых детишек в возрасте от 13 месяцев до шести лет.

Иларий работал в компаниях Microsoft, Merril Lynch, UBS-Paine Webber, MetLife, VoiceStream, Tiffany & Co., Pacific Bell, Cahners, Novartis, Petro-Canada, Associated Press и Johnson and Johnson. Вы можете пообщаться с ним на форумах, поддерживаемых компанией Microsoft, а также связаться с ним напрямую по адресу `hilary.cotter@gmail.com`.

Монте Холифилд (Monte Holyfield) — автор глав 46 и 47, посвященных службе отчетности, является основателем и управляющим компании MJH Software Solutions LLC (`www.mjhsoftware.com`). Он имеет более чем 10-летний опыт разработки приложений, занимал множество должностей — от главного консультанта до руководителя проекта, обладает статусом MCP (Microsoft Certified Professional). Если он не занят программированием, то занимается со своей семьей активными видами спорта, в частности ездой на велосипеде, катанием на лыжах, виндсерфингом и практическим всем, что позволяют условия штата Колорадо.

Кевин Ллойд (Kevin LLoyd) — технический редактор, опытный разработчик приложений для SQL Server, консультант и один из ведущих специалистов компании Data Solutions LLC. Он специализируется на архитектуре баз данных и интеграции приложений, моделировании данных, оценке их качества и повышении производительности систем. Помимо приложений SQL Server он разрабатывает Web-службы на языках C# и XML для репликации данных в SQL Mobile. Ему также нравится консультировать других разработчиков и писать приложения для Windows Mobile. С 1996 года он разрабатывал приложения для работы с данными в области финансов, электронной коммерции и для правительственных структур. Кевин работал консультантом в таких компаниях, как Microsoft, Frontier Airlines, Comparisson International и Keane. При желании вы можете связаться с ним по адресу `kevin@kevndeb.com`. В свободное время ему нравится проводить время с семьей и путешествовать.

Джордж Мудрак (George Mudrak) — автор главы 4, посвященной инсталляции сервера, главы 5, посвященной подключению клиентских программ, и главы 54, посвященной разработке высокопроизводительных поставщиков данных. Джордж работает архитектором корпоративных баз данных в компании Comparisson International в городе Колорадо-Спрингс, штат Колорадо. В течение двенадцати лет он занимается программированием на множестве языков, а в последнее время увлекся технологиями Microsoft .NET. Свободное время любит проводить с семьей, занимаясь садоводством.

Джон Мюллер (John Mueller) — технический редактор и один из авторов главы 34, посвященной конфигурированию сервера, главы 35, описывающей перенос баз данных, главы 37,

посвященной обслуживанию баз данных, главы 38, в которой описана автоматизация обслуживания баз данных с помощью SQL Server Agent, и главы 41, в которой рассматривается администрирование SQL Server Express. Страсть к написанию книг у него в крови. На данный момент он написал 69 книг и более 300 статей, посвященных разнообразным вопросам — от локальных сетей и управления базами данных до искусственного интеллекта и высокоуровнего программирования. Некоторые из его последних книг были посвящены оптимизации Windows, проблемам безопасности в среде .NET, Web-службам компаний Amazon, eBay и Google. В настоящее время он работает редактором электронного журнала .NET в издательстве Pinnacle Publishing. Более полную информацию о нем вы можете получить на сайте `www.freenewsletters.com`. Если Джон не сидит за компьютером, то проводит время в своей мастерской, изготавливая свечи или собирая очередной книжный шкаф. Его Web-сайт находится по адресу `www.mwt.net/~jmueller`.

Даррен Шаффер (Darren Shaffer) — автор главы 26, посвященной SQL Server Mobile, обладатель статуса Microsoft Compact Framework MVP. Он является ведущим архитектором решений .NET в компании Connected Innovation LLC (`www.connectedinnovation.com`). За последние четыре года он был ответственным за разработку более 25 решений Compact Framework для множества клиентов из списка Fortune 1000. Даррен часто выступает на конференциях компании Microsoft, таких как MEDC и TechEd, пишет статьи для MSDN, в которых подчеркивает важность среды .NET Compact Framework и SQL CE/SQL Mobile, а также ведет практические занятия по разработке приложений для мобильных устройств. Он возглавляет форум MSDN SQL Mobile и является автором достаточно популярного блога по адресу `www.dotnetjunkies.com/WebLog/darrenshaffer/default.aspx`. Даррен закончил колледж Уэст-Пойнт и служил офицером связи в армии США. Проживает с женой и дочерью в штате Колорадо.

Крис Шоу (Chris Shaw) — технический редактор и автор главы 52, посвященной повышению доступности сервера, президент компании SQL on Call в городе Колорадо-Спрингс, штат Колорадо. Он сотрудничал с компаниями Yellow Pages, Ford Fairline Motor Credit, AdminiQuest, Wells Fargo и AT&T Wireless, опубликовал множество статей для таких журналов, как *SQL Server Magazine* и *SQL Standard Magazine*. Крис является членом группы Rocky Mountain Microsoft Community Insiders, президентом группы Colorado Springs SQL Server Users и одним из организаторов семинаров Colorado PASS Camp. Он регулярно участвует в семинарах PASS Summits и SQL Connection. Последние его презентации были посвящены репликации в SQL Server 2005 и мониторингу производительности.

Эндрю Уатт (Andrew Watt) — обладатель статуса Microsoft SQL Server MVP с 2004 года. Он написал главу 33, посвященную интеграции программ InfoPath и SQL Server 2005. Эндрю — независимый консультант и опытный автор компьютерных книг.

Майкл Уайт (Michael White) занимается разработкой баз данных и их администрированием с 1992 года. Сосредоточившись с 2000 года на инструментах бизнес-логики компании Microsoft, он занимался проектированием и реализацией крупных хранилищ данных и приложений службы анализа. После многих лет работы в сфере корпоративных информационных технологий Майкл стал разработчиком программного обеспечения в компании Wall Street On Demand. В настоящее время Майкл возглавляет форум поддержки пользователей SQL Server в городе Боулдер, штат Колорадо, и является одним из организаторов семинаров Colorado PASS Camp. Майкл написал главу 42, посвященную службе интеграции, главу 43, о реализации бизнес-логики в службе анализа, главу 44, в которой описываются вопросы раскрытия данных, главу 45, посвященную программированию запросов MDX, а также главу 48, посвященную анализу данных в Excel.

Билл Вундер (Bill Wunder) — обладатель статуса Microsoft SQL Server MVP, написал главу 27, посвященную программированию сборок CLR в SQL Server, главу 29, в которой описываются пользовательские типы данных, и главу 30, посвященную вопросам программирования с использованием библиотеки ADO.NET 2.0. Технологии баз данных Билл начал постигать еще в 1993 году, когда занимался разработкой приложений для AS/400. Вскоре после этого он стал администратором базы данных в компании, занимающейся автоматизацией в торговле. База данных как раз была модернизирована до версии SQL Server 4.2.1 и работала под управлением операционной системы Novell. С тех пор Билл работает исключительно с SQL Server. Свои сценарии для Интернета он начал писать в 1995 году. На различных Web-сайтах он опубликовал более сотни статей, посвященных SQL Server. Он выступал на конференциях Back Office Administrators и Professional Association for SQL Server (PASS), которые проводила компания swynk.com в Америке и Европе. Он организовал форум поддержки пользователей SQL Server в городе Боулдер, штат Колорадо, и был его активным участником. С 2004 по 2006 год он получал статус MVP от компании Microsoft. В настоящее время Билл работает администратором баз данных на Кюрасао. Читатели могут связаться с ним по адресу `bwunder@yahoo.com`.

Введение

SQL Server — непревзойденный продукт, предназначенный для работы с базами данных. В этой СУБД великолепно сочетаются производительность, надежность, простота администрирования, новые технологии. Она позволяет администраторам баз данных при желании справляться с любой проблемой буквально за минуты. С точки зрения программиста баз данных, SQL Server можно без преувеличения назвать системой мечты. Что же касается рассматриваемой в настоящей книге версии SQL Server 2005, то она открывает ряд новых возможностей для разработки более масштабируемых и мощных систем. Основная цель книги — разделить с вами удовольствие от работы с этой СУБД.

Как и во всех книгах данной серии, вы найдете здесь пошаговые руководства и практические примеры, а также справочную информацию, посвященную обсуждаемой теме. Но чтобы описать все множество команд SQL Server, потребовался бы не один том, поэтому вторая цель книги — предложить вам лаконичное, но полноценное руководство, основанное на моем личном опыте разработчика приложений, консультанта и преподавателя. К тому времени, когда вы перевернете последнюю страницу книги, вы будете великолепно подготовлены к разработке и обслуживанию собственных баз данных.

Некоторые из моих постоянных читателей знакомы с моей книгой *SQL Server 2000 Bible*. Но даже если вы знаете мой подход к изложению материала, то в этом издании вы найдете ряд новых тем, в частности:

- обзор новых элементов архитектуры, таких как брокер служб, Web-службы, сервер отчетности и запросы XQuery;
- детальное описание реализации бизнес-логики;
- десяток глав, посвященных извлечению данных с помощью запросов.

Для кого предназначена книга

Кем бы вы ни были — разработчиком программного обеспечения, или администратором баз данных, или даже если вы только начинаете работать в этой предметной области, пусть у вас всего один год опыта работы или целых пять, — в этой книге вы найдете для себя множество полезного. Эта книга написана так, чтобы вы могли извлечь уроки из моего личного опыта и опыта моих соавторов. Каждая глава начинается так, будто у вас лично нет ни малейшего представления о данной теме. Далее, по мере чтения главы, вы начнете понимать обсуждаемые вопросы.

В следующем разделе вы поймете, как структурирована книга и что вы сможете найти в каждой конкретной главе. Это позволит вам определиться, с чего начинать. Независимо от своего опыта, обязательно ознакомьтесь с главой 1, где описаны два ключевых принципа, на которых основана остальная часть книги. Все последующие главы станут вам более понятными, если вы усвоите эти принципы и теорию оптимизации.

Структура книги

Одним из основных отличий этой книги, на которое сразу обратят внимание читатели предыдущих изданий, является новая организация материала. Я попытался отделить текст от листингов и сделать акцент на определенных темах, важных в контексте версии SQL Server 2005. Некоторые главы содержат всего десять страниц, в то время как другие — по тридцать и более.

Главы книги организованы так, чтобы их можно было читать последовательно, поскольку материал последующих глав основан на предыдущих. В то же время многие мне говорили, что ее можно использовать и как настольный справочник. В любом случае я желаю, чтобы эта книга стала полезной вам и помогла в работе.

Часть I. Основы

Часть I является введением в SQL Server. Начиная с базовых принципов информационной архитектуры и теории оптимизации, мы рассмотрим шесть ключевых тем, которые помогут вам успешно начать работать с SQL Server.

- Принципы информационной архитектуры.
- Моделирование реляционных баз данных.
- Архитектура SQL Server.
- Инсталляция пакета.
- Подключение клиентских программ.
- Утилита Management Studio.

Рассматривайте часть I как достаточно подробное изложение современного состояния информационной архитектуры и введение в основные технологии SQL Server.

Часть II. Манипулирование данными с помощью инструкции SELECT

В этой части мы начнем с изучения основных задач отбора данных, после чего рассмотрим отношения между таблицами, коррелированные подзапросы и распределенные запросы. Инструкции SELECT я посвятил десять глав, поскольку понимание правильных методик создания запросов — ключ к тому, чтобы стать успешным разработчиком или администратором SQL Server

Пожалуйста, не думайте, будто часть II предназначена только для начинающих. В этих десяти главах описано ядро SQL. В части VI мы поговорим о стратегиях оптимизации, однако следует помнить, что основа любой стратегии — качественный код, поэтому данные десять глав помогут вам разобраться с оптимизацией обычных запросов.

Часть III. Среда разработки SQL Server

В этой части вы сможете вплотную приступить к обработке данных. Здесь вы ознакомитесь с командами DDL ALTER, CREATE и DROP и сможете глубже изучить язык Transact-SQL.

Список технологий SQL Server продолжает расти, поэтому в этой части вы также узнаете об общеязыковой среде выполнения CLR, версии SQL Server Mobile, брокере служб, языках XML и XQuery, Web-службах и системе InfoPath. Это поможет вам заметно повысить уровень своей квалификации.

Часть IV. Управление данными на уровне предприятия

Эта часть посвящена тому, чтобы научить вас как администратора базы данных предприятия заставить SQL Server работать без сбоев. Несмотря на то что основные задачи обслуживания в SQL Server становятся все более автоматизированными и Microsoft иногда обманчиво представляет SQL Server как СУБД, не требующую присутствия администратора, правда заключается в том, что SQL Server должна работать круглосуточно, 7 дней в неделю и 365 в году.

Часть V. Бизнес-логика

Данная часть познакомит вас с самыми сложными концепциями SQL Server 2005. Технологии анализа исторических и текущих данных в настоящее время стремительно развиваются, и, вероятнее всего, продолжат развиваться в будущем. Эти технологии принято называть *бизнес-логикой* (Business Intelligence, BI).

В последних трех выпусках пакета SQL Server компания Microsoft постепенно развивала службы бизнес-логики, и версия SQL Server 2005 подвела итог многолетним усилиям. Сегодня СУБД готова к решению самых взыскательных задач бизнес-логики, располагая такими средствами, как инструменты ETL уровня предприятия, кубы данных и служба отчетности.

Часть VI. Стратегии оптимизации

Теория оптимизации раскрывает зависимости между различными технологиями оптимизации. В завершающей части мы применим эту теорию на практике. Здесь описаны технологии, гарантирующие *постоянную доступность* информации. Если SQL Server работает медленнее, чем хотелось бы, то после прочтения глав этой части вы заставите СУБД творить чудеса.

Часть VII. Приложения

В приложениях вы найдете справочную информацию, применимую практически к любой главе: спецификации SQL Server и описание примеров, приведенных в книге.

Как использовать данную книгу

Прежде чем начать знакомство с SQL Server, вам следует узнать о некоторых соглашениях, принятых в данной книге.

Принятые соглашения

В книге используются различные типы шрифтовых выделений. Все это предназначено для того, чтобы вам легче было ориентироваться в тексте книги.

- Новые термины, впервые упоминаемые в книге, выделены *курсивом*.
- Фрагменты программного кода выделены моноширинным шрифтом.

- Некоторые примеры программного кода могут показаться чересчур длинными. Чтобы обратить ваше внимание на основные фрагменты, заслуживающие внимания, они будут выделены полужирным шрифтом.
- Я не являюсь поклонником особо длинных результатов запросов, поэтому малосущественные фрагменты будут заменяться троеточиями.

Пиктограммы

Когда потребуется привлечь ваше внимание к некоторому фрагменту текста, будут использоваться специальные значки, называемые пиктограммами.



Предупреждения содержат информацию, на которую следует обратить внимание. Это могут быть малозаметные детали, а также потенциально опасные проблемы, с которыми можно столкнуться в работе.



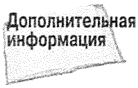
Советы содержат сведения, позволяющие облегчить работу с программой.



Примечания — это дополнительная, необязательная для запоминания информация, которая тем не менее может пригодиться.



Этой пиктограммой помечаются рекомендации по выполнению некоторых операций, основанные на личном опыте авторов книги.



Перекрестные ссылки направят вас к другим ресурсам, не обязательно находящимся в этой книге. Воспринимайте их как дополнение к основному материалу. В некоторых местах книги вы встретите повторения материала. Например, при установке SQL Server вам следует выбрать метод аутентификации пользователей. Вместо того чтобы направлять вас к другим разделам книги, я попытаюсь дать краткую информацию, которая поможет сразу решить проблему. Но все равно в разных частях книги вы встретите множество перекрестных ссылок, которые помогут вам найти более подробную информацию по обсуждаемому вопросу.



В некоторых местах вы встретите ссылки на Web-сайт `www.SQLServerBible.com`, где можно найти дополнительные ресурсы.

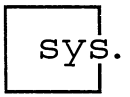


Данную пиктограмму описывать излишне — она просто поможет вам разобраться, что было раньше и что появилось в текущей версии.



Мне пришлось достаточно много поэкспериментировать с копиями экрана, чтобы сделать примеры как можно более наглядными. Я объединил некоторые рисунки в демонстрационные ролики ScreenCast, которые помогут вам разобраться с выполнением специфических задач в SQL Server. Например, чтобы увидеть ситуацию взаимоблокировки в окне SQL Server Profiler (глава 49), вы

можете прочитать последовательность команд в тексте и при этом просмотреть ее в виде ролика. Соответствующие ролики доступны на сайте книги по адресу `www.SQLServerBible.com`.



SQL Server 2005 имеет множество новых системных представлений, которые покажут вам состояние системы намного подробнее, чем в предыдущих версиях пакета. Эта пиктограмма обратит ваше внимание на новые возможности сервера.

Что можно найти на Web-сайте книги

На сайте издательств Wiley (`www.wiley.com`) и "Диалектика" (`www.dialekika.com`) вы сможете найти примеры программного кода и базы данных, используемые в книге. Все примеры сгруппированы в один архив, и каждая глава имеет в нем собственную подпапку.

На Web-сайте самого автора (`www.SQLServerBible.com`) вы сможете получить ту же информацию и кое-что еще:

- примеры программ и баз данных;
- исправления ошибок и дополнительные статьи;
- ресурсы и ссылки, связанные с SQL Server;
- демонстрационные ролики ScreenCast.

Ждем ваших отзывов!

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

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

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

Наши электронные адреса:

E-mail: `info@dialektika.com`

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

Наши почтовые адреса:

в России: 115419, Москва, а/я 783

в Украине: 03150, Киев, а/я 152

Основы

ЧАСТЬ



Начиная изложение принципов информационной архитектуры и теории оптимизации, в части I настоящей книги мы представим шесть ключевых тем, которые заложат основу успешного освоения SQL Server.

В своей книге *Бизнес со скоростью мысли* Билл Гейтс предложил концепцию "цифровой нервной системы". Его главная идея состояла в том, что эффективность работы любой организации возрастет, если она соберет всю свою жизненно важную информацию и сделает ее доступной для сотрудников компании. С каждым годом профессионалы информационных технологий все ближе подходят к глобальному видению концептуально важной для предприятия информации. Модели и технологии баз данных вносят немалый вклад в ускорение этого процесса. В новом столетии эти технологии стали развиваться на порядок быстрее, чем три десятилетия назад, когда доктор Эдгар Кодд представил миру свои революционные концепции реляционных баз данных. Без преувеличения, это можно назвать временем перемен.

Платформы баз данных нового поколения используют архитектуры "клиент/сервер", многоуровневую и ориентированную на службы (SOA), задействуя при этом XML и Web-службы. Это намного более масштабируемые и гибкие технологии, чем те, которые использовались раньше. В дополнение ко всем этим изменениям новые технологии в целях повышения масштабируемости, производительности и доступности данных используют такие новые средства, как общеязыковой интерпретатор CLR, что позволило усовершенствовать доставку бизнес-информации и управление бизнес-процессами.

В целом часть I этой книги нельзя рассматривать лишь как введение в теорию нормализации базы данных. Скорее это — современный взгляд на информационную архитектуру и технологии, задействованные в SQL Server.

В этой части...

Глава 1
Принципы информационной архитектуры

Глава 2
Моделирование реляционных баз данных

Глава 3
Архитектура SQL Server 2005

Глава 4
Установка SQL Server 2005

Глава 5
Подключение клиентского программного обеспечения

Глава 6
Использование утилиты Management Studio

ГЛАВА

1

Принципы информационной архитектуры

В этой главе...

Основные принципы
Простота
Полезность
Целостность данных
Производительность
Доступность
Масштабируемость
Безопасность

В любой сложной области в первую очередь стоит заняться разработкой общих принципов, управляющих проектированием данных, процедур и решений. Заслуживающие доверия принципы должны быть понятными, надежными, полноценными и устойчивыми. Когда общие принципы определены, они могут стать своеобразным мерилом конфликтующих мнений, в результате чего удастся выработать согласованные стандарты.

Следующий принцип информационной архитектуры охватывает три основных аспекта управления информацией: проектирование и разработку базы данных, организацию централизованного управления данными предприятия и интеллектуальный бизнес-анализ.

> *Информация является организационным активом и, согласно своему содержанию и значению, должна быть соответствующим образом организована, учтена, защищена и сделана доступной в удобном формате для ежедневных операций и анализа отдельными сотрудниками, их группами и процессами как сегодня, так и в будущем.*

Расшифровка этого принципа подразумевает несколько практических реализаций. Во-первых, должно существовать хранилище информации; должно быть известно его местонахождение, источник, конфиденциальность, настоящая и будущая ценность и текущий владелец. В то время как большинство организаций хранят свою информацию в глобальных базах данных, особо конфиденциальную информацию они хранят на локальных рабочих станциях; также она может быть распределена по всей организации.

Так же как ценность физических активов варьируется в зависимости от места их нахождения и времени, ценность информации также является переменной величиной, и ее следует

измерять. Ценность информации может быть велика для отдельного индивидуума или подразделения, и в то же время она может совершенно не представлять никакого интереса для всей организации. Информация, критичная сегодня, может потерять свое значение завтра или через месяц. Аналогично, информация, несущественная для отдельного сотрудника, может в обобщенном виде представлять большую ценность для всей организации, в частности, если речь идет о планировании ее развития.

Если данные должны быть легко доступными в будущем, то модель хранилища должна избегать блокирования информации в базе данных.

Следуя ключевым принципам информационной архитектуры, любое хранилище данных должно подчиняться семи независимым основным правилам: простота, полезность, целостность данных, производительность, доступность, масштабируемость и защищенность.



В настоящей главе закладывается фундамент, на котором будут базироваться все последующие главы книги. В то же время на рассматриваемых принципах основаны и некоторые расширенные концепции баз данных. Если вы мало знакомы с базами данных, то рекомендую сразу ознакомиться с главой 3 и только затем вернуться к настоящей главе, в которой описаны принципы информационной архитектуры.

Простота или сложность

Недостаточное внимание к основным принципам архитектуры информации и остальным правилам хранения данных — это методология проектирования, которую я в шутку называю *векселем*. Если для программистов компании неудачный проект может в худшем случае закончиться увольнением или понижением зарплаты, то для независимого консультанта последствия более чувствительные — по векселю придется платить.

Как консультанту в области баз данных мне нравится участвовать в особо рискованных проектах — тех, которые клиент уже рассматривает как катастрофу (в случае, если он обжегся на услугах других консультационных компаний). В данном случае клиент просит меня завершить оставшиеся 10% проекта или оптимизировать его, если тот не в полной мере соответствует изначальным требованиям. И, как правило, причину катастрофы я нахожу в изначально обреченном на провал проекте.

Первый урок, который я вынес из доброго десятка провалившихся проектов баз данных, следующий: главной причиной неудач являются вовсе не малокомпетентные программисты, а излишне сложный проект. А все потому, что проектировщики базы данных не смогли (или просто не захотели) придумать более простое, элегантное решение задачи. Также несколько уроков я вынес из собственного опыта, когда мне приходилось платить по своим "векселям".

Так называемый "антивексельный" принцип проектирования я могу сформулировать следующим образом.

> *Избегайте сложностей. Старайтесь упрощать модель базы данных до тех пор, пока вся команда разработчиков единогласно не согласится, что это максимально простое решение, удовлетворяющее требованиям задачи.*

Сложность

Одна сложность порождает другую сложность, что в конце концов вызывает проблемы. Наиболее распространенным результатом излишней сложности является полный провал проекта. Даже когда сложность кажется оправданной, она редко удовлетворяет конечной цели проекта. Сложность проекта также усложняет работу программистов в части понимания и реализации, что оттягивает сроки сдачи проекта.

Если проект уже реализован, сложность плохо сказывается на остальных шести правилах хранения данных (полезность, целостность данных, производительность, доступность, масштабируемость и защищенность). Сложная модель усложняет извлечение и обновление корректных данных, что влияет на их полезность и интеграцию. Дополнительные компоненты порождают излишнюю работу (чтение, объединение и обновления), взаимозависимые переменные, а также дополнительные сложности в настройке. Все это является причиной низкой производительности решений.

Сложная модель потенциально создает дополнительные точки сбоя. Когда такой сбой возникает, излишняя сложность маскирует источник проблемы, что, разумеется, усложняет и ее решение. Это приводит к удорожанию эксплуатации сложной системы. По сравнению с простой моделью сложную чрезвычайно тяжело адаптировать к изменяющимся условиям. И наконец, сложный набор компонентов уменьшает шансы надежной защиты данных.

Кстати, в тезаурусе синонимом слова "сложный" является слово "трудный".

Простота

Простое решение элегантно и кажется очевидным, однако сказать, что поиск такого решения прост, все равно что, смотря в телевизор, думать, что медалисты олимпийских игр с легкостью устанавливают свои рекорды. В связи с этим хотелось бы процитировать Эйнштейна: "Вещи нужно делать как можно более простыми — но не проще этого".

Фразу "как можно проще" нельзя рассматривать как односложный ответ на вопрос. Простейшее решение должно быть самым простым среди тех, которые полностью удовлетворяют требованиям поставленной задачи, — оно обречено на определенный уровень сложности. Простота измеряется количеством компонентов, техническими составляющими, внутренним интерфейсом и технологиями, вовлеченными в решение.

Разработка простого решения в какой бы то ни было области — очень сложная задача, которая требует следующих составляющих:

- полное понимание требований задачи;
- широкий выбор моделей и решений;
- полное понимание технических правил и нюансов предметной области;
- творческий подход и профессионализм;
- полное знание физических средств и материалов, используемых в проекте, а также среды, в которой решение должно функционировать;
- достаточный уровень доверия в команде разработчиков, позволяющий идеям развиваться и видоизменяться только на основе их ценности, а не на основе личных амбиций и заслуг их создателей;
- соглашение о том, что разработка будет продолжаться, пока не будет найдено самое элегантное решение;
- разумная боязнь сложности.

При разработке проекта за несколько часов или дней совещаний можно выработать идею, которая позволит избежать массы проблем, ранее казавшихся непреодолимыми. Любое оправданное уменьшение сложности значительно сократит усилия на реализацию проекта и повысит шансы получить действительно работоспособный продукт. Каждый лишний рубль, вложенный в процесс проектирования, окупится сторицей в процессе реализации и, тем более, эксплуатации продукта.

Простота именования объектов является одной из составляющих общей простоты модели базы данных. Из собственного опыта знаю, что если команда разработчиков уловит эту простоту, то существует большой шанс успеха всего проекта. Если команда не ценит простоту, я не хотел бы работать в ней.

Правило полезности

Вторым правилом проектирования хранилища данных является полезность. Под полезностью хранилища данных понимают его соответствие требованиям организации, удобство модели с точки зрения внесения изменений, эффективность формата данных для использующих его приложений, а также простоту извлечения информации. Наиболее распространенной причиной уменьшения полезности базы данных является чрезмерная сложность или непродуманность ее модели.

Удобство модели

В зависимости от назначения и наполнения хранилища данных, можно выбрать одну из нескольких моделей.

Состав данных базы должен учитывать гранулярность использования информации в организации (индивидуумом, подразделением, организацией), а также их временную природу (для ежедневного использования, для накапливания в течение фискального года или более продолжительного времени). Строгость модели и реализации должна быть пропорциональна составу данных.

В табл. 1.1 перечислены типы моделей хранилищ данных и указана степень их соответствия разным атрибутам конфигурации (○ обозначает плохую, ◖ — ограниченную, а ● — полную поддержку).

Таблица 1.1. Пригодность разных типов хранилищ данных

Атрибут	Реляционная СУБД	Объектная СУБД	Объектно-реляционная СУБД	Общая модель	Хранилище данных
Подходит для главного хранилища данных	●	◖	●	◖	○
Подходит для ссылочного хранилища данных	◖	◖	◖	○	●
Производительность извлечения данных	◖	◖	◖	○	●
Гибкость схемы	◖	●	●	●	○
Простота запросов SQL и традиционных инструментов отчетности	◖	○	◖	○	●
Качество поддержки производителя	●	○	◖	○	●
Требование включает несколько отношений “is-a”	○	●	●	○	○
Возможность хранения сложных типов данных	◖	●	◖	○	◖
Возможность хранения ассоциаций со многими отношениями	◖	◖	●	○	◖
Простота работы и настройки	◖	◖	●	◖	●
Поддержка объектов приложений	○	●	●	◖	○
Предотвращение аномалий обновления данных	●	○	●	◖	○

Конфигурации хранилищ данных

Как показано на рис. 1.1, хранилище данных предприятия может состоять из множества источников самых разных типов: главного, кэширования, ссылочного, хранилища информации и витрины данных.

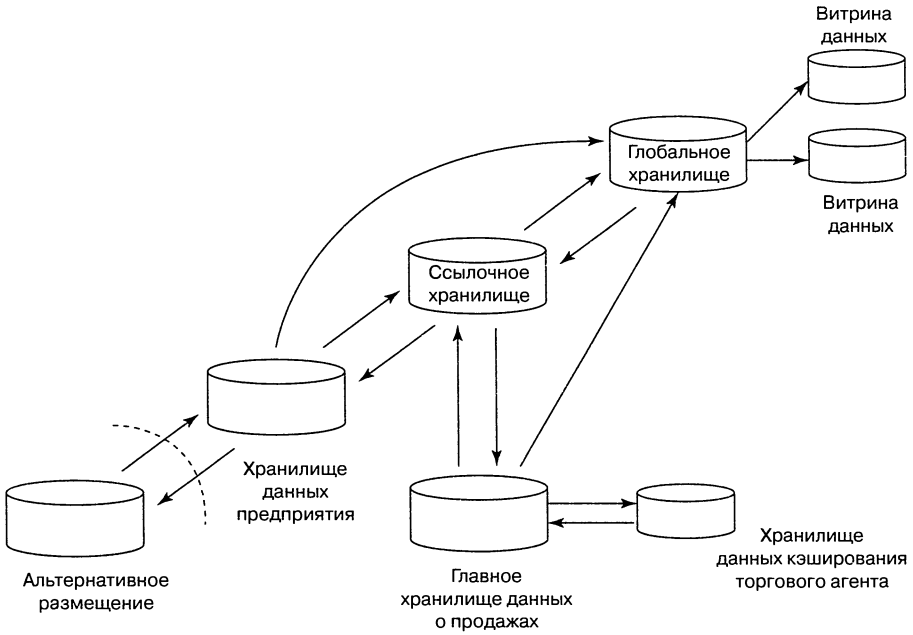


Рис. 1.1. Типовая конфигурация хранилища данных организации включает в себя несколько главных хранилищ

Главное хранилище данных, также называемое *операционной базой данных* или *базой данных оперативной обработки транзакций (OLTP)*, используют для сбора транзакционных данных, существенных для каждодневных операций организации, уникальных для нее. Как правило, в нем хранится информация о клиентах, заказах и методах доставки. Каждая организация может иметь одно или несколько главных хранилищ данных для обслуживания своих видов деятельности или подразделений.

Для увеличения производительности главное хранилище данных настраивают для поддержания баланса между обновлениями и извлечением данных. Так как этот тип баз данных содержит исходную информацию, он подвержен аномалиям обновления и выигрывает от нормализации (подробнее об этом мы поговорим в следующем разделе).

Согласно концепции Билла Гейтса, главное хранилище данных как часть "цифровой нервной системы" собирает информацию от всех элементов организационной структуры и группирует ее удобным образом для обработки остальной частью структуры информационной системы. Это хранилище используют для быстрой обработки информации и непосредственного контакта с окружающей средой. Например, быстро решая задачу обработки заказов, главное хранилище данных выступает в роли части нервной системы, реагирующей на раздражения.

Хранилище данных кэширования является необязательной и доступной только для чтения копией главного хранилища. Оно не является обязательным элементом структуры хранения информации и используется для снятия нагрузки с основного хранилища в части доставки информации конечным потребителям. Главное хранилище данных может иметь множество

кэширующих хранилищ для обработки запросов сотрудников организации. Оно может располагаться на среднем уровне информационной системы или представлять собой основу Web-службы. Кэширующее хранилище данных настраивается с целью повысить производительность извлечения данных.

Ссылочное хранилище данных доступно только для чтения и используется организацией для хранения редко изменяемой информации. В чем-то его можно сравнить с каталогом в библиотеке. Примерами ссылочных данных могут служить коэффициенты преобразования единиц измерения, а также индексы городов. Это хранилище данных также настраивается с целью ускорения извлечения информации.

Глобальное хранилище данных собирает громадные объемы информации из множества главных хранилищ данных, рассредоточенных по предприятию. В нем используется специальный процесс *Exstract-Transform-Load (ETL)* ("извлечь–преобразовать–загрузить") для преобразования данных из различных форматов в единый, специально предназначенный для облегченного извлечения информации. Глобальные хранилища данных чаще всего используются в качестве места размещения архивов, хранения исторических данных и освобождения оперативных хранилищ от редко используемых данных. Эти данные чаще всего консолидированы, что облегчает поиск и отчетность, а также сокращает вероятность ошибок.

Обычно хранилища данных используют как гарант того, что все подразделения организации используют одну и ту же информацию и в ответ на идентичные запросы получают идентичные наборы данных. Это является критичным аспектом закона Сарбанеса–Оксли и прочих законодательных требований.

Витрины данных являются подмножествами глобальных хранилищ, где консолидированные данные специально отобраны для обслуживания некоторой организационной группы или предметной области.

В процесс анализа обычно вовлечены не только обычные запросы SQL; он предполагает использование кубов данных, консолидирующих гигабайты информации в динамическую перекрестную таблицу. Бизнес-логика представляет собой комбинацию процесса ETL и хранилища данных и служит для создания и просмотра кубов.

В созданной Биллом Гейтсом "цифровой нервной системе" хранилище данных выступает в роли памяти организации. В ней хранится история, она используется для раскрытия данных, в частности для анализа тенденций (в чем и почему организация терпит поражение на рынке и в чем выигрывает). Эта часть цифровой нервной системы используется организациями для размышлений относительно того, как преодолеть проблемы и добиться успеха. В сущности, это и является основной задачей хранилищ данных и кубов бизнес-логики.

Поскольку основной задачей глобальных хранилищ данных является извлечение и анализ информации, проблема целостности данных, присутствующая в главных хранилищах, в данном случае не возникает. Глобальные хранилища создаются для быстрого извлечения информации; они не нормализованы, как главные хранилища. Эти хранилища, как правило, используют базовые схемы "звезды" или "снежинки"; блокировки обычно не применяются, а индексация выполняется без непосредственного влияния на операции вставки и обновления.

Стили проектирования главных хранилищ данных

Проектировщики баз данных не ограничены одной реляционной моделью. Существует несколько типов моделей баз данных, выбор из которых зависит от требований проекта.

Реляционные базы данных

Реляционные базы данных стали уже традиционными. Они объединяют сходные или связанные данные в единую таблицу. Реляционные базы идеально подходят для устойчивых схем данных, содержащих относительно немного отношений типа "заказчик является контактом".

Объектно-ориентированные базы данных

Объектно-ориентированные базы данных согласуют структуру данных с принципами объектно-ориентированного проектирования, чтобы обеспечить поддержку объектов самого приложения. В основе этого подхода лежит концепция, что любой объект является экземпляром некоторого класса. В классе определены все свойства и методы объекта; проще говоря, в нем запрограммирован объект. Каждый экземпляр объекта может иметь собственные внутренние переменные и взаимодействовать с внешним миром сам по себе. Несмотря на то что терминология в разных источниках отличается, основная концепция организации данных в этом случае аналогична представленной в табл. 1.2.

Таблица 1.2. Сравнение терминологии баз данных

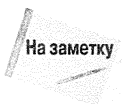
Стиль разработчиков	Общий список	Элемент списка	Элемент информации в списке
Электронная таблица	Электронная таблица, или рабочий лист, или именованный диапазон	Строка	Столбец или ячейка
Историческая информация	Файл	Запись	Поле
Реляционная алгебра или логическое проектирование	Сущность	Кортеж или отношение	Атрибут
SQL или физическое проектирование	Таблица	Строка	Столбец
Объектно-ориентированный анализ и проектирование	Класс	Экземпляр объекта	Свойство

Так как объектно-ориентированные базы данных в основном хранят объекты, их ключевой характеристикой является способность хранить сложные объекты, такие как файлы XML или классы .NET.

Объектно-ориентированные СУБД подходят для приложений, в которых ожидаются существенные изменения схемы, включение сложных типов данных, множественные отношения между классами, в том числе мультиассоциации, и требуется легкость подключения данных к приложению.

Существуют три основных типа объектно-ориентированных баз данных.

- *Хранилища данных поддержки объектов.* Они предназначены для ненамного большего объема задач, чем хранение состояний объектов. Все вопросы интеграции обслуживаются программным кодом объектно-ориентированных приложений, а сама схема базы данных проектируется так, чтобы быть как можно ближе к диаграмме классов приложения.
- *Объектно-ориентированные хранилища данных.* Эти хранилища поддерживают объекты приложения и используют метаданные (данные, описывающие организацию других данных) для моделирования объектно-ориентированной структуры классов и обеспечения совместимости объектов и классов.
- *Объектно-реляционные хранилища данных.* Эти хранилища поддерживают объекты приложений и моделируют структуру классов в реляционной СУБД. Такие базы совмещают преимущества объектно-ориентированных моделей с традиционными возможностями запросов и отчетов реляционных баз данных.



Более полную информацию о новой модели объектно-реляционных баз данных Nordic вы можете получить на сайте автора (`www.SQLServerBible.com`).

Обобщенная модель СУБД

Обобщенная модель СУБД (рис. 1.2) иногда используется в качестве объектно-ориентированной модели в реляционных базах данных. Эта модель может оказаться полезной, когда приложениям требуются динамические атрибуты. Например, спецификации материалов, используемых на производственном предприятии, будут требовать наличия разного состава атрибутов для разных материалов. Усложняя вопрос, можно сказать, что часто отслеживаемые атрибуты внутри компании изменяются в зависимости от процессов контроля качества (TOM) или ISO 9000. Это требует от обычной реляционной базы данных наличия сущностей для каждого типа материала, что, в свою очередь, связано с постоянными изменениями схемы.

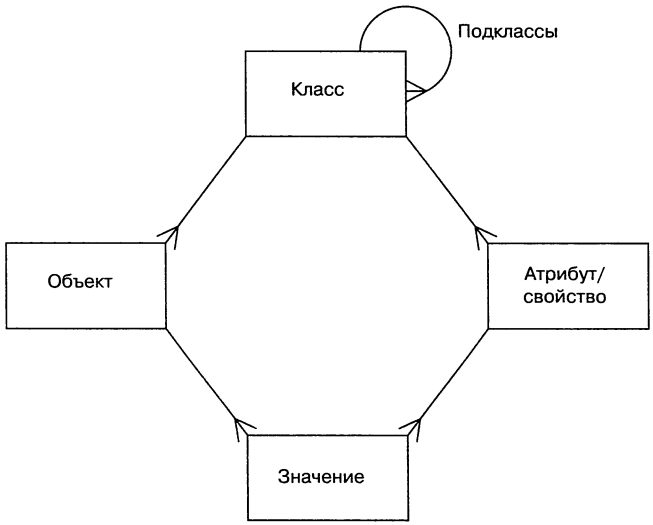


Рис. 1.2. Обобщенная схема, при которой в реляционную СУБД внедрены элементы объектно-ориентированной СУБД

Архитектура предприятия

Термин *архитектура программного обеспечения* имеет разное значение в зависимости от того, кто и в каком контексте его использует.

- ✧ Когда компания Microsoft использует термин *архитектура*, она обычно подразумевает свою среду .NET Framework или проектирование систем, требующих привлечения нескольких продуктов Microsoft для решения задачи.
- ✧ Термин *архитектура продукта* относится к компоновке различных компонентов в структуре программы.
- ✧ Термин *архитектура инфраструктуры* относится к разработке сетей, включающих в себя маршрутизаторы, коммутаторы, брандмауэры, серверы и прочие компоненты.
- ✧ Термин *архитектура данных* относится к организации и конфигурации хранилищ данных.
- ✧ Термин *архитектура предприятия* подразумевает управление всеми этими архитектурами в масштабах организации и обычно подразделяется на три области: инфраструктура, приложение и данные.

Насколько совершенна инфраструктура предприятия — предмет многочисленных дебатов. Используя метафору индустрии строительства, представление об инфраструктуре предприятия варьируется от пассивной проверки проектов на соответствие строительным нормати-

вам до активного составления плана общегородской застройки. Так как каждая точка зрения основывается на некотором множестве архитектурных принципов, они отличаются тем, в какой мере вовлечены эти принципы и какую инициативу использует команда архитекторов при их планировании.

С точки зрения строительных норм мы не начинаем с рекомендаций клиенту (точнее, представителю организации, принимающему решения), какое программное обеспечение он может использовать. Вместо этого команда архитекторов отвергает или принимает предложения клиента, основываясь на их соответствии архитектурным принципам и стандартам. Если предложения не соответствуют стандартам, клиент может предложить альтернативный вариант, используя стандартную процедуру (запрос на изменение). Какое приложение будет создано, как оно будет спроектировано и кто именно будет его писать, зависит целиком от клиента, а не от архитектора программы.

С точки зрения привязки проекта мы принимаем представление строительных норм и оцениваем их относительно плана развития программной структуры предприятия. Например, архитектор информационной структуры предприятия должен определить, что предприятию действительно нужен данный тип приложения, и сформулировать требования относительно того, как привязать его к инфраструктуре организации. В то же время он оставляет реальное программирование или покупку приложения на откуп клиенту.

Точка зрения городского планирования наиболее агрессивная и требует участия технического руководства предприятия. Как и в градостроительстве, архитектор предприятия должен попытаться предсказать потребности организации и работать напрямую с клиентом, чтобы точно определить нужный пакет операций и общую архитектуру. После этого архитектор представляет организации свой план будущего развития, создавая или закупая программный продукт, перерабатывая существующие программы и удаляя то, что не соответствует разработанному плану. Успех такого подхода обеспечивается двумя факторами. Во-первых, план развития разрабатывается совместно клиентом и архитектором, во-вторых, этот план перепроверяется и обновляется как с организационной, так и с технической стороны.

Вопрос управления несет в себе потенциальные проблемы. Если архитектор, который заложил принципы и стандарты, становится главой проекта, возникает конфликт интересов (законодатель, исполнитель, эксперт и судья). Он также ставит архитекторов программы в затруднительное положение, что может вызвать конфликт между организацией, разрабатывающей программный продукт, и заказчиком. Решение здесь одно: изменить механизм организации работы. Архитектор должен непосредственно передавать предлагаемые принципы разработчику программы. Если тот понимает выгоду, приносимую организации, и принимает предложенные принципы (возможно, за некоторыми исключениями) как собственные, то организация и разработчик их реализуют в реальном проекте.

Сущность *класса* управляет схемой базы данных. Эта сущность включает в себя возвратное отношение для поддержки наследования объектов этого класса. Как в иерархической структуре и организационной диаграмме, это отношение подразумевает возможность наличия у каждого класса множества дочерних классов, но каждый класс объектов должен иметь только один базовый класс. Сущность *свойства* второстепенна по отношению к *объекту*; каждый объект может иметь множество свойств. Объектом называют единичный экземпляр класса. Как таковому ему необходимо иметь собственные значения каждого свойства класса; то же относится и к объектам дочерних классов.

Несмотря на то что результаты такого подхода могут быть впечатляющими, в этом процессе может возникнуть множество сложностей. Отношения "многие ко многим", которые существуют в реальной жизни, могут моделироваться объектно-ориентированными базами данных с помощью коллекций объектов. Свойства должны соответствовать определенному типу данных и правилам проверки, которые должны моделироваться схемой данных, а не встроенными в SQL Server типами данных и правилами проверки.

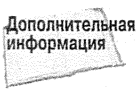
Целостность данных

Способность гарантировать корректность извлеченных данных является первым принципом информационной архитектуры. И одновременно это — главная проблема, которая возникает в мире баз данных. При отсутствии целостности данных ответ на любой запрос может оказаться некорректным, следовательно, уже нет особого смысла в обеспечении производительности и доступности.

Поскольку данные представляют собой сущности и атрибуты, их целостность обеспечивается целостностью сущностей, домена, а также ссылочной и определяемой пользователем целостностью. Целостность транзакций, т.е. порядок записи и извлечения данных, определяется принципами ACID (атомарность, постоянство, изоляция и живучесть), о которых мы поговорим далее в этой главе, сбоями транзакций и уровнем изоляции.

Целостность сущностей

Целостность сущности имеет дело с ее структурой (первичным ключом и атрибутами). Если первичный ключ уникален, а все атрибуты скалярны и полностью зависят от первичного ключа, то целостность сущности обеспечена. В физической схеме целостность сущности поддерживает первичный ключ таблицы. Обеспечение целостности сущности непосредственно связано с *нормализацией*.



Подробно нормализация будет описана в главе 2.

Целостность домена

В терминах реляционной теории *доменом* называют множество возможных значений атрибута, будь то целочисленных, битовых или текстовых. Понятие *целостности домена* подразумевает, что в атрибуте содержатся только допустимые данные. Допустимость нахождения в атрибуте пустых значений также входит в понятие целостности домена. В физической схеме основой поддержания целостности домена являются тип данных и допустимость пустых значений.

Ссылочная целостность

Понятие *ссылочной целостности*, прежде всего, связано с целостностью домена внешних ключей. Целостность домена подразумевает, что если некоторый атрибут имеет значение, значит, оно должно входить в домен. В случае внешних ключей доменом является список значений связанного первичного ключа.

В то же время ссылочная целостность является не вопросом целостности первичного ключа, а только связанного с ним внешнего.

Допустимость пустых значений в столбце — это отдельный вопрос, не связанный со ссылочной целостностью. Внешний ключ, в принципе, вполне может содержать пустые значения.

Некоторые методы поддержания ссылочной целостности реализуются на уровне физической схемы. В физической схеме может поддерживаться *декларативная ссылочная целостность* (DRI), и к таблице может быть прикреплен триггер.

Определенная пользователем целостность

В стороне от требований целостности реляционной теории существует и определяемая пользователем целостность. Она может поддерживаться следующим образом:

- простыми бизнес-правилами, такими как ограничение на домен, т.е. ограничение списка допустимых сущностей данных. Для поддержания этих правил в физической схеме часто используют ограничения;
- сложными бизнес-правилами, которые ограничивают список допустимых данных на основе какого-либо условия. Например, некоторые туры требуют наличия у туристов определенных медицинских справок. Реализация этих правил в физической схеме обычно требует написания хранимых процедур или триггеров.

Некоторые требования целостности данных не могут быть обеспечены ограничениями или триггерами. Неполные, ошибочные или недостоверные данные могут успешно пройти все проверки и попасть в базу данных. Например, заказ без единой товарной строки считается некорректным, но ни один автоматический метод не сможет запретить ему попасть в базу. Запросы SQL могут выявить незаполненные заказы и помочь в других менее важных вопросах поддержания целостности данных, в том числе в следующих:

- поиск некорректных данных;
- поиск неполных данных;
- поиск сомнительных данных;
- поиск несогласованных данных.

Качество данных непосредственно зависит от людей, которые их вводят или изменяют. Защита данных (т.е. контроль над тем, кто именно может просматривать или изменять их) также является одним из аспектов поддержания их целостности.

Целостность транзакций

Транзакцией называется единый логический блок работы, например вставка 100 строк, обновление 1000 строк или выполнение логической цепочки обновлений. Качество продукта базы данных зависит от того, насколько его возможности выполнения транзакций соответствуют принципам ACID. Как вы помните, эта аббревиатура расшифровывается как четыре взаимозависимых свойства: атомарность, целостность, изоляция и живучесть. Большая часть архитектуры SQL Server основана именно на этих свойствах, поэтому, чтобы понять SQL Server, нужно усвоить их значение.

- Свойство *атомарности* подразумевает, что транзакция должна либо выполниться вся, либо не выполниться в целом. В конце транзакции она должна быть либо подтверждена, либо отменена. Если частично выполненная транзакция записывается на диск, то свойство атомарности нарушается.
- Транзакция должна поддерживать *целостность* базы данных. Это значит, что транзакция начинает выполняться, когда база данных находится в целостном состоянии, при этом база должна остаться в целостном состоянии и после завершения транзакции. С точки зрения ACID целостность означает, что каждая строка и значение должны соответствовать моделируемой реальной ситуации, а все ограничения должны выполняться. Например, если заголовок заказа записан на диск, а строки заказанных в нем товаров — нет, то нарушается ссылочная целостность.

- Каждая транзакция должна быть *изолирована*, т.е. отделена от эффекта выполнения других транзакций. Независимо от того, что выполняет другая транзакция, первая должна иметь возможность продолжать работать с тем же набором данных, с которым начала. Изоляция является своеобразным забором между двумя транзакциями. Доказательством наличия изоляции является возможность многократно повторить последовательный набор транзакций над одним и тем же набором данных и каждый раз получить один и тот же результат.

  В качестве примера предположим, что некоторый Вася обновляет 100 строк. Пока его транзакция выполняется, некий Петя удалил одну из строк, над которыми работает Вася. Если такое удаление может произойти, значит, транзакция Васи не изолирована от транзакции Пети. Это свойство не так критично, если с базой данных работает всего один пользователь, однако в многопользовательской среде оно приобретает весомое значение.

- Понятие *живучести* транзакции относится к ее выполнению независимо от системных сбоев. После того как транзакция будет подтверждена, она остается подтвержденной. СУБД должна быть спроектирована так, чтобы базу данных можно было восстановить до состояния последней полностью выполненной транзакции, даже если устройство данных даст сбой.

Узким местом целостности транзакций является *конкуренция* — попытка множества пользователей одновременно извлечь или модифицировать одни и те же данные. Вопрос изоляции реже встает в маленьких базах данных, однако в производственной среде с тысячами пользователей конкуренция соревнуется с целостностью транзакций. Эти два процесса должны быть сбалансированы, в противном случае целостность данных или производительность сильно пострадает.

Архитектура SQL Server поддерживает все свойства ACID целостности транзакций. Это значит, что разработчику нужно понимать их, чтобы при создании баз данных взять на вооружение все возможности SQL Server, а администратору баз данных это поможет реализовать хороший план восстановления. Если вы объедините возможности SQL Server, аппаратного и программного обеспечения, модели базы данных, планов восстановления и обслуживания базы данных, то получите великолепный результат. Когда разработчик и администратор сотрудничают, чтобы правильно организовать эти компоненты, база данных работает отлично, а целостность транзакций высокая.

Ошибки транзакций

Понятие *полной изоляции* подразумевает, что одна транзакция никогда не помешает работе другой. Если изоляция полная, то все изменения, происходящие вне транзакции, она не увидит.

Изоляция между транзакциями может быть далеко не совершенной, что проявляется тремя способами: "грязное" чтение, неповторяющееся чтение и призрачные строки. К тому же транзакции могут завершаться неудачно из-за потерянных обновлений и взаимоблокировок.

"Грязное" чтение

Наиболее заметный сбой происходит, когда работа одной транзакции видна другой перед тем, как та подтвердит свои изменения. Когда одна транзакция может видеть неподтвержденные обновления другой, такая ситуация называется *"грязным" чтением*. Проблема "грязного" чтения состоит в том, что прочитанные данные еще не подтверждены, поэтому запись транзакцией данных может быть отменена.

Неповторяющееся чтение

Неповторяющееся чтение аналогично "грязному", однако оно происходит, когда одна транзакция в процессе работы видит подтвержденные в это время обновления другой транзакции. Чтение строки внутри транзакции всегда должно выдавать одни и те же результаты. Если при последовательных чтениях получаются разные результаты, значит, в транзакции произошла ошибка неповторяющегося чтения.

Призрачные строки

Наименее частой ошибкой являются *призрачные строки*. Подобно неповторяющемуся чтению, эта ошибка происходит, когда обновления другой транзакции повлияли не только на результирующий набор данных, но и вызвали извлечение инструкцией SELECT отличного набора строк.

Среди всех ошибок "грязное" чтение можно назвать самым опасным, неповторяющееся чтение — немного менее опасным, и практически самыми безобидными являются призрачные строки.

Потерянные обновления

Потерянные обновления происходят, когда два пользователя редактируют одну и ту же строку, завершают свою работу и сохраняют данные. При этом обновления, которые были выполнены и сохранены первыми, заменяются обновлениями, сохраненными вторыми.

Так как потерянные обновления происходят только в том случае, когда два пользователя одновременно модифицируют одну и ту же строку, эта проблема может не возникать месяцами. Тем не менее это брешь в целостности транзакций, и эту ошибку в базах данных нужно не допускать.

Взаимоблокировки

Взаимоблокировкой называется особая ситуация, когда несколько задач транзакций соревнуются за один и тот же ресурс данных. Для примера рассмотрим следующий сценарий.

- Первая транзакция блокировала данные А и собирается для своего завершения блокировать данные Б.
- В то же время вторая транзакция блокировала данные Б и собирается для своего завершения блокировать данные А.

Каждая из транзакций останавливается в ожидании, пока вторая освободит нужный ей ресурс. В результате ни одна из них не может завершить свою работу. Если извне не поспособствовать одной из них или если одна из транзакций не завершит свою работу, то эта ситуация может продолжаться до бесконечности.



В главе 51 будут приведены примеры ошибок транзакций, уровней изоляции и блокировок в SQL Server.

Уровни изоляции

На физическом уровне любое ядро базы данных, допускающее логические транзакции, должно содержать механизмы изоляции. Для контроля, какие ошибки транзакций допустимы, существуют *уровни изоляции*, т.е. образно говоря, высота "забора" между ними. В спецификации ANSI SQL-92 определены четыре уровня изоляции:

- `Read uncommited`
- `Read commited`
- `Repeatable read`;
- `Serializable`.

В дополнение к этому списку компания Microsoft в версии SQL Server 2005 добавила уровень изоляции `Snapshot`. В сущности, этот уровень создает виртуальную копию, или мгновенный снимок, данных первой транзакции, в результате чего остальные транзакции на нее не влияют. Этот метод может привести к потерянным обновлениям.

Пустые значения

Реляционные базы данных всегда представляют отсутствие данных с помощью специального *пустого значения* `null`. На самом деле `null` представляет три совершенно различных сценария отсутствия данных.

- Столбец не задействован в данной строке. Например, если человек не устроен на работу, значит, какое-либо определенное значение в столбце даты приема на работу будет некорректным.
- Данные еще не введены, но вскоре это будет сделано. Например, если контактное лицо имеет имя и номер телефона, а адрес будет введен в процессе приема заказа.
- Столбец в данной строке вообще не содержит значения. Например, столбец комментария заполняется не для всех сотрудников, представленных в списке.

В зависимости от типа отсутствующих данных некоторые проектировщики баз данных по умолчанию подставляют нули, пустые строки и т.п. Однако такие значения могут негативно отразиться на результатах некоторых запросов и привести к проблемам поддержания целостности данных.

Возможность существования в столбце пустого значения, как правило, определяется в момент его создания. Однако примите к сведению, что по умолчанию SQL Server не допускает пустых значений, в то время как стандарт ANSI — допускает.

Работа с пустыми значениями

В связи с тем что пустое значение подразумевает неизвестность такового, результат любого выражения, включающего в себя значение `null`, также не может быть известен. Например, если состояние банковского счета неизвестно, а он включен в сумму общих доходов, то эта сумма также будет неизвестна. Та же концепция внедрена и в SQL Server. Как сказал известный разработчик баз данных Фил Сенн: "Пустое значение прекращает жизнь любого другого". Например, результатом выражения `SELECT 1+NULL` будет `NULL`.

Еще одним следствием неопределенности пустого значения является то, что одно значение `NULL` не может быть равно другому. По этой причине для проверки наличия пустого значения в языке SQL введена специальное выражение `IS NULL`.

Однако такой стандартный режим работы может быть нарушен. SQL Server будет игнорировать присутствие пустого значения в выражении, если установить для параметра `concat_null_yelds_null` значение `off`. Другое выражение, `ANSI nulls`, управляет тем, будет ли одно пустое значение считаться равным другому.

Противоречивость пустого значения

Большая часть разработчиков баз данных разрешает помещать в столбцы пустые значения, когда это имеет смысл.

Крайние апологеты "чистоты" баз данных не одобряют пустые значения и в любой создаваемой базе данных запрещают их. Еще один метод предполагает вынесение всех столбцов, принимающих пустые значения, в отдельную таблицу. Этот метод всего лишь заменяет столбцы, принимающие значения `NULL`, строками, что требует введения левого внешнего

объединения для проверки наличия или отсутствия значения. Сложность такого подхода не только снижает производительность, но и отрицательно сказывается на целостности данных. Вместо извлечения данных с пустым столбцом разработчику нужно дополнительно проверить наличие левого внешнего объединения со специализированной таблицей. Такой подход является свидетельством недостаточного понимания принципов единства хранилища данных, целостности данных, их полезности и излишних затрат ресурсов компьютера.

Производительность

Предоставление потребителю готовой к использованию информации является ключевым аспектом принципов информационной архитектуры. Несмотря на то что современные базы данных достигли высокого уровня производительности, возможность распространения ее на сверхбольшие базы данных с громадным количеством подключений остается предметом конкуренции между основными производителями СУБД.

Так как производительность физического диска является самым узким местом, ключевым вопросом становится уменьшение количества чтений или записей физических страниц, необходимых для выполнения задачи. Этому могут способствовать пять основных факторов.

Модель

Схема базы данных может сильно повлиять на производительность. Физическая модель должна принимать в расчет пути выполнения запросов. При очень сложной модели и наличии множества таблиц для извлечения данных потребуется создавать дополнительные объединения, а при обновлении или вставке данных выполнять большее число операций записи.

Некоторые схемы баз данных просто обескураживают своей схемой, в которой при пакетной обработке данных информация должна перебрасываться из одной области в другую. Если в программе серверной стороны присутствует несколько курсоров, то, скорее всего, это недосмотр не программиста, а проектировщика базы данных.

Пакетная обработка

Реляционная алгебра, язык SQL и механизмы реляционных баз данных оптимизированы для работы с пакетами данных. Плохо написанная программа, в которой курсор передвигается от строки к строке, или плохо написанный запрос SQL — главные источники низкой производительности.



Написание хороших пакетных запросов требует творческого подхода к созданию объединений и подзапросов. Об этом мы поговорим в главах 9 и 10.

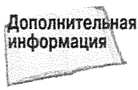
Хорошо написанный пакетный запрос способен выполнить всю операцию, считывая все затребованные страницы только один раз, в то время как решение, основанное на курсорах, будет считывать каждую строку отдельно. При тестировании простая операция обновления с помощью курсора выполняется в 70 раз дольше, чем с использованием пакетных запросов.

Индексация

Индексы являются своеобразным мостом между запросом и данными. Они повышают производительность за счет сокращения числа считывания страниц, на которых размещена таблица.

- Группа строк таблицы с кластеризованным индексом может быть считана за одно (или несколько) обращение к физическим страницам; при этом не приходится постоянно перескакивать по разным страницам.
- Индексы позволяют оптимизатору запросов напрямую переходить к конкретным строкам данных, а не сканировать для этого всю таблицу. Это аналогично тому, как предметный указатель книги помогает найти нужную страницу. Как только нужная строка найдена, оптимизатор выполняет операцию поиска закладки для перехода к конкретной странице.

Индексация является ключевым моментом физической схемы. Ее следует рассматривать скорее как искусство, нежели как науку. В то же время понимание работы оптимизатора запросов базы данных и структуры индексов в сочетании со знанием схемы конкретной базы данных и механизма доступа запросов к данным поможет спроектировать индексы более точно.

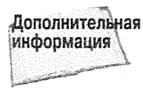


Стратегии настройки индексов будут подробно рассмотрены в главе 50.

Однако индексы имеют и отрицательную сторону. Несмотря на то что они помогают осуществлять поиск по таблице, они могут негативно влиять на скорость записи. Когда вставляется или обновляется строка, индексам может потребоваться провести и синхронизацию с данными. Таким образом, когда таблица имеет множество индексов, запись в таблицу будет замедляться. Другими словами, существует разногласие между созданием индексов для чтения и записью. Так как операции обновления и удаления должны найти измененные строки, операции записи дадут преимущества при экономной и обдуманной индексации. Различие в требованиях к индексации в операциях чтения и записи являются главным отличием транзакционной базы данных от баз данных, созданных для отчетности или хранилищ данных.

Разделы

Разбиение на разделы, или, другими словами, распределение данных по различным приводам дисков, — это метод повышения производительности особо больших баз данных (VLD).



Особенности операций разбития на разделы описаны в главе 53.

Кэширование

Кэширование представляет собой механизм предварительного извлечения пакетов данных с жесткого диска, чтобы в момент выполнения операций над базой данных они находились в памяти. В то время как кэширование является задачей ядра базы данных, выделение для этих операций достаточно большого объема оперативной памяти — задача администратора.

Доступность

Доступность информации подразумевает возможность пользователя в необходимом месте и в нужный момент времени получить доступ к запрошенным данным для их последующего анализа. На доступность влияют такие факторы, как возможность восстановления данных, их избыточность, архивирование и свойства сети.

Требования к доступности часто определяют в терминах "девяток". Например, термин "пять девяток" означает, что система доступна 99,999% необходимого времени (табл. 1.3).

Таблица 1.3. Таблица девяток

Проценты	Доступность	Недоступность	Описание
99,99999	364 дня 23 часа 59 минут 56 секунд	3 секунды	7 "девяток"
99,9999	364 дня 23 часа 59 минут 29 секунд	31 секунда	6 "девяток"
99,999	364 дня 23 часа 54 минуты 45 секунд	5 минут 15 секунд	5 "девяток"
99,99	364 дня 23 часа 7 минут 27 секунд	52 минуты 33 секунды	4 "девятки"
99,95	364 дня 19 часов 37 минут 12 секунд	4 часа 22 минуты 48 секунд	—
99,9	364 дня 15 часов 14 минут 24 секунды	8 часов 45 минут 36 секунд	3 "девятки"
99,8	364 дня 6 часов 28 минут 48 секунд	17 часов 31 минута 12 секунд	—
99,72603	364 дня ровно	1 день	Ровно 1 день
99,5	363 дня 4 часа 12 минут 00 секунд	1 день 19 часов 48 минут 00 секунд	—
99	361 день 8 часов 24 минуты 00 секунд	3 дня 15 часов 36 минут 00 секунд	2 "девятки"
98	357 дней 16 часов 48 минут 00 секунд	7 дней 7 часов 12 минут 00 секунд	—
97	354 дня 1 час 12 минут 00 секунд	10 дней 22 часа 48 минут 00 секунд	—



В главах 36 и 52 подробно изложены средства обеспечения доступности, предлагаемые в SQL Server 2005.

Избыточность

Избыточностью называют идентификацию возможных точек сбоя и устранение или максимальное уменьшение их отрицательного эффекта с помощью реализации альтернативного решения. В некоторых областях избыточность предполагает излишнюю трату ресурсов, однако в области хранения данных избыточность даже рекомендуется.

Сервер горячей замены — это копия базы данных на вторичном сервере, готовая к работе в случае необходимости. Как правило, этот метод для перемещения последних транзакций на резервный сервер использует технологию доставки журнала. *Кластеризованным сервером* называют дорогостоящее аппаратное соединение между двумя серверами, при котором при отказе одного из них в работу немедленно включается второй.

В некоторых случаях и самая надежная аппаратура отказывает. Если информация критична для организации, то руководство может посчитать, что потери времени, связанные с восстановлением резервной копии базы данных и журнала транзакций в случае сбоя, обойдутся ей слишком дорого. В этом случае лучшим решением будет создание сервера горячей замены, чтобы в случае выхода из строя основного сервера резервный стал немедленно доступен пользователям.

Восстановление

Для обеспечения доступности последним по эффективности методом является восстановление базы данных из файлов резервной копии. Так как в архив можно добавлять и журнал транзакций, вы достаточно оперативно сможете восстановить базу данных в состояние, предшествовавшее сбою компьютера.

Масштабируемость

Принципы информационной архитектуры гласят, что информация должна быть доступна как сегодня, так и в будущем, а это требует *масштабируемости* хранилища данных и способности адаптироваться к новым требованиям. Интеграция данных, производительность и доступность являются концепциями, которые легко понять, а вот масштабируемость — понятие более сложное.

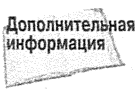
Для создания масштабируемых систем можно применить две концепции проектирования: распределение базы данных с использованием некоторого уровня абстракции и обобщение сущностей, где это возможно. Масштабируемость также тесно связана с простотой системы. Как говорится, сложность порождает сложность, что противодействует адаптации.

Уровень абстракции

Множество производственных баз данных проектируется на совесть и отлично функционирует продолжительное время. Однако как только возникает необходимость внесения изменений, трудность их внедрения с каждым следующим разом возрастает. В результате через несколько лет команда разработчиков может столкнуться с ситуацией, когда база данных уже перестала отвечать своему назначению, но ее чрезвычайно сложно изменить, поскольку придется потратить *слишком* большие средства на ее реанимацию.

Источником проблем здесь является не сама модель базы данных, а отсутствие взаимосвязи и взаимодействия между приложением и базой данных. *Уровень абстракции* — это первое средство отделения хранилища данных от приложения. При отсутствии уровня абстракции сложно логически разделить базу данных и приложение, поскольку даже малейшее изменение в базе приводит к переработке огромного объема программного кода.

Уровень абстракции хранилища данных — это набор интерфейсов программного обеспечения, которые служат своеобразным шлюзом, через который осуществляется обращение к базе данных. В сущности, уровень абстракции — это соглашение между программным кодом и базой данных, в котором определены все обращения к базе и их параметры. Уровень абстракции можно создать с помощью хранимых процедур на языке T-SQL или сторонних программ. Когда уровень абстракции существует и задействован, любое прямое обращение к таблицам посредством команд SQL DML полностью блокируется.



Разработка и программирование уровня абстракции на языке T-SQL описано в главе 25.

Уровень абстракции поддерживает масштабируемость, защищая программный код от изменений схемы, и наоборот. Реализация любой стороны контракта (хранилища данных или кода приложения) может быть изменена без влияния на уровень абстракции контракта или другой стороны контракта.

Обобщение

Обобщением называют группировку однотипных сущностей в единую сущность или таблицу так, чтобы эта единая таблица выполняла больше работы. База данных становится более гибкой, потому что обобщение сущностей с большей вероятностью позволит хранить новую аналогичную сущность.

Идея обобщения обычно реализуется тогда, когда не существует ни одной корректной нормализованной схемы. При возникновении любой проблемы базы данных возможно несколько

корректных решений. Часто различия между этими решениями идентифицируют пространство источников проблемы. Одно представление может идентифицировать несколько конкретных сущностей, которые технически отличаются, но подобны друг другу. Более общее представление может слить эти схожие сущности, получая при этом более простую и компактную схему.



Я не переоцениваю важность обобщения. Даже больше, чем нормализация, эта техника зависит от простоты модели и гибкости схемы. В последней схеме базы данных, которую мне приходилось анализировать, содержалось около 80 таблиц. С помощью обобщения, комбинируя по пять таблиц в одну сущность, мне удалось сократить схему до 17 таблиц, в результате чего она стала более мощной и гибкой, чем оригинал. Когда разработчик хвастается, что создал базу данных с великим множеством таблиц, я обычно считаю, что он создал нежизнеспособную схему.

Безопасность

Последним главным принципом организации информационной архитектуры является безопасность. В среде любой организации уровень безопасности должен регулироваться уровнем ее значимости и конфиденциальности. В программном обеспечении все начинается с организации системы безопасности центра данных и операционной системы. Безопасность информации включает три дополнительных компонента: ограничение доступа к конкретным данным с помощью ядра базы данных, идентификацию владельца информации, а также подтверждение достоверности данных с помощью идентификации их источника (в том числе это относится и к обновлениям).

Ограничение доступа

Любая корпоративная база данных содержит систему ограничения доступа к данным отдельных лиц и их групп. Гранулярность таких правил может варьироваться от обычного ограничения доступа до определения привилегий на создание, извлечение, обновление и удаление объектов.



Система безопасности SQL Server 2005, которая может быть достаточно сложной, описана в главе 40.

Информация о владельцах

Владельцем данных считается лицо или организация, которым необходимы эти данные. Это лицо обычно не является человеком или процессом, который ввел или создал данные. Владелец — это лицо, которое обслуживает эти данные. В некоторых системах одно лицо может быть идентифицировано владельцем целого приложения, в других каждая таблица или даже строка может идентифицировать своего владельца. Например, все данные в складской системе могут принадлежать одному вице-президенту по снабжению, в то время как информация о клиентах — подразделению, занимающемуся оформлением контрактов. Обеспечение собственности на уровне строки реализуется в схеме с помощью добавления столбца `DataOwner`.



Совместимость с законом Сарбанеса–Оксли требует не только защиты данных, но и документирования любых изменений владельцев данных или ограничений доступа.

Журнал аудита

Ведение журнала аудита позволяет отследить источник и время любого изменения в данных. На минимальном уровне в журнал аудита заносится дата и время создания, а также дата и время последнего обновления. На более высоком уровне в журнал аудита дополнительно заносится имя пользователя или процесса, который создает или изменяет данные. Полный аудит позволяет фиксировать любую историческую информацию. Способность воссоздавать данные в любой конкретный момент времени очень важен для временных, а также хронологических данных.



В главе 24 мы рассмотрим методы создания журналов аудита.

Теория оптимизации и SQL Server

Спросите двадцать администраторов баз данных об их любимых методах и стратегиях оптимизации, и они вам расскажут об индексации, добавлении дополнительной памяти и многом другом. Можно ли упорядочить массу разнообразных подходов к оптимизации? Скорее всего, нет. Можно ли сложить некоторую мозаику из разнообразных стратегий, проверяя при этом, как одна из них соотносится с другой?

Первое, что можно сказать, — не стоит применять все известные стратегии, потому что многие из них имеют внутренние и обойденные вниманием зависимости. Например, использование индексации для повышения скорости извлечения данных увеличит время выполнения транзакций, что связано с конкуренцией. Итак, мы видим непосредственную связь между индексацией и конкуренцией. Можно найти гораздо больше пересечений. Одни стратегии повышения производительности можно применять непосредственно, а эффективность других напрямую зависит от того, были ли перед этим применены другие.

Теория оптимизации исследует эти зависимости и предлагает среду планирования и разработки оптимизированного хранилища данных (рис. 1.3). Каждая из этих стратегий имеет несколько конкретных методик и активизируется с помощью поддерживающей ее стратегии, но ни одна стратегия не способна преодолеть недостатков поддерживающих ее стратегий.

Модель схемы

Моей главной стратегией повышения производительности является тщательная разработка схемы. Хорошая схема позволит применять пакетные запросы и облегчит планирование индексов.

Чтобы создать эффективную схему, следуйте следующим советам:

- избегайте чрезмерной сложности;
- тщательно выбирайте ключи;
- следите за необязательными данными;
- используйте некоторый уровень абстракции.

Я уверен, что плохая модель данных — главная причина плохой производительности; эти проблемы каскадом порождают другие. В результате базу данных можно считать "мертворожденной".

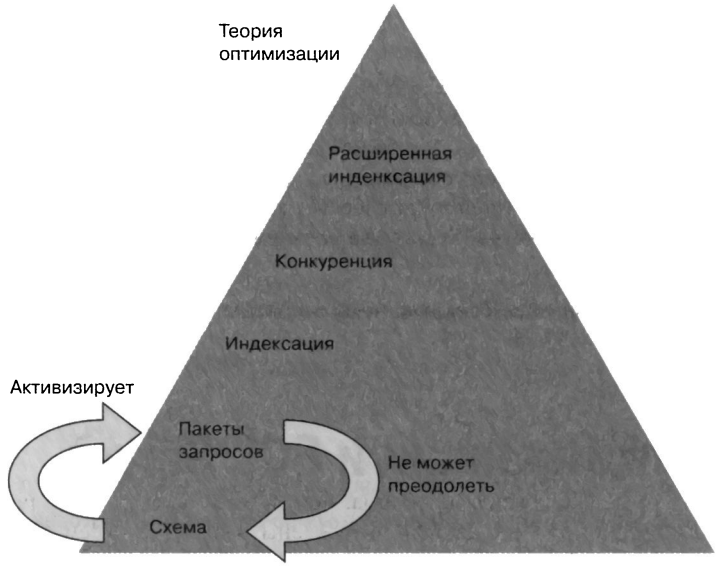


Рис. 1.3. Теория оптимизации гласит, что каждая стратегия зависит от другой и активизируется ею

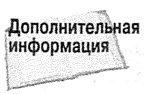
Запросы

SQL является пакетной системой запросов, и итеративные построчные операции на самом деле функционируют, как масса маленьких однострочных пакетов. Если итеративный метод реализован с помощью серверного курсора или циклов ADO, последовательный код будет довольно затратный. Моя вторая стратегия повышения производительности — использование пакетных решений. В то же время даже пакетная обработка не сможет преодолеть недостатки плохой или чересчур сложной модели схемы.

При принятии решения относительно использования последовательного кода или пакетных запросов ознакомьтесь с табл. 1.4 — она послужит хорошим руководством к действию.

Таблица 1.4. Методы программирования

Задача	Лучшее решение
Сложная бизнес-логика	Запросы, подзапросы, CTE
Динамическое генерирование DDL	Курсоры
Перестройка списка	Переменные или курсор
Перекрестная таблица	Запрос с предложением `PIVOT` или `CASE`
Прохождение по иерархии	Пользовательская функция или CTE
Накопительные суммы	Курсор



В главе 20 вы узнаете, как существенно повысить производительность, преобразуя сложные логические курсоры в пакетные запросы.

Индексация

Индексация в вопросах производительности является мостом между запросами и данными и одновременно ключевой стратегией повышения производительности. Стратегия, используемая в кластеризованных индексах для сокращения случайного поиска и группировки строк в одну страницу данных, использует некластеризованные индексы для обеспечения работы запросов и активизации поиска и при этом избегает излишних индексов, что может существенно ускорить выполнение пакетов. Однако даже хорошие индексы не смогут спасти от немасштабируемых программ.



Создание кластеризованных, некластеризованных и прочих индексов детально описано в главе 50.

Конкуренция

Блокировка — это более сложная проблема, чем думают большинство разработчиков, и большинство администраторов баз данных вынуждены решать ее путем понижения уровня изоляции транзакций до `nolock`, что очень опасно.

Конкуренцию можно сравнить с источником воды. Если люди, вальяжно прогуливаясь, ожидают своей очереди наполнить стаканчики, ситуация стабильна. Установка параметра `nolock` равносильна призыву: "Все бросайтесь к источнику!" Естественно, в этом случае ситуация выйдет из-под контроля и давка неизбежна. В такой ситуации лучше последовательно наливать воду в стаканчики, при этом сокращая время своего пребывания у источника. Аналогичная ситуация и в базах данных: лучше ограничиться небольшими запросами или сократить время выполнения транзакций. А для создания эффективных транзакций нужно разработать эффективную схему, использовать пакетный код и индексы.

После того как схема, запросы и индексы сократили время выполнения транзакций, оставьте в логических транзакциях только действительно необходимую логику и будьте осторожны с логикой в триггерах, так как она все равно будет задействована в транзакциях. В то же время сокращение блокированных ресурсов не сможет предотвратить излишнее сканирование таблиц.

Расширенная масштабируемость

Если схема, запросы, индексы и транзакции работают на славу, вы можете воспользоваться улучшенными средствами масштабируемости SQL Server:

- изоляцией мгновенных снимков;
- разделением таблиц;
- индексированными представлениями;
- брокером служб.

Вам кажется, что вы добились хорошей производительности без использования этих технологий? Возможно. Однако наилучших результатов вы добьетесь, используя технологии каждого из этих компонентов.



Теория оптимизации, в которой раскрыты зависимости между разными технологиями оптимизации, является революционной концепцией. Самую свежую информацию о теории оптимизации, а также мои презентации решений повышения производительности вы найдете на сайте `www.SQLServerBible.com`.

Резюме

Принципы информационной архитектуры являются основой разработки баз данных. Эти принципы раскрывают семь взаимозависимых правил организации хранилищ данных: простота, полезность, целостность данных, производительность, доступность, масштабируемость и безопасность. Каждое из этих правил исключительно важно при проектировании любой базы данных.

Итак, в части I книги продолжается изложение основных концепций баз данных. В следующей главе мы рассмотрим структуру реляционных баз данных.

Моделирование реляционных баз данных

ГЛАВА

2

В этой главе...

Видимые сущности

Идентификация множества сущностей

Реляционные шаблоны

Нормализация

Реляционная алгебра

Во время своего последнего путешествия я читал книгу Одри Ниффенеггер *Жена путешественника во времени*. Не вдаваясь в детали, скажу, что это прекрасная научно-фантастическая романтическая история. Женщина перемещалась по времени по соглашению, в то время как ее муж бесконтрольно бороздил пространственные и временные просторы, возвращаясь к ключевым моментам своей жизни снова и снова. Несмотря на то что сюжет в данном случае более сложный, чем в среднестатистическом романе, мне понравилось, как автор сплела все детали в бурлящий поток. Каждая деталь подчеркивала черты героев и постепенно выстраивала сюжет.

В некотором роде база данных напоминает хороший роман. Фабулой романа является модель базы данных, а данные представляют собой персонажей и их действия. *Нормализация* — это грамматика и структура базы данных.

Когда два писателя рассказывают одну и ту же историю, каждый представляет ее по-своему. Аналогично, нет единственно правильного способа моделирования базы данных, — если база данных содержит всю информацию, необходимую для извлечения рассказа, и следует нормализованным правилам грамматики, то она будет работать. Вывод здесь следующий: как некоторые книги лучше других, так и с некоторыми базами данных работать легче, в то время как запросы к другим вызывают массу сложностей.

Как и при написании романа, основой моделирования данных является внимательное исследование и осознание действительности. Основываясь на своих наблюдениях, проектировщик создает логическую систему — новый виртуальный мир, который моделирует образ реальности. Таким образом, то, насколько ясно проектировщик представляет себе реальность и идентифицирует сущности и их взаимодействие, настолько хорошей будет модель виртуального мира. Как и в постмодернизме, не существует единого безукоризненного и правильного представления — только точка зрения автора или разработчика.

Роль проектировщика баз данных является главной в любом программном проекте, так как он закладывает основы, точнее, структуру виртуальной реальности, которая либо поддержит программный код, либо похоронит его заживо. Любая функция, работающая с данными, должна быть предусмотрена в создаваемой схеме данных.

Жизненный цикл схемы данных обычно переживает несколько поколений кода приложения, и это еще одна причина, почему схема данных столь критична для жизнеспособности проекта. Данные продолжают жить, в то время как приложения приходят и уходят. Если разработчики приложений обычно рассчитывают на их эксплуатацию в течение двух-трех лет, базы данных продолжают существовать десятилетиями.

С точки зрения метафоры автора и проектировщика, эта глава раскроет перед вами основные структуры реляционной базы данных: сущности, нормализацию и реляционные модели.

Моделирование реальности

С точки зрения принципов информационной архитектуры (см. главу 1) целью моделирования данных является удовлетворение потребностей заказчика за счет определения такой структуры данных, которая логически представляет конкретные объекты и события.

Основным элементом данных является их элементарный контейнер. Большинство людей представляют себе этот контейнер как ячейку в электронной таблице — пересечение экземпляра сущности (строки) и его атрибута (столбца). Моделирование данных является искусством помещения этого элементарного контейнера в правильное место среди миллионов ячеек, формирующих всю систему данных. Чтобы моделировать данные, нужно рассматривать каждую ситуацию, сцену и документ с точки зрения элементов данных и целостности реляционной модели.

Моделирование данных включает в себя несколько процессов. Разработку модели данных нельзя рассматривать как обычный последовательный процесс, поскольку каждый шаг в нем зависит от множества других, и открытия ждут проектировщика на протяжении всего процесса. В то же время любой проектировщик обязательно проходит через следующие этапы.

1. Исследования и сбор требований.
2. Логическое представление реальности.
3. Идентификация и проектирование видимых сущностей.
4. Разработка схемы.
5. Проектирование структуры приложения.

Ни одна схема не может совершенно моделировать реальность. Каждая схема разрабатывается для того, чтобы удовлетворить требованиям проекта на данной стадии его жизненного цикла. Умение отсеивать только реальные требования, поддержание баланса между реальностью и моделью, а также умение создавать схемы данных, которые поддерживают текущую версию и оставляют место для роста, являются не столько результатом чтения книг, сколько результатом опыта.

Целью логического проектирования является описание сущностей, их взаимосвязей и общих правил существования. При построении физической схемы за основу берется логическая, а затем она встраивается в структуру конкретной системы управления базами данных (СУБД).

Логическая и физическая модели

Я слышал много дискуссий на тему различий между логической и физической моделями базы данных. В некоторых организациях эти задачи выполняются различными командами разработчиков, которые порой даже не контактируют друг с другом. Некоторые проектировщики

настаивают на том, что их реализация в точности соответствует логической модели. В противоположность этому есть те, которые всегда строят сразу физическую схему.

Целью логического моделирования является документирование сущностей и их взаимосвязей. Здесь главное хорошо понять основные бизнес-правила, и ничего больше. На основе этого понимания команда проектировщиков может построить физическую схему, которая отвечает как бизнес-правилам, так и требованиям производительности.

В главе 17 мы сконцентрируем внимание на физической модели.

Математический подход к моделированию реляционных баз данных включает в себя некоторые правила нормализации — правила, которые проверяют правильность модели базы данных. Разработав за свою жизнь сотни нормализованных моделей баз данных, я нашел эффективный подход к этому процессу, основанный на видимости и поддержке сущностей. *Видимая сущность* представляет собой элемент, который может распознать пользователь. *Поддерживаемой* называют абстрактную сущность, разработанную проектировщиком базы данных для физической поддержки общей логической модели.

Видимые сущности

Видимые сущности обычно представляют объекты, которые могут распознать люди. Многие объекты являются именами собственными, такими как люди, города или местности, а некоторые — нарицательными, такими как контакты или марки автомобилей. Видимые сущности могут представлять также действия, такие как сбор материала или упорядочивание списка. Обычно видимый объект уже представлен где-либо в документе.

Некоторые видимые объекты представляют собой методы, с помощью которых группируются или организуются некоторые другие первичные объекты. Например, покупатели понимают, что продукты делятся на множество категорий: молочные, мясные, хлебо-булочные и т.п. В качестве еще одного примера можно привести заказы на поставки, которые могут иметь несколько категорий срочности.

Каждая строка — это остров

Каждая строка должна представлять некоторый полный логический объект, а каждый атрибут в ней должен относиться именно к этому объекту.

Каждая сущность основана предполагает, что каждая строка представляет некоторый объект или действие. На этой посылке и основано моделирование данных. Ключевым моментом является идентификация этих уникальных и общих объектов и действий. Именно это подразумевает термин *система реляционной базы данных*. Это не значит, что одни сущности связаны с другими каким-то образом — это значит, что все однородные или связанные данные сведены в одну сущность.

Только данные, описывающие объекты, принадлежат этой сущности. Некоторые из возможных атрибутов, даже если они перечислены на листе бумаги для одного объекта, могут на самом деле описывать и другой объект. Стандартная форма заказа обычно включает в себя имя заказчика и его адрес, хотя это имя принадлежит именно заказчику, а не самому заказу. Адрес тоже принадлежит заказчику. В то же время, если в следующем году заказчик сменит свой адрес, в заказе останется старый адрес. Вывод следующий: для моделирования данных критичны основательный анализ объектов и их свойств.

Хорошо спроектированные сущности имеют массу однотипных элементов. Например, одна сущность овощной лавки может содержать массу товаров, вместо того, чтобы яблоко представляло собой одну сущность, арбуз — другую и т.д.

Первичные ключи

Каждая строка в сущности представляет собой единичный уникальный объект в реальном мире. Так же как не могут существовать два идентичных самолета, две строки не могут представлять один и тот же самолет. Для логической гарантии уникальности каждой из строк один из атрибутов назначается первичным ключом — именно по нему обращаются к строкам таблицы команды модификации данных. Если вы обеспечили, чтобы каждому объекту в реальности соответствовал всего один первичный ключ и наоборот, значит, вы сделали свою работу хорошо.

Чтобы привести пример первичного ключа в физической схеме, предположим, что в базе данных компании каждый клиент однозначно определяется своим идентификатором. С помощью специального столбца идентичности SQL Server для строки каждого нового клиента, когда она вставляется в таблицу, автоматически генерирует новое целое число.

Таблицы, строки и столбцы

Реляционные базы данных собирают однообразные или связанные данные в единый список. Например, вся информация о товарах может быть перечислена в одной таблице, а о клиентах — в другой.

Таблицы базы данных в чем-то сходны с электронными таблицами и также состоят из строк и столбцов. Электронные таблицы привлекательны своим стилем обработки информации, облегчающим их добавление и модификацию. На практике менеджеры склонны хранить критичную информацию в электронных таблицах, и многие базы данных берут свое начало именно из них.

Как в электронных таблицах, так и в таблицах баз данных каждая строка представляет некоторый элемент списка, а каждый столбец является некоторым фрагментом данных об этом элементе. В то время как электронные таблицы обычно имеют произвольную структуру, таблицы базы данных могут предъявлять строгие требования к данным в столбцах. Так как целостность строк и столбцов критична в таблице базы данных, то и сама конструкция таблицы критична.

За многие годы разные стили разработки обращались к столбцам таблиц с помощью разных терминов, которые перечислены в табл. 2.1.

Таблица 2.1. Сравнение терминов баз данных

Стиль разработки	Общий список	Элемент в списке	Элемент информации в списке
Электронная таблица	Электронная таблица, рабочий лист, именованный диапазон	Строка	Столбец, ячейка
Старое программное обеспечение	Файл	Запись	Поле
Реляционная алгебра, логическое проектирование	Сущность	Кортеж	Атрибут
SQL, физическое проектирование	Таблица	Строка	Столбец
Объектно-ориентированное проектирование	Класс	Экземпляр объекта	Свойство

Разработчики приложений SQL Server при обсуждении физической схемы обычно называют элементы базы данных таблицами, строками и столбцами, а иногда при обсуждении логической схемы используют термины *сущность*, *кортеж* и *атрибут*. В оставшейся части

книги мы будем использовать термины физической схемы, но поскольку эта глава посвящена теории проектирования, в ней мы будем пользоваться терминами реляционной алгебры (сущность, кортеж и атрибут).

Идентификация множества сущностей

Каждая из сущностей должна быть полной, т.е. содержать все атрибуты, зависящие от первичного ключа, в то же время сама по себе она не способна в достаточной мере отразить модель. Для моделирования любой бизнес-функции или организационной задачи требуется множество сущностей.

Дополнительные сущности отражают множество дополнительных фрагментов информации об основных объектах или их группах и объединяют их в единую систему. При разработке логической модели данных различные типы логических сценариев требуют наличия множества сущностей, в том числе следующих:

- множество объектов;
- отношения между объектами;
- организация или группировка объектов;
- целостность значений;
- сложные объекты.

Иногда разграничение между объектами, их значениями и группировкой нечетки. Однако при поддержке всех вышеперечисленных сущностей модель данных будет полной.

Множество объектов

Иногда оказывается, что один объект на самом деле является списком множества других объектов, как в приведенных ниже примерах.

- В базе данных туристического агентства один тур может упоминаться несколько раз, но каждый раз с разными датами.
- В базе данных семей каждый человек может иметь несколько детей.
- Табель рабочего времени сотрудника может содержать множество штампов времени. Этот табель рассматривается как один объект, однако при ближайшем рассмотрении он окажется списком временны*х* событий.
- Календарь сотрудника может содержать множество напоминаний.

Отношения между объектами

Наиболее распространенным назначением существования множества сущностей является описание некоторого типа отношений между двумя разными объектами. Соответствующие примеры приведены ниже.

- В базе данных турагентства клиенты участвуют в турах, а проводят их гиды. Существуют отношения между клиентами и турами, а также между гидами и турами.
- Материал может создаваться из множества других материалов.
- Участковый врач может обслуживать несколько членов семьи; это отношение между ним и этими членами.

- В системе проверки качества программного обеспечения в одной функции может быть обнаружено множество ошибок.



Модель схемы должна предусматривать обеспечение наилучшей производительности для наиболее вероятного сценария, а также обработку исключений. Для примера предположим, что 96% клиентов имеют всего один адрес, 4% имеют еще один адрес, а всего один клиент имеет множество филиалов, разбросанных по всему миру. Добавление в таблицу первых 96% клиентов не потребует дополнительных объединений. Добавление дополнительных адресов потребует наличия дополнительной таблицы, объединенной отношением "один ко многим". Придется использовать флаг в таблице клиентов для индикации наличия многочисленных офисов, чтобы избежать проверки левым внешним объединением наличия дополнительных офисов у всех клиентов. (Объединения мы детально рассмотрим в главе 9.)

Этот аспект проектирования может возникнуть при разработке логической схемы; физическая схема также может потребовать его для своей оптимизации.

Организация и группировка объектов

Объекты порой группируются в разные категории. Эти категории могут быть перечислены в отдельной сущности, как описано ниже.

- Клиенты могут быть сгруппированы по странам.
- Материалы могут группироваться по своему состоянию (сырье, товары в производстве, готовая продукция).
- В базе данных турагентства туры могут группироваться по странам.

Целостность значений

Атрибуты кортежа иногда требуют обеспечения целостности домена, т.е. вхождения в предопределенный диапазон значений. Соответствующие примеры приведены ниже.

- Тип кредитной карточки, используемой для покупок.
- Регион, которому принадлежат адреса.
- Код подразделения некоторой организации может принимать определенные значения.

Сложные объекты

Некоторые объекты в реальной жизни настолько сложные, что моделировать их одной сущностью невозможно. Информация принимает больше форм, нежели могут содержать один первичный ключ и одна строка таблицы. Чаще всего это касается составных объектов реального мира. Например, в одном заказе может содержаться множество строк. Эти строки являются частью заказа, однако для их моделирования потребуется создание дополнительной сущности.

Реляционные шаблоны

После того как объекты и действия будут организованы, следует определить отношения между ними. Каждое отношение связывает две сущности с помощью их ключей и имеет два следующих основных атрибута.

- **Строгость** — количество объектов, которые могут существовать на каждом конце отношения.
- **Обязательность** — является ли отношение обязательным.

Клиенты или бизнес-аналитики должны быть способны описать отношения между объектами с помощью терминов *включает*, *имеет* или *содержит*. Один клиент может разместить множество заказов. Один заказ может включать в себя множество товаров. Один и тот же товар может содержаться во множестве заказов.

Вторичные сущности и внешние ключи

Когда два объекта связаны отношением, одна из сущностей, как правило, является главной, а вторая — подчиненной, или вторичной. Один объект в главной сущности может быть связан с многочисленными объектами во вторичной, как показано на рис. 2.1.

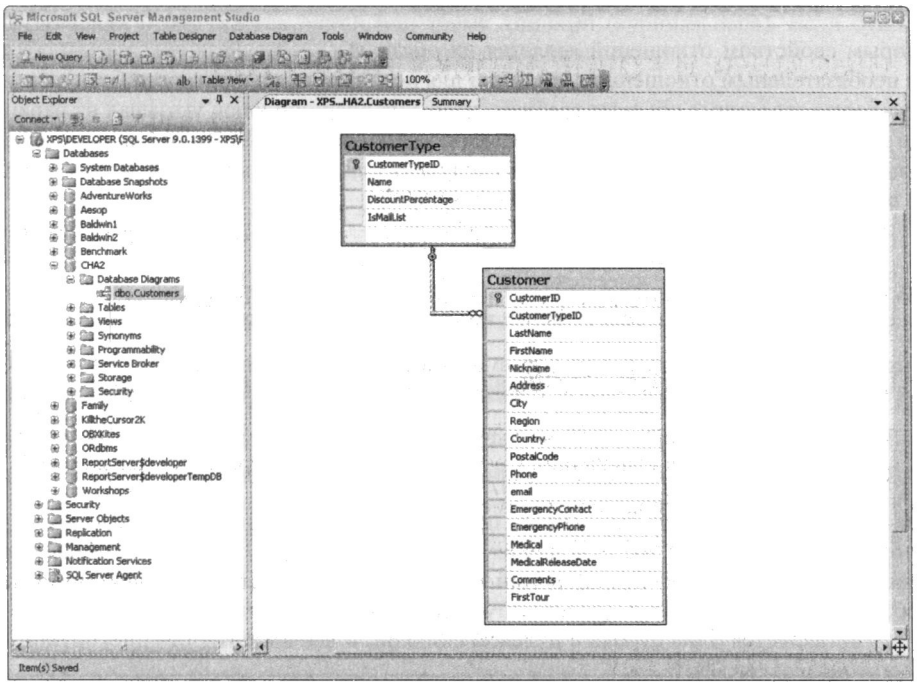


Рис. 2.1. Отношение "один ко многим" состоит из главной и вторичной сущностей. Внешний ключ вторичной сущности связан с первичным ключом главной

Назначение внешнего ключа — хранение значения первичного ключа главной сущности, чтобы можно было установить соответствие между сущностями.

Строгость отношения

Строгость отношения описывает количество элементов с каждой его стороны. Каждая из сторон отношения может допускать как один объект, так и их множество. Тип ключа определяет возможность множества объектов. Первичные ключи допускают наличие только одного объекта, в то время как внешние — их множества.

Существуют различные возможные варианты строгости; все они перечислены в табл. 2.2. В этом разделе мы подробно рассмотрим каждый из этих вариантов.

Таблица 2.2. Строгость отношений

Тип отношения	Ключ первой сущности	Ключ второй сущности
Один к одному	Первичный ключ главной сущности — один объект	Первичный ключ главной сущности — один объект
Один ко многим	Первичный ключ главной сущности — один объект	Внешний ключ вторичной сущности — множество объектов
Многие ко многим	Внешний ключ главной сущности — множество объектов	Внешний ключ вторичной сущности — множество объектов

Обязательность отношений

Вторым свойством отношений является их *обязательность*. Различие между обязательным и необязательным отношением критична для поддержания целостности базы данных.

Некоторые вторичные объекты требуют, чтобы их внешний ключ указывал на первичный. В этом случае вторичный объект будет неполным или бессмысленным без наличия связанного с ним первичного объекта. Например, в следующих ситуациях критично, чтобы отношение имело обязательный тип.

- Строка заказа без самого заказа бессмысленна.
- Заказ без заказчика не имеет смысла.
- В базе данных турагентства событие, не привязанное к некоторому туру, бессмысленно.

В противоположность этому некоторые отношения не обязательны. В этих случаях в реальном мире вторичный объект может существовать без первичного. Приведем два примера.

- Заказчик, не имеющий дисконтной карточки, все равно остается клиентом.
- Заказ может иметь и не иметь код приоритета — в любом случае он остается заказом.

Некоторые разработчики баз данных предпочитают избегать необязательных отношений, поэтому проектируют все отношения как обязательные и создают объекты, которым не нужен вторичный ключ, указывающий на суррогатный объект в главной таблице. Например, вместо того, чтобы разрешить пустые значения в столбце скидки, в отдельную сущность вставляется одна строка с нулевой скидкой, и все клиенты без дисконтной карты указывают на этот элемент.

Существуют две причины, чтобы избегать суррогатных пустых элементов. Во-первых, добавляется работа проектировщику базы данных (дополнительные вставки и проверки внешнего ключа). Во-вторых, гораздо легче работать без лишнего отношения при использовании предложения `WHERE` *`столбец`* `IS NOT NULL`. Пустые элементы являются полезным элементом общей модели, и игнорирование их достоинств только добавляет работы как разработчику, так и самой базе данных.

В некоторых редких ситуациях требуется поставить обязательность в зависимость от некоторого условия. В зависимости от правил отношение может устанавливаться в следующих случаях.

- Если организация иногда продает товары, изготовленные под заказ, которые отсутствуют в основной сущности. В этом случае обязательность отношения зависит от характера товара. Сущность строки заказа для товаров может использовать два атрибута.

Если используется атрибут идентификатор товара, то он должен указывать на корректный первичный ключ сущности товара.

- Если же используется атрибут заказного товара, то атрибут идентификатора товара остается пустым.

Реализация обязательности отношений зависит от конкретной физической схемы. Единственной целью логического проектирования является моделирование объектов организации, их отношений и бизнес-правил.

Диаграмма модели данных

Те, кто занимается моделированием данных для графического отображения создаваемых моделей, используют несколько методов. Популярным является ER-метод Чена, а система Visio Professional содержит еще пять других методов. Метод, который предпочитаю лично я, — самый простой. Он предполагает зарисовку схемы на доске, как показано на рис. 2.2. Строгость отношения отображается одной или тремя линиями (куриная лапа). Если отношение необязательное, то около внешнего ключа помещается окружность.

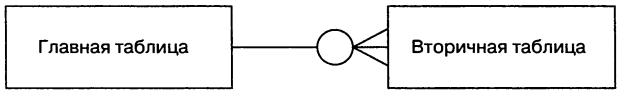


Рис. 2.2. Простой метод отображения на диаграммах логических схем

Еще одним преимуществом этого простого метода построения диаграмм является то, что он не требует установки расширенной версии программы Visio.

Отношения "один ко многим"

На сегодняшний день самым распространенным типом отношений является "один ко многим". Несколько элементов вторичной сущности связываются с одним элементом главной сущности. Отношение устанавливается между первичным ключом главной сущности и внешним ключом вторичной сущности. Приведем несколько примеров.

- В базе данных турагентства Cape Hatteras Adventures каждый базовый лагерь может иметь несколько туров, которые берут в нем начало. Каждый тур может начинаться только из одного базового лагеря; таким образом, один лагерь связан отношением "один ко многим" с несколькими турами. Отношение устанавливается между первичным ключом сущности `BaseCamp` (базовые лагеря) и внешним ключом сущности `Tour` (туры) (рис. 2.3). Каждый атрибут внешнего ключа таблицы `Tour` содержит копию первичного ключа таблицы `BaseCamp`.

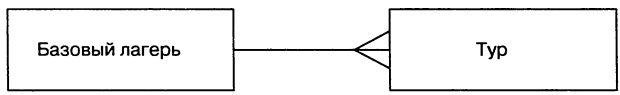


Рис. 2.3. Отношение "один ко многим" связывает первичный ключ с внешним ключом

- Каждый клиент может разместить множество заказов. Каждый заказ имеет свой уникальный первичный ключ `OrderID` (идентификатор заказа); сущность `Order` (заказы) также имеет вторичный ключ `CustomerID` (идентификатор клиента), указывающий

на клиента, разместившего заказ. Сущность `Order` может содержать несколько элементов, имеющих один и тот же идентификатор клиента, что указывает на наличие отношения "один ко многим".

- Благотворительная организация может получать ежегодные взносы своих спонсоров. Как только спонсор вносит ежегодный платеж, он заносится во вторую сущность, которая может хранить данные о взносах за многие годы. Структура этой сущности может быть следующей: `DonorName`, `2001pledge`, `2002pledge`, но возможны и другие варианты.
- Заказ может иметь несколько строк. Первичный ключ сущности `Order` дублируется во вторичном ключе сущности `OrderDetail`. Каждый заказ имеет один элемент в сущности `Order`, но может иметь множество ассоциированных элементов в сущности `OrderDetail`.

Отношение "один к одному"

Отношение "один к одному" связывает две сущности их первичными ключами. Так как первичный ключ должен быть уникальным, с каждой стороны отношения содержится всего один элемент.

Отношения "один к одному" иногда используют для расширения одной сущности за счет создания дополнительной со своими атрибутами. Например, сущность `Employee` (сотрудники) может хранить основную информацию о сотрудниках компании. В то же время более конфиденциальная информация может храниться в отдельной сущности. В этом случае защита может применяться на основе отдельных атрибутов; представление может отображать только избранные атрибуты. Многие организации предпочитают моделировать конфиденциальную информацию с помощью создания отношений "один к одному" между двумя или более сущностями.

Отношения между подтипом и супертипом

Элемент проектирования, использующий отношение "один к одному", на самом деле формирует отношение между супертипом и подтипом. Это отношение связывает одну сущность супертипа с несколькими сущностями подтипов, чтобы расширить состав атрибутов элемента. Сущность супертипа имеет необязательное отношение "один к одному" с каждой из сущностей подтипов.

Такая модель может пригодиться, когда несколько объектов имеют ряд идентичных атрибутов и отличаются только несколькими. К примеру, это может относиться к сущностям клиентов, поставщиков и производителей. Все три сущности имеют одинаковые атрибуты названия и адреса, но каждая из них имеет свой специфический состав остальных атрибутов. Например, только клиенты могут иметь дисконтные карты, и только поставщики имеют атрибуты, связанные с характеристиками договоров на поставку.

Несмотря на то что для поставщиков и клиентов можно использовать отдельные сущности, гораздо лучшее решение предполагает наличие единой сущности `Contact`, где будут храниться общие атрибуты и две отдельные сущности для атрибутов, уникальных для поставщиков и клиентов.

Если партнер является клиентом, то его дополнительная информация заносится в сущность `Customer`; если он поставщик — в сущность `Supplier`. При этом все три сущности (`Contact`, `Customer` и `Supplier`) имеют один и тот же первичный ключ. Один элемент в сущности `Contact` не обязательно может быть связан с одним элементом в сущности `Customer` и(или) с одним элементом в сущности `Supplier` (рис. 2.4).

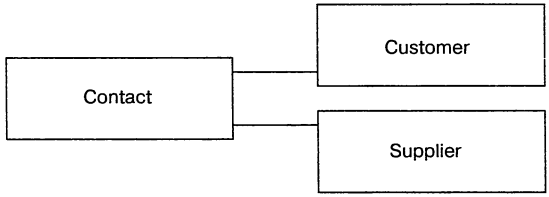
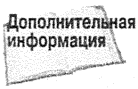


Рис. 2.4. Отношение "один к одному" связывает первичный ключ с другим первичным ключом

Большинство моделей "супертип–подтип" допускает наличие только одного элемента подтипа для каждого элемента супертипа. В то же время в приведенном примере информации о контактах одна и та же организация потенциально может быть одновременно и клиентом, и поставщиком, поэтому элементы могут добавляться одновременно в две сущности.



Модель "супертип–подтип" очень сходна с концепцией наследования классов в объектно-ориентированном анализе. Таким образом, если база данных содержит несколько моделей "супертип–подтип", то база данных может считаться кандидатом на объектно-реляционный стиль вместо реляционного. Объектно-реляционные СУБД для SQL Server можно загрузить с сайта `www.SQLServerBible.com`.

Следует обратить внимание на снижение производительности при использовании такой модели. Операция вставки предполагает добавление элементов в две сущности, а операция извлечения данных должна объединять сущности супертипа и подтипа. Поэтому старайтесь не использовать модели "супертип–подтип" для разделения элементов на категории. Эту модель следует применять только тогда, когда несколько столбцов уникальны для каждого подтипа. При этом при извлечении данных только из одного из подтипов снижается нагрузка на базу данных.

Отношение "многие ко многим"

В отношениях типа "многие ко многим" обе стороны могут быть связаны с множеством элементов противоположной стороны. Отношение "многие ко многим" часто встречается в реальной жизни. Приведу несколько примеров.

- Заказ может содержать несколько товаров, а каждый товар может находиться во множестве заказов.
- В базе данных турфирмы Cape Hatteras Adventures экскурсовод может быть квалифицирован для ведения нескольких туров, а каждый тур может иметь несколько квалифицированных экскурсоводов.
- В базе данных турфирмы клиент может принимать участие в нескольких турах, а каждый тур может иметь множество участников.

Возвращаясь к предыдущему примеру логического моделирования, мы видим, что отношение "многие ко многим" между клиентами и турами моделируется с помощью множественной связи с каждой стороны отношения (рис. 2.5). В данном случае отношение "многие ко многим" необязательно, поскольку как клиенты, так и туры могут существовать независимо друг от друга.

Отношения "один к одному" и "многие ко многим" могут проектироваться из объектов, которые пользователи способны описать и понять. В физической схеме отношение "многие ко многим" не может быть смоделировано с помощью только видимых объектов.

Для разрешения отношения "многие ко многим" используется ассоциированная таблица (рис. 2.6), иногда называемая *стыковочной*. Эта таблица искусственно создает два отношения "один ко многим" между двумя сущностями.

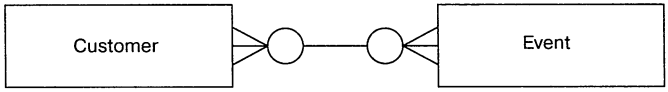


Рис. 2.5. Логическая модель "многие ко многим" содержит множество элементов с каждой из сторон отношения

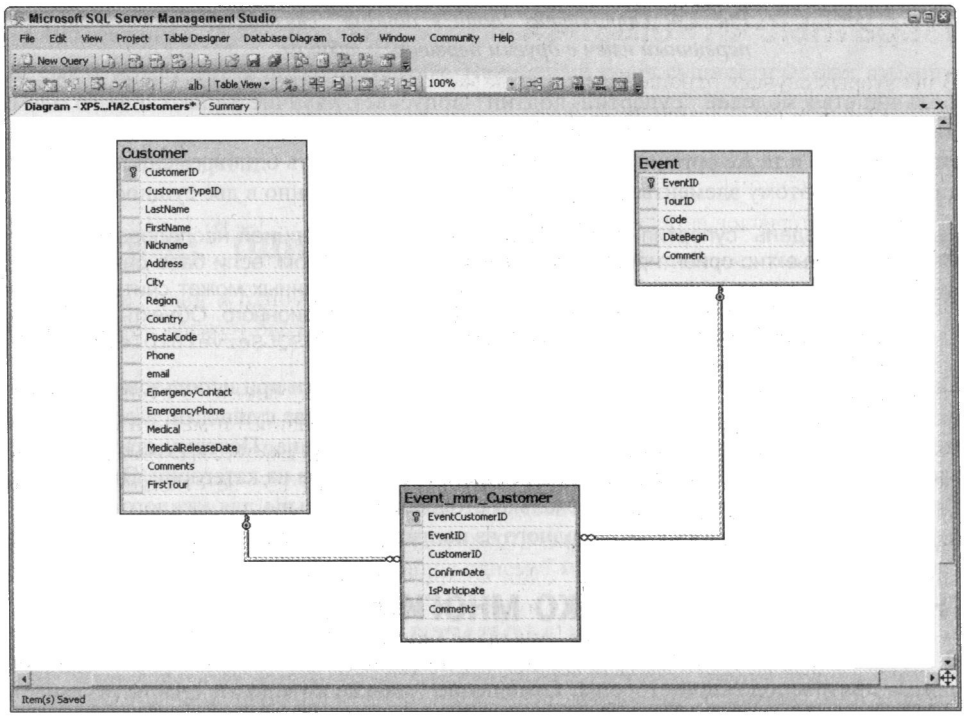


Рис. 2.6. Физическая модель отношения "многие ко многим" включает стыковочную таблицу, имеющую отношения "один ко многим" к обеим сущностям

В некоторых случаях дополнительная информация может описывать отношение "многие ко многим". Такая информация принадлежит к связывающей сущности. Например, в списке материалов отношение одного материала к другому может включать атрибут количества в связывающей сущности, описывающий, какое количество одного материала необходимо для создания второго.

Отношение между каждой основной сущностью и связывающей является обязательным, поскольку это отношение некорректно в отсутствие основного объекта. Если в базе данных турфирмы удалить либо клиента, либо тур, то соответствующий элемент в связывающей сущности станет некорректным.

Сущности категорий

Еще одним типом поддерживающей сущности является сущность *категорий*, иногда называемая *таблицей классификаторов*. Эти сущности обеспечивают целостность домена в смысле способа организации элементов. Отличным примером может служить таблица штатов США. Вместо того чтобы в таблице клиентов атрибут региона мог принимать некорректные

значения (например, `Fl` или `FLA` для штата Флорида), эта таблица связана с таблицей штатов, в которой все эти названия заведомо корректны. Такая целостность данных ускоряет и облегчает поиск и сортировку данных.

Видимые сущности обычно связаны с сущностями категорий отношением "один ко многим"; такое отношение может быть как обязательным, так и нет.

Возвратные отношения

В некоторых случаях отношение устанавливается между двумя элементами одного и того же типа. Приведем несколько примеров.

- Организационная диаграмма представляет сотрудника, отчитывающегося перед другим сотрудником.
- Список материалов детализирует, как один материал получается из других материалов.
- В базе данных семей некоторое лицо имеет отношение к своей матери и отцу.

Все это примеры *возвратных отношений*, чаще всего называемых *рекурсивными*, а иногда *унарными* или *собственными*. Благодаря форме на диаграмме их неформально называют отношениями *слоновьего уха*.

Используя базу данных семей в качестве примера, каждый элемент в сущности `Person` представляет одного человека. Каждый человек имеет (как правило) и мать, и отца, которые также находятся в сущности `Person`. Таким образом, внешние ключи `MotherID` (идентификатор матери) и `FatherID` (идентификатор отца) указывают соответственно на элементы матери и отца в той же сущности.

Так как первичным ключом является `PersonID` (идентификатор человека), а `MotherID` — внешним, устанавливается отношение "один ко многим" (рис. 2.7). Одна мать может иметь несколько детей, однако каждый ребенок имеет одну и только одну мать.

Более сложным примером является список материалов, так как каждый материал может быть создан на основе нескольких исходных и в то же время использоваться для изготовления ряда других материалов на следующей стадии производственного процесса. Это рекурсивное отношение "многие ко многим" проиллюстрировано на рис. 2.8.

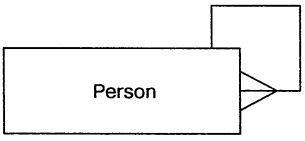


Рис. 2.7. Возвратное, или рекурсивное, отношение представляет собой отношение "один ко многим" в пределах одной сущности

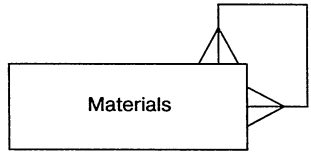


Рис. 2.8. Логическая схема возвратного отношения "многие ко многим" демонстрирует наличие нескольких элементов на каждом конце отношения

Для разрешения отношений "многие ко многим" требуется связывающая сущность. В примере таблицы материалов связывающая сущность `BillOfMaterials` имеет два внешних ключа, каждый из которых указывает на сущность `Material` (рис. 2.9).

Первый внешний ключ указывает на создаваемый материал, а второй — на исходный.

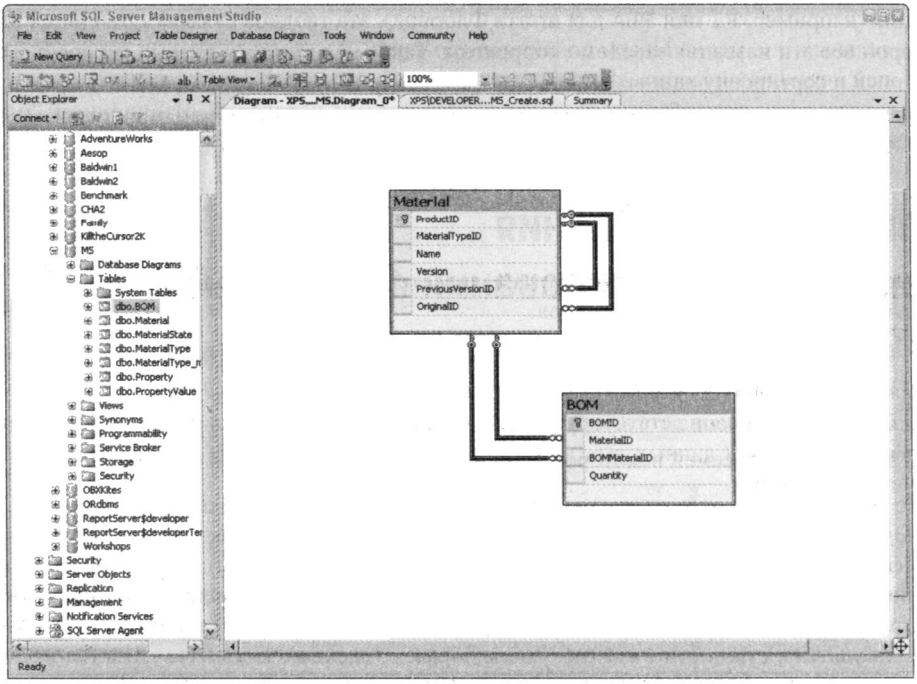


Рис. 2.9. Физическая схема базы данных возвратного отношения "многие ко многим" должна содержать связывающую сущность, как и в обычных отношениях

Нормализация

Самая известная концепция проектирования реляционных баз данных и их нормализации была разработана доктором Эдгаром Коддом. Целью *нормализации* является сокращение или полное устранение аномалий обновления, которые могут возникнуть, если одни и те же фактические данные хранятся в разных частях базы данных.

Нормализация не является ключевым этапом проектирования базы данных, а только средством сокращения аномалий обновления. Как таковая, она больше подходит для баз данных, которые содержат оперативные данные, такие как база данных текущих заказов. Соответственно, базы данных, не содержащие оперативные данные (такие, как база данных отчетов или хранилище данных), не обязательно выиграют от нормализации.

Когда индустрия переходила от модели, основанной на "плоских" файлах, к реляционным моделям данных, нормализацию можно было представить как процесс, начинающийся с "плоского" файла и проходящий постепенно через все нормальные формы, приобретая реляционный вид, в котором максимально сокращены аномалии обновления.

Хотя большинство разработчиков баз данных знакомы с нормализацией, реальные рабочие принципы нормализации зачастую понимаются неверно. Существуют три основных заблуждения относительно нормализации и логической схемы базы данных.

- Наиболее распространенное заблуждение представляет собой мнение о том, что чем больше таблиц, тем выше форма нормализации базы данных. Например, некоторые разработчики верят, что если конкретная схема базы данных в третьей форме нормализации содержит 12 таблиц, то в четвертой она уже будет содержать 16 таблиц.

- Еще одним заблуждением является рассмотрение нормальных форм как последовательных, т.е. мы начинаем с ненормализованной базы, затем корректируем ее для первой нормальной формы, затем для второй и т.д.
- И наконец, заблуждением является уверенность в том, что для каждой задачи существует единственная правильная нормализованная форма. Опытные проектировщики данных способны сразу увидеть несколько возможных конфигураций сущностей и моделей логической схемы.

Принципы проектирования сущностей и атрибутов

Ключевым моментом в проектировании схемы, которая свободна от аномалий обновления, является обеспечение того, чтобы каждый единичный факт из реальной жизни моделировался одной точкой данных в базе. Существуют три принципа, которые определяют единичную точку данных.

- Каждая сущность базы данных должна описывать одну "вещь".
- Между атрибутами и фактами должно существовать отношение "один к одному", т.е. каждый атрибут должен представлять один факт, и каждый факт должен быть представлен одним атрибутом. Например, "универсальный ключ", который шифрует несколько фактов в одном атрибуте, нарушает этот принцип; то же касается моделей, в которых для определения одного факта комбинируется два и более атрибута.
- Каждый атрибут должен описывать свою сущность и никакую другую, связанную с ней.

Нормальные формы

Нормализация обычно определяется в терминах нормальных форм. Каждая из нормальных форм описывает возможные ошибки в модели сущностей и атрибутов и формулирует правило, способное устранить эти ошибки. Таким образом, нормальные формы аналогичны тому, как правила грамматики способны выявить ошибки в структуре предложений.

При создании логической модели *нормализация* представляет собой математический метод оценки качества отношений в модели данных. В противоположность реляционной модели базы данных "плоский" файл или ненормализованные модели данных подвержены проблемам при обновлении данных, обычно выраженных дублированием информации. Каждая следующая форма нормализации устраняет одну из проблем, связанных с "плоскими" файлами.

Нормализованная модель базы данных имеет следующие преимущества перед ненормализованными:

- повышенная целостность данных делает невозможным двойное хранение одних и тех же данных;
- сокращение споров за блокировку ресурсов и улучшенная многопользовательская конкуренция;
- меньшие размеры файлов.

Модель данных не начинает с ненормализованного состояния и не обязательно затем проходит через все нормальные формы. Обычно проектировщик изначально разрабатывает логическую схему не менее чем в третьей нормальной форме и может принять решение привести часть схемы к более высокой форме нормализации.

Простота и нормализация

Простота не обязательно предполагает нарушение целостности данных. Формы нормализации так же связаны с проектированием базы данных, как правила грамматики с письмом. Хороший текст не должен нарушать правила грамматики. Продолжая мысль, следует сказать, что главный принцип, сформулированный в книге Странка и Уайта *Элементы стиля* (Будь точен!), не менее важен для проектирования баз данных, чем для письма.

Первая нормальная форма (1НФ)

Первая нормальная форма подразумевает, что данные объединены в сущности и удовлетворяют следующим условиям.

- **Каждая единица данных представлена в скалярном атрибуте**. Согласно Мерриаму–Вебстеру, *скалярным* называется значение, которое можно представить точкой на шкале.

  Каждый атрибут должен содержать одну единицу данных, и каждая единица данных должна заполнять один атрибут. Модели, в которых в одном атрибуте объединено несколько элементов информации, нарушают принципы первой нормальной формы. Аналогично. если несколько атрибутов должны быть скомбинированы, чтобы получить один элемент данных, значит, модель атрибута некорректна.

- **Все данные должны быть представлены уникальными атрибутами**. Каждый атрибут должен иметь уникальное имя и уникальное назначение. Сущность не должна содержать повторяющиеся атрибуты. Если некоторый атрибут повторяется или сущность охватывает чересчур широкую предметную область, значит, объект неправильно спроектирован.

  Модель, в которой атрибуты повторяются (например, сущность заказа, содержащая атрибуты `товар1`, `товар2` и `товар3` для хранения нескольких позиций заказа), нарушает первую нормальную форму.

- **Все данные должны содержаться в уникальных элементах**. Если модель сущности требует или допускает дублирование элементов, значит, она нарушает первую нормальную форму.

В качестве примера первой нормальной формы рассмотрим список базовых лагерей и туров базы данных туристической фирмы Cape Hatteras Adventures. В табл. 2.3 представлены данные о базовых лагерях, которые нарушают первую нормальную форму. Повторяющийся атрибут тура не является уникальным.

Таблица 2.3. Нарушение первой нормальной формы

BaseCamp	Tour1	Tour2	Tour3
Ashville	Appalachian Trail	Blue Ridge Parkway Hike	
Cape Hatteras	Outer Banks Lighthouses		
Freeport	Bahamas Dive		
Ft. Lauderdale	Amazon Trek		
West Virginia	Gauley River Rafting		

Чтобы исправить эту модель и привести ее к первой нормальной форме. сведем повторяющиеся группы атрибутов тура в единый уникальный атрибут (табл. 2.4). После этого мы переместим все повторяющиеся значения в уникальные элементы. Сущность `BaseCamp` содержит

уникальный элемент для каждого базового лагеря, а атрибут `BaseCampID` (идентификатор базового лагеря) сущности `Tour` ссылается на первичный ключ сущности `BaseCamp`.

Таблица 2.4. База данных, удовлетворяющая первой нормальной форме

Сущность `Tour`		Сущность `BaseCamp`	
`BaseCampID` (внешний ключ)	`Tour`	`BaseCampID` (первичный ключ)	`Name`
1	Apalachian Trail	1	Ashville
1	Blue Ridge Parkway Hike	2	Cape Hatteras
2	Outer Banks Lighthouses	3	Freeport
3	Bahamas Dive	4	Ft. Lauderdale
4	Amazon Trek	5	West Virginia
5	Gauley River Rafting		

Еще одним примером структуры данных, которая требует приведения к первой нормальной форме, является корпоративный каталог продукции, содержащий подразделение, модель, цвет, размер и прочие характеристики в единой сущности. Мне даже доводилось видеть каталоги, которые были настолько сложные, что содержали числа, указывающие на синтаксис последующих чисел.

С теоретической точки зрения этот тип модели некорректен, поскольку атрибут не является скалярным значением. С практической точки зрения в нем есть следующие проблемы.

- Использование одного или двух чисел для каждого элемента данных указывает на то, что база данных скоро преодолеет диапазон допустимых значений.
- Базы данных не индексируются на основе внутренних значений строки, таким образом, любой поиск потребует сканирования всей таблицы, т.е. перебора всех ее строк.
- Бизнес-правила тяжело программировать и поддерживать.

Сущности с нескалярными атрибутами нужно полностью перепроектировать таким образом, чтобы каждый отдельный атрибут данных имел свой собственный атрибут.

Вторая нормальная форма (2НФ)

Вторая нормальная форма гарантирует, что каждый атрибут на самом деле является атрибутом этой сущности. Это вопрос зависимости. Каждый атрибут должен требовать свой первичный ключ, в противном случае он не будет принадлежать базе данных.

Если первичный ключ сущности представляет собой одно значение, все не так сложно. Составные первичные ключи могут иногда привести к противоречию со второй нормальной формой, если атрибуты не зависят от каждого из атрибутов первичного ключа. Если атрибут зависит только от одного атрибута первичного ключа и не зависит от другого, то существует только частичная зависимость, что нарушает вторую нормальную форму.

Примером базы данных, нарушающей вторую нормальную форму, является база данных телефонных номеров базовых лагерей туристической компании Cape Hatteras Adventures, добавленная в сущность `BaseCampTour` (табл. 2.5). Предположим, что первичный ключ представляет собой объединение атрибутов `BaseCamp` и `Tour` и что телефонные номера закреплены именно за лагерями, а не за турами.

Таблица 2.5. Нарушение второй нормальной формы

BaseCamp (ПК)	Tour (ПК)	BaseCampPhoneNumber
Ashville	Appalachian Trail	828-555-1212
Ashville	Blue Ridge Parkway Hike	828-555-1212
Cape Hatteras	Outer Banks Lighthouses	828-555-1213
Freeport	Bahamas Dive	828-555-1214
Ft. Lauderdale	Amazon Trek	828-555-1215
West Virginia	Gauley River Rafting	828-555-1216

Проблема данной модели состоит в том, что телефонный номер является атрибутом только базового лагеря, но не тура, таким образом, он только частично зависит от первичного ключа сущности. (Более существенной проблемой является то, что составной первичный ключ неоднозначно определяет базовый лагерь.)

Очевидной практической проблемой этой модели является то, что обновление телефонных номеров требует обновления множества кортежей, в противном случае существует риск, что одному лагерю будут соответствовать два телефона.

Решение этой проблемы состоит в том, чтобы удалить из сущности частично зависимый атрибут и создать отдельную сущность с уникальным первичным ключом для базового лагеря (табл. 2.6). Эта новая сущность будет подобающим местонахождением зависимого атрибута.

Таблица 2.6. Приведение ко второй нормальной форме

Сущность Tour		Сущность BaseCamp	
BaseCamp (ПК)	Tour (ПК)	BaseCamp (ПК)	PhoneNumber
Ashville	Appalachian Trail	Ashville	828-555-1212
Ashville	Blue Ridge Parkway Hike	Cape Hatteras	828-555-1213
Cape Hatteras	Outer Banks Lighthouses	Freeport	828-555-1214
Freeport	Bahamas Dive	Ft.Lauderdale	828-555-1215
Ft.Lauderdale	Amazon Trek	West Virginia	828-555-1216
West Virginia	Gauley River Rafting		

Атрибут PhoneNumber теперь полностью зависит от первичного ключа сущности. Каждый телефонный номер хранится только в одном месте, при этом отсутствует какая-либо частичная зависимость.

Третья нормальная форма (3НФ)

Третья нормальная форма проверяет наличие транзитивных зависимостей. *Транзитивная зависимость* аналогична частичной в том, что они обе относятся к атрибутам, которые не в полной мере зависят от первичного ключа. Зависимость называется транзитивной, когда один атрибут зависит от другого, который, в свою очередь, зависит от первичного ключа.

Как и вторая нормальная форма, третья разрешается перемещением независимого атрибута в новую сущность.

Продолжая работать с примером базы данных турфирмы Cape Hatteras Adventures, назначим каждому базовому лагерю ответственного экскурсовода. Атрибут BaseCampGuide при-

надлежит сущности BaseCamp, однако если имя экскурсовода сопровождает какая-либо дополнительная информация, третья нормальная форма нарушается (табл. 2.7).

Таблица 2.7. Нарушение третьей нормальной формы

Сущность BaseCamp			
BaseCamp (ПК)	**BaseCampPhoneNumber**	**LeadGuide**	**DateOfHire**
Ashville	828-555-1212	Jeff Davis	1/5/1999
Cape Hateras	828-555-1213	Ken Frank	15/4/1997
Freeport	828-555-1214	Dab Smith	7/7/2001
Ft.Lauderdale	828-555-1215	Sam Wilson	1/1/2002
West Virginia	828-555-1216	Lauren Jones	1/6/2000

Атрибут DateOfHire (дата приема на работу) описывает экскурсовода, а не базовый лагерь, следовательно, он не напрямую зависит от первичного ключа сущности. Эта зависимость является транзитивной через атрибут LeadGuide.

Создание сущности Guide и перемещение атрибутов экскурсовода в нее устраняет нарушение третьей нормальной формы, а также проясняет логическую модель (табл. 2.8).

Таблица 2.8. Приведение к третьей нормальной форме

Сущность BaseCamp		Сущность LeadGuide	
BaseCamp (ПК)	**LeadGuide**	**LeadGuide (ПК)**	**DateOfHire**
Ashville	Jeff Davis	Jeff Davis	1/5/1999
Cape Hatteras	Ken Frank	Ken Frank	15/4/1997
Freeport	Dab Smith	Dab Smith	7/7/2001
Ft.Lauderdale	Sam Wilson	Sam Wilson	1/1/2002
West Virginia	Lauren Jones	Lauren Jones	1/6/2000



Если сущность имеет хороший первичный ключ, все ее атрибуты скалярны и полностью зависят от первичного ключа, то логическая модель находится в третьей нормальной форме. Большинство баз данных останавливается именно на третьей нормальной форме.

Дополнительные нормальные формы решают проблемы за счет более сложной логической модели. Если вам интересно работать с головоломными проблемами моделирования и создавать креативные решения, то вам поможет знание дополнительных нормальных форм.

Нормальная форма Бойса–Кодда (BNCF)

Нормальная форма Бойса–Кодда занимает положение между третьей и четвертой нормальными формами и решает проблемы сущностей, которые могут иметь два набора первичных ключей. Эта форма предполагает, что в данном случае сущность должна быть разбита на две — по одной для каждого первичного ключа.

Четвертая нормальная форма (4НФ)

Четвертая нормальная форма решает проблемы, создаваемые сложными составными первичными ключами. Если два независимых атрибута сведены для формирования первичного ключа с третьим и при этом оба атрибута однозначно не определяют элемент без третьего, то модель нарушает четвертую нормальную форму.

В качестве примера представим следующую ситуацию.

1. В качестве составного первичного ключа используются атрибуты `BaseCamp` (базовый лагерь) и `LeadGuide` (ответственный экскурсовод).
2. Атрибуты `Event` и `Guide` сведены вместе в качестве первичного ключа.
3. Так как оба ключа используют экскурсовода, все три сведены в единую сущность.

Приведенный пример нарушает четвертую нормальную форму.

Четвертая нормальная форма используется, чтобы помочь идентифицировать сущности, которые могут быть разбиты на несколько отдельных. Обычно это касается только сложных составных ключей, в которых сведено несколько независимых атрибутов.

Пятая нормальная форма (5НФ)

Пятая нормальная форма реализует метод проектирования сложных отношений, включающих в себя множество (три и более) сущностей. Если *тройственное* отношение правильно построено, оно не нарушает пятую нормальную форму. Строгость таких отношений может быть как однозначной, так и множественной. Тройственное отношение в таком случае составляют множество связанных сущностей.

В качестве примера тройственного отношения можно рассмотреть производственный процесс, включающий оператора, машину и список материалов. С определенной точки зрения это должна быть операционная сущность, содержащая три внешних ключа. В качестве альтернативы можно рассмотреть вариант тройственного отношения с дополнительными атрибутами.

Подобно отношению "многие ко многим" между двумя сущностями, тройственное отношение требует разделения сущностей в модели физической схемы, что приведет к преобразованию отношения "многие ко многим" в несколько искусственных отношений "один ко многим". Однако в данном случае связывающая сущность будет иметь три или более внешних ключа.

В таком сложном отношении пятая нормальная форма требует, чтобы каждая сущность, участвующая в разделенном тройственном отношении, оставалась целостной, без какой-либо потери данных.

Основной задачей реляционных баз данных является моделирование реальности. Здесь поддержание целостности данных важно как с теоретической, так и с практической стороны процесса разработки баз данных.

Одним из ключевых вопросов поддержания целостности базы данных является обучение владельцев данных тому, что они сами должны оценивать целостность и не только "владеть" задачей и проектом, но и поддерживать целостность данных. Вопрос целостности данных не возникает случайно, сам по себе. Им нужно заниматься с первого дня работы над проектом.

Одним из самых сложных факторов целостности данных являются старые данные, переносимые из устаревших систем. Если эти старые данные не удовлетворяют условиям целостности реляционных данных, то могут возникнуть серьезные проблемы.



Ответственный за данные старой системы должен чувствовать персональную ответственность за успех нового проекта. Нужно дать разработчикам старой системы почувствовать, что они отвечают за целостность данных и вовлечены в разработку нового проекта. Это лучше, чем представить им новый проект как готовый продукт и дать почувствовать себя оттесненными в сторону. Нужно позволить им самим ставить себе задачи и помогать их решать самым наилучшим образом.

Реляционная алгебра

Реляционные базы данных по своей природе сегментируют данные на несколько узких, но длинных таблиц. Редко когда в одной таблице содержатся данные, имеющие самостоятельное значение. Таким образом, важной задачей для разработчиков является слияние данных из множества таблиц. Теория слияния наборов данных называется *реляционной алгеброй*; она была разработана в 1970 году Эдгаром Коддом.

Реляционная алгебра содержит восемь *реляционных операторов*.

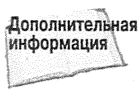
- **Ограничение**. Возвращает строки, которые удовлетворяют заданному условию.
- **Проекция**. Возвращает из набора данных заданные столбцы.
- **Произведение**. Реляционное умножение возвращает все возможные комбинации данных из двух наборов.
- **Слияние**. Реляционное сложение и вычитание, которое объединяет по вертикали две таблицы, помещая одну над другой и выравнивая столбцы.
- **Пересечение**. Возвращает строки, общие для двух наборов данных.
- **Разность**. Возвращает строки, уникальные для одного набора данных.
- **Объединение**. Возвращает объединение двух таблиц по горизонтали, сопоставляя их строки по общим данным.
- **Деление**. Возвращает строки, точно совпадающие в двух наборах данных.

В дополнение, в качестве метода, поддерживающего реляционную алгебру, SQL предлагает следующее.

- **Подзапросы**. Это сходный с объединением, но более гибкий механизм. Результаты подзапроса могут использоваться вместо выражения, списка или набора данных во внешнем запросе.

На формальном языке реляционной алгебры:

- таблица или набор данных называется *отношением* или *сущностью*;
- строка называется *кортежем*;
- столбец называется *атрибутом*.



Реляционную алгебру мы применим на практике в главах 9 и 10.

Резюме

Реляционная теория существует уже более трех десятилетий и была более четко определена, когда производители систем управления базами данных поработали над ее расширениями, а теоретики более четко определили задачу представления реальности с помощью структур данных. В то же время оригинальная работа доктора Кодда остается фундаментом проектирования и реализации реляционных баз данных.

Умение проектировать реляционные базы данных является фундаментальным умением любого разработчика. Чтобы состояться как разработчик, нужно четко представлять себе весь набор доступных шаблонов. В следующей главе мы перейдем к вопросу двухмерных реляционных и трехмерных объектно-ориентированных баз данных.

Архитектура SQL Server 2005

ГЛАВА

3

В этой главе...

Архитектура "клиент/сервер"

Ядро реляционной базы данных

Компоненты SQL Server

Службы SQL Server

Я читал Толкиена еще в старших классах школы, за десятилетия до того, как Голливуд представил истории Средиземья широким массам. Я до сих пор помню фразу: "...и одно кольцо, чтобы править всеми остальными". Хотя сюжет этой трилогии вращался вокруг всего одного кольца, существовало еще несколько других колец, и все они наверняка обладали интересными свойствами. В то же время всего одно кольцо было стратегическим, завершающим звеном. И именно на его истории был построен сюжет трилогии.

Одно время для меня ключевым вопросом был следующий: "А зачем обновлять СУБД до версии SQL Server 2005?" Существовало множество ответов, которые упоминали новые технологии, окупающие затраченные на обновление версии деньги. Это и повышенная производительность, и доступность, и новые средства бизнес-аналитики. Я уверен, что любое из этих улучшенных средств может оказаться решающим фактором при принятии решения относительно обновления.

В аналитическом отчете, опубликованном в еженедельнике *ComputerWeekly* 17 августа 2004 года, говорилось: "Yukon (это было рабочим названием SQL Server 2005) является наспех собранной редакцией пакета, в которой отсутствует четкая направленность и в*и*дение. Несмотря на то что компания Microsoft заявляет, будто откладывает выпуск этого продукта, чтобы повысить его качество, мы считаем, что просроченный выпуск версии Yukon связан с отсутствием четкого направления развития SQL Server в самой компании Microsoft".

Сказать, что я был не согласен с этим утверждением, — это не сказать ничего.

Подобно кольцам в романе Толкиена, в SQL Server 2005 вошло бесчисленное множество новых и улучшенных технологий. Можно ли среди них выбрать тот решающий фактор, который подвигнет пользователя на переход к версии SQL Server 2005? Можно ли в SQL Server 2005 найти то самое решающее кольцо?

Многие новые средства, такие как зеркальное отображение баз данных, являются результатом эволюции, но не революции. Многие считают, что наиболее существенным является инте-

грация SQL Server и среды .NET. В некотором смысле с этим утверждением согласен и я. Если среда .NET используется в SQL Server для построения быстрых скалярных функций и замены существующих хранимых процедур, то это тоже не более чем эволюция. Ни одно из этих средств не является тем самым единственным кольцом.

Архитектура, основанная на службах (SOA), позволяет на стандартном уровне абстракции создавать приложения, которые могут взаимодействовать с другими SOA-приложениями посредством XML, SOAP и Web-служб. При более широком распространении SOA по сравнению с SQL Server 2000 коренным образом изменилась среда вычислений. Введение новых SOA-оптимизированных технологий позволило SQL Server 2005 вписаться в этот новый архитектурный ландшафт.

Оставшись прекрасной системой управления базами данных для традиционной архитектуры "клиент/сервер" и многоуровневой архитектуры, SQL Server 2005 существенно нарастила технические мощности, которые теперь позволяют выполнять следующее:

- предоставлять данные XML в качестве хранилищ данных SOA;
- предоставлять данные XML и XSLT непосредственно Web-браузерам;
- осуществлять хранение, индексацию, массовую вставку и запросы к данным XML непосредственно в ядре СУБД;
- оптимизировать обслуживание особо крупных баз данных предприятий.

Для достижения новых горизонтов команда разработчиков SQL компании Microsoft усилила существующие службы и добавила такое множество новых технологий, что это может удивить даже умудренных опытом администраторов баз данных.

Существует масса веских причин перехода на версию SQL Server 2005. В их числе повышенная производительность, доступность и обилие функций бизнес-анализа. Однако ключевым "тем самым кольцом" является то, что в SQL Server 2005 заложены новые архитектурные принципы.

SQL Server 2005 является сервером баз данных третьего поколения от компании Microsoft (рис. 3.1). Это более чем один продукт, более чем ядро базы данных, несравненно больше, чем SQL Server 2000. SQL Server 2005 — это широкий набор средств и инструментов работы с данными, которые можно использовать для построения решений в такой архитектурной среде, в которой это было невозможно еще несколько лет назад.

В этой главе мы проведем краткий экскурс по СУБД SQL Server 2005, чтобы вы смогли получить общее представление о ней.



Термин *архитектура SQL Server* затрагивает такие концепции, как организация страниц данных и индексов, планирования и кэширования запросов, а также оптимизатора запросов. Для более полного разъяснения этих вопросов вы можете обратиться к главе 50.

Архитектуры доступа к данным

Прежде всего, SQL Server является сервером баз данных. Сама по себе СУБД не может удовлетворить потребности конечного пользователя (если, конечно, не рассматривать редактор запросов как интерфейс пользователя). Если вы мало знакомы с моделью "клиент/сервер", ее нужно понять, в противном случае будет сложно понять и саму СУБД.

Модель баз данных "клиент/сервер"

С технической точки зрения термин *клиент/сервер* связан с двумя взаимодействующими процессами. Клиентский процесс запрашивает у серверного процесса некую службу, которая,

в свою очередь, обрабатывает запрос клиента. Клиентский и серверный процессы могут быть запущены на разных компьютерах или на одном. В данном вопросе важно само взаимодействие процессов, а не их физическое размещение.

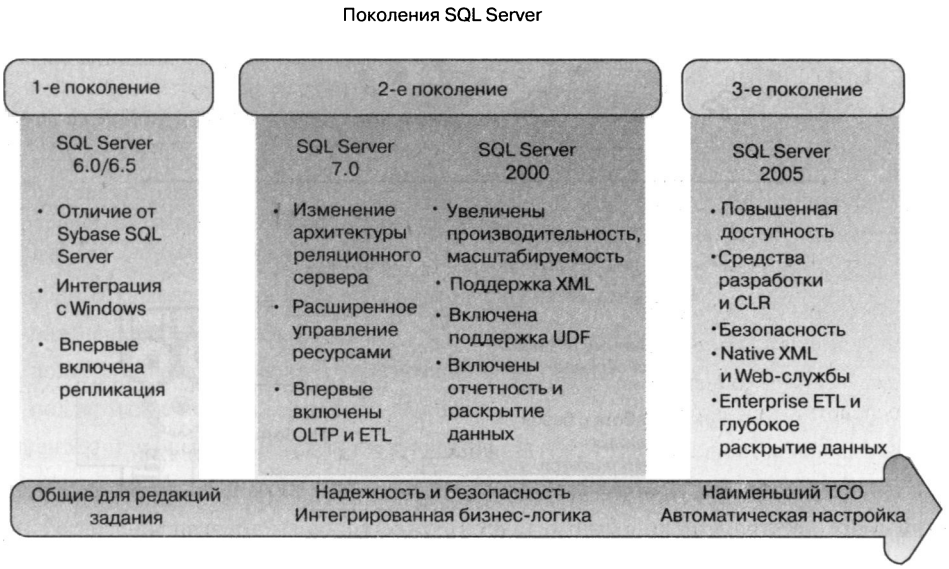


Рис. 3.1. SQL Server 2005 — более чем обновленная версия продукта. Учитывая такое множество совершенно новых архитектурных решений, эту систему можно назвать платформой баз данных третьего поколения

Термин *клиент/сервер* затрагивает многие аспекты вычислений. Файловые серверы, серверы печати и поставщики служб Интернета (провайдеры) — это примеры реализации модели "клиент/сервер". Файловые серверы открывают доступ пользователю к файлам, серверы печати обрабатывают запросы к принтерам, а провайдеры обслуживают запросы к службам Интернета. В базах данных архитектуры "клиент/сервер" сервер баз данных обслуживает запросы к базе от клиентского процесса.

Базы данных архитектуры "клиент/сервер"

В противоположность настольным базам данных, таким как Microsoft Access, выполняющим всю работу на компьютере клиента, базы данных с архитектурой "клиент/сервер" подобны библиотекарю, который принимает запрос клиента, ищет запрошенную информацию и возвращает фотокопию найденных материалов. Содержащиеся в библиотеке реальные материалы никогда не выходят из поля зрения библиотекаря.

В базах данных с архитектурой "клиент/сервер" клиент подготавливает запрос на языке SQL (небольшое текстовое сообщение) и отсылает его на сервер баз данных, который читает и обрабатывает его (рис. 3.2). В сервере поддерживается система безопасности, индексируются хранящиеся материалы, заносятся и обрабатываются данные, выполняются серверные программы, и выполняется доставка результатов запросов клиенту.

Вся работа с базой данных выполняется на сервере. Если клиент запрашивает какой-либо набор данных, он подготавливается на сервере, и его копия доставляется клиенту. Реальные данные и индексы никогда не покидают пределы сервера. Когда клиент запрашивает выполнение операции вставки, обновления или удаления, сервер получает этот запрос и сам обрабатывает его.

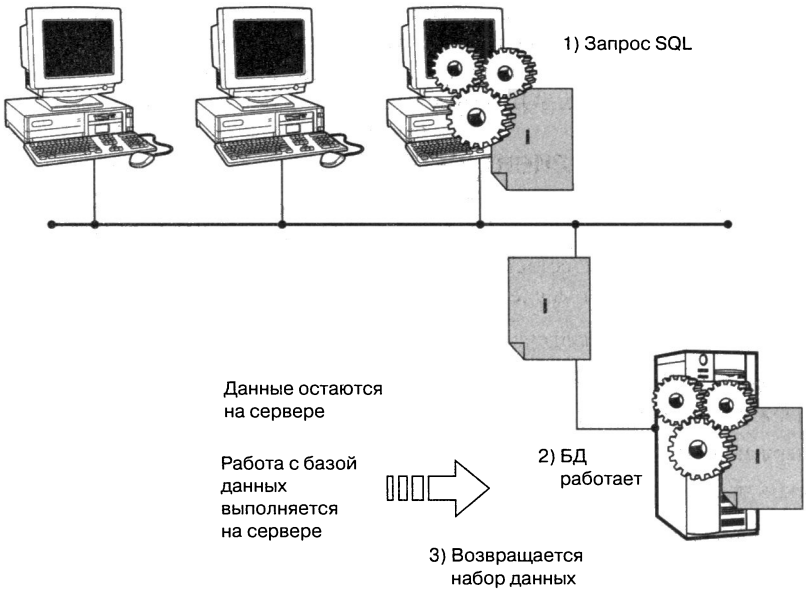


Рис. 3.2. База данных с архитектурой "клиент/сервер" выполняет всю работу на сервере

Клиент-серверная модель базы данных обладает рядом преимуществ по сравнению с настольной моделью.

- Повышена достоверность данных, поскольку они не разбросаны по всей сети и разным приложениям. Данные обслуживает только один процесс.
- Ограничения целостности данных и бизнес-правила могут поддерживаться на уровне сервера, в результате чего они строго соблюдаются.
- Повышена базопасность данных, поскольку база данных хранит их в пределах одного сервера. Открыть файл данных, защищаемый сервером, гораздо сложнее, чем файл на рабочей станции.
- Повышена производительность и лучше сбалансированы рабочие станции, поскольку большая часть работы (обработка базы данных) выполняется на сервере, а рабочие станции берут на себя только обслуживание интерфейса пользователя. Поскольку серверный процесс обеспечивает быстрый доступ пользователя к файлам данных, а большая часть данных кэширована в памяти, операции с базой данных выполняются быстрее, чем в многопользовательской настольной среде. Сервер баз данных обслуживает всех пользователей, работающих с приложениями баз данных, таким образом, гораздо проще оценить стоимость устанавливаемого сервера.
- В значительной мере сокращаются сетевые потоки. По сравнению с сетевыми потоками, создаваемыми многопользовательскими настольными системами, потоки в архитектуре "клиент/сервер" можно сравнить с одиноким мотоциклистом, несущимся по свободной 10-полосной автостраде. Без преувеличения! Замена перегруженной настольной системы базой данных "клиент/сервер" способна сократить сетевые потоки больше чем на 95%.

- Снижение сетевых потоков в системах "клиент/сервер" приводит к тому, что приложения хорошо работают даже в распределенной среде и даже при наличии медленных соединений. Такие маленькие сетевые потоки позволяют уравнять в производительности локальную сеть со скоростью 100 Мбит/с с модемным подключением со скоростью 56 Кбит/с для клиентских приложений, использующих .NET-технологии и подключенных к базе данных SQL Server.

Роли в архитектуре "клиент/сервер"

В клиент-серверной конфигурации базы данных каждая из сторон играет конкретную роль. Если не упорядочить эти роли, то производительность и целостность приложения клиент-серверной базы данных существенно снизятся.

Сервер баз данных отвечает за следующее:

- обработка запросов на извлечение и модификацию данных;
- интенсивная обработка данных;
- поддержание всех правил и ограничений базы данных;
- поддержание безопасности базы данных.

Клиентский процесс отвечает за следующее:

- представление данных пользователю в понятном и удобном формате;
- обеспечение интерфейса пользователя всевозможными инструментами, данными и отчетами;
- отправку запросов серверу.



Клиент-серверные реализации работают лучше всего, когда строго поддерживается уровень абстракции данных. Это значит, что любой запрос на доступ к данным пропускается через хранимые процедуры уровня абстракции данных. Небрежные клиент-серверные модели, позволяющие приложениям или отчетам напрямую обращаться к таблицам базы данных, нежизнеспособны, поскольку база данных не будет расширяемой. Внесение изменений в такую схему базы данных затронет такое множество других объектов, что информационный отдел просто испугается такой работы и предпочтет оставить устаревшую модель.

Многоуровневая архитектура

Часто в клиент-серверных приложениях баз данных, кроме клиентского и серверного, задействованы дополнительные процессы. Для обслуживания подключений и реализации бизнес-логики часто используются процессы среднего уровня.

Средний уровень, обслуживающий подключения, может оказаться полезным, поскольку множество пользователей смогут получить преимущества всего от нескольких открытых подключений к серверу баз данных. Однако этот тип подключения влияет на способ аутентификации сервером баз данных отдельных пользователей. Это следует принять во внимание при формировании плана обеспечения безопасности базы данных.

Если доступно больше одного сервера, то объект общего подключения облегчает переключение пользователей между серверами А и Б, если первый внезапно окажется недоступным. Объект подключения становится единственной точкой, которая способна автоматически выявить такую ситуацию и переключиться на второй сервер. Этот тип решения хорошо работает при зеркальном отображении баз данных.

Помещение бизнес-правил и уровня абстракции данных на логический уровень имеет смысл для облегчения разработки и обслуживания. Основной вопрос сводится к тому, где

именно разместить этот логический уровень. Некоторые разработчики доказывают, что логический средний уровень должен размещаться на собственном сервере в среде .NET, отдельно от сервера баз данных. Я с этим не согласен по двум причинам.

- Бизнес-правила и ограничения базы данных должны поддерживаться сервером баз данных на физическом уровне, чтобы их не мог обойти ни один процесс или приложение.
- Программирование бизнес-правил на сервере баз данных предоставляет программам более быстрый доступ к классификаторам базы данных, что повышает производительность.

Единственным недостатком программирования бизнес-правил на сервере является необходимость изучения разработчиками серверного программирования и то, что приложению базы данных труднее связываться с другими продуктами поддержки баз данных.

Архитектура, ориентированная на службы

Архитектура, ориентированная на службы (SOA), является альтернативой клиент-серверной архитектуре. Вместо программирования прикладного интерфейса "клиент/сервер" между несколькими системами, SOA использует стандартные вызовы HTTP и XML, позволяя множеству систем взаимодействовать с одной, используя один и тот же интерфейс.

Архитектура SOA может оказаться полезной на крупных предприятиях с несколькими особо большими системами, но за такую масштабируемость приходится платить. Данные проходят несколько уровней, и при этом выполняется их преобразование. Интерфейс конечного пользователя, использующий SOA, будет испытывать недостаток производительности по сравнению с прямым подключением в архитектуре "клиент/сервер" с использованием библиотеки ADO.NET.

Мне встречались организации, которые разбивали приложения, предназначенные для работы с данными среднего размера, на несколько маленьких и использовали SOA для организации взаимодействия между ними. Это можно сравнить с катастрофой. Просто смешно использовать вызовы Web-служб для проверки внешних ключей.

Когда же лучше использовать Web-службы SOA? Они больше подходят в качестве вторичного метода доступа к данным для облегчения взаимодействия с другими особо крупными системами. За исключением этих случаев старайтесь не использовать SOA.



Развертывание хранилищ данных SOA с применением SQL Server 2005 облегчается при использовании новых функций концевых точек HTTP. Более подробно этот вопрос мы обсудим в главе 32.

Службы SQL Server

SQL Server — нечто большее, чем просто ядро реляционной базы данных. Это набор служб и компонентов, связанных с базами данных, которые могут совместно использоваться для построения мощных решений для конечного пользователя (рис. 3.3).

Реляционное ядро

Реляционное ядро SQL Server 2005, иногда называемое *ядром базы данных*, является сердцем SQL Server. Это процесс, поддерживающий всю реляционную работу базы данных. SQL — декларативный язык, т.е. он описывает ядру базы данных запрос, который тот должен выполнить. Именно он заставляет работать этот механизм.

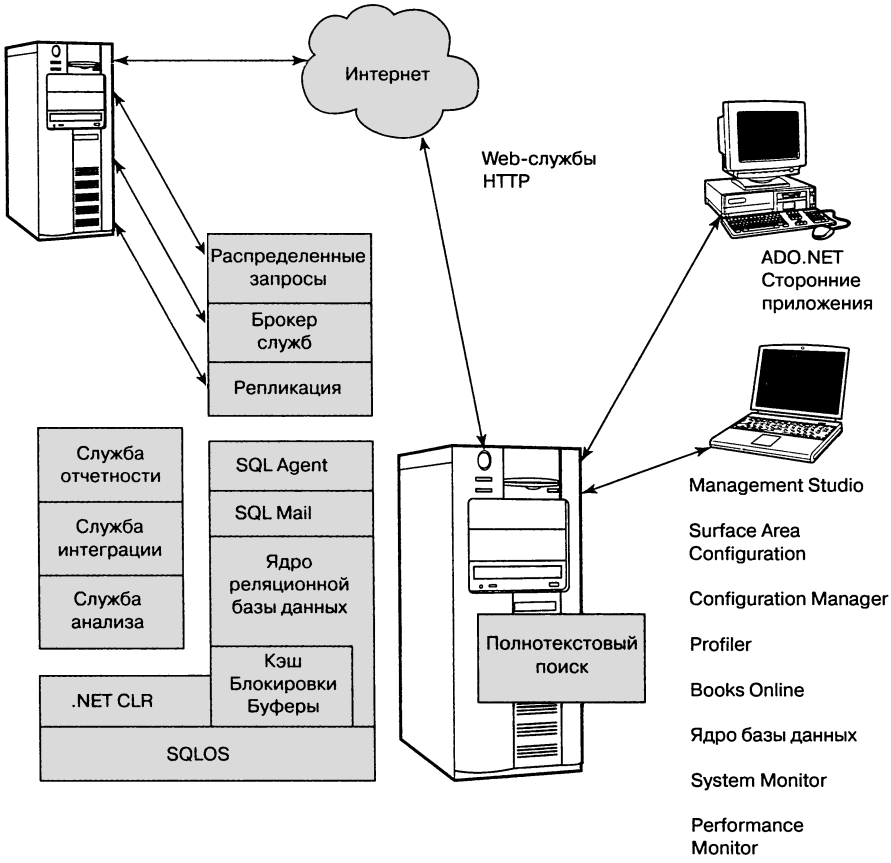


Рис. 3.3. SQL Server — это набор серверных и клиентских компонентов

SQL Server поддерживает установку до пятидесяти экземпляров реляционного ядра на одном физическом сервере. Несмотря на то что все установленные экземпляры совместно используют некоторые компоненты, каждый экземпляр функционирует как полноценная установка SQL Server.



С выходом пакета обновлений SP2 улучшенное кэширование плана запроса обеспечивает повышенную производительность системы, а также более эффективное использование оперативной памяти для страниц базы данных. Также обеспечивается возможность возвращения текстовых планов запросов XML, содержащих больше вложенных уровней XML, чем 128. При этом используется новая табличная функция `sys.dm_exec_text_query_plan`.

В реляционном ядре существует несколько ключевых процессов и компонентов, в том числе следующие.

- **Оптимизатор запросов (Query Optimizer).** Определяет, как обрабатывать запрос, основываясь на относительных затратах, которые влечет за собой выполнение определенных типов операций. Оцененный и фактический планы выполнения запроса можно просмотреть в графическом представлении или в форме XML с помощью утилиты Management Studio (SSMS).

- **Диспетчер буфера (Buffer Manager).** Анализирует используемые страницы данных и предварительно загружает определенные фрагменты файлов данных в память, снижая таким образом зависимость от производительности операций ввода-вывода на диск.
- **Lazy Writer.** Записывает страницы данных, которые были изменены в памяти, в файл данных.
- **Монитор ресурсов (Resource Monitor).** Оптимизирует кэш планов запросов в соответствии с нагрузкой на память и избирательно удаляет из него старые планы.
- **Диспетчер блокировок (Lock Manager).** Динамически управляет множеством блокировок, выполняя балансировку количества затребованных блокировок с их размерами.
- SQL Server поглощает ресурсы компьютера, поэтому ему нужен прямой контроль за доступными ресурсами (памятью, потоками, запросами на ввод-вывод и т.п.). Отдать управление ресурсами на откуп операционной системе совсем не сложно для SQL Server 2005. Она содержит собственный уровень ОС, называемый SQLOS, который управляет всеми внутренними ресурсами.



СУБД SQL Server прошла длинный путь с 1987 года, когда компания Sybase создала SQL Server для Unix. С тех пор СУБД пережила множество версий и обросла новыми технологиями. Если вам интересно прочитать об истории SQL Server, о различных кодовых названиях бета-версий, эволюции реализованных технологий, увидеть копии экранов прошлых версий, посетите специальную страницу "SQL Server History" по адресу `www.SQLServerBible.com`.



С выходом пакета обновлений SP1 было активизировано зеркальное отображение баз данных; оно поддерживается также и в промышленных реализациях.



Планы обслуживания теперь поддерживаются установкой службы SQL Server Database. До выхода пакета обновлений SP2 для запуска плана обслуживания в серверных установках требовалась инсталляция службы интеграции SQL Server 2005.

Transact-SQL

СУБД SQL Server основана на стандарте SQL с некоторыми добавленными компанией Mictrosoft расширениями. Стандарт SQL впервые был введен доктором Коддом в 1971 году, когда он работал в исследовательской лаборатории IBM в Сан-Хосе. СУБД SQL Server 2005 в своей основе совместима со стандартом ANSI SQL-92. Полная спецификация стандарта ANSI SQL размещена в пяти документах, которые можно приобрести на сайте `www.techstreet.com/ncits.html`.

Хотя стандарт ANSI SQL можно назвать идеальным для команд общего назначения отбора и определения данных, в него не входят команды, предназначенные для управления свойствами SQL Server и логического управления пакетами, необходимые для создания специфичных для приложений SQL Server. Исходя из этого, команда Microsoft SQL Server расширила спецификацию ANSI SQL рядом новых команд, а некоторые команды реализовала несколько по-своему. Результатом стал язык Transact-SQL, или, сокращенно, T-SQL, — диалект SQL, распознаваемый SQL Server.

В T-SQL нет некоторых команд ANSI SQL (таких, как определение каскадности внешних ключей, обработки пустых значений и умолчаний) в основном потому, что компания Microsoft

реализовала эти функции иначе. По умолчанию T-SQL несколько по-иному, чем требует стандарт ANSI, обрабатывает пустые значения, кавычки и заполнения, но этим режимом можно управлять. Основываясь на своем личном опыте, могу сказать, что ни одно из этих отличий не влияет на процесс разработки приложений для SQL Server. T-SQL гораздо больше добавил в стандарт ANSI SQL, чем проигнорировал в нем.

Чтобы освоить SQL Server, нужно ознакомиться с языком T-SQL, поскольку ядро SQL Server понимает только этот язык. Любая команда, посылаемая серверу, должна поддерживать правила этого языка. Пакеты хранимых команд T-SQL могут выполняться на сервере как хранимые процедуры. Другие инструменты, такие как Management Studio, реализующие графический интерфейс пользователя, позволяют управлять сервером, преобразуя щелчки мышью на определенных инструментах в соответствующие команды T-SQL, отправляемые ядру базы данных.

Команды SQL и T-SQL можно разбить на три категории.

- **Язык манипулирования данными (Data Manipulation Language — DML)**. Включает в себя популярные инструкции SQL `select`, `insert`, `update` и `delete`. DML иногда ошибочно называют языком модификации данных (Data Modification Language), но это неверно, поскольку инструкция `select` данные не модифицирует, а извлекает.
- **Язык определения данных (Data Definition Language — DDL)**. Содержит инструкции, создающие и модифицирующие таблицы данных, ограничения и прочие объекты базы данных.
- **Язык управления данными (Data Control Language — DCL)**. Содержит инструкции управления системой безопасности, такие как `grant`, `revoke` и `deny`.

Visual Studio и CLR

Одним из самых впечатляющих новшеств является возможность совместной работы систем Visual Studio и SQL Server 2005. Несмотря на то что и в прошлом эти системы находились в тесной связи, сейчас интеграция Visual Studio 2005 и SQL Server 2005 сильнее, чем когда-либо раньше. Эти два продукта разрабатывались и были выведены на рынок одновременно, и это сразу видно. Утилита Management Studio основана на интегрированной среде разработки (IDE) пакета Visual Studio. Однако интеграция сильнее, чем на первый взгляд видно по утилите Management Studio. Внутренняя операционная система пакета SQL Server 2005, SQLOS, на самом деле управляет общеязыковым интерпретатором .NET (CLR) внутри SQL Server.

Сборки, созданные в Visual Studio, могут быть развернуты и запущены в SQL Server в качестве хранимых процедур, триггеров, определенных пользователем обычных или агрегатных функций. К тому же типы данных, созданные в Visual Studio, могут использоваться для определения таблиц и хранения дополнительных данных.

Достаточно важно то, что в SQLOS поддерживается CLR. Это значит, что SQL Server управляет ресурсами CLR. СУБД может разрешить проблемы CLR, завершить выполнение и перезапустить процедуру CLR, которая стала источником проблем, и гарантировать, что в вечной войне за память победит нужный игрок.

По умолчанию общий интерпретатор языков CLR в SQL Server 2005 отключен. Но при желании его можно специально включить с помощью утилиты Surface Area Configuration Tool или команды T-SQL `set`. Когда интерпретатор включен, состояние любой сборки или способность доступа к программам вне SQL Server будет тщательно контролироваться.



Интеграция со средой .NET будет рассмотрена в главе 27 и 29.

Брокер служб

Впервые появившийся в версии SQL Server 2005 брокер служб (Service Broker) управляет асинхронными очередями данных, поддерживая ключевые средства производительности и масштабируемости и балансируя загрузку в течение времени.

- Брокер служб может буферизовать большие объемы вызовов Web-служб HTTP или хранимых процедур. Вместо того чтобы вызов тысячи Web-служб запустил одновременно на выполнение тысячу потоков хранимых процедур, эти вызовы помещаются в очередь, и хранимые процедуры могут быть вызваны несколькими экземплярами SQL Server, что более эффективно распределит нагрузку.
- Серверные процессы, содержащие сложную логику или периоды сильной нагрузки, могут поместить необходимые данные в очередь и вернуть их вызывающему процессу без нарушения логики. Брокер служб может провести через очередь вызов другой хранимой процедуры, которая возьмет на себя основную нагрузку.

Несмотря на то что в SQL Server можно создать собственные очереди, существует ряд преимуществ в использовании стандартных рабочих очередей Microsoft. SQL Server содержит команды DDL, управляющие брокером служб; есть команды T-SQL, помещающие данные в очередь и извлекающие их из нее. Информация об очередях брокера служб представлена в метаданных представлений, а также в утилитах Management Studio и System Monitor. Наиболее важно то, что брокер служб основательно протестирован и спроектирован специально для систем с исключительно большой нагрузкой.



Брокер служб является ключевой службой при создании хранилищ данных с архитектурой, ориентированной на службы. Более полную информацию по этому вопросу вы найдете в главе 32.

Служба репликаций

Данные SQL Server часто требуются в пространстве трансконтинентальных корпораций и прочих глобальных организаций, и при этом для перемещения данных часто используют репликации SQL Server. Дополненная новыми инструментами, служба репликаций SQL Server 2005 может перемещать транзакции в одностороннем направлении, а также объединять обновления из многих мест с помощью топологии издателя, распространителя и подписчика.

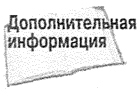


В главе 39 описаны различные модели репликации и методы их настройки.

Полнотекстовый поиск

Полнотекстовый поиск существовал в SQL Server с версии 7, но в каждой следующей версии этот прекрасный инструмент улучшался. Для быстрого поиска строк запросы SQL Server используют индексы. По умолчанию индекс строится по всему столбцу. Поиск отдельных слов внутри столбцов представляет собой очень медленный процесс. Полнотекстовый поиск (Full-Text Search) решил эту проблему, индексируя каждое из слов в столбце.

Когда полнотекстовый поиск был разработан для столбцов, запросы SQL Server получили возможность использовать при поиске полнотекстовые индексы и быстро находить в строках отдельные слова.



В главе 13 вы узнаете, как настроить и использовать полнотекстовый поиск в запросах SQL.

Служба Microsoft Search Service, выполняющая полнотекстовый поиск, на самом деле является компонентом операционной системы и предназначена для поиска текста в файлах. SQL Server при выполнении полнотекстового поиска берет на вооружение именно эту службу. Эта служба может быть остановлена и запущена в утилите SQL Server Configuration Manager или в консоли Службы панели управления Windows.

Служба уведомлений

Изначально выпущенная в 2003 году в качестве надстройки для SQL Server 2000, служба уведомлений (Notification Service) может быть запрограммирована для отправки сообщений при возникновении определенных событий с данными. Сообщения могут отсылаться практически на любое устройство, в том числе на пейджеры, мобильные телефоны и по электронной почте. Типичным приложением службы уведомлений может быть система сообщений на транспортных узлах, которая автоматически информирует пассажиров об изменении в расписании и задержках рейсов.

Объекты управления сервером (SMO)

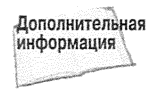
Server Management Objects (SMO) — это набор объектов, который открывает функции конфигурирования и управления SQL Server для программирования с помощью кода .NET. Объекты SMO не предназначены для разработки приложений баз данных — они используются производителями при разработке инструментария SQL Server, такого как утилита Management Studio, сторонние интерфейсы управления или утилиты резервного копирования. SMO использует пространство имен Microsoft.SQLServer.SMO.



Объекты SMO заменили старые объекты SQL-DMO, существовавшие в версии SQL Server 7 и SQL Server 2000. Объекты SMO имеют обратную совместимость с SQL Server 7 и SQL Server 2000.

SQL Server Agent

SQL Server Agent представляет собой дополнительный процесс, который выполняет задания SQL и обслуживает прочие автоматизированные задачи. Он может быть сконфигурирован для автоматического запуска при загрузке системы; также он может запускаться вручную из утилит SQL Server Configuration Manager и SQL Server Surface Area Configuration Tool.



В главе 38 вы подробнее узнаете об этом и других агентах SQL Server, а также о заданиях и почтовой службе.

Координатор распределенных транзакций

Координатор распределенных транзакций (Distributed Transaction Coordinator — DTC) представляет собой процесс, поддерживающий двухфазные подтверждения для транзакций, которые охватывают несколько серверов. Процесс DTC может быть запущен из консоли

служб панели управления Windows. Если приложение регулярно использует распределенные транзакции, то лучше сконфигурировать DTC для автоматического запуска при загрузке операционной системы.



В главе 15 вы узнаете, что такое двухфазное подтверждение и распределенные транзакции.

SQL Mail

Компонент SQL Mail позволяет серверу отправлять электронные сообщения на внешний почтовый ящик по протоколу SMTP. Почтовые сообщения могут генерироваться множеством источников в SQL Server, в том числе кодом T-SQL, заданиями, предупреждениями, службой интеграции и планами обслуживания.



В главе 38 будет показано, как настроить профиль электронной почты для SQL Server и отправлять сообщения.

Службы бизнес-аналитики

Бизнес-аналитика (Business Intelligence — BI) — это одна из областей, в которой SQL Server 2005 превосходит все остальные системы. Развитие средств бизнес-аналитики в SQL Server за последние несколько лет было стремительным. Результатом стало то, что версия SQL Server 2005 включает в себя три службы, специально предназначенные для бизнес-аналитики: службу интеграции (иногда для нее используется аббревиатура SSIS), службу отчетности (RS) и службу анализа (AS). Все эти три службы были созданы с помощью среды BI Developement Studio и управляются из утилиты Management Studio.

Служба интеграции

Служба интеграции (Integration Services) перемещает данные практически между любыми типами источников. Она является инструментом извлечения, преобразования и загрузки данных СУБД SQL Server. Как показано на рис. 3.4, служба интеграции для определения путей перемещения данных от одного подключения к другому использует графические средства. Пакеты службы интеграции достаточно гибкие, чтобы как напрямую копировать данные из столбца в столбец, так и выполнять сложные преобразования, использовать классификаторы и обработку исключений во время перемещения данных. Служба интеграции исключительно важна при преобразованиях данных, сборе информации из разнородных источников и обобщении информации хранилищ данных для анализа службой Analysis Services. Эта служба даже имеет инструменты для работы с нечеткими множествами данных.



Службу интеграции нельзя рассматривать лишь как развитие службы преобразования данных (DTS) СУБД SQL Server 2000. Служба интеграции — полностью переписанный инструмент извлечения, преобразования и загрузки данных уровня предприятия. Среди множества прекрасных новых инструментов SQL Server 2005 служба интеграции является одной из самых ярких.

Служба интеграции может вам сильно пригодиться. Если у вас есть опыт работы с другими базами данных, но в SQL Server вы новичок, это один из инструментов, который несомненно поразит вас. В то время как другая компания развернула бы маркетинг такой службы

интеграции как флагманского продукта, компания Microsoft глубоко ее запрятала в недрах SQL Server и не берет за ее установку дополнительной платы. Обязательно найдите время, чтобы открыть для себя службу интеграции, и вы не пожалеете.

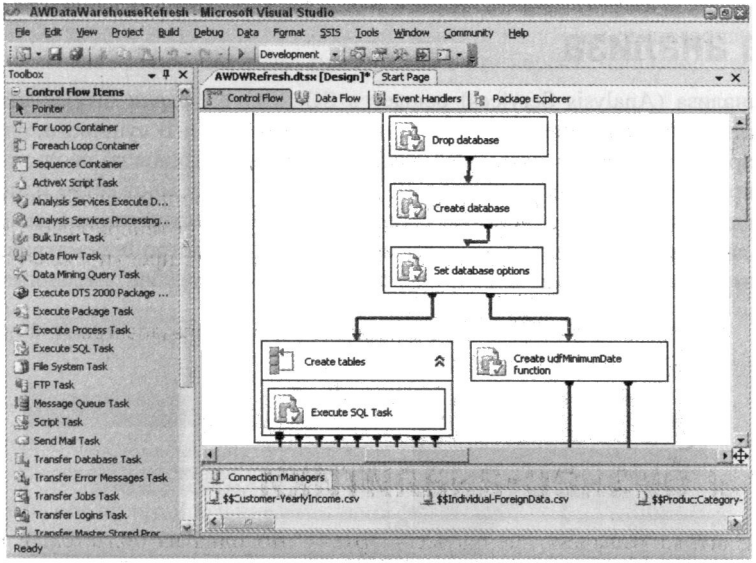


Рис. 3.4. Служба интеграции графически иллюстрирует преобразования данных при их перемещении

Дополнительная информация

В главе 42 описано, как создать и выполнить пакет SSIS.

Служба отчетности

Служба отчетности (Reporting Services) в SQL Server 2005 представляет собой полнофункциональное, основанное на Web-технологиях, управляемое решение. Ее отчеты могут быть экспортированы в форматы PDF, Excel и многие другие с помощью одного щелчка мышью; их также легко создавать и настраивать.

Отчеты определяются графически или программно и хранятся в качестве файлов `.rd` в базе данных службы отчетности в SQL Server. Их формированию может быть назначено расписание, они могут кэшироваться для пользователей, отправляться им по электронной почте или генерироваться по запросу с задаваемыми пользователями параметрами. Служба отчетности встроена в SQL Server, так что никаких дополнительных лицензий конечному пользователю не требуется. В то же время для обеспечения более высокой производительности многие администраторы баз данных выделяют для нее отдельный сервер. Если вы до сих пор используете пакет Crystal Reports, спросите себя — почему?

Новинка 2005

Изначально выпущенная в качестве надстройки к SQL Server 2000, служба отчетности в SQL Server 2005 дополнительно включила Web-ориентированный построитель отчетов (Report Builder), позволяющий конечным квалифицированным пользователям создавать основные типы отчетов на основе заранее сконфигурированных моделей данных.

Дополнительная информация

В главах 46 и 47 описываются процессы создания и развертывания отчетов с помощью службы Reporting Services.

Служба анализа

Служба анализа (Analysis Services) решает в SQL Server задачи бизнес-аналитики, или, точнее, оперативной аналитической обработки данных (OLAP). В сущности, служба анализа позволяет разработчику определять кубы, которые в чем-то сходны со сводными таблицами Excel и перекрестными запросами Access, но способные иметь множество измерений. Кубы содержат предварительно рассчитанные сводные или консолидированные данные из особо крупных баз данных. Это позволяет пользователю просто и быстро просматривать эти итоги без необходимости выполнения "долгоиграющих" запросов к терабайтам данных (рис. 3.5).

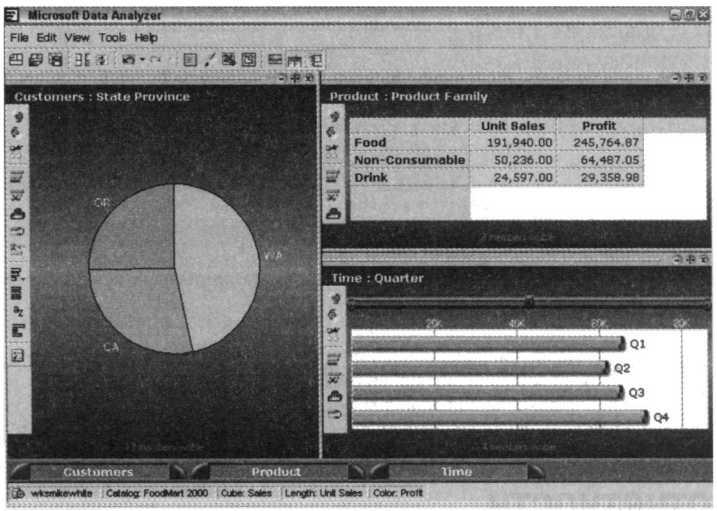


Рис. 3.5. Просмотр многомерного куба в службе анализа является упрощенным способом сравнения разнообразных аспектов данных

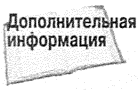
Служба анализа загружается независимо от SQL Server и рассматривается как одна из полноценных функций формирования хранилищ данных.

Дополнительная информация

В главах 43–45 рассматриваются вопросы формирования кубов, раскрытия данных и программирования запросов MDX в службе анализа.

Различные редакции SQL Server 2005

Пакет SQL Server 2005 доступен в нескольких редакциях, которые отличаются своими возможностями, требованиями к аппаратному обеспечению и, разумеется, стоимостью. В этом разделе мы в деталях рассмотрим различные редакции этого продукта и доступные в них функции. Так как условия лицензирования и стоимость продуктов компании Microsoft постоянно изменяются, с этими аспектами различных редакций SQL Server вы можете ознакомиться на сайте компании по адресу `www.microsoft.com/sql`.



Детальная информация об отличиях разных редакций SQL Server 2005 приведена в приложении А.



Достаточно легко спутать понятия *редакции* и *версии*. SQL Server 7, SQL Server 2000 и SQL Server 2005 — это версии. Редакции представляют собой различные наборы функций, включенные в эти версии, например Developer Edition или Enterprise Edition.

Enterprise (Developer) Edition

Это самая полная редакция продукта с повышенной производительностью и расширенным набором функций. Она способна поддерживать тысячи подключений и базы данных, измеряемые терабайтами. Для достижения такого уровня производительности требуются и соответствующие компьютеры — как минимум, с шестнадцатью двухъядерными процессорами, 32 Гбайт памяти и мощными сетевыми платами.

Чтобы получить полную отдачу от такой аппаратуры, редакция Enterprise Edition вобрала в себя некоторые прекрасные функции. Две из них стоит упомянуть отдельно.

- Разделение таблиц. Эта функция позволяет получить удивительные результаты производительности при использовании протокола TCP.
- Параллельное индексирование в реальном времени.

Редакция для разработчиков Developer Edition имеет тот же состав, что и Enterprise Edition, с двумя исключениями. Во-первых, Developer Edition лицензируется только для разработки приложений и их тестирования, так что легально использовать эту редакцию в производственной среде нельзя. Во-вторых, Developer Edition запускается в операционных системах, предназначенных для рабочих станций, например Windows NT Workstation, Windows 2000 Professional и Windows XP Professional. Эта редакция поставляется в составе пакета MSDN Universal и также может быть приобретена отдельно. Это самая дешевая редакция SQL Server — ее средняя цена колеблется в пределах 50 долларов.

Standard Edition

Потребности большей части средних и крупных производственных баз данных полностью удовлетворяет редакция Standard Edition. Это самая популярная редакция продукта, которая включает в себя все основные функции, в том числе службы интеграции и анализа, Web-службы, зеркальное отображение баз данных и кластеризацию.



В современном мире многоядерные процессоры в серверах стали обычным явлением. Вопрос состоит в том, как это отражается на лицензировании. В настоящее время компания Microsoft лицензирует SQL Server на основании количества процессоров, а не их ядер. Это значит, что четырехпроцессорный двухъядерный сервер будет физически функционировать как восьмипроцессорный одноядерный, но платить за лицензию придется гораздо меньшую сумму.

Standard Edition отлично подходит для предприятий среднего размера, которым нужны все предлагаемые сервером функции, но не требуется сверхвысокая доступность Enterprise Edition. Эта редакция лимитирована четырехпроцессорными компьютерами, независимо от количества ядер в процессорах, но в то же время не налагает никаких ограничений на память.

Я разговаривал со многими администраторами баз данных, компании которых приобрели Enterprise Edition, но запускают ее на серверах с четырьмя процессорами и 4 Гбайт памяти без кластеризации и прочих ультрасовременных дополнений. Учитывая производительность

современных серверов, для большинства компаний более правильным выбором было бы приобретение редакции Standard Edition. Не тратьте лишние деньги на Enterprise Edition, если только не уверены, что Standard Edition не справится с работой, возложенной на сервер.

В хорошо сконфигурированном четырехпроцессорном двухъядерном сервере, оснащенном большим объемом памяти, Standard Edition без труда сможет обслужить 500 одновременных подключений и базу данных, содержащую терабайт данных.



Если не требуются расширенные возможности Enterprise Edition, то переход на 64-разрядную версию Standard Edition и увеличение объема памяти будет лучшим решением, нежели переход на Enterprise Edition. У обеих редакций, Enterprise Edition и Standard Edition, существуют 64-разрядные версии для процессоров Intel и AMD.

Workgroup Edition

Эта редакция предназначена для серверов подразделений компаний и содержит рациональный набор функций для небольших транзакционных баз данных.

- Ядро базы данных поддерживает два процессора, максимум 3 Гбайт памяти и не имеет ограничений на размер базы данных.
- Поддерживаются некоторые функции высокой доступности, такие как доставка журналов, но не поддерживается зеркальное отображение баз данных и кластеризация.
- Включает службу отчетности, но не включает службу анализа для формирования кубов и бизнес-анализа.
- Имеет все программные возможности SQL Server 2005, включая T-SQL, хранимые процедуры, определяемые пользователем функции, полнотекстовый поиск, XML, Xquery. брокер служб (только подписку) и интеграцию .NET. Не поддерживаются Web-службы и служба уведомлений.
- Поддерживается администрирование автоматизации с помощью SQL Server Agent.
- Поддерживаются репликации транзакций и слияния.

Ключевой функцией, отсутствующей в этой редакции, является служба интеграции. При этом предполагалось, что база данных для рабочей группы скорее будет являться источником данных для других, более крупных серверов, но не будет содержать подчиненных источников данных. Это единственный фактор, который может склонить вас перейти на редакцию Standard Edition. По собственному опыту знаю, что базы данных такого масштаба иногда требуют перемещения данных, хотя бы для переноса их из баз данных Access.

Я рекомендовал бы Workgroup Edition для малых предприятий и подразделений компаний, которым не нужна сверхвысокая доступность или служба анализа. Сервер SQL Server 2005 Workgroup Edition с двумя двухъядерными процессорами, 3 Гбайт памяти и правильно спроектированной дисковой подсистемой может без труда обслуживать сто активных пользователей.

Express Edition

Express Edition нельзя рассматривать как замену ядра Access Jet. Это бесплатная, полноценная версия ядра SQL Server, предназначенная для использования с некоторым приложением. Эта редакция содержится в пакетах MSDN Universal, Office Developer Edition 11 и во всех продуктах Microsoft, предназначенных для разработчиков программ.



Редакция SQL Server Express заменила собой MSDE — бесплатную редакцию версии SQL Server 2000. Однако не следует воспринимать эту редакцию как MSDE 2005, поскольку в ней снято ограничение на количество подключений и существует полноценный интерфейс администратора.

Редакция Express имеет некоторые ограничения: максимальный объем базы данных составляет 4 Гбайт, поддерживается только один процессор (когда же AMD наконец-то выпустит 16-ядерный процессор?) и 1 Гбайт памяти. В настоящее время в этой редакции недоступна служба отчетности, однако компания Microsoft обещает включить ее в недалеком будущем.

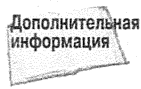


Редакция SQL Server Express имеет собственные версии утилит Management Studio и командной строки. В главе 41 описаны характерные отличия этих утилит и особенности работы с ними.

Я рекомендовал бы редакцию SQL Server Expresss Edition для небольших приложений .NET, которым нужны реальные базы данных. Они более чем удобны для приложений, предназначенных для двадцати пяти и менее пользователей и использующих менее 4 Гбайт данных. Когда служба отчетности войдет в редакцию Express, это станет реальным ударом по СУБД MySQL.

Everywhere Edition

Редакция SQL Server Everywhere, изначально называвшаяся Mobile Edition, с технической стороны представляет собой полностью отличное ядро базы данных, совместимое с SQL Server. Ее скромные потребности в памяти (всего 1 Мбайт) гарантируют ее использование на мобильных устройствах. Несмотря на то что она запускается на карманных компьютерах, это полноценное ACID-совместимое ядро базы данных.



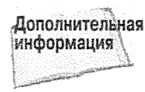
Поскольку набор программных средств редакции SQL Server Everywhere сильно отличается от полноценных редакций, в главе 26 вы узнаете о специфических требованиях, касающихся разработки и развертывания хороших клиентских реализаций SQL Server Everywhere.

Утилиты и компоненты SQL Server

Для управления SQL Server и взаимодействия с ней используются следующие компоненты и клиентские утилиты.

SQL Server Management Studio

По сути Management Studio представляет собой Visual Studio — специализированную интегрированную среду, используемую администраторами и разработчиками баз данных. Ядром этой утилиты является Object Explorer, укомплектованный фильтрами и способный обозревать все серверы в составе SQL Server (ядро базы данных, сервер анализа, сервер отчетности и т.д.). Редактор запросов (Query Editor) этой утилиты является удобным механизмом работы с кодом T-SQL; он интегрирован с обозревателем решений (Solution Explorer) для управления проектами. Несмотря на то что интерфейс этой утилиты кажется перегруженным (рис. 3.6), все окна легко конфигурируются и могут быть автоматически скрыты с экрана.



В главе 6 рассматривается все множество инструментов утилиты Management Studio, а также методы использования этого гибкого интерфейса разработки и управления.

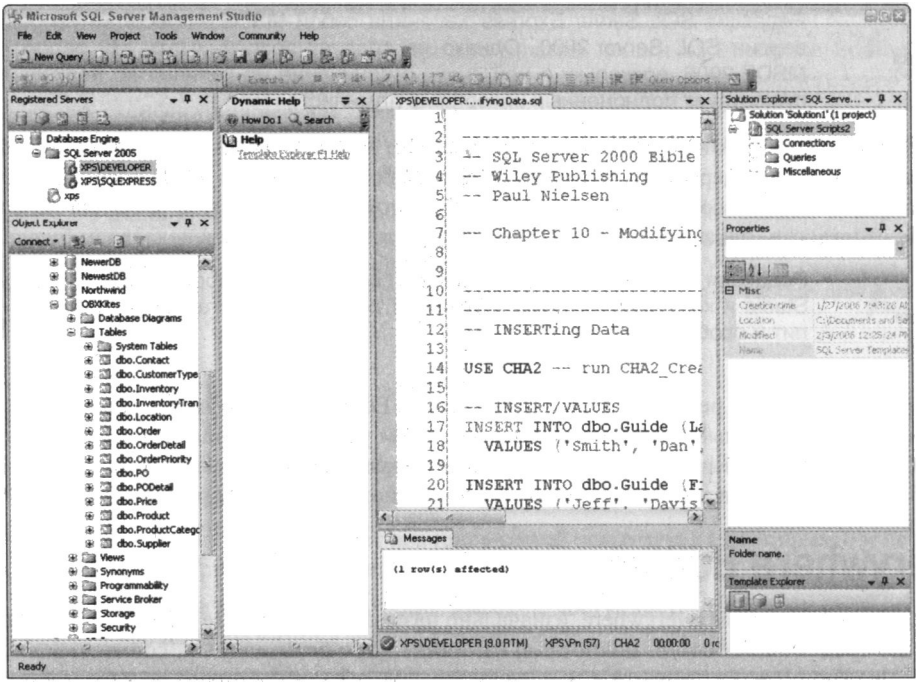


Рис. 3.6. Полный комплект окон и инструментов утилиты Management Studio может выглядеть перегруженным, однако гибкость интерфейса позволяет легко конфигурировать его по своему усмотрению

SQL Server Configuration Manager

Этот инструмент используется для запуска и остановки любого сервера, настройки параметров запуска и конфигурирования подключений (рис. 3.7). Эту утилиту можно вызвать из системного меню Пуск, а также из Management Studio.



Утилита Configuration Manager заменила собой утилиты Service Manager и конфигурирования клиента сети, существовавшие в версии SQL Server 2000.

Surface Area Configuration

В целях сокращения ресурсов, потребляемых SQL Server 2005, множество функций по умолчанию отключено. Несмотря на то что большинство этих функций можно включить с помощью инструкций T-SQL, утилита Surface Area Configuration открывает простейший путь к их конфигурированию.

Business Intelligence Development Studio

Аналогичная Management Studio, но оптимизированная под задачи бизнес-аналитики, эта утилита используется для разработки пакетов службы интеграции, отчетов служба отчетности, кубов службы анализа и раскрытия данных.

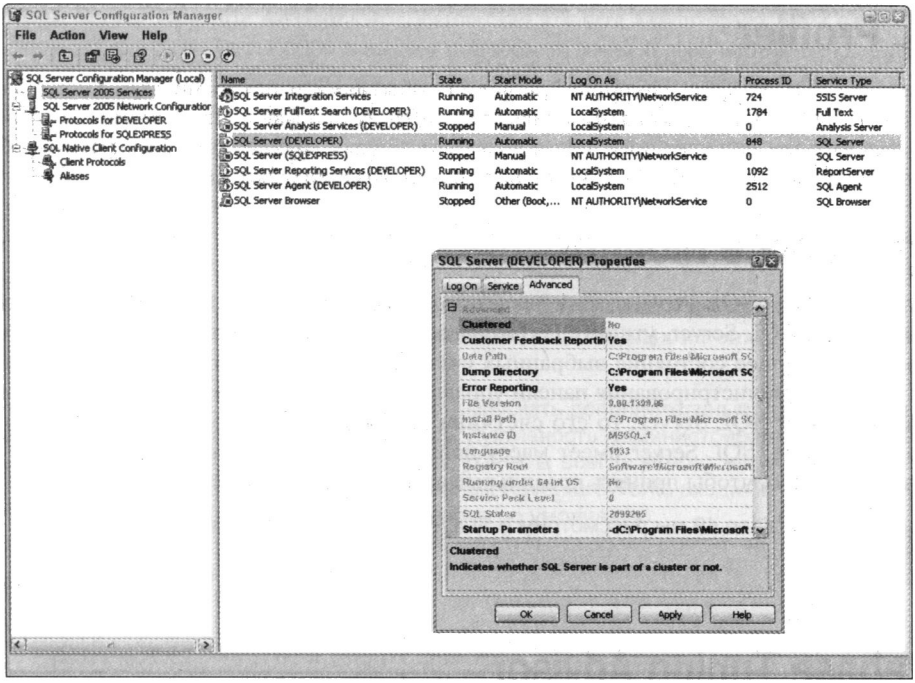


Рис. 3.7. Утилита Configuration Manager используется для включения множества серверов и управления ими

Интегрированная справка SQL

Команда программистов, сформированная для создания документации SQL Server, справилась со своей работой отлично — она создала утилиту Books Online. Это нечто большее, чем обычная справочная система: все ее статьи полностью описывают рассматриваемую тематику и включают ряд примеров. Примененная индексация позволяет получить краткий список связанных статей. Утилиту Books Online можно открыть из Management Studio или непосредственно из системного меню.

Утилита Books Online тесно интегрирована с основными интерфейсами SQL Server. После выбора ключевого слова в редакторе запросов утилиты Management Studio и нажатия комбинации клавиш <Shift+F1> откроется окно Books Online со статьей, посвященной данной теме. Аналогичным образом эту утилиту можно вызвать и из других приложений SQL Server.

Утилита Management Studio предоставляет также динамическое окно справки, которое автоматически отслеживает курсор и отображает справку по выбранному ключевому слову.

При выполнении поиска отображаются статьи MSDN, размещенные как в Интернете, так и на локальном компьютере. В дополнение к этому утилита Books Online ищет сходные по теме статьи на форуме Codezone Community.

Ссылки Community Menu и Developer Center позволяют загрузить Web-страницы, дающие пользователям возможность задать вопрос, чтобы узнать больше об SQL Server.



Компания Microsoft регулярно обновляет раздел Books Online. Самую свежую версию всегда можно загрузить с сайта `www.microsoft.con/sql`; ссылку на него я поместил на страницу этой книги по адресу `www.SQLServerBible.com`.

SQL Profiler

Утилита SQL Profiler наблюдает за всеми событиями и пакетами SQL Server, выводя выбранную информацию на экран, записывая в таблицу или файл. Эта утилита идеально подходит для отладки приложений и настройки базы данных. Функцию Database Tuning Advisor можно использовать для сбора данных с целью оптимизации базы данных.

Performance Monitor

В то время как SQL Profiler записывает большие наборы данных, касающиеся потоков SQL и событий SQL Server, утилита Performance Monitor (или System Monitor) выводит в открытое окно текущее состояние выбранных счетчиков. Утилиту Performance Monitor можно найти в папке администрирования панели управления системы Windows. Если на компьютере установлен пакет SQL Server, то его счетчики автоматически добавляются в Performance Monitor. Поверьте, SQL Server имеет массу полезных счетчиков производительности. Этого вполне достаточно, чтобы принять правильные административные решения.

В главе 49 мы рассмотрим и SQL Profiler, и Performance Monitor.

Database Tuning Advisor

Утилита Database Tuning Advisor анализирует пакет запросов (полученный из утилиты Profiler) и рекомендует изменить структуру индексов и разделов для повышения производительности. Этот пакет изменений можно легко сконфигурировать и применить полностью или частично либо сразу после анализа, либо позже.

Утилита Database Tuning Advisor подробно рассмотрена в главе 50.

Утилиты командной строки: SQLCmd и BulkCopy

Эти интерфейсы командной строки позволяют разработчику выполнить инструкции SQL или операции массового копирования из операционной системы DOS или командной строки Windows. Служба интеграции и SQL Server Agent делают эти утилиты устаревшим наследием прошлого, однако для обеспечения максимальной гибкости компания Microsoft решила включить их в состав SQL Server.

Утилита Management Studio имеет режим SQLCmd, позволяющий использовать редактор запросов так, будто он является утилитой командной строки.

Пакет дополнительных функций SQL Server 2005

Этот пакет включает в себя несколько десятков драйверов и фрагментов кода для обратной совместимости с другими системами, а также со сторонними утилитами.

- **SQL Server Upgrade Advisor**. Эта бесплатная утилита анализирует базы данных SQL Server 7 или SQL Server 2000 и выдает подробный отчет о всех вопросах, которые следует решить перед миграцией этих баз данных в SQL Server 2005.
- **Database Migration Assistant (DMA)**. Это не просто утилита, а нечто гораздо большее. Она анализирует существующую базу данных, а затем выводит рекомендации относительно ее миграции в SQL Server 2005, при этом оценивая вероятные риски. В настоящее время DMA позволяет импортировать базы данных Oracle в SQL Server, но в будущем эта операция будет расширена и на другие платформы, возможно, на MySQL и MS Access.
- **Best Practices**. Эта утилита анализирует конфигурацию сервера и его баз данных, а затем выводит отчет обо всех замечаниях, основанных на опыте работы профессионалов.



Компания Microsoft продолжает выпускать дополнительные утилиты, средства и ресурсы, которые повышают продуктивность администрирования и разработки в SQL Server. Эти бесплатные утилиты всегда можно загрузить с сайта `www.microsoft.com/sql`. База данных AdventureWorks входит в инсталляцию SQL Server 2005, но не устанавливается по умолчанию.

AdventureWorks

`AdventureWorks` — это база данных примеров, поставляемая в пакете SQL Server 2005. Она заменила собой существовавшие ранее базы `Northwind` и `Pubs`. В то время как база `AdventureWorks` имеет преимущества в объеме перед маленькими `Northwind` и `Pubs`, она страдает от довольно сложной схемы. Мнения сообщества пользователей SQL Server о ней нельзя назвать однозначными. Иногда ее даже называют *AdventureWorst* (наихудшее приключение).



При желании вы можете загрузить из Интернета сценарии, создающие базы данных `Northwind` и `Pubs` для SQL Server 2005.

`www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53a68034&displaylang=en`

Соответствующую ссылку вы найдете и на сайте автора по адресу `www.SQLServerBible.com`.

В области бизнес-аналитики база данных `AdventureWorksDW` заменила базу `FoodMart` и была довольно хорошо воспринята сообществом пользователей.

Обзор метаданных

После изначальной установки SQL Server уже содержит некоторые системные и пользовательские объекты. Для самообслуживания SQL Server использует четыре системные базы данных и две базы данных примеров для экспериментов пользователей. При этом каждая из баз данных содержит несколько системных объектов, включая таблицы, представления, хранимые процедуры и функции.

В утилите Management Studio эти системные объекты могут быть скрыты. На странице Registered SQL Server Properties вы можете выбрать, что можно отображать, с помощью параметра Show system database and system objects.

Системные базы данных

SQL Server использует четыре системные базы данных для хранения системной информации, отслеживания операций и поддержки временной рабочей области. В дополнение база данных моделей может служить прототипом для новых пользовательских баз. Пять системных баз данных описаны ниже.

- **`Master`**. Содержит информацию обо всех базах данных, размещенных на сервере. К тому же объекты базы `Master` доступны и для остальных бах данных. Например, хранимые процедуры базы данных `Master` можно вызвать из любых пользовательских баз.
- **`MSDB`**. Содержит список действий, таких как резервное копирование и выполнение заданий, а также определяет порядок резервного копирования всех пользовательских баз данных.
- **`Model`**. Это база данных шаблонов, на основе которой создаются новые базы данных. Любой объект, помещенный в базу данных `Model`, будет скопирован во все новые базы данных.
- **`Tempdb`**. Используется для хранения временных таблиц, создаваемых пользователями, пакетами, хранимыми процедурами (включая системные) и самим ядром SQL Server. Если серверу необходимо создать временные кучи или списки в ходе выполнения запроса, то все они создаются именно в базе данных `Tempdb`. Эта база данных полностью очищается при перезапуске SQL Server.
- **`Reference`**. Эта скрытая база данных является разделом базы данных Master и размещена в том же каталоге. Она предназначена для облегчения установки пакетов обновлений.

Представления метаданных

Метаданными называют общую информацию о самих данных. Одним из исходных правил доктора Кодда для реляционных баз данных было хранение информации о базе данных в самой базе с помощью таблиц, строк и столбцов, что ничем не отличается от хранения данных пользователя. Эта информация о данных облегчает программную навигацию по схеме базы и ее конфигурации. SQL Server 2005 имеет несколько типов метаданных: представления каталогов, представления динамического управления и два представления системных таблиц.



Функции метаданных в SQL Server 2005 были полностью пересмотрены и теперь представляют пользователю гораздо больше информации, чем в предыдущих версиях.

- **Представления каталогов** содержат информацию о статических метаданных, таких как таблицы, а также конфигурация системы безопасности и самого сервера.
- **Динамические представления управления** и **функции** позволяют заглянуть в текущее состояние сервера в данную секунду и увидеть информацию о таких характеристиках, как память, потоки, хранимые процедуры в кэше и подключения.
- **Представления совместимости** позволяют имитировать системные таблицы такими, какими они были в предыдущих версиях SQL Server.
- **Представления информационной схемы** не поддерживаются стандартом ANSI SQL-92 и служат для обзора схемы любого продукта базы данных. Это представление используется в крайнем случае и имеет малое практическое применение у администраторов баз данных и разработчиков, которые исследуют возможности SQL Server.

Резюме

SQL Server является сложным и емким продуктом, который обслуживает потребности транзакционных баз данных и бизнес-аналитику. Эта глава позволяет получить общее представление о том, что будет детально рассмотрено далее.

Следующие несколько глав посвящены установке и конфигурированию пакета SQL Server, подключению его к клиентским приложениям, обзору утилиты Management Studio, а также интерфейса администраторов и разработчиков баз данных.

ГЛАВА

4

Установка SQL Server 2005

В этой главе...

Планирование установки

Рекомендации относительно аппаратного и программного обеспечения

Установка множества экземпляров SQL Server

Обновление предыдущих версий SQL Server

Миграция баз данных SQL Server

Конфигурирование области поддержки SQL Server

Несмотря на то что установить и заставить работать SQL Server относительно просто, среда и методы, которые следует заложить при установке этой СУБД, не так прозрачны. Дополнительное время, затраченное на постановку этих и других вопросов, и поиск ответов на них сполна окупятся успешной инсталляцией.

Хотя эта глава фокусирует внимание на основных аспектах установки SQL Server, в ней также пойдет речь и об обновлении версии СУБД.



SQL Server 2005 можно использовать параллельно с другими версиями сервера.

Планирование установки

Подумайте немного об аппаратном обеспечении сервера. Примите во внимание учетные записи пользователей, конфигурацию дисковой системы и особенности доступа клиентов. Если вы обновляете версию сервера, возьмите на вооружение утилиту Upgrade Advisor Tool, о которой мы поговорим немного позже в этой главе.

Операционная система

SQL Server 2005 можно устанавливать на множестве операционных систем и их редакций — от Windows XP Home Edition до Windows Server 2003 Enterprise Edition. В табл. 4.1 приведены операционные системы, совместимые с конкретными редакциями SQL Server 2005.

Таблица 4.1. Операционные системы, поддерживаемые SQL Server 2005

Редакция SQL Server	Windows XP Home Edition (SP2+)	Windows Server 2003 Web Edition	Windows 2000 Professional (SP 4+)	Windows XP Tablet Edition	Windows XP Media Edition	Windows XP Professional	Windows 2000 Server, Advanced и DataCenter Server (SP 4+)	Windows Small Business Server 2003 Standard и Premium Edition	Windows Server 2003? Standard, Enterprise и DataCenter Edition
Enterprise / Standard Edition							Да	Да	Да
Workgroup / Evaluation Edition			Да	Да	Да	Да	Да	Да	Да
Developer / Express Edition	Да	Да	Да	Да	Да	Да	Да	Да	Да

Вопросы безопасности

Для доступа к SQL Server необходимо наличие у пользователя соответствующей учетной записи для аутентификации. При использовании аутентификации Windows SQL Server запрашивает регистрационные данные пользователя у операционной системы. Для регистрации пользователей в Windows используется подсистема Учетные записи пользователей панели управления. В качестве альтернативы в SQL Server можно создать собственные регистрационные записи. Для упрощения работы старайтесь, когда это возможно, использовать для управления доступом к SQL Server аутентификацию Windows.

Аутентификация Windows открывает пользователям доступ к базе данных, проверяя их регистрационные данные относительно учетных записей операционной системы. Это позволяет централизовать включение, отключение и применение особых настроек системы безопасности, связанных с учетными записями пользователей. Все эти настройки автоматически применяются к регистрационным записям, управляющим доступом к базе данных. Если какой-либо работник увольняется из организации, то отключение его учетной записи Windows автоматически закрывает ему доступ и к базе данных.



Компания Microsoft рекомендует, когда это возможно, использовать для проверки пользователей аутентификацию Windows.

Процессы SQL Server также требуют наличия учетных записей Windows. Они могут быть определены независимо от других во время инсталляции выбором пункта Customize for each service account. По умолчанию SQL Server, SQL Server Agent, Analysis Server и SQL Browser используют одну и ту же учетную запись. При установке проверьте, существуют ли у этой учетной записи достаточные привилегии для доступа к соответствующим файлам и ресурсам. Зависимость учетных записей и служб приведена в табл. 4.2.

Таблица 4.2. Устанавливаемые учетные записи для служб SQL Server

Название службы	Учетная запись по умолчанию	Дополнительные учетные записи
SQL Server	SQL Express в Windows 2000 — local system	SQL Express — domain user, local system, network service, local service
	SQL Express во всех остальных операционных системах — local service	Все остальные редакции — domain user, local system, network service
	Все остальные редакции во всех операционных системах — domain user	
SQL Server Agent	Domain user	Domain user, local system, network service, local service
Analysis Services	Domain user	Domain user, local system, network service, local service
Reporting Services	Domain User	Domain user, local system, network service, local service
Notification Services	—	—
Integration Services	Windows 2000 — local system	Domain user, local system, network service, local service

Окончание табл. 4.2

Название службы	Учетная запись по умолчанию	Дополнительные учетные записи
	Все остальные операционные системы — network service	
Full-Text Search	То же, что и SQL Server	Domain user, local system, network service, local service
SQL Browser	Domain user	Domain user, local system, network service, local service
SQL Server Active Directory Helper	Network Service	Local system, network service
SQL Writer	Local system	Local system



Создание отдельной учетной записи для служб SQL Server обеспечит большую защиту и надежность, так как при этом можно снабдить ее минимальным набором необходимых разрешений, а недоступность пароля для обычных пользователей не позволит им воспользоваться учетной записью службы для удовлетворения собственных потребностей.

Если инсталляция включает в себя серверы, которые будут взаимодействовать с данным и участвовать в обработке распределенных запросов, нужно использовать учетную запись уровня домена, при этом данная доменная учетная запись должна быть членом группы Windows Administrators.

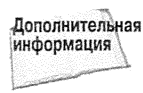
Режим аутентификации

SQL Server предполагает использование двух различных режимов аутентификации: регистрационные записи SQL Server и учетные записи Windows. В первом случае учетные записи создаются и управляются в самой СУБД. Эти записи могут быть использованы только для доступа к серверу баз данных и ни к какому другому серверу сети. Учетные записи Windows создаются для сетевого доступа и могут быть использованы также для доступа к серверу баз данных. Использование для доступа к SQL Server обоих методов называют *смешанным режимом*. В табл. 4.3 описаны эти два режима аутентификации.

Таблица 4.3. Режимы аутентификации

Метод регистрации	Режим аутентификации Windows	Смешанный режим
Пользователи могут регистрироваться с помощью учетной записи Windows	Да	Да
Специальные регистрационные записи SQL Server	Нет	Да

При использовании смешанного режима назначайте административной записи сложный пароль. Такой пароль, как правило, состоит из комбинации символов верхнего и нижнего регистра, цифр и специальных символов. Если не задать хороший пароль или вообще опустить его, то в системе безопасности образуется большая брешь.



Подробно о системе безопасности баз данных мы поговорим в главе 40.

Экземпляры SQL Server

СУБД SQL Server 2005 способна поддерживать до пятидесяти своих экземпляров, запущенных на одном физическом сервере, в том числе и экземпляры разных редакций (Enterprise, Standard или Developer). Возможность запуска множества экземпляров становится особенно ценной, когда используется в пространстве с ограниченным количеством физических серверов, в частности при тестировании совместимости продуктов, влияния пакетов обновлений, репликаций и т.п. Для оптимизации производительности сервера при тестировании запускайте только необходимые экземпляры SQL Server. После завершения тестирования не забывайте останавливать ненужные экземпляры сервера. Использование множества экземпляров сервера в производственной среде вообще не рекомендуется ввиду ограничений производительности и доступности.



Запуск множества экземпляров SQL Server на одном физическом сервере негативно сказывается на производительности. Каждый экземпляр потребляет ресурсы, такие как память и циклы работы процессора, для обработки запросов. При использовании процессоров с множеством ядер и потоков вопросы производительности не имеют такой остроты, но все же использование одного сервера для обслуживания нескольких баз данных является лучшим решением.

По умолчанию SQL Server и ассоциированные файлы устанавливаются в каталог `c:\Program Files\Microsoft SQL Server\MSSQL.#` (в данном пути знак решетки заменяет следующее число в последовательности). Например, SQL Server займет один номер, а сервер анализа — следующий за ним. Таким образом, если на компьютере установлены и SQL Server, и Analysis Server, структура каталогов будет следующей:

- `C:\Program Files\ Microsoft SQL Server\MSQL.1` (SQL Server)
- `C:\Program Files\ Microsoft SQL Server\MSQL.2` (Analysis Server)

Если будет установлен еще один сервер, то он займет в последовательности следующий номер.

Первый установленный экземпляр SQL Server становится экземпляром, используемым по умолчанию; он будет иметь то же имя, что и сам сервер. Каждый дополнительный экземпляр будет устанавливаться с уникальным именем, максимальная длина которого составляет 16 символов.

Не все установленные службы совместно используются всеми экземплярами сервера. В табл. 4.4 представлен список служб, совместно используемых разными экземплярами. Специфичные для каждого экземпляра службы имеют собственные устанавливаемые компоненты.

Таблица 4.4. Совместно используемые службы SQL Server

Служба	Используется совместно	Специфичная для экземпляра
Notification Services (служба уведомлений)	Да	
Data Transformation Services (служба преобразования данных)	Да	
SQL Browser	Да	
SQL Server Active Directory Helper	Да	
SQL Writer	Да	
SQL Server		Да
SQL Server Agent		Да
Analysis Server (сервер анализа)		Да
Report Server (сервер отчетности)		Да
Full-Text Search (полнотекстовой поиск)		Да

Рекомендации относительно аппаратной части

Стоимость оборудования в настоящее время достаточно низкая и продолжает снижаться с каждым днем. Теперь оборудование уже не является сдерживающим фактором, и правильный выбор устройств и конфигурации способен значительно повысить эффективность капиталовложений. В этом разделе приведено несколько рекомендаций относительно планирования аппаратного обеспечения сервера.

Выделенный сервер

SQL Server 2005 динамически управляет памятью, основываясь на своих текущих потребностях, а также потребностях операционной системы и других приложений. Когда запускаются другие приложения и начинают потреблять память, серверу может ее не хватить для обработки запросов от потенциально больших групп пользователей, что вызовет отключение некоторых запросов из-за превышения времени ожидания.

Предполагая необходимость постоянного доступа к базе данных (24 часа в день и 7 дней в неделю), перегруженность сервера другими, пусть даже связанными приложениями, может иметь негативное влияние на производительность. К тому же, чем больше приложений запущено на сервере, тем выше риск того, что их обновление может иметь побочный эффект в смысле снижения производительности и доступности SQL Server.

Наличие выделенного сервера может ослабить остроту этих вопросов, а в некоторых случаях предпочтительнее иметь для этих целей даже несколько физических выделенных серверов.

Потребности в памяти

SQL Server 2005 динамически захватывает и высвобождает оперативную память по мере необходимости и выполняет балансировку области памяти, используемой для обработки запросов. Сервер способен каждую секунду резервировать и высвобождать несколько мегабайтов памяти.

Такие редакции SQL Server 2005, как Standard, Workgroup, Express и Evaluation, способны использовать до 4 Гбайт оперативной памяти. Редакции Enterprise и Developer имеют более высокий предел за счет использования расширенной адресации Address Windowing Extensions (AWE).

Редакции Enterprise и Developer СУБД SQL Server 2005 могут использовать AWE в следующих 32-разрядных операционных системах: Windows XP Professional, Windows 2000 Standard Edition. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows Server 2003 Enterprise Edition и Windows Server 2003 Datacenter Edition.

Если редакции Enterprise и Developer способны удовлетворить определенные потребности производства и приложений, то наличие памяти свыше 4 Гбайт принесет свои плоды. В этом случае пул буфера может находиться в размеченном AWE пространстве памяти, высвобождая таким образом нижние адреса для использования. Во всех остальных случаях наличие более 4 Гбайт памяти нисколько не повысит производительность SQL Server, хотя доступные 4 Гбайт будут использоваться настолько, насколько это возможно.



Для активизации AWE СУБД SQL Server 2005 должна быть запущена с учетной записью с установленным параметром блокировки страниц в памяти (Lock Pages In Memory) и параметром `AWE Enabled`, который имеет значение `1`. Все эти настройки можно выполнить с помощью системной хранимой процедуры `sp_configure`.

Использование множества процессоров

Традиционно стоимость лицензирования зависит от количества установленных в компьютере процессоров, достигая порой нескольких тысяч долларов. Расширенные технологии, используемые в процессорах (такие как множество потоков или ядер), могут существенно повысить производительность сервера, при этом совершенно не влияя на стоимость лицензирования.

Компания Intel впервые представила технологию поддержки множества потоков в процессорах (HyperThreading, HT) в 2002 году. Эта технология позволяет процессорам одновременно обрабатывать множество потоков в одном ядре за счет общего использования ресурсов выполнения. Многопотоковые приложения, такие как SQL Server, могут выполнять на таких процессорах одновременно несколько операций, при этом сокращая время отклика системы. Для операционной системы многопотоковый процессор представляется как два разных процессора в одном физическом гнезде.



В то время как компания Intel повышала производительность процессоров с помощью технологии многопотоковости, компания AMD решала те же вопросы с помощью технологии гипертранспорта (HyperTransport). Эта технология сокращает задержку и повышает скорость взаимодействия интегрированных цепей в компьютерах и других электронных устройствах, в том числе и периферийных. Технология HyperTransport реализует механизмы с общей производительность до 22,4 Гбайт/с, повышая в 70 раз пропускную способность стандартных устройств PCI. Несмотря на то что технология HyperTransport в целом связана не только с процессором, общая производительность системы заметно повышается. Процессоры и системы, поддерживающие эту технологию, увеличивают производительность за счет повышения пропускной способности каналов данных. Эта технология в настоящее время поддерживается следующими марками процессоров компании AMD: Opteron, Athlon 64/FX, Sempron, а также двухъядерными процессорами.

Как компания AMD, так и Intel в качестве первого шага в технологии многоядерности выпустили двухъядерные процессоры. Эти процессоры содержат два ядра выполнения инструкций в одном физическом корпусе. Если программное обеспечение поддерживает параллельные вычисления, то операционная система рассматривает каждое из этих ядер как отдельный процессор с собственными ресурсами выполнения инструкций. Политика лицензирования в компании Microsoft до сих пор основана на количестве микросхем процессоров, независимо от числа ядер и наличия многопотоковости. Исходя из этого, если запустить SQL Server в системе с одним процессором, лицензировать нужно только его. Если тот же сервер запустить в системе с многопотоковостью или несколькими ядрами, то лицензировать придется все тот же один процессор как физическую единицу. Преимущества многопотоковых и многоядерных процессоров сказываются не только на повышении скорости работы, но и на сокращении средств, необходимых для лицензирования.



Сравнивая выгоду от приобретения нового сервера и модернизации процессора, принимайте во внимание в качестве решения либо многопотоковые, либо многоядерные процессоры, так как вы существенно выиграете в производительности без дополнительных затрат на лицензирование SQL Server. В многоядерном процессоре каждое ядро обладает собственными ресурсами выполнения, в то время как в многопотоковых ресурсы выполнения совместно используются несколькими процессами.

Дисковая подсистема

Устройства с интерфейсом малых компьютерных систем SCSI продолжают доминировать в промышленных системах благодаря своей исключительной пропускной способности. Серверу для обеспечения своих потребностей лучше использовать такую дисковую систему. Если

серверу не требуется высокая пропускная способность, то в качестве более экономичной альтернативы можно рассмотреть устройства SATA. Нужно принять во внимание все потенциальные потребности роста и развития системы — это поможет вам правильно выбрать тип устройств для дисковой подсистемы.

Множество производителей интегрируют RAID-контроллеры на материнских платах, которые также поддерживают и устройства SATA.

Дисковые RAID-массивы

Аббревиатура *RAID* расшифровывается как *избыточный массив независимых/недорогих дисков*. Это категория дисковых устройств, в которых задействовано два и более привода в комбинации с повышенной производительностью и защитой от сбоев. RAID-массивы обычно можно найти в высокопроизводительных дисковых системах, используемых серверами для повышения надежности хранения данных и их доступности. В табл. 4.5 описаны различные уровни RAID-массивов.

Таблица 4.5. Уровни RAID-массивов

Уровень RAID	Процент избыточности	Описание
0	0	*Полосование данных*. Данные распределяются по нескольким устройствам, повышая скорость операций чтения и записи. Основными характеристиками являются отсутствие проверки на четность, избыточности и отказоустойчивости
1	50	*Зеркальное отображение данных*. Данные записываются на два устройства и считываются с каждого из них, что обеспечивает защиту от сбоев
5	Зависит от количества устройств. Если используется пять устройств и одно для проверки четности, то избыточность составляет 20%	Полосование данных с битом четности, записываемым на одно из устройств. Благодаря биту четности, при сбое любого из устройств массива система будет продолжать функционировать. Когда сбойный диск будет заменен, на нем будут воссозданы данные, содержавшиеся на вышедшем из строя диске. RAID 5 популярен благодаря обеспечению высокой надежности хранения информации, высокой скорости работы с данными и низкой себестоимости
0+1	50	*Зеркальное отображение и полосование*. Объединяет в себе высокую скорость доступа к данным и отказоустойчивость. В то же время это одно из самых дорогостоящих решений


Проверено

Несмотря на то что существуют программные решения, имитирующие поддержку RAID-массивов, все они не такие эффективные, как аппаратные. Программные решения связывают циклы работы процессора для выполнения операций поддержки RAID, используемых в серверной обработке данных.

Производительность сети

Современные материнские платы содержат встроенные карты сетевого интерфейса, способные автоматически переключаться между скоростями 10, 100 и 1000 Мбит/с. Как и большинство других встроенных устройств, они имеют тенденцию потреблять некоторые ресурсы процес-

сора, что снижает общую производительность системы. Множество производителей предлагают обособленные сетевые карты, которые берут на себя обработку пакетов, высвобождая процессор. Такое решение способно повысить общую производительность как сети, так и сервера.

Установка пакета

Как только план установки будет готов и вы удостоверитесь, что сервер удовлетворяет техническим требованиям SQL Server, можно приступать к установке пакета.

Сопровождаемая установка

Программа инсталляции SQL Server должна запуститься автоматически сразу же после запуска соответствующего DVD. При этом на выполнение запускается программа `Setup.exe`.



Антивирусное программное обеспечение может изначально блокировать выполнение файла `Splash.hta`. Если разрешить его одноразовое выполнение, то установка будет продолжена в обычном режиме. Если антивирусная программа так и не позволила запустить данный файл, отключите ее перед началом установки пакета.

Программа SQL Server Component Update установит компоненты, необходимые для продолжения установки SQL Server, а именно Windows Installer 2.0, среду .NET Framework и дополнительные файлы поддержки инсталляции SQL Server.

Программа System Configuration Check откроет страницу состояния выполнения технических требований, показанную на рис. 4.1. Возможны три варианта: *Success*, *Warning* и *Error*.

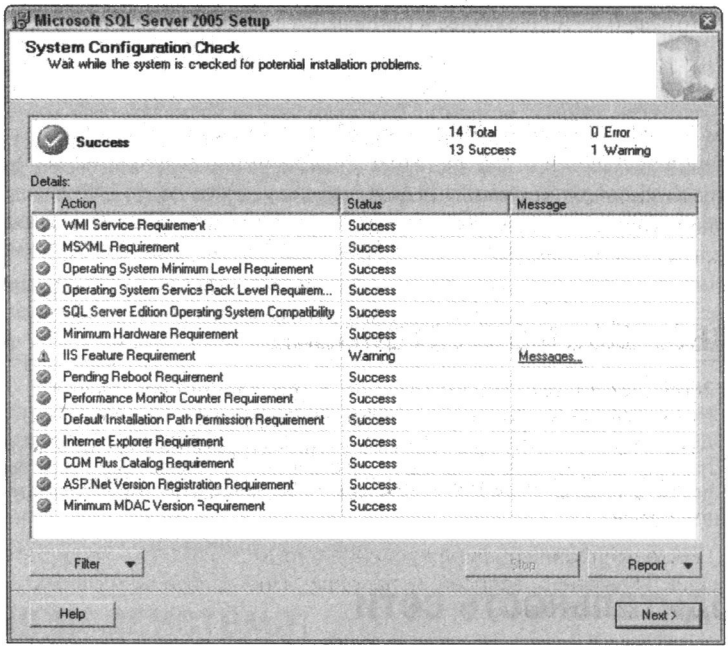


Рис. 4.1. В диалоговом окне System Configuration Check *отображаются результаты проверки соответствия компьютера системным требованиям SQL Server*

Чтобы просмотреть дополнительную информацию о предупреждениях и ошибках, выберите в меню Report пункт View Report.

Для выполнения сопровождаемой установки выполните следующее.

1. Выберите устанавливаемые компоненты. Варианты установки *Typical*, *Minimum* и *Custom*, существовавшие в версии SQL Server 2000, теперь недоступны. В версии SQL Server 2005 нужно явным образом выбирать устанавливаемые компоненты. Если необходимо изменить путь установки или состав подкомпонентов, щелкните на кнопке Advanced.

   В ходе некластеризованной установки SQL Server вариант *Virtual Server* будет отключен для служб SQL Server Database Services и Analysis Services. Эти параметры активны только при установке пакета на кластер.

2. Введите имя экземпляра. Если дополнительные серверы уже установлены, введите имя экземпляра во время установки, в противном случае оставьте имя, предложенное по умолчанию. Щелкните на кнопке Installed Instances, чтобы отобразить серверы, уже установленные на компьютере.

3. Выберите настройку Service Account. Ко всем устанавливаемым службам может быть применена одна учетная запись домена и локального компьютера; при желании каждой службе можно выделить отдельную учетную запись.



Если для службы выбирается локальная учетная запись, проверьте, достаточны ли ее привилегии.

4. Выберите метод аутентификации (Authentication Mode). При определении смешанного режима для учетной записи `sa` следует дополнительно ввести пароль. Обязательно вводите этот пароль, иначе подвергнете свой сервер существенному риску.

5. Установите режим упорядочения (Collation). Выбранный режим может быть применен как к SQL Server, так и к Analysis Server, как вместе, так и по отдельности.

6. Установите флажки Error и Usage Report. Это позволит автоматически отправить отчет об ошибках и данные об использовании функций в компанию Microsoft.

7. Щелкните на кнопке Install, чтобы начать установку SQL Server 2005. Все вопросы, возникающие в ходе установки, будут записаны в файлы журналов, которые можно просмотреть после ее завершения.

Несопровождаемая установка

SQL Server 2005 следует устоявшейся традиции, предлагая инсталляцию на основе файла конфигурации `.ini`. Хорошо прокомментированный пример конфигурационного файла называется `template.ini`, и его можно найти в корневом каталоге установочного диска SQL Server 2005. Этот файл содержит раздел `[Options]`, в котором можно настроить конкретную инсталляцию пакета.



Текстовый файл конфигурирования инсталляции `.ini` не содержит какой-либо защиты учетных записей и паролей. Постарайтесь максимально ограничить доступ к этому файлу.

В следующем примере команды показан синтаксис запуска несопровождаемой инсталляции:

```
Setup.exe /параметры <полный_путь_к_файлу_INI>
```

Предположим, что мы используем файл `mySQLSettings.ini`, размещенный в каталоге `SQLTemp`. В этом случае командная строка запуска установки будет иметь следующий вид:

```
Setup.exe /параметры c:\SQLTemp\MySQLSettings.ini
```

На режим установки влияют следующие параметры.

- `/qn`. Позволяет выполнить "тихую" установку, без открытия диалоговых окон.
- `/qb`. Позволяет отображать только диалоговые окна хода установки.

После того как будет создан файл конфигурации установки, его можно использовать как для несопровождаемой, так и для удаленной инсталляции.

Удаленная установка

Пакет SQL Server 2005 может быть установлен на удаленный сетевой компьютер. Удаленная установка может использовать тот же файл `.ini`, что и несопровождаемая. Однако в этом файле должны содержаться три дополнительных значения (табл. 4.6).

Таблица 4.6. Обязательные параметры удаленной установки

Параметр	Описание
`TargetComputer`	Имя сетевого компьютера, на который выполняется установка SQL Server
`AdminAccount`	Учетная запись администратора удаленного сервера, на который устанавливается SQL Server
`AdminPassword`	Пароль учетной записи администратора удаленного сервера



Удаленная установка может выполняться только в доменной среде, но не на компьютеры рабочей группы.

Установка на кластер

При установке SQL Server 2005 на кластер в диалоговом окне выбора устанавливаемых компонентов SQL Server и(или) Analysis Server установите флажок Install as a Virtual Server. Эти флажки будут доступны только в кластерной среде. Мастер установки после этого запросит дополнительную информацию о конфигурации, относящуюся к кластеру установки.



СУБД SQL Server 2005 может быть установлена на кластеры, содержащие до восьми компьютеров.

Убедитесь, что используемая административная учетная запись домена имеет достаточно прав и что для устанавливаемых служб выбрана доменная учетная запись.

Установка множества экземпляров

SQL Server 2005 позволяет запустить на одном физическом компьютере до шестнадцати экземпляров сервера. Наличие множества экземпляров имеет смысл, когда сторонние производители ориентируются на разный состав пакетов обновлений или разные уровни защиты.

Для установки дополнительных экземпляров перезапустите программу инсталляции сервера, но не принимайте предложенное по умолчанию имя сервера, а введите другое. Однако учтите, что за запуск нескольких экземпляров SQL Server придется расплачиваться. Каждый экземпляр требует выделения отдельного блока памяти и должен делить другие ресурсы с остальными экземплярами.

Тестирование установки

Лучшим способом тестирования установки является подключение к экземпляру SQL Server с помощью утилиты SQL Server Management Studio и просмотр доступных баз данных.

Если доступ к SQL Server должен осуществляться программным путем, потребуются дополнительные действия. Если вы подключаетесь с помощью библиотеки ADO.NET, создайте проект, который импортирует пространство имен `System.Data.SqlClient`, и напишите соответствующий программный код для подключения и получения тестового набора данных. Если для подключения используется SQL Native Client, установите динамические библиотеки этого клиента с помощью файла `sqlncli.msi`, включенного в состав установочного компакт-диска SQL Server. После этой установки создайте проект Visual Studio и напишите соответствующий тестовый программный код.



Более подробно о клиенте SQL Native Client речь пойдет в главе 5.

Обновление предыдущих версий

SQL Server 2005 включает поддержку обновления версий SQL Server 7 и SQL Server 2000. Для версий SQL Server 6.5 и более ранних обновление не поддерживается; вначале они должны быть обновлены до версии SQL Server 2000 и только затем до SQL Server 2005. Перед выполнением любого обновления запустите утилиту Upgrade Advisor и получите информацию об эффекте, к которому может привести обновление.

Утилита SQL Server 2005 Upgrade Advisor

Утилита SQL Server 2005 Upgrade Advisor проверяет существующую установку SQL Server 7.0 или SQL Server 2000 на возможность потенциальных помех, которые могут возникнуть при их обновлении до SQL Server 2005. Этой проверке подлежат ядро базы данных, служба анализа, служба уведомлений, служба отчетности и служба преобразования данных. Дополнительную информацию о программе Upgrade Advisor и вопросах обновления можно получить, воспользовавшись ссылками на главной странице утилиты. В табл. 4.7 перечислены системные требования, выдвигаемые утилитой Upgrade Advisor.

Таблица 4.7. Системные требования утилиты Upgrade Advisor

Windows 2000 SP4+, Windows XP SP2+ или Windows Server 2003 SP1+
Windows Installer 3/1+
.NET Framework 2.0; эта среда содержится на установочном диске пакета SQL Server 2005
SQL Server 2000 DSO (decision support objects) для сканирования службы анализа
Компоненты SQL Server 2000 Client для сканирования пакетов службы преобразования данных (DTS)



Утилиту Upgrade Advisor вы найдете на установочном компакт-диске SQL Server. Она инсталлируется независимо от самого сервера.

Утилита Upgrade Advisor состоит из двух компонентов: Analysis Wizard и Report Viewer. Запустите мастер Analysis Wizard для анализа существующих установок SQL Server. Для автоматического определения серверных компонентов можно сконфигурировать локальные и удаленные серверы. Может быть выполнен анализ одной, избранных или всех баз данных и пакетов DTS. На рис. 4.2 показано диалоговое окно процесса выполнения после его запуска.

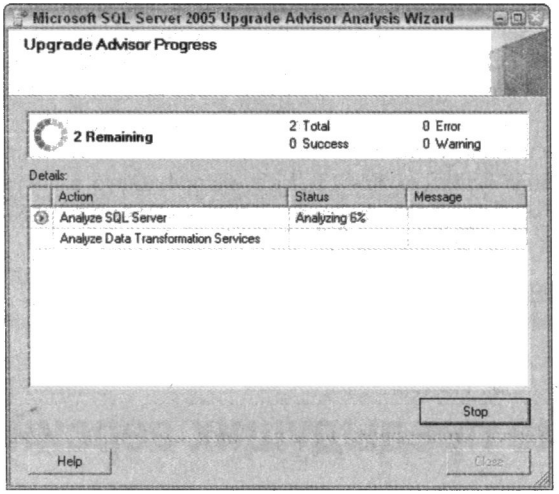


Рис. 4.2. Диалоговое окно индикатора хода выполнения анализа



Утилита Upgrade Advisor переписывает результаты всех предыдущих анализов заданного сервера. Если нужно сохранить какие-либо файлы анализа, переместите их в другую папку или переименуйте.

Ошибки, выявленные утилитой Upgrade Advisor, следует исправить. Запустите компонент Report Viewer, чтобы получить более подробную информацию о результатах анализа. В отчете утилиты подробно описываются характер выявленных ошибок, а также пути их устранения (рис. 4.3).

Несколько параметров позволяют отфильтровать список отображаемых ошибок. Можно выбрать сервер, конкретный экземпляр или компонент и статус. Отчет также поддерживает отслеживание разрешенных вопросов. Как только какой-либо вопрос будет решен, установите флажок This issue has been resolved.

Отчет при желании можно экспортировать в текстовый файл. Отдельные элементы отчета в нем будут разделены запятыми. Для этого щелкните на ссылке Export Report.

Обновление версии SQL Server 2000

До версии SQL Server 2005 могут быть обновлены ядро базы данных, служба анализа миграций (Migration Analysis Services), служба отчетности (Reporting Services), служба уведомлений (Notification Services), а также служба преобразования данных (Data Transformation

Services). В то время как некоторые из этих компонентов должны сосуществовать, другим это не обязательно. В табл. 4.8 показано, как могут устанавливаться эти компоненты.

При обновлении SQL Server 2000 до версии 2005 был улучшен оптимизатор запросов ядра базы данных, чтобы соответствовать процессу обновления сторонних продуктов, которые используют SQL Server в качестве клиентской базы данных.

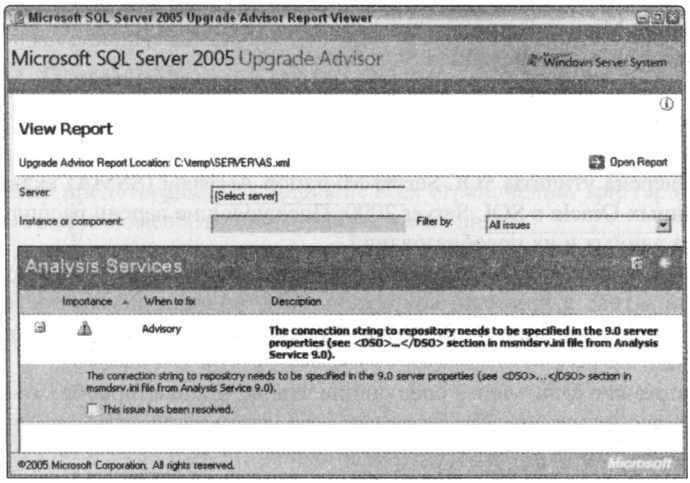


Рис. 4.3. Окно просмотра отчета утилиты Upgrade Advisor

Таблица 4.8. Варианты обновления компонентов

Компонент	Бок о бок		Требуется миграция при обновлении
Ядро базы данных	Да	Да	Нет
Служба анализа миграций	Да	Да	Да
Служба отчетности	Нет	Да[1]	Да[2]
Служба уведомлений	Нет	Нет	Да[3]
Служба преобразования данных	Да	Да	Да

[1] Это обновление прозрачно, если при установке не вносятся изменения.

[2] При установке на измененную, не принятую по умолчанию инсталляцию. В противном случае миграция не требуется.

[3] Миграцию следует выполнять, когда установлено ядро SQL Server 2005 и служба уведомлений.

Если требуется доступ к компонентам SQL Server 2000 и его данным, то установку SQL Server 2005 следует выполнять при наличии установленной версии SQL Server 2000.



Если вы обновляете версию SQL Server 2000, то обновляйте вначале клиентский и целевой серверы баз данных, чтобы минимизировать повреждение данных на первичных серверах.

Миграция в SQL Server 2005

В течение жизненного цикла данных существуют определенные моменты, когда преобразование данных в новую базу просто необходимо. В таких ситуациях следует идентифицировать возможности новой базы данных, ее требования, а также потребности производства. Если поддержка миграции данных достаточно очевидна, то длительные процессы начнут преобразовывать данные, их схемы и бизнес-логику в новую базу данных. Чтобы облегчить эти процессы, компания Microsoft предлагает помощь утилиты SQL Server Migration Assistant (SSMA), поставляемой в составе пакета SQL Server 2005.

Утилита Migration Assistant

Изначальная версия утилиты SQL Server Migration Assistant (SSMA) включала поддержку миграции баз данных Oracle в SQL Server 2000. Последующие версии расширили список поддерживаемых баз данных и их преобразований.

Утилита SSMA сделала большой шаг вперед в определении сложности проектов баз данных в смысле затрат и времени, сопряженных с традиционным определением. Схема, данные, ограничения, миграция и проверка корректности могут контрлироваться с помощью новой среды.

Любая миграция проходит через следующие этапы: оценка, преобразование схемы, миграция данных, преобразование бизнес-логики, проверка корректности, интеграция и анализ производительности.

Оценка

Утилита SSMA оценивает работу, необходимую для выполнения миграции, после чего выводит отчет о том, что может быть сделано автоматически, а что требует ручной работы. Для характеристики базы данных используется около ста статистик, которые позволяют заглянуть глубоко в "сердце" базы данных. Утилита SSMA также выполняет оценку количества часов, необходимых для выполнения задач преобразования вручную.



В то время как утилита SSMA позволяет быстро оценить сложность базы данных, ей нужно некоторое время для оценки сложности клиентского программного обеспечения и приложений среднего уровня.

Преобразование схемы

После подключения к исходной базе данных Oracle и целевой базе данных SQL Server интерфейс отображает различные их атрибуты и объекты. Для сравнения могут отображаться программы на языках PL/SQL и T-SQL. Интерфейс поддерживает прямое редактирование отображаемого программного кода.

Функции системы Oracle, которые не имеют двойников в целевой базе данных SQL Server, будут поддерживаться с помощью определяемых пользователем функций и хранимых процедур. Ограничения, представления и индексы будут преобразованы в соответствующие сущности SQL Server, а типы данных будут отображены на новые так, как показано в табл. 4.9.

Таблица 4.9. Отображение типов данных PL/SQL на T-SQL

PL/SQL	T-SQL	PL/SQL	T-SQL	PL/SQL	T-SQL
Varchar2	Varchar	Date	Datetime	Boolean	Smallint
Char	Char	Long	Text		
Number	Numeric	Nvarchar2	Nvarchar		

Миграция данных

Схема Oracle может быть автоматически преобразована в схему SQL Server, при этом все данные будут перенесены в базу данных SQL Server. Во время процесса миграции данных администратор должен предотвращать выполнение всех ограничений, триггеров и прочих зависимостей, которые могут помешать вставке записей в таблицы.

Преобразование бизнес-логики

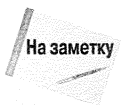
В табл. 4.10 приведены все преобразования, которые могут выполняться при миграции из PL/SQL в SQL Server.

Таблица 4.10. Преобразования PL/SQL в T-SQL

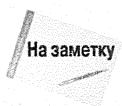
PL/SQL	T-SQL
Внешние объединения	Внешние объединения стандарта ANSI
Параметры	Поддерживаются такие параметры, как `First_Rows`, `Index`, `Append`, `Merge_Aj`, `Merge_Sj` и `Merge`. Неподдерживаемые параметры игнорируются
`Boolean`	`smallint`
Строковые параметры с не явно указанной длиной	`Varchar(8000)`
Числовые параметры с не явно указанной длиной и точностью	`Numeric(38,10)`
Функции	Пользовательские функции
Триггеры:	Триггеры:
`Before`	`Instead Of`
`After`	`After`
`Row-level`	эмуляция с помощью курсора
`Multiple`	комбинируются в один
Пакетные функции	Пользовательские функции, использующие соглашение *ИмяПакета_ИмяПроцедуры*.
Пакетные переменные	Эмулируются с помощью таблицы и поддерживающих функций
Системные функции	Системные или пользовательские функции
`If-Elsif...Elsif-Else-End`	Вложенные операторы `If`

Окончание табл. 4.10

PL/SQL	T-SQL
NULL	SYSDB.SYS.DB_NULL_STATEMENT
Case	Case
Goto	Goto
Loop с оператором Exit или Exit When	While (1=1) с оператором Break
While	While
For	While
Курсоры:	Курсоры:
с параметрами	множество курсоров
цикл FOR	курсор с локальными переменными
Close *имя_курсора*	Close *имя_курсора* и Deallocate *имя_курсора*
Return	Return
Comments	Comments
Переменные:	Переменные:
Static	разрешаются во время преобразования
с %Type	группа локальных переменных
с %RowType	группа локальных переменных
Вызовы процедур	Вызовы процедур
Вызовы функций	Вызовы функций
Begin Tran	Begin Tran
Commit	Commit
Rollback	Rollback
SavePoint	Save Transaction
Исключения	В T-SQL эмулируются



Транзакции в T-SQL могут задаваться неявно с помощью установки параметра SET IMPLICIT_TRANSACTIONS ON или явно с помощью инструкций BEGIN TRAN и COMMIT TRAN.



Если исключения на целевом сервере отключены, то их обработка выполняться не будет. Если обработка исключений включена, то они будут преобразованы с помощью операторов IF/GOTO и пользовательских функций.

Проверка корректности и интеграция

Интерфейс среды разработки (IDE) предлагает представление SQL, аналогичное сравнению программ новой и старой версий. Он поддерживает модификацию, принятие и отклонение предложенных изменений. Дополнительные параметры синхронизации позволяют переписывать объекты базы данных текущими объектами рабочего пространства, объекты рабочего пространства — объектами базы данных, а также слияние объектов.

Конфигурирование рабочего пространства SQL Server

Под рабочим пространством понимается степень уязвимости системы безопасности конкретного фрагмента программного обеспечения. Чем больше способов атак на программу возможно через службы, порты, интерфейс программирования, сценарии, сообщения и т.п., тем шире рабочее пространство и тем большему риску подвергается система безопасности. Сокращая рабочее пространство приложения, мы уменьшаем этот риск.

Сокращение рабочего пространства приложения включает в себя идентификацию и остановку или полное отключение неиспользуемых компонентов, функций и т.п. Для реализации этой задачи в SQL Server 2005 компания Microsoft создала специальную утилиту Surface Area Configuration Tool, способную помочь в более безопасной установке.

Утилита Surface Area Configuration

СУБД SQL Server 2005 можно назвать защищенной по умолчанию, так как большая часть служб в ней *изначально отключена*. Изначально конфигурируемое рабочее пространство сервера содержит ядро базы данных, а также службы анализа и отчетности. Ядро базы данных и служба анализа позволяют точнее конфигурировать службы, удаленные подключения и функции.

После завершения установки SQL Server 2005 запустите утилиту SQL Server Surface Area Configuration. Если выполнялось обновление версии SQL Server 2000, то все службы, запущенные в старой версии, будут запущены и в новой. Для доступа к инструментам выберите в меню пункт Microsoft SQL Server 2005⇨Configuration Tool⇨SQL Server Configuration Manager.

Изначальное окно ознакомит вас с некоторой информацией об этой утилите и предоставит средства для конфигурирования служб и подключений, равно как и функций заданного сервера, будь он локальным или удаленным. На рис. 4.4 показана изначальная форма. Вы мо-

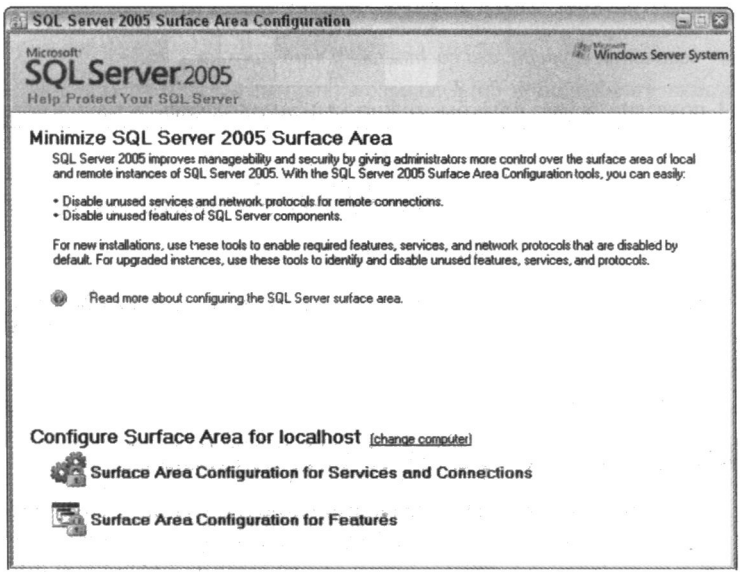


Рис. 4.4. Утилита Surface Area Configuration предоставляет механизм настройки служб, подключений и функций SQL Server 2005

жете сменить сервер, щелкнув на ссылке Change computer, расположенной сразу над параметрами конфигурации.

Бегло взглянув на эту утилиту, начинайте конфигурирование сервера, щелкнув на ссылке Services and Connections для данного сервера.

Конфигурирование рабочей области для служб и подключений

Ссылка Services and Connections открывает доступ к различным службам SQL Server. Как видно на рис. 4.5, можно запустить, приостановить и возобновить различные службы непосредственно из диалогового окна. К тому же в этом окне можно задать сетевые подключения и протоколы, определяя локальные либо удаленные подключения.

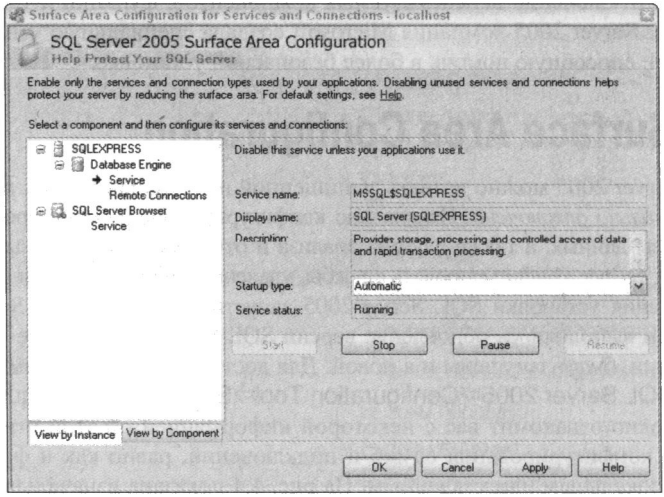


Рис. 4.5. Диалоговое окно Services and Connections утилиты конфигурирования. Здесь можно сконфигурировать подключения сервера, его состояние и тип запуска

В табл. 4.11 показано, какие службы можно конфигурировать, в зависимости от того, какая версия SQL Server установлена.

Таблица 4.11. Возможности конфигурирования сервера

Служба	Службы и подключения	Функции
Ядро базы данных	Да	Да
Служба анализа	Да	Да
Служба отчетности	Да	Да
Служба интеграции	Да	
Служба уведомлений	Да	
SQL Server Agent	Да	
Полнотекстовый поиск	Да	
SQL Server Browser	Да	

Конфигурирование функций

Диалоговое окно Surface Area Configuration for Features обеспечивает возможность управления дополнительными функциями ядра базы данных, а также служб анализа и отчетности.

Ядро базы данных

К конфигурируемым функциям ядра базы данных относятся следующие.

- Возможность запуска удаленных запросов с помощью инструкций `OPENROWSET` и `OPENDATASOURCE`.
- Интеграция CLR, позволяющая использовать управляемый код .NET для хранимых процедур, триггеров, пользовательских функций и типов данных. Эта функция по умолчанию отключена.
- Выделенное административное подключение (DAC) — специальное подключение с высоким приоритетом, с помощью которого администратор может проводить диагностику.
- Замена механизмом *Database mail* устаревшего механизма SQL Mail для отправки электронной почты.
- Встроенные Web-службы XML открывают доступ к ядру базы данных посредством протокола HTTP и сообщений SOAP. Эта функция по умолчанию отключена.
- Автоматизация OLE.
- Брокер служб реализует новый механизм работы с очередями в SQL Server для отправки сообщений другим приложениям и компонентам.
- Модуль Web Assistant состоит их хранимых процедур, которые генерируют код HTML. В версии SQL Server 2005 считается устаревшим.
- Системная хранимая процедура `xp_cmdshell` для запуска команд операционной системы. По умолчанию эта функция отключена.

Служба анализа

Конфигурируемые функции этой службы приведены ниже.

- Разовые запросы раскрытия данных, позволяющие использовать внешние источники данных.
- Анонимные подключения для неаутентифицированных пользователей.
- Связанные объекты, позволяющие связывать измерения и размерности разных экземпляров сервера.
- Возможность использования пользовательских функций. По умолчанию отключена.

Служба отчетности

Конфигурируемые функции этой службы приведены ниже.

- Генерируемые по расписанию события и доставка отчетов позволяют службе отправлять отчеты потребителям. Если функциональность подписки не нужна, эту функцию можно отключить, после чего отчеты будут доставляться по требованию пользователей.
- Web-службы и HTTP, используемые службой отчетности для доставки отчетов.



Конфигурирование этих функций также доступно с помощью хранимой процедуры `sp_configure`.

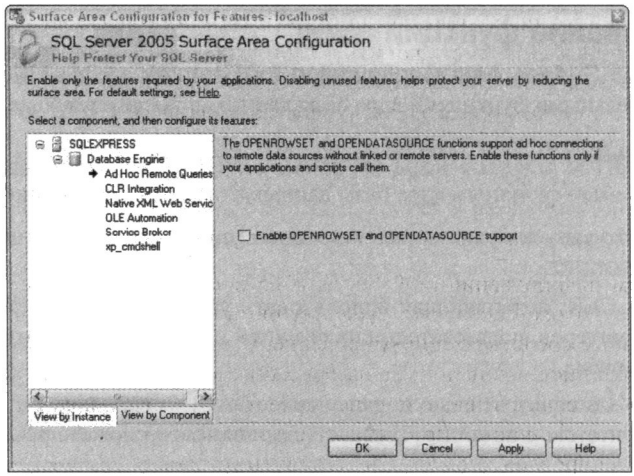


Рис. 4.6. Диалоговое окно Configuration for Features. *Из него вы можете конфигурировать различные функции служб*

На рис. 4.6 показан пример диалогового окна Configuration for Features.



В утилите SQL Server Surface Area Configuration служба интеграции и SQL Server Browser не представлены — они имеют собственные настройки.



Для улучшения защиты отключите все неиспользуемые функции, службы и подключения.

Утилиты командной строки

При наличии множества экземпляров SQL Server запуск утилиты Surface Area Configuration на каждом из них потребует много времени и усилий. К счастью, эта утилита сопровождается множеством утилит командной строки, упрощающих эти задачи. Утилита `sac` позволяет импортировать и экспортировать настройки конфигурации. После того как будет сконфигурирован один экземпляр, запустите эту утилиту для экспорта настроек рабочей области и применения их к другим экземплярам сервера. Утилиту `sac` можно найти в каталоге `<устройство>:\Program Files\Microsoft SQL Server\90\Shared`.

Удаление SQL Server

Для удаления SQL Server используйте консоль Установка и удаление программ панели управления операционной системы Windows. Каждый экземпляр SQL Server содержится в списке этой консоли отдельно и может быть удален без ущерба остальных экземпляров. Даже после удаления экземпляра, используемого по умолчанию, все оставшиеся именованные экземпляры будут функционировать.

При удалении сервера пользовательские базы данных не удаляются, и структура их каталогов остается нетронутой. Если отключить и скопировать базу данных на другой сервер перед удалением экземпляра SQL Server, можно не прерывать доступ к данным. Если такой ход событий невозможен, создайте резервную копию базы данных и восстановите ее на другом сервере или подключите безнадзорную базу данных к другому серверу.

Резюме

При правильном планировании SQL Server 2005 достаточно легко установить, а стремительное развитие технологий позволяет существенно повысить производительность, не вкладывая дополнительные средства в лицензирование. С выходом версии SQL Server 2005 компания Microsoft представила дополнительные механизмы миграции и конфигурирования, которые заменили собой существовавшие ранее средства, помогающие установить и обновить сервер. Установка по умолчанию продолжает оставаться предельно простой, а небольшое предварительное планирование поможет не оставить установку "сырой". Следуя философии "безопасности по умолчанию", в SQL Server 2005 изначально отключено большинство функций. Все функции, включенные до обновления версии SQL Server, остаются включенными и после. Клиент SQL Native Client (SNAC), который мы рассмотрим детально в главе 5, предлагает еще один вариант доступа к SQL Server 2005.

ГЛАВА

5

Подключение клиентского программного обеспечения

В этой главе...

Активизация подключений к серверу

Встроенные средства разработки клиентских приложений

Версия SQL Server 2005 внесла в этот продукт философию "безопасности по умолчанию" и существенно уменьшила рабочее пространство приложений. После изначальной установки сервера к нему разрешен только локальный доступ, а сетевые подключения запрещены. Таким образом, стандартные клиентские приложения по умолчанию к серверу подключиться не смогут.



В главе 4 мы обсуждали настройку рабочей области как части процесса установки сервера.

Утилита Server Configuration Manager, входящая в комплект поставки SQL Server 2005 и устанавливаемая вместе с сервером, практически всегда может подключиться к серверу, что позволяет сконфигурировать параметры подключения и открыть сервер для доступа по локальной сети.

Когда к серверу разрешен доступ по сети, SQL Server 2005 предоставляет пользователю новый механизм доступа с помощью функций SQL Server Native Client (SNAC). Но прежде чем воспользоваться этими функциями, к новому серверу следует открыть доступ по сети.

Разрешение подключений к серверу

После изначальной установки SQL Server активизирует протокол Shared Memory и отключает все остальные, что обеспечивает максимальную безопасность сервера по умолчанию, так как к нему могут подключиться только локальные приложения.

Для расширения области доступности сервера на нем следует активизировать дополнительные сетевые протоколы.

Дополнительная информация

В главе 34 мы более подробно обсудим конфигурирование SQL Server.

Утилита Server Configuration Manager

Сетевые протоколы определяют общий набор правил и форматов, которые используют компьютеры и приложения для взаимодействия друг с другом. Как уже говорилось ранее, протокол Shared Memory определяет, как приложения должны запускаться локально на сервере, чтобы взаимодействовать с ним.

Протокол именованных каналов Named Pipes является протоколом взаимодействия процессов, независимо от того, запущены ли они на одном компьютере или на разных. Он использует общее пространство памяти. Этот протокол хорошо зарекомендовал себя в малых и быстрых локальных сетях, поскольку он создает дополнительные сетевые потоки. В крупных и медленных локальных сетях лучше работает стек TCP/IP.

Семейство протоколов TCP/IP (*протокол управления передачей данных/протокол Интернета*) на сегодняшний день является одним из самых распространенных. Протокол TCP гарантирует доставку и упорядочение информации, пересылаемой между компьютерами, в то время как протокол IP определяет формат или структуру передаваемых данных. Стек TCP/IP также содержит расширенные функции обеспечения безопасности, что делает его особо привлекательным для организаций и пользователей, обладающих конфиденциальной информацией. Это семейство протоколов хорошо зарекомендовал себя в крупных и медленных локальных и глобальных сетях.

Утилита Server Configuration Manager позволяет включать и отключать все эти протоколы в соответствии с конкретной операционной средой и запускается из системного меню Пуск выбором команды Все программы⇨Microsoft SQL Server 2005⇨Configuration Tools⇨SQL Server Configuration Manager. В ее окне представлен список всех доступных протоколов и их параметров взаимодействия (рис. 5.1).

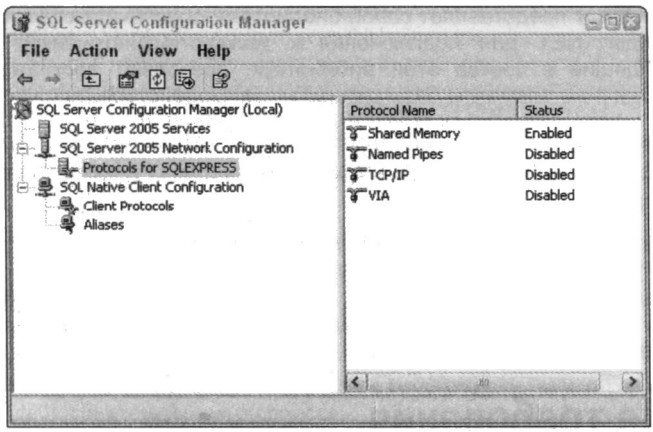


Рис. 5.1. Утилита SQL Server Configuration Manager позволяет настроить протоколы взаимодействия SQL Server с клиентами

Подключения SQL Native Client (SNAC)

Подключения SQL Native Client также управляются с помощью утилиты Server Configuration Manager. SNAC автоматически устанавливается для возможности активизации любого сетевого протокола, будь то Shared Memory, Named Pipes или TCP/IP (рис. 5.2).

Если вы не используете подключения SNAC, отключение соответствующих сетевых протоколов существенно сократит риски, связанные с безопасностью системы (т.е. рабочее пространство).

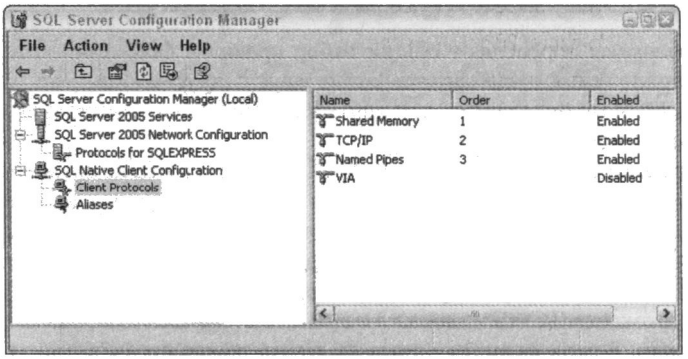


Рис. 5.2. Представление утилиты Server Configuration Manager для конфигурирования клиентских протоколов SNAC

Функции SQL Server Native Client

Сообщество разработчиков не приветствует доступ к новым функциям SQL Server 2005 посредством SQL Server Native Client. Если новые функции не нужны, а для доступа к данным необходим управляемый код, то будет достаточно использования библиотеки ADO.NET. Несмотря на то что детальное рассмотрение новых функций SQL Server 2005 выходит за рамки главы, приведем здесь их краткий обзор.



ADO.NET представляет собой библиотеку, обеспечивающую функциональность среды .NET для подключения к множеству источников данных. Классы, входящие в состав этой библиотеки, позволяют программистам создавать различные элементы базы данных и взаимодействовать с ними.

С точки зрения разработчиков, библиотека MDAC (Microsoft Data Access Components) совместима с SQL Server 2005, однако она не поддерживает новые расширенные возможности этого сервера.



Поскольку SQL Native Client является компонентом SQL Server 2005, его следует устанавливать на компьютере разработчика приложений и включать в программу установки приложения. На установочном компакт-диске SQL Server 2005 вы найдете файл `sqlncli.msi`, который позволяет установить SQL Native Client без установки СУБД.

Системные требования

Системные требования для установки SQL Server Native Client перечислены в табл. 5.1. Сама операционная система диктует требования к аппаратному обеспечению, в том числе к памяти, объему жестких дисков, производительности процессора и т.д.

Таблица 5.1. Системные требования для установки SNAC

Инсталлятор	Операционная система	Совместимая версия SQL Server
Windows Installer 3.0	Windows XP SP1+	SQL Server 7.0 или более поздние версии
	Windows 2000 Professional	
	Windows 2000 Server	
	Windows 2000 Advanced Server	
	Windows 2000 Datacenter	
	Windows 2003 Server	
	Windows 2003 Enterprise Server	
	Windows 2003 Datacenter Server	



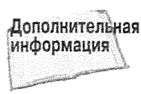
Дополнительную информацию о системных требованиях, необходимых для установки SQL Server 2005, см. в главе 4.

Зеркальное отображение баз данных

Когда установлено подключение к базе данных SQL Server 2005, в сценарии зеркального отображения автоматически определяется резервный сервер. Эта информация используется в SNAC для незаметного переключения на резервный сервер, если на основном произойдет сбой. Идентифицировать резервный сервер можно непосредственно в строке подключения к базе данных с помощью ключевого слова `Failover_Partner`.



Если подключение к основному серверу неожиданно обрывается, то все данные, с которыми велась работа в данный момент, будут утеряны, при этом, если выполнялась какая-либо транзакция, она будет аннулирована. Для соединения с резервным сервером подключение должно быть закрыто и снова открыто, только после этого работа с данными может быть возобновлена. Подключение к резервному серверу будет устанавливаться автоматически, если идентификация этого сервера будет частью объекта подключения, либо упоминалась в строке подключения, либо в предыдущем успешном подключении к основному серверу перед тем, как он вышел из строя.



В главе 39 описывается еще один инструмент, используемый для обеспечения доступности данных.

Асинхронные операции

Часто возникает необходимость получить немедленный отклик от отправленного вызова к базе данных. К счастью, теперь можно открывать и закрывать подключения к базе данных, не ожидая установки соответствующего свойства.

К тому же асинхронные вызовы позволяют получить результирующий набор данных. В этом случае результирующий набор данных будет существовать, но его заполнение данными может быть продолжено. Таким образом, всегда нужно проверять асинхронный статус результирующего набора данных и обрабатывать его только после завершения заполнения.



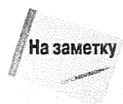
В выполнении асинхронных операций скрыто несколько подводных камней, в частности, при заполнении объектов подключения или при использовании механизмов курсоров. В этих случаях асинхронный статус невозможно узнать.

Множества активизированных результирующих наборов данных (MARS)

В SQL Server 2005 реализована поддержка множества активизированных инструкций SQL в одном и том же подключении, благодаря чему можно обращаться к нескольким наборам данных и выполнять дополнительные инструкции, не закрывая открытый набор данных.

Компания Microsoft дает ряд советов для создания приложений, использующих технологии MARS (Multiple Active Result Sets).

- Результирующий набор каждой инструкции должен иметь малую продолжительность жизни.
- Если результирующий набор данных имеет продолжительную жизнь или он очень большой, то лучше использовать серверный курсор.
- Всегда дочитывайте до конца результирующий набор данных и используйте вызовы API для изменения свойств подключения.



По умолчанию функция MARS отключена. Она включается в строке подключения с помощью значения `MarsConn` для поставщика данных OLE DB или значения `Mars_Connection` для поставщика ODBC.

Типы данных XML

Подобно существовавшему ранее для строк переменной длины типу данных `VarChar`, новый тип данных `XML` предназначен для документов XML и их фрагментов. Этот тип можно использовать для объявления переменных в хранимых процедурах, объявления параметров, а также для возвращаемых данных и их преобразований.



В главе 31 приводится дополнительная информация об использовании `XML`.

Типы, определяемые пользователем

Эти типы определяются с помощью кода .NET Common Language Runtime (CLR). При этом можно использовать популярные языки программирования C# и VB.NET. Сами данные представляются как поля и свойства, а их поведение описывается с помощью методов класса.

Особо крупные типы данных

Для обработки значений длиной $2^{31}-1$ байтов или символов в SQL Server 2005 введены три новых типа данных. Эти типы можно использовать в объявлениях переменных, что позволяет преодолеть существовавший ранее барьер в 8 Кбайт. Эти новые типы и соответствующие им старые приведены в табл. 5.2.

Таблица 5.2. Новые особо крупные типы данных в SQL Server 2005

Новый тип данных	Старый тип данных
`Varchar(max)`	`text`
`Nvarchar(max)`	`ntext`
`Varbinary(max)`	`Image`

Замена устаревших паролей

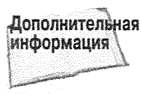
Эта новая функция SQL Server 2005 позволяет пользователям без вмешательства администратора изменять свои пароли, срок действия которых истек.

Пароль пользователя можно изменить одним из следующих способов:

- программным путем, например указывая в строке подключения как старый, так и новый пароль;
- с помощью интерфейса пользователя для изменения пароля до окончания срока его действия;
- с помощью интерфейса пользователя для изменения пароля уже после окончания его срока действия.



Если старый и новый пароли указываются явно в строке подключения, убедитесь, что старый пароль не остался в каком-либо внешнем файле. В то же время лучше встраивать новый пароль в динамические конструкции, чтобы избежать проблем взаимодействия с системой безопасности.



Система безопасности SQL Server 2005 подробно рассматривается в главе 40.

Уровень изоляции `SNAPSHOT`

Новый уровень изоляции `SNAPSHOT` повышает конкуренцию и увеличивает производительность за счет недопущения блокировки чтения-записи.

Этот уровень изоляции основан на поддержке версий записей таблиц. Транзакция начинается с инструкции `BeginTransaction`, но ей не присваивается последовательный номер, пока не будет выполнена первая инструкция T-SQL. Для поддержания версий строк таблицы используются их временные логические копии, которые хранятся в базе данных `tempdb`.



Если в базе данных `tempdb` недостаточно места для хранения версий, то любые функции и операции, такие как триггеры, MARS, индексация, выполнение клиентских инструкций T-SQL и поддержание версий строк, завершатся ошибкой. Таким образом, убедитесь, что база данных `tempdb` имеет размер, достаточный для своего беспроблемного использования.

Резюме

Утилита SQL Server Configuration Manager обеспечивает управление серверными протоколами, а также протоколом SQL Server Native Client.

SQL Server поддерживает новые функции, которые расширяют возможности приложений и клиентов. Доступ разработчиков к этим новым функциям реализуется с помощью клиента SQL Server Native Client (SNAC). Теперь клиент может сам динамически изменять свои пароли, расширять возможности блокировки и пользоваться возможностями асинхронных вызовов. В дополнение к этому за счет возможности использования зеркальных серверов и прочих полезных функций существенно повысилась устойчивость системы.

Использование утилиты Management Studio

ГЛАВА

6

В этой главе...

Введение в Management Studio

Навигация к объектам SQL Server

Организация проектов

Использование редактора запросов

Нововведения в пакете обновлений SP2

Основной интерфейс работы с SQL Server реализован с помощью утилиты Management Studio. Она вобрала в себя мощный набор инструментов, подобный Visual Studio, позволяющий разработчикам и администраторам баз данных создавать проекты баз данных и управлять СУБД SQL Server 2005 с помощью графического интерфейса пользователя либо с помощью инструкций T-SQL. Для задач бизнес-аналитики и обеспечения работы со службами анализа, отчетности и интеграции в пакете SQL Server содержится специализированная среда разработки — SQL Server Business Intelligence Development Studio.

Подобно множеству вещей в реальном мире, самые сильные черты утилиты Management Studio являются одновременно и самыми слабыми. Количество возможных задач, узлов деревьев и инструментов способно просто ошеломить нового пользователя. Все окна могут быть прикрепленными и плавающими, поэтому интерфейс выглядит перегруженным и беспорядочным, в чем вы могли убедиться на рис. 3.6 в главе 3.

Но как только назначение отдельных страниц становится понятным, а параметры интерфейса правильно настроены, Management Studio превращается в достаточно гибкий инструмент в руках администратора. Сам интерфейс может быть легко настроен на конкретные задачи управления базой данных. Его можно даже настроить так, чтобы он стал похожим на интерфейс Enterprise Manager и Query Analyser версии SQL Server 2000.

Последующие главы будут посвящены особенностям выполнения конкретных задач в Management Studio (MS) и Business Intelligence Development Studio (BIDS), так что все возможности в настоящей главе раскрыты не будут. Представьте себе эту главу как путеводитель по местности, в которой внимание обращается только на самые интересные объекты, встречающиеся по дороге.



Утилита Management Studio явилась результатом естественной эволюции утилит предыдущей версии SQL Server (Enterprise Manager и Query Analyser), которые она заменила. Инструменты Object Explorer и Query Editor унаследовали удобства своих предшественников, одновременно вобрав в себя гибкость Visual Studio. Утилита Management Studio имеет обратную совместимость с предыдущими версиями сервера (SQL Server 2000 и частью SQL Server 7), так что ее можно использовать в смешанной среде.

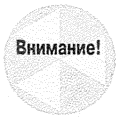
Наибольшим заблуждением в среде администраторов баз данных является вера в то, что Management Studio на самом деле и есть SQL Server. Однако это далеко не так. Management Studio является всего лишь инструментом в руках клиента, позволяющим управлять сервером и создавать базы данных. Подобно любому другому клиентскому приложению, Management Studio посылает команды T-SQL на сервер или использует специальные объекты управления сервером (SMO). Она также инспектирует сервер и отображает в понятном представлении его объекты и конфигурацию. Важным элементом организации работы в многосерверной среде является возможность подключения ко множеству экземпляров SQL Server, которая освобождает администратора баз данных от бесконечных переходов от компьютера к компьютеру.



Большая часть работы с Management Studio покажется профессионалам очевидной, так что в этой главе мы остановимся лишь на ключевых (в моем понимании) моментах и не станем рассматривать все пункты меню подряд. Если вас заинтересует обучающий курс по Management Studio, то вы найдете его в книге Элизабет Кори *An Introduction to SQL Server 2005 Management Studio* на сайте `www.SQLServerCentral.com`.



Довольно интересно посмотреть на команды, которые утилита Management Studio отсылает на сервер. Эта утилита генерирует сценарий практически для каждого действия, а реальные пакеты можно увидеть с помощью утилиты SQL Profiler, которую мы рассмотрим в главе 49.



С выходом пакета обновлений SP2 в режиме совместимости с SQL Server 2000 теперь можно установить поддержку диаграмм объектов базы данных. Также в утилиту был добавлен новый монитор зеркального отображения баз данных.

Организация интерфейса

Утилита Management Studio содержит богатый набор функций, организованных в десяток основных панелей, которые можно открыть с помощью меню View, стандартной панели инструментов или ассоциированных с командами "горячих" клавиш.

- **Object Explorer (<F8>)**. Эта панель используется для создания и администрирования объектов баз данных SQL Server 2005. Она вобрала в себя все лучшее от утилиты Enterprise Manager и панели Object Browser утилиты Query Analyzer предыдущей версии сервера.
- **Summary (<F7>)**. Эта панель содержит основную информацию о выделенном объекте; на ней также можно увидеть некоторые отчеты.
- **Registered Servers (<Ctrl+Alt+G>)**. Эта панель используется для управления подключениями к нескольким ядрам SQL Server 2005. Здесь можно зарегистрировать ядра серверов баз данных, служб анализа, отчетности и интеграции, а также SQL Server Mobile.
- **Template Explorer (<Ctrl+Alt+T>)**. Эта панель используется для управления шаблонами программ T-SQL.

- **Solution Explorer** (**<Ctrl+Alt+L>**). Эта панель используется для организации проектов и управления текстами программ.
- **Properties** (**<F4>**). На этой панели отображаются свойства выделенного объекта.
- **Bookmarks** (**<Ctrl+K>, <Ctrl+W>**). На этой панели перечислены закладки, установленные в окне Query Editor.
- **Web Browser** (**<Ctrl+Alt+R>**). Используется в редакторе запросов для отображения текстов XML или HTML.
- **Output Window** (**<Ctrl+Alt+O>**). На этой панели отображаются сообщения от интегрированных в Management Studio средств разработки.
- **Query Editor**. Наследник анализатора запросов версии SQL Server 2000, этот редактор запросов позволяет создавать, редактировать и выполнять пакеты инструкций T-SQL. Этот редактор открывается с помощью пункта меню File⇨New, выбора существующего файла запросов (разумеется, если расширение `.sql` ассоциировано в операционной системе с Management Studio), щелчка на кнопке New Query панели инструментов, а также запуска сценария запроса из объекта в окне Object Explorer.



С выходом пакета обновлений SP2 в SQL Server Management Studio был добавлен монитор зеркального отображения баз данных.

Чаще всего вам придется работать со следующими панелями: Registered Servers, Summary, Object Explorer, Template Explorer и Properties. Все они доступны на стандартной панели инструментов.

Размещение окон

Как и в Visual Studio, любое окно можно сделать прикрепленным или плавающим, включить как закладку в другое окно, а также скрывать за одной из границ главного окна программы. Текущий режим можно выбрать с помощью щелчка правой кнопки на заголовке окна, выбора значка со стрелкой вниз на крайней правой границе прикрепленного окна или с помощью меню Window. Также можно автоматически изменить режим отображения окна, перетаскивая его за заголовок в соответствующее место экрана. Вот какие режимы вам доступны.

- Установка режима окна floating (плавающее) открепляет его от границы главного окна Management Studio. Далее оно ведет себя как обычное не модальное диалоговое окно.
- Установка режима tabbed (режим вкладки) немедленно перемещает его в разделенный на вкладки документ в центре окна Management Studio в качестве одной из вкладок. Порядок вкладок можно изменить с помощью перетаскивания их указателем мыши. При перетаскивании вкладки в положение отдельно от документа автоматически создается новый документ. В любом положении (в центре, справа, слева, вверху и внизу) может находиться несколько разделенных на вкладки инструментов и документов.
- Рабочая область документа может хранить больше вкладок, чем может на ней физически поместиться. Существует два способа просмотреть скрытые вкладки. Наиболее очевидным из них является использование полосы горизонтальной прокрутки. В то же время эффективнее всего развернуть список документов с помощью стрелки Active File в верхнем правом углу рабочей области документа.
- При перемещении окна в режиме dockable (прикрепленное) Management Studio отображает несколько синих индикаторов прикрепления, как показано на рис. 6.1. Если

оставить окно около одного из этих индикаторов, оно станет прикрепленным к соответствующей границе. Если оставить его около индикатора в центре окна, то окно превратится в документ, разделенный на вкладки.

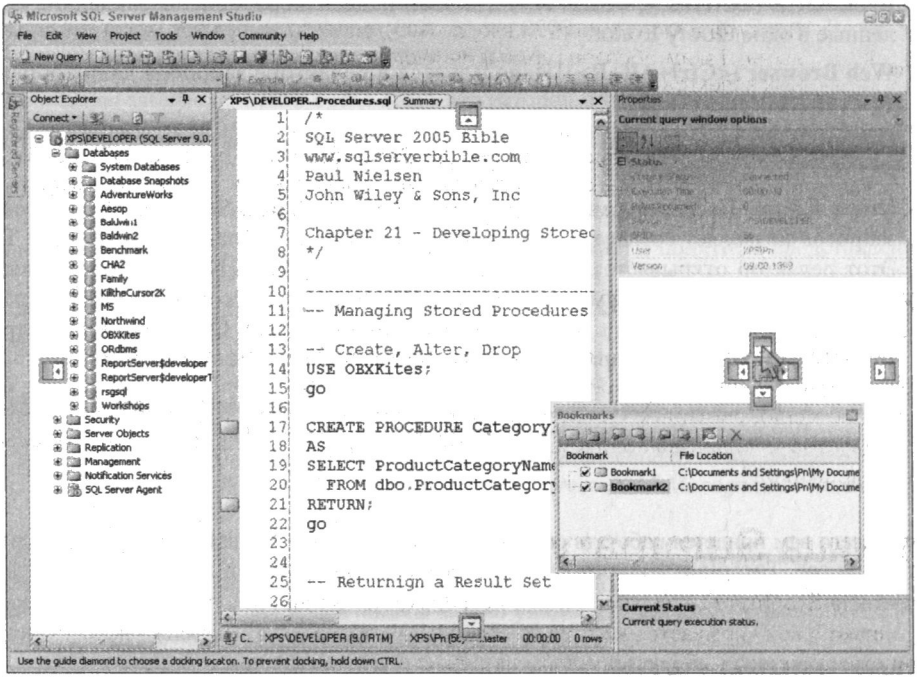


Рис. 6.1. Если оставить окно около одного из синих маркеров со стрелочкой, то оно прикрепится к соответствующей границе главного окна Management Studio. На представленном рисунке мы собираемся прикрепить окно Bookmarks *непосредственно над окном* Properties

- Одновременное открытие нескольких окон позволяет держать все необходимые инструменты под рукой. Однако если у вас нет широкоформатного монитора, вы не сможете эффективно воспользоваться ни одним из окон. Я, например, предпочитаю использовать функцию автоматического сокрытия окон. Оно становится невидимым до тех пор, пока вы не щелкнете на соответствующей вкладке. Для настройки этого режима выберите пункт меню View⇨Auto-Hide или щелкните на значке булавки в заголовке окна. Когда булавка находится в вертикальном положении, окно остается открытым, в противном случае оно автоматически скрывается. Автоматически скрываемое окно нужно снова открыть, прежде чем можно будет изменить его режим на плавающий или режим вкладки (см. рис. 6.1).



Чтобы изменить режим отображения окон Management Studio на принятый по умолчанию (т.е. будут открыты только окна Object Explorer, Tabbed Documents и Property), выберите пункт меню Window⇨Reset Window Layout. Эта команда не способна восстановить какой-либо измененный пользователем режим.

Для переключения между открытыми окнами используют комбинацию клавиш <Ctrl+Tab>.

Для сокрытия всех прикрепленных окон и сохранения на экране только открытых в центре документов с вкладками используют команду Window-Auto Hide All.

Такой гибкий механизм позиционирования окон позволяет сконфигурировать интерфейс так, чтобы открыть наиболее эффективный доступ к используемым инструментам. Мне нравится автоматически скрывать окно Object Explorer и размещать на двух вертикальных панелях несколько окон редакторов запросов.



Все окна в Management Studio, в том числе и диалоговые, не являются модальными. Таким образом, компания Microsoft преодолела один из самых неприятных аспектов интерфейса Enterprise Manager версии SQL Server 2000.



Бесплатная демонстрация *ScreenCast*, которую можно найти на сайте `www.SQLServerBible.com`, иллюстрирует различные способы перемещения, сокрытия и прикрепления окон, доступных в Management Studio.

Контекстное меню

В соответствии со стандартом интерфейса Microsoft Windows контекстное меню является основным механизмом выбора действий и просмотра свойств в Management Studio. Контекстное меню, связанное с большинством типов объектов, содержит пункты создания новых объектов и задач, выполняемых с ними. В Management Studio эти меню являются "рабочей лошадкой".

Страница Summary

Страница Summary может показаться несколько не связанной с общим интерфейсом утилиты, однако в ней скрывается несколько полезных отчетов. Эта страница также является местом множественного выбора подузлов, связанных с текущим узлом.



Страница Summary в утилите Enterprise Manager выглядела пустой, однако с тех пор в нее был внесен ряд улучшений, и теперь в сокрытых в ней отчетах можно отыскать массу полезного.

Окно Registered Servers

Панель со списком зарегистрированных серверов является дополнительным элементом интерфейса. Если вы обслуживаете один или два сервера, то она вам может понадобиться достаточно редко. Если же вы отвечаете за множество серверов, то именно на этой панели вы сможете подключаться к любому из них.

На панели Registered Servers может вестись работа с подключениями к ядру базы данных, серверам анализа, отчетности и интеграции, а также к экземплярам SQL Server Mobile Edition. Панель инструментов в верхней части панели Registered Servers позволяет выбрать множество типов служб.

Контекстное меню, показанное на рис. 6.2, можно использовать для запуска и останова служб, а также для открытия страницы свойств регистрации. Это меню также позволяет импортировать и экспортировать информацию о подключении для переноса регистраций между разными инсталляциями утилиты Management Studio.

В дереве Registered Servers отдельные серверы можно организовать в группы. В организации таких групп не скрыты какие-либо механизмы наследования или что-то другое — они являются всего лишь средством улучшения визуализации связанных серверов в дереве.

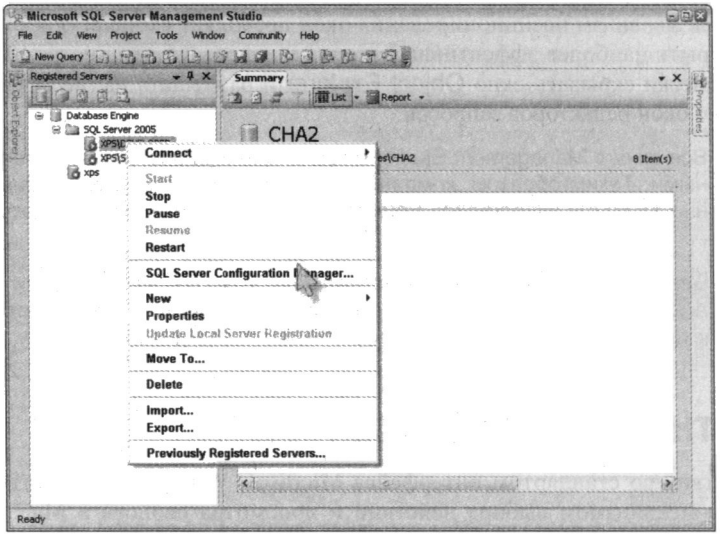


Рис. 6.2. Регистрация экземпляра SQL Server в Management Studio позволяет этой утилите подключиться к серверу

Окно Object Explorer

Являющееся результатом слияния окна Object Browser утилиты Query Analyzer и утилиты Enterprise Manager, это окно предлагает отлично организованное представление всех объектов сервера. На верхнем уровне дерева перечисляются все подключенные серверы. В окне Object Explorer можно подключиться к любому серверу, независимо от того, содержится ли он в списке зарегистрированных. Основным преимуществом операции регистрации сервера является возможность его запуска и остановки в окне Registered Servers. Цвет значка сервера отражает его текущее состояние — запущен он или остановлен.

Навигация по дереву

Подобно Проводнику Windows, окно Object Explorer (рис. 6.3) является иерархическим, развертываемым представлением объектов, доступных в подключенных серверах. Все дерево состоит из корней и узлов. Например, в дереве Мои документы Проводника Windows узел Рабочий стол является корневым, а все остальные папки и устройства раскрываются как его отдельные подузлы.

Среди узлов сервера можно встретить базы данных, средства защиты, серверные объекты, репликация, управление, службы уведомлений и агент SQL Server Agent. Большая часть структуры этого дерева фиксирована, но в процессе работы в нее автоматически добавляются объекты, создаваемые на сервере.

Узел `Databases` содержит все базы данных, находящиеся на сервере. Если щелкнуть правой кнопкой мыши на любой из этих баз данных, откроется контекстное меню, содержащее основные операции, выполняемые над базой данных. Узел каждой базы данных содержит связанные с ней подузлы (рис. 6.4), позволяющие управлять следующими объектами базы данных.

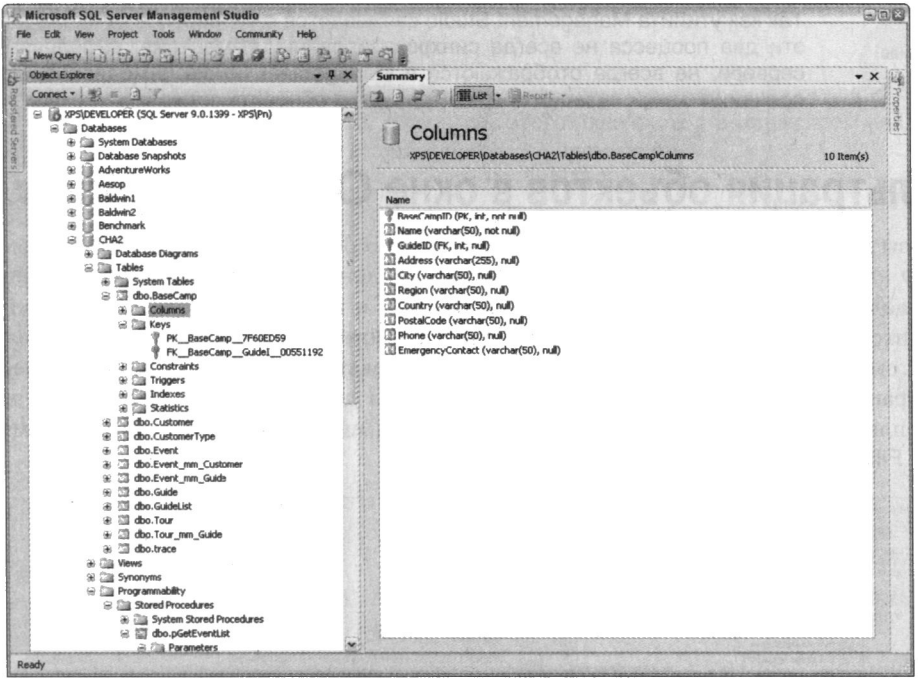


Рис. 6.3. Структура дерева в окне Object Explorer *предлагает вам исследовать различные компоненты сервера, в том числе средства управления и разработки*

- **Диаграммы базы данных** (`Database Diagrams`). Диаграммы отображают в графическом представлении отдельные таблицы базы и отношения между ними. Одна база данных может иметь несколько диаграмм, при этом в каждой из диаграмм не обязательно отображаются все таблицы. Такой подход облегчает организацию крупных баз данных в модульные диаграммы.
- **Таблицы** (`Tables`). Этот узел используется для создания и модификации структур таблиц, а также для работы с их индексами, разрешениями и публикациями. Здесь могут создаваться и редактироваться триггеры и хранимые процедуры, отвечающие за операции модификации данных (вставку, обновление и удаление). Только отсюда можно запустить конструктор запросов (Query Designer).
- **Представления** (`Views`). В этом узле можно создавать и редактировать инструкции создания представлений, а также просматривать результаты их работы.
- **Синонимы** (`Synonims`). Здесь содержатся альтернативные имена объектов базы данных.
- **Программирование** (`Programmability`). В этом большом разделе содержится большая часть объектов разработки, хранимых процедур, функций, триггеров базы данных, сборок, типов данных, правил и установок по умолчанию.
- **Брокер служб** (`Service Broker`). Этот узел используется для просмотра содержимого асинхронных очередей брокера служб.
- **Хранение** (`Storage`). Этот узел используется для управления нестандартными хранилищами и содержит операции, такие как полнотекстовый поиск и разбиение таблиц на разделы.



Так как утилита Management Studio связывается с сервером в качестве клиента, эти два процесса не всегда синхронизированы. Изменения, выполняемые на сервере, не всегда отображаются в Management Studio немедленно. Для выполнения синхронизации обычно требуется обновить содержимое окна.

Фильтрация объектов в окне Object Explorer

Базы данных могут содержать великое множество объектов. Для облегчения навигации между ними компания Microsoft предусмотрела фильтрацию фрагмента дерева, содержащего пользовательские объекты, такие как таблицы и представления. Кнопка фильтрации находится на панели инструментов окна Object Explorer. Эта кнопка активизирована только в том случае, когда выделен узел типа пользовательского объекта. Например, для фильтрации таблиц выделите соответствующий узел дерева, а затем щелкните на кнопке Filter или щелкните на этом узле правой кнопкой мыши и выберите в контекстном меню пункт Filter⇨Filter Settings (рис. 6.4).

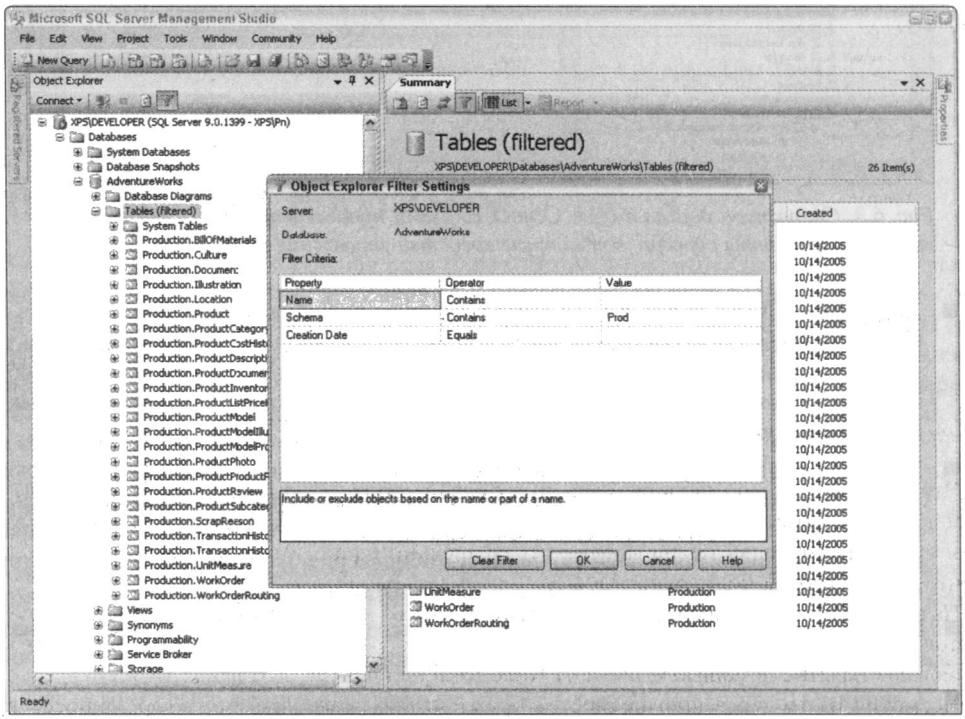


Рис. 6.4. Диалоговое окно Filter Settings используется для ограничения числа объектов, отображаемых в окне Object Explorer

Диалоговое окно Filter Settings позволяет фильтровать объекты по имени, схеме и дате создания. Для того чтобы снять фильтр, можно воспользоваться все тем же контекстным меню или открыть диалоговое окно Filter Settings и щелкнуть на кнопке Clear Filter. К сожалению, фильтр для каждого из параметров воспринимает всего одно значение. Булевы операции здесь недопустимы.

Конструктор таблиц

Создавать новые таблицы, а также модифицировать структуру уже существующих проще всего в окне Table Designer (рис. 6.5). Интерфейс этого окна схож с интерфейсом аналогичного окна Microsoft Access и конструкторами прочих приложений.

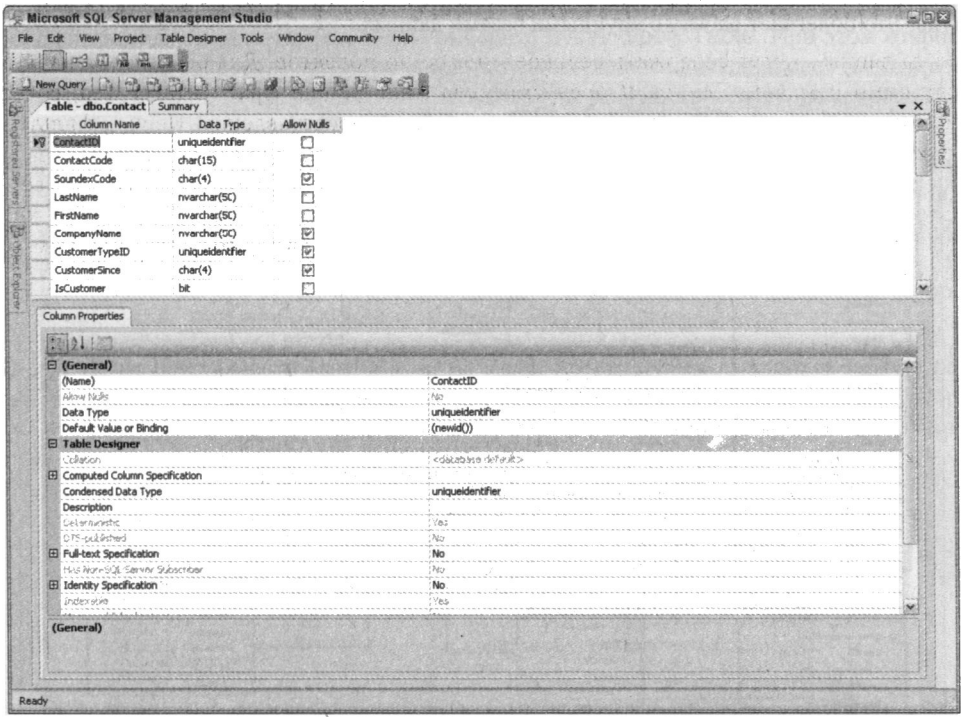
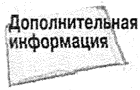


Рис. 6.5. В окне Table Designer можно создавать и изменять схемы любых таблиц

Чтобы создать новую таблицу, щелкните правой кнопкой мыши на узле `Tables` в дереве базы данных, а затем выберите в контекстном меню пункт New Table. Для изменения структуры уже существующей таблицы выделите ее, щелкните на ней правой кнопкой мыши и выберите в контекстном меню пункт Modify.

На верхней панели можно выделять и редактировать отдельные столбцы. Свойства выделенного столбца отображаются на нижней панели. Диалоговое окно для редактирования внешних ключей можно открыть с помощью меню окна Table Designer или его панели инструментов.

Несмотря на то что я программист и предпочитаю текстовый редактор запросов всем графическим инструментам, должен признать, что страница конструктора таблиц имеет прозрачный и простой интерфейс и способна генерировать правильные сценарии любых изменений структуры таблиц.



О логических схемах таблиц и их столбцов см. в главе 2. Особенности практической реализации логических схем мы рассмотрим в главе 17.

Создание диаграмм баз данных

Инструмент Database Diagram берет за основу структуры таблиц, созданные в узле Table Designer, и создает графическое представление таблиц и отношений между ними (рис. 6.6). Этот инструмент имеет собственный узел в каждой из баз данных. Каждая база данных может иметь множество диаграмм, так как в каждой из них может быть представлен ее конкретный модуль. Это облегчает работу с крупными базами данных.

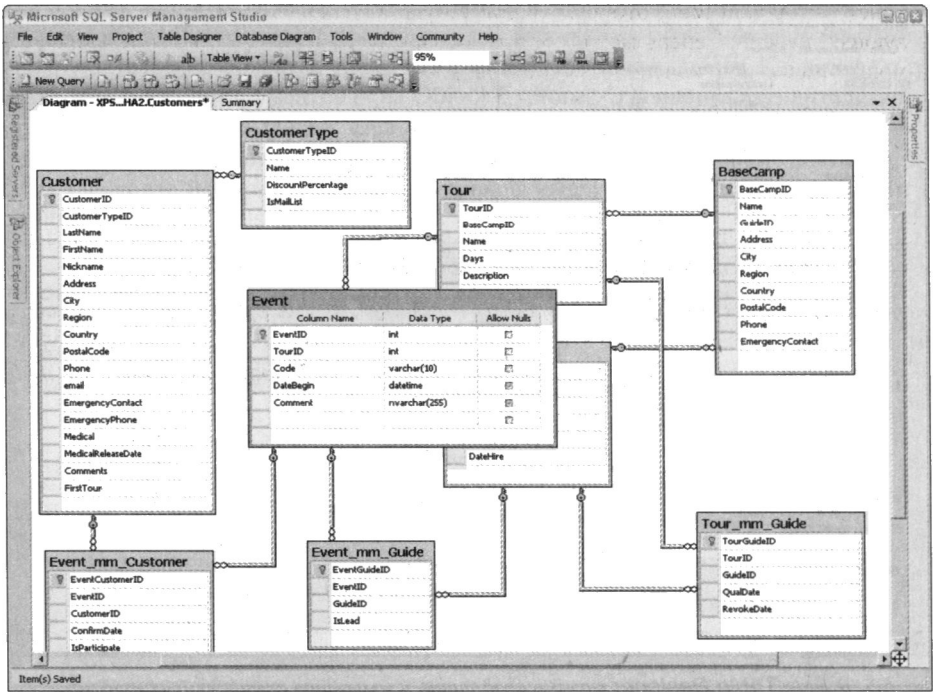


Рис. 6.6. Связи базы данных `Cape Hatteras Adventures`*, представленные с помощью инструмента* Database Diagram*. Представление таблицы* `Events` *было изменено на стандартное*



Пример процесса создания физической схемы с помощью инструмента Database Diagram можно просмотреть и загрузить с Web-сайта книги по адресу `www.ServerBible.com`.

Мне нравится инструмент Database Diagram, однако некоторые разработчики заявляют о его несовершенстве. Самым главным недостатком является то, что линии отношений соединяют пары таблиц, явно не указывая на конкретные первичный и внешний ключи. Эта проблема усугубляется другой — при перемещении таблиц линии скручиваются. Тем не менее диаграмма базы данных может оказаться особенно полезной при просмотре схем очень больших баз данных. В этой ситуации главные таблицы имеют десятки связанных с ними линий. Если позволить линиям автоматически связываться с первичными ключами, то результат будет представлять собой нечитаемое месиво.

Конструктор запросов

Конструктор запросов (Query Designer) — прекрасное средство извлечения и модификации данных, хотя его нельзя назвать простейшим инструментом утилиты Management Studio. Открыть его можно несколькими способами.

- В окне Object Explorer выделите таблицу и щелкните на ней правой кнопкой мыши. В открывшемся контекстном меню выберите пункт Open Table. При этом откроется конструктор запросов, отображающий результат запроса "извлечь все" на соответствующей панели. Теперь можно открыть и другие панели конструктора запросов с помощью меню или панели инструментов.
- Из редактора запросов (Query Editor) конструктор можно открыть с помощью команды меню Query⇨Design Query или кнопки Query Designer панели инструментов.
- Конструктор запросов интегрирован в редактор запросов. В редакторе запросов выделите любой запрос и щелкните на кнопке Query Designer панели инструментов. Следует отметить, что когда конструктор запросов открывается из редактора запросов, окно первого является модальным, а панель результатов отключена.

В отличие от других инструментов работы с запросами, переключающихся между графическим, текстовым представлением инструкции SQL и результатами запроса, конструктор запросов утилиты Management Studio способен одновременно отображать множество панелей, выбранных на панели инструментов (рис. 6.7).

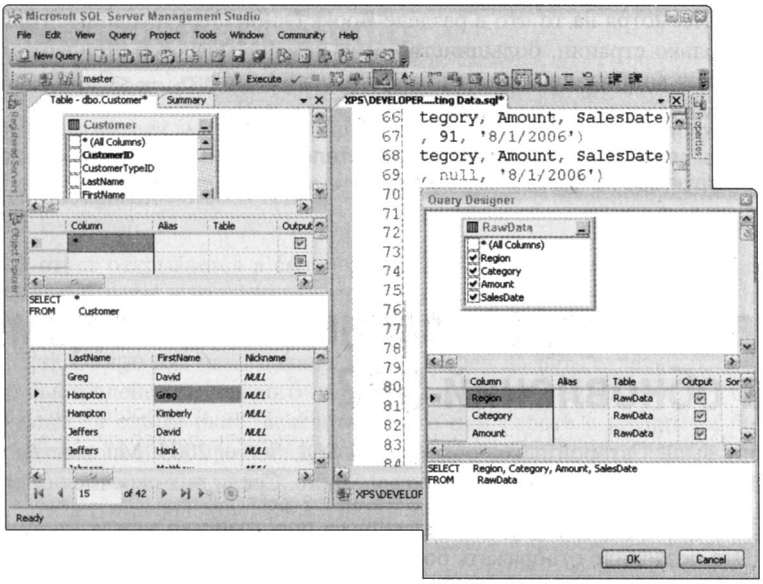


Рис. 6.7. Конструктор запросов окна Object Explorer

- **Панель диаграммы**. В запросе может участвовать множество таблиц и представлений. Для связывания их отношениями с целью формирования предложения FROM инструкции SELECT можно использовать это графическое представление.
- **Панель сетки**. На этой панели перечисляются отображаемые, фильтруемые и сортируемые столбцы.

- **Панель SQL**. На этой панели можно в текстовом виде ввести и отредактировать инструкцию SELECT.
- **Панель результатов**. Когда запрос выполняется с помощью кнопки Run (и только!), на этой панели отображаются его результаты. Если результаты запроса остаются нетронутыми долгое время, Management Studio запрашивает у пользователя разрешение закрыть подключение к серверу.



Одним из моих самых любимых нововведений в SQL Server 2005 является возможность создавать и графически связывать таблицы на панели диаграмм конструктора запросов. Наслаждайтесь и вы!

Кроме инструкции SELECT в конструкторе запросов можно выполнять и другие инструкции языка манипулирования данными (DML): INSERT, UPDATE и DELETE. Однако в отличие от редактора запросов здесь нельзя выполнять пакеты, а также инструкции, не входящие в состав DML.

Конструктор запросов можно использовать для редактирования данных непосредственно на панели результатов — это самый быстрый и "грязный" способ корректировки данных. Иногда вы можете столкнуться с ошибкой "Cannot edit data in Firehose mode". Это значит, что конструктор запросов еще не завершил извлечение данных. Подождите пару минут и дайте возможность утилите Management Studio завершить диалог с SQL Server, после чего эту ошибку вы уже не встретите.

Навигация в конструкторе запросов должна быть интуитивно понятна опытным пользователям Windows. Несмотря на то что в разделе Books Online перечислению "горячих" клавиш посвящено несколько страниц, большинство этих команд являются стандартными для Windows. Единственная комбинация, которую стоит здесь упомянуть, — <Ctrl+0>. Она позволяет вставить в ячейку пустое значение NULL.



Вы можете ознакомиться с показательным примером создания запроса в конструкторе на сайте книги по адресу www.SQLServerBible.com.

Нововведения, связанные с выходом пакета обновлений SP2

В 2007 году вышел второй пакет обновлений SQL Server 2005. Множество улучшений и нововведений коснулись и утилиты Management Studio. Среди них перечислим следующие.

- В графическом представлении плана запроса пространство между узлами было сокращено. Это позволило отображать больший объем информации плана.
- Диалоговое окно создания связанного сервера New Linked Server теперь содержит кнопку Test Connection. Узел Linked Servers в Object Explorer теперь отображает системные каталоги и объекты пользовательских баз данных. После создания связанного сервера можно раскрыть его узел и увидеть таблицы и представления удаленного сервера.
- Пользователи теперь могут автоматически видеть основной список объектов, доступ к которым им разрешен, отозван или запрещен.

- Диалоговое окно Restore Database позволяет восстановить одну базу данных, даже если в одном файле `.bak` их зарезервировано несколько.
- Диалоговое окно Restore Database позволяет редактировать размещение резервной копии, чтобы восстанавливать базы данных с неотображенных сетевых устройств.
- Диалоговое окно Restore Database распознает европейскую дату и время при восстановлении базы данных к заданному состоянию.
- Диалоговое окно Attach Database позволяет изменять владельца базы данных пользователям, имеющим учетную запись домена Windows, такую как `BULTIN\Administrators`.
- Диалоговое окно Backup Database позволяет использовать NULL-устройство в качестве места размещения резервной копии.
- Диалоговое окно Backup Database в Management Studio Express позволяет создавать резервные копии баз в экземплярах SQL Server Desktop Engine (также известного как MSDE 2000).
- Производительность диалогового окна Database Property была существенно повышена для лучшей поддержки баз данных со множеством файлов и файловых групп.
- Диалоговое окно Full-Text Catalog Properties позволяет создавать индексы, используя расписания именованных экземпляров SQL Server.
- Страница Server Properties позволяет включить параметр Common Criteria для SQL Server 2005 Enterprise Edition.

Отчеты Management Studio

Выход пакета обновления SP2 затронул и отчеты Management Studio.

- Теперь пользователи могут интегрировать собственные отчеты управления в Management Studio.
- Отчеты Management Studio теперь можно открыть с помощью контекстного меню в Object Explorer.
- Каждый отчет открывается в своем окне. Это позволяет пользователю легко сравнивать разные отчеты и отслеживать недавно использовавшиеся отчеты, что облегчает доступ к наиболее часто выполняемым отчетам.
- Отчет Disk Usage для базы данных был разделен на пять отдельных отчетов, что повысило быстродействие каждого из них. Существовавший ранее отчет отображал всю информацию в одном окне, вследствие чего при работе с крупными базами данных вывод данных занимал продолжительное время. Новый отчет Disk Usage отображает только данные верхнего уровня и файлы протоколов, используемые в базе данных. Остальные отчеты следующие: Disk Usage per Top Tables для наибольших 1000 таблиц, Disk Usage Per Table (использование диска в разрезе таблиц), Disk Usage per Partition (использование диска в разрезе разделов) и Index Physical Statistics (статистика физических индексов).
- Отчет Disk Usage для базы данных больше не рекомендует реорганизацию всех индексов и основывается на рекомендациях для `sys.dm_index_usage_stats`.

Использование редактора запросов

Редактор запросов (Query Editor) пришел на смену анализатору запросов (Query Analyzer), улучшив функциональность последнего.

Подключение к серверу

Редактор запросов может поддерживать множество открытых окон и подключений в области документов с вкладками. На самом деле разные окна могут быть подключены к серверу как разные пользователи, что особенно полезно при тестировании системы безопасности.

После первого запуска редактора запросов он запрашивает изначальное регистрационное имя. Затем с помощью команды меню File-New Connection вы можете создать дополнительные подключения. В заголовке окна отображается имя сервера и имя зарегистрировавшегося пользователя.



Редактор запросов утилиты Management Studio может работать с файлом `.sql` и не будучи подключенным к серверу. Инструмент Query Analyzer в предыдущих версиях сервера этого делать не позволял и требовал наличия подключения к серверу даже при редактировании файла запроса.

Открытие файла `.sql`

Существует множество способов открытия ранее сохраненного пакетного файла пакетного запроса и одна ловушка, которой следует избегать.

- Если утилита Management Studio не открыта, дважды щелкните на файле `.sql` в Проводнике Windows. При этом будет запущена утилита Management Studio, запрошено ваше разрешение на подключение к серверу и открыт файл. Если в Проводнике выбрать одновременно несколько файлов `.sql` и попытаться открыть их как группу, то для каждого их файлов будет запущен отдельный экземпляр Management Studio, что я не назвал бы хорошей идеей.
- Если утилита Management Studio уже открыта, каждый двойной щелчок позволит вам открыть один файл или их группу в окне редактора запросов. Следует отметить, что каждый файл запросит у вас подключение к серверу.
- Множество файлов `.sql` можно перетащить из окна Проводника Windows в окно Management Studio. После запроса подключения к серверу каждый файл откроется в отдельном окне редактора запросов.
- Если вы недавно работали с файлами `.sql`, то они могут быть перечислены в группе последних открывавшихся файлов системного меню Пуск. Щелчок на таком файле в меню приведет к его открытию в редакторе запросов.
- Пункт меню File⇨Open или соответствующая кнопка панели инструментов позволяет открыть стандартное окно открытия файла, в котором можно выбрать один или несколько файлов.

Выполнение пакетов SQL

Будучи средством разработки, редактор запросов позволяет выполнять пакеты T-SQL, состоящие из множества инструкций этого языка. Чтобы передать пакет серверу для обработки,

выберите команду меню Query⇨Execute Query, или щелкните на кнопке Run Query панели инструментов, или нажмите клавишу <F5>, или комбинацию клавиш <Ctrl+E>.

Так как пакеты инструкций, как правило, довольно длинные, часто предпочтительнее для тестирования выполнять отдельные инструкции пакета или его фрагменты. Также можно выполнять пакет пошагово — команда разработчиков SQL Server теперь предоставила вам эту удобную функцию. Если не выделен какой-либо фрагмент текста, то будет выполнен весь пакет, если выделен, то будет выполнен только он.

Следует отметить, что команда меню Parse Query и соответствующая кнопка панели инструментов позволяют проверить только правильность кода SQL. Эта функция не проверяет имена объектов (таблиц, столбцов, хранимых процедур и т.п.). На самом деле это не ошибка программы, а свойство функции. Данная функция всего лишь проверяет синтаксис инструкций SQL, но в то же время SQL Server позволяет создавать в пакетах объекты и затем ссылаться на них.

Пакет T-SQL будет выполняться в контексте текущей базы данных. Текущая база данных отображается в комбинированном списке панели инструментов и при желании может быть изменена.

Результаты выполнения запроса отображаются на нижней панели, при этом формат может быть как табличным, так и текстовым — между ними можно переключаться с помощью комбинаций клавиш <Ctrl+D> и <Ctrl+T> соответственно. Новый формат будет применен при выполнении следующего пакета.

При работе с кодом T-SQL в редакторе запросов вы можете получить справку по выделенному ключевому слову, нажав комбинацию клавиш <Shift+F1>. В качестве альтернативы в Management Studio можно открыть окно динамической справки, при этом ваша работа будет отслеживаться, а в этом окне отображаться справка по соответствующей теме.

"Горячие" клавиши и закладки

Закладки представляют собой отличный способ навигации по большим сценариям. Их можно устанавливать вручную, а также автоматически с помощью команды Find. Закладки работают с двойными комбинациями "горячих" клавиш. Например, удерживая нажатой клавишу <Ctrl> и последовательно нажимая клавиши <K> и <N>, можно перейти к следующей закладке. Комбинация клавиш <Ctrl+K> управляет некоторыми другими функциями редактирования, в частности комментирования кода. Закладками можно управлять также с помощью команды меню Edit⇨Bookmarks и кнопок Next и Previous панели инструментов. В табл. 6.1 перечислены "горячие" клавиши, которые я считаю особенно полезными.

Таблица 6.1. Полезные "горячие" клавиши редактора запросов

"Горячие" клавиши	Описание
<Ctrl+K+K>	Добавить или удалить закладку
<Ctrl+K+A>	Активизировать все закладки
<Ctrl+K+N>	Перейти к следующей закладке
<Ctrl+K+P>	Перейти к предыдущей закладке
<Ctrl+K+L>	Удалить все закладки
<Ctrl+K+C>	Комментировать выделение
<Ctrl+K+U>	Снять комментарий с выделения

Новинка 2005

Несмотря на то что эти комбинации клавиш отличаются от привычных по инструменту Query Analyzer предыдущей версии SQL Server, мне нравится гибкость, реализуемая комбинацией <Ctrl+K>.

Просмотр плана выполнения запроса

Одной из самых существенных функций редактора запросов является возможность просматривать план выполнения запроса в графическом представлении (рис. 6.8). Планы выполнения делает еще более важными то, что SQL является описательным языком — он не указывает оптимизатору, как именно извлекать данные, а лишь описывает, какие данные ему нужны. Хотя некоторого улучшения производительности можно добиться за счет правильного моделирования инструкции, основная настройка выполняется правильной установкой индексов — именно они влияют на то, как оптимизатор запросов скомпилирует запрос. План выполнения запроса указывает серверу, как оптимизировать запрос, как взять на вооружение существующие индексы, запросить данные из других источников и создать объединения. Чтение плана выполнения запроса и понимание его взаимодействия со схемой базы данных является одновременно и наукой, и искусством.

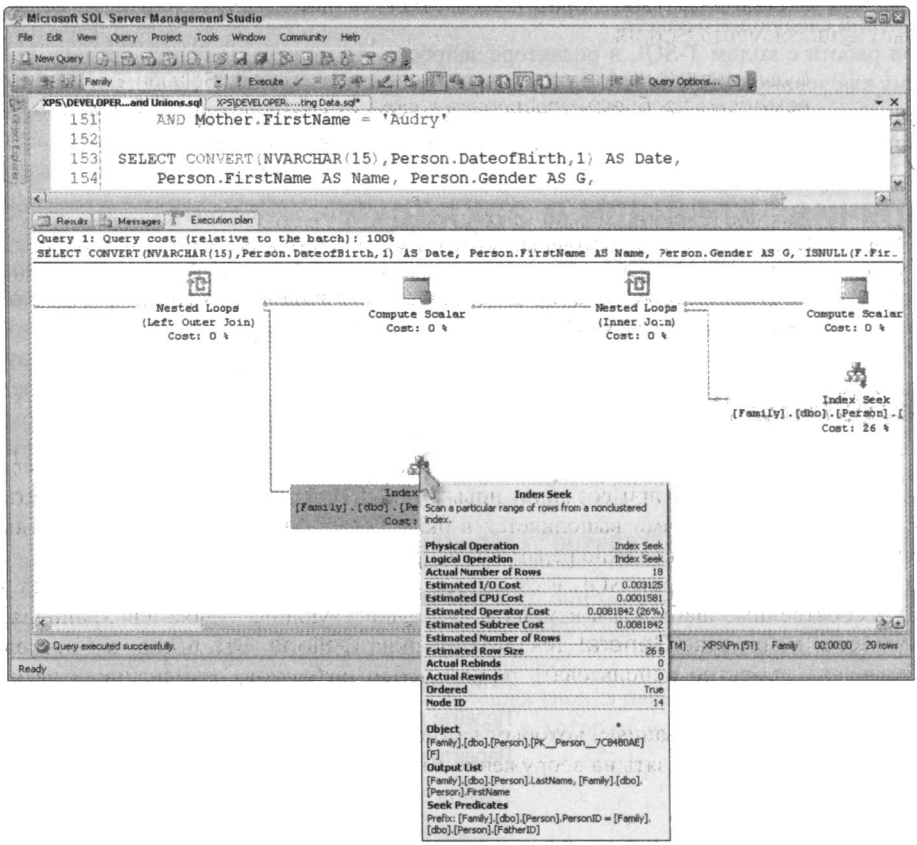
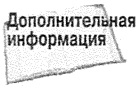


Рис. 6.8. Способность редактора запросов графически отображать план выполнения можно назвать самой важной его функцией



В главе 50 мы подробно поговорим о чтении плана выполнения запроса и настройке необходимых индексов.

Редактор запросов может отображать примерный план перед выполнением запроса или фактический уже после его выполнения.

Окно Solution Explorer

Дополнительное окно Solution Explorer позволяет организовать в проектах файлы и подключения. Его работа аналогична одноименному окну в Visual Studio. Вам не обязательно его использовать; для открытия и сохранения файлов достаточно использования команд меню File⇨Open и File⇨Save. В то же время, если вам приходится работать одновременно с несколькими проектами баз данных, это окно поможет организовать вашу жизнь или, по крайней мере, программный код. Окно Solution Explorer открывается из меню View; кнопку открытия этого окна можно добавить на панель инструментов с помощью команды Customize toolbar.

Чтобы использовать окно Solution Explorer для управления сценариями запросов, нужно вначале открыть окно редактора запросов. Для создания нового проекта воспользуйтесь контекстным меню элемента Scripts.



Будучи новой функцией утилиты Management Studio, Solution Explorer снимает некоторые проблемы управления кодом, существовавшие в более ранних версиях SQL Server.



Solution Explorer можно интегрировать с Visual Studio Source Safe или любыми другими сторонними утилитами управления, чтобы реализовать полноценное управление документами и их версиями. Более полную информацию по этому вопросу вы можете получить на Web-сайте книги по адресу `www.SQLServerBible.com`.

Введение в шаблоны

Шаблоны утилиты Management Studio могут оказаться полезными, так как они являются отличной стартовой точкой при создании новых типов кода, обеспечивая целостность программ. Управление шаблонами выполняется в окне Template Explorer. Для использования шаблона следует создать новое окно редактора запросов, указав данный шаблон.

Шаблоны хранят сценарии SQL в структуре папок. Это значит, что можно без труда создавать собственные шаблоны и изменять уже существующие. Папка для хранения шаблонов задается во вкладке General диалогового окна Options. Это позволяет нескольким разработчикам совместно использовать общий набор шаблонов, размещенный на какомлибо сетевом устройстве.

Разработчикам и организациям, которым важно поддерживать свои стандарты или стили разработки, я бы советовал взять на вооружение достоинства шаблонов Management Studio.

Резюме

Management Studio и Query Editor являются двумя самыми важными интерфейсами администраторов баз данных и разработчиков SQL Server. Освоение этих двух инструментов жизненно важно для успешной работы с SQL Server.

Ознакомившись с основами интерфейсов разработки, в следующей части книги мы займемся созданием баз данных, манипулированием данными и программированием процедур, функций и триггеров на языке T-SQL.

Манипулирование данными с помощью инструкции SELECT

Часть II

В этой части...

Глава 7
Основы выполнения запросов

Глава 8
Использование выражений и скалярных функций

Глава 9
Объединение данных

Глава 10
Включение данных с помощью подзапросов и CTE

Глава 11
Консолидация данных

Глава 12
Навигация по иерархическим данным

Глава 13
Использование полнотекстового поиска

Глава 14
Создание представлений

Глава 15
Работа с распределенными запросами

Глава 16
Модификация данных

Ключевое слово SELECT является самым "сильным" в SQL. Пожалуй, ни в одном языке программирования нет ключевого слова, такого же гибкого и значительного. С его помощью можно извлекать, перестраивать, объединять и группировать данные практически любым вообразимым способом. К тому же область его действия легко расширить с помощью ключевых слов INSERT, UPDATE и DELETE, позволяющих модифицировать данные. SQL Server позволяет даже расширить инструкцию SELECT с помощью языка XQuery, и этот вопрос мы рассмотрим в главе 31.

Часть II начинается с обзора основной логики обработки запросов, после чего мы перейдем к таким вопросам, как коррелированные подзапросы, запросы со сравнениями и распределенные запросы. Я посвятил десять глав инструкции SELECT по той причине, что понимание многочисленных параметров, связанных с этой инструкцией, и творческих методов, доступных при построении запросов, является ключом к тому, чтобы стать успешным разработчиком, архитектором или администратором SQL Server.

Часть II предназначена не только для начинающих. В этих десяти главах будет представлено ядро языка SQL. В части IV мы рассмотрим вопросы оптимизации, и второй стратегией теории оптимизации является использование хороших пакетных инструкций. В то же время настоящие десять глав расскажут вам, как оптимизировать базу данных путем написания хороших запросов.

ГЛАВА

7

Основы выполнения запросов

В этой главе...

Логический поток выполнения запроса

Усечение результирующего набора данных

Проектирование данных

Определение порядка сортировки

SQL — это язык взаимодействия с данными, однако получение единственно правильного ответа из гигабайтов реляционных данных покажется вам невозможным, пока вы не поймете логической цепочки задач в создании запроса.

Одним из самых важных моментов является понимание того, что SQL — *декларативный* язык. Это значит, что запрос SQL логически описывает вопрос для оптимизатора, который затем определяет самый лучший метод физического выполнения запроса. Как вы узнаете в следующих девяти главах, часто существует множество методов создания одного и того же запроса, но каждый из методов обычно оптимизируется в один и тот же план выполнения запроса. Это значит, что вы вольны выразить запрос SQL таким способом, который имеет для вас наибольший смысл. В некоторых случаях один из методов оказывается проще и быстрее остальных, и на эти нюансы мы также обратим внимание.

Запросы SQL не ограничены использованием единственного ключевого слова `SELECT`. Четыре команды языка манипулирования данными (DML) — `INSERT`, `SELECT`, `UPDATE` и `DELETE` — часто рассматривают как совершенно разные и не зависящие друг от друга. Я же рассматриваю весь запрос как единый структурированный метод манипулирования данными. Другими словами, лучше рассматривать эти четыре инструкции как четыре глагола, которые можно произносить для получения от языка SQL полной отдачи.

Нельзя думать, что создание запроса SQL ограничено рамками графического интерфейса. Многие разработчики, которые выросли на программе Access и всегда создавали запросы с помощью ее графического интерфейса, были удивлены, когда осознали всю силу, скрытую в полноценных запросах SQL.

В этой главе мы создадим простой запрос к одной таблице и на его основе установим, что логический порядок выполнения запроса критичен при создании как простых, так и сложных запросов. На основе этого фундамента в остальных главах данной части мы рассмотрим различные способы использования инструкции `SELECT`, которую я считаю самой элегантной, гибкой и сильной командой во всем мире компьютерных вычислений.

Основы создания запроса

Каждый может подойти к задаче создания запроса разными путями. Я, например, при создании кода SQL рассматриваю запрос с помощью логического метода, хотя многие подходят к нему с точки зрения конструктора запросов утилиты Management Studio. Еще один подход предлагает сам синтаксис инструкции SELECT. Для того чтобы проиллюстрировать декларативную природу запроса, следует сказать, что как бы вы ни поступили, при физическом выполнении запроса будет все равно использован другой, оптимизированный порядок.

Синтаксическая организация инструкции запроса

В своей базовой форме инструкция SELECT сообщает серверу, какие данные следует извлечь, в частности, какие столбцы и строки из каких таблиц получить и как сортировать данные.

Ниже приведен упрощенный синтаксис инструкции SELECT.

```
SELECT *, столбцы или выражения
  [FROM таблица]
     [JOIN таблица
          ON условие]
    [WHERE условия]
    [GROUP BY столбцы]
    [HAVING условия]
    [ORDER BY столбцы];
```

Инструкция SELECT начинается со списка столбцов или выражений. Как минимум обязательно наличие хотя бы одного выражения, все остальное — необязательно. Простейшая из возможных инструкций SELECT выглядит следующим образом:

```
SELECT 1;
```

Предложение FROM в инструкции SELECT собирает все источники данных в единый набор, над которым будет работать остальная часть инструкции. В предложении FROM может участвовать множество таблиц, которые ссылаются друг на друга с помощью одного из нескольких типов объединений.

Предложение WHERE фильтрует строки набора данных, собранного предложением FROM, на основе некоторых условий.

Агрегатные функции выполняют в наборе данных итоговые подсчеты. Предложение GROUP BY может группировать большие множества в несколько небольших подмножеств на основе значений столбцов, упомянутых в этом предложении. Затем агрегатные функции применяются к этим небольшим подмножествам данных, после чего результаты агрегатных функций фильтруются с помощью предложения HAVING.

Наконец, предложение ORDER BY определяет порядок сортировки результирующего набора данных.

Графическое представление инструкции запроса

Утилита SQL Server Management Studio содержит два основных инструмента формирования и отправки запросов: конструктор запросов (Query Designer) и редактор запросов (Query Editor). Конструктор запросов предлагает графический метод создания запроса. В то же время редактор запросов является идеальным средством разового извлечения данных, так как не содержит графики, указывающей пользователю, как именно он должен создавать запрос. В редакторе пользователь работает с кодом SQL настолько близко, насколько это возможно.

С точки зрения SQL Server не имеет значения, откуда именно исходит запрос. Каждая инструкция оценивается и обрабатывается как одна инструкция SQL.

Когда данные отбираются с помощью конструктора запросов, инструкции SQL можно вручную вводить и редактировать на третьей панели (рис. 7.1). Нижняя панель отображает результаты выполнения запроса в табличном или текстовом режиме; также на ней отображаются различные сообщения. В окне Object Browser отображается дерево всех объектов SQL Server, равно как и шаблоны для создания новых объектов с помощью кода.



Если в окне запроса SQL выделен какой-либо текст, то после нажатия клавиши <F5> или щелчка на кнопке Execute Query будет выполнена только эта часть запроса. Это отличный способ тестирования кода SQL по частям.

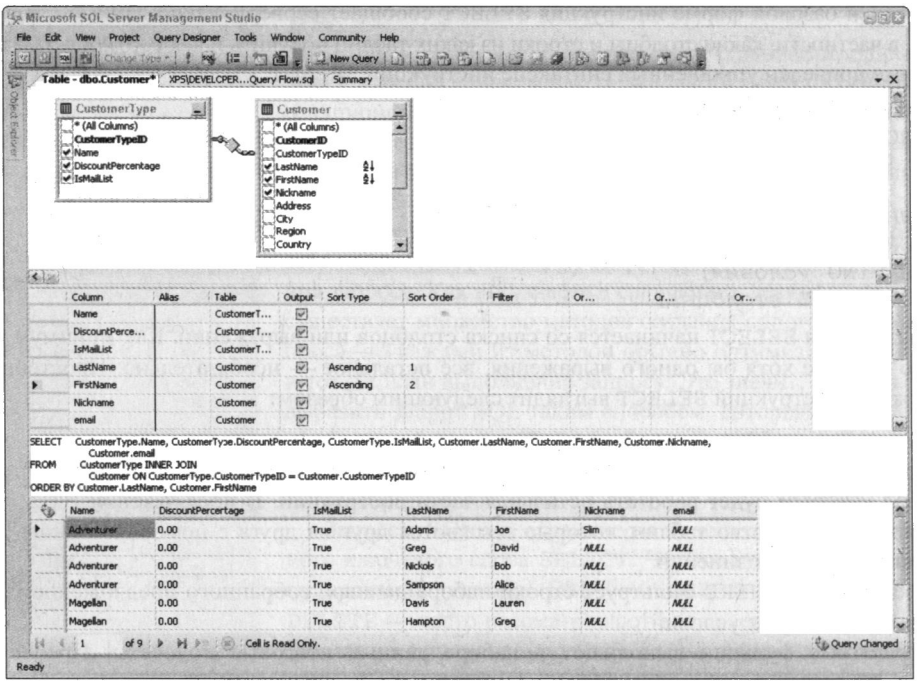


Рис. 7.1. Создание запроса отбора



Базой данных по умолчанию является `master`, хотя этот факт сильно зависит от настроек системы безопасности. Перед выполнением запроса убедитесь, что в комбинированном списке на панели инструментов выбрана пользовательская база данных; также для этого можно воспользоваться командой `USE` *база_данных*.

Логическая структура запроса

Лучше всего рассматривать инструкции DML языка SQL с точки зрения их логической организации. Поскольку SQL является декларативным языком, логическая организация может соответствовать, а может и не соответствовать физической схеме выполнения запроса на сервере. Логическая организация также чаще всего не сходна с синтаксической. Как бы там ни было, я рекомендую рассматривать запрос в следующем порядке.

1. **FROM**. Выполнение запроса начинается с подготовки изначального набора данных; порядок сбора источников данных указан в предложении FROM.
2. **WHERE**. Процесс фильтрации, определяемый в предложении WHERE, отбирает только те строки, для которых квалификационная характеристика содержит фразу "%First Aid%".
3. **Вычисление столбцов**. Как только строки доступны и отфильтрованы, из них отбираются и вычисляются столбцы, указанные в списке инструкции SELECT.



Структура выражений SQL детально рассматривается в главе 8.

4. **Консолидация**. При необходимости SQL может осуществлять консолидацию на множестве данных, таких как поиск среднего значения, группировка данных по значениям в заданном столбце, или преобразование результата в сводную (или перекрестную) таблицу.



Итоговые функции SQL будут рассмотрены в главе 11.

5. **ORDER BY**. Как только строки будут собраны предложением FROM и отфильтрованы предложением WHERE, они могут быть отсортированы в соответствии с предложением ORDER BY.
6. **Предикат**. После того как строки отобраны, вычисления выполнены и данные отсортированы в заданном порядке, SQL может ограничить результирующий набор только первыми несколькими строками или вернуть заданный набор строк.

Чем более сложная инструкция SELECT, тем сложнее становится разбор ее логической организации. Индексы и таблицы, доступные оптимизатору запросов, также влияют на построение *плана выполнения запроса*.

Как только вы начнете мыслить категориями инструкции SELECT, а не графического интерфейса пользователя, понимание структуры этой инструкции и умение читать план выполнения запроса поможет вам обойти все трудности отладки запроса.

Физическая структура запроса

SQL Server принимает инструкцию SELECT и вырабатывает план выполнения запроса. Как вы уже догадались, это не что иное, как порядок выполняемых операций (рис. 7.2).

Предложение FROM для выбора источников данных

Первым компонентом типичной инструкции SELECT является предложение FROM. В простых инструкциях отбора предложение FROM содержит всего одну таблицу. Однако это предложение может содержать и множество связанных между собой таблиц, подзапросы в качестве временных таблиц, а также представления. Максимальное количество таблиц, доступных одной инструкции SELECT, составляет 256.

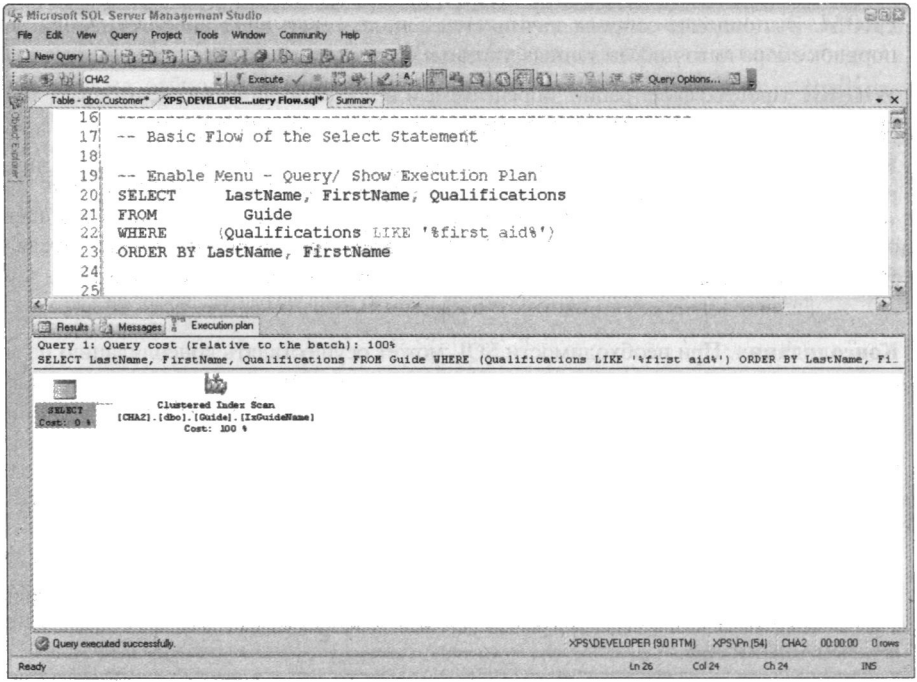


Рис. 7.2. Физический план выполнения запроса сильно отличается от синтаксической и логической структуры запроса

Предложение FROM закладывает фундамент всех последующих операций инструкции SELECT. Для того чтобы столбец таблицы содержался в результирующем наборе данных, или был доступен условиям предложения WHERE, или стал основой сортировки в предложении ORDER BY, эта таблица должна быть упомянута в предложении FROM.

Возможные источники данных

Язык SQL является чрезвычайно гибким и может принимать данные из множества различных типов источников, упомянутых в предложении FROM.

- Таблицы SQL Server.
- Подзапросы, выступающие в роли временных таблиц, также называемые *подвыборками* и *оперативными представлениями*. О них мы подробно поговорим в главе 10.
- Общие табличные представления (CTE), впервые введенные в SQL Server 2005, добавляют новые функции и форматирование в традиционные подзапросы.
- Представления, или хранимые инструкции SELECT, доступны предложению FROM так, будто они являются обычными таблицами. Представления мы подробно рассмотрим в главе 14.
- Определенные пользователем функции, возвращающие таблицы. Более полная информация о пользовательских функциях приведена в главе 22.
- Распределенные источники данных, перемещенные из других баз и приложений (например, Oracle, Excel или Access) с помощью функции `openquery()` и других, описанных в главе 15.

- Работа с источниками данных XML выполняется с помощью запросов Xquery. Более подробно эта тема освещена в главе 31.

Именованные диапазоны

Любой таблице в предложении FROM может быть присвоен именованный диапазон, или *псевдоним*. Как только у таблицы появляется псевдоним, по нему к ней можно обращаться в других предложениях инструкции SELECT. Ключевое слово AS является необязательным, и часто его обходят вниманием. В следующем примере доступ осуществляется к таблице Guide, но обращение к ней выполняется по псевдониму G:

```
-- FROM таблица [AS] переменная_диапазона
USE CHA2
SELECT G.LastName, G.FifstName
   FROM Guide AS G
```



На сайте книги (www.SQLServerBible.com) вы найдете примеры программ для всех глав этой книги, а также сценарии, создающие и заполняющие базы данных примеров.



На языке SQL команда USE задает текущую базу данных. Это программная версия выбора базы данных в утилите Management Studio.

Имя таблицы

Если имя объекта базы данных, такого как таблица или столбец, конфликтует с каким-либо ключевым словом SQL, вы можете указать серверу, что это именно имя объекта, заключив его в квадратные скобки. Таблица [Order] (заказ) в базе данных OBXKites является типичным примером использования ключевого слова в имени таблицы.

```
USE OBXKites
  SELECT OrderID, OrderDate
  FROM [Order]
```

Несмотря на то что считается дурным тоном включать в имена объектов базы данных пробелы, некоторые разработчики настроены по-другому. В данном случае при указании имени объекта также используются квадратные скобки. Таблица Order Details в базе данных примеров Northwind иллюстрирует это правило:

```
USE Northwind
SELECT OrderID, ProductID, Quantity
  FROM [Order Details]
```

Четырехкомпонентные имена таблиц

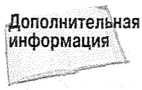
Полное и правильное имя таблицы складывается из четырех составных частей:

Сервер.База_данных.Схема.Таблица

Если таблица находится в текущей базе данных, то имена сервера и базы не обязательны. Хотя это и не обязательно, но считается хорошим тоном указывать явно имя схемы. Обычно схемой является dbo — это наследие предыдущих версий SQL Server, где все объекты принадлежали владельцам, а имя dbo представляло владельца базы данных.

Использование четырехкомпонентного имени позволяет многократно использовать план выполнения запроса. Указание имени владельца — это не только хорошая программистская практика, но и способ повышения быстродействия запросов.

Теперь, когда вы ознакомились с четырехкомпонентной формой имен объектов базы данных, во всех примерах книги в именах в явном виде будет указываться владелец.



Чтобы ближе ознакомиться с вопросами схем, областей определения и разрешений, обратитесь к главе 40. Многократное использование планов выполнения запросов рассматривается в главе 50.

Условия WHERE

Условия WHERE фильтруют набор данных, сформированный предложением FROM, и отбирают из него только те строки, которые войдут в результирующий набор данных. Условия могут ссылаться на данные в таблицах, выражения, встроенные скалярные функции SQL и пользовательские функции. В условиях WHERE могут также использоваться операторы сравнения и символы макроподстановки, перечисленные в табл. 7.1.



Одним из способов существенного повышения производительности баз данных с архитектурой "клиент/сервер" является разрешение ядру базы данных выполнять работу по ограничению числа возвращаемых строк, вместо того чтобы нагружать вызывающее приложение ненужными ему данными. В то же время, если структура базы данных для обнаружения нужных строк требует использования в предложении WHERE функций, производительность существенно упадет, так как функции будут выполняться над каждой из строк. Таким образом, правильно составленное предложение WHERE, основанное на хорошо спланированной структуре базы данных, является самым главным рычагом повышения производительности запросов в руках разработчика.

Таблица 7.1. Стандартные операторы сравнения

Описание	Оператор	Пример
Равно	=	Quantity=12
Больше	>	Quantity>12
Больше или равно	>=	Quantity>=12
Меньше	<	Quantity<12
Меньше или равно	<=	Quantity<=12
Не равно	<>, !=	Quantity<>12, Quantity!=12
Не меньше	!<	Quantity!<12
Не больше	!>	Quantity!>12



Операторы сравнения, содержащие восклицательный знак, не совместимы со стандартом ANSI. В частности, оператор <> допустим, а оператор != — нет.

Дополнительная информация

В дополнение к стандартным операторам сравнения, которые, без сомнения, вам знакомы, в языке SQL содержатся специальные операторы `between`, `in`, `like` и `is`. Первые три из них будут описаны в настоящем разделе. Тестирование отсутствующих значений, выполняемое оператором `is`, а также сама обработка пустых значений будут описаны в главе 8.



Лучшим способом найти какую-либо вещь является поиск именно ее, а не отсев всех вещей, которыми она не является. Гораздо проще найти в городе конкретную фирму, чем доказать, что ее не существует. То же справедливо и для поиска в базах данных. Доказательство того, что строка удовлетворяет условию, выполняется быстрее, чем отсев всех строк, не удовлетворяющих условию. В общем случае замена отрицательного условия `WHERE` положительным условием значительно повышает производительность запроса.

Использование условия `BETWEEN`

Условие `BETWEEN` проверяет значение на его принадлежность некоторому диапазону. В данном случае диапазон включает граничные значения. Например, условие `between 1 and 10` будет справедливо для чисел `1` и `10`. При использовании условия `between` первая граница диапазона должна быть меньше второй, так как в переводе на "человеческий" язык это условие звучит так: "Больше или равно первому значению и меньше или равно второму".

Чаще всего условие `between` используют с датами. В следующем примере (рис. 7.3) выполняется поиск всех событий из базы данных Cape Hatteras Adventures, произошедших с июля 2001 года. В начале программы база данных `CHA2` объявляется текущей, а затем к ней выполняется запрос:

```
USE CHA2
SELECT EventCode, DateBegin
  FROM dbo.Event
  WHERE DateBegin BETWEEN '07/01/01' AND '07/31/01';
```

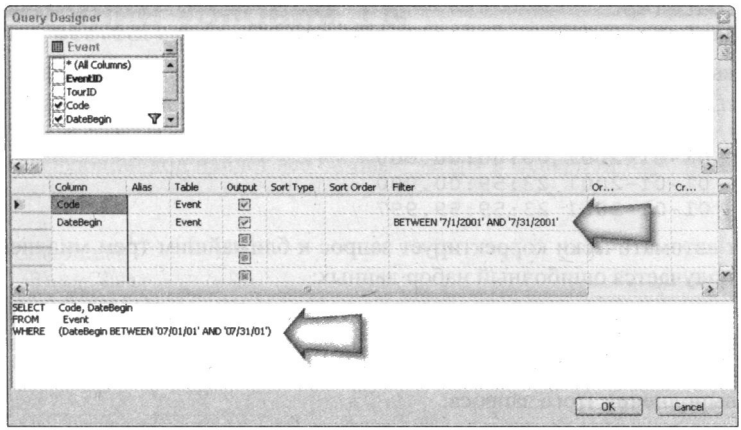


Рис. 7.3. Условия предложения `WHERE` в конструкторе запросов утилиты Management Studio можно поместить как на панель таблицы, так и на панель SQL

Результат выполнения запроса следующий:

```
EventCode  DateBegin
---------- -------------------------
01-006     2001-07-03 00:00:00.000
01-007     2001-07-03 00:00:00.000
01-008     2001-07-14 00:00:00.000
```

Приведенный в качестве примера запрос возвращает точный результат только в том случае, если даты хранятся без указания времени. В то же время в большинстве приложений дата и время извлекаются с помощью встроенной функции SQL Server `GetDate()`, которая одновременно извлекает обе величины. При этом точность извлечения времени составляет 3 миллисекунды. Таким образом, в данном случае приведенный в качестве примера запрос теоретически должен не включать в себя строки с датой и временем более 00:00:00.000 31/07/2001. Однако в случае одновременного хранения даты и времени последним известным временем для SQL Server будет 23:59:59:998, что и продемонстрировано в следующем примере. В этом примере мы вначале создаем базу данных для тестирования, заполняем ее несколькими строками данных, а затем выполняем запрос, иллюстрирующий рассматриваемый вопрос:

```
CREATE TABLE dbo.DateTest(
PK INT IDENTITY,
OrderDate DATETIME
)
go
INSERT dbo.DateTest(OrderDate)
  VALUES('1/1/01 00:00')
INSERT dbo.DateTest(OrderDate)
  VALUES('1/1/01 23:59')
INSERT dbo.DateTest(OrderDate)
  VALUES('1/1/01 11:59:59.995 pm')
INSERT dbo.DateTest(OrderDate)
  VALUES('1/2/01');
```

Следующий пример демонстрирует последнее доступное время дня:

```
SELECT *
  FROM dbo.DateTest
  WHERE OrderDate BETWEEN '1/1/1' AND '1/1/1 11:59:59.998 PM';
```

Результат выполнения этого запроса:

```
PK          OrderDate
----------- -----------------------
1           01-01-2001 00:00:00.000
2           01-01-2001 23:59:00.000
3           01-01-2001 23:59:59.997
```

SQL Server автоматически корректирует запрос к ближайшим трем миллисекундам, в результате чего получается ошибочный набор данных:

```
SELECT *
  FROM dbo.DateTest
  WHERE OrderDate BETWEEN '1/1/1' AND '1/1/1 11:59:59.999 PM';
```

Результат выполнения этого запроса:

```
PK          OrderDate
----------- -----------------------
1           01-01-2001 00:00:00.000
2           01-01-2001 23:59:00.000
3           01-01-2001 23:59:59.997
4           02-01-2001 00:00:00.000
```

Теперь удалим тестовую таблицу:

```
DROP TABLE DateTest
```

Мы видим, что последнее время диапазона последнего запроса было скорректировано до ближайших трех миллисекунд, и в итоге в результат ошибочно была включена строка, относящаяся к следующему дню.

Та же проблема возникает и при использовании типа данных `smalldatetime`, в котором время округляется до минуты. В результате условие предложения `WHERE column<=23:59:30` будет округлено до нуля часов следующего дня.

Следующий запрос из сценария `FamilyQueries.Sql` использует условие `between` для поиска матерей, которые родили детей менее чем через 9 месяцев после замужества.

В предложении `FROM` данный запрос собирает информацию о матерях, о датах заключения их брака и рождении детей из таблицы `person`. После этого предложение `WHERE` ограничивает результат теми строками, в которых поле `DateOfBirth` (дата рождения ребенка) попадает в конкретный временной диапазон:

```
SELECT Person.FirstName + ' ' +  Person.LastName AS Mother,
 Convert(Char(12), Marriage.DateOfWedding, 107) as Wedding,
 Child.FirstName + ' ' + Child.LastName as Child,
 Convert(Char(12), Child.DateOfBirth, 107) as Birth
FROM Person
 JOIN Marriage
   ON Person.PersonID = Marriage.WifeID
 JOIN Person Child
   ON Person.PersonID = Child.MotherID
WHERE Child.DateOfBirth
 BETWEENMarriage.DateOfWedding
   AND DATEADD(mm, 9, Marriage.DateOfWedding);
```

Результат выполнения данного запроса:

```
Mother           Wedding      Child           Birth
---------------- ------------ --------------- ------------
Alysia Halloway  Jan 01, 1975 James Halloway  May 24, 1975
```

Использование условия `IN`

Условие поиска `in` аналогично оператору сравнения `equals`, однако в данном случае ищется соответствие данным из списка. Если значение содержится в списке, то результатом выражения будет `true`. Например, если данные о регионе ввести в базу Cape Hatteras Adventures, то следующий код найдет все базовые лагеря в Северной Каролине (NC) и Западной Виргинии (WV):

```
USE CHA2
SELECT BaseCampname
   FROM dbo.BaseCamp
   WHERE Region IN('NC', 'WV');
```

Результат выполнения запроса будет следующим:

```
Result:
BaseCampName
-----------
West Virginia
Cape Hatteras
Asheville NC
```

Фактически условие запроса `in` является эквивалентом объединения оператором `or` нескольких сравнений `equals`. Предыдущий запрос можно было бы заменить следующим:

```
USE CHA2
SELECT BaseCampname
```

```
  FROM dbo.BaseCamp
  WHERE Region = 'NC'ORRegion = 'WV';
```

Как видим, результат его выполнения не отличается от предыдущего:

```
BaseCampName
-----------
West Virginia
Cape Hatteras
Asheville NC
```

Оператор `in` можно комбинировать с оператором `not` для исключения соответствующих строк. Например, предложение `where not in ('NC', 'SC')` вернет все строки базовых лагерей за исключением находящихся в Северной и Южной Каролине:

```
USE CHA2
SELECT BaseCampname
  FROM dbo.BaseCamp
  WHERE Region NOT IN ('NC', 'SC');
```

Результат будет следующим:

```
BaseCampName
-----------
FreePortFt
LauderdaleWest
Virginia
```

Очень трудно доказать обратное, особенно если в списке содержится пустое значение `NULL`. Пустое значение как бы неизвестно, поэтому потенциально любое значение может находиться на его месте. Следующая инструкция демонстрирует, как включение в список пустого значения делает невозможным доказательство отсутствия в нем буквы `'A'`:

```
SELECT 'IN' WHERE 'A' NOT IN ('B',NULL)
```

Эта инструкция не вернет результат, поскольку пустое значение потенциально может быть буквой `'A'`. Поскольку SQL не может логически доказать отсутствия этой буквы в списке, предложение `WHERE` возвращает значение `false`. Когда бы оператор `not in` ни использовался со списком, содержащим пустое значение, любая строка будет оценена как не удовлетворяющая условию.

Совершенно очевидно, что оператор `in` является довольно мощным подспорьем. Несмотря на то что в приведенных выше примерах содержались только списки с предопределенными значениями, при комбинировании этого оператора с подзапросами (о них мы поговорим в следующей главе), генерирующими динамический список, его использование решает массу проблем.

Использование условия `LIKE`

Условие `like` использует символы макроподстановки для поиска строк, соответствующих заданному шаблону. Однако эти символы макроподстановки отличаются от использующихся в DOS, с которыми вы, наверное, хорошо знакомы. В табл. 7.2 представлены символы макроподстановки как SQL, так и MS-DOS.

В следующем примере условие `like` используется для поиска всех товаров, названия которых начинаются с `'Air'`:

```
USE OBXKites
SELECT ProductName
  FROM dbo.Product
  WHERE ProductName LIKE 'Air%';
```

Таблица 7.2. Символы макроподстановки SQL

Описание	Символ SQL	Символ MS-DOS	Пример
Несколько символов	%	*	'Able' LIKE 'A%'
Один символ	_	?	'Able' LIKE 'Abl_'
Вхождение в диапазон символов	[]	Не используется	'a' LIKE '[a-g]' 'a' LIKE '[abcdefg]'
Невхождение в диапазон символов	[^]	Не используется	'a' LIKE '^[w-z]' 'a' LIKE '[^wxyz]'

Результат будет следующим:

```
ProductName
------------------
Air Writer 36
Air Writer 48
Air Writer 66
```

Следующий запрос ищет названия всех товаров, начинающихся с букв диапазона от 'a' до 'd' включительно:

```
SELECT ProductName
  FROM Product
  WHERE ProductName LIKE  '[a-d]%';
```

Результат его выполнения будет следующим:

```
ProductName
--------------------------------------------------
Basic Box Kite 21 inch
Dragon Flight
Chinese 6" Kite
Air Writer 36
Air Writer 48
Air Writer 66
Competition 36"
Competition Pro 48"
Black Ghost
Basic Kite Flight
Advanced Acrobatics
Adventures in the OuterBanks
Cape Hatteras T-Shirt
```

Для поиска по шаблонам, содержащим символы макроподстановки, можно использовать два метода: либо заключить шаблон в квадратные скобки, либо поместить перед ним символ Escape. Нюанс последнего подхода состоит в том, что символ Escape должен быть определен в выражении like.

В следующих двух примерах выполняется поиск фразы "F-15" в таблице product базы данных OBXKites. В первом запросе дефис заключается в квадратные скобки, что указывает на то, что это символ макроподстановки. Во втором запросе символ амперсанда (&) определяется как символ Escape:

```
SELECT ProductCode, ProductName
  FROM Product
  WHERE ProductName LIKE '%F[-]15%';

SELECT ProductCode, ProductName
```

```
  FROM Product
  WHERE ProductName LIKE '%F&-15%' ESCAPE '&';
```

Оба запроса возвращают один и тот же результат:

```
ProductCode      ProductName
---------------- ------------
1013             Eagle F-15
```



Из двух методов поиска с использованием символов макроподстановки квадратные скобки являются специфичным методом T-SQL, не соответствующим стандарту ANSI SQL. В то же время метод Escape является одновременно соответствующим стандарту ANSI SQL и, следовательно, переносимым.

При использовании оператора `like` учтите, что порядок сортировки символов в базе данных должен различать символы верхнего и нижнего регистра и соответствовать предусмотренному вами в запросе. При желании вы можете использовать необязательное ключевое слово `collate`, чтобы явно определить порядок сопоставления, используемый в операторе `like`.



Несмотря на то что оператор `like` может оказаться исключительно полезным, его использование может сказаться на производительности. Индексы основываются на первых символах столбцов, а не на фразах в их середине. Если оказалось, что приложению часто нужно использовать оператор `like`, то лучше активизируйте полнотекстовую индексацию. Это метод индексации, который принимает в расчет взвешенные комбинации слов и в возвращаемой таблице указывает степень соответствия найденной фразы заданной фразе. Более подробно о полнотекстовой индексации вы узнаете в главе 13.

Множественные условия WHERE

В предложении `WHERE` можно комбинировать множество условий с помощью логических булевых операторов `and`, `or` и `not`. Как и в математических операторах, в булевых логических операторах существует свой порядок приоритетов. Наивысший приоритет имеет оператор `and`, далее следует `or` и, наконец, `not`. Рассмотрим пример:

```
SELECT ProductCode, ProductName
  FROM dbo.Product
  WHERE
      ProductName LIKE  'Air%'
   OR
      ProductCode BETWEEN '1018' AND '1020'
   AND
      ProductName LIKE '%G%';
```

Результат будет следующим:

```
ProductCode      ProductName
---------------- --------------------
1009             Air Writer 36
1010             Air Writer 48
1011             Air Writer 66
1019             Grand Daddy
1020             Black Ghost
```

Если в выражении использовать скобки, то результат запроса радикально изменится:

```
SELECT ProductCode, ProductName
  FROM dbo.Product
  WHERE
     (ProductName LIKE  'Air%'
```

```
   OR
      ProductCode BETWEEN '1018' AND '1020')
   AND
      ProductName LIKE '%G%';
```

Теперь результат выглядит так:

```
ProductCode      ProductName
---------------- --------------------
1019             Grand Daddy
1020             Black Ghost
```

Несмотря на то что два приведенных запроса очень похожи, в первом из них использовался естественный порядок приоритетов булевых операторов, т.е. `and` вычислялся перед `or`. При этом оператор `or` включил в результаты товары Air Writer.

Во втором запросе с помощью скобок была явно задана последовательность булевых операторов. Оператор `or` объединил все товары Air Writer с товарами, имеющими коды 1018, 1019 и 1020. После этого оператор `and` отобрал из этого списка только те товары, в названиях которых содержалась буква `g`. Только товары с кодами 1019 и 1020 прошли оба этих теста.



При программировании сложных булевых или математических выражений явно определяйте последовательность операций с помощью скобок, и вы сможете избежать массы недоразумений и ошибок, связанных с вашими предположениями о приоритетах.

SELECT...WHERE

Это удивительно, но при использовании в инструкции `SELECT` предложения `WHERE` совершенно не обязательно использование предложения `FROM` или каких-либо ссылок на таблицы. Инструкция `SELECT` без предложения `FROM` выдает одну строку.

Например, результатом инструкции

```
SELECT 'abc'
```

будет строка

```
abc
```

Предложение `WHERE` в нетабличной инструкции `SELECT` служит ограничением, применяемым ко всей инструкции. Если условие `WHERE` истинно, то инструкция `SELECT` будет работать, как и предполагалось.

Результатом инструкции

```
SELECT 'abc' WHERE 1>0
```

будет та же строка

```
abc
```

Если же условие `WHERE` ложно, то инструкция `SELECT` вообще не выполняется:

```
DECLARE @test NVARCHAR(15) ;
SET @test = 'z';
SELECT @test = 'abc' WHERE 1<0;
SELECT @test;
```

Результатом будет символ `'z'`. По своей функции предложение `WHERE` в нетабличной инструкции `SELECT` является сокращенным аналогом оператора `IF`. Следующий пример является полным аналогом предыдущего:

```
DECLARE @test NVARCHAR(15);
SET @test = 'z';
```

```
IF 1<0SELECT @test = 'abc';
SELECT @test;
```

Результатом этого кода также будет символ 'z'.

Упорядочение результирующего набора данных

Данные в таблице SQL имеют форму неупорядоченного списка. Основной задачей первичного ключа является уникальная идентификация строк, а совсем не их упорядочение. Некоторые реализации СУБД могут представлять таблицы в порядке, поддерживаемом первичным ключом. Однако лучше все же не надеяться на такое поведение. Если явно не задать предложение ORDER BY, то порядок строк в результирующем наборе данных может оставаться неопределенным.

Грубо говоря, если предложение ORDER BY отсутствует, SQL Server вернет строки в том порядке, в котором они извлекались из таблицы. Если исходная таблица имела кластеризованный индекс, то порядок результирующего набора данных будет соответствовать ему. Некоторые логические операции сортируют данные для своей поддержки. Например, некоторые объединения сортируют данные так, чтобы объединение было легче выполнить. Таким образом, даже в отсутствии предложения ORDER BY результирующий набор данных может оказаться отсортированным. И все же, если данные должны иметь конкретный порядок, лучше явно задать его в предложении ORDER BY, как показано на рис. 7.4.

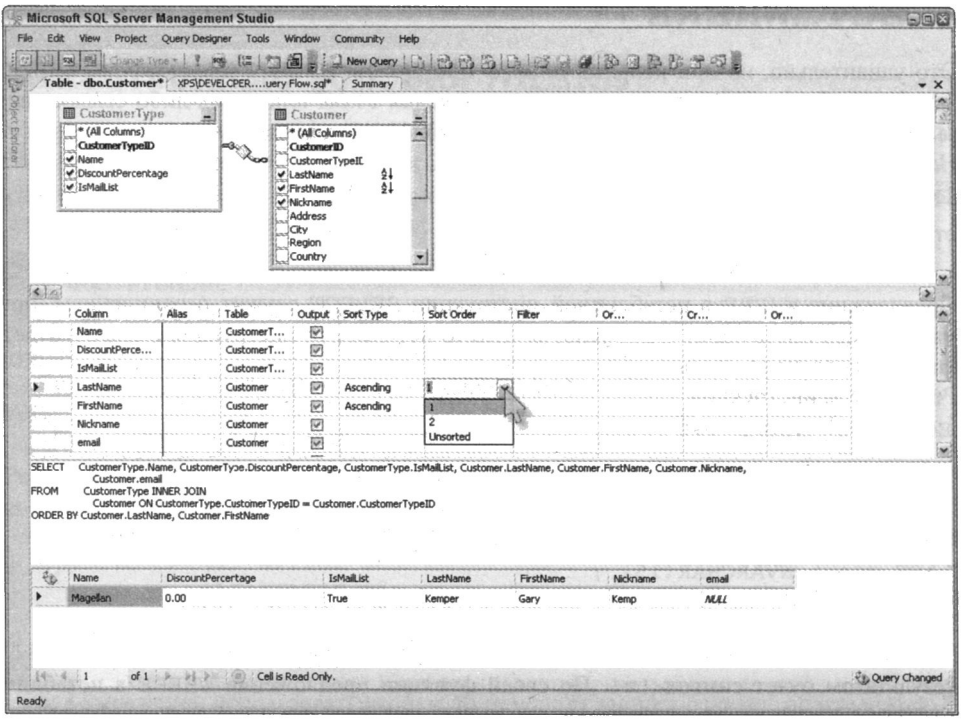


Рис. 7.4. В конструкторе запросов утилиты Management Studio вы можете задать порядок сортировки, щелкнув на кнопке Ascending *или* Descending *панели инструментов или указав порядок на панели* Grid

SQL позволяет выполнять сортировку по множеству столбцов, и ими не обязательно должны быть столбцы, извлекаемые инструкцией SELECT. Это предоставляет достаточную гибкость в определении столбцов сортировки.

Определение порядка сортировки с помощью имен столбцов

Простейшим способом сортировки результирующего набора данных является явное указание порядка столбцов, по которым выполняется упорядочение:

```
USE CHA2
SELECT FirstName, LastName
  FROM dbo.Customer
  ORDER BY LastName, FirstName;
```

Результат будет следующим:

```
FirstName      LastName
-------------- --------------------
Joe            Adams
Missy          Anderson
Debbie         Andrews
Dave           Bettys
...
```



Предложение ORDER BY и порядок столбцов в списке отбора полностью независимы друг от друга.

Определение порядка сортировки с помощью выражений

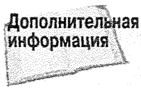
При сортировке по выражению все оно должно быть полностью повторено в предложении ORDER BY. Это не приводит к снижению производительности, поскольку оптимизатор запросов достаточно умен, чтобы не повторять одно и то же вычисление дважды:

```
SELECT LastName + ', ' + FirstName
  FROM dbo.Customer
  ORDER BY LastName + ', ' + FirstName;
```

Результат будет следующим:

```
FullName
----------------------
Adams, Joe
Anderson, Missy
Andrews, Debbie
Bettys, Dave
...
```

Использование выражений в предложении ORDER BY может решить множество проблем. Например, некоторые разработчики хранят заголовки в двух столбцах: в одном из них содержится полное название, а в другом — избавленное от ведущего предлога "The". С точки зрения производительности такая денормализация может оказаться полезной, но в то же время использование выражения CASE в предложении ORDER BY позволило бы выполнять корректную сортировку и без дублирования заголовка.



Полный синтаксис выражения CASE будет рассмотрен в главе 8.

База данных примеров Aesop's Fables содержит список заголовков. Если столбец Title содержит предлог "The", то выражение CASE удаляет его из него и передает в усеченном виде предложению ORDER BY:

```
USE Aesop;
SELECT Title, Len(FableText) AS TextLength
  FROM Fable
  ORDER BY
    CASE
      WHEN SubString(Title, 1,3) = 'The'
        THEN SubString(Title, 5, Len(Title)-4)
      ELSE Title
    END;
```

Результат запроса будет следующим:

```
FableName                          TextLength
---------------------------------- -----------
Androcles                          1370
The Ant and the Chrysalis          1087
The Ants and the Grasshopper       456
The Ass in the Lion's Skin         465
The Bald Knight                    360
The Boy and the Filberts           435
The Bundle of Sticks               551
The Crow and the Pitcher           491
...
```

Определение порядка сортировки с помощью псевдонимов столбцов

В качестве альтернативы для явного задания столбцов в предложении ORDER BY можно использовать их псевдонимы. Этот метод предпочтительнее, так как значительно облегчает чтение программы. Обратите внимание на то, что в следующем примере сортировка выполняется по убыванию (предикат DESC), а не по возрастанию, как принято по умолчанию:

```
SELECT LastName + ', ' + FirstName as FullName
  FROM dbo.Customer
  ORDER BY FullName DESC
```

Результат будет следующим:

```
FullName
-------------
Zeniod, Kent
Williams, Larry
Valentino, Mary
Spade, Sam
...
```

Псевдонимы разрешено использовать в предложениях ORDER BY, но не в предложениях WHERE, так как последнее логически выполняется раньше в запросе, чем определяется сам псевдоним. В то же время предложение ORDER BY является последней логической операцией, следующей за сбором столбцов и псевдонимов.

Определение порядка сортировки с помощью порядковых номеров столбцов

В предложении ORDER BY вместо имен столбцов могут использоваться их порядковые номера в запросе, однако я не рекомендую использовать этот метод. Если в начале инструкции SELECT изменится состав или порядок столбцов, то и предложение ORDER BY будет работать по-другому. Порядковые номера столбцов можно использовать в сложных запросах объединения, о которых мы поговорим в главе 9. В следующем примере продемонстрировано использование порядковых номеров столбцов для сортировки:

```
SELECT LastName + ', ' + FirstName AS FullName
  FROM dbo.Customer
  ORDER BY 1
```

Результат будет следующим:

```
FullName
----------------------
Adams, Joe
Anderson, Missy
Andrews, Debbie
Bettys, Dave
...
```

Упорядочение и порядок сопоставления

Установленный порядок сопоставления, применяемый в SQL Server 2005, принципиально важен для правильной сортировки данных. Кроме определения используемого алфавита, порядок сопоставления указывает, будут ли учитываться при упорядочении регистр символов и прочие характеристики алфавита. Например, если порядок сопоставления учитывает регистр символов, строчные буквы будут упорядочены перед прописными. Следующая функция поможет вам узнать установленные на сервере параметры порядка сопоставления и используемые в настоящее время:

```
SELECT * FROM ::fn_helpcollations()
```

Результат будет следующим:

```
name                  description
--------------------- -------------------------
Albanian_BIN          Albanian, binary sort
Albanian_CI_AI        Albanian, case-insensitive,
                      accent-insensitive,
                      kanatype-insensitive, width-nsensitive
Albanian_CI_AI_WS     Albanian, case-insensitive,
                      accent-insensitive,
                      kanatype-insensitive, width-sensitive
...
SQL_Latin1_General_CP1_CI_AI
                      Latin1-General, case-insensitive,
                      accent-insensitive,
                      kanatype-insensitive, width-insensitive
                      for Unicode Data, SQL Server Sort Order
                      54 on Code Page 1252 for non-UnicodeData
...
```

Следующий запрос сообщит вам о порядке сопоставления, используемом в настоящий момент на сервере:

```
SELECT SERVERPROPERTY('Collation') AS ServerCollation
```

Результат может быть следующим:

```
ServerCollation
-----------------------
SQL_Latin1_General_CP1_CI_AS
```

В то время как порядок сопоставления сервера определяется во время установки, это свойство базы данных или столбца может быть установлено с помощью ключевого слова `collate`. В следующем фрагменте порядок сопоставления в базе данных `Family` изменяется на зависимый от регистра символов:

```
ALTER DATABASE Family
  COLLATE SQL_Latin1_General_CP1_CS_AS
SELECT DATABASEPROPERTYEX(Family,'Collation')
  AS DatabaseCollation
```

Результат будет следующим:

```
DatabaseCollation
---------------------------------
SQL_Latin1_General_CP1_CS_AS
```

SQL Server устанавливает порядок сопоставления не только на уровне сервера, базы данных и столбца. Порядок сопоставления может устанавливаться на уровне отдельного запроса. Следующий запрос будет использовать для упорядочения датский порядок сопоставления без учета акцентов и регистра символов:

```
SELECT *
  FROM dbo.Product
  ORDER BY ProductName
    COLLATE Danish_Norwegian_CI_AI
```

Не все запросы нуждаются в сортировке, но в тех, в которых она нужна, предложение `ORDER BY` в сочетании со всеми возможными порядками сопоставления обеспечивают максимальную гибкость при упорядочении результирующего набора данных.

SELECT DISTINCT

Первым предикатом, используемым в сочетании с ключевым словом `SELECT`, является `DISTINCT`. Он исключает дублирование строк в результирующем наборе данных запроса. Эти дублирования оцениваются на уровне столбцов результирующего набора данных, а не исходных таблиц. Противоположную функцию выполняет предикат `ALL`; так как он используется по умолчанию, в запросах его обычно игнорируют.

В следующем примере демонстрируется различие между предикатами `DISTINCT` и `ALL`. Объединения будут описаны в главе 9, сейчас же отметим, что ключевое слово `JOIN` между именами таблиц `tour` и `event` генерирует строку каждый раз, когда тур проходит как событие. Так как инструкция `SELECT` возвращает только столбец `tourname` (название тура), это отличный пример дублирования строк, которое можно будет устранить с помощью предиката `DISTINCT`:

```
SELECT ALL TourName
  FROM Event
    JOIN Tour
       ON Event.TourID = Tour.TourID
```

Результат будет следующим:

```
TourName
-----------------------------------
Amazon Trek
Amazon Trek
Appalachian Trail
Appalachian Trail
Appalachian Trail
Bahamas Dive
Bahamas Dive
Bahamas Dive
Gauley River Rafting
Gauley River Rafting
Outer Banks Lighthouses
Outer Banks Lighthouses
Outer Banks Lighthouses
Outer Banks Lighthouses
Outer Banks Lighthouses
Outer Banks Lighthouses
```

А теперь выполним тот же запрос с предикатом DISTINCT:

```
SELECT DISTINCT TourName
  FROM Event
    JOIN Tour
       ON Event.TourID = Tour.TourID
```

Результат на этот раз будет другим:

```
TourName
--------------------------------------------------
Amazon Trek
Appalachian Trail
Bahamas Dive
Gauley River Rafting
Outer Banks Lighthouses
```

В то время как первый запрос вернул шестнадцать строк, предикат DISTINCT, использованный во втором запросе, исключил дублирование и вернул только пять строк.



Функционально предикат DISTINCT в SQL Server отличается от предиката DISTINCTROW, используемого в Microsoft Access, поскольку последний исключает дублирование, основываясь на данных исходных таблиц, а не результирующего набора данных.

Инструкция SELECT DISTINCT функционирует так, будто во всех столбцах результирующего набора данных установлена группировка предложением GROUP BY (подробно о ней вы узнаете в главе 11). Сравнивая план выполнения двух предыдущих запросов (рис. 7.5), мы явно видим работу предиката DISTINCT в качестве потокового агрегирования. Таким образом, предикат DISTINCT создает дополнительное действие в плане выполнения запроса. Однако при этом оказывается небольшое влияние на производительность (исследования показали, что на выполнение операции потокового агрегирования затрачивается всего 0,000006% общего времени выполнения запроса). Если уникальность строк логически необходима, не избегайте предиката DISTINCT только из-за его влияния на производительность.

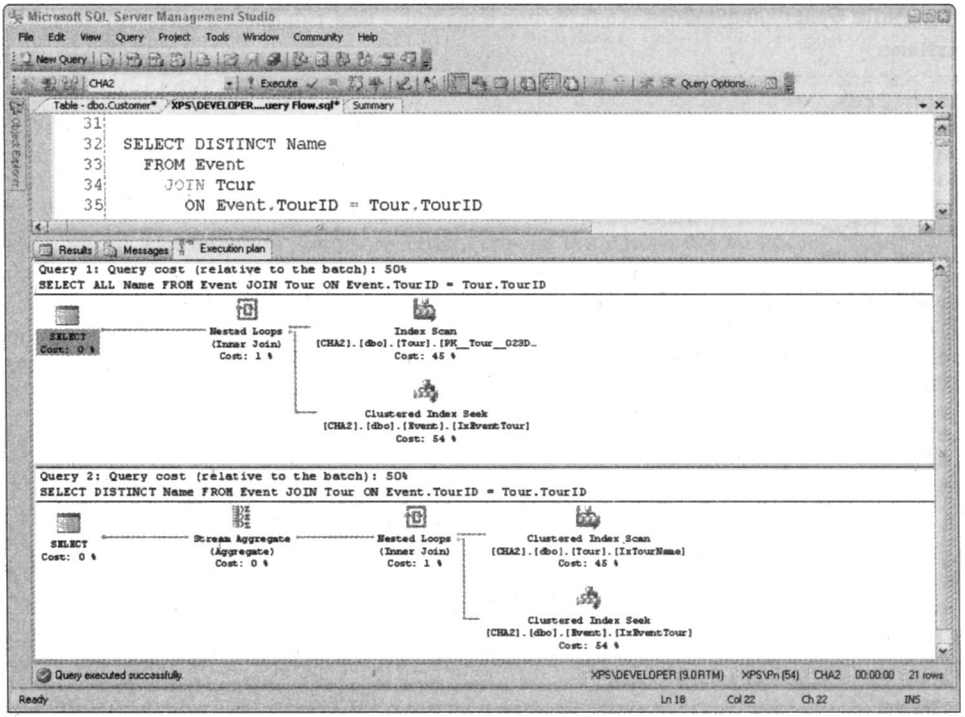


Рис. 7.5. Сравнение планов выполнения двух запросов выявило операцию потокового агрегирования (Stream Aggregate), которую выполняет предикат DISTINCT, чтобы не допустить дублирование строк

Ранжирование

По определению инструкция SELECT работает с наборами данных, однако иногда пользователя интересуют только первые несколько строк этого набора. Для таких ситуаций в SQL Server предусмотрено несколько способов фильтрации результатов и выявления экстремальных строк.

TOP

Как уже говорилось ранее, SQL Server по умолчанию возвращает в инструкции SELECT все строки результирующего набора данных. Необязательный предикат TOP указывает серверу возвращать только определенное количество строк (в абсолютном или процентном выражении), основываясь на заданном параметре (рис. 7.6).

Предикат TOP работает рука об руку с предложением ORDER BY, так как именно оно определяет, какие строки будут первыми в результирующем наборе данных. Если же в инструкции SELECT отсутствует предложение ORDER BY, то предикат TOP все равно отработает, возвращая заданное количество строк неупорядоченного набора данных.

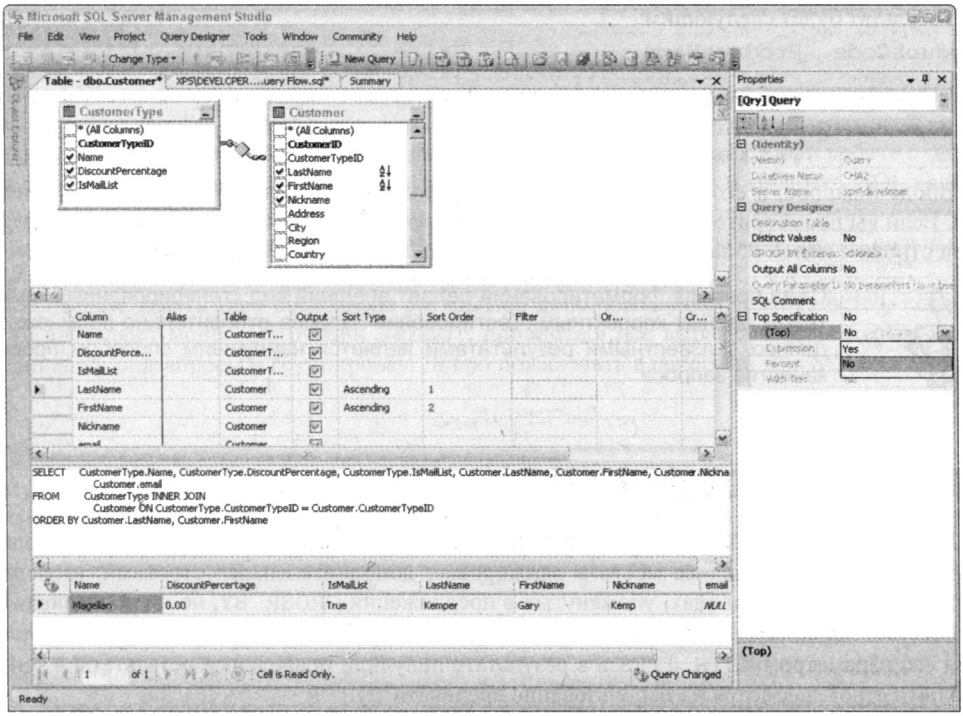


Рис. 7.6. Предикат TOP устанавливается в утилите Management Studio на странице свойств запроса

Отличным полигоном для испытания предиката TOP может стать база данных OBXKites. Следующий запрос найдет 3% самых больших цен в таблице `price`. В таблице `price` для каждого товара может быть указано несколько цен, выбор которых основан на столбце `EffectiveDate`:

```
SELECT TOP 3 PERCENT Code, ProductName, Price,
    CONVERT(VARCHAR(10),EffectiveDate,1) AS PriceDate
  FROM Product
    JOIN Price ON Product.ProductID = Price.ProductID
  ORDER BY Price DESC
```

В результате мы получим следующий список:

```
ProductCode  ProductName              Price      PriceDate
------------ ---------------------- ---------- ----------
1018         Competition Pro 48"    284.9500   05/01/01
1018         Competition Pro 48"    264.9500   05/01/02
1017         Competition 36"        245.9500   05/20/03
1017         Competition 36"        225.9500   05/01/01
```

А приведенный ниже запрос вернет три самые низкие цены в той же таблице.

```
SELECT TOP 3 Code, ProductName, Price,
    CONVERT(VARCHAR(10),EffectiveDate,1) AS PriceDate
  FROM Product
    JOIN Price ON Product.ProductID = Price.ProductID
  ORDER BY Price
```

Результат будет следующим:

```
ProductCode  ProductName              Price      PriceDate
------------ ---------------------- ---------- ----------
1044         OBX Car Bumper Sticker .7500      05/01/01
1045         OBX Car Window Decal   .7500      05/20/01
1045         OBX Car Window Decal   .9500      05/20/02
```

Вроде и запрос, и его результаты выглядят хорошо, но, к сожалению, результаты неверные. Если вы посмотрите на исходные данные, отсортированные по цене, то найдете не одну, а три строки с ценой 95 центов. Эту проблему решает параметр `WITH TIES`.



Сама природа форматирования делает внешний вид сгенерированных компьютером данных корректным. Тестирование запроса относительно подмножества данных с известными результатами является наилучшим способом проверки качества запроса.

Параметр `WITH TIES`

Параметр `WITH TIES` исключительно важен для предиката `TOP`. Он позволяет дополнить строку, занявшую последнее место в ранжировке, дополнительными строками, имеющими такое же значение в столбцах, упомянутых в предложении `ORDER BY`, но не попадающими в количество, заданное в предикате `TOP`. В следующей версии предыдущего запроса мы дополнили его параметром `WITH TIES` и в результате получили правильные результаты в количестве пяти строк, несмотря на использование предиката `TOP 3`:

```
SELECT TOP 3 WITH TIES Code, ProductName, Price,
    CONVERT(VARCHAR(10),EffectiveDate,1) AS PriceDate
  FROM Product
    JOIN Price ON Product.ProductID = Price.ProductID
  ORDER BY Price
```

Ниже приведен результат этого запроса.

```
ProductCode  ProductName              Price      PriceDate
------------ ---------------------- ---------- ----------
1044         OBX Car Bumper Sticker .7500      05/01/01
1045         OBX Car Window Decal   .7500      05/20/01
1045         OBX Car Window Decal   .9500      05/20/02
1041         Kite Fabric #6         .9500      05/01/01
1042         Kite Fabric #8         .9500      05/01/01
```



Если вы переходите на SQL Server с Microsoft Access, то учтите, что во второй СУБД параметр `WITH TIES` добавлялся автоматически к предикату `TOP`. В SQL Server это не так.



Предикат `TOP` является расширением стандарта ANSI SQL; он не переносим. Если базу данных придется переносить на другую платформу, то использование предиката `TOP` может вызвать проблемы преобразования. В противоположность этому переменная `rowcount` является переносимой.



В SQL Server 2005 появилось несколько новых функций ранжирования, в том числе `rownumber()`, `rank()`, `denSerank()` и `ntile()`. Эти функции могут использоваться как дополнительные команды предиката `TOP` (подробнее об этом речь пойдет в главе 8).

Резюме

Язык SQL создавался как инструмент манипулирования данными, а инструкция SELECT является "королевой" в этой области. Несмотря на то что в нескольких примерах этой главы использовались объединения, та же методика может быть использована и при работе с одной таблицей. В простой инструкции SELECT скрыта неимоверная мощь и гибкость. SQL является описательным языком — вы всего лишь формулируете вопрос, а тем, как именно будет получен на него ответ, занимается уже оптимизатор запросов на сервере. Так что вы вольны использовать любой стиль при формировании запроса.

Следующие девять глав посвящены раскрытию дополнительных возможностей, которые только добавляют мощь инструкции SELECT, привнося в нее выражения, множество типов объединений, подзапросы и группировку. Добро пожаловать в пакетный мир SQL!

ГЛАВА

8

Использование выражений и скалярных функций

В этой главе...

Работа с выражениями и скалярными функциями

Использование в запросах логики

Работа с пустыми значениями, строками и датами

Когда мой сын Дейвид был младше, он конструировал невообразимых монстров из отдельных блоков K'NEX. Если вы еще не знаете, что такое K'NEX, найдите соответствующий сайт с помощью Google и посмотрите, какие невероятные вещи дети способны создавать с его помощью.

Что делает конструктор K'NEX столь прекрасным, так это то, что любой блок можно вставить в другой блок. Такая взаимосвязанность отдельных блоков и обеспечивает конструктору гибкость. Аналогично, взаимосвязанность выражений и функций SQL обеспечивает этому языку такую силу и гибкость.

Выражения могут извлекать данные из подзапросов, обрабатывать сложную логику, преобразовывать типы и манипулировать данными. Если основной секрет мастерства разработчика баз данных заключается в умении создавать запросы, то определенно главным его арсеналом будут выражения и скалярные функции.

Выражение — это некоторая комбинация констант, функций и формул, возвращающая одно значение. Выражения могут быть такими простыми, как предопределенное число, и такими сложными, как выражения `CASE`, включающие в себя массу функций и формул.

Выражения могут быть задействованы во множестве элементов синтаксиса SQL. Практически везде, где может использоваться значение, можно вставить выражение. Это могут быть значения столбцов, предложения `JOIN`, `WHERE`, `HAVING` и `ORDER BY`. В то же время выражения нельзя подставлять вместо имен объектов, таких как таблицы или столбцы.



Инструкции и выражения SQL не зависят от регистра символов — они будут выполнены одинаково, если будет использован верхний или нижний регистр или их комбинация.

Создание выражений

Выражения SQL можно создать практически из бесконечного списка констант, переменных, операторов и функций, как показано в табл. 8.1. На рис. 8.1 проиллюстрировано создание выражения с помощью графического интерфейса и назначение столбцам псевдонимов.

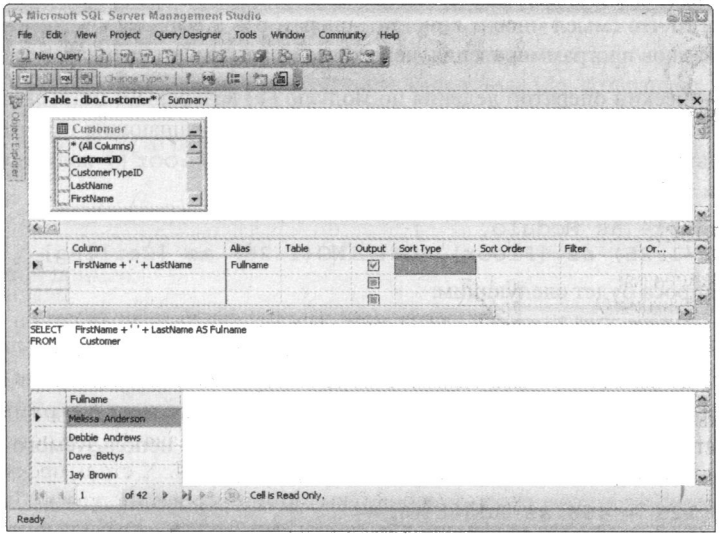


Рис. 8.1. Создание выражения и назначение псевдонима в конструкторе запросов утилиты Management Studio

Таблица 8.1. Создание выражений

Компоненты выражений	Примеры
Числовые константы	`1, 2, 3`
Строковые литералы	`'Фамилия','Жизнь прекрасна'`
Даты	`'6/1/80','Jan 6,1980','19800106'`
Математические операторы (в порядке приоритетов)	`*, /,%, ... +, -`
Строковый оператор (конкатенация)	`+`
Битовый оператор	И (&), ИЛИ (|), исключающее ИЛИ (^), НЕ (~)
Столбцы	`LastName, PramaryKeyID`

Выражения `CASE`	`CASE Column1`
	`  WHEN 1 THEN 'on'`
	`  ELSE 'OFF'`
	`END AS Status`

Подзапросы	`(SELECT 3)`
Пользовательские переменные	`@MyVariable`
Глобальные переменные	`error`
Скалярные функции	`GetDate(), SysUser()`
Пользовательские функции	`Dbo.myUDF()`

Дополнительная информация

Подзапросы рассмотрены в главе 10, переменные — в главе 18, а пользовательские функции детально описаны в главе 22.

Операторы

Несмотря на то что смысл многих констант, операторов и выражений очевиден и не отличается от других языков программирования, некоторые из них имеют специальное толкование.

- Математический оператор деления по модулю (`%`) возвращает остаток от деления. Математические функции `floor()` и `ceiling()`. возвращающие округленное большее или меньшее целое число, связаны с ним. Функция `floor()` является эквивалентом функции `int()` языка Basic:

```
SELECT 15%4 as Modulo,
  FLOOR(1.25) as [Floor], CEILING(1.25) as [Ceiling]
```

Результат запроса будет следующим:

```
Modulo      Floor Ceiling
----------- ----- -------
3           1     2
```

- Оператор + используется как в математических выражениях, так и для конкатенации строк. Этот оператор отличается от символа MS-DOS, используемого для строковой конкатенации (`&`):

```
SELECT 123 + 456 as Addition,
'abc' + 'defg' as Concatenation
```

Результат будет следующим:

```
Addition    Concatenation
----------- -------------
579         abcdefg
```

Данные из столбцов таблиц и строковых литералов можно объединять оператором конкатенации для возвращения произвольных данных:

```
Select 'Product: ' + ProductName as [Product]
  From Product
```

Результат будет следующим:

```
Product
---------------------------------
Product: Basic Box Kite 21 inch
Product: Dragon Flight
Product: Sky Dancer
...
```

Бинарные операторы

Бинарные операторы используют для манипуляций двоичные числа. Их редко используют в транзакционных базах данных, однако они доказали свою незаменимость в некоторых операциях с метаданными. Например, одним из способов определения, какие из столбцов были обновлены *триггером* (программой, которая автоматически выполняется при вставке, обновлении или удалении данных; подробно она рассмотрена в главе 23), является использование функции `columns_updated()`, которая возвращает двоичное представление этих столбцов.

Затем триггер может протестировать возвращенное этой функцией значение с помощью бинарных операторов и отреагировать на обновления каждого из столбцов.

Булевы битовые операторы (and, or и not) являются основным строительным блоком цифровой электроники и двоичного программирования. В то время как в цифровой электронике булевы операторы работают с отдельными битами, бинарные операторы работают с каждым битом значений целочисленных типов (int, smallint, tinyint и bit).

Булев оператор AND

Булев оператор AND, представленный символом амперсанда (&), возвращает значение true (т.е. "истина"), если оба значения равны true. Если любой из операндов или оба имеют значение false (т.е. "ложь"), то результатом выражения будет также false. Например, результатом выражения

```
SELECT 1&1
```

будет 1.

А вот еще один пример:

```
SELECT 1&0
```

Теперь результатом будет 0.

Если операндами являются целые числа, то битовая операция выполняется над каждой парой соответствующих значений в двоичном представлении. Проиллюстрируем это на примере:

```
-- 3 = 011
-- 5 = 101
-- AND ---
-- 1 = 001
```

Таким образом, инструкция

```
SELECT 3&5
```

в результате даст единицу.

Булев оператор OR

Булев оператор OR, представленный символом вертикальной черты (|), возвращает значение true, если любой из операндов (или оба сразу) имеет значение true. Например, результатом выражения

```
SELECT 1|1
```

будет единица.

Следующая инструкция, комбинирующая ложное и истинное значения

```
SELECT 1|0,
```

также в результате даст единицу.

Если операндами являются целые числа, то битовая операция выполняется над каждой парой соответствующих значений в двоичном представлении. Проиллюстрируем это на примере:

```
-- 3 = 011
-- 5 = 101
-- OR ---
-- 7 = 111
```

Таким образом, инструкция

```
SELECT 3|5
```

в результате даст число 7.

Булев оператор Exclusive OR

Булев оператор Exclusive OR, представленный символом галочки (^), возвращает значение true, если любой из операндов имеет значение true, но не оба одновременно. Его использование аналогично применению оператора OR к двум парам операторов AND, к каждому операнду которых, в свою очередь, применен оператор NOT. Хотя это достаточно просто реализовать в цифровой электронике, в программе этот оператор использовать еще легче.

Результатом выражения SELECT 1^1 будет 0.

В то же время сочетание ложного и истинного значений

```
SELECT 0^1
```

даст в результате 1.

Бинарный оператор NOT

Последний бинарный оператор, представленный символом тильды (~), является двоичной функцией NOT. Обычно этот оператор применяется к одному из операндов для корректировки результата операций AND или OR. В данном случае все несколько иначе. Оператор NOT выполняет логическое обращение каждого бита в выражении, а результат зависит от длины самого выражения. Приведем пример:

```
DECLARE @A BIT

SET @A = 1
SELECT ~@A
```

Результатом будет нуль.

Бинарный оператор NOT не подходит для использования в булевых выражениях, таких как условие IF. Например, следующий код некорректен:

```
SELECT * FROM Product WHERE ~(1=1)
```

Следует заметить, что оператор NOT может служить дополнением к другим булевым операторам.

Оператор CASE

Команда CASE в SQL Server — гибкое и удобное средство создания динамических выражений. Если вы программист, то несомненно используете команду CASE в других языках программирования. Однако команда CASE в данном случае несколько отличается. Она используется не для программного переключения управления, а для логического определения значения выражения на основе заданного условия, подобно функции if() в других языках программирования.

Подобно всем другим выражениям, CASE не может автоматически задать имя столбца. Таким образом, как правило, каждое выражение CASE имеет псевдоним.


Проверено

Когда программист пишет текст процедуры, часть формулы может изменяться в зависимости от данных. При процедурном мышлении в данном случае лучше организовать цикл по строкам и использовать множество операторов if для выбора корректной формулы. В то же время использование выражения CASE для выполнения различных вычислений и операций в одном запросе позволяет SQL Server оптимизировать и значительно ускорить процесс.

Так как оператор CASE возвращает выражение, его можно использовать в любом месте любой инструкции SQL DML (SELECT, INSERT, UPDATE и DELETE), где может использо-

ваться обычное выражение (например, выражения для столбцов, а также предложения JOIN, WHERE, HAVING и ORDER BY).

Оператор CASE имеет две формы: простую и булеву. Они будут описаны в следующих разделах.

Простая форма оператора CASE

В простой форме оператора CASE вначале представлено само значение, после чего перечисляются все тестовые значения. Однако эта форма ограничена тем, что можно использовать только сравнения на предмет равенства. Данный оператор CASE последовательно проверяет все условия WHEN и возвращает значение THEN из первой строки, в которой выполнится условие WHEN.

В следующем примере, основанном на базе данных OBXKites, один тип (поле customertype) назначается по умолчанию для всех новых клиентов, при этом в столбец isdefault заносится значение true. Оператор CASE сравнивает значение в столбце isdefault со всеми возможными значениями бита, после чего возвращает символьную строку 'default type' или 'possible' в зависимости от текущего значения:

```
USE OBXKites
SELECT CustomerTypeName,
CASE [IsDefault]
  WHEN 1 THEN 'default type'
  WHEN 0 THEN 'possible'
ELSE '-'
End as AssignStatus
From CustomerType
```

Будет получен следующий результат:

```
CustomerTypeName    AssignStatus
-----------------   ------------
Preferred           possible
Wholesale           possible
Retail              default type
```

Оператор CASE завершается ключевым словом END и псевдонимом. В данном примере оператор CASE тестировал значения столбца isdefault, но формировал столбец AssignStatus в результирующем наборе данных инструкции SELECT.

Булева форма оператора CASE

Булева форма оператора CASE более гибкая, чем простая. В ней каждый оператор WHEN имеет свое условие. Таким образом, имеется возможность не только использовать условные операторы, отличные от равенства, но и ссылаться на разные столбцы.

```
SELECT
CASE
  WHEN 1<0 THEN 'Реальность призрачна.'
  WHEN GetDate() = '11/30/2005'
    THEN 'Дейвид получил водительские права.'
  WHEN 1>0 THEN 'Жизнь продолжается.'
END AS RealityCheck
```

Результат запроса будет получен, когда будет выполнена проверка шестнадцатилетия Дейвида:

```
RealityCheck
------------------------------
Дейвид получил водительские права.
```

Как и в простой форме оператора CASE, выполняется оператор THEN в первой строке, в котором условие WHEN было выполнено. В данном случае, даже если единица вдруг окажется меньше нуля (только представьте!), оператор CASE вернет строку 'Жизнь продолжается'. Когда Дейвиду исполнится 16 лет, оператор сообщит о получении им водительских прав. Если ни одно из перечисленных событий не произойдет, мы получим сообщение, что жизнь продолжается.

Приведенный в качестве примера код продемонстрировал, что булева форма оператора CASE более гибкая, чем простая. В примере были использованы разные операторы сравнения, при этом в операторе WHEH сравнивались разные данные.

Булева форма оператора CASE может содержать сложные условия, в том числе операторы or и and. В следующем примере для создания оператора CASE использован пакет (в том числе задействованы переменные T-SQL, которые мы рассмотрим в главе 18), а само выражение CASE содержит операторы and и between:

```
DECLARE @b INT, @q INT
SET @b = 2007
SET @q = 25
Select CASE
  WHEN @b = 2007 AND @q BETWEEN 10 AND 30 THEN 1
ELSE NULL
END AS Test
```

Результат выполнения этого пакета будет следующим:

```
Test
---------
1
```

Работа с пустыми значениями

Реляционная модель базы данных представляет отсутствие данных с помощью специального значения NULL. В переводе на обычный язык его можно перевести так: "Значение не известно". На практике такие ситуации возникают, когда данные еще не введены полностью или когда данный столбец не применим к конкретной строке. Фактически NULL представляет собой *неопределенное* или *пустое* значение.

Поскольку значение NULL не известно, то и результат любой операции, включающей в себя NULL, не может быть известен. Если величина банковского счета не известна, а он включен в общий список состояния, то и общая величина состояния не известна. Та же концепция справедлива и в SQL, что и демонстрирует следующий код. Фил Шенн, известный разработчик баз данных, как-то сказал: "Пустое значение убивает жизнь любого другого".

```
SELECT 1 + NULL
```

Результатом будет NULL.

В связи с тем что пустые значения оказывают такое сокрушительное воздействие на выражения, некоторые разработчики исключают возможность их появления. Они проектируют свои базы данных так, чтобы значения NULL в столбцах не допускались, при этом заменяя их какими-либо предопределенными значениями (пустыми строками, нулями или строкой "n/a"). Другие разработчики считают, что замена пустых значений нулями или другими заменителями только для упрощения программирования неправомочна. Лично я принадлежу ко второму лагерю. Пустые значения в базах данных имеют свою ценность, так как информиру-

ют о состоянии данных. Таким образом, появляется возможность написать программу, которая проверяет базу данных на наличие пустых значений и обрабатывает найденные погрешности соответствующим образом.

Проверка на пустые значения

Так как значение NULL не известно, одно значение NULL не может быть равно другому значению NULL. Возвращаясь к примеру с банковскими счетами, предположим, что величина счета 123 не известна и величина счета 234 также не известна. Так каким же образом можно утверждать, что состояния этих счетов равны? Поскольку оператор равенства (=) не применим к пустым значениям, в языке SQL введен специальный оператор IS, который используется для тестирования на равенство специальным значениям. Например:

```
WHERE выражение IS NULL
```

Условие IS NULL используется для тестирования на пустые значения.

```
IF NULL = NULL
   SELECT '='
ELSE
   SELECT '!='
```

Результатом выполнения этого кода будет '!='.

В той же ситуации оператор IS ведет себя по-другому:

```
IF NULL IS NULL
  SELECT 'IS'
ELSE
  SELECT 'IS NOT'
```

Результат данного выражения — 'IS'.

Оператор сравнения IS может использоваться в предложении WHERE инструкции SELECT для отбора строк, содержащих пустые значения. Большинство клиентов компании Cape Hatteras Adventures не имеют в базе данных псевдонимов. Следующий запрос вернет только тех клиентов, у которых в столбце Nickname (псевдоним) содержится пустое значение:

```
USE CHA2
SELECT FirstName, LastName, Nickname
  FROM dbo.Customer
  WHERE Nickname IS NULL
  ORDER BY LastName, FirstName
```

Результат будет следующим:

```
FirstName    LastName       Nickname
------------ -------------- ----------------
Debbie       Andrews        NULL
Dave         Bettys         NULL
Jay          Brown          NULL
Lauren       Davis          NULL
...
```

Оператор IS можно комбинировать с оператором NOT для тестирования на наличие значения. Например, для отбора только тех клиентов, у которых есть псевдонимы, можно использовать условие Nickname IS NOT NULL:

```
SELECT FirstName, LastName, Nickname
  FROM dbo.Customer
  WHERE Nickname IS NOT NULL
  ORDER BY LastName, FirstName
```

Результат будет следующим:

```
FirstName    LastName       Nickname
------------ -------------- ----------------
Joe          Adams          Slim
Melissa      Anderson       Missy
Frank        Goldberg       Frankie
Raymond      Johnson        Ray
...
```

Обработка пустых значений

Когда данные отбираются для отчетов, конечных пользователей или приложений, пустые значения редко приветствуются. Часто пустое значение нужно преобразовать в некоторое допустимое, чтобы данные можно было понять или чтобы выражение имело результат.

Пустые значения требуют специальной обработки при использовании в выражениях, и язык SQL содержит ряд функций, специально предназначенных для работы с пустыми значениями. Функции `Isnull()` и `coalesce()` преобразуют пустые значения в пригодные для использования, а функция `nullif()` создает пустое значение, если выполняется определенное условие.

SQL Server при работе с булевыми выражениями использует логику из трех состояний. Так, сравнение значения `true` с `NULL` дает `NULL`.

Использование функции `IsNull()`

Наиболее часто используемой функцией, предназначенной для работы с пустыми значениями, является `IsNull()`, которая на самом деле отличается от условия `IS NULL`. Эта функция в качестве аргумента принимает одно выражение или столбец, а также подстановочное значение. Если первый аргумент является допустимым значением (т.е. не пустым), эта функция возвращает его. Однако если первый аргумент представляет собой пустое значение, то возвращается значение второго аргумента. Общий синтаксис функции следующий:

```
IsNull(исходное_выражение, замещающее_значение)
```

Функция `Isnull()` равносильна следующему оператору `CASE`:

```
CASE
  WHEN source_expression IS NULL THEN replacement_value
  ELSE source_expression
END AS ISNULL
```

Следующий пример построен на предыдущих запросах; он подставляет строку `'none'` вместо пустого значения там, где клиенты не имеют псевдонима:

```
SELECT FirstName, LastName, ISNULL(Nickname,'none')
FROM Customer
ORDER BY LastName, FirstName
```

Результат выполнения инструкции следующий:

```
FirstName    LastName       Nickname
------------ -------------- ----------------
Joe          Adams          Slim
Melissa      Anderson       Missy
Debbie       Andrews        none
Dave         Bettys         none
...
```

Если строка в столбце Nickname имеет определенное значение, то оно передается через функцию isnull() нетронутым. Однако если значение этого столбца пустое, то оно обрабатывается функцией Isnull() и преобразуется в строку 'none'.



Функции isnull() и nullif() специфичны для языка T-SQL и не входят в стандарт ANSI SQL.

Функция Coalesce()

Эта функция используется довольно редко, возможно, потому, что она мало кому известна. В то же время это довольно полезная функция. Она принимает список выражений или столбцов и возвращает первое значение, которое окажется не пустым. Ее общий синтаксис следующий:

```
Coalesce(выражение, выражение, ...)
```

Функция Coalesce() получила свое название из комбинации латинских слов *co* и *alesce*, что значит движение к общему концу или совместный рост. Ключевое слово в SQL произошло от альтернативного значения — "вырасти из комбинации различных элементов". В этом смысле функция coalesce() сводит вместе несколько различных значений с неизвестной степенью полезности и извлекает из них одно допустимое значение.

Функционально она является аналогом следующего оператора CASE:

```
CASE
  WHEN выражение1 IS NOT NULL THEN выражение1
  WHEN выражение2 IS NOT NULL THEN выражение2
  WHEN выражение3 IS NOT NULL THEN выражение3
   . . .
END AS COLEASCE
```

В следующем примере продемонстрирована функция coalesce(), возвращающая первое непустое значение (в данном случае это 1+2):

```
SELECT Coalesce(NULL, 1+NULL, 1+2, 'abc')
```

Результатом будет число 3.

Функция coalesce() идеально подходит для слияния разрозненных данных. Например, если в таблице в разных столбцах содержатся части одного целого, то эта функция поможет собрать их воедино. В одном проекте, над которым я работал, клиент хотел собрать названия и адреса контактов из нескольких баз данных и приложений в одну таблицу. Контактные лица и названия компаний содержались в корректных столбцах, однако адреса были разбросаны по столбцам Address1, Address2 и Address3. Некоторые строки имели вторую часть адреса в столбце Address2. Если в каком-нибудь из адресных столбцов содержался адрес, то в столбце SalesNotes действительно находилось примечание. Однако во многих случаях сам адрес находился в столбце SalesNotes. Следующий код помог собрать адреса из всего этого месива:

```
SELECT Coalesce(
Address1 + str(13)+str(10) + Adress2,
Address1,
Address2,
Address3,
SalesNote) AS NewAddress
FROM TempSalesContacts
```

В каждой строке таблицы TempSalesContact функция coalesce() выполняет поиск в перечисленных столбцах и возвращает первое непустое значение. Первое выражение возвра-

щает значение только в том случае, если и Address1, и Address2 содержат непустые значения, поскольку конкатенация с пустым значением дает в результате также пустое значение. Таким образом, если существует двустрочный адрес, он будет возвращен; в противном случае возвращает адрес из одного из столбцов, Address1, Address2 или Address3. Если ни одно из условий не выполнено, будет возвращен адрес из столбца SalesNotes. Естественно, результат, полученный из такой сложной таблицы, нужно проверить вручную.

Вы можете не использовать функцию coalesce ежедневно, однако ее полезно иметь в арсенале своих инструментов.

Функция Nullif()

Иногда пустое значение нужно создать на месте заменяющего его суррогатного. Если база данных заполнена значениями n/a, – или пустыми строками там, где должны находиться пустые значения, вы можете воспользоваться функцией nullif() и расчистить базу данных.

Функция nullif() принимает два аргумента. Если они равны, то возвращается пустое значение, в противном случае возвращается первый параметр. Функционально nullif() является аналогом следующего оператора CASE:

```
CASE
WHEN Expression1 = Expression2 THEN NULL
ELSE Expression1
END AS NULLIF
```

Следующий фрагмент кода преобразует все пробелы в столбце Nickname в пустые значения. Первая инструкция, заносящая в одну из строк пробелы, создана только в целях тестирования:

```
UPDATE Customer
SET Nickname = ''
WHERE LastName = 'Adams'
SELECT LastName, FirstName,
CASE NickName
WHEN '' THEN 'blank'
ELSE Nickname
END AS Nickname,
NullIf(Nickname, '') as NicknameNullIf
FROM dbo.Customer
WHERE LastName IN ('Adams', 'Anderson', 'Andrews')
ORDER BY LastName, FirstName
```

Результат будет следующим:

```
LastName    FirstName   Nickname    NicknameNullIf
----------- ----------- ----------- ------------
Adams       Joe         blank       NULL
Anderson    Melissa     Missy       Missy
Andrews     Debbie      NULL        NULL
```

Третий столбец использует операцию CASE для представления пробелов как строки "blank", после чего функция nullif() преобразует его в пустое значение четвертого столбца. Следующие строки демонстрируют другие ситуации. Псевдоним Мелиссы (Melissa) был оставлен функцией nullif() без изменений, а у Дебби (Debbie) пустое значение так и осталось пустым.

Нестандартное обращение с пустыми значениями

До сих пор мы обсуждали только обработку пустых значений, принятую в SQL Server по умолчанию. Однако SQL Server — в высшей мере гибкая СУБД, и режим работы с пустыми значениями можно также изменить.

Чисто логически конкатенация с пустым значением должна в результате дать пустое значение. Однако такой режим можно изменить. Параметр подключения `concat_null_yields_null` определяет результат конкатенации с пустым значением. Этот параметр подключения изначально определен одноименным умолчанием в базе данных. Изменение режима обращения с пустым значением трудно проверить, поскольку инструмент Query Analyzer также имеет принятый по умолчанию набор параметров подключения, применяемых к каждому новому подключению.

Следующий фрагмент кода устанавливает данный параметр базы данных, изменяя режим, принятый по умолчанию:

```
-- установка параметров базы данных
sp_dboption 'CHA2', CONCAT_NULL_YIELDS_NULL, 'false'
-- проверяем параметры
SELECT DATABASEPROPERTYEX('CHA2', 'IsNullConcat')
```

В результате будет получено значение `0`.

Устанавливаем параметр подключения:

```
SET CONCAT_NULL_YIELDS_NULL OFF
```

Теперь выполняем конкатенацию с пустым значением:

```
SELECT NULL + 'abc'
```

Результатом операции будет строка `'abc'`.

При обычных условиях стандарт ANSI SQL (равно как и SQL Server) предполагает, что сравнение с пустым значением дает в результате также пустое значение. Например, результатом выражения `(1>NULL)` будет значение `NULL`. Однако такой режим можно изменить с помощью оператора `ANSI NULLS OFF`. Наибольшим эффектом от такого изменения будет то, что пустые значения могут быть протестированы с помощью оператора равенства, а не только оператора `IS`.

Как и в предыдущем случае с конкатенацией, учитываются параметры подключения. В следующем фрагменте устанавливается режим работы базы данных по умолчанию, при этом отключается стандартный режим ANSI:

```
-- установка параметров базы данных
sp_dboption 'CHA2', ANSI_NULLS, 'false'
-- проверка установленных параметров
SELECT DATABASEPROPERTYEX('CHA2','IsAnsiNullsEnabled')
```

Результатом этого выражения будет `0`.

Устанавливаем режим работы с пустым значением:

```
SET ANSI_NULLS OFF
```

Теперь проверяем операцию сравнения двух пустых значений:

```
SELECT 'true' WHERE (NULL=NULL)
```

Результатом будет строка `'true'`.

Скалярные функции

Скалярная функция возвращает одно значение. Обычно скалярные функции используются в выражениях столбцов инструкции `SELECT`, в предложениях `WHERE` и коде T-SQL. SQL Server содержит десятки встроенных функций (рис. 8.2); в этом разделе будут описаны только те из них, которые я считаю наиболее полезными.

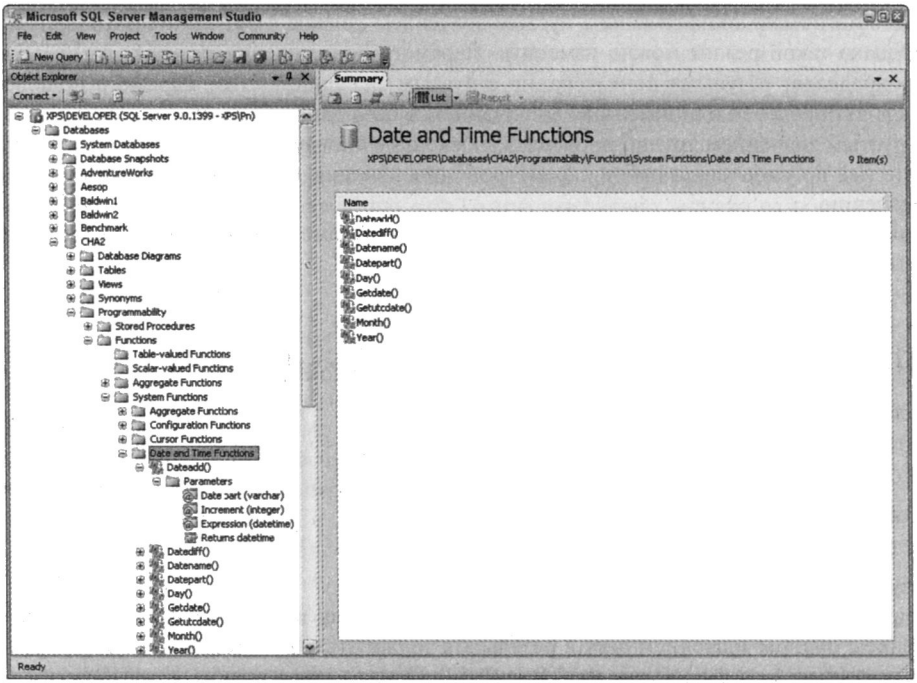


Рис. 8.2. Просмотреть все доступные функции SQL Server лучше всего в окне **Object Explorer**



Проверено

Производительность частично зависит от схемы данных и частично от структуры самого запроса. При планировании хранения данных нужно учитывать возможные условия `WHERE`. При этом следует максимально исключать использование функций в запросах. В то время как использование функций при формировании вычисляемых строк может быть и не столь проблематичным, их использование в предложении `WHERE` принуждает применять их к каждой из строк.



Дополнительная информация

В SQL Server 2005 можно создавать три типа пользовательских функций. Более подробно мы поговорим об этом в главе 22.

Информационные функции

В архитектуре "клиент/сервер" полезно знать, кем является конкретный клиент. В этом смысле очень полезными окажутся следующие четыре функции, особенно при сборе информации для аудита.

- `User_name()`. Возвращает имя текущего клиента, каким он представился базе данных. Когда пользователю открыт доступ к базе данных, его имя может отличаться от регистрационного имени входа на сервер.
- `Suser_sname()`. Возвращает регистрационное имя пользователя, под которым он вошел на SQL Server. Даже если тот был аутентифицирован как член одной из групп пользователей Windows, функция все равно возвращает имя его учетной записи Windows.
- `Host_name()`. Возвращает имя рабочей станции пользователя.
- `App_name()`. Возвращает имя приложения, подключенного к SQL Server.

Рассмотрим пример:

```
SELECT
USER_NAME() AS 'User',
SUSER_SNAME() AS 'Login',
HOST_NAME() AS 'Workstation',
APP_NAME() AS 'Application'
```

Результат будет следующим:

```
User    Login             Workstation  Application
------- ----------------- ------------ ------------------
dbo     NOLI\Paul CHA2    NOLI         SQL Query Analyzer
```

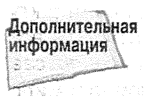
Функции работы с датой и временем

Базы данных часто работают с датой и временем, и SQL Server содержит для работы с ними ряд полезных функций. SQL Server хранит дату и время в одном типе данных.

Две функции T-SQL возвращают текущие дату и время.

- `GetDate()`. Возвращает текущие серверные дату и время, округленные до ближайших трех миллисекунд.
- `GetUTCDate()`. Возвращает текущие серверные дату и время, преобразованные во время Гринвичского меридиана. Возвращаемое значение округляется до ближайших трех миллисекунд. Эта функция особенно полезна компаниям, имеющим подразделения в разных временны*х* зонах.

Следующие четыре функции извлекают определенную часть значения с типом данных `datetime` и работают с ним.



Подробная информация о `datetime` и других типах данных содержится в главе 17.

- `DateName(`*`фрагмент`*`, `*`дата`*`)`. Возвращает правильное имя заданного фрагмента значения `datetime` (фрагменты для функций `datename()` и `datepart()` приведены в табл. 8.2).

```
SELECT DATENAME(Year, GetDate()) as Year
```

Результат будет следующим:

```
Year
--------
2001
```

В следующем примере для мистера Франка вводится дата рождения, а затем извлекаются правильные названия отдельных фрагментов этой даты с помощью функции `datename()`:

```
UPDATE Guide
  SET DateOfBirth = '9/4/58'
  WHERE lastName = 'Frank'
SELECT LastName,
    DATENAME(yy,DateOfBirth) AS [Year],
    DATENAME(mm,DateOfBirth) AS [Month],
    DATENAME(dd,DateOfBirth) AS [Day],
    DATENAME(weekday, DateOfBirth) AS BirthDay
  FROM dbo.Guide
  WHERE DateOfBirth IS NOT NULL
```

Результат выполнения будет следующим:

```
LastName  Year   Month       Day   BirthDay
--------- ------ ----------- ----- ---------------
Frank     1958   September   4     Thursday
```

Таблица 8.2. Фрагменты типа данных Datetime, используемые функциями даты и времени

Фрагмент	Аббревиатура	Фрагмент	Аббревиатура	Фрагмент	Аббревиатура
Год	yy, yyyy	День	dd, d	Минута	mi, n
Квартал	qq, q	Неделя	wk, ww	Секунда	ss, s
Месяц	mm, m	День недели	dw	Миллисекунда	ms
День года	dy, d	Час	hh		

- DatePart(*фрагмент, дата*). Возвращает выбранный фрагмент значения типа datetime. В следующих фрагментах извлекаются день года и день недели в виде целых чисел:

  ```
  SELECT DATEPART(DayofYear, GetDate()) AS DayCount
  ```

  Результат будет следующим:

  ```
  DayCount
  -----------
  321
  SELECT DATEPART(dw, GetDate()) AS DayWeek
  ```

  Результат будет следующим:

  ```
  DayWeek
  -----------
  7
  ```

  Простейший способ получить только дату, обрезав время, — это использовать несколько функций.

  ```
  Select Cast(Char(10), GetDate(), 101) as DateTime
  ```

- DateAdd(*фрагмент, величина, дата_начала*) и DateDiff(*фрагмент, величина, дата_начала*). Выполнение сложения и вычитания с данными типа datetime, что часто требуется в базах данных. Функции datediff() и dateadd() созданы специально для этих целей. Результат функции datediff() не выглядит как полноценный тип datetime, так как извлекается только фрагмент даты.

  В следующем запросе вычисляется количество дней, которые я женат на Мелиссе:

  ```
  SELECT
  DATEDIFF(yy,'1984/5/20', Getdate()) AS MarriedYears,
  DATEDIFF(dd,'1984/5/20', Getdate()) AS MarriedDays
  ```

  Результат будет следующим:

  ```
  MarriedYears MarriedDays
  ------------ -----------
  17           6390
  ```

  Следующий запрос добавляет 100 часов к текущей миллисекунде:

  ```
  SELECT DATEADD(hh,100, GETDATE()) AS [100HoursFromNow]
  ```

Результат будет следующим:

```
100HoursFromNow
-----------------------
2006-11-21 18:42:03.507
```

Следующий пример основан на базе данных Family и вычисляет возраст матери на момент рождения каждого из детей с помощью функции `datediff()`:

```
USE Family
SELECT Person.FirstName + ' ' + Person.LastName AS Mother,
DATEDIFF(yy, Person.DateOfBirth,
Child.DateOfBirth) AS Age,Child.FirstName
  FROM Person
    JOIN Person Child
      ON Person.PersonID = Child.MotherID
  ORDER By Age DESC
```

Функция `datediff` в этом запросе возвращает разницу в годах между днем рождения матери и детей. Так как функция находится в выражении для столбца, она вычисляется для каждой строки в следующем наборе данных:

```
Mother                          Age         FirstName
------------------------------- ----------- ---------------
Audrey Halloway                 33          Corwin
Kimberly Kidd                   31          Logan
Elizabeth Campbell              31          Alexia
Melanie Campbell                30          Adam
Grace Halloway                  30          James
...
```



В SQL Server не содержится тип данных только для даты, однако многим приложениям он нужен. Во избежание недоразумений обрезайте время и оставляйте только дату с помощью триггера. О триггерах речь пойдет в главе 23.

Строковые функции

Как и большинство современных языков программирования, T-SQL содержит множество функций работы со строками.

- `SubString(`*`строка, начальная_позиция, длина`*`)`. Возвращает фрагмент строки. Первым параметром является сама строка, вторым — номер символа, с которого вырезается фрагмент, третьим — длина вырезаемого фрагмента.

  Например, результатом инструкции `SELECT SUBSTRING('abcdefg', 3, 2)` будет подстрока `'cd'`.

- `Stuff(`*`строка, позиция_вставки, число_удаляемых_символов, вставляемая_строка`*`)`. Противоположная по характеру функции `substring()`, функция `stuff()` вставляет одну строку в другую; при этом в позиции вставки может быть удалено заданное количество символов исходной строки. Например, результатом инструкции `SELECT STUFF('abcdefg', 3, 2, '123')` будет строка `'ab123efg'`. В следующем примере использованы вложенные функции `stuff()` для форматирования номера социального страхования США:

  ```
  SELECT STUFF(STUFF('123456789', 4, 0, '-'), 7, 0, '-')
  ```

Результат будет следующим:

```
123-45-6789
```

- `CharIndex(`*символ_поиска*`,`*строка*`,`*начальная_позиция*`)`. Возвращает позицию заданного символа в строке. Например, инструкция

  ```
  SELECT CHARINDEX('c', 'abcdefg', 1)
  ```

  вернет результат 3.

  Пользовательская функция `TitleCase()`, описываемая далее в этом разделе, использует функцию `CharIndex()` для обнаружения пробелов, разделяющих слова.

- `PatIndex(%`*шаблон*`%,`*строка*`)`. Выполняет поиск по шаблону, который может содержать в строке символы макроподстановки. В следующем примере ищется первое вхождение в строку символа c или d:

  ```
  SELECT PATINDEX('%[cd]%', 'abdcdefg')
  ```

  Результатом данного запроса будет число 3.

- `Right(`*строка*`,`*число*`)` и `Left(`*строка*`,`*число*`)`. Возвращает крайнюю правую или левую часть строки. Например, результатом запроса

  ```
  SELECT Left('Nielsen',2) AS '[Left]',
  RIGHT('Nielsen',2) AS [Right]
  ```

  будет

  ```
  Left Right
  ----- ----
  Ni    en
  ```

- `Len(`*строка*`)`. Функция возвращает длину строки. Например, инструкция

  ```
  SELECT LEN('Supercalifragilisticexpialidocious') AS Len
  ```

  вернет результат

  ```
  Len
  -----------
  34
  ```

- `Rtrim(`*строка*`)` и `Ltrim(`*строка*`)`. Эти функции удаляют соответственно пробелы в начале и в конце строки. Несмотря на то что это можно не заметить на печатном листе, для иллюстрации работы функции мы удалим по три начальных и завершающих пробела:

  ```
  SELECT RTRIM('   middle earth   ') AS [RTrim],
  LTRIM('   middle earth   ') AS [LTrim]
  ```

  Результат будет следующим:

  ```
  RTrim            LTrim
  ---------------- ---------------
     middle earth middle earth
  ```

- `Upper(`*строка*`)` и `Lower(`*строка*`)`. Преобразует целую строку в верхний или нижний регистр. Символы нижнего регистра впервые использовались в IX веке для имитации рукописного письма. С появлением печатного станка в XV веке печатники вручную устанавливали регистры символов на каждой странице. Они хранили буквы в ящиках над печатной матрицей. Выше располагались ящики с большими символами. По этой причине и возникли термины *верхний регистр* и *нижний регистр*. А вот как использовать описываемые функции:

  ```
  Select UPPER('one TWO tHrEe') as [UpperCase],
  LOWER('one TWO tHrEe') as [LowerCase]
  ```

Результат будет получен следующий:

```
UpperCase      LowerCase
-------------- -------------
ONE TWO THREE one two three
```

- `Replace(`*строка*`,`*строка*`)`. Эта функция оперирует со строкой в целом и производит в ней замены. При использовании функции `replace()` в инструкции `update` можно легко исправить ошибки в данных, например удалить лишние символы табуляции или скорректировать шаблоны строк. В следующем примере удаляются апострофы в столбце `LastName` (фамилия) таблицы `Contact` базы данных OBXKites:

```
USE OBXKites
UPDATE Contact
SET LastName = 'Adam''s'
WHERE LastName = 'Adams'
SELECT LastName, REPLACE(LastName, '''', '')
FROM Contact
WHERE LastName LIKE '%''%'
UPDATE Contact
SET LastName = REPLACE(LastName, '''', '')
WHERE LastName LIKE '%''%'
```



При работе со строковыми константами довольно тяжело вставить кавычку в строку, не завершив ее или не вызвав синтаксическую ошибку. В SQL Server эта проблема решается с помощью двух последовательных символов кавычек, которые затем преобразуются в одну. Например, константа `'Паб О''Брайенс'` будет сохранена как `Паб О'Брайенс`.

- `pTitleCase(`*источник*`,`*приемник*`,`*замена*`)`. В языке T-SQL отсутствует функция, преобразующая слова в титульный формат (первая буква каждого слова всегда имеет верхний регистр, а все остальные — нижний). Следующая пользовательская функция способна справиться с этой задачей:

```
CREATE FUNCTION pTitleCase (
@StrIn NVARCHAR(1024))
RETURNS NVARCHAR(1024)
AS
BEGIN
  DECLARE
    @StrOut NVARCHAR(1024),
    @CurrentPosition INT,
    @NextSpace INT,
    @CurrentWord NVARCHAR(1024),
    @StrLen INT,
    @LastWord BIT
SET @NextSpace = 1
SET @CurrentPosition = 1
SET @StrOut = ''
SET @StrLen = LEN(@StrIn)
SET @LastWord = 0
WHILE @LastWord = 0
  BEGIN
    SET @NextSpace =
      CHARINDEX(' ',@StrIn, @CurrentPosition+ 1)
    IF @NextSpace = 0 -- no more spaces found
      BEGIN
        SET @NextSpace = @StrLen
        SET @LastWord = 1
```

```
      END
    SET @CurrentWord =
       UPPER(SUBSTRING(@StrIn, @CurrentPosition, 1))
    SET @CurrentWord = @CurrentWord +
      LOWER(SUBSTRING(@StrIn, @CurrentPosition+1,
      @NextSpace - @CurrentPosition))
    SET @StrOut = @StrOut +@CurrentWord
    SET @CurrentPosition = @NextSpace + 1
  END
RETURN @StrOut
END
```

Запуск любой пользовательской функции требует включения в ее вызов имени владельца:

```
Select dbo.pTitleCase('one TWO tHrEe') as [TitleCase]
```

Результат будет следующим:

```
TitleCase
------------------------
One Two Three
```



Функция `pTitleCase` не принимает во внимание возможность нестандартного сочетания строчных и прописных букв, как, например, в фамилиях *McDonald*, *VanCamp* или *de Jonge*. Вряд ли было бы целесообразным учитывать весь список таких исключений. Возможно, было бы лучшим решением хранить в обновляемом списке приставки, формирующие нестандартные фамилии (*Van*, *de*, *Mc* и т.п.). Обновленную функцию вы сможете найти на сайте `www.isnotnull.com`; если хотите, можете внести в нее дополнительные улучшения.



Текст функции `pTitleCase` можно загрузить с сайта книги по адресу `www.SQLServerBible.com`.

Функции системы Soundex

Система поиска фонетических соответствий Soundex была создана в Америке для задач переписи населения. Президент Рузвельт потребовал от Бюро архивов США создать метод каталогизации населения, который учел бы всевозможные вариации одинаковых фамилий. Система Soundex была создана Маргарет Оделл и Робертом Расселом, на что были получены патенты 1261167 (1918 г.) и 1435663 (1922 г.). Далее карточки переписи населения заполнялись с использованием этого метода.

Целью метода Soundex является такая сортировка, чтобы похоже звучащие имена находились рядом, что очень полезно при работе с информацией о контактах в приложениях базах данных. Например, если я звоню в банк и диктую свою фамилию *Нильсон*, то оператор вполне может услышать ее как *Нильсен* и занести в таком виде в форму базы данных. Однако если в базе данных используется система Soundex, то моя фамилия все равно появится в списке результатов поиска.

Более подробно о системе Soundex и ее истории вы можете узнать на следующих Web-сайтах:

`www.nara.gov/genealogy/coding.html`

`www.amberskyline.com/treasuremaps/uscensus.html`

`http://www.blueproof.com/soundex`

А вот как работает эта система. Первая буква имени хранится в виде символа, а следующие три фонетических звука хранятся по следующей схеме:

1 — B, F, P, V

2 — C, G, J, K, Q, S, X, Z

3 — D, T

4 — L

5 — M, N

6 — R

Двойные буквы A, E, I, O, U, H, W, Y и некоторые предлоги в одном коде Soundex во внимание не принимаются. Таким образом, моя фамилия Nilsen в этом коде принимает вид "N425" следующим образом.

1. Буква `N` сохраняется.
2. Буквы `i` и `e` игнорируются.
3. Буква `l` кодируется цифрой `4`.
4. Буква `s` кодируется цифрой `2`.
5. Буква `e` игнорируется.
6. Буква `n` сохраняется под кодом `5`.

Таким образом, избавившись от фонетически сходных звуков, фамилии Nielsen, Nelson и Neilson будут иметь один и тот же код — `N425`.

Позвольте привести еще ряд примеров кодировки фамилий.

- Brown — B650 (r — 6, n — 5).
- Jeffers — J162 (ff — 1, r — 6, s — 2)
- Letterman — L365 (tt — 3, r — 6, m — 5)
- Nelson — N425 (l — 4, s — 2, n — 5)
- Nicholson — N242 (c — 2, l — 4, s — 2)
- Nichols — N242 (c — 2, l — 4, s — 2)

Использование функции `Soundex()`

SQL Server содержит две функции, связанные с системой Soundex: `soundex()` и `difference()`. Функция `soundex(`*строка*`)` вычисляет код Soundex строки. Приведем пример:

```
SELECT SOUNDEX('Nielsen') AS Nielsen,
SOUNDEX('Nelson') AS NELSON,
SOUNDEX('Neilson') AS NEILSON
```

Результат этого запроса будет следующим:

```
Nielsen NELSON NEILSON
------- ------ -------
N425    N425   N425
```



Существует еще один, более совершенный метод кодировки Soundex. Кен Хендерсон в своей книге *The Guru's Guide to Transact SQL* (Addison–Wesley, 2000) предложил улучшенный алгоритм кодировки и соответствующую хранимую процедуру. Если вы собираетесь внедрять метод Soundex в производственной среде, я бы рекомендовал воспользоваться его версией. При желании вы можете поискать и другие алгоритмы на вышеперечисленных Web-сайтах и написать свою хранимую процедуру.

Существуют два способа внедрения поиска Soundex в базу данных. Простейший из них заключается во вставке функции `soundex()` в предложение `WHERE`, например:

```
USE CHA2
SELECT LastName, FirstName
FROM dbo.Customer
WHERE SOUNDEX('Nikolsen') = SOUNDEX(LastName)
```

Результат будет следующим:

```
LastName        FirstName
--------------- -------------------
Nicholson       Charles
Nickols         Bob
```

Несмотря на то что такая реализация имеет небольшое влияние на схему данных, она приводит к снижению производительности благодаря увеличению объема обрабатываемых данных, так как функция `soundex()` обрабатывает каждую строку в базе данных. Более производительная вариация предполагает изначальную сортировку по первой букве имени и сокращение объема поиска. Это позволяет применить функцию `soundex()` только к строкам, выбранным индексом.

```
SELECT LastName, FirstName
FROM dbo.Customer
WHERE SOUNDEX('Nikolsen') = SOUNDEX(LastName)
AND LastName LIKE 'N%'
```

Первый запрос обрабатывался 37,5 миллисекунды, в то время как усовершенствованный всего 6,5. Думаю, что на большем объеме данных разница будет более чувствительной.

Еще один метод реализации предполагает создание кластеризованного индекса по столбцу. Так как значение Soundex для каждой строки вычисляется во время записи, функцию `soundex()` не придется вызывать в инструкции `SELECT` при чтении каждой строки. Этот метод я бы порекомендовал для приложений, которые сильно зависят от метода Soundex при поиске контактов.

Именно этот метод продемонстрирован в базе данных OBXKites. Хранимая процедура `pContactAddNew` вычисляет код Soundex для каждого нового контакта и сохраняет результат в столбце `SoundexCode`. В этом случае поиск строк, основанный на сохраненном коде, выполняется экстремально быстро.

Для примера попробуем найти код Soundex для фамилии *Smith*.

```
USE OBXKites
SELECT SOUNDEX('Smith')
-------
S530
```

Узнав код, теперь можно выполнить поиск Soundex без вызова самой функции `soundex()` в инструкции `SELECT`.

```
SELECT LastName, FirstName, SoundexCode
FROM Contact
WHERE SoundexCode = 'S530'
```

Результат получится следующим:

```
LastName      FirstName       SoundexCode
------------- --------------- -----------
Smith         Ulisius         S530
Smith         Oscar           S530
```

Использование функции `Difference()`

Второй функцией работы с методом Soundex в SQL Server является `difference()`. Она возвращает разность Soundex между двумя строками в форме числа от 1 до 4, где четверка соответствует полному совпадению кода.

```
USE CHA2
SELECT LastName, DIFFERENCE ('Smith', LastName) AS NameSearch
FROM Customer
ORDER BY DIFFERENCE ('Smyth', LastName) DESC
```

Результат получится следующим:

```
LastName             NameSearch
-------------------- -----------
Smythe               4
Spade                3
Zeniod               3
Kennedy              3
Kennedy              3
Quinn                2
...
Kemper               1
Nicholson            0
...
```

Главное достоинство функции `difference()` состоит в том, что она расширяет диапазон поиска, не учитывая первых букв, а недостаток заключается в том, что она вычисляет значения двух параметров, что делает невозможным получить преимущество в производительности от предварительного вычисления и хранения в таблице значений Soundex.

Функции преобразования данных

В SQL Server преобразования одного типа данных в другой часто выполняются автоматически. Многие из них неявные или автоматические. Исключения из этого правила приведены в табл. 8.3.

Таблица 8.3. Исключения в преобразованиях типов данных

Из какого типа	В какой тип	Проблемы в преобразовании
`binary`, `varbinary`	`float`, `real`, `ntext`, `text`	Преобразование недопустимо
`char`, `varchar`, `nchar`, `nvarchar`	`binary`, `varbinary`, `money`, `smallmoney`, `timestamp`	Требуется явное преоброваниe
`nchar`, `nvarchar`	`image`	Преобразование недопустимо
`datetime`, `smalldatetime`	`decimal`, `numeric`, `float`, `real`, `bigint`, `int`, `smallint`, `tinyint`, `money`, `smallmoney`, `bit`, `timestamp`	Требуется явное преобразование
`datetime`, `smalldatetime`, `decimal`, `numeric`, `float`, `real`, `bigint`, `int`, `smallint`, `tinyint`, `money`, `smallmoney`, `bit`	`Uniqueidentifier`, `image`, `ntext`, `text`	Преобразование недопустимо

Из какого типа	В какой тип	Проблемы в преобразовании
`decimal`, `numeric`	`decimal`, `numeric`	Требуется явное указание точности во избежание потери данных
`float`, `real`	`timestamp`	Преобразование недопустимо
`money`, `smallmoney`	`char`, `varchar`, `nchar`, `nvarc7har`	Требуется явное преобразование
`timestamp`	`nchar`, `nvarchar`, `float`, `real`, `uniqueidentifier`, `ntext`, `text`, `sql_variant`	Преобразование недопустимо
`uniqueidentifier`	`datetime`, `smalldatetime`, `decimal`, `numeric`, `float`, `real`, `bigint`, `int`, `smallint`, `tinyint`, `money`, `smallmoney`, `bit`, `timestamp`, `image`, `ntext`	Преобразование недопустимо
`image`	`char`, `varchar`, `nchar`, `nvarchar`, `datetime`, `smalldatetime`, `decimal`, `numeric`, `float`, `real`, `bigint`, `int`, `smallint`, `tinyint`, `money`, `smallmoney`, `bit`, `ntext`, `sql_variant`	Преобразование недопустимо
`ntext`, `text`	`binary`, `varbinary`, `datetime`, `smalldatetime`, `decimal`, `numeric`, `float`, `real`, `bigint`, `int`, `smallint`, `tinyint`, `money`, `smallmoney`, `bit`, `timestamp`, `uniqueidentifier`, `image`, `sql_variant`	Преобразование недопустимо
`ntext`	`char`, `varchar`	Необходимо явное преобразование
`text`	`nchar`, `nvarchar`	Преобразование недопустимо
`sql_variant`	`timestamp`, `image`, `ntext`, `text`	Преобразование недопустимо

Для явных преобразований используют функции `cast()` и `convert()`.

- `Cast(`*исходные_данные* `AS` *тип данных*`)`. Стандарт ANSI SQL рекомендует явное преобразование одного типа данных в другой. Даже если такое преобразование может быть выполнено неявно сервером, использование функции `cast()` гарантирует получение нужного типа.

  Функция `cast()` программируется несколько отлично от других. Вместо разделения двух своих аргументов запятой используется ключевое слово `AS`, за которым следует требуемый тип данных, например:

```
SELECT CAST('Away' AS NVARCHAR(5)) AS 'Tom Hanks'
```

Результат запроса будет таким:

```
Tom Hanks
---------
Away
```

А вот еще один пример:

```
SELECT CAST(123 AS NVARCHAR(15)) AS Int2String
```

и его результат:

```
Int2String
---------------
123
```

- `Convert(`*тип_данных*`,`*выражение* `[,` *стиль*`])`. Эта функция возвращает значение, преобразованное в другой тип данных с произвольным форматированием. Эта функция не предусмотрена стандартом ANSI SQL. Первым ее аргументом является желаемый тип данных, применяемый к выражению.

  Аргумент *стиль* предполагает применение к результату некоторого стиля. Все допустимые стили приведены в табл. 8.4. Стиль обычно применяется при преобразовании из типа даты-времени в символьный и наоборот. Как правило, одно- или двухцифровой стиль предполагает двухцифровой год, а трехцифровой — четырехцифровой год. К примеру, стиль 1 подразумевает следующий формат данных: 01/01/03, в тоже время стиль 3 — 01/01/2003. Стили, отмеченные в таблице звездочкой, являются исключением из этого правила.

  SQL Server также предлагает числовые стили форматирования, однако это больше относится к интерфейсу пользователя, чем к самой базе данных.

Таблица 8.4. Стили дат функции `convert()`

Стиль	Описание	Формат
0/100*	Используется по умолчанию	ммм дд гггг чч:миАМ (или PM)
1/101	США	мм/дд/гг
2/102	ANSI	гг.мм.дд
3/103	Великобритания, Франция	дд/мм/гг
4/104	Германия	дд.мм.гг
5/105	Италия	дд-мм-гг
6/106	-	дд ммм гг
7/107	-	ммм дд,гг
8/108	-	чч:мм:сс
9/109*	По умолчанию + миллисекунды	ммм дд гггг чч:мм:сс:ммммАМ (или PM)
10/110	США	мм-дд-гг
11/111	Япония	гг/мм/дд
12/112	ISO	ггммдд
13/113	Европейский по умолчанию + миллисекунды	дд ммм гггг чч:мм:сс:ммм (24 часа)
14/114	-	чч:мм:сс:ммм (24 часа)

Стиль	Описание	Формат
20/120*	ODBC канонический	гггг-мм-дд чч:мм:сс (24часа)
20/121*	ODBC канонический + миллисекунды	гггг-мм-дд чч:мм:сс:ммм (24часа)
126	ISO8601 для XML	гггг-мм-дд чч:мм:сс (без пробелов)
130	Кувейт	дд ммм гггг чч:мм:сс:ммммAM (или PM)
131	Кувейт	дд/мм/гг чч:мм:сс:ммммAM (или PM)



В чистой среде "клиент/сервер" данные передаются клиенту без форматирования — этим занимается клиентское приложение в зависимости от потребностей конечного пользователя. Неформатированные данные более независимы, чем форматированные, так как могут использоваться разными приложениями.

В следующем примере проиллюстрировано использование функции `convert()`:

```
SELECT GETDATE() AS RawDate,
CONVERT (NVARCHAR(25), GETDATE(), 100) AS Date100,
CONVERT (NVARCHAR(25), GETDATE(), 1) AS Date1
```

Результат операции следующий:

```
RawDate                    Date100                 Date1
-------------------------- ----------------------- ----------
2001-11-17 10:27:27.413    Nov 17 2001 10:27AM     11/17/01
```

Две дополнительные функции преобразуют данные между текстовым и числовым типами.

- `Str(`*`число, длина, десятичных_знаков`*`)`. Преобразует число в строку.

  ```
  SELECT STR(123,5,2) AS [STR]
  ```

  Результат будет следующим:

  ```
  Str
  -----
  123.0
  ```

Информация о среде сервера

Системные функции возвращают информацию текущей среде сервера. В этом разделе мы рассмотрим наиболее популярные из них.

- `Db_name()`. Возвращает имя текущей базы данных, что и продемонстрировано в следующем примере:

  ```
  SELECT GETDATE() AS 'Date',
  DB_NAME() AS 'Database'
  ```

  Результат выполнения инструкции следующий:

  ```
  Date                       Database
  -------------------------- -------
  2001-11-15 18:38:50.250    CHA2
  ```

- `GetUTCDate()`. Возвращает текущее время часового пояса Гринвичского меридиана. Эта функция очень пригодится в приложениях, к которым обращаются из разных часовых поясов.

- `ServerProperty()`. Поставляет информацию о сервере, которая может понадобиться функции `serverproperty(`*свойство*`)`, в частности следующую:
  - `Collation`. Тип сопоставления.
  - `Edition`. Редакция сервера: Enterprise, Developer и т.п.
  - `InstanceName`. Имя экземпляра сервера или NULL, если используется экземпляр по умолчанию.
  - `ProductVersion`. Версия SQL Server.
  - `ProductLevel`. "RTM" — для исходной промышленной версии; "SP*n*" — для пакетов обновлений; "B*n*" — для бета-версии.
  - `ServerName`. Полное имя сервера и экземпляра.

  В следующем примере с помощью этой функции мы получим редакцию ядра и уровень продукта (результат получен на моем сервере):

```
SELECT
SERVERPROPERTY ('ServerName') AS ServerName,
SERVERPROPERTY ('Edition') AS Edition,
 SERVERPROPERTY ('ProductLevel') AS ProductLevel
```

  Будет получен следующий результат:

```
ServerName  Edition             ProductLevel
----------- ------------------- ------------
NOLI        Developer Edition   SP1
```

Резюме

В предыдущей главе мы ознакомились с инструкцией SELECT и порядком выполнения запросов. В этой главе мы расширили сферу действия этой инструкции, дополнив ее выражениями и вычислениями, которые могут быть вставлены в различные ее части, что значительно повышает гибкость работы с запросами. В последующих главах вы узнаете, как выражения могут получать данные из подзапросов и пользовательских функций.

В частности, в следующей главе описывается, как запрос получает данные из нескольких источников.

ГЛАВА

9

Объединение данных

В этой главе...

Основы реляционной алгебры

Создание масштабируемого кода с пакетными запросами

Использование внутренних, внешних, составных и тета-объединений

Слияние данных по вертикали

Во введении к этой книге я говорил, что хочу разделить с вами радость разработки приложений для SQL Server. В этой главе вы поймете, что я имел в виду. Заставьте данные группироваться нужным образом, получите ответ на свой вопрос с помощью оригинального запроса, замените сотни строк программы, написанной на традиционном языке программирования, одним пакетным запросом SQL — это действительно принесет вам радость, и обо всем этом мы поговорим в данной главе.

Реляционные базы данных по своей природе сегментируют данные в несколько узких, но достаточно длинных таблиц. Очень редко бывает, когда данные одной таблицы могут обеспечить пользователя смысловой информацией. Таким образом, объединение данных из множества таблиц является одной из основных задач программиста SQL. Теория, стоящая за объединением данных, называется *реляционной алгеброй* и была создана в 1970 году Эдгаром Коддом.

Реляционная алгебра содержит восемь реляционных операторов.

- **Ограничение**. Возвращает строки, удовлетворяющие заданному критерию.
- **Проекция**. Возвращает выбранные столбцы или вычисленные значения из набора данных.
- **Произведение**. Реляционное произведение возвращает все возможные комбинации данных из двух наборов.
- **Слияние**. Реляционное сложение и вычитание объединяют две таблицы по вертикали, располагая одну над другой и состыковывая столбцы.
- **Пересечение**. Возвращает строки, общие для двух таблиц.
- **Разность**. Возвращает строки, уникальные для некоторого набора данных.
- **Объединение**. Слияние двух таблиц по горизонтали на основе совпадения данных в определенных столбцах.

- **Деление**. Операция, обратная реляционному произведению. Возвращает строки одного множества данных, которые соответствуют всем строкам сопоставляемого набора данных.

В языке SQL существуют следующие методы реализации операций реляционной алгебры.

- **Подзапросы**. Аналогичны объединениям, но при этом более гибкие. Результат подзапроса используется вместо выражения, списка или набора данных во внешнем запросе.

На формальном языке реляционной алгебры:

- таблица, или набор данных, называется *ссылкой* или *сущностью*;
- строка называется *кортежем*;
- столбец называется *атрибутом*.

Именно эти термины мы будем использовать на протяжении всей главы.

Реляционная теория — это не панацея. За годы существования этой теории разработчики баз данных расширили ее и дополнили более строгими определениями, а теоретики более четко очертили круг задач для представления объектов и отношений реального мира с помощью структур данных. Однако оригинальная работа Эдгара Кодда все равно осталась фундаментом теории проектирования и реализации баз данных.

В настоящей главе будут описаны различные типы объединений и слияний. В следующей главе мы сфокусируем внимание на простых и коррелированных подзапросах, а также их использовании для решения реляционных задач.



Эта глава полностью основана на работах Э.Ф. Кодда и С. Дж. Дейта. Полный список рекомендованных первоисточников вы найдете на странице Books сайта `www.SQLServerBible.com`.

Использование объединений

В реляционной алгебре *объединением* называют произведение двух наборов данных с применением к результату следующего ограничения: он содержит только пересечение двух множеств. В целом объединение сливает два набора данных по горизонтали (этими наборами данных могут быть подзапросы, представления, общие табличные выражения и пользовательские функции) и получает новый результат, комбинируя строки одного набора данных с соответствующими строками другого (рис. 9.1). В этом разделе будут описаны разные типы объединений и их использование для отбора данных.

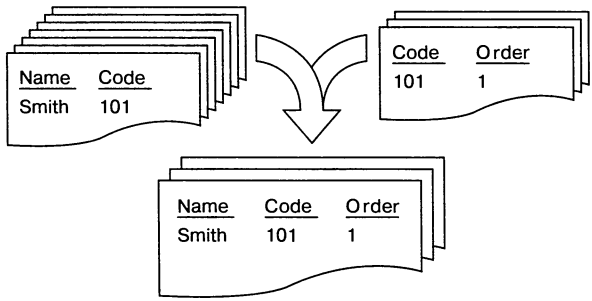


Рис. 9.1. Объединение сливает строки одного набора данных со строками другого, создавая новое множество строк, включающее столбцы обоих наборов данных. На этой диаграмме код 101 является общим для клиента `Smith` и заказа с номером `1` — он используется для объединения двух исходных строк в одну результирующую строку

Связывая информацию с помощью объединения, инструкция SELECT способна получить доступ к данным обеих таблиц и применить к ним выражения для столбцов, функции агрегирования, группировку и условия WHERE. Эти возможности являются сердцем и душой языка SQL и определяют его основную силу.

Я прошу прощения, если это напомнит вам домашнюю работу вашего ребенка по математике, но объединения основаны на идее пересечения множеств данных. Как показано на рис. 9.2, реляционное объединение имеет дело с двумя наборами данных с общим значением, и именно эти значения определяют, как пересекаются таблицы.

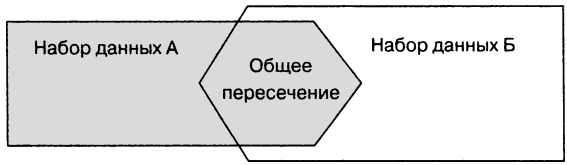
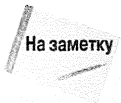


Рис. 9.2. Реляционные объединения основаны на наложении, или пересечении, двух множеств данных



Этот набор диаграмм является одним из типов диаграмм Венна. Более подробную информацию о диаграммах Венна вы можете получить на странице:

`www.combinatorics.org/Surveys/ds5/VennEJC.html`

Пересечение демонстрирует тот факт, что некоторый общий атрибут может связать некоторую строку одного набора данных со строкой другого набора. Эти общие значения обычно являются парой первичного и внешнего ключей, как в следующем примере из базы данных OBXKites.

- Поле `ContactID` связывает таблицы `Contact` и `[Order]`.
- Поле `OrderID` связывает таблицы `[Order]` и `OrderDetail`.
- Поле `ProductID` связывает таблицы `Product` и `OrderDetail`.

Язык SQL содержит множество типов объединений, определяющих, как именно отбираются строки из разных концов объединения. В табл. 9.1 перечислены эти типы объединений, а подробно они будут рассмотрены в следующих разделах.

Таблица 9.1. Типы объединений

Тип	Определение
Внутреннее объединение	Включает только соответствующие строки
Левое внешнее объединение	Включает все строки из левой таблицы, независимо от того, существуют ли в правой таблице соответствующие им, а также связанные строки правой таблицы
Правое внешнее объединение	Включает все строки из правой таблицы, независимо от того, существуют ли в левой таблице соответствующие им, а также соответствующие строки левой таблицы
Полное внешнее объединение	Включает все строки из обеих таблиц, независимо от того, существуют ли соответствия между ними
Тета-объединение	Соответствия между строками ищутся с использованием неравенств (<, >, <=, >= и <>)
Перекрестное объединение	Декартово произведение — соответствие между каждой строкой первого и второго источника данных, без каких бы то ни было условий и ограничений

Внутренние объединения

Внутренние объединения являются самыми распространенными; в работах Эдгара Кодда они назывались *естественными*. Внутренние объединения возвращают только те строки, которые соответствуют друг другу в двух наборах данных. Внутреннее объединение получило свое название по той причине, что оно извлекает данные из внутренней части пересечения двух множеств данных (рис. 9.3).

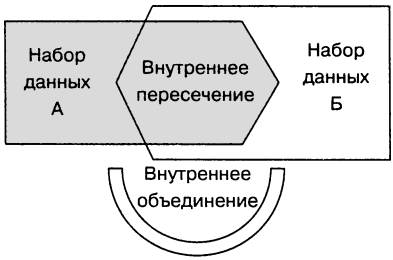


Рис. 9.3. Внутреннее объединение включает только те строки каждой своей стороны, которые содержатся в области пересечения двух множеств данных

Внутренние объединения легко создать в утилите Management Studio с помощью графического инструмента Query Designer (рис. 9.4). После того как две таблицы помещены на панель диаграммы с помощью функции Add Table или метода перетаскивания из списка таблиц, объединение создается автоматически, если эти таблицы содержат общие поля. Если вы не собираетесь использовать это объединение, то его придется удалить вручную.

В конструкторе запросов для каждого типа объединения используется свой символ (см. табл. 9.1). Символ, используемый для внутреннего объединения, имеет вид ромбика.

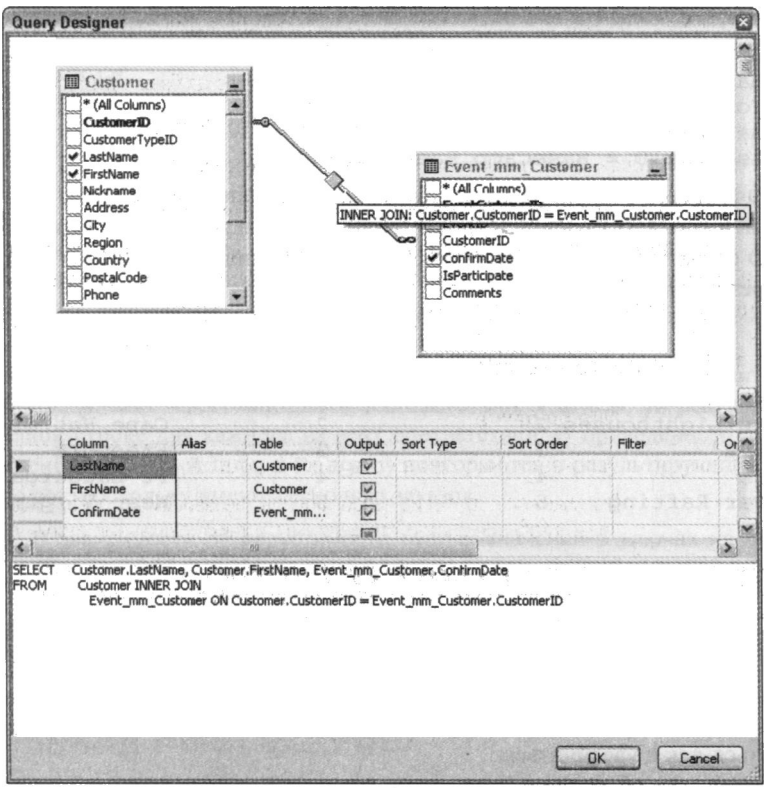


Рис. 9.4. Создание внутреннего объединения в конструкторе запросов Management Studio

Создание внутреннего объединения в коде SQL

В коде SQL объединения определяются в предложении FROM инструкции SELECT. Ключевое слово JOIN идентифицирует вторую таблицу, а ON — поля, на которых строится поиск соответствия между таблицами. Внутреннее объединение используется в SQL по умолчанию, поэтому ключевое слово INNER является необязательным.

```
SELECT *
  FROM Table1
  [INNER] JOIN Table2
    ON Table1.column = Table2.column
```



Примеры баз данных и кода, используемые в настоящей главе, можно загрузить с сайта www.SQLServerBible.com.

Так как объединения сводят данные из двух наборов, то желание SQL знать, как искать соответствия, имеет веские причины. SQL Server связывает строки на основе общих значений в полях объединения обеих таблиц. Обычно первичный ключ одной таблицы сравнивается с внешним ключом второй. Если строка из одной таблицы находит соответствие с некоторой строкой второй таблицы, то обе эти строки объединяются в одну, содержащую данные из обеих таблиц.

В следующем примере объединяются таблицы Tour (внешний ключ) и BaseCamp (первичный ключ) базы данных Cape Hatteras Adventures:

```
USE CHA2
SELECT Tour.Name, Tour.BaseCampID,
BaseCamp.BaseCampID, BaseCamp.Name
FROM dbo.Tour
JOIN dbo.BaseCamp
ON Tour.BaseCampID = BaseCamp.BaseCampID
```

Запрос начинается с таблицы Tour. Для каждой строки этой таблицы SQL Server будет пытаться найти соответствующие строки таблицы BaseCamp, сравнивая значения полей BasecampID обеих таблиц. Соответствующие строки таблиц Tour и BaseCamp сформируют следующий результат:

```
Tour.                Tour.          Basecamp.  Basecamp.
TourName             BaseCampID     BaseCampID BaseCampName
---------            -------------- ---------- -----------
Appalachian Trail    1              1          Ashville NC
Outer Banks Lighthouses 2           2          Cape Hatteras
Bahamas Dive         3              3          Freeport
Amazon Trek          4              4          Ft Lauderdale
Gauley River Rafting 5              5          West Virginia
```

Количество возвращаемых строк

В приведенном примере все строки таблиц Tour и BaseCamp имели соответствия, при этом из объединения не было исключено ни одной строки. Однако в реальной жизни такие случаи — редкость. В зависимости от количества соответствующих строк в каждом из источников данных и типа объединения количество строк в результирующем наборе данных может как уменьшиться, так и увеличиться.

Чтобы понять, как объединение влияет на количество возвращаемых строк, рассмотрим таблицы Contact и [Order] базы данных OBXKites. Исходное количество клиентов со-

ставляет 21, однако после сопоставления с заказами их число сократилось до 10. В следующем фрагменте кода мы сравним бок о бок два запроса и их результаты:

```
USE OBXKites
SELECT ContactCode, LastName   SELECT ContactCode, OrderNumber
  FROM dbo.Contact               FROM dbo.Contact
  ORDER BY ContactCode           JOIN dbo.[Order]
                                   ON [Order].ContactID
                                        = Contact.ContactID
                                 ORDER BY ContactCode
ContactCode LastName ContactCode OrderNumber
----------- -------- ----------- -----------
101         Smith    101         1
                     101         2
                     101         5
102         Adams    102         6
                     102         3
103         Reagan   103         4
                     103         7
104         Franklin 104         8
105         Dowdry   105         9
106         Grant    106         10
107         Smith
108         Hanks
109         James
110         Kennedy
111         Williams
112         Quincy
113         Laudry
114         Nelson
115         Miller
116         Jamison
117         Andrews
118         Boston
119         Harrison
120         Earl
121         Zing
```

Только контакты 101–106 имеют соответствующие им заказы, остальные были исключены из объединения.

Объединения могут также приводить к увеличению количества строк. Если строка с одной стороны объединения имеет множество соответствий с другой, то результат будет содержать по одной строке для каждого из этих соответствий. В приведенном выше примере некоторые клиенты (`Smith`, `Adams` и `Reagan`) повторяются в результирующем наборе данных несколько раз, поскольку имеют несколько заказов.



В зависимости от допустимости пустых значений в ключах на обоих концах, объединения имеют тенденцию терять строки, потому что одна или другая таблица воспроизводит некорректные данные. Когда данные извлекаются из множества таблиц, для запроса рекомендуется тщательно выбирать тип объединения (внутреннее, левое внешнее или правое внешнее), чтобы в результирующем наборе данных содержались все и каждая нужные строки.

Объединения ANSI SQL-89

Объединение — это не что иное, как отбор данных из двух таблиц, для которых выполняется условие равенства между значениями общих столбцов. Условия объединения в предло-

жении ON аналогичны условиям в предложении WHERE. На самом деле, пока ANSI SQL-92 не стандартизировал синтаксис выражения JOIN...ON, стандарт ANSI SQL-89, также называемый *объединением устаревшего стиля*, выполнял ту же задачу с помощью перечисления таблиц в предложении FROM и определения условий объединения в предложении WHERE. SQL Server поддерживает оба синтаксиса: ANSI SQL-92, и ANSI SQL-89.

Приведенное в предыдущем примере объединение таблиц Contact и [Order] в старом стандарте синтаксиса имело бы следующий вид:

```
SELECT Contact.ContactCode, [Order].OrderNumber
 FROM dbo.Contact, dbo.[Order]
  WHERE [Order].ContactID = Contact.ContactID
  ORDER BY ContactCode
```

Лично я предпочитаю использовать для объединения стандарт ANSI SQL-92 JOIN...ON, поскольку он позволяет полностью определить объединение в предложении FROM. Стандарт ANSI-89 разбивает определение объединения на два предложения, что, по моему мнению, в большей степени подвержено вероятности ошибок. В то же время ни один из этих стилей не имеет преимуществ в производительности, поскольку SQL Server создает идентичный план выполнения запроса для каждого из них.

Объединение множества таблиц

В некоторых продемонстрированных примерах инструкция SELECT не ограничивалась одним или двумя источниками данных. В общем случае инструкция SELECT может обращаться к 255 источникам данных. Представьте себе, сколько объединений может содержать такая инструкция! Так как SQL является декларативным языком, порядок источников данных в инструкции SELECT не имеет значения. Объединения могут быть сформированы в ней во множество цепочек, даже кольцевых (например, A объединяется с B; B — с C; C — с A). Именно здесь на первое место выходит последовательный стиль разработки.

Когда объединения между множеством таблиц комбинируются с ограничениями предложения WHERE (т.е. эти ограничения влияют на все объединяемые таблицы), порой происходят нетривиальные вещи. Ограничение, накладываемое на одну таблицу, подразумевает, что только строки, удовлетворяющие ему, участвуют в объединении.

Следующий запрос (показанный на рис. 9.5 и описанный в коде) отвечает на вопрос: "Кто покупал воздушных змеев?" В формировании ответа на этот вопрос участвуют следующие таблицы.

1. Таблица Contact отвечает на вопрос: "Кто?"
2. Таблица [Order] отбирает заказы.
3. Таблица OrderDetails отбирает товары.
4. Таблица Product выбирает категории товаров.
5. Таблица ProductCategory ищет категорию "kites".

Следующая инструкция SELECT начинается с фрагмента вопроса "кто" и определяет таблицы, участвующие в объединении, и накладываемые на них условия. Запрос, показанный графически на рис. 9.5 в окне конструктора запросов утилиты Management Studio, на языке SQL имеет следующий вид (обратите внимание на то, что предложение WHERE ограничивает строки таблицы ProductCategory, участвующие в объединении):

```
USE OBXKites
SELECT LastName, FirstName, ProductName
  FROM dbo.Contact
```

```
    JOIN dbo.[Order]
      ON Contact.ContactID = [Order].ContactID
   JOIN dbo.OrderDetail
     ON [Order].OrderID = OrderDetail.OrderID
   JOIN dbo.Product
     ON OrderDetail.ProductID = Product.ProductID
   JOIN dbo.ProductCategory
     ON Product.ProductCategoryID
          = ProductCategory.ProductCategoryID
  WHERE ProductCategoryName = 'Kite'
  ORDER BY LastName, FirstName
```

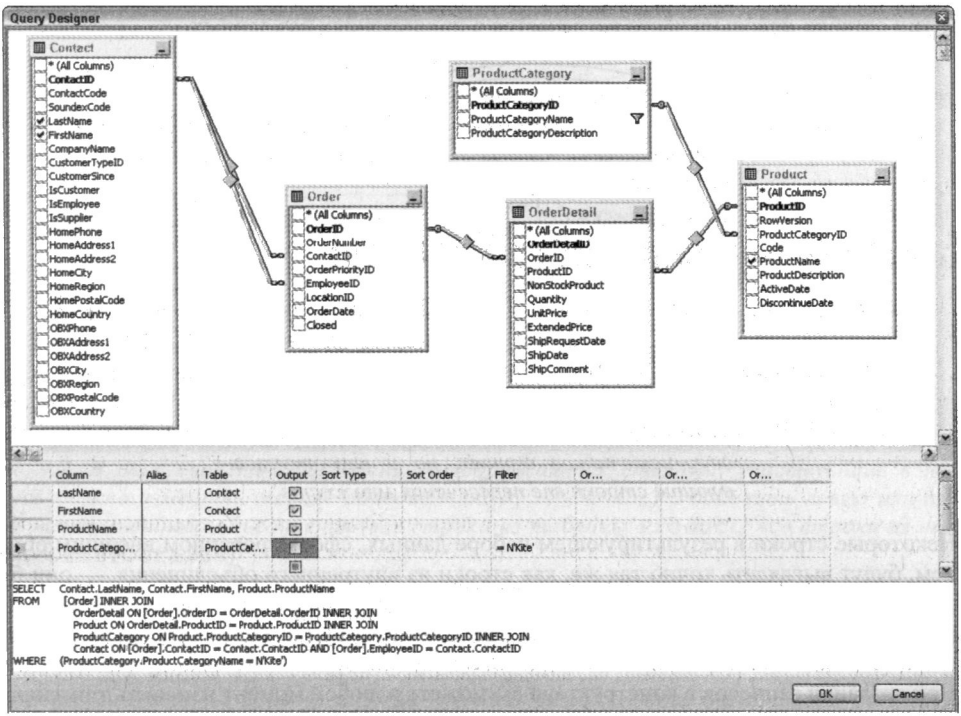


Рис. 9.5. Поиск ответа на вопрос "Кто покупал воздушных змеев?" в конструкторе запросов утилиты Management Studio

Результат выполнения запроса будет следующим:

```
LastName            FirstName             ProductName
------------------- --------------------- ---------------
Adams               Terri                 Dragon Flight
Dowdry              Quin                  Dragon Flight
...
Smith               Ulisius               Rocket Kite
```

По сравнению с запросом, сгенерированным конструктором запросов, запрос, приведенный в предыдущем коде, легче понять. Хотя конструктор в значительной мере облегчает создание запросов, сгенерированный им код труден для восприятия. Конструктор запросов иногда отдаляет условия ON от объединяемых таблиц, а создаваемое им форматирование просто ужасно.

Внешние объединения

В то время как внутренние объединения содержат только пересечение двух множеств данных, *внешние объединения* расширяют это множество за счет несоответствующих данных левого или правого набора данных (рис. 9.6).

Внешние объединения решают существенную проблему для множества запросов, включая в результат данные, независимо от их соответствия. В приведенном выше примере "клиент–заказ" эта проблема проиллюстрирована достаточно хорошо. Если нам требуется перечислить всех клиентов вместе с заказами, выполненными ими за последнее время, то внутреннее объединение отсеет заказчиков, совершивших покупки раньше начальной даты диапазона. Этот тип ошибки довольно часто встречается в приложениях баз данных.

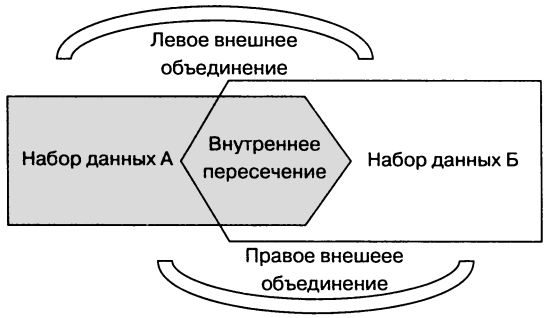


Рис. 9.6. Внешнее объединение включает не только соответствующие друг другу строки двух источников данных, но и несоответствующие строки вне пересечения множеств

Некоторые строки в результирующем наборе данных, сформированном внешним объединением, будут выглядеть точно так же, как строки из внутреннего объединения, — они будут содержать столбцы из каждого источника. В то же время некоторые строки, не нашедшие соответствия на противоположной стороне объединения, будут содержать только столбцы внешней таблицы — столбцы противоположной стороны будут содержать пустые значения.

При создании запросов в конструкторе вы можете в любой момент изменить тип запроса с внутреннего, принятого по умолчанию, на внешний. Это можно сделать с помощью контекстного меню, а также окна свойств объединения (рис. 9.7). Конструктор запросов отлично справляется с демонстрацией типа объединения, используя для этого специальные символы.

В коде SQL внешнее объединение объявляется с помощью ключевых слов `left outer` или `right outer` перед ключевым словом `join` (технически ключевое слово `outer` необязательное).

```
SELECT *
  FROM Table1
       LEFT|RIGHT [OUTER] JOIN Table2
     ON Table1.column = Table2.column
```



Несмотря на то что некоторые слова в SQL являются необязательными (такие, как INNER или OUTER) или могут заменяться сокращениями (например, `proc` для `procedure`), следование полному синтаксису повышает читаемость кода. В то же время многие разработчики опускают необязательный синтаксис.

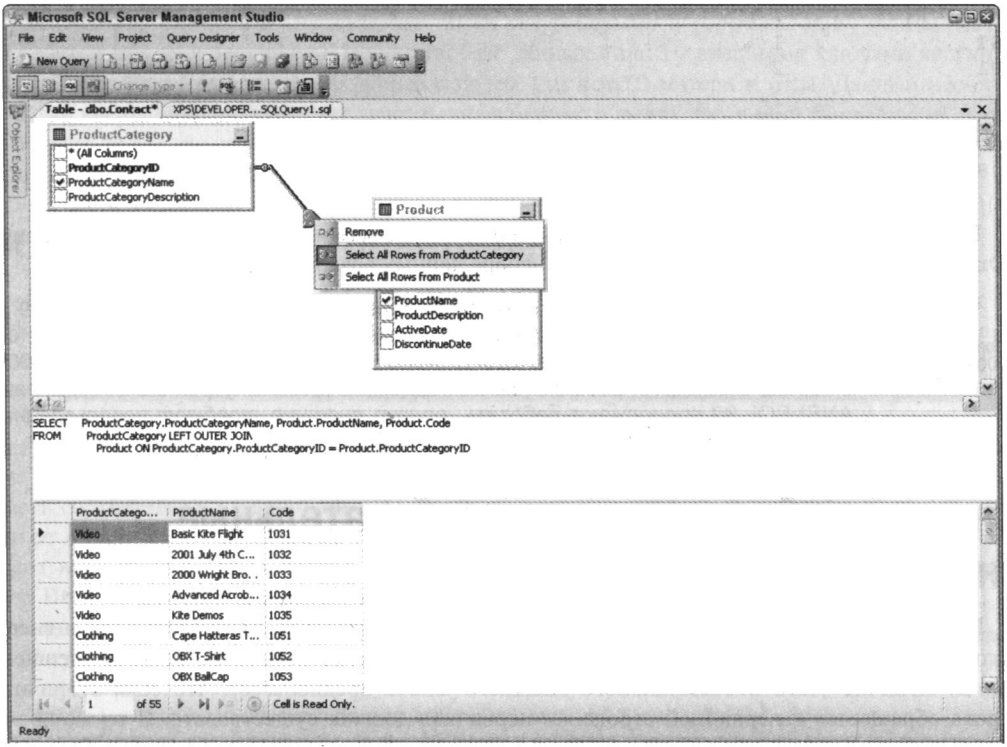


Рис. 9.7. В окне свойств объединения отображаются столбцы, участвующие в нем. Здесь можно изменить условие объединения (=, >, < и т.п.) и добавить левую или правую часть внешнего объединения (все строки из таблицы `Product`, все строки из таблицы `OrderDetails`)

Совсем не сложно отличить внешнее правое объединение от левого. В коде понятия "левое" и "правое" относятся к таблице, все записи которой будут включены в результат, независимо от найденных соответствий. Эта таблица (иногда называемая *управляющей*) обычно перечисляется в запросе первой; по этой причине левое внешнее объединение используется чаще правого. Я полагаю, что основную путаницу в вопрос отличия правого внешнего объединения от левого вносит графическое представление. Все дело в том, что слова "правое" и "левое" относятся к порядку перечисления таблиц в коде SQL, а их расположение на экране в графическом представлении только вносит путаницу.

Чтобы приведенный ранее в качестве примера запрос "клиент–контакт" возвращал всех заказчиков независимо от наличия у них заказа, требуется лишь заменить внутреннее объединение левым внешним объединением.

```
SELECT ContactCode, OrderNumber
  FROM dbo.Contact
    LEFT OUTER JOIN dbo.[Order]
      ON [Order].ContactID = Contact.ContactID
  ORDER BY ContactCode
```

Данное левое внешнее объединение включит все строки из таблицы `Contact` и соответствующие им строки из таблицы `[Order]`. Сокращенный результат этого запроса выглядит так:

```
Contact.          [Order].
ContactCode       OrderNumber
```

```
--------------- -----------
101             1
101             2
...
106             10
107             NULL
108             NULL
...
```

Поскольку контакты с номерами 107 и 108 не имеют соответствующих им строк в таблице `[Order]`, в ее столбцах результирующего набора данных возвращаются пустые значения.



Язык Transact-SQL расширил синтаксис внешнего объединения ANSI SQL-89 за счет добавления звездочки справа от знака равенства в условии предложения `WHERE`. Несмотря на то что этот синтаксис работает в версии SQL Server 2000, в версии SQL Server 2005 он уже не поддерживается. Внутренние объединения ANSI SQL-89 продолжают работать, однако внешние *требуют* использования синтаксиса ANSI SQL-92.

Внешние объединения и необязательные внешние ключи

Внешние объединения часто задействуют, если вторая таблица имеет ограничения внешнего ключа, влияющие на первую таблицу, и допускают пустые значения в столбце внешнего ключа. Наличие этого необязательного внешнего ключа предполагает, что если вторичная строка обращается к первичной строке, то первичная должна существовать. В то же время допустимо, чтобы вторичная строка воздерживалась от обращения к первичной таблице.

Еще одним примером использования необязательного внешнего ключа является столбец приоритетов или предупреждений. Однако в данном случае он должен указывать на допустимую строку в таблице приоритетов.

Учебная база данных OBXKites использует аналогичную схему приоритетов заказов, поэтому отчет по всем заказам с их приоритетами требует наличия внешнего объединения.

```
SELECT OrderNumber, OrderPriorityName
  FROM dbo. [Order]
    LEFT OUTER JOIN dbo.OrderPriority
    ON [Order].OrderPriorityID =
      OrderPriority.OrderPriorityID
```

Данное левое внешнее объединение извлекает все заказы и приоритеты, соответствующие им (если таковые имеются). Сценарий `OBXKites_Populate.sql` устанавливает двум заказам приоритет `Rush` (немедленно).

```
OrderNumber OrderPriorityName
----------- -----------------
1           Rush
2           NULL
3           Rush
4           NULL
5           NULL
6           NULL
7           NULL
8           NULL
9           NULL
10          NULL
```

Возвратные отношения (называемые также *рекурсивными* и *собственными*) используют и необязательные внешние ключи. В учетной базе данных `Family` внешними ключами являются поля `MotheID` и `FatherID`, связанные с полем `PersonID` матери и отца. Данный необязательный внешний ключ позволяет вводить информацию о людях, даже если данные об их родителях еще не введены в базу данных. В то же время, если поля `MotherID` и `FatherID` заполнены, они должны указывать на лицо, информация о котором занесена в базу данных.

Полные внешние объединения

Полное внешнее объединение возвращает все данные из обоих наборов независимо от их пересечения (рис. 9.8). Функционально такое объединение идентично слиянию с требованием уникальности правого и левого внешних объединений (о слияниях речь пойдет немного позже в этой главе).

В реальной жизни ссылочная целостность уменьшает потребность в полных внешних объединениях, так как любая строка из вторичной таблицы должна иметь соответствие в первичной таблице (в зависимости от обязательности внешнего ключа), так что чаще всего вполне достаточно левого внешнего объединения. Полные внешние объединения особенно полезны для очистки данных, не имеющих ограничений, поскольку отфильтровывают недопустимые значения.

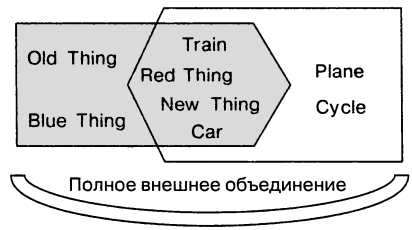


Рис. 9.8. Полное внешнее объединение возвращает все данные из обоих наборов, сопоставляя столбцы, где это возможно, и заполняя недостающие значения пустыми

Следующий пример иллюстрирует такую ситуацию и сравнивает полное внешнее объединение с правым и левым объединением. Таблица `Two` является подчиненной и имеет внешний ключ, ссылающийся на таблицу `One`. В ней не существует ограничений на внешний ключ, поэтому могут появляться некоторые несоответствия, которые легко выявить с помощью полного внешнего объединения:

```
CREATE TABLE dbo.One (
  OnePK INT,
  Thing1 VARCHAR(15)
  )
CREATE TABLE dbo.Two (
  TwoPK INT,
  OnePK INT,
  Thing2 VARCHAR(15)
  )
```

Данные в этом примере содержат строки, которые нарушают ссылочную целостность. Внешний ключ (`OnePK`) для строк `Plane` и `Cycle` в таблице `Two` не имеет соответствий в таблице `One`. Следующий пакет инструкций вставляет в таблицы восемь строк:

```
INSERT dbo.One(OnePK, Thing1)
  VALUES (1, 'Old Thing')
INSERT dbo.One(OnePK, Thing1)
  VALUES (2, 'New Thing')
INSERT dbo.One(OnePK, Thing1)
  VALUES (3, 'Red Thing')
INSERT dbo.One(OnePK, Thing1)
  VALUES (4, 'Blue Thing')

INSERT dbo.Two(TwoPK, OnePK, Thing2)
  VALUES(1,0, 'Plane')
INSERT dbo.Two(TwoPK, OnePK, Thing2)
```

```
  VALUES(2,2, 'Train')
INSERT dbo.Two(TwoPK, OnePK, Thing2)
  VALUES(3,3, 'Car')
INSERT dbo.Two(TwoPK, OnePK, Thing2)
  VALUES(4,NULL, 'Cycle')
```

Внутреннее объединение таблиц One и Two вернет только две строки, для которых было найдено соответствие:

```
SELECT Thing1, Thing2
  FROM dbo.One
   JOIN dbo.Two
     ON One.OnePK = Two.OnePK
```

Результат будет следующим:

```
Thing1          Thing2
--------------- ---------------
New Thing       Train
Red Thing       Car
```

Левое внешнее объединение расширяет этот список за счет включения строк из таблицы One, не имеющих соответствий в таблице Two:

```
SELECT Thing1, Thing2
  FROM dbo.One
    LEFT OUTER JOIN dbo.Two
  ON One.OnePK = Two.OnePK
```

Теперь из таблицы One возвращаются все строки, но в то же время отсутствуют две строки из таблицы Two:

```
Thing1          Thing2
--------------- ---------------
Old Thing       NULL
New Thing       Train
Red Thing       Car
Blue Thing      NULL
```

Полное внешнее объединение вернет все строки из обеих таблиц, независимо от того, найдено ли для них соответствие или нет:

```
SELECT Thing1, Thing2
  FROM dbo.One
    FULL OUTER JOIN dbo.Two
    ON One.OnePK = Two.OnePK
```

Теперь позиции Plane и Car перечислены наряду со всеми строками из таблицы One:

```
Thing1          Thing2
--------------- ---------------
NULL            Plane
New Thing       Train
Red Thing       Car
NULL            Cycle
Blue Thing      NULL
Old Thing       NULL
```

Как было показано в примере, полное внешнее объединение является прекрасным инструментом поиска всех данных, включая некорректные. Создавая запросы различий (о них мы поговорим позже в этой главе), построенные на внешних объединениях, можно выявлять и аннулировать некорректные данные.

Помещение во внешние объединения условий

Если речь идет о внутренних объединениях, условие производит тот же эффект, будучи помещенным в предложение JOIN или WHERE, однако это утверждение не относится к внешним объединениям. Когда условие находится в предложении JOIN, SQL Server включает в результат все строки из внешней таблицы, а затем использует условие для включения строк из вторичной таблицы. Когда ограничение помещено в предложение WHERE, выполняется объединение, а затем условие применяется к объединенным строкам. Следующие два примера демонстрируют эффект от помещения условия в разные предложения.

В первом запросе левое внешнее объединение включает все строки из таблицы One, а затем объединяет их с теми строками из таблицы Two, для которых значения поля OnePK в обеих таблицах равно и значение поля Thing1 равно New Thing. В результате мы увидим все строки из таблицы One, но меньше строк из таблицы Two:

```
SELECT Thing1, Thing2
  FROM dbo.One
    LEFT OUTER JOIN dbo.Two
      ON One.OnePK = Two.OnePK
      AND One.Thing1 = 'New Thing'
```

Результат будет следующим:

```
Thing1          Thing2
--------------- ---------------
Old Thing       NULL
New Thing       Train
Red Thing       NULL
Blue Thing      NULL
```

Второй запрос выполняет левое внешнее объединение, которое дает в результате четыре строки. После этого предложение WHERE применяет к этому результату ограничение, оставляя только одну строку — с товаром New Thing.

```
SELECT Thing1, Thing2
  FROM dbo.One
    LEFT OUTER JOIN dbo.Two
    ON One.OnePK = Two.OnePK
  WHERE One.Thing1 = 'New Thing'
```

Результат получится следующим:

```
Thing1          Thing2
--------------- ---------------
New Thing       Train
```

Аналогия объединений

Когда я обучал студентов созданию запросов, то для объяснения разных типов объединений часто использовал следующую аналогию. Представьте себе церковь пилигримов в XVII веке, разделенную по половому признаку. Все мужчины в ней сидели по одну сторону, а женщины — по другую. Теперь представьте себе каждую сторону церкви как таблицу базы данных, а различные комбинации людей, уходящих из церкви, — как разные типы объединений.

Если все супружеские пары встанут со своих мест, возьмут друг друга под руку и выйдут из церкви, то это будет внутреннее объединение между мужчинами и женщинами. Результирующий набор данных покидающих церковь будет содержать только соответствующие друг другу пары.

Если встанут все мужчины и каждый супруг возьмет руку жены, а все остальные выйдут поодиночке или группами, то это будет примером левого внешнего объединения. Строй покидающих церковь будет состоять как из супружеских пар, так и неженатых мужчин.

Аналогично, если встанут все женщины и выйдут из церкви со своими мужьями (разумеется, у кого они есть), это будет примером правого внешнего объединения. В этом случае все незамужние женщины покинут церковь поодиночке.

В случае полного внешнего объединения церковь покинут все, при этом супружеские пары выйдут под руку.

Собственные объединения

Собственными называют такие объединения, в которых таблица ссылается сама на себя. Этот тип отношений часто используют для извлечения данных из возвратных (также называемых *рекурсивными*) связей. Примером может служить база данных производства строительных материалов (выпускаемый материал ссылается на материалы своих компонентов) и база данных сотрудников предприятия (начальник ссылается на своих подчиненных). Представьте себе собственное объединение как таблицу, объединенную со своей временной копией.

Учебная база данных Family использует два собственных объединения: между ребенком и каждым из родителей (рис. 9.9). Матери и отцы — тоже люди и перечислены в той же таблице. Они ссылаются на собственных родителей и т.д. Учебная база данных наполнена пятью фиктивными поколениями, которые можно использовать для создания пробных запросов.

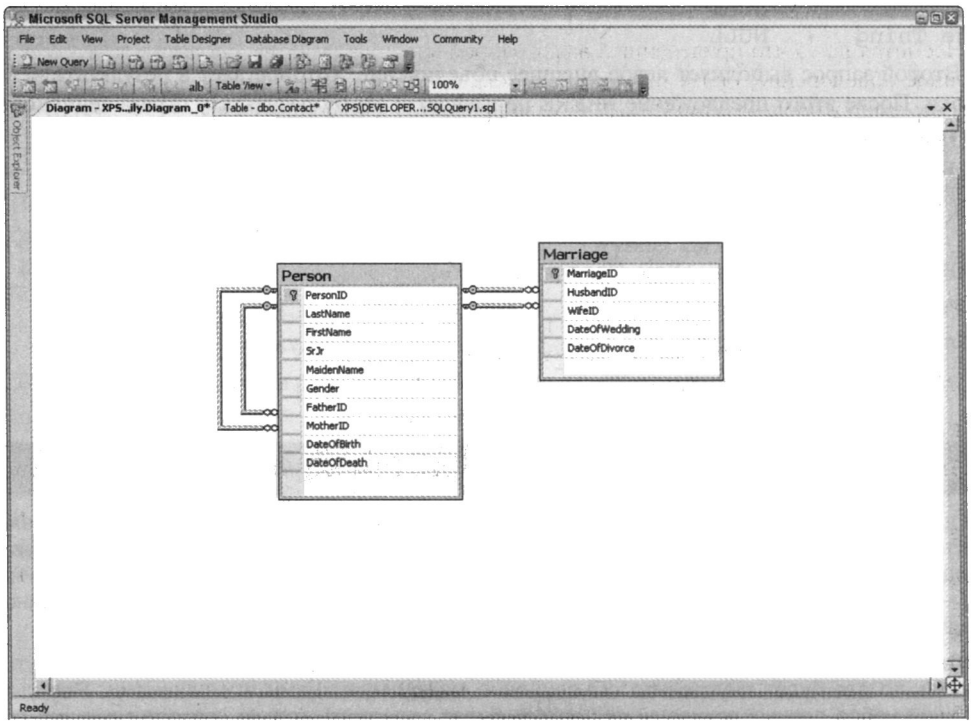


Рис. 9.9. Диаграмма базы данных Family содержит два унарных отношения (ребенка на родителей) слева и одно унарное отношение "многие ко многим" (мужа к жене) — справа

Ключ к созданию собственных объединений лежит в создании второй ссылки на таблицу в виде именованного диапазона или псевдонима. Как только таблица становится доступной в инструкции SELECT в двух ипостасях, функция собственного объединения начинает работать точно так же, как и для любого другого типа объединения. В следующем примере ссылка на таблицу dbo.Person выполняется с помощью именованного диапазона 'Mother'.

Следующий запрос поможет выявить детей Одри Халлоуэй (Audry Halloway):

```
USE Family
SELECT Person.PersonID, Person.FirstName,
    Person.MotherID, Mother.PersonID
  FROM dbo.Person
    JOIN dbo.Person Mother
      ON Person.MotherID = Mother.PersonID
  WHERE Mother.LastName = 'Halloway'
    AND Mother.FirstName = 'Audry'
```

В этом запросе таблица Person использована дважды. Первая ссылка без именованного диапазона связана со второй ссылкой; все это ограничено предложением WHERE с привязкой к имени Audry Halloway. Во внутреннем объединении будут участвовать только те записи, в которых поле MotherID ссылается на записи с именем Одри Халлоуэй. Идентификатором Одри является число 6, и запрос вернет следующих ее детей:

```
PersonID    FirstName       MotherID    PersonID
----------- --------------- ----------- -----------
8           Melanie         6           6
7           Corwin          6           6
9           Dara            6           6
10          James           6           6
```

Несмотря на то что приведенный выше запрос вполне адекватно продемонстрировал собственное объединение, более полезно было бы, если бы имя матери не было строго запрограммировано в предложении WHERE и было предоставлено больше информации, в частности о дате рождения каждого человека.

```
SELECT CONVERT(NVARCHAR(15),Person.DateofBirth,1) AS Date,
    Person.FirstName AS Name, Person.Gender AS G,
    ISNULL(F.FirstName + ' ' + F.LastName, '* unknown *')
      as Father,
    M.FirstName + ' ' + M.LastName as Mother
  FROM dbo.Person
    Left Outer JOIN dbo.Person F
      ON Person.FatherID = F.PersonID
    INNER JOIN dbo.Person M
      ON Person.MotherID = M.PersonID
  ORDER BY Person.DateOfBirth
```

Этот запрос создает три ссылки на таблицу Person: на ребенка, на отца и мать. В результате получается более информативный листинг:

```
Date     Name      G  Father             Mother
-------- -------- --- ------------------ ---------------
5/19/22  James    M   James Halloway     Kelly Halloway
8/05/28  Audry    F   Bryan Miller       Karen Miller
8/19/51  Melanie  F   James Halloway     Audry Halloway
8/30/53  James    M   James Halloway     Audry Halloway
2/12/58  Dara     F   James Halloway     Audry Halloway
3/13/61  Corwin   M   James Halloway     Audry Halloway
3/13/65  Cameron  M   Richard Campbell   Elizabeth Campbell
...
```

Перекрестные (неограниченные) объединения

Перекрестные объединения, также называемые *неограниченными*, являются реализацией реляционного произведения двух исходных таблиц. При отсутствии условия объединения результирующий набор данных будет содержать все возможные комбинации строк двух источников. Каждая строка исходного набора 1 будет сопоставляться с каждой строкой исходного набора 2. Например, если первый набор данных содержит пять строк, а второй — четыре строки, то результирующий набор данных будет содержать двадцать строк. Такой тип результирующего множества называют *декартовым произведением*.

Используя в качестве примера таблицы `One` и `Two`, перекрестное объединение двух таблиц, созданное в Management Studio при отсутствии условия объединения, будет выглядеть так, как показано на рис. 9.10.

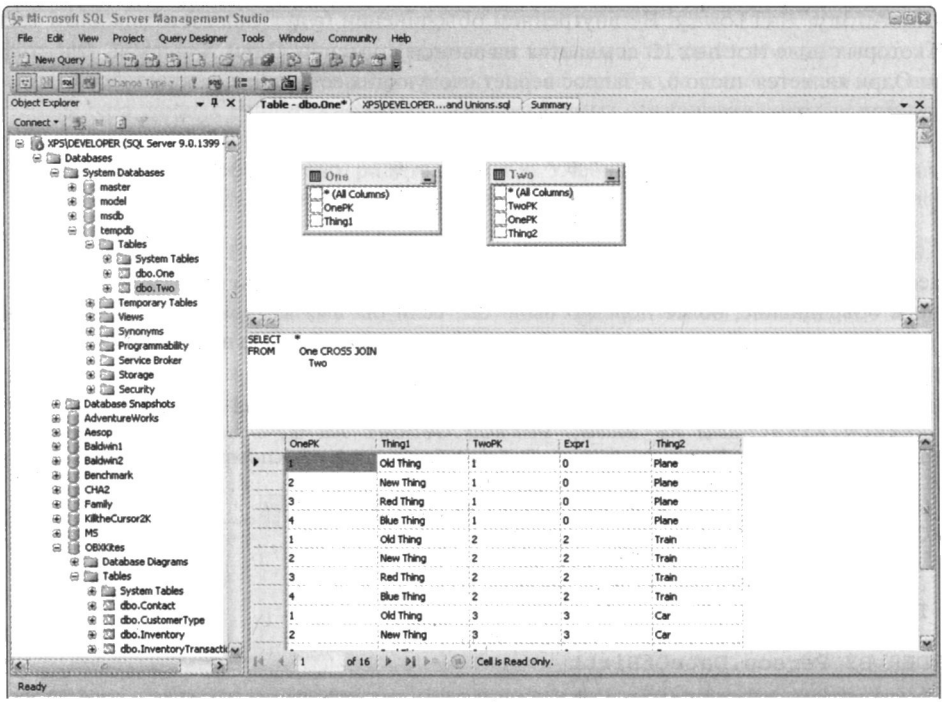


Рис. 9.10. Перекрестное объединение представляет собой две таблицы без условия объединения

В коде этот тип объединения определяется ключевыми словами `cross join` при отсутствии условия `ON`.

```
SELECT Thing1, Thing2
  FROM dbo.One
    CROSS JOIN dbo.Two
```

Результатом такого объединения без ограничений будут все строки таблицы `One`, связанные с каждой строкой таблицы `Two`.

```
Thing1          Thing2
--------------- ---------------
Old Thing       Plane
New Thing       Plane
Red Thing       Plane
Blue Thing      Plane
Old Thing       Train
New Thing       Train
Red Thing       Train
Blue Thing      Train
Old Thing       Car
New Thing       Car
Red Thing       Car
Blue Thing      Car
Old Thing       Cycle
New Thing       Cycle
Red Thing       Cycle
Blue Thing      Cycle
```

Большинство перекрестных объединений являются результатом того, что кто-то забыл определить отношение в графическом интерфейсе. В то же время эти объединения полезны для наполнения базы данных учебными данными или для создания пустых строк с целью заполнения некоторой процедурой.

Понимание того, как перекрестное объединение умножает данные, также полезно при изучении реляционного деления — операции, обратной произведению. Реляционное деление требует использования подзапросов (рис. 9.12); оно будет описано ниже в этой главе.

Экзотические объединения

Практически все объединения базируются на условии равенства первичного ключа главной таблицы и внешнего ключа подчиненной, именно поэтому внутреннее объединение иногда называют *объединением равенства*. Однако такая популярность объединений на основе равенства совершенно не отрицает ценности и других условий.

Условие объединения `ON` в действительности является ничем иным, как условием `WHERE`, ограничивающим произведение двух объединяемых наборов данных. Предложения `WHERE` могут быть достаточно гибкими и эффективными, однако то же можно сказать и об условиях объединения. Доказательством тому является возможность использования следующих технологий: *тета-объединений*, *объединений с множеством условий* и *неключевых объединений*.

Тета-объединения (Θ-объединения)

Тета-объединения (обозначаемые греческой буквой Θ) — это объединения, основанные на условиях неравенства. В реляционной теории все условные операторы (=, <, <=, >, >= и <>) называют Θ-операторами. Несмотря на то что с теоретической точки зрения условия равенства также используют Θ-оператор, в теории баз данных только объединения, использующие в своих условиях другие операторы, называют Θ-объединениями.

Θ-условие можно создать в конструкторе запросов Management Studio с помощью диалогового окна свойств объединения (см. рис. 9.7).

Θ-объединения часто комбинируют со множеством условных объединений, использующих неключевые столбцы. Во всех дальнейших примерах этой главы мы будем использовать Θ-объединения.

Объединения с множеством условий

Если объединения — это не что иное, как условие связывания двух наборов данных, то имеет смысл использовать в них множество условий. На практике объединения с множеством условий идут рука об руку с Θ-объединениями; без возможности использовать первые вторые практически утратили бы свое значение.

Условия объединения могут ссылаться на любую таблицу, перечисленную в предложении FROM, и это позволяет создавать интересные трехсторонние объединения. Приведем пример:

```
SELECT *
  From A
    JOIN B
      ON A.col = B.col
   JOIN C
     ON B.col = C.col
     AND A.col = C.col
```

Неключевые объединения

Объединения не ограничены только первичными и внешними ключами — в них можно сопоставлять строки двух наборов данных по любым столбцам, разумеется, если эти столбцы имеют совместимые типы данных.

Например, система складского распределения может использовать неключевое объединение для поиска товаров, поступление которых ожидается от поставщика, с целью информирования клиента о сроках доставки. Неключевое объединение между таблицами PurchaseOrder и OrderDetail с Θ-условием между полями PO.DateExpected и OD.DateRequired отфильтрует результат объединения только теми товарами, которые могут быть размещены в заказе клиента. В следующем примере продемонстрировано данное неключевое объединение (его нет в учебной базе данных):

```
SELECT OD.OrderID, OD.ProductID, PO.POID
FROM OrderDetail OD
  JOIN PurchaseOrder PO
    ON OD.ProductID = PO.ProductID
      AND OD.DateRequired > PO.DateExpected
```

При работе с внутренними объединениями неключевые условия можно поместить в предложение WHERE или JOIN. Так как эти условия сравнивают аналогичные значения двух связанных таблиц, я чаще всего помещаю эти условия в часть JOIN предложения FROM, а не в предложение WHERE. Основное различие между этими подходами кроется в том, как именно вы рассматриваете это условие: как элемент создания набора данных, с которым будет работать остальная часть инструкции SELECT, или как фильтр для задач, которые следуют за предложением FROM. В любом случае оптимизированный план запроса будет идентичным, поэтому вы вольны использовать любой метод, который вам покажется более читаемым и логичным. Следует еще раз отметить, что при создании внешних объединений помещение условий в предложения JOIN и WHERE дает разные результаты, что и было продемонстрировано в разделе, посвященном внешним объединениям.

Если посмотреть на учебную базу данных Family, то вы увидите, что ответ на вопрос "Кто является близнецами?" использует все три экзотических типа объединения. Условие объединения по полю MotherID, хотя и использует внешний ключ, является нестандартным, поскольку связано еще с одним внешним ключом. Условие DateOfBirth по определению является неключевым. Условие WHERE DateOfBirth is not null удаляет из запроса тех,

кто вошел в семью через замужество и, таким образом, не имеет зарегистрированных в базе данных родителей:

```
SELECT Person.FirstName + ' ' + Person.LastName,
    Twin.FirstName + ' ' + Twin.LastName as Twin,
    Person.DateOfBirth
  FROM dbo.Person
    JOIN dbo.Person Twin
      ON Person.PersonID <> Twin.PersonID
        AND Person.MotherID = Twin.MotherID
        AND Person.DateOfBirth = Twin.DateOfBirth
  WHERE Person.DateOfBirth IS NOT NULL
```

Далее описан тот же запрос, но на этот раз с экзотическим условием объединения, перемещенным в предложение WHERE. Неудивительно, что оптимизатор SQL Server для обоих запросов создаст идентичный план выполнения:

```
SELECT Person.FirstName + ' ' + Person.LastName AS Person,
    Twin.FirstName + ' ' + Twin.LastName as Twin,
    Person.DateOfBirth
  FROM dbo.Person
    JOIN dbo.Person Twin
      ON Person.MotherID = Twin.MotherID
        AND Person.DateOfBirth = Twin.DateOfBirth
  WHERE Person.DateOfBirth IS NOT NULL
    AND Person.PersonID != Twin.PersonID
```

Результат будет следующим:

```
Person          Twin            DateOfBirth
--------------- --------------- -------------------------
Abbie Halloway  Allie Halloway  1979-08-14 00:00:00.000
Allie Halloway  Abbie Halloway  1979-08-14 00:00:00.000
```

В учебной базе Northwind, поставляемой компанией Microsoft, неключевое объединение может быть создано путем сравнения значений столбцов `Region` таблиц `Customers`, `Shippers` и `Orders`.

Сложные сценарии запросов, приведенные в конце настоящей главы, также продемонстрируют экзотические объединения, часто используемые с подзапросами.

Разность множеств

Аналогичный тип запросов, не менее полезный для анализа корреляции между двумя наборами данных, называется *запросом на разность множеств*. Он ищет различия между двумя наборами данных, основываясь на условии объединения. В терминах реляционной алгебры он удаляет делитель из делимого, оставляя разность. Этот тип запросов является обратным внутреннему объединению. Неформально его называют *запросом поиска несоответствующих строк*.

Запросы на разность множеств идеально подходят для поиска неуместных или несоответствующих данных (например, строк, которые присутствуют в одном наборе данных и отсутствуют в другом) (рис. 9.11).

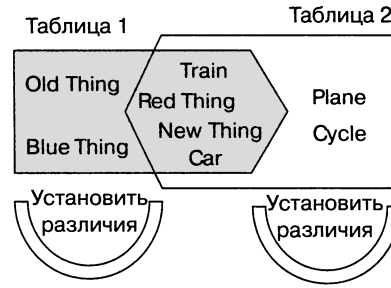
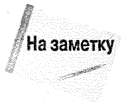


Рис. 9.11. Запрос на разность множеств ищет данные, находящиеся за пределами пересечения двух наборов



В стандарте ANSI SQL запросы на разность реализованы с помощью ключевого слова EXCEPT, которое SQL Server не поддерживает.

Запрос на разность множеств аналогичен слиянию разности за исключением того, что последний является операцией, основанной на строках таблиц с идентичным определением столбцов, в то время как первый имеет дело только со столбцами, участвующими в условии объединения. В этом смысле запрос на разность множеств является слиянием разности только тех столбцов, которые участвуют в объединении.

Используя наш пример с таблицами One и Two, мы можем создать запрос, который ищет все строки таблицы One, не имеющие соответствий в таблице Two, удаляя множество *Два* (делитель) из множества *Один* (делимое).

Внешнее объединение уже содержит строки, находящиеся вне пересечения, так что нам остается только создать запрос объединения разности, использующий внешнее объединение с ограничением IS NULL, накладываемым на первичный ключ второго набора данных.

```
USE Tempdb
SELECT Thing1, Thing2
  FROM dbo.One
    LEFT OUTER JOIN dbo.Two
      ON One.OnePK = Two.OnePK
  WHERE Two.TwoPK IS NULL
```

Разность в первом наборе данных останется следующей:

```
Thing1          Thing2
--------------- ---------------
Old Thing       NULL
Blue Thing      NULL
```

Перенести теорию на сценарий из реальной жизни вы сможете с помощью учебной базы данных OBXKites. В следующем примере запрос на разность множеств находит всех клиентов, еще не разместивших заказы. В данном примере таблица Contact является делимым, а запрос на разность множеств удаляет из нее записи тех клиентов, у которых есть соответствия в таблице [Order]. Предложение WHERE ограничивает результирующий набор данных только теми строками, которым не найдены соответствия в таблице [Order]:

```
USE OBXKites
SELECT LastName, FirstName
  FROM dbo.Contact
    LEFT OUTER JOIN dbo.[Order]
      ON Contact.ContactID = [Order].ContactID
  WHERE OrderID IS NULL
```

В результате получим разность между таблицами Contact и [Order], т.е. список клиентов, которые еще не разместили заказы:

```
LastName      FirstName
------------ ---------------
Andrews       Ed
Boston        Dave
Earl          Betty
Hanks         Nickolas
Harrison      Charlie
```

Запрос разности множеств также можно написать, используя подзапрос. Условие WHERE NOT IN удаляет строки результата подзапроса (делителя) из внешнего запроса (делимого) следующим образом:

```
SELECT LastName, FirstName
  FROM dbo.Contact
  WHERE ContactID NOT IN
   (SELECT ContactID FROM dbo.[Order])
  ORDER BY LastName, FirstName
```

Любая из форм этого запроса (левое внешнее объединение или подзапрос `NOT IN`) хорошо справятся с задачей, имея очень сходные планы выполнения (рис. 9.12).

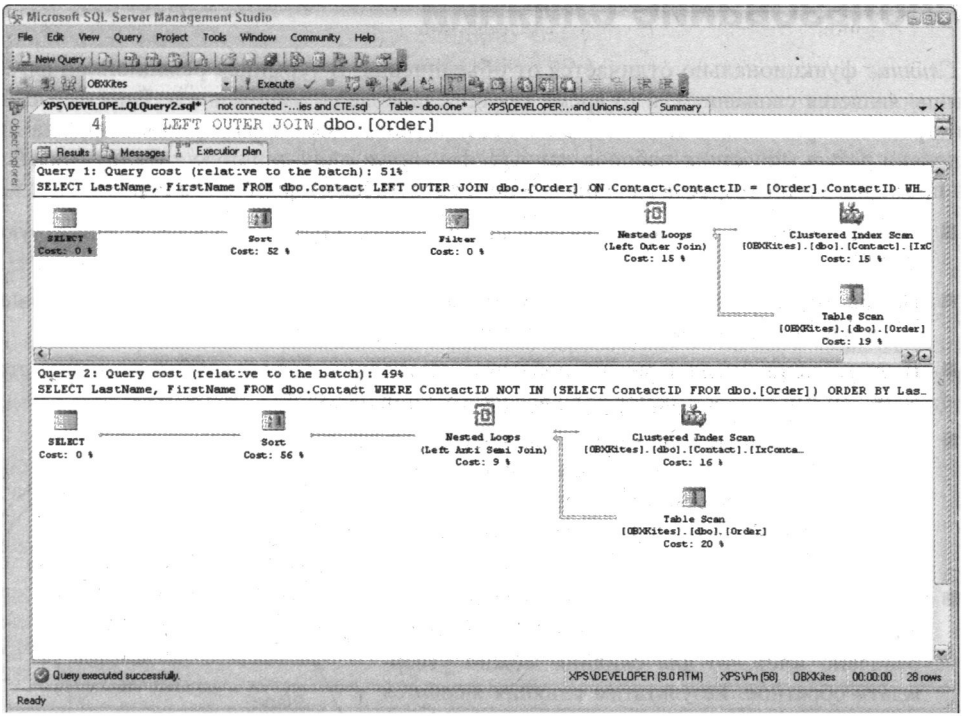


Рис. 9.12. Запрос на разность множеств, использующий подзапрос, оптимизируется практически в тот же план выполнения, что и решение, использующее левое внешнее объединение

Я часто использую модифицированную версию этой методики для удаления некорректных данных, появившихся в ходе преобразований. С логической точки зрения *запрос на разность полных множеств* является прямой противоположностью внутреннему объединению. Он находит все строки вне пересечения в любом из наборов данных, комбинируя полное внешнее объединение с ограничением `WHERE`, отсеивающем только пустые значения в любом из первичных ключей:

```
SELECT Thing1, Thing2
  FROM One
    FULL OUTER JOIN Two
      ON One.OnePK = Two.OnePK
  WHERE Two.TwoPK IS NULL
    OR One.OnePK IS NULL
```

Результатом станут все строки, не имеющие соответствия в таблице на другом конце объединения:

```
Thing1          Thing2
--------------- ---------------
NULL            Plane
NULL            Cycle
Blue Thing      NULL
Old Thing       NULL
```

Использование слияний

Слияние функционально отличается от объединения. В терминах реляционной алгебры слияние является сложением, в то время как объединение — умножением. Вместо расширения строк по горизонтали, как это делает объединение, слияние накладывает друг на друга несколько результирующих наборов данных, формируя одну длинную таблицу (рис. 9.13). При проектировании запросов слияния нужно следовать некоторым правилам.

- Имена столбцов или псевдонимов должны быть определены в первой инструкции SELECT.
- Все инструкции SELECT должны иметь одинаковое число столбцов, и каждая линейка столбцов должна использовать типы данных из одного семейства.
- В инструкции SELECT можно добавлять выражения, определяющие источник строк, если данный столбец включен во все инструкции SELECT.
- Слияния можно использовать как часть инструкции SELECT INTO (эту форму глагола вставки мы подробно рассмотрим в главе 16), однако ключевое слово INTO должно находиться в первой инструкции SELECT.
- Если для инструкции SELECT по умолчанию определено все множество строк (ALL) и не определено иное, то для слияния можно утверждать обратное. Результатом слияния являются уникальные строки. Если вы хотите изменить этот режим, следует явно указать ключевое слово ALL.
- Предложение ORDER BY сортирует результаты всех инструкций SELECT, однако при этом использует имена столбцов из первой инструкции SELECT.

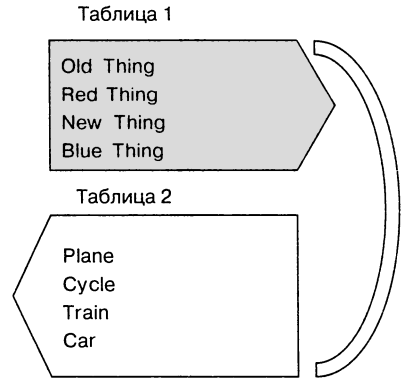


Рис. 9.13. Слияние добавляет по вертикали результаты одной инструкции SELECT к результатам другой

В следующем запросе слияния предложение ORDER BY обращается к столбцу Thing1 первой инструкции SELECT:

```
SELECT OnePK, Thing1, 'из таблицы One' as Source
  FROM dbo.One
UNION ALL
SELECT TwoPK, Thing2, 'из таблицы Two'
  FROM dbo.Two
ORDER BY Thing1
```

Результирующий набор данных использует имена столбцов из первой инструкции SELECT:

```
OnePK       Thing1          Source
----------- --------------- ---------------
4           Blue Thing      из таблицы One
3           Car             из таблицы Two
4           Cycle           из таблицы Two
2           New Thing       из таблицы One
1           Old Thing       из таблицы One
1           Plane           из таблицы Two
3           Red Thing       из таблицы One
2           Train           из таблицы Two
```

Слияния не ограничены всего двумя таблицами — лично мне однажды приходилось работать с 90 таблицами (впоследствии такой подвиг я больше не повторял). На самом деле общее количество таблиц, на которые ссылается запрос, может доходить до 256, и SQL Server успешно справляется с такой нагрузкой.

Слияние пересечения

Слияние пересечения ищет строки, общие для обоих наборов данных. Внутреннее объединение ищет соответствия по горизонтали, а слияние пересечения — по вертикали. SQL Server не рассматривает пересечение и разность как "родные" операции, поэтому они несколько увеличивают нагрузку. Чтобы подготовить базис для примера, вначале дополним таблицу Two двумя строками, создающими пересечение:

```
INSERT dbo.Two(TwoPK, OnePK, Thing2)
  VALUES(5,0, 'Red Thing')
INSERT dbo.Two(TwoPK, OnePK, Thing2)
  SELECT Thing1
    FROM dbo.One
INTERSECT
SELECT Thing2
  FROM dbo.Two
ORDER BY Thing1
```

Результат будет следующим:

```
Thing1
---------------
Blue Thing
Red Thing
```

Запрос слияния пересечения аналогичен внутреннему объединению; в нем каждый столбец участвует в условии ON. Однако запрос слияния пересечения отображает пустые значения как общие и включает такие строки в пересечение, в то время как внутреннее объединение трактует пустые значения как разные и не связывает две строки, содержащие их.



Ключевые слова INTERSECT и EXCEPT впервые появились в версии SQL Server 2005. Несмотря на то что получить корректные результаты для слияний пересечения и разности можно было и в версии SQL Server 2000, в программах приходилось прибегать к обходным маневрам.

Слияние разности/Except

Слияние разности аналогично слиянию пересечения, однако ограничение HAVING выявляет только те строки, которые существуют только в одном из двух наборов данных.

Слияние разности аналогично запросу на разность множеств в том, что ищет все строки, которые присутствуют в одном наборе данных, но отсутствуют в другом. В то время как запрос на разность множеств интересуется только условиями объединения (обычно между первичным и внешним ключами) и объединяет строки по горизонтали, слияние разности смотрит на строки в целом (точнее, на все столбцы, участвующие в инструкциях слияния SELECT) по вертикали.

SQL Server 2005 для выполнения слияния разности использует ключевое слово EXCEPT стандарта ANSI:

```
SELECT Thing1
  FROM dbo.One
EXCEPT
SELECT Thing2
  FROM dbo.Two
ORDER BY Thing1
```

Результат будет следующим:

```
Thing1
---------------
New Thing
Old Thing
```

Резюме

Объединение данных лежит в самом сердце языка SQL. Оно раскрывает всю глубину реляционной алгебры, равно как силу и гибкость SQL. С помощью разных средств, от естественных объединений и до коррелированных подзапросов, SQL отлично справляется с извлечением данных из многочисленных источников. Основной задачей разработчика приложений для SQL Server является освоение теории реляционной алгебры и многочисленных средств языка T-SQL, и это позволит ему эффективно манипулировать данными. Наградой будет полученное удовлетворение.

Несмотря на то что появились технологии .NET и CLR, манипулирование данными с помощью инструкций SELECT остается ключевой технологией SQL Server. И хотя объединения являются более естественным методом работы с реляционными данными, подзапросы открывают многочисленные возможности реализации творческого подхода к извлечению данных из множества источников.

В следующей главе мы рассмотрим ряд способов использования подзапросов в теле запроса. Там же будет представлена новая функция SQL Server — общие табличные выражения (CTE).

Включение данных с помощью подзапросов и CTE

ГЛАВА

10

В этой главе...

Типы подзапросов

Создание простых и коррелированных подзапросов

Использование общих табличных выражений (CTE)

Решение задач с помощью реляционного деления

Реальная сила языка SQL проявляется в его способности комбинировать различные методы извлечения данных. Именно это дает возможность воплощать сложные запросы в коде для выполнения задач, которые нельзя выполнить с помощью графического инструментария. Умение использовать эти средства отличает истинного профессионала от новичка. Поэтому без колебаний я приветствую ваше желание изучить простые и коррелированные подзапросы, управляемые таблицы и общие табличные выражения, а затем применить эти компоненты запроса для решения сложных реляционных задач, таких как реляционное деление.

Методы и расположение

Подзапросом называют внедренную во внешний запрос инструкцию SELECT. Подзапросы дают ответ на вопрос, заданный внешним запросом, в виде скалярного значения, списка значений или набора данных. Подзапросы могут заменять собой выражения, списки и таблицы в структуре внешнего запроса. Матрица типов подзапросов и использования инструкции SELECT приведена в табл. 10.1. Так как подзапросы могут содержать только инструкцию SELECT и никакую другую (например, инструкцию обновления данных), их иногда называют *подвыборками*.

При создании подзапроса возможны три основные формы, в зависимости от необходимых данных и предпочитаемого вами синтаксиса.

- **Простые подзапросы**. Простые подзапросы могут выступать в роли обособленных запросов и запускаться отдельно. Они выполняются один раз, а свой результат передают внешнему запросу. Простые подзапросы создаются как обычные инструкции SELECT и помещаются в скобки.

Таблица 10.1. Использование подзапросов и общих табличных выражений

	Подзапрос возвращает		
	Выражение	**Список**	**Набор данных**
Элемент инструкции `SELECT`	Подзапрос возвращает скалярное значение	Подзапрос возвращает список значений	Подзапрос возвращает источник данных с множеством столбцов
`SELECT` (подзапрос)	Результат подзапроса используется в качестве выражения, поставляющего значение для столбца	X	X
`FROM` (источник данных)			
Это единственное место, где подзапрос может использовать псевдоним таблицы или именованный диапазон	Набор данных подзапроса трактуется как управляемая таблица во внешнем запросе	Набор данных подзапроса трактуется как управляемая таблица во внешнем запросе	Набор данных подзапроса трактуется как управляемая таблица во внешнем запросе
`WHERE` x {=, >, <, >=, <=, <>} (подзапрос)	Предложение `WHERE` истинно, если тестовое значение совпадает с возвращенным подзапросом		
`WHERE` x `IN` (подзапрос)	Условие `WHERE` истинно, если тестовое значение совпадает с возвращенным подзапросом	Условие `WHERE` истинно, если тестовое значение находится в списке, возвращенном подзапросом	X
`WHERE EXISTS` (подзапрос)	Условие `WHERE` истинно, если подзапрос возвращает хотя бы одну строку	Условие `WHERE` истинно, если подзапрос возвращает хотя бы одну строку	Условие `WHERE` истинно, если подзапрос возвращает хотя бы одну строку

- **Общие табличные выражения (CTE)**. Являются синтаксическими вариациями простых подзапросов, аналогичны представлениям и определяют подзапрос в начале запроса с помощью команды `WITH`. К CTE можно обращаться множество раз в основном запросе, как к представлению или управляемой таблице.



Общие табличные выражения были впервые введены в версии SQL Server 2005. Их главной целью является создание рекурсивных запросов. Подробно мы рассмотрим их в главе 12.

- **Коррелированные подзапросы**. Аналогичны простым подзапросам, за исключением того, что они не могут запускаться обособленно. Внешний запрос запускается первым, а коррелированный вызывается по одному разу для каждой строки внешнего запроса.

Простые подзапросы

Порядок выполнения простых подзапросов приведен ниже.

1. Простой подзапрос выполняется один раз.

2. Результаты передаются внешнему запросу.

3. Внешний запрос выполняется один раз.

Самый простой подзапрос возвращает одно (скалярное) значение, которое затем используется как выражение во внешнем подзапросе, например:

```
SELECT (SELECT 3) AS SubqueryValue
```

Результат будет следующим:

```
SubqueryValue
-------------
3
```

Подзапрос (select 3) возвращает скалярное значение 3, которое передается внешней инструкции SELECT. После этого выполняется внешняя инструкция SELECT; при этом она уже имеет следующий вид:

```
SELECT 3 AS SubqueryValue
```

Естественно, подзапросы с запрограммированным значением не имеют никакой ценности. Реальные подзапросы извлекают данные из таблицы, например:

```
USE OBXKites
SELECT ProductName
  FROM dbo.Product
  WHERE ProductCategoryID
    = (Select ProductCategoryID
          FROM dbo.ProductCategory
          Where ProductCategoryName = 'Kite')
```

Выполняя этот запрос, SQL Server вычисляет результаты подзапроса, а затем передает полученное значение внешнему запросу (полученный вами уникальный идентификатор будет отличаться от приведенного в примере):

```
Select ProductCategoryID
    FROM dbo.ProductCategory
    Where ProductCategoryName = 'Kite'
```

Результат будет следующим:

```
ProductCategoryID
------------------------------------
C38D8113-2BED-4E2B-9ABF-A589E0818069
```

После получения результата подзапроса внешний запрос будет иметь следующий вид:

```
SELECT ProductName
FROM dbo.Product
WHERE ProductCategoryID
= 'C38D8113-2BED-4E2B-9ABF-A589E0818069'
```

Результат внешнего запроса:

```
ProductName
--------------------------------------------------
Basic Box Kite 21 inch
Dragon Flight
Sky Dancer
Rocket Kite
...
```

Если вы думаете, что подзапросы похожи на объединения, то вы абсолютно правы. Оба являются средством обращения к множеству источников в одном запросе, и большинство запросов, использующих объединения, можно переписать как запросы, использующие подзапросы.



Используйте объединения для сбора данных из двух источников и дальнейших манипуляций ими как единым целым. Если же данные нужно обработать до объединения, то используйте подзапросы.

Общие табличные выражения

Общие табличные выражения (CTE) определяют, что должно рассматриваться как временное представление, которое в дальнейшем будет использовано в запросе в роли обычного представления. Поскольку CTE можно использовать таким же способом, как и простые подзапросы, я включил их рассмотрение в этот раздел и буду приводить примеры простых подзапросов наряду с CTE.

Общие табличные выражения используют предложение `WITH`, которое определяет их. В предложении `WITH` находятся имя, псевдонимы столбцов и код SQL подзапроса CTE. После этого основной запрос может обратиться к CTE как к источнику данных:

```
WITH CTEName (параметры)
AS (простой_подзапрос)
SELECT...
  FROM CTEName
```

Следующий код идентичен приведенному в предыдущем примере, только он имеет формат CTE. Именем табличного выражения является `CTEQuery`; оно возвращает столбец `ProductionCategoryID` и использует точно такую же инструкцию `SELECT`, как и подзапрос, приведенный в предыдущем примере:

```
WITH CTEQuery (ProductCategoryID)
AS (Select ProductCategoryID
      FROM dbo.ProductCategory
      Where ProductCategoryName = 'Kite')
```

После того как общее табличное выражение было определено в предложении `WITH`, основная часть запроса может обращаться к нему по имени, как к обычному табличному источнику, например таблице или представлению. Приведем полный пример, содержащий как основной запрос, так и CTE:

```
WITH CTEQuery (ProductCategoryID)
AS (Select ProductCategoryID
      FROM dbo.ProductCategory
      Where ProductCategoryName = 'Kite')
SELECT ProductName
  FROM dbo.Product
  WHERE ProductCategoryID
    = (SELECT ProductCategoryID FROM CTEQuery)
```

Чтобы включить в один запрос множество CTE, их нужно последовательно определить перед основным запросом:

```
WITH CTE1Name (параметры)
AS (простой подзапрос)
WITH CTE2Name (параметры)
AS (простой подзапрос)
SELECT...
FROM CTE1Name
JOIN CTE2Name
ON ...
```

Несмотря на то что CTE сами могут содержать сложные запросы, они имеют два ключевых ограничения.

- В отличие от подзапросов они не могут быть вложены друг в друга.
- Они не могут ссылаться на главный запрос. Подобно простым подзапросам, они должны быть состоятельны сами по себе.



Пусть синтаксис общих табличных выражений может изначально показаться чужеродным, в очень сложных запросах, ссылающихся на один и тот же подзапрос во многих местах, использование CTE позволяет значительно сократить объем кода и повысить его читаемость.

За все время тестирования я не заметил разницы в производительности между простым подзапросом и CTE, — они оба компилируются в один и тот же план выполнения.

Использование скалярных подзапросов

Если подзапрос возвращает одно значение, то он может использоваться в любом месте инструкции SELECT, где может использоваться выражение, в частности в выражениях столбцов, а также в условиях JOIN, WHERE и HAVING. Обычные операторы (такие, как +, –, between и т.п.) будут работать со скалярным значением, возвращаемым подзапросом, однако при этом могут потребоваться функции преобразования типов, такие как cast() или convert().

В примере, приведенном в предыдущем разделе, использовался подзапрос в условии WHERE. В следующем примере мы используем подзапрос в выражении для столбца, вычисляя общий объем продаж, который будет затем использован для получения процентного отношения к нему продажи по данной строке:

```
SELECT ProductCategoryName,
    SUM(Quantity * UnitPrice) AS Sales,
    Cast(SUM(Quantity * UnitPrice) /
           (SELECT SUM(Quantity * UnitPrice)
              FROM dbo.OrderDetail) *100 AS INT)
        AS PercentOfSales
  FROM dbo.OrderDetail
    JOIN dbo.Product
      ON OrderDetail.ProductID = Product.ProductID
    JOIN dbo.ProductCategory
      ON Product.ProductCategoryID = ProductCategory.ProductCategoryID
  GROUP BY ProductCategoryName
  ORDER BY Count(*) DESC
```

Подзапрос SELECT SUM(Quantity * UnitPrice) FROM dbo.OrderDetail возвращает значение 1729,895, которое затем передается столбцу PercentageOfSales внешнего запроса.

```
ProductCategoryName   Sales          PercentOfSales
--------------------- -------------- --------------
Kite                  1499.902500    86.70
OBX                   64.687500      3.74
Clothing              113.600000     6.57
Accessory             10.530000      0.61
Material              5.265000       0.30
Video                 35.910000      2.08
```

Следующая инструкция SELECT извлекается из пользовательской функции fGetPrice() учебной базы данных OBXKites. Эта база данных имеет таблицу цен для конкретных дат, при

этом магазин может предварительно определять некоторые цены для будущих дат, чтобы не вводить их в ночь накануне изменения. В дополнение эта модель ценообразования позволяет хранить историю цен на товары.

Функция `fGetPrice()` возвращает корректную цену для любого продукта на заданную дату с учетом любой скидки. Для достижения этой цели функция должна определить действующую цену для любой заданной даты. Например, если пользователю нужна цена на 16 июля 2002 года, а текущая цена вступила в силу 1 июля того же года, запросу нужно узнать ближайшую дату установки цены, используя выражение `max(@orderdate)`. Как только подзапрос определит эту дату, внешний запрос извлечет нужную цену. С целью демонстрации примера некоторые переменные этой функции были заменены статическими значениями:

```
SELECT @CurrPrice = Price * (1-@DiscountPercent)
  FROM dbo.Price
    JOIN dbo.Product
      ON Price.ProductID = Product.ProductID
  WHERE ProductCode = '1001'
    AND EffectiveDate =
      (SELECT MAX(EffectiveDate)
          FROM dbo.Price
             JOIN dbo.Product
                  ON Price.ProductID = Product.ProductID
        WHERE ProductCode = '1001'
           AND EffectiveDate <= '6/1/2001')
```

При вызове функции

```
Select dbo.fGetPrice('1001','5/1/2001',NULL)
```

подзапрос определяет, что датой вступления в силу цены является `'05/01/2001'`. После этого внешний запрос может найти корректную цену, основываясь на идентификаторе товара и дате вступления в силу цены. После того как функция `fGetPrice()` вычислит скидку, она вернет значение `@CurrPrice` вызывающей инструкции `SELECT`.

```
14,95
```

Использование подзапросов в качестве списков

Подзапросы предстают во всей своей красе, когда их начинают использовать в качестве списков. При этом некоторое значение, как правило, столбца внешнего запроса, сравнивается со списком, возвращенным подзапросом, с помощью оператора `IN`. Подзапрос должен возвращать только один столбец, в противном случае произойдет ошибка.

Оператор `IN` возвращает истинное значение (`true`), если значение столбца найдено в списке, возвращенном подзапросом; таким же образом действует выражение `WHERE...IN` при работе с запрограммированным списком:

```
SELECT *
  FROM dbo.Contact
  WHERE HomeRegion IN ('NC', 'SC', 'GA', 'AL', 'VA')
```

Подзапрос, возвращающий список, служит механизмом динамического формирования списка для условия `WHERE...IN`:

```
SELECT *
  FROM dbo.Contact
  WHERE Region IN (Подзапрос, возвращающий список состояний)
```

Приведенный в следующем примере запрос дает ответ на вопрос: "Что еще покупают в магазине OBX Kites при покупке воздушного змея?" В этом запросе мы будем использовать

исключительно подзапросы — никаких объединений. Все эти подзапросы являются простыми, в том смысле, что каждый из них можно выполнить обособленно как запрос.

Этот запрос найдет все заказы, содержащие воздушных змеев, и передаст их идентификаторы главному запросу. В получении ответа на вопрос будут задействованы четыре таблицы: `ProductCategory`, `Product`, `OrderDetail` и `Order`. Вложенные подзапросы выполняются от самого внутреннего к внешнему.

1. Подзапрос находит идентификатор категории для воздушных змеев (`ProductCategoryID`).
2. Подзапрос находит список продуктов, попадающих в категорию воздушных змеев (т.е. с найденным идентификатором).
3. Подзапрос находит список заказов, содержащих воздушных змеев.
4. Подзапрос находит список товаров, содержащихся в заказах с воздушными змеями.
5. Внешний запрос извлекает названия этих товаров.

```
SELECT ProductName
 FROM dbo.Product
  WHERE ProductID IN
   -- 4. Поиск всех товаров, содержащихся в заказах с воздушными змеями
   (SELECT ProductID
     FROM dbo.OrderDetail
           WHERE OrderID IN
     -- 3. Поиск заказов с воздушными змеями
      (SELECT OrderID -- Find the Orders with Kites
       FROM dbo.OrderDetail
              WHERE ProductID IN
       -- 2. Поиск товаров категории воздушных змеев
        (SELECT ProductID
          FROM dbo.Product
          WHERE ProductCategoryID =
          -- 1. Поиск категории воздушных змеев
           (SELECT ProductCategoryID
             FROM dbo.ProductCategory
             WHERE ProductCategoryName
               = 'Kite' ) ) ) )
```



При желании вы можете выделить любой из подзапросов в окне Query и запустить его как обособленный запрос, нажав клавишу <F5>.

Подзапрос 1 находит идентификатор категории змеев (`ProductCategoryID`) и возвращает одно значение.

Подзапрос 2 использует возвращенное подзапросом 1 значение в предложении `WHERE` для составления списка товаров, у которых поле `ProductCategoryID` имеет это значение.

Подзапрос 3 использует подзапрос 2 в качестве списка в предложении `WHERE` и ищет все строки таблицы `OrderDetail`, содержащие любой из идентификаторов товаров (`ProductID`), возвращенных подзапросом 2.

Подзапрос 4 использует подзапрос 3 в качестве списка в предложении `WHERE` и ищет все строки таблицы `OrderDetail`, идентификатор заказа (`OrderID`) которых содержится в списке, возвращенном запросом 3.

Внешний запрос использует подзапрос 4 в качестве списка в условии предложения `WHERE` и находит названия всех товаров, идентификаторы которых (`ProductID`) вернул подзапрос 4. Результат получается следующим:

```
ProductName
-----------------------
Falcon F-16
Dragon Flight
OBX Car Bumper Sticker
Short Streamer
Cape Hatteras T-Shirt
Sky Dancer
Go Fly a Kite T-Shirt
Long Streamer
Rocket Kite
OBX T-Shirt
```

Несложно заметить, что в полученный список попали и сами воздушные змеи. Чтобы исключить их из результатов запроса, во внешний запрос нужно внести некоторые изменения.

- Идентификатор товара (`ProductID`) должен находиться в заказе, содержащем список змеев.
- Идентификатор товара не должен содержаться в списке змеев.

Мы знаем, что подзапрос 2 возвращает список идентификаторов всех воздушных змеев. Если добавить копию этого подзапроса в оператор `NOT IN` внешнего запроса, из списка будут исключены все воздушные змеи:

```
SELECT ProductName
 FROM dbo.Product
  WHERE ProductID IN
   -- 4. Поиск всех товаров, содержащихся в заказах с воздушными змеями
   (SELECT ProductID
     FROM dbo.OrderDetail
           WHERE OrderID IN
     -- 3. Поиск заказов с воздушными змеями
      (SELECT OrderID -- Find the Orders with Kites
       FROM dbo.OrderDetail
              WHERE ProductID IN
       -- 2. Поиск товаров категории воздушных змеев
        (SELECT ProductID
          FROM dbo.Product
          WHERE ProductCategoryID =
          -- 1. Поиск категории воздушных змеев
           (SELECT ProductCategoryID
             FROM dbo.ProductCategory
             WHERE ProductCategoryName
               = 'Kite' ) ) ) )
     -- продолжаем внешний запрос
     AND ProductID NOT IN
      (SELECT ProductID
                FROM dbo.Product
               WHERE ProductCategoryID =
                  (SELECT ProductCategoryID
                      FROM dbo.ProductCategory
                      WHERE ProductCategoryName
                           = 'Kite'))
```

Результат запроса будет следующим:

```
ProductName
--------------------------------------------------
OBX Car Bumper Sticker
Short Streamer
```

```
Cape Hatteras T-Shirt
Go Fly a Kite T-Shirt
Long Streamer
OBX T-Shirt
```

Для сравнения приведем еще один запрос, выполняющий ту же задачу, но написанный с помощью объединений. Таблица `Product` дважды участвует в ссылках, поэтому вторая ссылка имеет именованный диапазон `Kite`. Как и в предыдущих подзапросах, первая версия возвращает в результатах весь список товаров, а вторая исключает из него воздушных змеев:

```
SELECT Distinct Product.ProductName
  FROM dbo.Product
    JOIN dbo.OrderDetail OrderRow
      ON Product.ProductID = OrderRow.ProductID
    JOIN dbo.OrderDetail KiteRow
      ON OrderRow.OrderID = KiteRow.OrderID
    JOIN dbo.Product Kite
      ON KiteRow.ProductID = Kite.ProductID
    JOIN dbo.ProductCategory
      ON Kite.ProductCategoryID
         = ProductCategory.ProductCategoryID
  WHERE ProductCategoryName = 'Kite'
```

Единственным изменением, которое нужно внести для исключения воздушных змеев, является дополнительное условие в объединении с таблицей `ProductCategory`. Ранее это было объединение равенства между таблицами `Product` и `ProductCategory`. Добавление условия Θ-объединения (`!=`) между таблицами `Product` и `ProductCategory` устраняет все товары, попадающие в категорию воздушных змеев, как показано в следующем примере:

```
SELECT Distinct Product.ProductName
  FROM dbo.Product
    JOIN dbo.OrderDetail OrderRow
      ON Product.ProductID = OrderRow.ProductID
    JOIN dbo.OrderDetail KiteRow
      ON OrderRow.OrderID = KiteRow.OrderID
    JOIN dbo.Product Kite
      ON KiteRow.ProductID = Kite.ProductID
    JOIN dbo.ProductCategory
      ON Kite.ProductCategoryID
         = ProductCategory.ProductCategoryID
      AND Product.ProductCategoryID
         !=Kite.ProductCategoryID
  WHERE ProductCategoryName = 'Kite'
```

Мы рассмотрели два множества запросов, использующих различный синтаксис, но дающих ответ на один и тот же вопрос. Какой из них лучше? Все зависит от вас. В зависимости от сложности подзапросы могут выполняться быстрее, поскольку они отбирают на каждом шаге меньшее количество строк. Более сложные подзапросы имеют тенденцию выполняться быстрее, чем большие запросы с объединениями.



SQL является очень гибким языком — обычно в нем существуют десятки способов задать один и тот же вопрос. Выбор метода в первую очередь зависит от предпочитаемого вами стиля — текст должен быть читаемым и логически корректным. И уже затем рассматривается вопрос быстродействия. Медленные, но правильные запросы всегда лучше быстрых, но некорректных.

Приведем еще один пример, в котором творческий подход к созданию подзапроса помогает справиться с задачей. SQL с легкостью справляется с отбором десятка наибольших или наименьших значений; гораздо сложнее обстоят дела с отбором среднего диапазона. В наши

дни, когда поиск в Web возвращает сотни, если не тысячи ссылок, поиск строк, например со 101-й по 125-ю, может оказаться довольно полезным.

В приведенном ниже примере мы снова будем использовать учебную базу данных OBXKites. Мы отберем пятерку товаров, начиная с 26-го. Подзапрос ищет первые 25 товаров, после чего они пропускаются в основном запросе с помощью предложения WHERE NOT IN:

```
USE OBXKites
SELECT TOP 5 ProductName, ProductID
  FROM dbo.Product
  WHERE ProductID NOT IN
    (SELECT TOP 25 ProductID
      FROM dbo.Product
      ORDER BY ProductID)
  ORDER BY ProductID
```

Получим следующий результат:

```
ProductName              ProductCode
------------------------ ---------------
Handle                   1026
Third Line Release       1027
High Performance Line    1028
Kite Bag                 1029
Kite Repair Kit          1030
```

Использование подзапросов в качестве таблиц

Так же как представление может быть использовано вместо таблицы в предложении FROM инструкции SELECT, подзапрос в форме *управляемой таблицы* может заменить таблицу, если он имеет именованный диапазон. Этот метод часто используется для разбиения сложных задач на несколько меньших.

Использование подзапроса в качестве управляемой таблицы — это отличное решение задач консолидации данных. При создании итогового запроса все столбцы должны участвовать в консолидации тем или иным образом либо в предложении GROUP BY, либо в итоговой функции (SUMM(), MIN(), COUNT(), MAX() или AVERAGE()). Это соглашение усложняет получение дополнительной информации, в частности описаний. В то же время выполнение итоговых функций в подзапросах и передача найденных строк внешнему запросу в качестве управляемых таблиц позволяет последнему получить любые нужные столбцы.



Подробную информацию об итоговых функциях и предложении GROUP BY вы найдете в главе 11.

На вопрос "Какое количество каждого товара было продано?" легко ответить, если включить в результат только один столбец из таблицы Product:

```
SELECT ProductCode, SUM(Quantity) AS QuantitySold
  FROM dbo.OrderDetail
    JOIN dbo.Product
      ON OrderDetail.ProductID = Product.ProductID
GROUP BY ProductCode
```

Получим следующий результат:

```
ProductCode     QuantitySold
--------------- ------------
1002            47.00
1003            5.00
```

```
1004            2.00
1012            5.00
```

В результат был включен код товара (ProductCode), но не были включены ни названия, ни описания. Естественно, можно группировать значения по столбцам, но такой подход будет слишком грубым. Приведенный ниже запрос выполняет консолидацию значений в подзапросах, которые затем объединяются с таблицей Product. Таким образом, мы получаем доступ ко всем столбцам, не выполняя дополнительной работы.

```
SELECT Product.ProductCode, Product.ProductName,
     Sales.QuantitySold
  FROM dbo.Product
  JOIN (SELECT ProductID, SUM(Quantity) AS QuantitySold
          FROM dbo.OrderDetail
          GROUP BY ProductID) Sales
    ON Product.ProductID = Sales.ProductID
  ORDER BY ProductCode
```

Если вы используете конструктор запросов утилиты Management Studio, то управляемую таблицу можно добавить к запросу. На рис. 10.1 показан приведенный выше запрос, спроектированный в графическом интерфейсе этой утилиты.

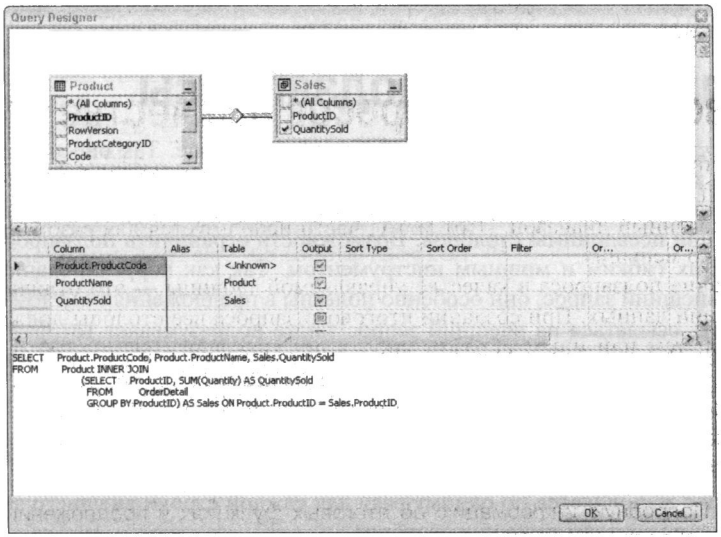


Рис. 10.1. Управляемые таблицы можно добавить в запрос в конструкторе запросов с помощью пункта **Add Derived Table** *контекстного меню*

Этот запрос быстрый и эффективный; он выполняет необходимую консолидацию данных, при этом все столбцы можно добавить в результат. Результат запроса, приведенного в качестве примера:

```
ProductCode ProductName          QuantitySold
----------- -------------------- ------------------
1002        Dragon Flight        47.00
1003        Sky Dancer           5.00
1004        Rocket Kite          2.00
1012        Falcon F-16          5.00
...
```

Еще один пример использования управляемой таблицы позволяет ответить на вопрос "Сколько детей родила каждая из матерей?" с помощью учебной базы данных Family:

```
USE Family
SELECT PersonID, FirstName, LastName, Children
  FROM dbo.Person
    JOIN (SELECT MotherID, COUNT(*) AS Children
            FROM dbo.Person
            WHERE MotherID IS NOT NULL
            GROUP BY MotherID) ChildCount
      ON Person.PersonID = ChildCount.MotherID
  ORDER BY Children DESC
```

Подзапрос выполняет суммирование, и эти столбцы объединяются с таблицей `Person` для представления окончательных результатов:

```
PersonID FirstName  LastName Children
-------- ---------  -------- ---------
6        Audry      Halloway 4
8        Melanie    Campbell 3
12       Alysia     Halloway 3
20       Grace      Halloway 2
...
```

Коррелированные подзапросы

Термин *коррелированные подзапросы* звучит впечатляюще. Так оно и есть. Они используются так же, как и простые подзапросы; отличие состоит в том, что коррелированные подзапросы ссылаются на столбцы внешнего запроса, при этом они используют его именованный диапазон или псевдонимы столбцов. Возможность ограничить подзапросы внешним запросом делает их гибким и мощным инструментом. Так как коррелированные подзапросы ссылаются на внешний запрос, они особенно полезны в предложениях `WHERE`.

Возможность ссылаться на внешний запрос также подразумевает тот факт, что коррелированные подзапросы невозможно запустить самостоятельно, так как попытка обратиться к внешнему запросу закончится неудачей. Логический порядок выполнения коррелированных подзапросов приведен ниже.

1. Внешний запрос выполняется один раз.
2. Подзапрос выполняется по одному разу для каждой строки внешнего запроса, используя значения соответствующих столбцов.
3. Результаты подзапроса интегрируются в результирующий набор данных.

Если внешний запрос возвращает 100 строк, то SQL Server выполнит логический эквивалент 101 запроса: один раз выполнит внешний запрос и по одному разу выполнит подзапрос для каждой строки его результата. На самом деле оптимизатор найдет способ, как выполнить коррелированный подзапрос, фактически не выполняя 101 запрос. На практике мне приходилось видеть, когда коррелированные подзапросы превосходили по производительности другие планы запросов. Если они способны решить стоящую перед вами задачу, не пренебрегайте ими только из соображений производительности.

При рассмотрении коррелированных подзапросов в качестве примера мы будем использовать учебную базу данных Outer Banks Adventures. При этом мы будем сравнивать местоположение заказчиков и базовых лагерей. В первую очередь мы скорректируем данные таблиц с помощью следующего пакета запросов:

```
USE CHA2
UPDATE dbo.BaseCamp SET Region = 'NC' WHERE BaseCampID = 1
UPDATE dbo.BaseCamp SET Region = 'NC' WHERE BaseCampID = 2
UPDATE dbo.BaseCamp SET Region = 'BA' WHERE BaseCampID = 3
UPDATE dbo.BaseCamp SET Region = 'FL' WHERE BaseCampID = 4
UPDATE dbo.BaseCamp SET Region = 'WV' WHERE BaseCampID = 5
UPDATE dbo.Customer SET Region = 'ND' WHERE CustomerID = 1
UPDATE dbo.Customer SET Region = 'NC' WHERE CustomerID = 2
UPDATE dbo.Customer SET Region = 'NJ' WHERE CustomerID = 3
UPDATE dbo.Customer SET Region = 'NE' WHERE CustomerID = 4
UPDATE dbo.Customer SET Region = 'ND' WHERE CustomerID = 5
UPDATE dbo.Customer SET Region = 'NC' WHERE CustomerID = 6
UPDATE dbo.Customer SET Region = 'NC' WHERE CustomerID = 7
UPDATE dbo.Customer SET Region = 'BA' WHERE CustomerID = 8
UPDATE dbo.Customer SET Region = 'NC' WHERE CustomerID = 9
UPDATE dbo.Customer SET Region = 'FL' WHERE CustomerID = 10
```

На основании местоположения базовых лагерей и клиентов сформированный набор данных позволяет создать матрицу:

```
SELECT DISTINCT Customer.Region, BaseCamp.Region
  FROM dbo.Customer
    JOIN dbo.Event_mm_Customer
      ON Customer.CustomerID = Event_mm_Customer.CustomerID
    JOIN dbo.Event
      ON Event_mm_Customer.EventID = Event.EventID
    JOIN dbo.Tour
      ON Event.TourID = Tour.TourID
    JOIN dbo.BaseCamp
      ON Tour.BaseCampID = BaseCamp.BaseCampID
  WHERE Customer.Region IS NOT NULL
  GROUP BY Customer.Region, BaseCamp.Region
  ORDER BY Customer.Region, BaseCamp.Region
```

Получим следующий результат:

```
Customer BaseCamp
Region   Region
-------  ------
BA       BA
BA       FL
BA       NC
FL       FL
FL       NC
FL       WV
NC       BA
NC       FL
NC       NC
NC       WV
ND       BA
ND       FL
ND       NC
NE       FL
NE       WV
NJ       FL
NJ       NC
NJ       WV
```

Имея в наличии эти данные, можно задать первый вопрос: "Кто живет в регионе, в котором находится один из базовых лагерей?" Запрос использует коррелированный подзапрос для

поиска базовых лагерей, которые находятся в регионе заказчика. Этот подзапрос выполняется для каждой строки таблицы Customer, используя именованный диапазон внешнего запроса (C) для обращения к нему. Если существует некоторый базовый лагерь, соответствующий данной строке, то условие EXISTS становится истинным и строка добавляется в результирующий набор данных:

```
SELECT C.FirstName, C.LastName, C.Region
  FROM dbo.Customer AS C
  WHERE EXISTS
   (SELECT * FROM dbo.BaseCamp AS B
      WHERE B.Region = C.Region)
  ORDER BY LastName, FirstName
```

Тот же запрос, написанный с помощью объединений, требует наличия предиката DISTINCT во избежание дублирования строк. Однако он может обращаться к столбцам каждой из таблиц, на которые имеет ссылку, чего иногда не может сделать коррелированный подзапрос, находящийся в предложении WHERE EXISTS:

```
SELECT DISTINCT C.FirstName, C.LastName, C.Region
  FROM Customer C
  JOIN dbo.BaseCamp B
    ON C.Region = B.Region
  ORDER BY LastName, FirstName
```

Получим следующий результат:

```
FirstName LastName      Region
--------  ----------  --------
Jane      Doe         BA
Francis   Franklin    FL
Melissa   Anderson    NC
Lauren    Davis       NC
Wilson    Davis       NC
John      Frank       NC
```

Можно сформулировать более сложный вопрос: "Кто заказывал тур в своем регионе?"

Ответ находится в таблице Event_mm_Customer — разрешающей (или объединяющей) таблице между таблицами Event и Customer, служащей для хранения логических отношений "многие ко многим" между клиентами и событиями (много клиентов могут участвовать в одном мероприятии, а один клиент может посещать несколько мероприятий). Таблицу Event_mm_Customer можно воспринимать как аналог билета, приобретенного одним заказчиком на одно мероприятие.

Внешний запрос логически запускается для каждой строки таблицы Event_mm_Customer, чтобы определить, даст ли операция EXISTS какой-либо результат из коррелированного подзапроса. Подзапрос осуществляет фильтрацию по текущему идентификатору события (EventID) и региону клиента (RegionID), взятому из внешнего запроса.

В информационном смысле запрос проверяет каждый билет и создает список мероприятий в родном регионе клиента, которые он посещал. Если в списке что-нибудь есть, то условие WHERE EXISTS для данной строки становится истинным. Если список пуст, то условие WHERE EXISTS не удовлетворяется и рассматриваемая строка не попадает в результирующий набор данных:

```
USE CHA2
SELECT DISTINCT C.FirstName, C.LastName, C.Region AS Home
  FROM dbo.Customer C
  JOIN dbo.Event_mm_Customer E
    ON C.CustomerID = E.CustomerID
  WHERE C.Region IS NOT NULL
```

```
    AND EXISTS
        (SELECT *
          FROM dbo.Event
          JOIN dbo.Tour
           ON Event.TourID = Tour.TourID
          JOIN dbo.BaseCamp
           ON Tour.BaseCampID = BaseCamp.BaseCampID
          WHERE BaseCamp.Region = C.Region
           AND Event.EventID = E.EventID)
```

Будет получен следующий результат:

```
FirstName LastName    Home
--------- ----------- ------
Francis   Franklin    FL
Jane      Doe         BA
John      Frank       NC
Lauren    Davis       NC
Melissa   Anderson    NC
```

Тот же запрос можно написать и с использованием объединений. Несмотря на то что его легче читать, его выполнение заняло на моем компьютере 131 миллисекунду, по сравнению с 80 миллисекундами времени выполнения предыдущего примера, использующего коррелированные подзапросы.

```
SELECT Distinct C.FirstName, C.LastName, C.Region AS Home,
Tour.TourName, BaseCamp.Region
  FROM dbo.Customer C
  JOIN dbo.Event_mm_Customer
    ON C.CustomerID = Event_mm_Customer.CustomerID
  JOIN dbo.Event
    ON Event_mm_Customer.EventID = Event.EventID
  JOIN dbo.Tour
    ON Event.TourID = Tour.TourID
  JOIN dbo.BaseCamp
    ON Tour.BaseCampID = BaseCamp.BaseCampID
      AND C.Region = BaseCamp.Region
      AND C.Region IS NOT NULL
  ORDER BY C.LastName
```

Этот запрос имеет преимущество: он включает столбцы из таблицы `Tour`, не требуя их явного возвращения подзапросом. В то же время в результатах этого запроса клиенты *Lauren Davis и John Frank* упоминаются дважды — по одному разу для каждого тура:

```
FirstName LastName    Home   TourName                 Region
--------- ----------- ------ ------------------------ --------
Melissa   Anderson    NC     Outer Banks Lighthouses  NC
Lauren    Davis       NC     Appalachian Trail        NC
Lauren    Davis       NC     Outer Banks Lighthouses  NC
Jane      Doe         BA     Bahamas Dive             BA
John      Frank       NC     Appalachian Trail        NC
John      Frank       NC     Outer Banks Lighthouses  NC
Francis   Franklin    FL     Amazon Trek              FL
```

Несмотря на то что коррелированные подзапросы могут быть трудными для восприятия, их гибкость и потенциальная выгода в производительности компенсируют это с лихвой. Просто проверьте, возвращает ли коррелированный подзапрос корректный результат.

Реляционное деление

Перекрестное объединение, о котором мы говорили ранее в этой главе, представляет собой реляционное произведение — два набора данных умножаются друг на друга, чтобы создать декартово произведение. С теоретической точки зрения все объединения являются перекрестными с применением некоторых условных ограничений. Даже внутреннее объединение является результатом произведения двух таблиц с применением к результату ограничения на совпадение значений.

Реляционное деление дополняет реляционное произведение так же, как в математике операция деления дополняет умножение. Если целью реляционного умножения является получение произведения двух множеств-множителей, то целью реляционного деления является деление одного множества (делителя) на другое (делитель), чтобы найти множество-частное (рис. 10.2). Другими словами, если известно картезианское произведение, реляционное деление поможет найти недостающий множитель.

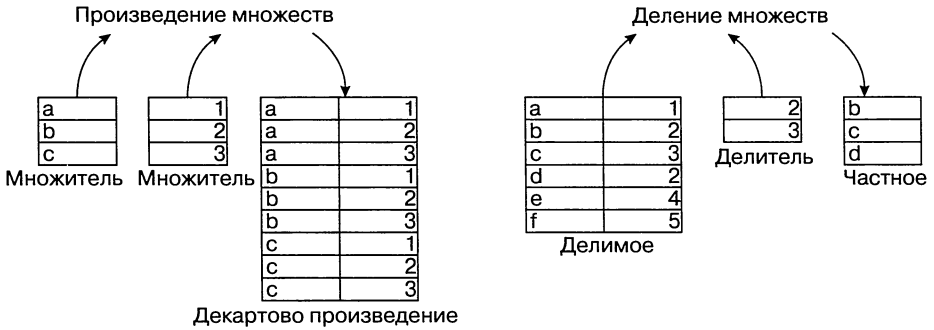


Рис. 10.2. Реляционное деление является операцией, обратной реляционному произведению. Оно вычисляет множество-частное путем деления множества-делимого на множество-делитель

Несмотря на то что определение реляционного деления звучит несколько академически, оно может очень пригодиться на практике. Классической пример реляционного деления дает ответ на вопрос: "Какие студенты посещали все необходимые курсы?" Запрос *точного реляционного деления* выявит только тех студентов, которые посетили только требуемые курсы. Запрос *реляционного деления с остатком*, также называемый *приблизительным делением*, перечислит всех студентов, которые посетили все нужные курсы, включив в список и тех, кто посещал дополнительные курсы. Естественно, последняя операция чаще применяется на практике.

Реляционное деление — более сложное понятие, чем объединение. Объединение просто ищет соответствия между двумя наборами данных. Объединения/подзапросы и реляционное деление отвечают на два качественно разных типа вопросов. В качестве примера применим следующие вопросы к учебной базе данных и сравним два данных метода.

- Объединения/подзапросы
  - **CHA2**: Кто когда-либо ездил в тур?
  - **CHA2**: Кто живет в том же регионе, где находится базовый лагерь?
  - **CHA2**: Кто участвовал в событиях в своем родном регионе?
- Точное реляционное деление
  - **CHA2**: Кто брал все туры в своем родном регионе и ни одного вне его?

- **OBXKites**: Кто покупал только воздушных змеев и ничего больше?
- **Family**: Какие женщины (включая вдов и разведенных) выходили замуж за одних и тех же мужчин и больше ни за каких других?

■ Реляционное деление с остатком

- **CHA2**: Кто брал все туры в своем родном регионе и, возможно, какие-либо другие?
- **OBXKites**: Кто покупал всех воздушных змеев и, может быть, что-либо еще?
- **Family**: Какие женщины выходили замуж за одних и тех же мужчин и, может быть, за кого-либо еще?

Реляционное деление с остатком

Реляционное деление с остатком выделяет частное, оставляя в то же время место для строк, которые удовлетворяют заданному критерию, но содержат и дополнительные данные. На практике такой тип деления оказывается более полезным, чем точное деление.

Для демонстрации реляционного деления с остатком мы выберем упомянутый ранее вопрос, задаваемый базе данных OBXKites: "Кто покупал воздушных змеев и, может быть, что-либо еще?" Так как в запросе участвуют пять таблиц, от категории товаров до строк заказа, и так как вопрос предполагает наличие объединения таблиц `OrderDetails` и `Product`, запрос будет достаточно сложный, чтобы промоделировать реальную задачу, встречающуюся на практике и связанную с реляционными базами данных.

Категория игрушек послужит хорошим примером категории, поскольку содержит только два товара, и в учебной базе данных нет информации об их покупке. Вопрос мы сформулируем следующим образом: "Кто покупал хотя бы по одной игрушке каждого типа, продаваемой в магазине OBX Kites?" (Мои дети предложили стать добровольцами в решении этой задачи.)

В реализации сценария нам помогут вымышленные данные, вставляемые в базу данных OBXKites. К категории игрушек относятся только товары с кодами 1049 и 1050. База данных OBXKites использует уникальные идентификаторы для первичных ключей и соответственно для вставки использует хранимые процедуры.

В первых вызовах хранимых процедур вставки в таблицы `Order` и `OrderDetails` будут перечислены их параметры, так что их будет несложно понять:

```
USE OBXKites
DECLARE @OrderNumber INT
```

Первый клиент с кодом 110 приобрел только игрушки:

```
EXEC pOrder_AddNew
  @ContactCode = '110',
  @EmployeeCode = '120',
  @LocationCode = 'CH',
  @OrderDate= '6/1/2002',
  @OrderNumber = @OrderNumber output
EXEC pOrder_AddItem
  @OrderNumber = @OrderNumber,
  @Code = '1049',
  @NonStockProduct = NULL,
  @Quantity = 12,
  @UnitPrice = NULL,
  @ShipRequestDate = '6/1/2002',
  @ShipComment = NULL
EXEC pOrder_AddItem
  @OrderNumber, '1050', NULL, 3, NULL, NULL, NULL
```

Второй клиент с кодом 111 также приобрел только игрушки:

```
EXEC pOrder_AddNew
  '111', '119', 'JR', '6/1/2002', @OrderNumber output
EXEC pOrder_AddItem
  @OrderNumber, '1049', NULL, 6, NULL, NULL, NULL
EXEC pOrder_AddItem
  @OrderNumber, '1050', NULL, 6, NULL, NULL, NULL
EXEC pOrder_AddNew
  '111', '119', 'JR', '6/1/2002', @OrderNumber output
EXEC pOrder_AddItem
  @OrderNumber, '1050', NULL, 6, NULL, NULL, NULL
```

Третий клиент с кодом 112 приобрел игрушки и еще некоторые товары:

```
EXEC pOrder_AddNew
  '112', '119', 'JR', '6/1/2002', @OrderNumber output
EXEC pOrder_AddItem
  @OrderNumber, '1049', NULL, 6, NULL, NULL, NULL
EXEC pOrder_AddItem
  @OrderNumber, '1050', NULL, 5, NULL, NULL, NULL
EXEC pOrder_AddItem
  @OrderNumber, '1001', NULL, 5, NULL, NULL, NULL
EXEC pOrder_AddItem
  @OrderNumber, '1002', NULL, 5, NULL, NULL, NULL
```

Четвертый клиент с кодом 113 приобрел только одну игрушку:

```
EXEC pOrder_AddNew
  '113', '119', 'JR', '6/1/2002', @OrderNumber output
EXEC pOrder_AddItem
  @OrderNumber, '1049', NULL, 6, NULL, NULL, NULL
```

Другими словами, только клиенты 110 и 111 приобретали только игрушки; клиент 112 приобрел игрушки, равно как и воздушные змеи; клиент 113 не должен попасть в выборку, так как он приобрел всего одну игрушку.

Существует совсем немного методов программирования задач реляционного деления. Оригинальный метод, предложенный Крисом Дейтом, использует вложенные коррелированные подзапросы для поиска строк в заданном множестве и вне его. Более прямой метод был популяризирован Джо Селко; он использует сравнение количества строк наборов данных делителя и делимого.

В своей основе решение Селко строится на перефразировании вопроса: "У кого количество приобретенных игрушек совпадает с количеством доступных?"

В запросе задаются два вопроса. Внешний запрос группирует заказы с игрушками для каждого клиента, а подзапрос подсчитывает количество товаров в категории игрушек. После этого предложение HAVING внешнего запроса сравнивает число различных товаров, приобретенных клиентом в категории игрушек, с общим количеством товаров в этой категории:

```
-- Является ли количество приобретенных игрушек...
SELECT Contact.ContactCode
  FROM dbo.Contact
  JOIN dbo.[Order]
    ON Contact.ContactID = [Order].ContactID
  JOIN dbo.OrderDetail
    ON [Order].OrderID = OrderDetail.OrderID
  JOIN dbo.Product
    ON OrderDetail.ProductID = Product.ProductID
  JOIN dbo.ProductCategory
    ON Product.ProductCategoryID
      =ProductCategory.ProductCategoryID
```

```
  WHERE ProductCategory.ProductCategoryName = 'Toy'
  GROUP BY Contact.ContactCode
  HAVING COUNT(DISTINCT Product.ProductCode) =
-- ...равным количеству доступных игрушек?
     (SELECT Count(ProductCode)
        FROM dbo.Product
        JOIN dbo.ProductCategory
          ON Product.ProductCategoryID
              = ProductCategory.ProductCategoryID
        WHERE ProductCategory.ProductCategoryName = 'Toy')
```

Будет получен следующий результат:

```
ContactCode
---------------
110
111
112
```

Точное реляционное деление

Точное реляционное деление ищет точные соответствия без какого-либо остатка. Оно оставляет основной вопрос реляционного деления с остатком и сужает метод так, чтобы делимое не содержало дополнительных строк, вызывающих появление остатка.

На практике это означает, что задается вопрос наподобие следующего: "Кто приобрел все игрушки, но больше ничего?"

Если адресовать этот вопрос модифицированной форме метода Селко, то он будет перефразирован следующим образом: "У кого количество приобретенных игрушек равно количеству доступных игрушек и общему количеству приобретенных товаров?" Если клиент покупал дополнительные товары, отличные от игрушек, то последняя часть вопроса отсеет его из результирующего набора данных.

Следующий код SQL содержит два главных изменения по сравнению с предыдущим. Первое заключается в том, что внешний запрос должен найти и количество приобретенных игрушек, и общее количество приобретенных товаров. Это выполняется с помощью поиска приобретенных игрушек в управляемой таблице и объединения двух наборов данных. Второе изменение заключается в модификации предложения HAVING для сравнения количества доступных игрушек как с количеством приобретенных игрушек, так и с общим количеством приобретенных товаров:

```
-- Точное реляционное деление
-- Является ли количество всех приобретенных товаров...
SELECT Contact.ContactCode
  FROM dbo.Contact
  JOIN dbo.[Order]
    ON Contact.ContactID = [Order].ContactID
  JOIN dbo.OrderDetail
    ON [Order].OrderID = OrderDetail.OrderID
  JOIN dbo.Product
    ON OrderDetail.ProductID = Product.ProductID
  JOIN dbo.ProductCategory P1
    ON Product.ProductCategoryID = P1.ProductCategoryID
  JOIN
-- ... и количество приобретенных игрушек...
      (SELECT Contact.ContactCode, Product.ProductCode
         FROM dbo.Contact
         JOIN dbo.[Order]
```

```
          ON Contact.ContactID = [Order].ContactID
        JOIN dbo.OrderDetail
          ON [Order].OrderID = OrderDetail.OrderID
        JOIN dbo.Product
          ON OrderDetail.ProductID = Product.ProductID
        JOIN dbo.ProductCategory
          ON Product.ProductCategoryID =
             ProductCategory.ProductCategoryID
       WHERE ProductCategory.ProductCategoryName = 'Toy'
       ) ToysOrdered
    ON Contact.ContactCode = ToysOrdered.ContactCode
  GROUP BY Contact.ContactCode
  HAVING COUNT(DISTINCT Product.ProductCode) =
-- ...равным количеству доступных игрушек...
     (SELECT Count(ProductCode)
       FROM dbo.Product
       JOIN dbo.ProductCategory
         ON Product.ProductCategoryID
          = ProductCategory.ProductCategoryID
       WHERE ProductCategory.ProductCategoryName = 'Toy')
--... и равным общему количеству всех приобретенных товаров?
         AND COUNT(DISTINCT ToysOrdered.ProductCode) =
          (SELECT Count(ProductCode)
            FROM dbo.Product
            JOIN dbo.ProductCategory
              ON Product.ProductCategoryID
                 = ProductCategory.ProductCategoryID
            WHERE ProductCategory.ProductCategoryName = 'Toy')
```

Результатом выполнения этого запроса станет список клиентов, содержащий количество приобретенных игрушек (2) и общее количество приобретенных товаров (2). Оба этих числа равны количеству доступных игрушек (2):

```
ContactCode
-----------
110
111
```

Резюме

Хотя основные принципы создания подзапросов могут показаться простыми, они открывают безбрежные возможности. Они позволяют создавать сложные вложенные запросы, которые перетасовывают данные так, чтобы они приняли форму, необходимую для решения сложной задачи. Если вы будете постоянно работать с подзапросами, то согласитесь с моим утверждением, что именно в них заключена основная сила языка SQL. И если до сих пор вы создавали запросы в графическом интерфейсе, то подзапросы станут тем катализатором, который переместит ваше рабочее место создания запросов в текстовый редактор.

В предыдущих главах мы заложили фундамент работы с SQL, рассмотрев инструкцию SELECT, выражения, объединения и слияния. В этой главе мы дополнили инструкцию SELECT возможностями подзапросов и общих табличных выражений. Если вы читаете эту книгу последовательно, примите поздравления: вы перешли Рубикон в изучении языка SQL. Если вы освоили реляционную алгебру и подзапросы, все остальное — сущие пустяки.

В следующей главе будет продолжено описание инструментов извлечения данных. В ней мы коснемся запросов консолидации данных, в которых подзапросы играют немаловажную роль.

Консолидация данных

ГЛАВА

11

В этой главе...

Вычисление сумм и средних

Статистический анализ

Группировка данных в запросе

Решение проблемных задач консолидации данных

Создание динамических перекрестных запросов

Основные принципы информационной архитектуры, изложенные в главе 1, гласят, что в активе находится информация, а не просто данные. Преобразование сырых списков данных и ключей в полезную информацию часто требует консолидации и группировки данных некоторым осмысленным образом. Несмотря на то что определенную часть задач консолидации и анализа можно выполнить с помощью служб анализа и отчетности SQL Server 2005, бо́льшую часть задач группировки и консолидации можно выполнить и непосредственно в инструкции SQL `SELECT`.

Язык SQL успешно справляется с суммированием, вычислением максимальных, минимальных и средних значений всего множества данных или некоторого его сегмента. К тому же SQL позволяет создавать *перекрестные таблицы*, также называемые *сводными*.



В то время как стандарт ANSI SQL-92 включает в себя массу стандартных итоговых функций, SQL Server 2005 добавил к этому возможность создавать собственные итоговые функции с помощью общеязыковой исполняющей среды CLR. У меня нет никаких сомнений, что вскоре появятся библиотеки дополнительных итоговых функций, созданные сторонними разработчиками.

Простая консолидация

Основная сущность запросов консолидации заключается в возвращении вместо всех выбранных строк только одной, консолидирующей исходный набор данных (рис. 11.1). При этом могут использоваться различные типы вычислений — от суммирования до основных статистических операций.

Важно отметить, что в логическом порядке выполнения запроса SQL итоговые функции следуют сразу за предложением `FROM` и фильтрами `WHERE`. Это значит, что данные могут быть собраны и отфильтрованы до консолидации, без необходимости использования подзапросов.

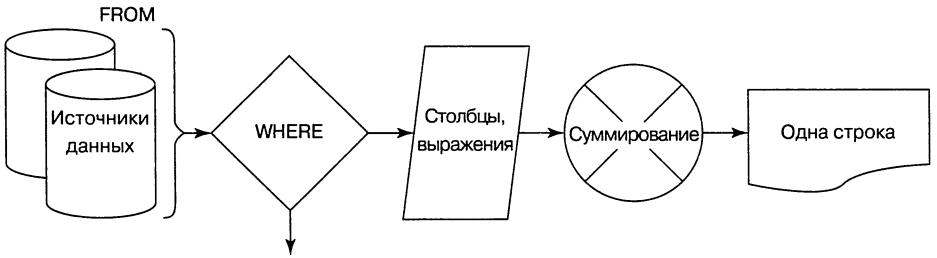


Рис. 11.1. Итоговая функция создает одну строку на базе набора данных

Основные итоговые функции

Язык SQL содержит множество итоговых функций, которые можно использовать в качестве выражений в инструкции `SELECT` для получения итоговых данных (табл. 11.1).

Таблица 11.1. Основные итоговые функции

Итоговая функция	Поддерживаемый тип данных	Описание
`sum()`	Числовой	Суммирует все непустые значения в столбце
`avg()`	Числовой	Усредняет все непустые значения в столбце. Возвращается тип данных, соответствующий исходному, так что часто его приходится явно преобразовывать к большей точности, например `avg(cast col as float)`
`min()`	Числовой, строковый, даты-времени	Возвращает из столбца наименьшее число, самую раннюю дату-время или первую строку, соответствующую текущему порядку символов
`max()`	Числовой, строковый, даты-времени	Возвращает из столбца наибольшее число, самую последнюю дату-время или последнюю строку, соответствующую текущему порядку символов
`count ([distinct] *)`	Любой тип данных	Выполняет подсчет всех строк в результирующем наборе данных вплоть до 2147483647. Не выполняет подсчет уникальных идентификаторов и пустых значений
`oucnt_big ([distinct*])`	Любой тип данных	Аналогична функции `count()`, но использует тип данных `bigint`; в результате может подсчитать вплоть до $2^{63}-1$ строк



Для примеров кода, приведенных в настоящей главе, использована небольшая таблица `RowData`. Программа для создания и заполнения этого набора данных приведена в начале сценария настоящей главы. Также вы можете загрузить данный сценарий с сайта `www.SQLServerBible.com`.

```
CREATE TABLE RawData (
  Region VARCHAR(10),
  Category CHAR(1),
  Amount INT,
  SalesDate DateTime
  )
```

Следующий простой запрос консолидации данных подсчитывает количество строк в таблице и сумму значений столбца Amount. Вместо возвращения реальных строк из таблицы RawData, этот запрос формирует одну итоговую строку с количеством и суммой значений:

```
SELECT
     Count(*) as Count,
     Sum(Amount) as [Sum]
  FROM RawData
```

Будет получен следующий результат:

```
Count       Sum
----------- -----------
20          946
```



Пустые значения не учитываются ни одной итоговой функцией или другой операцией SET.

Если вы используете конструктор запросов утилиты Management Studio, то любой запрос можно преобразовать в запрос консолидации данных с помощью кнопки Group By панели инструментов (рис. 11.2). Столбец Group By используется для выбора столбцов запроса для итоговых функций и группировки. Чтобы найти конструктор запросов, откройте таблицу в окне Object Explorer.

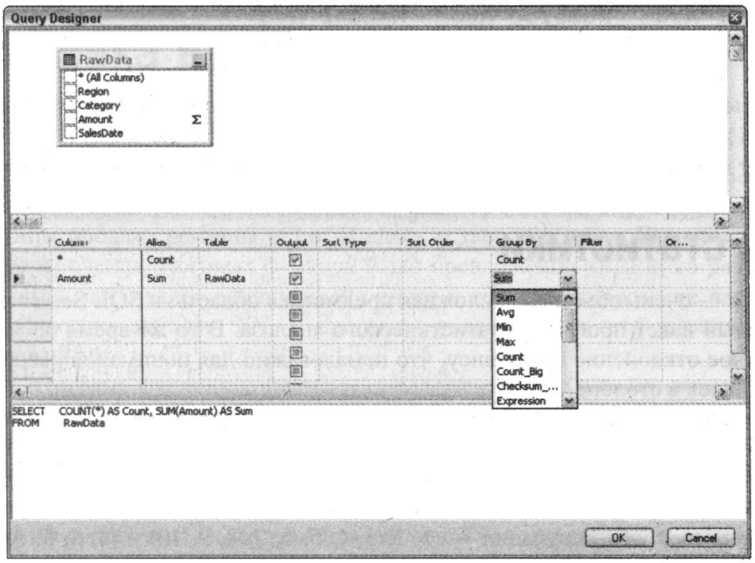


Рис. 11.2. Выполнение запроса консолидации данных в конструкторе запросов утилиты Management Studio

Использовать итоговые функции в инструкции SELECT довольно просто, однако при этом нельзя забывать некоторые правила.

- Так как SQL при использовании итоговых функций в запросе возвращает информацию из множества, а не создает набор данных, все столбцы (содержащиеся в списке столбцов инструкции или в предложении ORDER BY) должны участвовать в каких-либо итоговых функциях. Это вполне логично, поскольку если в некоторой строке запрос

возвращает общее количество принятых заказов, то он не вернет в ней же номер конкретного заказа.

- Параметр консолидации DISTINCT служит той же цели, что и выражение SELECT DISTINCT, за исключением того, что в данном случае предотвращается дублирование не строк, а значений. По этой причине этот параметр не имеет смысла использовать в функции sum() или avg(). Функция count(distinct *) не верна: в ней явно должен быть указан столбец.
- Функция count(*) подсчитывает все строки, в то время как функция count (*столбец*) подсчитывает только непустые значений столбца.
- Поскольку итоговые функции являются выражениями, результат будет иметь пустое имя столбца. По этой причине для именования таких столбцов используйте псевдонимы.
- Точность итоговых функций определяется точностью типа данных исходного столбца. Например, если столбец имеет целочисленный тип, результатом функции усреднения avg() также будет целое число. Преобразование данных в тип numeric(9,5) может повысить точность результата:

```
SELECT Avg(amount) as [Integer Avg],
  Avg(Cast((Amount)as Numeric(9,5))) as [Numeric Avg],
    Sum(amount) / Count(*) as [Manual Avg]
  FROM RawData
```

Будет получен следующий результат:

```
Integer Avg   Numeric Avg   Manual Avg
------------- ------------- ----------------
47            47.300000     39
```

- Запросы консолидации данных игнорируют пустые значения, так что функции sum() и avg() не завершатся по ошибке, когда их встретят. В то же время по этой причине результат выражения sum()/count(*) может отличаться от результата функции avg().

Основы статистики

Статистика — очень объемная и сложная предметная область, и SQL Server не претендует заменить полный пакет программ статистического анализа. В то же время он может подсчитать стандартное отклонение и вариансу, что немаловажно для понимания распределения чисел, содержащихся в столбце.

Одного среднего значения совершенно недостаточно для консолидации множества значений (в терминах статистики "множество" называют *популяцией*). Значение, находящееся в самом центре популяции, называют *медианой* (это не одно и то же, что и среднее значение). Средняя ширина полосы разброса данных относительно значения медианы называется *вариансой* популяции. Например, обе популяции — (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) и (4, 4, 5, 5, 5, 5, 6, 6) — имеют среднее значение 5, но значения первого множества сильнее отклоняются от медианы, чем второго. Стандартное отклонение равно квадратному корню вариансы — оно описывает форму колоколообразной кривой, формируемой распределением популяции.

Следующий запрос использует функции StDevP() и VarP() для получения стандартного отклонения и вариансы всей популяции таблицы RawData:

```
SELECT
    StDevP(Amount) as [StDev],
    VarP(Amount) as [Var]
 FROM RawData
```

Будет получен следующий результат:

```
StDevP                VarP
--------------------- --------
24.2715883287435      589.11
```



Если вам нужно выполнить расширенный статистический анализ данных, то рекомендуется экспортировать набор данных в программу Excel, где вам будет предложен широкий выбор статистических функций.

При вычислении вариансы и стандартного отклонения используемые формулы слегка отличаются для случаев, когда в расчет берется вся популяция или только выборка из нее[1]. Если запрос консолидации принимает в расчет всю популяцию, используйте функции `StDevP()` и `VarP()` — они используют *смещенный* или *n-метод* вычисления отклонения.

В то же время, если запрос использует выборку из популяции (т.е. ее подмножество), используйте итоговые функции `StDev()` и `Var()`, которые используют *несмещенный метод* или *метод n–1*. Так как группировка данных в запросе разбивает популяцию на подмножества, в таких запросах рекомендуется использовать функции `StDev()` и `Var()`.



О функциях расстановки, включая вычисляющие процентили, см. в главе 7.

Группировка в результирующем наборе данных

Функции консолидации хороши, однако часто ли у вас возникает потребность в консолидации данных всей таблицы? Обычно в данных задачах в расчет берут диапазоны дат, подразделения, типы продаж, регионы и т.п. И здесь встает проблема. Если бы единственным механизмом ограничения итоговых функций было предложение `WHERE`, то разработчики баз данных часами бы занимались репликацией одного и того же запроса или написанием множества динамических запросов SQL и программ последовательного выполнения запросов консолидации данных.

К счастью, итоговые функции дополнены предложением `GROUP BY`, которое автоматически разбивает набор данных на подмножества, основываясь на значении некоторого столбца. Когда данные разделены на подгруппы, итоговая функция выполняется в каждой из них. В результате в каждой группе появляется дополнительная итоговая строка (рис. 11.3).

Самым распространенным примером группировки является подсчет объема продаж в разрезе торговых. Функция `sum()`, будучи примененной без группировки, вернет в результате сумму всех продаж. Написание запроса, учитывающего каждого торгового представителя, потребует всего лишь вставить функцию `sum()` для каждого из них, однако постоянное выполнение такой работы способно вызвать головную боль. Функция группировки же автоматически создаст подмножества данных для каждого уникального торгового представителя, после чего будет вычислено значение функции `sum()` для их объемов продаж. Вот и все.

---

[1] *Если вас заинтересуют упоминаемые далее формулы вычисления вариансы и стандартного отклонения, зайдите на сайт* `http://rrc.dgu.ru/res/www.statsoft.ru/home/portal/glossary/GlossaryTwo/S/StandardDeviation.htm`. — Примеч. ред.

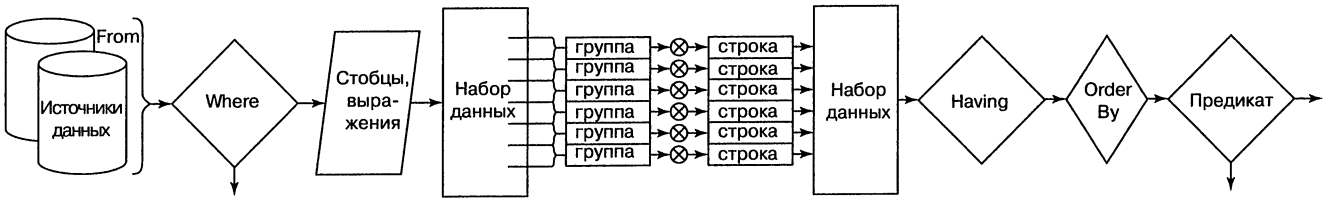


Рис. 11.3. Предложение `GROUP BY` *разбивает набор данных на множество подгрупп*

Простая группировка

Некоторые группировки используют описательную информацию, поэтому данные, используемые в предложении GROUP BY, достаточны для понимания группировки. Однако такой подход больших реляционных базах данных встречается редко. Например, следующий запрос выполняет группировку по категориям:

```
SELECT Category,
    Count(*) as Count,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData
GROUP BY Category
```

Будет получен следующий результат:

```
Category  Count     Sum       Avg       Min       Max
--------- --------- --------- --------- --------- ---------
X         5         225       45        11        86
Y         11        506       46        12        91
Z         4         215       53        33        83
```

В первом столбце этот запрос возвращает столбец Category. Несмотря на то что этот столбец не имеет итоговой функции, он все равно участвует в консолидации, поскольку именно по нему выполняется группировка, и по этой причине он включается в результирующий набор данных. Каждая строка в результирующем наборе данных консолидирует одну категорию; при этом итоговые функции вычисляют количество строк, сумму, а также среднее, минимальное и максимальное значения для каждой из категорий.

SQL не ограничивает вас группировкой только по одному столбцу. Приведенный выше запрос можно расширить, дополнив группировкой по полю ProductCategoryName:

```
SELECT Year(SalesDate) as [Year], DatePart(q,SalesDate) as [Quarter],
    Count(*) as Count,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData
    GROUP BY Year(SalesDate), DatePart(q,SalesDate)
```

Будет получен следующий результат:

```
Year   Quarter   Count   Sum   Avg   Min   Max
------ --------- ------- ----- ----- ----- -----
2006   1         6       218   36    11    62
2006   2         6       369   61    33    86
2006   3         8       280   70    54    91
2005   4         4       79    19    12    28
```

Решение проблем в запросах консолидации данных

Существует несколько аспектов группировки в запросах, которые могут создать сложности при разработке приложений. Некоторые программисты вообще избегают запросов консолидации данных и возлагают эта задачу на отчеты. В то же время ядро базы данных может

оказаться куда более эффективным, чем любая из клиентских утилит. Далее будут описаны пять типичных проблем консолидации данных и предложены мои рекомендации по их решению.

Включение групп по описаниям

Приведенные выше в качестве примеров запросы консолидации данных выполняются без ошибок, поскольку все столбцы участвуют в итогах. Для тестирования проблемы мы создадим с помощью следующего пакета инструкций таблицу категорий, а затем попытаемся вернуть столбец, не включенный в итоговую функцию или группировку:

```
CREATE TABLE RawCategory (
    RawCategoryID CHAR(1),
    CategoryName VARCHAR(25)
)
INSERT RawCategory (RawCategoryID, CategoryName)
  VALUES ('X', 'Sci-Fi')
INSERT RawCategory (RawCategoryID, CategoryName)
  VALUES ('Y', 'Philosophy')
INSERT RawCategory (RawCategoryID, CategoryName)
  VALUES ('Z', 'Zoology')
-- включение данных, находящихся за пределами группировки и итогов
SELECT Category, CategoryName,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData R
  JOIN RawCategory C
    ON R.Category = C.RawCategoryID
  GROUP BY Category
```

Как и ожидалось, включение поля `region` в список столбцов привело к следующей ошибке выполнения запроса:

```
Msg 8120, Level 16, State 1, Line 1
Column 'RawCategory.CategoryName' is invalid in the select list
because it is not contained in either an aggregate function or
the GROUP BY clause.
```

Существуют два решения проблемы включения в запрос неконсолидируемых описательных столбцов.

Если создается одноразовый запрос, запустите его на выполнение и щелкните на кнопке OK, чтобы включить дополнительные столбцы в предложение `GROUP BY`.

```
SELECT Category, CategoryName,
   Sum(Amount) as [Sum],
   Avg(Amount) as [Avg],
   Min(Amount) as [Min],
   Max(Amount) as [Max]
  FROM RawData R
  JOIN RawCategory C
    ON R.Category = C.RawCategoryID
  GROUP BY Category, CategoryName
  ORDER BY Category, CategoryName
```

Будет получен следующий результат:

```
Category CategoryName    Sum   Avg   Min   Max
-------- -------------- ----- ----- ----- -----
X        Sci-Fi         225   45    11    86
Y        Philosophy     506   46    12    91
Z        Zoology        215   53    33    83
```

Проблема данного подхода состоит в том, что он форсирует выполнение сервером операции группировки по всем столбцам, независимо от того, действительно ли необходима такая группировка. Разумеется, это оказывает ненужное отрицательное влияние на быстродействие запроса.

Для запросов, используемых в производственных условиях, лучшим решением будет вынести итоговые функции в подзапрос, а затем включить дополнительные столбцы во внешний запрос.

```
SELECT sq.Category, CategoryName,
    sq.[Sum], sq.[Avg], sq.[Min], sq.[Max]
  FROM (SELECT Category,
           Sum(Amount) as [Sum],
           Avg(Amount) as [Avg],
           Min(Amount) as [Min],
           Max(Amount) as [Max]
         FROM RawData
         GROUP BY Category ) sq
  JOIN RawCategory C
    ON sq.Category = C.RawCategoryID
  ORDER BY Category, CategoryName
```

Подзапрос должен выполнить работу по консолидированию данных с группировкой по категориям. Чтобы вызвать имя категории, результаты подзапроса передаются во внешний запрос, который использует объединение с таблицей `RawCategory` для доступа к ее столбцу `CategoryName`.

Включение всех групп с помощью значений

Группировка в логическом порядке выполнения запроса функционально следует за предложением `WHERE`. Это может создать проблему, если требуется создать отчет по всем группам, даже если некоторые из них были отфильтрованы предложением `WHERE` и не содержат значений. Хотя такая необходимость возникает достаточно редко, существует решение данной проблемы запроса консолидации данных, не требующее внешних объединений и подзапросов. Параметр `ALL` предложения `GROUP BY` позволяет включить в результат запроса все группы, даже если их значения были отфильтрованы предложением `WHERE`. Следующий запрос демонстрирует использование этого приема. Он возвращает строку для группы `2005`, несмотря на то, что данные за 2005 год не участвуют в итоговых вычислениях:

```
SELECT Year(SalesDate) as Year,
    Count(*) as Count,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg],
    Min(Amount) as [Min],
    Max(Amount) as [Max]
  FROM RawData
  WHERE Year(SalesDate) = 2006
  GROUP BY ALL Year(SalesDate)
```

Будет получен следующий результат:

```
Year   Count  Sum   Avg    Min    Max
------ ------ ----- ------ ------ -----
2005   0      NULL  NULL   NULL   NULL
2006   20     867   54     11     91
```

Вложенные консолидации

Консолидация данных часто может оказаться полезной. На практике иногда оказывается даже более полезным консолидировать уже полученные итоговые данные. Например, запрос консолидации данных может с легкостью просуммировать данные по каждой категории и году

(кварталу) с помощью подзапроса. Однако может возникнуть вопрос, в какой из категорий содержится максимальное значение. Совершенно очевидно, что композиция функций max(sum()) не сработает, поскольку в ней недостаточно информации для SQL Server о том, как вкладывать эти группировки. Решение этой задачи требует, чтобы подзапрос создал набор данных на основе первой консолидации, а затем внешний запрос реализовал второй уровень консолидации. Например, следующий запрос выполняет суммирование значений по году и категории, а затем внешний запрос с помощью функции max() определяет, какая из сумм оказалась в каждом квартале наибольшей:

```
SELECT Y,Q, Max(Sum) as MaxSum
  FROM ( -- Вычисление сумм
         SELECT Category, Year(SalesDate) as Y,
             DatePart(q,SalesDate) as Q, Sum(Amount) as Sum
           FROM RawData
           GROUP BY Category, Year(SalesDate),
              DatePart(q,SalesDate)
         ) sq
  GROUP BY Y,Q
  ORDER BY Y,Q
```

В результате выполнения запроса будет получен следующий результат:

```
Y     Q    MaxSum
----- ---- ------
2005  4    79
2006  1    147
2006  2    215
2006  3    280
```

Включение описания деталей

Естественно, неплохо знать, что максимальный объем продаж среди 147 категорий в первом квартале 2006 года достиг некоторой величины. Однако кому захочется вручную искать, к какой именно категории принадлежит эта сумма? Следующим логическим шагом станет включение в результаты запроса информации о консолидированных данных. Чтобы добавить такую информацию для строк деталей, объединим их значения с подзапросом:

```
SELECT MaxQuery.Y, MaxQuery.Q, AllQuery.Category,
       MaxQuery.MaxSum as MaxSum
  FROM (-- Поиск максимальной суммы за квартал/год
    SELECT Y,Q, Max(Sum) as MaxSum
    FROM ( -- Вычисление сумм
           SELECT Category, Year(SalesDate) as Y,
                  DatePart(q,SalesDate) as Q, Sum(Amount) as Sum
             FROM RawData
             GROUP by Category, Year(SalesDate),
                    DatePart(q,SalesDate)
                ) sq
               GROUP BY Y,Q
          ) MaxQuery
JOIN (-- Все данные запроса
      SELECT Category, Year(SalesDate) as Y,
             DatePart(q,SalesDate) as Q,
             Sum(Amount) as Sum
        FROM RawData
        GROUP BY Category, Year(SalesDate),
             DatePart(q,SalesDate)
       ) AllQuery
```

```
    ON MaxQuery.Y = AllQuery.Y
       AND MaxQuery.Q = AllQuery.Q
       AND MaxQuery.MaxSum = AllQuery.Sum
  ORDER BY MaxQuery.Y, MaxQuery.Q
```

Будет получен следующий результат:

```
Y      Q    Category MaxSum
------ ---- -------- -----------
2005   4    Y        79
2006   1    Y        147
2006   2    Z        215
2006   3    Y        280
```

Хотя на первый взгляд данный запрос кажется сложным, на самом деле он является всего лишь расширением предыдущего.

Последний подзапрос (с псевдонимом `AllQuery`) ищет все суммы в категориях за год/квартал. Объединение `MaxQuery` с `AllQuery` по сумме и году/кварталу используется для поиска названия категории, что позволяет вывести в результаты запроса вместе с данными и их описания.

Фильтрация сгруппированных результатов

Фильтрация в комбинации с группировкой может вызвать проблемы. Возникает вопрос, применяются ли ограничения до группировки или после нее. Некоторые базы данных используют вложенные запросы для корректной фильтрации до или после предложения `GROUP BY`. Язык SQL для фильтрации групп содержит специальное предложение `HAVING`. В начале этой главы вы видели упрощенный порядок выполнения инструкции `SELECT`. Более полный порядок следующий.

1. Предложение `FROM` собирает данные из разных источников.
2. Предложение `WHERE` ограничивает число строк на основании некоторого условия.
3. Предложение `GROUP BY` собирает подмножества данных.
4. Вычисляются итоговые функции.
5. Предложение `HAVING` фильтрует подмножества данных.
6. Вычисляются все выражения.
7. Предложение `ORDER BY` сортирует результат.

Продолжая работать с учебной таблицей `RawData`, мы рассмотрим еще один запрос, который удаляет из результатов все группы, имеющие среднее значение, меньшее двадцати пяти:

```
SELECT Year(SalesDate) as [Year],
    DatePart(q,SalesDate) as [Quarter],
    Count(*) as Count,
    Sum(Amount) as [Sum],
    Avg(Amount) as [Avg]
  FROM RawData
  GROUP BY Year(SalesDate), DatePart(q,SalesDate)
  HAVING Avg(Amount) > 25
   ORDER BY [Year], [Quarter]
```

Будет получен следующий результат:

```
Year   Quarter   Count   Sum   Avg
------ --------- ------- ----- -----
2006   1         6       218   36
2006   2         6       369   61
2006   3         8       280   70
```

Без предложения HAVING четвертый квартал 2005 года, имеющий среднее значение 19, вошел бы в результирующий набор данных. Однако предложение HAVING, включенное после GROUP BY и вычисления итоговых функций, послужило постконсолидационным фильтром.

Генерирование итогов

Служба отчетности позволяет без труда добавить итоговые и промежуточные суммы, не требуя написания запросов, но может оказаться полезным вычислять итоги для приложений .NET, чтобы отображать их в окне формы или Web-страницы. Следующие три итоговые команды являются идеальным решением этой задачи.

Запросы сведения

Итоговые функции rollup и cube генерируют общие и промежуточные итоги как отдельные строки и подставляют пустое значение в столбец group by, чтобы обозначить общий итог. Функция rollup последовательно генерирует строки общих и промежуточных итогов для столбцов group by. Функция cube расширяет эти возможности, генерируя промежуточные итоги для всех столбцов group by. Специальная функция grouping() имеет истинное значение, когда строка является общим или промежуточным итогом. В этом разделе я продемонстрирую функцию rollup.

Параметр ROLLUP, помещенный в предложение GROUP BY, дает указание SQL Server генерировать дополнительную строку итогов. В приведенном ниже примере функция rollup() используется в выражении CASE для преобразования стандартной итоговой строки в нечто осмысленное.

```
SELECT
    CASE Grouping(Category)
      WHEN 0 THEN Category
      WHEN 1 THEN 'All Categories'
    END AS Category,
    Count(*) as Count
  FROM RawData
  GROUP BY Category
     WITH ROLLUP
```

Результат выполнения запроса будет следующим:

```
Category          Count
-----------       -----------
X                 5
Y                 15
Z                 4
All Categories    24
```

Добавляя второй столбец, Year(SalesDate), с подведением промежуточных итогов по годам, мы указываем SQL Server вычислять по нему промежуточные итоги:

```
SELECT
    CASE Grouping(Category)
      WHEN 0 THEN Category
      WHEN 1 THEN 'All Categories'
    END AS Category,
    CASE Grouping(Year(SalesDate))
      WHEN 0 THEN Cast(Year(SalesDate) as CHAR(8))
      WHEN 1 THEN 'All Years'
```

```
    END AS Year,
    Count(*) as Count
  FROM RawData
  GROUP BY Category, Year(SalesDate)
    WITH ROLLUP
```

Результат этого запроса следующий:

```
Category        Year        Count
--------------  ----------- -----------
X               2006        5
X               All Years   5
Y               2005        4
Y               2006        11
Y               All Years   15
Z               2006        4
Z               All Years   4
All Categories All Years   24
```

Кубические запросы

Кубические запросы являются логическим развитием запросов сведения. Они добавляют промежуточные итоги для всех измерений. В рассмотренном ранее примере запрос сведения создавал промежуточные итоги для каждой категории; кубический запрос создаст итоги и для каждого года.

```
SELECT
    CASE Grouping(Category)
      WHEN 0 THEN Category
      WHEN 1 THEN 'All Categories'
    END AS Category,
    CASE Grouping(Year(SalesDate))
      WHEN 0 THEN Cast(Year(SalesDate) as CHAR(8))
      WHEN 1 THEN 'All Years'
    END AS Year, Count(*) as Count
  FROM RawData
  GROUP BY Category, Year(SalesDate)
    WITH CUBE
  ORDER BY IsNull(Category, 'zzz')
```

Результат запроса будет следующим:

```
Category        Year       Count
--------------- ---------- -----------
X               2006       5
X               All Years  5
Y               2005       4
Y               2006       11
Y               All Years  15
Z               2006       4
Z               All Years  4
All Categories All Years  24
All Categories 2005       4
All Categories 2006       20
```

Вычисления итогов

Предложение COMPUTE полностью отличается от всех запросов консолидации данных. Добавленное в конец, оно позволяет превратить любой обычный запрос в запрос консолида-

ции данных. Такой запрос вернет обычный набор данных со строками деталей, а затем добавит в его конец сводную информацию о том же наборе данных. В следующем примере возвращается вся таблица RowData, за которой следует нечто подобное второму набору данных со средними значениями и суммами:

```
SELECT Category, SalesDate, Amount
  FROM RawData
  WHERE Year(SalesDate) = '2006'
  COMPUTE Avg(Amount), sum(Amount)
```

Результат выполнения запроса будет следующим:

```
Category SalesDate               Amount
-------- ----------------------- -----------
X        2006-01-01 00:00:00.000 11
X        2006-01-01 00:00:00.000 24
...
Y        2006-08-01 00:00:00.000 NULL
avg         sum
----------- -----------
54          867
```

Предложение COMPUTE может даже содержать собственное миниатюрное предложение GROUP BY. В таком случае все строки деталей будут разделены на группы со своими промежуточными итогами, почти как в полноценном отчете:

```
SELECT Category, SalesDate, Amount
  FROM RawData
  WHERE Year(SalesDate) = '2006'
  ORDER BY Category
  COMPUTE Avg(Amount), sum(Amount)
    BY Category
```

Результат запроса будет следующим:

```
Category SalesDate               Amount
-------- ----------------------- -----------
X        2006-01-01 00:00:00.000 11
...
X        2006-06-01 00:00:00.000 86
avg         sum
----------- -----------
45          225
Category SalesDate               Amount
-------- ----------------------- -----------
Y        2006-07-01 00:00:00.000 54
Y        2006-07-01 00:00:00.000 63
...
Y        2006-03-01 00:00:00.000 62
avg         sum
----------- -----------
61          427
Category SalesDate               Amount
-------- ----------------------- -----------
Z        2006-04-01 00:00:00.000 33
...
Z        2006-05-01 00:00:00.000 55
avg         sum
----------- -----------
53          215
```

Предложение COMPUTE BY выполняет практически ту же работу, добавляя промежуточные итоги, однако оно не выводит общий итог:

```
SELECT Category, SalesDate, Amount
  FROM RawData
  WHERE Year(SalesDate) = '2006'
  ORDER BY Category
    COMPUTE avg(Amount), sum(amount)
    COMPUTE sum(Amount)
        BY Category
```

Результат выполнения запроса следующий:

```
Category SalesDate               Amount
-------- ----------------------- -----------
X        2006-01-01 00:00:00.000 11
X        2006-06-01 00:00:00.000 86
sum
-----------
225
Category SalesDate               Amount
-------- ----------------------- -----------
Y        2006-07-01 00:00:00.000 54
...
Y        2006-03-01 00:00:00.000 62
sum
-----------
427
Category SalesDate               Amount
-------- ----------------------- -----------
Z        2006-04-01 00:00:00.000 33
...
Z        2006-05-01 00:00:00.000 55
sum
-----------
215
avg         sum
----------- -----------
54          867
```

Обратите внимание на то, что в то время как другие функции агрегирования автоматически не подставляют имя итогового столбца — его нужно идентифицировать в результирующем наборе данных как псевдоним, — функция COMPUTE автоматически подставляет имя столбца.

Создание перекрестных запросов

Хотя запросы консолидации позволяют группировать данные по нескольким столбцам, результат продолжает сохранять внешний вид списка, что делает быстрый поиск нужного значения довольно проблематичным. Перекрестная табуляция столбцов (или измерений) группировки может развернуть строку на 90° и преобразовать ее в столбец (рис. 11.4). Естественно, здесь существует ограничение: если запрос с группировкой содержит множество итоговых функций, перекрестная таблица будет отображена, но только по одному измерению.

Термин *перекрестный запрос* описывает результирующий набор данных, а не сам метод создания перекрестной таблицы. Дело в том, что существует множество программных методов генерирования перекрестных запросов; одни из них хорошие, другие, как говорится, еще лучше. В следующих разделах будет описано несколько методов получения того же результата.

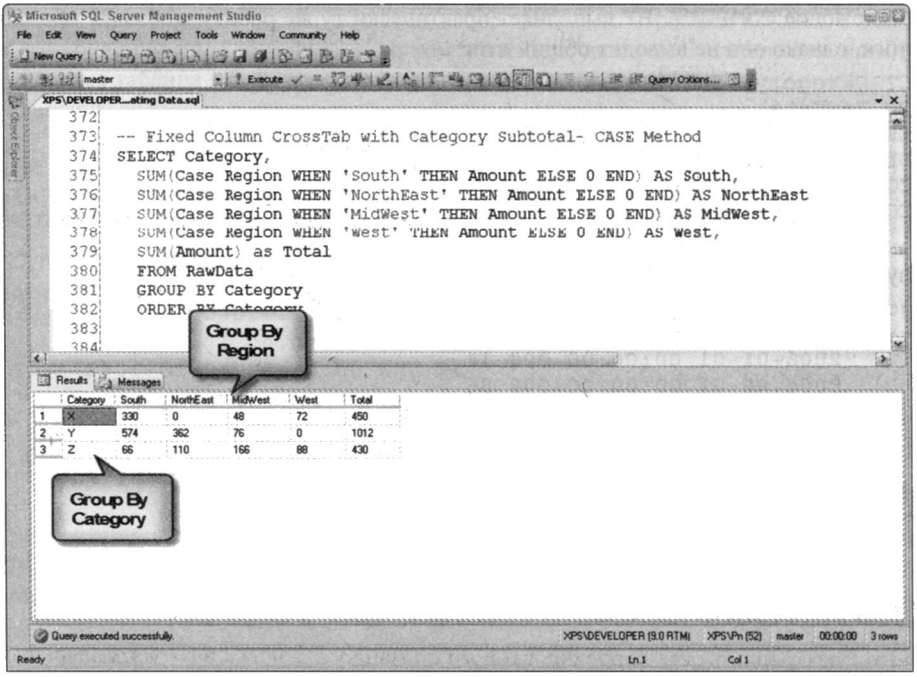


Рис. 11.4. Поворот группы в столбец создает перекрестный запрос

Перекрестные запросы с фиксированным столбцом

В SQL Server 2005 существуют три метода генерирования перекрестного запроса с известным, фиксированным столбцом. Если столбцы для любого перекрестного запроса пока невозможно узнать на стадии разработки, имеет смысл с ними определиться, поскольку программирование форм и отчетов приложений будет значительно легче при известных столбцах.

Метод коррелированных подзапросов

Первый метод T-SQL является самым распространенным, однако его производительность выглядит убогой на фоне остальных методов, поэтому я не рекомендовал бы его использовать. В то же время он полезен для понимания требований к созданию результирующего набора данных.

Основная идея состоит в выполнении подзапроса для каждого экземпляра каждого измерения всех столбцов группировки. Чтобы пройтись по этому запросу, внимательно всмотритесь в каждую строку инструкции `SELECT`. Первый столбец — это поле `category` из таблицы `RawData`. Столбцы `South`, `NorthEast`, `Midwest` и `West` представляют собой коррелированные подзапросы, суммирующие значения столбца `Amount`, отфильтрованного по соответствующей категории. Если перекрестный запрос проходит 1000 строк, а столбцы перекрестной таблицы охватывают каждую неделю года, то серверу, используя этот метод, придется выполнить 52 000 подзапросов.

Последний столбец отображает промежуточный итог для каждой категории:

```
SELECT R.Category,
      (SELECT SUM(Amount)
```

```
     FROM RawData
     WHERE Region = 'South' AND Category = R.Category) AS 'South',
      (SELECT SUM(Amount)
     FROM RawData
     WHERE Region = 'NorthEast'
         AND Category = R.Category) AS 'NorthEast',
     (SELECT SUM(Amount)
    FROM RawData
    WHERE Region = 'MidWest' AND Category = R.Category) AS 'MidWest',
    (SELECT SUM(Amount)
    FROM RawData
    WHERE Region = 'West' AND Category = R.Category) AS 'West',
     SUM(Amount) as Total
FROM RawData R
GROUP BY Category
```

Результат выполнения запроса будет следующим:

```
Category South   NorthEast   MidWest   West    Total
-------- ------- ----------- --------- ------- --------
X        165     NULL        24        36      225
Y        287     181         38        NULL    506
Z        33      55          83        44      215
```

Метод выражений CASE

Вместо фильтрации в теле коррелированного подзапроса для каждого экземпляра измерения этот метод использует выражение CASE для фильтрации данных, суммированных консолидированием. Это позволяет ядру базы данных обработать весь перекрестный запрос как одну пакетную операцию. Именно по этой причине производительность этого метода великолепна, и он остается самым популярным методом вычисления перекрестных запросов.

Рассмотрим этот метод подробно. Данные из таблицы RawData не ограничены предложением WHERE. Предложение GROUP BY разбивает набор данных на категории. А вот и сама хитрость этого метода: функция sum() включает выражение CASE; таким образом, каждый столбец видит данные только своего региона.

```
SELECT Category,
  SUM(Case Region WHEN 'South' THEN Amount ELSE 0 END) AS South,
  SUM(Case Region WHEN 'NorthEast' THEN Amount ELSE 0 END) AS   NorthEast,
  SUM(Case Region WHEN 'MidWest' THEN Amount ELSE 0 END) AS MidWest,
  SUM(Case Region WHEN 'West' THEN Amount ELSE 0 END) AS West,
  SUM(Amount) as Total
  FROM RawData
  GROUP BY Category
  ORDER BY Category
```

Результат будет таким же, как и у метода коррелированных подзапросов.

Метод поворота

Метод поворота вытекает из обычного порядка выполнения запроса. В нем вычисляются обычные итоговые функции, а результат создается как источник данных в предложении FROM.



SQL Server 2005 содержит новую команду PIVOT, предназначенную для облегчения создания перекрестных запросов. В паре с противоположной командой UNPIVOT они также облегчают нормализацию и денормализацию данных.

Если представить себе поворот как некоторую табличную функцию, используемую в качестве источника данных, то она принимает два параметра. Первым параметром является итоговая функция, используемая для значений перекрестной таблицы; второй параметр перечисляет поворачиваемые столбцы. В рассматриваемом примере итоговая функция суммирует значения столбца Amount, а поворачиваемым столбцом является region. Так как PIVOT является частью предложения FROM, создаваемому набору данных нужно присвоить именованный диапазон или псевдоним.

```
SELECT Category, SalesDate, South, NorthEast, MidWest, West
  FROM RawData
     PIVOT
     (Sum (Amount)
      FOR Region IN (South, NorthEast, MidWest, West)
     ) AS pt
```

Результат выполнения будет следующим:

```
Category SalesDate                South  NorthEast MidWest West
-------- ------------------------ ------ --------- ------- ----
Y        2005-11-01 00:00:00.000  36     NULL      NULL    NULL
Y        2005-12-01 00:00:00.000  15     28        NULL    NULL
X        2006-01-01 00:00:00.000  11     NULL      24      NULL
X        2006-02-01 00:00:00.000  NULL   NULL      NULL    36
Y        2006-02-01 00:00:00.000  47     NULL      NULL    NULL
Y        2006-03-01 00:00:00.000  NULL   62        38      NULL
Z        2006-04-01 00:00:00.000  33     NULL      83      NULL
Z        2006-05-01 00:00:00.000  NULL   55        NULL    44
X        2006-06-01 00:00:00.000  154    NULL      NULL    NULL
Y        2006-07-01 00:00:00.000  117    NULL      NULL    NULL
Y        2006-08-01 00:00:00.000  72     91        NULL    NULL
```

Результат получился не совсем тот, который мы ожидали. Дело в том, что команда PIVOT использует все столбцы. Так как столбцы Amount и Region были использованы в запросе, подразумевается, что все остальные столбцы должны использоваться для группировки. Таким образом, группировка выполнялась по полям категории товара (Category) и даты продажи (SalesDate).

Решение этой проблемы сводится к использованию подзапроса для выбора только тех столбцов, которые должны быть предоставлены команде PIVOT.

```
SELECT Category, South, NorthEast, MidWest, West
  FROM (Select Category, Region, Amount from RawData) sq
    PIVOT
      (Sum (Amount)
      FOR Region IN (South, NorthEast, MidWest, West)
      ) AS pt
```

Результат будет таким же, как и в предыдущем примере, только без лишнего столбца SalesDate.

Последним методом создания перекрестных запросов является метод шлифовки. Так как команда PIVOT является логической составляющей предложения FROM, она выполняется раньше предложения WHERE. Таким образом, существует еще одна причина использовать подзапрос для подготовки данных для команды PIVOT — фильтрация данных или объединение с другими источниками данных.

В следующем примере мы отфильтруем данные по категории Z и добавим столбец итогов по категории. В результате команда PIVOT создаст тот же результат, что и приведенный ранее метод выражения CASE.

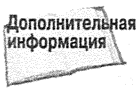
```
SELECT Category, South, NorthEast, MidWest, West,
  IsNull(South,0) + IsNull(NorthEast,0) + IsNull(MidWest,0) +
IsNull(West,0) as Total
  FROM (Select Region, Category, Amount
          From RawData
                      Where Category = 'Z') sq
    PIVOT
      (Sum (Amount)
      FOR Region IN (South, NorthEast, MidWest, West)
      ) AS pt
```

Результат выполнения запроса будет таким же, как у методов коррелированных подзапросов и выражения CASE.

Динамические перекрестные запросы

Строки перекрестного запроса динамически генерируются при консолидации, однако во всех перечисленных выше методах столбцы (в рассматриваемом примере Region) должны быть явно указаны в инструкции SQL. Единственным методом создания перекрестного запроса с динамическими столбцами является использование пакета SQL (возможно, сохраненного как хранимая процедура или пользовательская функция), определяющего столбцы на этапе выполнения и собирающего команду SQL для выполнения перекрестного запроса.

Традиционно для последовательного перемещения по данным или для сборки столбцов используют курсоры. Таким образом, динамический SQL может выполнить динамический перекрестный запрос. Использование команды PIVOT вместе с многозначной инструкцией SELECT (она будет описана немного позже в этой главе) ускоряет работу динамического перекрестного запроса.



По своей сущности куб службы анализа является динамическим перекрестным запросом на стероидах. Более подробно о создании высокопроизводительных интерактивных кубов мы поговорим в главе 43.

Курсор и метод поворота

Задачей курсора является последовательное прохождение по уникальным регионам и сборка разделенной запятыми строки в переменной @RegionColumn. Как только замыкающая запятая удаляется из переменной @RegionColumn, последняя может использоваться как часть команды PIVOT в динамической инструкции SQL, которая выполняется с помощью системной хранимой процедуры sp_executesql.

```
DECLARE
  @SQLStr NVARCHAR(1024),
  @RegionColumn VARCHAR(50),
  @SemiColon BIT
SET @Semicolon = 0
SET @SQLStr = ''
DECLARE ColNames CURSOR FAST_FORWARD
  FOR
  SELECT DISTINCT Region as [Column]
    FROM RawData
    ORDER BY Region
OPEN ColNames
FETCH ColNames INTO @RegionColumn
WHILE @@Fetch_Status = 0
  BEGIN
```

```
    SET @SQLStr = @SQLStr + @RegionColumn + ', '
    FETCH ColNames INTO @RegionColumn -- fetch next
  END
CLOSE ColNames
DEALLOCATE ColNames
SET @SQLStr = Left(@SQLStr, Len(@SQLStr) - 1)
SET @SQLStr = 'SELECT Category, '
            + @SQLStr
            + ' FROM RawData PIVOT (Sum (Amount) FOR Region IN ('
            + @SQLStr
            + ')) AS pt'
PRINT @SQLStr
EXEC sp_executesql @SQLStr
```

Результат выполнения этого пакета будет таким же, как и предыдущих примерах, однако если появятся дополнительные регионы, то в перекрестном запросе в качестве новых столбцов.



Так как курсоры плохо масштабируются, их использование не считается хорошей практикой, и задача создания перекрестного запроса не является исключением. Согласно многочисленным тестам, метод курсора выполняется более чем в десять раз медленнее, чем предлагаемый далее пакетный метод многочисленного присвоения переменной. Подробную информацию о программировании курсора вы можете получить в главе 20.

Переменная с многочисленными присвоениями и метод поворота

Этот метод использует точно такую же динамическую строку SQL для выполнения динамического перекрестного запроса, как и метод курсора, однако для создания списка регионов он задействует пакетную инструкцию SELECT. Инструкция SELECT добавляет каждый из регионов в список, разделенный запятыми, который затем выполняется так же, как и в предыдущем примере:

```
DECLARE @XColumns NVARCHAR(1024)
SET @XColumns = ''
SELECT @XColumns = @XColumns + [a].[Column] + ', '
  FROM
    (SELECT DISTINCT Region as [Column]
      FROM RawData) as a
SET @XColumns = Left(@XColumns, Len(@XColumns) - 1)
SET @XColumns = 'SELECT Category, '
    + @Xcolumns
    + ' FROM RawData PIVOT (Sum (Amount) FOR Region IN ('
    + @Xcolumns
    + ')) AS pt'
PRINT @Xcolumns
EXEC sp_executesql @XColumns
```

Задачей программы динамического запроса является сборка фиксированного перекрестного запроса без предварительного указания значений столбцов. Список столбцов возвращает подзапрос, после чего рекурсивная переменная SELECT добавляет значения, а также текст, необходимый для создания динамического перекрестного запроса в переменной @XColumns. Заключительная инструкция SET завершает сборку динамической строки SQL.



Противоположностью перекрестному запросу является команда UNPIVOT, которая исключительно полезна для нормализации денормализованных данных. Описание этой команды и примеры вы найдете в главе 24.

Резюме

SQL Server выделяется многообразием итоговых функций с уже ставшим притчей во языцех богатым набором средств. Он способен вычислять суммы и прочие консолидации, с лихвой покрывающие любые мыслимые потребности. Эти методы, от простой итоговой функции `count()` и до сложных динамических перекрестных запросов, а также новой команды `PIVOT`, позволяют создавать мощные запросы анализа данных для впечатляющих отчетов.

В следующей главе мы исследуем иерархические данные, продолжив тему использования инструкции `SELECT` для выполнения все более сложных задач.

ГЛАВА

12

Навигация по иерархическим данным

В этой главе...

Хранение иерархических данных

Иерархические пользовательские функции

Рекурсивные запросы CTE

Ни один инструмент не является уникальным для любой работы, и, несмотря на мою веру в то, что язык SQL является идеальным для работы с данными, ему иногда тяжело справляться с некоторыми типами данных. Откровенно говоря, SQL довольно неуклюже работает с различными уровнями иерархических данных. Отсутствие элегантного решения становится очевидным при работе с генеалогическими деревьями, списками материалов, организационными диаграммами, уровнями юрисдикции и моделированием наследования.

Среди проблем, окружающих иерархические данные, — моделирование данных, навигация по дереву, выбор различных поколений предков и наследников, а также перемещение фрагментов дерева в другое место. Если задача при этом требует создания отношения "многие ко многим" (как, например, в списке материалов, когда один материал создается из множества других, но и сам может входить в качестве компонента в другие материалы), то она становится еще более сложной.

В настоящей главе мы рассмотрим различные методы хранения иерархических данных и навигации по ним: от одноуровневого возвратного (рекурсивного) объединения до громоздкого курсора и эффективных пользовательских функций.



Стандарт ANSI SQL-99 (ссылка на который содержится на сайте этой книги по адресу `www.SQLServerBible.com`) попытался справиться с иерархическими данными с помощью введения общих табличных выражений (CTE), которые можно использовать для выборки нескольких уровней данных из иерархического дерева. Со временем, возможно, я изменю свое мнение, но этот новый синтаксис даже не похож на SQL, и я остаюсь приверженцем пользовательских функций навигации.

Шаблон смежных списков

Изначальная проблема, возникающая при работе с иерархическими данными, заключается в методологии их хранения, поскольку иерархию нельзя назвать естественной схемой для реляционной модели. Существуют два основных метода: смежный список и материализованный путь.

Основные шаблоны смежных списков

До сих пор самой распространенной схемой хранения иерархических данных остаются *шаблоны смежных списков*, неформально называемые *парами данных* и *моделями с собственным объединением*. Они хранят ключ текущего узла и его непосредственного предка в строке текущего узла (в этой главе мы будем обращаться к элементам пары данных как к *текущему* и *родительскому* узлам).

Эта схема хорошо известна вам по таблице Employee учебной базы данных Northwind. В ней в полях `EmployeeID` и `ReportsTo` хранятся соответственно идентификаторы сотрудника и его непосредственного начальника (рис. 12.1).

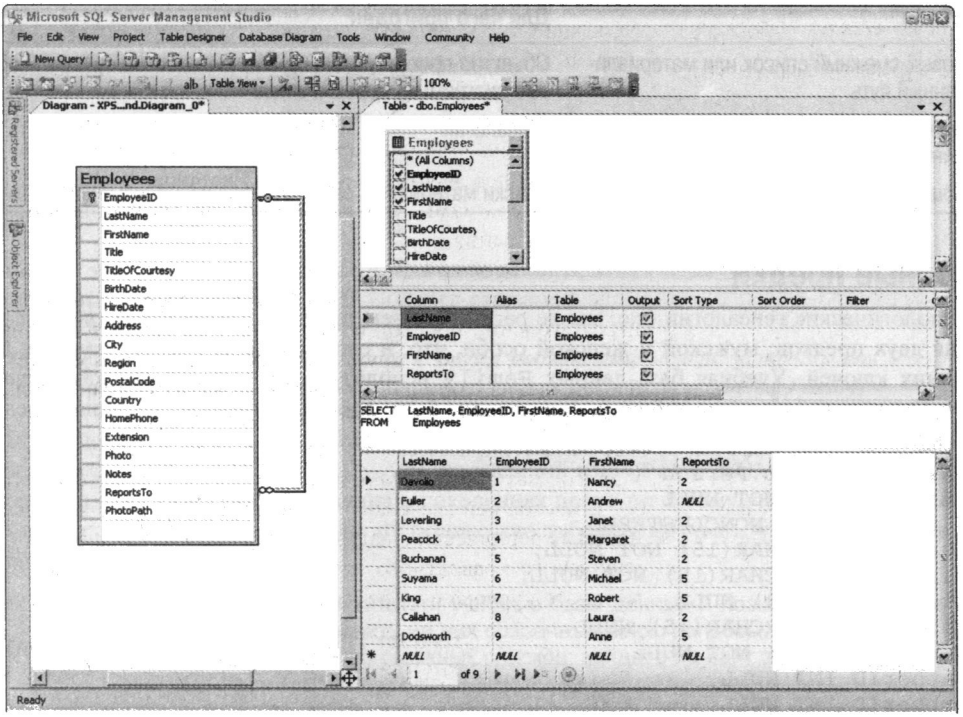


Рис. 12.1. Таблица `Employee` базы данных Northwind является прекрасным примером использования схемы шаблона смежных списков для хранения организационной диаграммы

Между работником, выступающим в роли начальника, и сотрудником, который перед ним отчитывается, существует отношение "один ко многим". Начальник может иметь много подчиненных, однако каждый работник имеет только одного непосредственного начальника. Любой работник может быть одновременно и начальником и подчиненным.

Столбец ReportsTo является внешним ключом таблицы и ссылается на столбец EmployeeID той же таблицы. В столбце ReportsTo хранится идентификатор EmployeeID непосредственного начальника. Если узел работника является текущим, то столбец ReportsTo указывает на родительский узел. Этот столбец может содержать пустое значение, поскольку глава всей организации ни перед кем не отчитывается.

С точки зрения начальника, его идентификатор хранится в поле ReportsTo всех его подчиненных.

Вариации смежных списков

Базовая схема шаблона смежных списков оказывается полезной в ситуациях, когда существует только отношение "один родитель–множество узлов", однако этого недостаточно для серьезных производственных иерархий. К счастью, базовый шаблон пары данных можно легко видоизменить для работы со сложными иерархиями. В табл. 12.1 представлены три вариации схемы смежных списков и области их применения.

Таблица 12.1. Иерархические схемы

Шаблон	Для чего пригоден
Базовый смежный список или материализованный путь	Объектно-ориентированные классы, простые организационные диаграммы, деревья видов
Смежный список с двумя родителями или двойной материализованный путь	Генеалогии
Ассоциативные таблицы	Списки материалов, сложные организационные диаграммы

Двойные предки

Биологические генеалогии (т.е. мы не рассматриваем приемных родителей) требуют хранения двух предков: мужской и женской особи, что, в свою очередь, требует наличия двух внешних ключей. Учебная база данных Family использует эту модель. Так как внешние ключи ссылаются на ту же таблицу, они должны быть добавлены при ее создании. Следующий код извлечен из сценария создания базы данных Family:

```
CREATE TABLE dbo.Person (
  PersonID INT NOT NULL
    PRIMARY KEY NONCLUSTERED,
  LastName VARCHAR(15) NOT NULL,
  FirstName VARCHAR(15) NOT NULL,
  SrJr VARCHAR(3) NULL,
  MaidenName VARCHAR(15) NULL,
  Gender CHAR(1) NOT NULL,
  FatherID INT NULL,
  MotherID INT NULL,
  DateOfBirth DATETIME NULL,
  DateOfDeath DATETIME NULL
);
go
CREATE CLUSTERED INDEX IxPersonName
  ON dbo.Person (LastName, FirstName);
ALTER TABLE dbo.Person ADD CONSTRAINT
    FK_Person_Father FOREIGN KEY (FatherID) REFERENCES dbo.Person
(PersonID);
```

```
ALTER TABLE dbo.Person ADD CONSTRAINT
    FK_Person_Mother FOREIGN KEY (MotherID) REFERENCES dbo.Person
(PersonID);
```

В базе данных `Family` каждая запись ссылается на два родительских узла: биологической матери и биологического отца. Оба внешних ключа указывают на один и тот же первичный ключ (рис. 12.2).

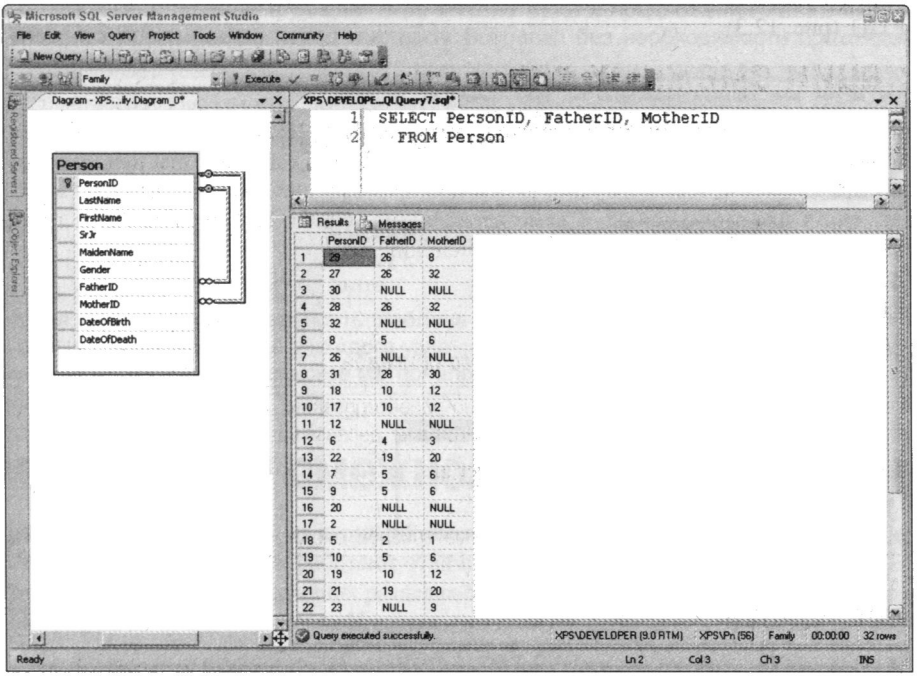


Рис. 12.2. Шаблон смежного списка с двумя родителями можно использовать для хранения генеалогий (в частности, он использован в учебной базе данных Family)

Можно доказать, что данный шаблон нарушает первую нормальную форму, поскольку оба столбца — `MotherID` и `FatherID` — хранят внешний ключ `PersonID`. Однако отношения к матери и к отцу уникальны, и разница в поколениях приводит к тому, что каждый внешний ключ реально указывает на отличное подмножество таблицы `Person`, что может и должно поддерживаться триггером.



Если вы хотите загрузить эти учебные базы данных и поэкспериментировать с ними, а также опробовать на них созданные мною запросы, использующие разные вариации шаблона смежных списков, зайдите на сайт этой книги по адресу `www.SQLServerBible.com`.

Ассоциативные таблицы

При наличии отношений "многие ко многим" между текущими и родительскими узлами требуется наличие ассоциативной таблицы (аналогичным образом реализуются отношения "многие ко многим" и в любой другой модели). Например, заказ может содержать множество товаров, а любой товар может входить во множество заказов. Поэтому таблица строк заказа выступает в роли ассоциативной между таблицами заказов и товаров.

Аналогичная ситуация возникает на производстве при разработке схем для списков материалов. Например, компонент `a23` может использоваться при производстве множества других компонентов, будучи при этом сам изготовлен их нескольких других составляющих. В данном контексте любой компонент одновременно является и дочерним, и родительским для других компонентов.

В иерархическом списке материалов сам этот список выступает в роли смежной таблицы между текущим и родительскими узлами, которые хранятся в одной и той же таблице `Material` (рис. 12.3).

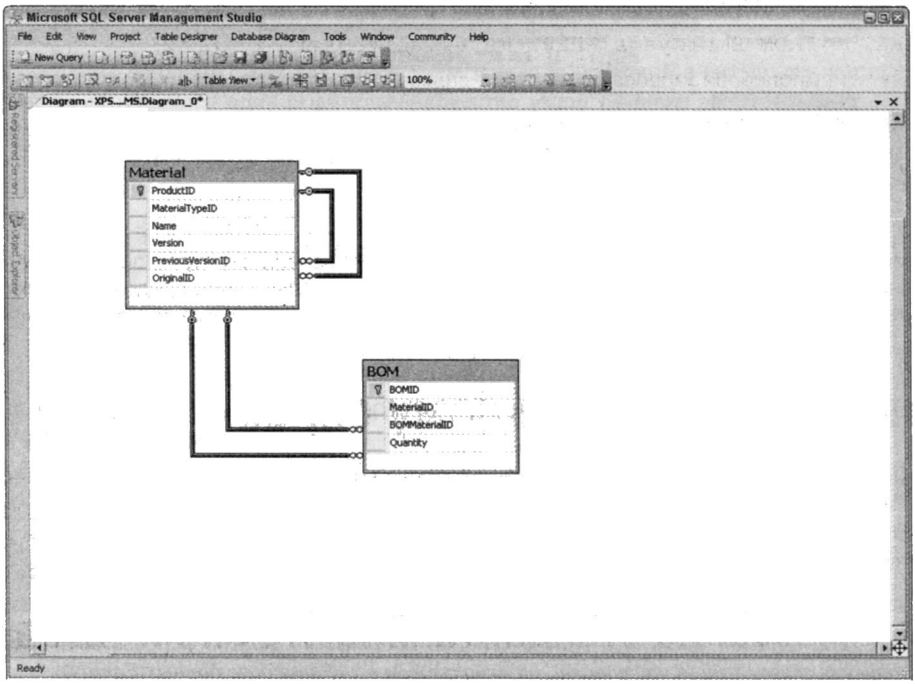


Рис. 12.3. Схема списка материалов использует ассоциативную таблицу, хранящую информацию о том, из каких составляющих создается каждый компонент

Шаблон материализованного пути

Шаблон материализованного пути представляет собой еще один прекрасный метод хранения иерархических данных и навигации по ним. В своей основе он хранит денормализованный список всех предков текущего узла, включая все их поколения. Например, в хорошо знакомой таблице `Employee` базы данных `Northwind` столбец `MaterializedPath` хранит путь в организационной диаграмме от корневого до текущего узла. Этот столбец был добавлен в таблицу исключительно с целью демонстрации модели материализованного пути — само это решение не требует наличия столбца `ReportsTo`.

```
EmployeeID  LastName     FirstName  ReportsTo   MaterializedPath
----------- ------------ ---------- ----------- ----------------
1           Davolio      Nancy      2           21
2           Fuller       Andrew     NULL        2
3           Leverling    Janet      2           23
4           Peacock      Margaret   2           24
```

```
5           Buchanan     Steven     2           25
6           Suyama       Michael    5           256
7           King         Robert     5           257
8           Callahan     Laura      2           28
9           Dodsworth    Anne       5           259
```

Шаблон материализованного пути имеет свои достоинства и недостатки. Положительным является то, что существует несколько хороших методов манипуляции иерархией и ее анализа с помощью строковых функций. Ограничение `LIKE` в предложении `WHERE` быстро найдет всех, кто отчитывается перед мистером Buchanan без необходимости прохождения по всем уровням иерархии.

```
SELECT * FROM Employees WHERE MaterializedPath Like '25_'
```

Результат выполнения запроса будет следующим:

```
EmployeeID  LastName     FirstName  ReportsTo   MaterializedPath
----------- ------------ ---------- ----------- ----------------
6           Suyama       Michael    5           256
7           King         Robert     5           257
9           Dodsworth    Anne       5           259
```

Несмотря на всю привлекательность шаблона материализованного пути, он ограничен только базовыми иерархиями, где у каждого узла существует только один предок. Это исключает реализацию с его помощью списков материалов и сложных организационных диаграмм.

Навигация по смежному списку

Хранение данных в любой из вариаций смежного списка не вызывает проблем — обычные вставка, обновление и удаление работают отлично. Вся хитрость состоит в извлечении данных и навигации по иерархии. В следующих разделах будут описаны некоторые решения этих задач — от тривиальных до крайне сложных. Надеюсь, что хотя бы одно из них вам понравится.

Использование стандартной инструкции `SELECT`

Генерация списка результатов с помощью всего одной инструкции `SELECT` проблематична, поскольку количество поколений динамично, а запрос выборки требует известного набора таблиц. Инструкция `SELECT` может справиться с известным количеством поколений, но когда оно не известно, заранее созданная инструкция не справится с задачей. В качестве примера приведем запрос, извлекающий данные о дедушке и двух последующих поколениях:

```
USE Family
SELECT
    Person.FirstName + ' ' + IsNull(Person.SrJr,'')
     as Grandfather,
    Gen1.FirstName + ' ' + IsNull(Gen1.SrJr,'') as Gen1,
    Gen2.FirstName + ' ' + IsNull(Gen2.SrJr,'') as Gen2
  FROM Person
      LEFT JOIN Person Gen1
          ON Person.PersonID = Gen1.FatherID
   LEFT JOIN Person Gen2
         ON Gen1.PersonID = Gen2.FatherID
  WHERE Person.PersonID = 2
```

Результат будет следующим:

```
Grandfather Gen1       Gen2
----------- ---------- -------------
```

```
James 1      James 2     Melanie
James 1      James 2     Corwin
James 1      James 2     Dara
James 1      James 2     James 3
```

В качестве альтернативы аналогичный запрос может извлекать данные обо всех родителях и всех их детях, объединяя два экземпляра таблицы `Person`. Однако этот метод не позволяет создать работоспособное дерево.

Использование рекурсивного курсора

Наиболее распространенным программным решением является навигация по иерархии с помощью итеративного процесса, что подразумевает использование курсора, устрашающего всех.



Подробную информацию о программировании курсоров (и, что самое главное, о том, как их избегать) вы найдете в главе 20.

Для создания дерева курсор проверяет наличие детей у конкретного родителя и выводит их на печать со смещением. Эту работу курсор выполняет, извлекая лиц, у которых идентификаторы матери или отца совпадают с текущей персоной. Как только курсор будет объявлен и открыт, каждая инструкция `FETCH` будет выводить на печать детей и рекурсивно вызывать следующий экземпляр процедуры для определения того, имеют ли эти дети своих детей. Если это так, будут проверены эти дети, и т.д.

Для каждого человека вызывается рекурсивная процедура, чтобы выявить наличие у него детей.

Рекурсивная природа этой процедуры заставляет ее последовательно спускаться вниз по дереву, находя первого ребенка (PK: “5”, “8”, “15”), а затем его братьев и сестер (“16”, “29”). Результаты этой процедуры вы увидите в следующем примере. Затем рекурсивная процедура начинает искать братьев ребенка “8” и находит номер “10”. После этого проверяется наличие у “10“ детей, и первым найденным ребенком будет “19”. Затем процедура вызывается для “19”, в результате чего возвращаются “21” и “22”. И наконец процедура вызывается для “21” и “22”, но детей у них не находит.

По умолчанию область видимости курсора распространяется на все вызываемые процедуры, однако задача построения иерархического дерева требует, чтобы каждая процедура создавала собственный курсор. Установка параметра `cursor` в значение `local` ограничивает область видимости курсора и разрешает рекурсию. Этот параметр может быть задан при объявлении курсора или в параметрах базы данных. В следующем примере продемонстрированы оба этих метода:

```
ALTER DATABASE Family SET CURSOR_DEFAULT LOCAL
SELECT DATABASEPROPERTYEX('Family', 'IsLocalCursorsDefault')
```

Результат будет следующим:

```
1
```

Следующий пакет создает процедуру `ExemineChild`, содержащую курсор, который проверяет наличие детей для текущей строки таблицы `Person`. Если дети будут обнаружены, то эта хранимая процедура вызывает себя рекурсивно:

```
CREATE PROCEDURE ExamineChild
    (@ParentID INT)
AS
SET Nocount On
DECLARE @ChildID INT,
  @Childname VARCHAR(25)
```

```
DECLARE cChild CURSOR LOCAL FAST_FORWARD
  FOR SELECT PersonID,
           Firstname + ' ' + LastName + ' ' + IsNull(SrJr,'')
           as PersonName
      FROM Person
      WHERE Person.FatherID = @ParentID
        OR Person.MotherID = @ParentID
      ORDER BY Person.DateOfBirth
  OPEN cChild
  FETCH cChild INTO @ChildID, @ChildName -- prime the cursor
  WHILE @@Fetch_Status = 0
    BEGIN
      PRINT
        SPACE(  tLevel * 2) + '+ '
          + Cast(@ChildID as VARCHAR(4))
          + ' ' + @ChildName
 -- Рекурсивно ищем внуков
      EXEC ExamineChild @ChildID
      FETCH cChild INTO @ChildID, @ChildName
    END
  CLOSE cChild
DEALLOCATE cChild
```

Вызывается рекурсивная хранимая процедура `ExemineChild`, и ей передается идентификатор человека, находящегося на первом уровне родословного дерева, — Джеймса Халлоуэя (идентификатор равен двум).

```
EXEC ExamineChild 2
```

Будет получен следующий результат:

```
+ 5 James Halloway 2
  + 8 Melanie Campbell
    + 15 Shannon Ramsey
    + 16 Jennifer Ramsey
    + 29 Adam Campbell
  + 10 James Halloway 3
    + 19 James Halloway 4
      + 22 Chris Halloway
      + 21 James Halloway 5
  + 18 Abbie Halloway
  + 17 Allie Halloway
+ 9 Dara Halloway
  + 23 Joshua Halloway
  + 24 Laura Halloway
+ 7 Corwin Halloway
  + 14 Logan Halloway
```

Использование курсора является адекватным решением задачи построения иерархического дерева, когда набор данных невелик, однако для больших массивов информации этот метод не подходит, и на то есть две причины. Во-первых, SQL Server ограничивает вложенность хранимых процедур 32 уровнями, так что рекурсии большей глубины (за вычетом вложенности кода, вызвавшего рекурсивную процедуру) не будут выполнены. Вторая причина связана с вопросом производительности (этот вопрос всегда встает при использовании курсоров). Так как для каждого узла требуется итерация, производительность падает в буквальном смысле этого слова. Семиуровневая иерархия с пятью миллионами узлов потребует пять миллионов итераций. Пакетное решение сможет выполнить ту же задачу за семь проходов.

Использование пакетных решений

Первое из трех пакетных решений начинает с одного человека, принимая в расчет временную таблицу #FamilyTree. Затем пакет пошагово проходит по всем поколениям во временной таблице, используя объединение с несколькими условиями.

Для каждого лица во временной таблице #FamilyTree столбец FamilyLine содержит данные FamilyLine предков, объединенные с идентификатором родителей (PersonID). Столбец FamilyLine содержит данные, необходимые для сортировки дерева.

Если не найден ни один человек, то условие WHILE больше не удовлетворяется и пакет завершает выполнение. Ниже приведен пример пакетного кода, который демонстрирует, что метод курсора действительно устарел для решения рекурсивных задач навигации по дереву.

```
CREATE TABLE #FamilyTree (
    PersonID INT,
    Generation INT,
    FamilyLine VarChar(25) Default ''
)
DECLARE
  @Generation INT,
  @FirstPerson INT
SET @Generation = 1
SET @FirstPerson = 2
-- подготавливаем временную таблицу с главной персоной во главе
INSERT #FamilyTree (PersonID, Generation, FamilyLine)
  SELECT @FirstPerson, @Generation, @FirstPerson
 WHILE @@RowCount > 0
  BEGIN
    SET @Generation = @Generation + 1
    INSERT #FamilyTree (PersonID, Generation, FamilyLine)
      SELECT Person.PersonID,
           @Generation,
                       #FamilyTree.FamilyLine
                       + ' ' + Str(Person.PersonID,5)
        FROM Person
           JOIN #FamilyTree
             ON #FamilyTree.Generation = @Generation - 1
                 AND
                (Person.MotherID = #FamilyTree.PersonID
                 OR
                Person.FatherID = #FamilyTree.PersonID)
END
```

Когда временная таблица #FamilyTree заполнена, следующий запрос проверяет оперативные данные.

Результат будет следующим (сокращенно):

```
PersonID    Generation  FamilyLine
----------- ----------- -------------------------
2           1           2
5           2           2 5
7           3           2 5 7
14          4           2 5 7 14
...
22          5           2 5 10 19 22
```

Аналогично предыдущему решению с курсором, следующий запрос использует ту же функцию space() для форматирования результатов, после чего результаты объединяются с таблицей Person для отображения имен:

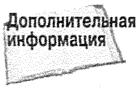
```
SELECT SPACE(Generation * 2) + '+ '
        + Cast(#FamilyTree.PersonID as VARCHAR(4)) + ' '
        + FirstName + ' ' + LastName
        + IsNull(SrJr,'') AS FamilyTree
  FROM #FamilyTree
  JOIN Person
    ON #FamilyTree.PersonID = Person.PersonID
  ORDER BY FamilyLine
```

Будет получен следующий результат:

```
FamilyTree
--------------------------------------------------
+ 2 James Halloway 1
  + 5 James Halloway 2
    + 7 Corwin Halloway
      + 14 Logan Halloway
    + 8 Melanie Campbell
      + 15 Shannon Ramsey
      + 16 Jennifer Ramsey
      + 29 Adam Campbell
    + 9 Dara Halloway
      + 23 Joshua Halloway
      + 24 Laura Halloway
    + 10 James Halloway 3
      + 17 Allie Halloway
      + 18 Abbie Halloway
      + 19 James Halloway 4
        + 21 James Halloway 5
        + 22 Chris Halloway
```

Использование пользовательских функций

Несмотря на то что пакетные решения демонстрируют саму методику, внедрение их в пользовательские функции значительно повышает их способность использования в различных программах.



Этот метод заставляет нас заглянуть немного вперед в расширенные средства программирования SQL Server для решения иерархических задач. Подробную информацию о моих любимых средствах SQL Server вы найдете в главе 22.

Основное достоинство табличных пользовательских функций заключается в том, что они возвращают набор данных, который может быть использован в качестве источника данных в любом другом обычном запросе. Создание пользовательской функции, которая может проходить по иерархическому дереву и возвращать результат в виде таблицы, предполагает возможность ее объединения с любой другой таблицей.

Используя базу данных Family, данная пользовательская функция способна вернуть наследников любого лица в отдельном наборе данных. Полученный набор данных впоследствии можно объединить с таблицей Person для получения более подробной информации об этих людях:

```
CREATE FUNCTION dbo.FamilyTree
  PersonID CHAR(25))
  TURNS @Tree TABLE (PersonID INT, LastName VARCHAR(25), FirstName
```

```
VARCHAR(25), Lv INT)
AS
BEGIN
  DECLARE @LC INT
  SET @LC = 1
  -- вставка начального уровня
  INSERT @Tree
    SELECT PersonID, LastName, FirstName, @LC
      FROM dbo.Person with (NoLock)
      WHERE PersonID = @PersonID
      -- Цикл по подуровням
      WHILE @@RowCount > 0
         BEGIN
            SET @LC = @LC + 1
            -- Вставка всех поколений
            INSERT @Tree
              SELECT Tree.PersonID, Tree.LastName,
                  Tree.FirstName, @LC
                FROM dbo.Person FamilyNode with (NoLock)
                JOIN dbo.Person Tree with (NoLock)
                  ON FamilyNode.PersonID = Tree.MotherID
                     OR FamilyNode.PersonID = Tree.FatherID
                JOIN @Tree CC
                  ON CC.PersonID = FamilyNode.PersonID
               WHERE CC.Lv = @LC - 1
         END
       RETURN
     END
```

Если посмотреть на текст функции, то можно увидеть, что в качестве аргумента она принимает идентификатор лица `PersonID`. Эта персона будет служить корнем для начала поиска в иерархии. Конечным результатом функции является список всех потомков в иерархическом порядке, начиная с корня дерева. Все данные хранятся в табличной переменной `@Tree`, которая передается назад в запрос, вызвавший функцию. Этот запрос может использовать результат в качестве источника данных в предложении `FROM`.

В первую очередь мы ищем в функции идентификатор класса `ClassID` и добавляем его в табличную переменную `@Classes`. После этого функция проходит по всем уровням иерархии, пока не достигнет уровня, в котором потомков уже нет. Эта пакетная пользовательская функция обладает высокой производительностью и масштабируемостью.

```
Select * From dbo.FamilyTree(10);
```

Результат будет следующий.

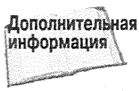
```
PersonID    LastName    FirstName   Lv
----------- ----------- ----------- ----
10          Halloway    James       1
18          Halloway    Abbie       2
17          Halloway    Allie       2
19          Halloway    James       2
22          Halloway    Chris       3
21          Halloway    James       3
```



Этот метод я широко использовал в своих объектно-реляционных экспериментах. Предлагаю вам загрузить последнюю версию этих экспериментов с сайта `www.SQLServerBible.com`, опробовать их и убедиться в том, как иерархические пользовательские функции решают задачи наследования классов.

Использование рекурсивных общих табличных выражений

Последним решением задачи навигации в иерархии смежных пар являются рекурсивные общие табличные выражения (CTE). Следуя той же базовой логике, что и в тексте программы пользовательской функции, CTE изначально устанавливает якорь, который затем итеративно объединяется с идентифицируемыми строками до тех пор, пока ни одной новой строки не будет найдено.



Об основах создания запросов, использующих общие табличные выражения, см. в главе 10.

Основным преимуществом рекурсивных CTE является то, что они не используют процедурный код. Вместо этого они используют совершенно новый синтаксис CTE. Рекурсивные CTE используют два запроса, связанных вместе. Первый запрос служит для выбора якоря. Второй запрос объединяет его с общим табличным выражением для выбора следующего поколения в иерархии. SQL Server продолжает выполнять второй запрос до тех пор, пока не будут обнаружены новые строки. Здесь используется та же логика, что и в пользовательских функциях:

```
WITH FamilyTree( LastName, FirstName, PersonID, lv)
AS (
    -- Якорь
    SELECT LastName, FirstName, PersonID, 1
      FROM Person A
      WHERE PersonID = 10
  -- Рекурсивный вызов
  UNION ALL
    SELECT Node.LastName, Node.FirstName, Node.PersonID, lv + 1
      FROM Person Node
       JOIN FamilyTree ft
         ON Node.MotherID = ft.PersonID
            OR Node.FatherID = ft.PersonID
    )
SELECT PersonID, LastName, FirstName, lv
FROM FamilyTree;
```

В данном случае результат рекурсивного общего табличного выражения будет идентичен результату приведенной ранее пользовательской функции.



Ознакомиться с рекурсивным CTE можно на сайте `www.SQLServerBible.com`.

Из всех приведенных в настоящей главе решений задачи навигации в иерархических структурах самым быстрым, пожалуй, является пользовательский запрос. Он выполняется вдвое быстрее, чем общее табличное выражение, и в зависимости от конкретного набора данных на порядок быстрее, чем курсор.

Резюме

Иерархические данные могут вызвать определенные затруднения, если вы не вооружены моделями и решениями. Однако, зная возможные шаблоны и имея в запасе несколько методов навигации, вы повышаете вероятность того, что ваш следующий проект в области иерархических данных будет успешным.

В следующей главе мы продолжим тему выборки данных, изучив средства полнотекстового поиска.

Использование полнотекстового поиска

ГЛАВА

13

В этой главе...

Настройка каталога индексов полнотекстового поиска в Management Studio и с помощью SQL

Поддержка полнотекстовых индексов

Использование полнотекстовых индексов в запросах

Выполнение нечеткого поиска

Поиск текста, хранимого в двоичных объектах

Производительность полнотекстового поиска

Несколько лет назад я написал программу поиска слов в крупной базе данных. Программа сканировала все документы и создавала таблицу частоты слов как ассоциацию "многие ко многим" между ней и таблицей документов. Программа работала достаточно хорошо, и поиск выполнялся довольно быстро. Я получил удовольствие от выполненной работы, но вам предоставляются более благоприятные условия.

Серверная версия операционной системы Windows содержит структурированную систему индексации слов и фраз, называемую MS Search. Это не просто обычное сканирование слов — механизм полнотекстового поиска осуществляет лингвистический анализ, отделяя корни слов и спрягая глаголы на разных языках. SQL Server взял на вооружение этот механизм для полнотекстового поиска в строках и столбцах.

Стандарт ANSI SQL использует оператор `like` для поиска слов, позволяя использовать символы макроподстановки. Следующий фрагмент программы выполняет поиск в учебной базе данных `Fables`, содержащей басни Эзопа:

```
-- SQL Where Like
SELECT Title
FROM Fable
WHERE Fabletext LIKE '%lion%'
AND Fabletext LIKE '%bold%'
```

Результат выполнения инструкции следующий:

```
Title
-------------------------------------------
The Hunter and the Woodman
```



Во всех примерах, которые приводятся в настоящей главе, используется база данных `Fables`. Сценарий `Aesop_Create.sql` создает эту базу данных и наполняет ее двадцатью пятью баснями Эзопа. Все примеры программ, приведенные в настоящей главе, содержатся в файле `Ch13.sql`.

Основная проблема использования выражения `where...like` заключается в низкой производительности. Индексы создаются по первым буквам слова, поэтому оператор `like 'word%'` выполнится достаточно быстро, но в то же время производительность оператора `like '%word%'` оставляет желать лучшего. Поиск подстрок в строках не может использовать структуру двоичного дерева для ускорения работы, поэтому приходится прибегать к сканированию таблиц (рис. 8.1). Это все равно что искать всех Иванов в телефонной книге. Телефонные книги не упорядочены по именам, поэтому данная задача потребует просмотра абсолютно всех страниц.

Принимая это во внимание, компания Microsoft в конце 1998 года переработала свои поисковые механизмы для версии SQL Server 7. Новый механизм получил название MSSearch и был также использован для осуществления поиска в программах Exchange и SharePoint. Совершенно отдельная версия этого механизма была подготовлена к выходу операционной системы SQL Server 2000.

Полнотекстовый поиск в версии SQL Server 2005 представляет уже третье поколение поисковых механизмов и является наиболее полнофункциональным и масштабируемым. Служба полнотекстового поиска входит в состав редакций Standard, Workgroup и Enterprise.

Поисковый механизм, поставляемый с SQL Server 2005, носит название MSFTESQL; по умолчанию он отключен. Для включения этой службы нужно запустить утилиту SQL Server Surface Area Configuration Tool, развернуть пункт Surface Area Configuration for Services and Components, раскрыть свой экземпляр сервера и найти узел Full-Text Search. Затем следует установить переключатель для этого узла в положение run automatically.

Механизм полнотекстового поиска создает индексы для всех существенных слов и фраз, решая тем самым проблему производительности. К тому же механизм полнотекстового поиска обладает рядом дополнительных функций, в частности следующими.

- Поиск одного слова, находящегося рядом с другим.
- Использование в поисковой фразе символов макроподстановки.
- Поиск различных словоформ (например, `ran`, `run` и `running`).
- Взвешивание слов и фраз, которые более существенны, чем остальные.
- Выполнение нечеткого поиска слов и фраз.
- Поиск символьных данных с внедренными двоичными объектами.



Наиболее существенным улучшением в версии SQL Server 2005 является повышение производительности практически в 10–100 раз по сравнению с полнотекстовым поиском версии SQL Server 2000. Существуют и другие улучшения, например доверительные фильтры, улучшенная поддержка языков, нечувствительность к ударениям, поддержка тезауруса, многостолбцовых запросов, связанных серверов, репликации, резервирования баз данных, подключения и отключения каталогов, индексации свойств документа и устранения ошибок.

Новый механизм поиска MSFTESQL (аббревиатура от Microsoft Full-Text Engine for SQL Server), входящий в состав SQL Server 2005, был полностью переработан для повышения производительности и теперь полностью интегрирован в сервер.

Конфигурирование каталогов полнотекстового поиска

Каталог полнотекстового поиска представляет собой набор индексов для одной базы данных SQL Server. Каждый каталог может хранить индексы для нескольких таблиц, но каждая таблица привязана только к одному каталогу. Обычно один каталог обслуживает все полно-

текстовые индексы базы данных, хотя с целью повышения производительности для особо крупных таблиц (с миллионами строк) могут выделяться отдельные каталоги.

Каталоги могут индексировать только пользовательские таблицы (не представления, не временные таблицы, не табличные переменные и не системные таблицы).

Создание каталога с помощью мастера

Хотя создание и конфигурирование каталогов полнотекстового поиска относительно не сложно выполнить и программным путем, эта задача выполняется один раз, и для нее вполне достаточно вызова специального мастера (если сценарий не требует многочисленных запусков).

Этот мастер запускается в окне Object Explorer утилиты Management Studio. Для этого нужно выделить соответствующую таблицу и в ее контекстном меню выбрать команду Full-Text Index⇨Define Full-Text Index.

При конфигурировании каталога полнотекстового поиска мастер проходит несколько этапов.

1. Выберите базу данных, которой будет принадлежать каталог.
2. Выделите таблицу для включения в каталог.
3. Задайте уникальный индекс, который будет использовать механизм полнотекстового поиска. Как правило, для такого индекса лучше использовать первичный ключ, однако иногда оказывается достаточно и индекса по одному столбцу, если он не содержит пустых и дублирующихся значений. Если таблица использует составной первичный ключ, то для полнотекстового поиска можно использовать другой уникальный индекс.
4. Выделите столбцы, которые будут использоваться для индекса (рис. 13.1), при этом можно использовать данные любых символьных типов (`char`, `nchar`, `varchar`, `nvarchar`, `text`, `ntext` и `xml`), а также двоичные изображения (`image`) (об индексировании двоичных данных речь пойдет в последнем разделе этой главы). Также нужно выбрать язык для сканирования слов, хотя компьютер по умолчанию может автоматически справиться и с этой задачей.

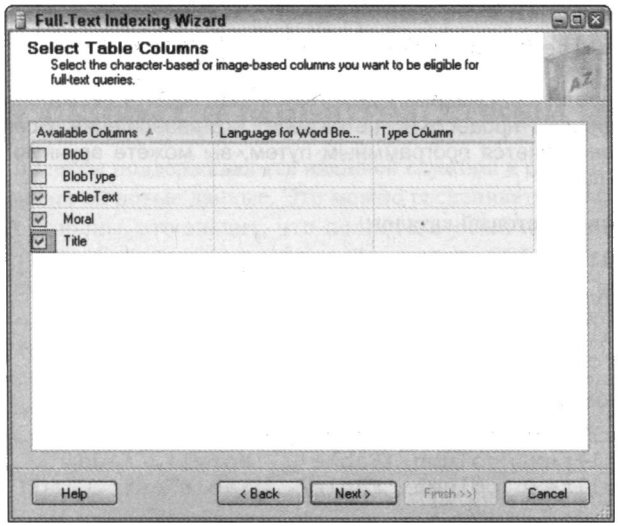


Рис. 13.1. Для индексации может быть использован любой допустимый столбец, перечисленный в окне мастера

Полнотекстовый поиск может читать документы, сохраненные в столбцах с типом данных `image`. Использование полнотекстового поиска с особо большими двоичными объектами мы обсудим в одном из следующих разделов.

5. Желательно установить отслеживание изменений; при этом любые изменения в данных будут сразу же отражаться в каталоге. Если не включать отслеживание изменений, то обновление каталога придется выполнять вручную.
6. Выберите существующий каталог или создайте новый.
7. Не создавайте расписание заполнения каталога: существуют лучшие способы поддержания его в обновленном состоянии. (О стратегиях поддержания полнотекстового индекса мы поговорим немного позже в этой главе.)
8. Завершите работу мастера.

Когда работа мастера завершится, каталог будет создан, но пока останется пустым. Для изначального заполнения каталога щелкните правой кнопкой мыши на таблице и выберите в контекстном меню пункт Full-Text Index Table⇨Enable Full-Text Index, а затем Full-Text Index Table⇨Start Full Population. SQL Server начнет передавать данные службе полнотекстового поиска для индексации. В зависимости от объема данных в индексируемых столбцах процесс заполнения каталога может продолжаться как несколько секунд, так и несколько часов.

Создание каталога на языке T-SQL

Для реализации полнотекстового поиска с возможностью повторения на других серверах лучше использовать язык SQL. Создание каталога с помощью программы предполагает практически те же действия, что и при использовании мастера. Для создания и конфигурирования полнотекстовых индексов используется набор системных хранимых процедур. В следующей последовательности действий продемонстрировано, как создать полнотекстовый каталог в учебной базе данных басен Эзопа (`Aesop`).

1. Откройте базу данных, в которой нужно сконфигурировать полнотекстовый поиск:

```
USE AESOP
EXEC sp_fulltext_database 'enable'
```



Каждое из описываемых действий может занять несколько секунд. SQL Server инициирует процесс, но не ожидает его завершения. Если конфигурирование осуществляется программным путем, вы можете вручную вставить необходимую паузу.

2. Создайте полнотекстовый каталог:

```
EXEC sp_fulltext_catalog 'AesopFable', 'create'
```

3. Выберите таблицу для полнотекстового поиска:

```
EXEC sp_fulltext_table
'Fable', 'create', 'AesopFable', 'FablePK'
```

4. Добавьте в полнотекстовый каталог столбцы:

```
EXEC sp_fulltext_column 'Fable', 'Title', 'add'
EXEC sp_fulltext_column 'Fable', 'Moral', 'add'
EXEC sp_fulltext_column 'Fable', 'FableText', 'add'
```

Хранимая процедура `sp_fulltext_column` имеет два других параметра, определяющих язык сканирования и информацию об индексации изображений. Полный синтаксис этой процедуры имеет следующий вид:

```
sp_fulltext_column
@tabname ='имя_таблицы',
@colname ='имя_столбца',
@action = 'действие',
@language = 'язык',
@type_colname ='тип_имени_столбца'
```

Параметр *действие* может принимать значение `'add'` или `'drop'`. Полнотекстовый поиск автоматически сканирует информацию на следующих языках.

Нейтральный — 0
Упрощенный китайский — 0x804, 2052
Традиционный китайский — 0x0404, 1028
Датский — 0x0413, 1043
Английский (Великобритания) — 0x0809, 2057
Английский (США) — 0x0409, 1033
Французский — 0x040c, 1036
Немецкий — 0x0407, 1031
Итальянский — 0x410, 1040
Японский — 0x0411, 1041
Корейский — 0x412, 1042
Современный испанский — 0x0c0a, 3082
Шведский — 0x041d, 1053
Тайский
Китайский (Гонконг)
Китайский (Макао)
Китайский (Сингапур)

Выбор языка определяет правила переноса слов, а также их видоизмененные формы. Если в базе данных использовано несколько языков или неподдерживаемый язык, выбирайте нейтральный. Поддерживаемый шестнадцатеричный код передается параметру системной хранимой процедуры `sp_fulltext_columns`. Все столбцы таблицы должны использовать один и тот же язык.

5. Активизируйте полнотекстовый каталог:

```
EXEC sp_fulltext_table 'Fable','activate'
```

Несмотря на то что полнотекстовый каталог создан, он пока еще не заполнен. Для его заполнения запустите следующую хранимую процедуру:

```
EXEC sp_fulltext_table 'Fable', 'start_full'
```

Помещение данных в полнотекстовый индекс

Полнотекстовые индексы поддерживаются внешней службой и обновляются только тогда, когда SQL Server передает ей новые данные. Это можно расценивать и как достоинство, и как недостаток. С одной стороны, это значит, что полнотекстовый индекс не тормозит работу с большими текстовыми базами данных, с другой — то, что SQL Server не поддерживает в полнотекстовых индексах режим реального времени. Если пользователь вводит данные, а затем выполняет поиск, предварительно не обновив полнотекстовый индекс, то новая информация найдена не будет.

Любой полнотекстовый индекс создается пустым, и если в таблицах SQL Server содержатся данные, то они должны быть вытолкнуты в индекс посредством *полного заполнения*. Полное заполнение заново инициализирует индекс и заполняет его данными обо всех строках. Полное заполнение можно выполнить как программным путем, так и в утилите Management Studio. Так как выталкивание данных управляется сервером, одновременно службе полнотекстового поиска отсылается только одна таблица, независимо от того, сколько их должно быть проиндексировано в каталоге. Если полнотекстовый индекс создается для пустой таблицы, то полное заполнение не требуется.

Существуют два основных метода выталкивания текущих изменений в полнотекстовый индекс.

- **Постепенное заполнение**. Постепенное заполнение использует время создания для выявления строк, созданных после последнего заполнения. Этот метод можно использовать в утилите Management Studio, в программном коде T-SQL, а также в заданиях агента SQL Server Agent (обычно такие задания планируются на вечернее время). Для последовательного заполнения требуется наличие в таблице столбца типа `timestamp`.

  Постепенное заполнение выявляет две проблемы. Во-первых, между вводом данных и возможностью полнотекстового поиска существует временной разрыв. Во-вторых, все обновления консолидируются в единый процесс, который может отнять массу процессорного времени. В загруженных базах данных существует выбор между постепенным заполнением каждый вечер и выделением для этой операции отдельного дня, чтобы не влиять на общую производительность сервера.

- **Отслеживание изменений и фоновое заполнение**. SQL Server может отслеживать изменения данных в столбцах, по которым созданы полнотекстовые индексы. Как только изменение обнаружено в некоторой строке, индекс будет обновлен только по ней. Несмотря на то что этот метод также расходует ресурсы компьютера, на практике эффект от его использования почти не заметен. Полнотекстовое обновление не вызывается триггером, так что транзакциям обновления не приходится ждать, пока данные будут вытолкнуты в полнотекстовый индекс. Вместо этого обновление полнотекстового индекса выполняется в фоновом режиме практически сразу же после транзакции SQL DML. В результате загрузка процессора балансируется, и индексация происходит практически в режиме реального времени.


Проверено

Если проект базы данных предполагает поиск слов в столбцах, то использование полнотекстовой индексации с отслеживанием изменений и заполнением в фоновом режиме является лучшей стратегией повышения производительности.

Обслуживание каталога в Management Studio

В утилите Management Studio полнотекстовый поиск выполняется с помощью щелчка правой кнопкой мыши и выбора команды в контекстном меню каждой таблицы. Все команды обслуживания полнотекстового поиска собраны в подменю Full-Text Index Table.

- Define Full-Text Indexing on Table. Запуск мастера полнотекстовой индексации так, как было описано выше в этой главе.
- Edit Full-Text Indexing. Запускает мастера полнотекстовой индексации для изменения каталога выделенной таблицы.
- Remove Full-Text Indexing from a Table. Удаляет выделенную таблицу из ее каталога.
- Start Full Population. Инициирует заполнение данных из всех строк выделенной таблицы в каталог полнотекстовой индексации.
- Start Incremental Population. Инициирует заполнение всех данных, измененных после последнего заполнения полнотекстового индекса.
- Stop Population. Останавливает любой текущий процесс заполнения.
- Change Tracking. Выполняет полное или постепенное заполнение, после чего включает отслеживание изменений данных для обновления индексов.

- Update Index in Background. Запускает процесс обновления строк, отмеченных отслеживанием изменения данных.
- Update Index. Запускает обновления всех строк, отмеченных отслеживанием изменений, в полнотекстовом индексе.

Обслуживание каталога в программном коде T-SQL

Все перечисленные выше команды меню утилиты Management Studio можно выполнить и программным путем. Приведенные ниже фрагменты демонстрируют использование команд управления каталогом на примере учебной базы данных басен Эзопа.

- **Полное заполнение**.

```
EXEC sp_fulltext_table 'Fable', 'start_full'
```

- **Постепенное заполнение**.

```
EXEC sp_fulltext_table 'Fable', 'start_incremental'
```

- **Удаление полнотекстового каталога**.

```
EXEC sp_fulltext_table 'AesopFable', 'drop'
```

- **Отслеживание изменений и фоновое обновление**.

```
EXEC sp_fulltext_table Fable, 'Start_change_tracking'
EXEC sp_fulltext_table Fable,
'Start_background_updateindex'
```

В дополнение хранимые процедуры содержат следующие расширенные функции обслуживания каталогов.

- **Перестройка**. Эта команда удаляет и полностью переопределяет полнотекстовый каталог, но не заполняет его индексами, — после этой операции нужно выполнять полное заполнение. Преимущество переопределения каталога заключается в том, что оно автоматически переконфигурирует таблицу и столбцы, гарантируя чистоту внутренней структуры каталога.

```
EXEC sp_fulltext_catalog 'AesopFable', 'rebuild'
```

- **Очистка неиспользуемых полнотекстовых каталогов**. Следующая команда очищает неиспользуемые полнотекстовые каталоги:

```
EXEC sp_fulltext_service 'clean_up'
```

Начиная с версии SQL Server 2000 хранимая процедура `sp_help` обеспечивает пользователей информационной справочной поддержкой. Версии этой процедуры следующие.

- `sp_help_fulltext_catalogs`. Эта системная хранимая процедура возвращает информацию о каталоге, включая статус текущего заполнения.

```
EXEC sp_help_fulltext_catalogs 'AesopFable'
```

Результат выполнения этой процедуры следующий:

```
                                                    NUMBER
                                                    FULLTEXT
ftcatid NAME         PATH                  STATUS   TABLES
------- ------------ --------------------- ------ ------
5       AesopFable   C:\Program Files      0      1
                     \Microsoft SQL Server
                     \MSSQL\FTDATA
```

Столбец статуса заполнения возвращает следующие состояния каталога.

- 0 — нет активности
- 1 — идет полное заполнение
- 2 — заполнение приостановлено
- 3 — перегрузка
- 4 — восстановление
- 5 — закрыт
- 6 — идет постепенное наполнение
- 7 — создание индекса
- 8 — диск полон, работа приостановлена
- 9 — отслеживание изменений

- `sp_help_fulltext_tables`. Этот вариант хранимой процедуры возвращает информацию о таблицах, занесенных в каталог.

```
EXEC sp_help_fulltext_tables 'AesopFable'
```

Результат (форматированный):

```
             FULLTEXT
             KEY        FULLTEXT   FULLTEXT   FULLTEXT
TABLE TABLE  INDEX      KEY        INDEX      CATALOG
OWNER NAME   NAME       COLID      ACTIVE     NAME
----- -----  --------   ---------  ---------  -----------
dbo   Fable  FablePK    1          1          AesopFable
```

- `sp_help_fulltext_columns`. Возвращает информацию о столбцах, включенных в каталог.

```
EXEC sp_help_fulltext_columns 'fable'
```

Результат (форматированный и сокращенный):

```
TABLE_ FULLTEXT BLOBTP
OWNER NAME   COLUMNNAME   COLNAME    LANGUAGE
----- -----  -----------  ---------  ----------
dbo   Fable  Title        NULL       1033
dbo   Fable  Moral        NULL       1033
dbo   Fable  FableText    NULL       1033
```

Файлы шумов

Когда несколько лет назад я писал процедуру поиска слов, то с целью оптимизации и повышения производительности исключил такие слова, как `a`, `the` и `of`. Как только слово было просканировано, я проверял его наличие в "черном списке" и в случае получения положительного результата переходил к следующему слову. Таким образом, время сканирования обычных сайтов сократилось более чем вдвое, а таблица частоты слов стала значительно короче.

Полнотекстовый поиск использует аналогичный подход, сохраняя список игнорируемых слов в так называемом *файле шума*. Занесенные в этот файл слова полностью игнорируются полнотекстовым поиском. На самом деле, если запрос содержит игнорируемые слова, SQL Server генерирует ошибку.

Решение включать слово в файл шума принимается на основании частоты его использования и относительной важности. Если слово распространенное, то оно будет часто встречаться в тексте и тем самым снижать общую производительность сервера. По этой причине такие слова лучше включать в файл шума.

В качестве альтернативы можно выполнять поиск и в файле шума. Например, если для базы данных важно определение "язык С", букву "С" следует удалить из файла шума.

Так как файлы шума являются обычными текстовыми файлами, они должны удовлетворять требованиям конкретных приложений. Перед редактированием файла шума нужно остановить механизм полнотекстового поиска. Если установка SQL Server выполнялась в каталог, предложенный по умолчанию, то файл шума можно найти по следующему пути:

```
C:\Program Files\Microsoft SQL Server\MSSQL\FTDATA\noiseENU.txt
```

Для тестирования файла шума остановите механизм полнотекстового поиска в утилите SQL Server Configuration Manager, добавьте в файл какое-либо слово и попытайтесь выполнить по нему поиск. Если SQL Server выдаст следующую ошибку, значит, слово было добавлено в корректный файл шума:

```
Server: Msg 7619, Level 16, State 1, Line 1
A clause of the query contained only ignored words
```

Поиск слов

Как только каталог будет создан, функция полнотекстового поиска готова к работе. Поиск слов выполняется с помощью функции `contains`, которая помогает передать искомое слово механизму полнотекстового поиска и дождаться ответа. Поиск слова в запросе может выполняться одним из двух методов: `contains` и `ContainsTable`.

Функция `contains`

Функция `contains` оперирует в предложении `WHERE`, почти как `WHERE IN (`*критерий*`)`. Параметры, помещенные в скобки, передаются службе полнотекстового поиска, которая возвращает список первичных ключей, удовлетворяющих критерию.

Первым параметром, передаваемым службе MS Search, является имя столбца или звездочка, если поиск нужно выполнить во всех столбцах одной таблицы. Если предложение `FROM` содержит несколько таблиц, то все они должны быть перечислены в параметре `contains`. Приведенный в качестве примера запрос полнотекстового поиска ищет слово "Lion" во всех проиндексированных столбцах:

```
USE Aesop
SELECT Title
  FROM Fable
  WHERE CONTAINS (Fable.*,'Lion')
```

Следующие таблицы содержат искомое слово либо в заголовке, либо в морали, либо в основном тексте:

```
Title
--------------------------------------------------
The Dogs and the Fox
The Hunter and the Woodman
The Ass in the Lion's Skin
Androcles
```

Функция `ContainsTable`

В предложении `WHERE` может содержаться не только ключевое слово полнотекстового поиска — функция `ContainsTable` работает как управляемая таблица или подзапрос и возвращает результирующий набор данных. Эта функция SQL Server открывает широкие возможности.

Результирующий набор данных эта функция возвращает в двух столбцах. Первый столбец, Key, идентифицирует строку, используя уникальный индекс, использованный при конфигурировании каталога.

Второй столбец, Rank, ранжирует строки, используя значения от 1 (наименьшее значение) до 1000 (наибольшее значение). Не существует большего или меньшего значения, которое может воспринять сервер. Этот диапазон определяется несколькими факторами.

- Частота и уникальность слов в таблице.
- Частота и уникальность слов в столбце.

В то же время не существует случаев, когда какое-либо слово не было ранжировано выше общеупотребительного.

Практически те же параметры определяют синтаксис функции ContainsTabele. Следующий запрос возвращает текстовые данные из службы MS Search:

```
SELECT *
   FROM CONTAINSTABLE (Fable, *, 'Lion')
```

Будет получен следующий результат:

```
KEY         RANK
----------- -----------
3           86
4           80
20          48
14          32
```

Сам по себе ключ (поле Key) бесполезен для человека, однако, будучи объединенным с результатами обращения функции ContainsTable к таблице Fable, позволяет получить столбцы ранга и заголовка:

```
SELECT Fable.Title, FTS.Rank
  FROM Fable
    JOIN CONTAINSTABLE (Fable, *, 'Lion') FTS
    ON Fable.FableID = FTS.[KEY]
  ORDER BY FTS.Rank DESC
```

Результат будет следующим:

```
Title                        Rank
---------------------------- -----
Androcles                    86
The Butt in the Lion's Skin  80
The Hunter and the Woodman   48
The Dogs and the Fox         32
```

Четвертый параметр функции ContainsTable ограничивает объем результатов, возвращаемых механизмом полнотекстового поиска, практически как предикат TOP инструкции SELECT. Это подразумевает упорядочивание по рангу, так что в результат войдет заданное количество значений с наибольшим рангом. В следующем примере продемонстрировано использование этого параметра:

```
SELECT Fable.Title, Rank
  FROM Fable
    JOIN CONTAINSTABLE (Fable, *, 'Lion', 2) FTS
    ON Fable.FableID = FTS.[KEY]
  ORDER BY FTS.Rank DESC
```

В результат поиска войдут всего два значения:

```
Title                              Rank
---------------------------------- -----------
```

```
Androcles                          86
The Ass in the Lion's Skin         80
```

Основное преимущество использования этого параметра заключается в том, что сам механизм полнотекстового поиска ограничивает объем результатов и не приходится дополнительно выполнять инструкцию SELECT с предикатом TOP. Этот пример иллюстрирует принципы переноса нагрузки с клиентского компьютера на сервер, так как в данном случае механизм полнотекстового поиска MS Search является серверным процессом, а SQL Server — клиентским.



Проверено

Поскольку механизм полнотекстового поиска является компонентом, отличным от SQL Server, он будет соревноваться с последним за ресурсы процессора. Исходя из этого, использование механизма MS Search наряду с SQL Server позволит более эффективно использовать многопроцессорные серверы. Механизм MS Search также интенсивно использует память и файл подкачки. В сильно загруженных базах данных с регулярными обновлениями и частым поиском информации в больших текстовых столбцах такое распределение функций будет лучшим решением.

Расширенные параметры поиска

Полнотекстовый поиск является мощным инструментом и предоставляет пользователю массу возможностей, которые можно реализовать с помощью функций `contains` и `ContainsTable`.

Поиск нескольких слов

В поисковую фразу может быть включено несколько слов; при этом используются связки OR и AND. Следующий запрос найдет все басни, одновременно содержащие в тексте слова "Tortoise" (черепаха) и "Hare" (заяц):

```
SELECT Title
  FROM Fable
  WHERE CONTAINS (FableText,'Tortoise AND Hare')
```

Будет получен следующий результат:

```
Title
-------------------------
The Hare and the Tortoise
```

Однако существует одно ограничение. Несмотря на то что механизм полнотекстового поиска позволяет искать одно слово в нескольких столбцах, того же нельзя сказать о поиске нескольких слов. Выполнить его возможно только по одному столбцу. Например, басня "The Ants and the Grasshopper" содержит слово "thrifty" (бережливый) в морали и слово "supperless" — в тексте самой басни. В данном случае следующий поиск не даст никаких результатов:

```
SELECT Title
FROM Fable
WHERE CONTAINS (*,' "Thrifty AND supperless" ')
```

Вместо результата мы получим сообщение:

```
(0 row(s) affected)
```

Существуют два решения этой проблемы, но ни одно из них нельзя назвать лучшим. Можно перестроить запрос таким образом, чтобы связка AND находилась в предложении WHERE, а не в параметре функции CONTAINS. Проблема этого решения заключается в низкой производительности. Следующий запрос выполняет два отдельных сканирования с помощью

механизма полнотекстового поиска (рис. 13.2), каждое из которых занимает 363 миллисекунды; при этом общее время выполнения запроса составляет 811 миллисекунд:

```
SELECT Title
  FROM Fable
    WHERE CONTAINS (*,'Thrifty')
       AND CONTAINS(*,'supperless')
```

Будет получен следующий результат:

```
Title
----------------------------
The Ants and the Grasshopper
```

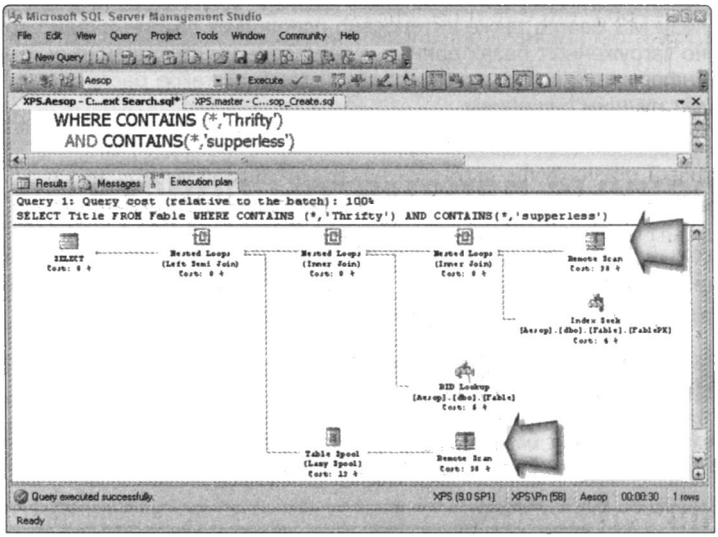


Рис. 13.2. Каждая функция `contains` требует отдельного удаленного вызова механизма полнотекстового поиска. Только после этого результаты просматриваются сервером

Еще одно решение поиска в нескольких столбцах заключается в создании дополнительного столбца для хранения всего потенциально искомого текста и дублировании данных из исходных столбцов с помощью триггера. Это решение тоже не лучшее, поскольку данные дублируются, а производительность понижается за счет запуска триггера. Ключ к решению задачи заключается в использовании OLAP вместо OLTP.

Поиск с использованием символов макроподстановки

Так как механизм полнотекстового поиска является частью операционной системы, а не компонентом SQL Server, при использовании символов макроподстановки используются обычные соглашения DOS (т.е. звездочка, двойные кавычки и т.п.), а не соглашения SQL Server.

Следует также запомнить, что символы макроподстановки работают только в конце слова, но не в начале. Индексы начинают поиск от начала строки, как в следующем примере:

```
SELECT Title
  FROM Fable
  WHERE CONTAINS (*,' "Hunt*" ')
```

Будет получен следующий результат:

```
Title
--------------------------
The Hunter and the Woodman
The Ass in the Lion's Skin
The Bald Knight
```

Если во фразе используется символ макроподстановки "звездочка", то он заменяет собой любое слово. Например, следующие два запроса эквивалентны:

```
CONTAINS (*,'He pulled out the thorn*')
CONTAINS (*,'He* pulled* out* the* thorn*')
```

Поиск фраз

Полнотекстовый поиск будет пытаться искать полные фразы, если таковые заключены в функции `contains` в двойные кавычки. Например, для поиска басни про мальчика, который кричал на волка, мы можем использовать следующий трюк:

```
SELECT Title
  FROM Fable
  WHERE CONTAINS (*,' "Wolf! Wolf!" ')
```

Будет получен следующий результат:

```
Title
-------------------------------
The Shepherd's Boy and the Wolf
```

Поиск близких слов

При выполнении поиска в больших документах хорошо иметь возможность задавать в поисковой фразе близкие слова. Для этого в механизме полнотекстового поиска предназначено ключевое слово `near`. При его использовании подсчитывается относительное расстояние между словами и, если они находятся достаточно близко (в пределах около тридцати слов, в зависимости от размера текста), для данной строки возвращается истинное значение.

Басня об Андрокле, который вынимал колючку из лапы льва, является самой длинной в рассматриваемой учебной базе данных. Ее мы и используем в качестве примера поиска в тексте близких слов "pardoned" и "forest":

```
SELECT Title
  FROM Fable
  WHERE CONTAINS (*,'pardoned NEAR forest')
```

Будет получен следующий результат:

```
Title
---------------
Androcles
```

Существует и возможность поиска нескольких близких слов. В следующем примере мы будем искать близко расположенные слова "lion", "paw" и "bleeding":

```
SELECT Title
  FROM Fable
  WHERE CONTAINS (*,'lion NEAR paw NEAR bleeding')
```

Результат будет получен тот же:

```
Title
---------------
Androcles
```

Параметр близости можно использовать и в функции ContainsTable и получать ранг от 0 до 64, указывающий на степень близости слов. Следующий запрос ранжирует басни, содержащие слова "life" и "death":

```
SELECT Fable.Title, Rank
  FROM Fable
  JOIN CONTAINSTABLE (Fable, *,'life NEAR death') FTS
    ON Fable.FableID = FTS.[KEY]
  ORDER BY FTS.Rank DESC
```

Будет получен следующий результат:

```
Title                              Rank
---------------------------------- -----------
The Serpent and the Eagle          7
The Eagle and the Arrow            1
The Woodman and the Serpent        1
```

Поиск словоформ

Механизм полнотекстового поиска может осуществлять лингвистический анализ и искать однокоренные слова. Это позволяет искать слова, не волнуясь об их склонениях и спряжениях. Например, поиск слова "flying" может найти и слово "flew". Для случаев, подобных данному, язык, используемый в таблице, имеет критическое значение. Также следует учесть, что данная функция не смешивает разные части речи, даже если они имеют один корень. Например, поиск существительного не найдет глагол с тем же корнем. Следующий запрос демонстрирует поиск слова "flew", несмотря на то, что в параметрах функции contains задано слово "fly":

```
SELECT Title
  FROM Fable
  WHERE CONTAINS (*,'FORMSOF(INFLECTIONAL,fly)')
```

Будет получен следующий результат:

```
Title
-------------------------
The Crow and the Pitcher
The Bald Knight
```



Хорошие клиентские программы предоставляют пользователю возможность выделения в документе искомых слов. Если пользователь ввел слово "fly", а найденное слово оказалось "flew", то простой поиск и замена при форматировании пропустит его. Сценарий webhits.dll в сервере индексации поможет справиться с этой проблемой.

Поиск синонимов

Новым в версии SQL Server 2005 является поиск синонимов для замены слов. Чтобы сконфигурировать собственные параметры тезауруса, отредактируйте соответствующий файл в каталоге:

```
C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Binn\FTERef
```

Файл тезауруса для вашего языка будет иметь вид TS*XXX*.xml, где *XXX* — код языка (например, ENU — для американского диалекта английского языка). Из своего файла тезауруса вам придется удалить линии комментариев. Если вы откроете этот файл в текстовом редакторе, то обнаружите в нем два раздела: *expansion* (узел расширения) и *replacement* (узел замены). Первый из них позволяет расширить аргумент поиска за счет других. Например, в английском файле тезауруса вы найдете следующие строки:

```
<expansion>
    <sub>Internet Explorer</sub>
    <sub>IE</sub>
    <sub>IE5</sub>
  </expansion>
```

Этот фрагмент позволяет преобразовать любой поиск аббревиатуры "IE" в поиск как "IE", так и "IE5" или "Internet Explorer".

Узел замены используется для замены одного аргумента поиска другим. Например, слово "sex" может быть вполне заменено словом "gender"; для этого можно создать следующий раздел:

```
<replacement>
  <pat>sex</pat>
  <sub>gender</sub>
</replacement>
```

Элемент <pat> определяет шаблон, а элемент <sub> — подставляемое вместо него слово.

Запросы FREETEXT автоматически используют соответствующие для данного языка тезаурусы. В следующем примере для использования в поиске тезауруса мы используем ключевое слово Thesaurus.

В результате будут возвращены строки, содержащие термины IE, IE5 и Internet Explorer.

Поиск с использованием веса слов

При поиске множества слов может браться в расчет их относительный вес (т.е. одно может быть более критично, другое — менее). Веса устанавливаются в диапазоне от нуля до единицы.

Параметр isabout позволяет взвешивать слова в строках, подлежащих возвращению, поэтому его работа подобна булеву оператору OR.

В следующих двух запросах параметр weight используется для установки критичности слов "lion", "brave" и "eagle". В них будет проверяться только столбец текста басни fabletext, чтобы результаты поиска не смешивались с данными столбцов заголовка и морали. В первом примере все слова имеют равный вес:

```
SELECT Fable.Title, FTS.Rank
  FROM Fable
  JOIN CONTAINSTABLE
   (Fable, FableText,
        'ISABOUT (Lion weight (.5),
         Brave weight (.5),
         Eagle weight (.5))',20) FTS
  ON Fable.FableID = FTS.[KEY]
  ORDER BY Rank DESC
```

Получим следующий результат:

```
Title                              Rank
---------------------------------- --------
Androcles                          92
The Eagle and the Fox              85
The Hunter and the Woodman         50
```

```
The Serpent and the Eagle          50
The Dogs and the Fox               32
The Eagle and the Arrow            21
The Ass in the Lion's Skin         16
```

Если повысить важность слова "eagle", то результат будет иным:

```
SELECT Fable.Title, FTS.Rank
FROM Fable
JOIN CONTAINSTABLE
(Fable, FableText,
'ISABOUT (Lion weight (.2),
Brave weight (.2),
Eagle weight (.8))',20) FTS
ON Fable.FableID = FTS.[KEY]
ORDER BY Rank DESC
```

Будет получен следующий результат:

```
Title                              Rank
---------------------------------- -------
The Eagle and the Fox              102
The Serpent and the Eagle          59
The Eagle and the Arrow            25
Androcles                          25
The Hunter and the Woodman         14
The Dogs and the Fox               9
The Ass in the Lion's Skin         4
```

Когда в полнотекстовом поиске участвуют все столбцы, маленький размер морали и заголовка сделает относительно более важными слова, расположенные в тексте. Следующий запрос использует те же веса, что и предыдущий, однако в него включены все столбцы:

```
SELECT Fable.Title, FTS.Rank
  FROM Fable
    JOIN CONTAINSTABLE
     (Fable, *,
         'ISABOUT (Lion weight (.2),
          Brave weight (.2),
          Eagle weight (.8))',20) FTS
     ON Fable.FableID = FTS.[KEY]
  ORDER BY Rank DESC
```

Будет получен следующий результат:

```
Title                              Rank
---------------------------------- ------
The Wolf and the Kid               408
The Hunter and the Woodman         408
The Eagle and the Fox              102
The Eagle and the Arrow            80
The Serpent and the Eagle          80
Androcles                          25
The Ass in the Lion's Skin         23
The Dogs and the Fox               9
```

Ранжирование относительно. Оно основано на частоте, близости и относительной важности слов. В басне "The wolf and the Kid" не содержится ни слова "lion", ни "eagle", упомянутых в запросе, однако в то же время в двух местах встречается слово "brave". Данное слово реже встречается в тексте, но является одним из всего десяти слов, составляющих мораль. Даже несмотря на то, что слово "brave" имеет меньший вес, оно поставило басню во главу списка. Все основано на частоте слов и статистике (и, по-моему, на фазах луны).

Нечеткий поиск

В то время как предикат contains и создаваемая функцией ContainsTable таблица выполняют точный поиск слов, функция contains позволяет выполнять и приблизительный. так называемый *нечеткий поиск*.

Вместо поиска двух или трех слов и добавления параметров близости и взвешивания, нечеткий поиск берет на вооружение все возможности механизма полнотекстового поиска в попытке решить заданную вами задачу. Произвольный текст разбивается на множество слов и фраз, после чего выполняется поиск с проверкой близости и взвешиванием.

Параметр FREETEXT

Параметр FREETEXT используется в предложении WHERE, практически как CONTAINS, но не задействует дополнительные атрибуты. Следующий запрос выполнит нечеткий поиск басни о зайце и черепахе:

```
SELECT Title
  FROM Fable
  WHERE FREETEXT
   (*,'The tortoise beat the hare in the big race')
```

Будет получен следующий результат:

```
Title
------------------------------
The Hare and the Tortoise
```

Параметр FREETEXTTABLE

Этот параметр имеет преимущество перед FREETEXT в том, что ранжирует результаты практически так же, как функция ContainsTable. Два примера, представленных в настоящем разделе, демонстрируют нечеткий полнотекстовый поиск с использованием функции FREETEXTTABLE.

```
SELECT Fable.Title, FTS.Rank
  FROM Fable
    JOIN FREETEXTTABLE
     (Fable, *, 'The brave hunter kills the lion',20) FTS
      ON Fable.FableID = FTS.[KEY]
  ORDER BY Rank DESC
```

Результат выполнения запроса:

```
Title                              Rank
---------------------------------- -------
The Hunter and the Woodman         257
The Ass in the Lion's Skin         202
The Wolf and the Kid               187
Androcles                          113
The Dogs and the Fox               100
The Goose With the Golden Eggs     72
The Shepherd's Boy and the Wolf    72
```

А вот и второй запрос:

```
SELECT Fable.Title, FTS.Rank
  FROM Fable
```

```
  JOIN FREETEXTTABLE
   (Fable, *, 'The eagle was shot by an arrow',20) FTS
     ON Fable.FableID = FTS.[KEY]
  ORDER BY Rank DESC
```

Результат данного запроса следующий:

```
Title                              Rank
---------------------------------- -----------
The Eagle and the Arrow            288
The Eagle and the Fox              135
The Serpent and the Eagle          112
The Hunter and the Woodman         102
The Father and His Two Daughters   72
```

Индексация двоичных объектов

SQL Server может хранить любые двоичные объекты, размер которых не превышает 2 Гбайт. Полнотекстовый поиск может индексировать такие объекты, если удовлетворяются следующие критерии.

- В Windows должен быть установлен фильтр для данного типа объекта. SQL Server устанавливает в файле `offfilt.dll` фильтры для следующих типов файлов: `.doc`, `.xls`, `.ppt`, `.txt` и `.htm`.
- Отдельный столбец типа `char(3)` должен хранить расширение документа, содержащегося в данной строке.
- Отдельный столбец должен быть добавлен для каталога полнотекстового поиска, а тип документа (например, `.doc`, `.xls` или `.ppt`) должен храниться в прилегающем столбце.
- Каталог полнотекстового поиска может заполняться полностью и частично, при этом отслеживание изменений и фоновое обновление для таких столбцов не применимо.
- После загрузки в SQL Server большие документы должны быть правильно инициализированы с помощью программы Bulk Image Insert.

Даже если полнотекстовый поиск правильно установлен для обслуживания особо больших документов, эта технология более чем несовершенна, и нужно уметь заставить ее работать.

Следующая хранимая процедура запускается для подготовки столбца с особо большими документами к полнотекстовому поиску:

```
EXEC sp_fulltext_column
    'Fable','Blob','add',0x0409,'BlobType'
```

Параметры вызова те же, что и при создании текстового столбца, за исключением того, что последний из них идентифицирует столбец, используемый для расширения документа.

SQL Server использует для вставки особо больших документов специальную программу загрузки крупных изображений, которая называется `BII.exe`. Это модифицированная версия программы загрузки очень больших документов. Она заархивирована в файле `unzip_util.zip`, находящемся в каталоге `C:\Program Files\Microsoft SQL Server\80\Tools\DevTools\Samples\utils`. После разархивации создается подкаталог `bii`, куда помещаются все файлы утилиты.

Утилита копирования больших изображений копирует данные из текстового файла в SQL Server. Данные в этом текстовом файле должны быть разделены точками с запятой (несмотря на то, что документация утверждает, что это должны быть запятые). Имя самого документа должно быть указано в файле после символа `@`.

В следующем примере выполняется загрузка документа `fox.doc` в рассматриваемую нами учебную базу басен Эзопа. В шестой столбец добавляется тип документа, а в седьмой вставляется содержимое файла (одна строка была разбита на две, чтобы поместиться на странице):

```
Sample.txt:
26; Test Fable; Persistence Pays Off;
        Try, try again.;doc;@fox.doc
```

После вызова утилиты `bii.exe` из командной строки все данные из файла `sample.txt` будут перемещены в таблицу `fable`. Остальные параметры определяют имя сервера и базы данных, имя пользователя и его пароль. Параметр `-v` указывает программе сопроводить свою работу созданием отчета (приведенный ниже текст также был отформатирован, чтобы он мог поместиться на листе).

```
>bii "fable" in "sample.txt"
        -S"Noli" -D"Aesop" -U"sa" -P"sa" -v
```

Результат выполнения утилиты:

```
BII - Bulk Image Insert Program for Microsoft SQL Server.
Copyright 2000 Microsoft Corporation, All Rights Reserved.
Version: V1.0-1
Started at 2001-12-07 16:28:09.231 on NOLI
Table Noli.Aesop.sa.fable
FableID int (4)
Title varchar (50)
Moral varchar (100)
FableText varchar (1536)
BlobType char (3) null
Blob image (16) null
Inserted 1 rows Read 1 rows 0 rows with errors
Total Bytes = 19508 inserted 19456 File Bytes
Total Seconds = 0.02 Kb Per Second = 952.539063
BII - Bulk Image Insert Program for Microsoft SQL Server.
Copyright 2000 Microsoft Corporation, All Rights Reserved.
Version: V1.0-1
Finished at 2001-12-07 16:28:09.332 on NOLI
```

Как только двадцать шестая басня будет загружена в базу данных и будет заполнен полнотекстовый каталог, вы сможете использовать следующую команду для поиска документа Word в SQL Server:

```
EXEC sp_fulltext_table 'Fable', 'start_full'
```

Например, следующий запрос ищет слово "jumped", содержащееся во вставленной двадцать шестой басне:

```
SELECT Title, BlobType
  FROM Fable
  WHERE CONTAINS (*,'jumped')
```

Результат выполнения запроса:

```
Title          BlobType
-------------- -----------
Test Fable     doc
```

Вопросы производительности

Производительность полнотекстового поиска в SQL Server 2005 на несколько порядков выше, чем в SQL Server 2000 и SQL Server 7. Однако у вас все равно может возникнуть желание добиться в своей системе оптимальной производительности.

- Полнотекстовый поиск в значительной мере зависит от быстродействия компьютерной системы. Поместите каталог на отдельный контроллер, желательно с использованием дискового массива RAID 10. Чтобы сконфигурировать полнотекстовый поиск для оптимальной производительности, выполните следующую команду:

  ```
  sp_fulltext_service 'resource_usage'
  ```

- После полного или частичного заполнения консолидируйте все индексы в главный индекс, выполнив следующую команду:

  ```
  ALTER FULLTEXT CATALOG catalog_name REORGANIZE
  ```

- Также вы можете максимизировать количество рангов, которые может использовать процесс сбора информации, выполнив следующую команду:

  ```
  sp_configure 'max full-text crawl range', 100.
  ```

- Вы также можете изменить настройку `'ft crawl bandwidth (max)'`, которая управляет объемом памяти, выделяемым для заполнения. Повышая этот объем, мы выделяем больше памяти для индексации, но тем самым забираем ресурсы у сервера. Этот объем памяти можно увеличить с помощью следующей команды:

  ```
  sp_configure 'ft crawl bandwidth (max)', 1000
  ```

Резюме

Индексы SQL Server не предназначены для поиска слов и фраз в середине поля. В то же время, если того требует создаваемый проект, на помощь придет полнотекстовый поиск, хотя он и потребует от вас дополнительной работы, в том числе административной.

Эту часть книги мы посвятили описанию операций с данными с помощью инструкции `SELECT`. В этой главе вы узнали, как использовать в этой инструкции для извлечения данных функции полнотекстового поиска. В следующей главе мы продолжим тему извлечения данных, описав, как использовать предопределенные инструкции SQL в роли представлений.

Создание представлений

ГЛАВА

14

В этой главе...

Грамотное планирование представлений

Создание представлений в Management Studio

Обновление посредством представлений

Производительность и представления

Вложенные представления

Представлением называют сохраненную инструкцию SQL, на которую можно сослаться как на источник данных в запросе, так же как на подзапрос, — ни больше ни меньше. Представление не может быть выполнено самостоятельно; оно должно использоваться в запросе.

Представления иногда называют виртуальными таблицами. Это определение не совсем отвечает действительности, так как представления сами по себе не хранят данные. Подобно любому запросу SQL, представления обращаются к данным, хранимым в таблицах.

Теперь важно понять, как работают представления, рассмотреть их достоинства и недостатки, а также выяснить, где их лучше использовать в архитектуре проекта.

Зачем использовать представления

Существует несколько мнений относительно использования представлений: от полного их игнорирования до чрезмерного использования. Принципы информационной архитектуры, которые мы рассмотрели в главе 1, могут лучше всего пояснить, где уместно использование представлений. Соответствующий принцип гласит: "*информация... должна быть... представлена в удобном формате для повседневных операций и анализа пользователями, их группами и процессами...*".

Собственно, основной задачей представлений и является отображение данных в удобном формате.

Данные в нормализованной СУБД редко организованы в формате, приспособленном для просмотра. Создание разовых запросов, извлекающих корректную информацию из нормализованных запросов, достаточно трудно для большинства конечных пользователей. Хорошо написанное представление может скрыть эту сложность и при этом предоставить пользователю корректные данные.



Я рекомендую использовать представления даже для разовых запросов и отчетов. С запуском запросов представления справляются лучше, чем хранимые процедуры.

Представления помогают проверить целостность данных, поскольку даже когда пользователи понимают, как использовать объединения, они зачастую не могут правильно выбрать между внутренним и внешним объединением. А, как мы знаем, неверный выбор типа объединения приводит к некорректным результатам запроса.

Некоторые разработчики используют представления как ключевой компонент архитектуры доступа к данным и изолируют физическую схему от динамического SQL. Я продолжаю использовать хранимые процедуры на определенном уровне абстракции, поскольку они решают проблемы обработки ошибок, позволяют лучше управлять транзакциями и предлагают больше параметров для программирования (подробнее об этом — в главе 25).

Однако и представления отлично справляются с этой задачей. Я предпочитаю использовать представления для прямого доступа к данным, а не использовать для этого код SQL. Если ваше приложение .NET содержит много внедренного кода SQL, то использование представлений для уровня абстракции базы данных может оказаться более простым, чем переработка приложения для вызова хранимых процедур.

Основываясь на своей уверенности, что представления лучше использовать для поддержки разовых запросов, а не как центральную часть приложения, разрешите высказать несколько мыслей относительно создания представлений разовых запросов.

- **Используйте представления для упрощения сложных объединений и сокрытия мистических ключей, используемых для связывания данных в схеме базы данных**. Хорошо созданное представление будет отличным подспорьем в доступе пользователей к необходимым им данным.
- **Сохраняйте сложные запросы консолидации данных как представления**. Так как в итоговой функции или предложении `GROUP BY` должны участвовать все столбцы, многие запросы включают в себя подзапросы для представления неконсолидируемых столбцов. Конечные пользователи будут благодарны за включение этих сложных запросов в представления.
- **Присваивайте столбцам с труднопроизносимыми названиями понятные псевдонимы**. Как инструкция `SELECT` может использовать псевдонимы и именованные диапазоны для имен таблиц и столбцов, так и представления могут использовать эту возможность для представления набора данных конечному пользователю. Например, столбец `au_lname` в базе данных Microsoft Pubs мало что скажет конечному пользователю, если не присвоить ему понятный псевдоним `LastName` (фамилия):

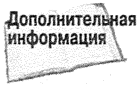
  ```
  SELECT au_lname AS LastName FROM Pubs.dbo.Author
  ```

  Теперь представление, созданное на основе указанного выше запроса, будет перечислять фамилии авторов в столбце `LastName`, а не `au_lname`, как раньше.
- **Включайте в представления только те столбцы, которые интересуют пользователя,** — это облегчит ему задачу просмотра результатов запроса. Столбцы, включаемые в представление, называют *проецируемыми*. В этом термине подразумевается, что представления проецируют на экран из таблиц только избранные столбцы.
- **Тщательно планируйте представления для их долгосрочного использования**. Представления, созданные только для одной цели, быстро устаревают и только засоряют базу данных. Создавайте представления, предполагая, что они будут использо-

ваться с предложением WHERE для выборки подмножества данных; в противном случае представление должно возвращать все множество данных. Например, представление vEventList возвращает все события; чтобы выбрать из них только конкретные, пользователь должен сопроводить его предложением WHERE.

Если представление создается для извлечения только ограниченного множества данных, то оно должно учитывать следующие периоды времени, чтобы продолжать функционировать в будущем.

Создание приложения преследует две цели: облегчить доступ пользователей к нужным им данным и защитить данные от пользователей. Создавая представления, извлекающие корректные данные, вы защищаете остальную информацию от ошибочного или злонамеренного использования.



Распределенные представления или *федеративные базы данных* делят большие таблицы на несколько маленьких, которые иногда даже размещаются на разных серверах. Это позволяет улучшить производительность систем. Распределенные представления затем собирают эту информацию из разных мест, используя несколько дисковых подсистем.

Индексированные представления, поставляемые с редакцией SQL Server 2005 Enterprise, представляют собой мощное средство, создающее индекс на денормализованном множестве данных, определенных представлением. Эти индексы впоследствии могут использоваться при создании запросов, объединяющих множества данных. Данные представления используются в запросах, так что сам термин звучит несколько странно.

Оба этих продвинутых представления полезны только для решения вопросов производительности в особо крупных базах данных, размер которых превышает терабайт. О них мы поговорим в главе 53.

Работа с представлениями

В утилите SQL Server Management Studio представления можно создавать, редактировать, выполнять и вставлять в другие запросы.

Создание представлений в Management Studio

Поскольку представление является ничем иным, как сохраненной инструкцией SELECT, его создание начинается с проектирования этой инструкции. Инструкция SELECT, если она является корректной, может быть вырезана и вставлена в представление практически из любого инструмента.

В утилите Management Studio представления перечислены в собственном узле в каждой базе данных.



SQL Server Management Studio предлагает удобную среду для работы с представлениями, предлагая инструменты создания запросов, редактирования данных, сохранения графической структуры таблицы и многое другое.

Команда New View в контекстном меню позволит запустить конструктор запросов в режиме создания представлений (рис. 14.1).

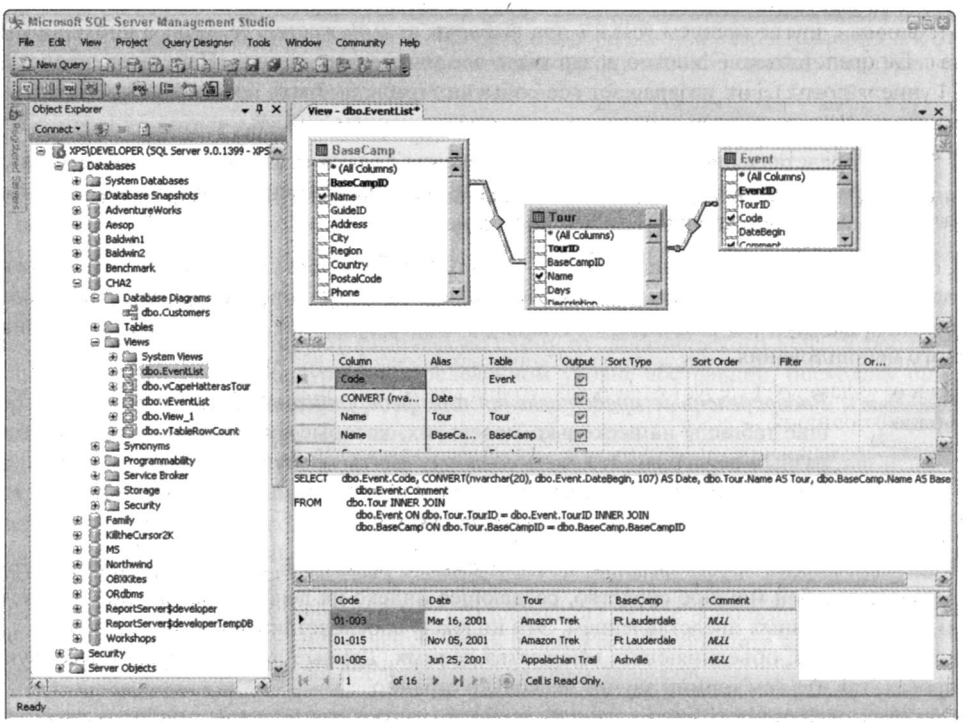


Рис. 14.1. Создание представления в конструкторе запросов утилиты Management Studio

Представления, реляционная алгебра, абстракция и безопасность

Одной из самых основных реляционных операций в создании проекций является способность представлять пользователю конкретные столбцы. Одним из основных достоинств приложений является их способность проецировать предопределенный набор столбцов. Вы можете проецировать столбцы, необходимые пользователям, и скрывать от них содержащие конфиденциальную или неподобающую информацию (например, информацию о зарплате и номерах кредитных карточек).

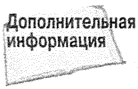
SQL Server поддерживает систему безопасности на уровне столбцов, и это очень мощный механизм. Проблема состоит в том, что разовые запросы, создаваемые конечными пользователями, не понимающими схему, очень часто вызывают ошибки безопасности. Рекомендуемое решение состоит во внедрении защиты на уровень абстракции, который изолирует физическую схему от доступа к данным. Этот уровень абстракции базы данных может быть создан с помощью представлений и хранимых процедур. Он может ограничить то, что может проектироваться во внешний мир, конкретным набором столбцов. Предоставление пользователям доступа к определенному множеству данных или возможности выполнения только представлений и хранимых процедур защищает от них данные в физических таблицах.

Более подробно об уровнях абстракции вы узнаете в главах 25 и 40, где рассматриваются вопросы конфигурирования безопасности объектов.

Конструктор представлений работает в конструкторе запросов Management Studio. Фактический код SQL отображается и редактируется на панели SQL. Столбцы в представление можно добавлять на панелях Diagram, Grid и SQL. Функция добавления таблиц доступна в контекстном меню, а также на панели инструментов. Здесь можно добавлять таблицы, другие представления, синонимы и табличные функции.

Таблицы и другие представления могут быть добавлены посредством перетаскивания их на панель Diagram из окна Object Explorer или с помощью пункта контекстного меню Add Table.

Функция добавления управляемых таблиц (Add Derived Table) способна добавить в предложение FROM представления в качестве источника данных подзапрос. Код SQL этого подзапроса можно ввести вручную на панели SQL.



Более подробно об использовании конструктора запросов см. в главе 6, посвященной работе с утилитой Management Studio.

Кнопка Verify SQL Syntax позволяет проверить синтаксис инструкций SQL. В то же время она не проверяет имена таблиц, столбцов и представлений в инструкции SELECT.

Кнопка Save панели инструментов запускает сценарий фактического создания представления в базе данных. Следует отметить, что для сохранения инструкция SELECT должна быть свободна от ошибок.

После создания представления его можно редактировать в Management Studio, выделяя название и выбирая в контекстном меню команду Modify View.

Для тестирования инструкции SELECT представления в конструкторе запросов щелкните на кнопке Execute SQL или нажмите клавишу <F5>.

Контекстное меню представления также содержит команды управления его полнотекстовой индексацией и его переименования. Свойства приложения содержат расширенные параметры и разрешения системы безопасности. Для удаления представления из базы данных достаточно выделить его, выбрать в контекстном меню команду Delete или нажать одноименную клавишу.

Создание представлений с помощью кода DDL

Представлениями можно управлять в редакторе запросов, выполняя сценарии SQL, которые используют команды языка DDL: CREATE, ALTER и DROP. Основной синтаксис создания представления следующий:

```
USE CHA2
CREATE dbo.имя_представления
AS
инструкция_SELECT
```

Например, чтобы создать представление vEventList программным путем, в окне запросов должны быть выполнены следующие команды:

```
CREATE VIEW dbo.vEventList
AS
SELECT dbo.CustomerType.Name AS Customer,
    dbo.Customer.LastName, dbo.Customer.FirstName,
    dbo.Customer.Nickname,
    dbo.Event_mm_Customer.ConfirmDate, dbo.Event.Code,
    dbo.Event.DateBegin, dbo.Tour.Name AS Tour,
    dbo.BaseCamp.Name, dbo.Event.Comment
    FROM dbo.Tour
      INNER JOIN dbo.Event
        ON dbo.Tour.TourID = dbo.Event.TourID
      INNER JOIN dbo.Event_mm_Customer
        ON dbo.Event.EventID = dbo.Event_mm_Customer.EventID
      INNER JOIN dbo.Customer
        ON dbo.Event_mm_Customer.CustomerID
           = dbo.Customer.CustomerID
      LEFT OUTER JOIN dbo.CustomerType
        ON dbo.Customer.CustomerTypeID
```

```
        = dbo.CustomerType.CustomerTypeID
    INNER JOIN dbo.BaseCamp
     ON dbo.Tour.BaseCampID = dbo.BaseCamp.BaseCampID
```

Попытка создать представление, которое уже существует, вызовет ошибку. Когда представление создано, инструкцию SELECT можно с легкостью отредактировать с помощью команды ALTER:

```
ALTER dbo.ViewName
AS
SQL Select Statement
```

Команда ALTER поставляет представлению новую инструкцию SELECT. Именно здесь вступает в игру Object Explorer. Чтобы автоматически сгенерировать инструкцию ALTER из существующего представления, в окне Object Explorer перейдите к списку представлений и выберите в контекстном меню команду Script View as⇨Alter to⇨New Query Editor Windows.

Если необходимо изменить представление, предпочтительнее удалить его и создать заново, поскольку удаление представления также приводит и к удалению разрешений доступа, установленных ранее.

Чтобы удалить представление из базы данных, используйте команду DROP:

```
DROP VIEW dbo.ViewName
```

В представлении, предназначенном для многократного запуска, следующий код удалит его и воссоздаст заново:

```
IF EXISTS(SELECT * FROM SysObjects WHERE Name = 'имя_представления')
  DROP VIEW dbo.имя_представления
CREATE dbo.имя_представления
AS
инструкция_SELECT
```

Предложение ORDER BY и представления

Представления служат источником данных для других запросов и не поддерживают сортировку внутри себя. Например, следующий код извлекает данные из представления vEventList и упорядочивает их по полям EventCode и name. Предложение ORDER BY не является частью представления vEventList, а применяется к нему с помощью вызова инструкции SQL:

```
SELECT EventCode, LastName, FirstName, IsNull(NickName,'')
  FROM dbo.vEventList
  ORDER BY EventCode, LastName, FirstName
```

В то же время синтаксис языка T-SQL допускает использование в представлениях предиката TOP, хотя последний бесполезен без предложения ORDER BY. В то же время если представление включает в себя предикат TOP 100 PERCENT, оно может включать и ORDER BY.

```
CREATE VIEW dbo.vCapeHatterasTour
  AS
    SELECT TOP 100 PERCENT TourName, BaseCampID
      FROM dbo.Tour
      ORDER BY TourName
```



В версии SQL Server 2000 включение в представление выражения TOP 100 PERCENT допускало также использование предложения ORDER BY. Однако в SQL Server 2005 эта ошибка была исправлена, и предложение ORDER BY существует только для поддержки предиката TOP. Предикат TOP без порядка сортировки будет возвращать только случайные строки и не будет иметь практического значения. Предикат TOP 100 PERCENT с предложением ORDER BY в SQL Server 2005 не будет сортировать данные в представлении.

Ограничения в представлениях

Несмотря на то что представление может содержать практически любую допустимую инструкцию SQL, существуют и некоторые ограничения.

- Представления не могут включать параметр `SELECT INTO`, который создает новую таблицу на основе выбранных столбцов. Инструкция `SELECT INTO` завершится ошибкой, если таблица уже существует, при этом оно не возвращает каких-либо значений. Таким образом, следующее представление не имеет места для существования:

  ```
  SELECT * INTO Table
  ```

- Представление не может обращаться к временной таблице (содержащей в имени знак решетки) или табличной переменной, поскольку такие таблицы недолговечны.
- Представления не могут содержать столбцы `COMPUTE` и `COMPUTE BY`. Вместо этого они могут использовать стандартные итоговые функции и группировку. (`COMPUTE` и `COMPUTE BY` считаются устаревшими и используются только из соображений обратной совместимости.)

Выполнение представлений

Представление не может быть выполнено само по себе. Инструкция `SELECT`, на основе которой создано представление, может быть выполнена, однако в этой форме, с технической стороны, инструкция SQL не является представлением. Инструкция SQL выполняется только один раз и сохраняется как представление. Представление может быть полезно только как источник данных в запросе.

Именно поэтому контекстное меню Open View утилиты Management Studio автоматически генерирует простой запрос, извлекая из представления все столбцы. Представление отображает только результаты. Однако включение других панелей конструктора запросов позволяет увидеть и сам запрос, извлеченный из представления.

Панель SQL отобразит представление в предложении `FROM` инструкции `SELECT`. Именно в такой форме на представление ссылаются пользователи:

```
SELECT *
  FROM vEventList
```

Когда представления вызываются из пользовательских приложений или из разовых запросов, условие `WHERE` обычно используется для извлечения из представления корректных данных. Условие `WHERE` может быть введено на панели `GRID` или `SQL`:

```
SELECT * FROM dbo.vEventList WHERE (EventCode = '101')
```

Защита представлений

Представления создаются для управления доступом к данным. Существует несколько параметров защиты данных и приложений.

Защита данных

Параметр `WITH CHECK OPTION` заставляет предложение `WHERE` представления проверять вставляемые, извлекаемые или обновляемые данные. В этом смысле он вставляет в предложение `WHERE` двустороннее ограничение.

Этот параметр может оказаться полезным, когда представление должно ограничить вставки.

Чтобы понять смысл использования параметра WITH CHECK OPTION, вначале важно уяснить, как функционируют представления при использовании этого параметра. Следующее представление генерирует список туров для базового лагеря Cape Hatteras:

```
ALTER VIEW dbo.vCapeHatterasTour
AS
SELECT TourName, BaseCampID
  FROM dbo.Tour
  WHERE BaseCampID = 2
SELECT * FROM dbo.vCapeHatterasTour
```

Результат будет следующий:

```
TourName                    BaseCampID
--------------------------- -----------
Outer Banks Lighthouses     2
```

Если базовый лагерь Ashville добавит тур Blue Ride Parkway Hike и вставит его в представление без параметра CHECK OPTION, вставка будет разрешена:

```
INSERT dbo.vCapeHatterasTour (TourName, BaseCampID)
  VALUES ('Blue Ridge Parkway Hike', 1)
(1 row(s) affected)
```

Инструкция INSERT сработает, новая строка появится в базе данных, но она не будет видимой в представлении, поскольку предложение WHERE представления фильтрует строки. Этот феномен известен как *исчезающие строки*:

```
SELECT * FROM dbo.vCapeHatterasTour

TourName                    BaseCampID
--------------------------- -----------
Outer Banks Lighthouses     2
```

Если целью данного представления было предоставление пользователям в лагере Cape Hatteras доступа только к собственным турам, то это представление некорректно. Несмотря на то что эти пользователи могут видеть только собственные туры, они могут с таким же успехом изменять данные и других лагерей.

Параметр WITH CHECK OPTION позволит избежать такого казуса. Следующий сценарий выполнит те же действия, но при этом сделает невозможной вставку записей, относящихся к другим лагерям. Обратите внимание на то, что на этот раз мы используем параметр WITH CHECK OPTION:

```
DELETE dbo.vCapeHatterasTour
  WHERE TourName = 'Blue Ridge Parkway Hike'
ALTER VIEW dbo.vCapeHatterasTour
  AS
  SELECT TourName, BaseCampID
    FROM dbo.Tour
    WHERE BaseCampID = 2
    WITH CHECK OPTION

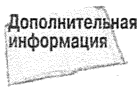
INSERT dbo.vCapeHatterasTour (TourName, BaseCampID)
  VALUES ('Blue Ridge Parkway Hike', 1)

Server: Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view
either specifies WITH CHECK OPTION or spans a view that
specifies WITH CHECK OPTION and one or more rows resulting from
```

```
the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

На этот раз инструкция INSERT не была выполнена, а сообщение об ошибке указывало на наличие в представлении параметра WITH CHECK OPTIONS, которое, собственно, позволило получить желаемый эффект.

Некоторые разработчики используют представления с параметром WITH CHECK OPTION как средство реализации системы безопасности на уровне строк. Такая методика называется *горизонтально позиционированными представлениями*. Как в продемонстрированном примере с базовыми лагерями, такие представления должны создаваться для каждого подразделения или отрасли, и они обеспечивают доступ пользователей только к относящимся к ним данным. Несмотря на то что этот метод помогает достичь нужного уровня безопасности, его обслуживание требует больших затрат.



В приложениях защита на уровне строк может устанавливаться с помощью пользовательских таблиц и хранимых процедур (подробнее об этом — в главе 27), однако представления могут установить такую защиту и для разовых запросов.

В конструкторе представлений утилиты Management Studio параметр WITH CHECK OPTION можно увидеть на странице View Properties.

Защита представлений

Три параметра защищают представления от изменения схемы и посторонних любопытных глаз. Эти параметры просто добавляются в инструкцию CREATE и применяются к представлению практически так же, как и WITH CHECK OPTION.

Изменения схемы

Код базы данных очень хрупкий и имеет тенденцию к разрушению при изменении структуры данных. Так как представления — это не что иное, как хранимые запросы SELECT, то изменения, внесенные в таблицы, к которым обращается инструкция, способны разрушить представление — даже в случае добавления новых столбцов.

Создание представления со связыванием схемы блокирует таблицы и не допускает изменений, как продемонстрировано в следующем примере:

```
CREATE TABLE Test (
     [Name] NVARCHAR(50)
)
go

CREATE VIEW vTest
WITH SCHEMABINDING
AS
SELECT [Name] FROM dbo.Test
Go

ALTER TABLE Test
    ALTER COLUMN [Name] NVARCHAR(100)

Server: Msg 4922, Level 16, State 1, Line 1
ALTER TABLE ALTER COLUMN Name failed
because one or more objects access this column.
```

К созданию представлений со связанной схемой предъявляются некоторые ограничения. Инструкция SELECT должна содержать имя владельца всех объектов, на которые она ссылается, также не допускается применение символа звездочки для извлечения всех столбцов.

В конструкторе запросов утилиты Management Studio параметр WITH SCHEMABINDING может быть активизирован на странице View Properties.

Шифрование инструкции SELECT представления

Параметр WITH ENCRYPTION — еще одно связанное средство защиты. Когда создаются представления или хранимые процедуры, их текст сохраняется в системной таблице SysComments. Таким образом, текст доступен для просмотра. Представление при этом может содержать предложение WHERE, которое желательно скрыть от посторонних глаз; могут существовать и другие причины шифрования кода. Параметр WITH ENCRYPTION шифрует код в таблице SysComments, таким образом, предохраняя его от чужих глаз.

В следующем примере текст представления проверяется в таблице SysComments, затем выполняется шифрование, после чего происходит повторная проверка кода. Как и ожидается, в конце концов инструкцию SELECT приложения прочитать больше невозможно:

```
SELECT Text
FROM SysComments
JOIN SysObjects
ON SysObjects.ID = SysComments.ID
WHERE Name = 'vTest'
```

Результатом выполнения инструкции является текст представления vTest:

```
Text
----------------------------
CREATE VIEW vTest
WITH SCHEMABINDING
AS
SELECT [Name] FROM dbo.Test
```

Следующая инструкция ALTER перестраивает представление путем шифрования:

```
ALTER VIEW vTest
WITH ENCRYPTION
AS
SELECT [Name] FROM dbo.Test
```

Будьте осторожны с использованием этого параметра. Как только код был зашифрован, анализатор запросов больше не сможет создать в нем исправления; вместо этого он сгенерирует сообщение:

```
/****** Encrypted object is not transferable,
and script cannot be generated. ******/
```

К тому же учтите, что шифрование влияет на репликацию. Зашифрованное представление не может быть опубликовано.

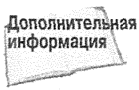
Метаданные приложения

Пользовательское приложение или уровень доступа к данным в запросе к серверу наряду с данными может запросить информацию о схеме, называемую *метаданными*. Обычно SQL Server возвращает информацию о соответствующих таблицах, однако параметр VIEW_METADATA указывает серверу возвращать информацию о схеме представлений, а не таблиц, к которым оно обращается. Это оградит слишком любопытных от изучения схемы таблиц, что особенно полезно, когда целью создания представления являлось сокрытие столбцов с конфиденциальной информацией.

Обновление информации с помощью представлений

Одним из самых главных недостатков представлений является то, что их редко можно использовать для обновления данных. На практике, если представление является чем-то большим, нежели простой инструкцией SELECT, велика вероятность, что данные с его помощью обновить будет невозможно.

Любая из следующих причин может сделать представление непригодным для обновлений.



На самом деле существуют и другие стандартные потенциальные трудности, связанные с обновлением и вставкой данных с помощью представлений. В главе 16 мы поговорим о них более подробно.

- Обновляться может только одна таблица. Если представление использует объединения, то инструкция UPDATE, ссылающаяся на представление, может обновить только одну из таблиц.
- Триггер INSTEAD OF в приложении или таблице, на которую то ссылается, будет изменять операцию модификации данных. Вместо требуемого изменения данных будет исполнен код триггера.
- Итоговые функции и группировка, используемые в представлении, сделают невозможным обновление. Дело в том, что SQL Server не сможет определить, какую из строк, участвовавших в консолидации, изменять.
- Если представление в качестве управляемой таблицы использует подзапрос, ни один из столбцов этой таблицы не может участвовать в выходных данных представления. В то же время в подзапросах, участвующих в качестве управляемых таблиц, допускается консолидация.
- Если представление создано с использованием параметра WITH CHECK OPTION, то операции вставки и удаления должны удовлетворять условиям предложения WHERE.
- Имена удаляемых и обновляемых столбцов должны быть уникальными. Если одно и то же имя встречается в нескольких таблицах, на которые ссылается приложение, то имя таблицы должно предшествовать имени столбца (*таблица.столбец*).

Как вы уже увидели, создать необновляемое представление крайне просто. Однако если проект использует представление для разовых запросов и отчетов, обновление данных может оказаться проблематичным.



Существует один способ обойти невозможность обновления в представлениях. Для этого нужно создать триггер INSTEAD OF, который инспектируемые данные, а затем выполняет допустимую инструкцию UPDATE на основе этих данных. В главе 23 мы обсудим создание триггеров INSTEAD OF.

Альтернативы представлениям

Если ваш стиль программирования предполагает создание множества представлений, то эта глава могла оставить у вас гнетущее впечатление. К счастью, SQL Server 2005 предлагает несколько других альтернатив.

В некоторых случаях хранимые процедуры и функции по производительности не могут сравниться с представлениями. К тому же хранимые процедуры не предлагают связывание схемы (а оно часто требуется), в то время как представления предлагают. В то же время поль-

зовательские функции объединяют довольно высокую скорость, возможность ввода параметров и связывание схемы. Если вам нравится создавать модульные инструкции SQL, такие как представления (как, в частности, и мне), то вам понравятся пользовательские функции.

В главах 18–25 мы подробно обсудим T-SQL, хранимые процедуры и функции.

Если вы используете представления для поддержки разовых запросов (что я и предполагаю), то можете предоставить кубы службы анализа тем пользователям, которым нужно выполнять сложные исследования данных. Кубы представляют собой предварительно подготовленные итоговые данные, которые можно рассмотреть с разных точек зрения (они называются *измерениями*). С точки зрения разработчика, создание одного куба часто заменяет собой разработку нескольких запросов или отчетов.

Подробно о создании кубов вы узнаете в главе 43.

Вопросы производительности представлений часто связаны с необходимостью блокировки данных. Ничего страшного в блокировках нет. Если данные извлекаются через представление и инструкция SELECT немедленно выполняется, то блокировка сразу же снимается. Проблема возникает, когда пользователи открывают все множество данных в приложениях, и эти данные остаются открытыми на протяжении всей сессии их просмотра. По этой причине представления незаслуженно получили репутацию черных дыр блокировки. На самом деле проблему создают не сами представления, а программный код или используемые средства. Я решил написать эту небольшую ремарку в защиту представлений и предупредить о потенциальной возможности уменьшения производительности.

Вложенные представления

Так как представления являются не более чем инструкциями SELECT, а инструкция SELECT может сама ссылаться на представление как на источник данных, одно представление может ссылаться на другое. Такие представления иногда называют *вложенными*.

Следующее представление использует представление vEventList и добавляет предложение WHERE для ограничения результатов теми событиями, которые будут иметь место в последующие 30 дней:

```
CREATE VIEW dbo.vEventList30days
  AS
  SELECT dbo.vEventList.EventCode, LastName, FirstName
    FROM dbo.vEventList
    JOIN dbo.Event
      ON vEventList.EventCode = Event.EventCode
    WHERE Event.DateBegin
      BETWEEN GETDATE() and GETDATE() + 30
```

В данном примере представление vEventList вложено в vEventList30Days. Еще один способ выразить отношение — это определить зависимость представления vEvent30Days от vEventList. (В утилите Management Studio зависимости объектов можно просмотреть после выбора в контекстном меню объекта пункта All Tasks⇨Display Dependencies.) На рис. 14.2 показаны зависимости и вложенное представление.



Еще одним типом высокотехнологичных зависимостей является *разделенное представление*. Оно объединяет данные, разбитые на несколько сегментированных таблиц, из соображений повышения производительности. Разделенные представления мы рассмотрим в главе 53.

Представления являются не единственным средством вложения инструкций SELECT. Подзапросы и общие табличные выражения поставляют данные в виде таблиц или управляемых

таблиц, которые, в свою очередь, могут быть вложены в запрос. В этом случае весь код рассматривается как единый компонент, что повышает производительность. Вложенные представления в приведенном выше примере можно переписать как вложенные управляемые таблицы (подзапрос в этом коде заключен в скобки).

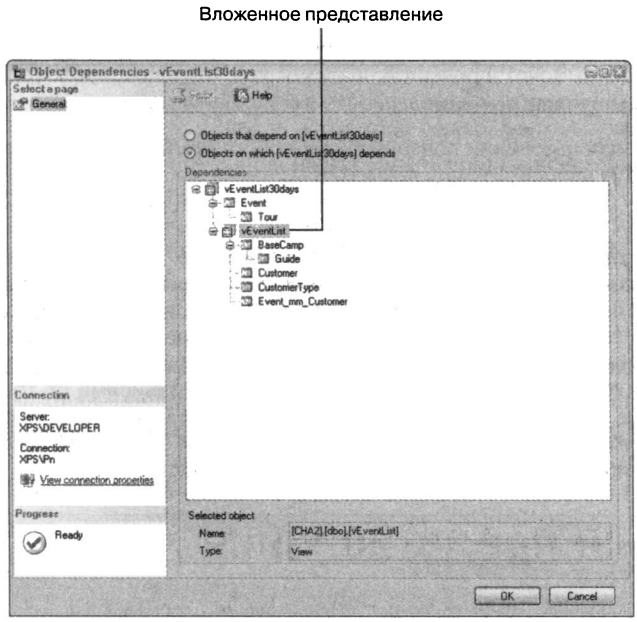
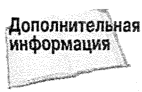


Рис. 14.2. Цепочка зависимостей легко распознается в диалоговом окне **Dependencies**. *В данном примере мы видим, что представление* `vEventList30Days` *включает в себя вложенное представление* `vEventList`

```
SELECT E.EventCode, LastName, FirstName
 FROM
  (SELECT dbo.CustomerType.CustomerTypeName,
         dbo.Customer.LastName, dbo.Customer.FirstName,
         dbo.Customer.Nickname,
         dbo.Event_mm_Customer.ConfirmDate, dbo.Event.EventCode,
         dbo.Event.DateBegin, dbo.Tour.TourName,
         dbo.BaseCamp.BaseCampName, dbo.Event.Comment
    FROM dbo.Tour
    INNER JOIN dbo.Event
     ON dbo.Tour.TourID = dbo.Event.TourID
    INNER JOIN dbo.Event_mm_Customer
     ON dbo.Event.EventID = dbo.Event_mm_Customer.EventID
    INNER JOIN dbo.Customer
     ON dbo.Event_mm_Customer.CustomerID
      = dbo.Customer.CustomerID
    LEFT OUTER JOIN dbo.CustomerType
      ON dbo.Customer.CustomerTypeID
     = dbo.CustomerType.CustomerTypeID
   INNER JOIN dbo.BaseCamp
    ON dbo.Tour.BaseCampID = dbo.BaseCamp.BaseCampID
   ) E
```

```
JOIN dbo.Event
   ON E.EventCode = Event.EventCode
WHERE Event.DateBegin BETWEEN GETDATE()
    and GETDATE() + 30
```

Данному подзапросу присваивается псевдоним E; по этому имени к нему выполняется обращение во внешнем запросе. Следует отметить, что эту методику нельзя назвать подходящей для конечных пользователей. Однако если вы разработчик, который использует вложенные представления, но хочет прояснить и сжать код, то вложенные управляемые таблицы представляют собой лучшее решение.



Об использовании подзапросов и общих табличных элементов (CTE) см. в главе 10.



Тестирование производительности показало, что когда данные и план выполнения находятся в кэше, хранимые процедуры выигрывают в производительности перед представлениями. В то же время, если план выполнения не кэширован (т.е. недавно не выполнялся), то представления выполняются быстрее хранимых процедур. Как представления, так и хранимые процедуры проявили себя лучше, чем обычная инструкция SELECT, независимо от того, что находится в кэше. Это доказывает то, что практика использования представлений для поддержки разовых запросов пользователей и хранимых процедур для данных приложений подтвердила свою состоятельность.

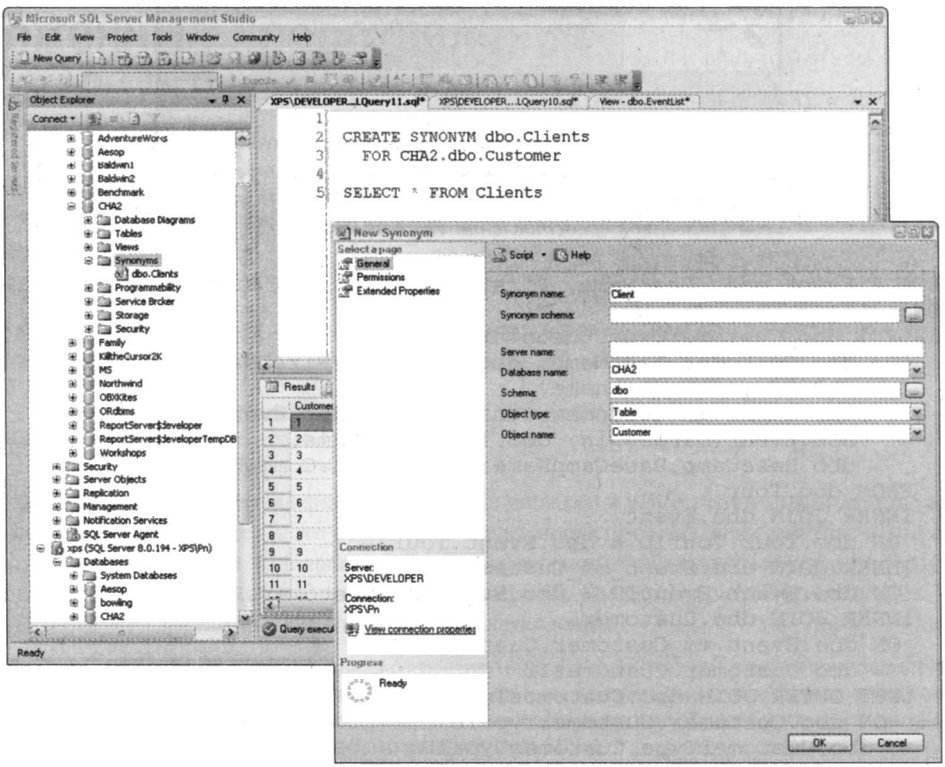


Рис. 14.3. Синонимы можно создавать и настраивать в окне Object Explorer утилиты Management Studio

Использование синонимов

Представления иногда используют для сокрытия зашифрованных имен схемы базы данных. *Синонимы* аналогичны представлениям, однако более ограничены. В то время как представления могут проецировать столбцы и присваивать им псевдонимы, а также группировать данные с помощью объединений и подзапросов, синонимы могут присваивать альтернативные имена таблицам, представлениям и хранимым процедурам (см. рис. 14.3).

Хотя синонимы и несколько ограничены по сравнению с представлениями, они имеют определенное практическое применение — изменение схемы объектов. В базах данных, схема которых интенсивно используется пользователями, отличными от `dbo`, синонимы могут переопределить принадлежность объектов пользователю `dbo` и облегчить таким образом создание запросов.

Резюме

Представления являются ничем иным, как сохраненными запросами `SELECT`. В этом нет ничего магического. Любая допустимая инструкция `SELECT` может быть сохранена как представление, включая свои сложные объединения, подзапросы и итоговые функции.

В предыдущих главах мы обсуждали вопросы извлечения данных с помощью мощной инструкции `SELECT`. В следующей главе мы продолжим изучение этой инструкции, отойдем от локальных баз данных и начнем использовать распределенные запросы.

ГЛАВА

15

Работа с распределенными запросами

В этой главе...

Концепции распределенных запросов

Установка подключения к удаленным источникам данных

Распределенные запросы T-SQL

Сквозные запросы

Двухфазное подтверждение и распределенные транзакции

Данные редко находятся в одном месте. В современном мире большинство новых проектов подключается к существующим источникам данных или, по крайней мере, расширяется за счет них. Вообще-то, это не является проблемы. SQL Server позволяет считывать данные из множества разных источников и записывать их туда. Гетерогенные объединения даже позволяют объединять данные SQL Server и электронных таблиц Excel.

SQL Server предлагает несколько различных методов доступа к внешним по отношению к текущей базе данным. От простой ссылки на другую локальную базу данных до выполнения сквозных запросов, которые задействуют другую базу данных с архитектурой "клиент/сервер", — все это позволяет осуществить SQL Server.


Проверено

Несмотря на то что SQL Server решает технические проблемы запросов к внешним данным, если две системы на самом деле обслуживаются разными приложениями, то прямое обращение к внешнему хранилищу данных в большинстве случаев нарушает принцип инкапсуляции. Таким образом, одновременное использование двух источников данных снижает гибкость архитектуры. Во многих программистских фирмах эта технология не находит поддержки. Вместо этого обмен данными осуществляется с помощью XML, SOAP и SOA, о которых мы говорили в главе 1 и еще раз к ним вернемся в главе 32.

Основные концепции распределенных запросов

Связывание с внешними источниками данных представляет собой конфигурирование имени подключаемого сервера, а также реквизитов входа на него, что позволяет получить доступ к его данным.

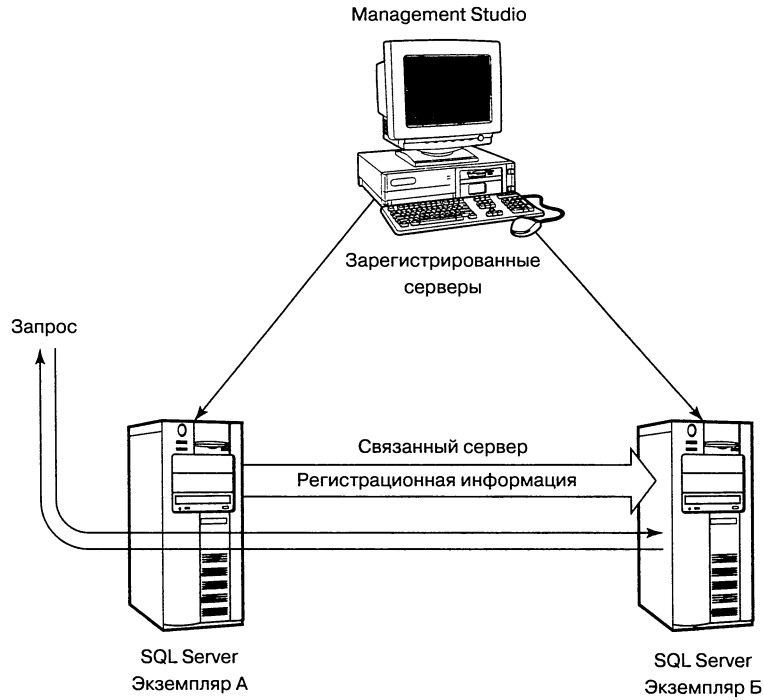


Рис. 15.1. Связывание серверов представляет собой одностороннее подключение и не зависит от регистрации серверов в Management Studio. На данной диаграмме экземпляр А видит экземпляр Б как связанный сервер, поэтому сервер А может получить доступ к данным сервера Б

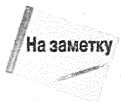
Связывание — это одностороннее конфигурирование (рис.15.1). Если сервер А подключается к серверу Б, то это значит, что первый из них знает, как получить доступ к второму. Все время, пока сервер Б является доступным, сервер А для него является обычным пользователем.

Если связывание серверов является для вас новой концепцией, вы можете легко ее перепутать с регистрацией серверов в Management Studio. Как показано на рис. 15.1, Management Studio связывается с серверами как обычное клиентское приложение. В противоположность этому связывание серверов позволяет серверу А напрямую обращаться к серверу Б.

Связи могут быть установлены как в утилите Management Studio, так и в коде T-SQL. Последний метод имеет преимущество возможности повторения подключения, если соединение будет неожиданно разорвано. В то же время создание подключения в коде T-SQL требует большей работы.

Связываемый сервер может быть экземпляром SQL Server, либо другим источником данных с провайдером OLE DB, либо с драйвером ODBC. Распределенные запросы могут как извлекать данные, так и изменять их с помощью инструкций `INSERT`, `UPDATE` и `DELETE` (разумеется, в соответствии с требованиями провайдера OLE DB или драйвера ODBC).

Запросы SQL Server могут ссылаться на внешние данные с помощью обращения к предварительно сконфигурированному и подключенному серверу или определять подключение в самом коде запроса.



В этой главе будем называть рассматриваемые источники данных *локальным* и *внешним*. В других документах вы можете встретить такие определения, как *локальный* и *удаленный*, а также *отправляющий* и *принимающий*.

В некотором смысле подключение к внешнему источнику данных всего лишь перемещает объявление связи от кода запроса к заданию администрирования сервера. Так как запросы могут обращаться к именованным связям, не зная о реальном местонахождении сервера и регистрационных данных, запросы, использующие подключенные серверы, более портативные и легкие в обслуживании, чем те, которые определяют подключение в коде запроса. Если база данных перемещается на новый сервер, то, как только администратор создаст соответствующее подключение, запросы будут работать без какого-либо видоизменения.

В случае распределенного запроса SQL Server является клиентским процессом, получающим результаты от внешнего источника данных. Распределенные запросы могут либо извлекать данные в SQL Server для обработки, либо передавать запрос внешнему источнику данных.



Существует множество способов распределения данных. Вы можете использовать репликацию (подробнее об этом — в главе 39) либо настраивать отдельный компьютер как сервер отчетности (подробнее об этом — в главе 52).

Доступ к базе данных локального сервера

Когда осуществляется доступ к другой базе данных того же сервера, данные обрабатывает одно и то же ядро SQL Server. Таким образом, несмотря на то, что данные находятся вне текущей базы данных, запрос на самом деле не является распределенным.

Запрос SQL Server может осуществлять доступ к другой базе данных на том же сервере, обращаясь к таблице с использованием имени базы данных:

```
База_данных.Схема.Имя_Объекта
```

Поскольку база данных находится на том же сервере, его имя использовать не обязательно. Имя схемы может быть заменено именем владельца объекта. Обычно таблицы принадлежат владельцу схемы (dbo). Если это на самом деле так, то часть dbo можно опустить:

```
USE CHA2;
SELECT LastName, FirstName
FROM OBXKites.dbo.Contact;
```



Все листинги, приведенные в настоящей главе, находятся в файле `ch15.sql`. Также существует сценарий преобразования базы данных Cape Hatteras Adventures `CHA2_Convert.sql`, использующий распределенные запросы для перемещения данных из Access и Excel в SQL Server.

Приведенный выше запрос является функциональным эквивалентом следующего:

```
SELECT LastName, FirstName
  FROM OBXKites..Contact;
```

Будет получен следующий результат (в сокращенном виде):

```
LastName      FirstName
------------- ------------
Adams         Terri
Andrews       Ed
...
```

Подключение к внешним источникам данных

SQL Server также способен устанавливать подключение к любому совместимому с OLE DB или ODBC источнику данных. Эта связь может быть установлена как в Management Studio, так и с помощью кода SQL.

Подключение в Management Studio

Подключение к другому серверу может быть установлено с помощью Management Studio или программного кода. В Management Studio связанные серверы перечислены в узле `Security` — здесь в каждой связи уже определено, как подключиться к удаленному серверу и зарегистрироваться в нем. Щелкните правой кнопкой мыши на узле `Security` под именем сервера и выберите пункт New Linked Server. Откроется форма свойств связанного сервера, показанная на рис. 15.2.

Подключение к источникам данных, отличным от SQL Server, описывается далее в этой главе.

Выбор сервера

Во вкладке General формы Linked Server Property введите имя внешнего сервера и установите переключатель в положение SQL Server. Для подключения к именованному экземпляру SQL Server введите это имя в формате *сервер*`\`*экземпляр* без квадратных скобок. На рис. 15.2 подключаемым сервером является `Noli\SQL2`.

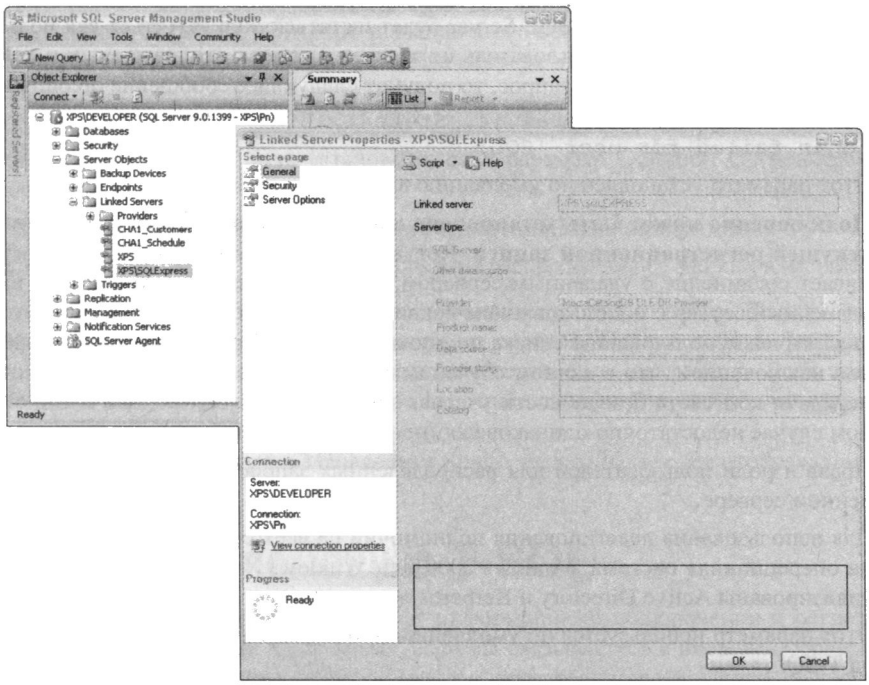


Рис. 15.2. Форма свойств связанного сервера

SQL Server 2005 может подключиться к любому экземпляру SQL Server 2000, SQL Server 7, но не может подключиться к SQL Server 6.5 без использования драйвера ODBC.

Конфигурирование регистрационных данных

Основной целью связывания серверов является предоставление возможности локальным пользователям запускать запросы, работающие с информацией других источников данных. Если внешний источник данных является экземпляром SQL Server, это потребует некоторого типа аутентификации, что, в свою очередь, требует привязки регистрационных данных. Для тех пользователей, чьи регистрационные данные не удалось связать, используются условия, действующие по умолчанию.

Привязка регистрационных данных предполагает либо преобразование имен, если установлен параметр Impersonate, либо предоставление регистрационных данных другого сервера. Естественно, на удаленном сервере имя пользователя должно быть допустимым и наделенным достаточными полномочиями для выполнения операции.

Параметры, применяемые по умолчанию к пользователям, не отображенным в списке, приведены ниже.

- **Подключение не может быть установлено**. Ограничение возможности запускать распределенные запросы для тех, кто включен в список отображаемых пользователей. Если не включенные в этот список пользователи пытаются подключиться к серверу, они получат следующее сообщение об ошибке:

```
Server: Msg 7416, Level 16, State 1, Line 1
Access to the remote server is denied
   because no login-mapping exists.
```

- **Подключение может быть установлено без использования контекста безопасности**. Этот параметр устанавливается на серверах, отличных от SQL Server, и на последнем не имеет смысла. SQL Server будет пытаться подключиться как пользователь без пароля. Если такой пользователь не занесен в список отображаемых и попытается запустить распределенный запрос, то получит следующее сообщение об ошибке:

```
Server: Msg 18456, Level 14, State 1, Line 1
Login failed for user 'SQL'.
```

  Этот параметр установлен по умолчанию в Management Studio.

- **Подключение может быть установлено с использованием контекста безопасности текущей регистрационной записи**. Когда локальный экземпляр SQL Server устанавливает соединение с удаленным сервером, он делегирует полномочия (т.е. входит на удаленный сервер с использованием реквизитов текущего пользователя). Этот метод аналогичен использованию списка пользователей и выбору параметра Impersonate, за тем исключением, что в данном случае используется делегирование полномочий. Для передачи контекста безопасности учетная запись должна быть одной и той же; в данном случае недостаточно одинакового имени пользователя и пароля.

  Права и роли пользователей для распределенных запросов будут назначаться на удаленном сервере.

  Для использования делегирования полномочий на всех серверах должна быть запущена операционная система Windows 2000 или Windows XP; при этом должны быть активизированы Active Directory и Kerberos.

  Этот параметр используется по умолчанию при создании подключения с помощью кода T-SQL.



В большинстве распределенных запросов от SQL Server к SQL Server контекст безопасности текущей учетной записи является лучшим параметром подключения, поскольку сохраняет идентичность пользователя и подтверждает план безопасности. Если инфраструктура не поддерживает Active Directory и Kerberos, лучше отображать список пользователей.

- **Подключение может быть установлено с использованием текущего контекста безопасности**. Последний параметр назначает каждому пользователю, не отображенному в списке, заранее запрограммированную учетную запись внешнего сервера. Хотя этот метод может показаться простейшим, он предоставляет всем локальным пользователям одинаковый доступ к серверу. Использование этого параметра может разрушить любой стоящий план защиты информации, поскольку не позволяет внешнему экземпляру SQL Server достичь уровня защиты C2.

Конфигурирование параметров

На третьей вкладке формы связанного сервера, Server Options, доступны следующие параметры, управляющие тем, как SQL Server ожидает получения данных от удаленного сервера.

- Collation Compatibility. Установите этот параметр в истинное значение (`true`), если два сервера используют одинаковый набор символов и порядок сопоставления.
- Data Access. Если этот параметр установлен в ложное значение (`false`), то он запрещает распределенные запросы к удаленному серверу.
- RPC. Если этот параметр установлен в истинное значение, то к внешнему серверу могут быть выполнены вызовы удаленных процедур.
- RPC Out. Если этот параметр имеет истинное значение, то вызовы удаленных процедур разрешено принимать от внешнего сервера.
- Use Remote Collation. Истинное значение этого параметра определяет использование набора символов и их порядка внешнего сервера вместо существующих на текущем сервере.
- Collation Name. Определяет порядок сопоставления для распределенных запросов. Этот параметр может быть установлен, если параметр Collation Compatibility имеет значение `true`.
- Connection Timeout. Время ожидания соединения в миллисекундах.
- Query Timeout. Время ожидания выполнения распределенного запроса в миллисекундах.

Как только соединение будет корректно установлено, скорее всего, станет доступным список таблиц в соответствующем узле внешнего сервера. Перечисленные таблицы будут принадлежать базе данных, принятой на внешнем сервере по умолчанию. Если этой базой данных является `master`, а утилита Management Studio сконфигурирована для сокрытия системных объектов, то никаких таблиц вы не увидите.

Удаление связанного сервера в Management Studio также удаляет все отображения регистрационных записей.

Подключение с помощью T-SQL

Утилита Management Studio обрабатывает информацию о подключении и регистрационной записи в единой форме. В то же время, если вы связываетесь с внешним сервером с помощью кода SQL, подключение и информация о регистрационной записи обслуживаются разными командами.

Установка подключения

Для установки подключения к внешнему серверу с помощью программного кода используется системная хранимая процедура `sp_addlinkedserver`. Если соединение к удаленному серверу было установлено и имя экземпляра этого сервера доступно как имя подключения, то требуются всего два параметра: имя внешнего сервера и серверный продукт. Следующая команда создает подключение к экземпляру `SQL2` на моем сервере тестирования (`[XPS\Developer]`):

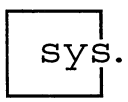
```
-- Примечание: сервер разработки автора называется XPS
-- Экземпляры этого сервера:
-- [XPS] SQL Server 2000 Developer Edition
-- [XPS\Developer] SQL Server 2005 Developer Edition
-- [XPS\SQLExpress] SQL Server 2005 Express Edition
-- [XPS\Standard] SQL Server 2005 Standard Edition
EXEC sp_addlinkedserver
@server = 'XPS\SQLExpress',
@srvproduct = 'SQL Server';
```



Если вы собираетесь запустить этот сценарий, вам потребуется изменить имя сервера и его экземпляра, чтобы они соответствовали вашей конфигурации.

Для подключения к экземпляру сервера с использованием имени, отличного от реального имени экземпляра, в инструкцию добавляются два параметра. Первый параметр, `provider`, должен иметь значение `SQLOLEDB`, а параметр `@datasrc` (источник данных) передает реальное имя экземпляра SQL Server. Параметр `@srvproduct` (серверный продукт) остается пустым. Параметр `@server` должен содержать имя связанного сервера, которое должно быть известно. В следующем примере показано, как выполняется подключение к экземпляру `SQL2` сервера `Noli`, однако в запросах этот сервер будет упоминаться как `Yonder`:

```
EXEC sp_addlinkedserver
  @server = 'Yonder',
  @datasrc = 'Noli\SQL2',
  @srvproduct = '',
  @provider='SQLOLEDB';
```



Представление каталога `sys.servers` перечисляет все серверы, включая подключенные. Системная хранимая процедура `sp_linkedservers` также возвращает информацию обо всех подключенных серверах.

```
SELECT [Имя], Продукт, Провайдер, Источник_данных
  FROM sys.servers
  WHERE Is_Linked = 1;
```

Для удаления существующего подключения к серверу (не влияющего на сам внешний сервер) используется системная хранимая процедура `sp_dropserver`:

```
EXEC sp_DropServer @server = 'Yonder';
```

Если для связанного сервера существуют какие-либо ограничения, то они также удаляются.

Распределенная защита и регистрация

В утилите Management Studio вопрос безопасности разбит на две части: отображение регистрационных данных, режим работы с не отображенными регистрациями. В T-SQL для решения обоих вопросов используется системная хранимая процедура `sp_addlinkedsrvlogin`:

```
sp_addlinkedsrvlogin
  @rmtsrvname = 'имя_удаленного_сервера',
  @useself = 'useself', (по умолчанию True)
  @locallogin = 'локальная_учетная_запись', (по умолчанию Null)
  @rmtuser = 'удаленный_пользователь', (по умолчанию Null)
  @rmtpassword = 'удаленный_пароль' (по умолчанию Null);
```

Если связанный сервер был добавлен с помощью кода T-SQL, а не средствами Management Studio, то параметры безопасности для не отображенных регистраций уже сконфигурированы для использования текущего контекста безопасности.

Если параметр `@locallogin` имеет пустое значение `NULL`, то параметры применяются ко всем пользователям, не отображенным в списке. Параметр `@useself` идентичен параметру `Impersonate`, о котором мы говорили ранее.

Следующая хранимая процедура использует учетную запись `Noli\Paul` для доступа к серверу `Noli\SQL2` под именем `sa` и с паролем `secret`:

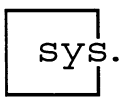
```
sp_addlinkedsrvlogin
  @rmtsrvname = 'XPS\SQLExpress',
  @useself = 'false',
  @locallogin = 'NOLI\Paul',
  @rmtuser = 'sa',
  @rmtpassword = 'secret';
```

В следующем примере все не отображенные в списке пользователи настраиваются для подключения с использованием собственного контекста безопасности (рекомендуемый параметр). Имя локального пользователя равно `NULL`, так что эта регистрация на внешнем сервере применяется ко всем пользователям, не отображенным в списке. Параметр `@useself` не определяется, так что используется его значение, принятое по умолчанию, — `true`. Это значит, что все пользователи будут использовать текущий контекст безопасности:

```
EXEC sp_addlinkedsrvlogin
  @rmtsrvname = 'NOLI\SQL2';
```

В третьем примере мы запретим всем не отображенным в списке пользователям выполнение распределенных запросов. В нем второй параметр, `@useself`, установлен в значение `false`, а регистрационное имя и пароль отображенных пользователей равны пустым значениям (`NULL`):

```
EXEC sp_addlinkedsrvlogin 'NOLI\SQL2', 'false';
```



Представление каталога `sys.Linked_Logins` перечисляет регистрационные записи. Системная хранимая процедура `sp_helplinkedsrvlogin` также возвращает информацию о подключенных регистрационных записях:

```
SELECT [Имя], Продукт, Провайдер, Источник_данных
  FROM sys.servers
  WHERE Is_Linked = 1;
```

Для сброса подключения к связанному серверу используется системная хранимая процедура `sp_droplinkedsrvlogin`:

```
sp_droplinkedsrvlogin
@rmtsrvname = 'имя_удаленного_сервера', (нет умолчаний)
@locallogin = 'локальная_учетная_запись' (нет умолчаний);
```

В следующем примере мы удалим регистрационную запись `Noli\Paul`, отображенную на `Noli\SQL2`:

```
EXEC sp_droplinkedsrvlogin
  @rmtsrvname = 'XPS\SQLExpress',
  @locallogin = 'NOLI\Paul';
```

Для удаления отображения, применяемого к не отображаемым пользователям, запустите ту же процедуру, только задайте пустую локальную учетную запись (NULL):

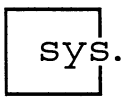
```
EXEC sp_droplinkedsrvlogin 'XPS\SQLExpress', NULL;
```

Параметры связанного сервера

Параметры связанного сервера, перечисленные во вкладке Server Options формы Linked Server Options, можно установить и программным путем с помощью системной хранимой процедуры `sp_serveroption`. Эта процедура должна вызываться для каждого из устанавливаемых параметров:

```
sp_serveroption
  @server = 'сервер',
  @optname = 'имя_параметра',
  @optvalue = 'значение_параметра';
```

Параметры в этом примере те же, что и в форме, только с дополнением параметра `lazy schema validation`, который отключает проверку схемы таблиц на распределенные запросы. Вы можете его использовать, когда уверены в безопасности схемы, но хотите уменьшить сетевую нагрузку.



Представление каталога `sys.servers` возвращает параметры связанного сервера. Системная хранимая процедура `sp_helpserver` также возвращает информацию о связанных серверах:

```
SELECT [Имя], Продукт, Провайдер, Источник_данных
  FROM sys.servers
  WHERE Is_Linked = 1;
```

Подключение к источникам данных, отличным от SQL Server

Если внешний источник данных — не SQL Server, вы все равно имеете возможность доступа к данным. Все зависит от доступности и функций драйверов ODBC или поставщиков OLE DB. SQL Server для доступа к внешним данным использует механизм OLE DB, и некоторые его компоненты входят в комплект сервера. Если по какой-либо причине OLE DB не доступен для некоторого внешнего источника данных, используйте поставщик `Microsoft OLE DB Provider for ODBC Drivers`. Практически любой тип источника данных имеет драйвер ODBC.

Для установки подключения к серверу либо в Management Studio, либо программным путем строке подключения требуется дополнительная информация (кроме имени подключаемого сервера, провайдера и имени продукта). Некоторые настройки распространенных источников данных приведены в табл. 15.1.

В качестве примеров подключения к источникам данных, отличным от SQL Server, мы с помощью распределенных запросов пополним учебную базу данных Cape Hateras Adventures информацией из Access и Excel. Данная учебная база данных моделирует типичный малый бизнес, который в настоящее время использует Access и Excel для хранения списка клиентов и расписания.

Таблица 15.1. Настройки подключения к другим источникам данных

Подключение к...	Имя провайдера	Продукт	Источник данных	Строка провайдера
MS Access	MS Jet 4.0 OLE DB	Access 2003	Местоположение файла базы данных	`null`
Excel	MS Jet 4.0 OLE DB	Excel	Местоположение файла с рабочим листом	`Excel 5.0`
Oracle	MS OLE Provider for Oracle	Oracle	Системный идентификатор Oracle	`null`

Подключение к Excel

Пример, использованный в настоящем разделе, можно непосредственно взять из сценария `CHA2_Convert.sql`. Он перемещает данные из старой версии (Access и Excel) в новую (SQL Server). Сотрудники компании ранее хранили расписание туров в Excel (рис. 15.3).

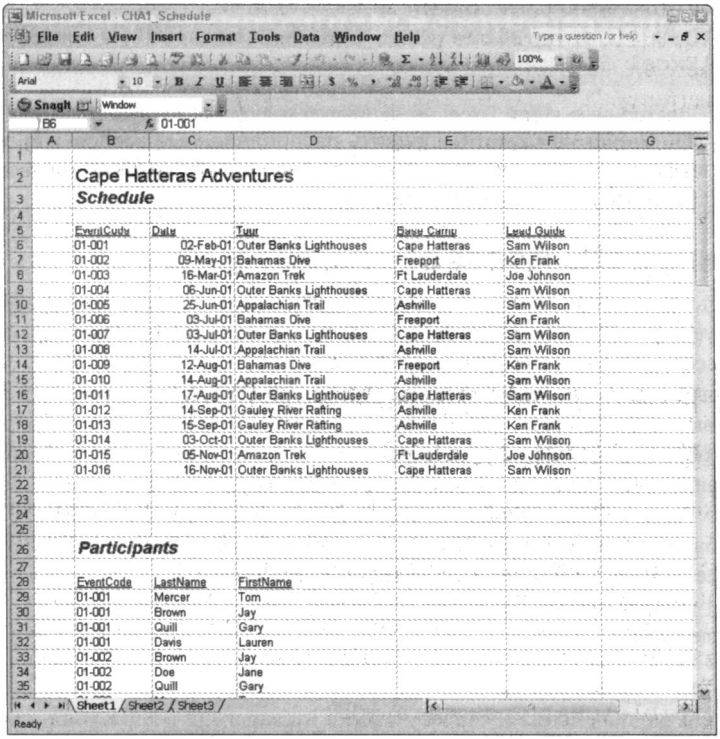


Рис. 15.3. Перед переходом на платформу SQL Server компания Cape Hatteras Adventures обслуживала график своих туров в электронной таблице `CHA1_Schedule.xls`

При работе с Excel каждая электронная страница или именованный диапазон книги появляется в SQL Server в виде таблицы, когда доступ осуществляется с помощью провайдера. В Excel именованные диапазоны определяются с помощью команды меню Insert⇨Name⇨Define. Для создания нового именованного диапазона и редактирования существующих используется

диалоговое окно определения имени. Рабочая книга CHA1_Schedule имеет пять именованных диапазонов (рис. 15.4), которые выглядят практически так же, как представления SQL Server. Каждый из пяти именованных диапазонов рабочей книги при подключении появляется как таблица. SQL Server при этом может использовать стандартные инструкции SELECT, INSERT, UPDATE и DELETE, как при обращении к обычной собственной таблице.

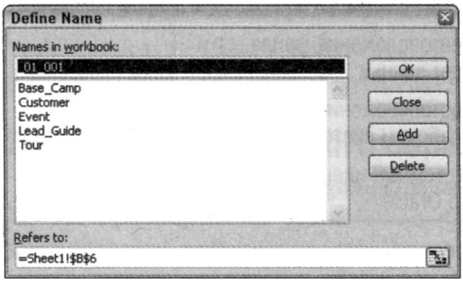


Рис. 15.4. Пять таблиц определены в Excel как именованные диапазоны

Следующий пример кода SQL настраивает рабочую книгу Excel как связанный сервер:

```
Execute sp_addlinkedserver
@server = 'CHA1_Schedule',
@srvproduct = 'Excel',
@provider = 'Microsoft.Jet.OLEDB.4.0',
@datasrc = 'C:\SQLServerBible\CHA1_Schedule.xls',
@provstr = 'Excel 5.0'
```



Электронные книги Excel не являются многопользовательскими. По этой причине SQL Server не может выполнить распределенный запрос, в котором участвует книга Excel, пока она открыта каким-либо пользователем.

Подключение к MS Access

Неудивительно, что SQL Server без труда подключается к базам данных MS Access. SQL Server использует провайдер OLE DB Jet для доступа к механизму Jet, который задействуется программой Access для доступа к данным в файлах .mdb.

Так как программа Access сама представляет собой СУБД, не существует никаких хитростей в подготовке ее баз данных, как это было в случае с Excel. Каждая таблица базы данных Access будет отображена в виде таблицы в узле Linked Servers утилиты Management Studio.

Согласно нашему сценарию, до перехода на платформу SQL Server список клиентов хранился в базе данных Access. Следующий код, который можно взять из файла CHA2_Convert.sql, связывается с базой данных Access CHA1_Customers.mdb, чтобы СУБД SQL Server могла запросить из нее данные и заполнить собственные таблицы:

```
EXEC sp_addlinkedserver
'CHA1_Customers',
'Access 2003',
'Microsoft.Jet.OLEDB.4.0',
'C:\SQLServerBible\CHA1_Customers.mdb';
```

Если у вас появились сложности при работе с распределенными запросами, в первую очередь проверьте контекст безопасности. Access не ожидает установки какого-либо контекста безопасности, поэтому не отображенные регистрационные записи будут установлены в незащищенный контекст.

```
EXEC sp_addlinkedsrvlogin
@rmtsrvname = 'CHA1_Schedule',
@useself = 'false';
```

Проектирование распределенных запросов

После того как подключение к удаленному источнику данных установлено, SQL Server может обращаться к внешним данным с помощью запросов. В табл. 15.2 представлены четыре основных доступных синтаксических метода, которые отличаются методами обработки и настройки.

Таблица 15.2. Матрица методов распределенных запросов

Настройка подключения	Место выполнения запроса	
	Локальный экземпляр SQL Server	Внешний источник данных
Связанный сервер	Четырехкомпонентное имя	Четырехкомпонентное имя `OpenQuery()`
Подключение, определенное в запросе	`OpenDataSource()`	`OpenRowSet()`

Распределенные запросы и Management Studio

Утилита Management Studio не поддерживает графические методы инициализации распределенных запросов. Не существует методов перетаскивания подключенного сервера или удаленной таблицы в конструктор запросов. Однако распределенный запрос можно ввести вручную на панели SQL (рис. 15.5), а затем выполнить его.

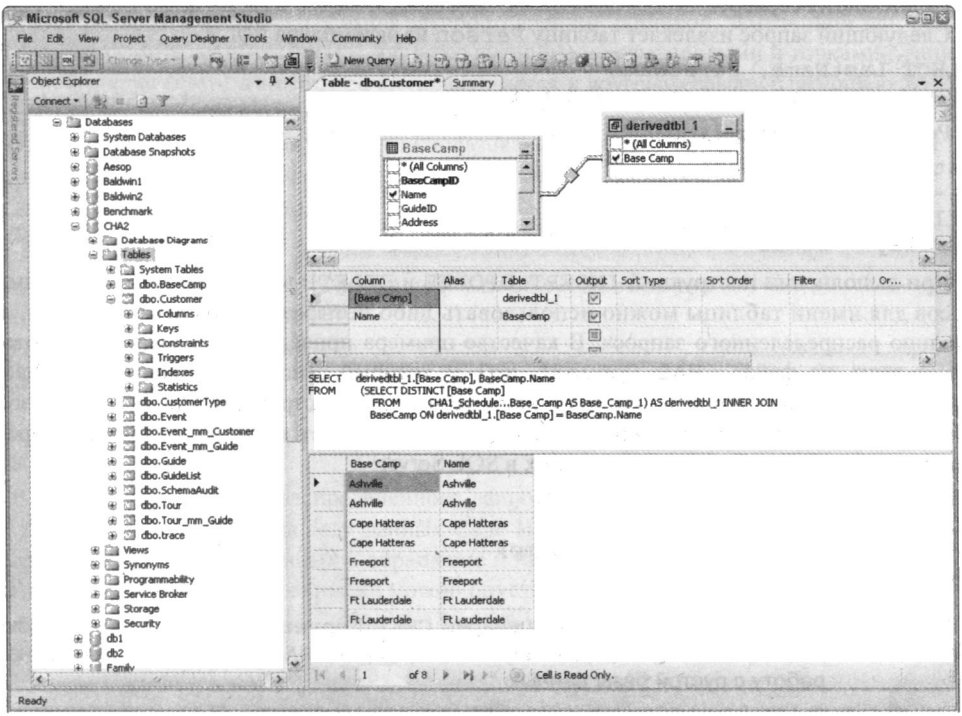


Рис. 15.5. Распределенный запрос можно выполнить в Management Studio, если ввести его вручную на панели SQL ([XPS].Family.dbo.Person)

Распределенные представления

Представления — это сохраненные инструкции SQL. Хотя я не рекомендую основывать приложения клиент/сервер на представлениях, они могут оказаться полезными для текущих запросов. Так как бо́льшая часть пользователей (в том числе и разработчики) не знакома с различными методами выполнения распределенных запросов, помещение такового в представление может оказаться удачным решением проблемы.

Локальные распределенные запросы

Хотя термин *локальный распределенный запрос* звучит странно, все не так уж сложно. Это запрос, который собирает внешние данные в SQL Server, а затем выполняет запрос на локальном сервере. Так как обработка таких запросов выполняется на локальном сервере, в них используется синтаксис T-SQL, и поэтому их иногда называют *локальными запросами T-SQL.*

Использование четырехкомпонентного имени

Если данные находятся на другом экземпляре SQL Server, то полный синтаксис четырехкомпонентного имени следующий:

Сервер.База_данных.Схема.Имя_Объекта

Четырехкомпонентное имя может использоваться в любых запросах извлечения или модификации данных. На моем компьютере существует второй экземпляр SQL Server с именем `[XPS\Yukon]`. Имя владельца объекта является обязательным, если обращение осуществляется к внешнему серверу.

Следующий запрос извлекает таблицу `Person` из экземпляра `SQL2`:

```
SELECT LastName, FirstName
  FROM [XPS\Yukon].Family.dbo.person
```

Результат запроса следующий:

```
LastName   FirstName
---------- ---------------
Halloway   Kelly
Halloway   James
```

При выполнении инструкций `INSERT`, `UPDATE` и `DELETE` в качестве распределенных запросов для имени таблицы можно использовать либо четырехкомпонентную форму, либо функцию распределенного запроса. В качестве примера приведем следующий код, который можно взять из файла `CHA2_Convert.sql` и который заполняет учебную базу данных `CHA2`. В этом примере в качестве источника данных для инструкции `INSERT` использовано четырехкомпонентное имя таблицы. Этот запрос извлекает названия базовых лагерей из электронной таблицы Excel и вставляет их в SQL Server:

```
INSERT BaseCamp(Name)
SELECT DISTINCT [Base Camp]
  FROM CHA1_Schedule...[Base_Camp]
  WHERE [Base Camp] IS NOT NULL
```



Если вы уже выполняли сценарий `CHA2_Convert.sql` и заполнили свою копию базы `CHA2`, еще раз запустите сценарий `CHA2_Create.sql`, чтобы начать работу с пустой базы данных.

А вот еще один пример использования четырехкомпонентного имени для распределенного запроса. В нем обновляется база данных `Family`, находящаяся на другом экземпляре SQL Server:

```
UPDATE [Noli\SQL2].Family.dbo.Person
   SET LastName = 'Wilson'
   WHERE PersonID = 1
```

Использование функции OpenDataSource()

Использование функции OpenDataSource() ничем не отличается от использования четырехкомпонентного имени для доступа к связанному серверу, за исключением того, что функция определяет подключение в своем теле, а не использует предопределенный связанный сервер. В то время как определение связи в программе пропускает определение самого сервера, если местонахождение связи изменяется, это коснется всех запросов, использующих функцию OpenDataSource(). К тому же функция OpenDataSource() не принимает переменные в качестве аргументов.

Функция OpenDataSource() заменяет имя сервера в четырехкомпонентном имени и может использоваться в любой инструкции DML.

Синтаксис функции OpenDataSource() выглядит довольно простым:

OpenDataSource(*провайдер, строка_инициализации*)

Однако первое впечатление обманчиво. *Строка_инициализации* представляет собой символьную строку с несколькими параметрами, разделенными точками с запятыми (точный список параметров зависит от конкретного источника данных). Потенциально в строке инициализации указывается источник данных, его местоположение, дополнительные параметры, время ожидания подключения, идентификатор пользователя, его пароль и каталог. В строке инициализации должны быть указаны все необходимые параметры подключения к источнику данных, в том числе контекст безопасности. Отдельные параметры в строке инициализации не нужно заключать в кавычки. Самой распространенной ошибкой, замеченной при реализации функции OpenDataSource(), является путаница между запятыми и точками с запятой.

Если функция OpenDataSource() подключается к другому серверу с помощью Windows, то необходима аутентификация с поддержкой Kerberos.

Вот относительно простой пример использования функции OpenDataSource() в качестве механизма доступа к таблице в другом экземпляре SQL Server:

```
SELECT FirstName, Gender
  FROM OPENDATASOURCE(
      'SQLOLEDB',
      'Data Source=NOLI\SQL2;User ID=Joe;Password=j'
       ).Family.dbo.Person;
```

Результат выполнения запроса:

```
FirstName       Gender
--------------- ------
Adam            M
Alexia          F
```

В следующем примере распределенного запроса, использующего функцию OpenDataSource(), мы ссылаемся на базу данных Cape Hatteras Adventures. Поскольку файл Access содержит всего одну базу данных и обращение к таблицам не требует указания владельца, эти части в четырехкомпонентном имени можно опустить:

```
SELECT ContactFirstName, ContactLastName
  FROM OPENDATASOURCE(
        'Microsoft.Jet.OLEDB.4.0',
        'Data Source =
                  C:\SQLServerBible\CHA1_Customers.mdb'
         )...Customers;
```

Результат выполнения запроса:

```
ContactFirstName     ContactLastName
-------------------- ---------------
Neal                 Garrison
Melissa              Anderson
Gary                 Quill
```

Для иллюстрации использования функции `OpenUpdateSource()` в запросе `UPDATE` мы обновим все строки в рабочей книге Excel `CHA1_Schedule.xls`. Именованный диапазон был определен заранее: `Tours '=Sheet1!$E$5:$E$24'`. Теперь он будет использован в запросе SQL в качестве таблицы в источнике данных. Вместо того чтобы обновлять отдельно каждую ячейку рабочего листа, этот запрос выполняет инструкцию `UPDATE`, затрагивающую все строки, в которых названием тура является `Gauley River Rafting`, и обновляет столбец `Base Camp` значением `Ashville`.

Распределенный запрос SQL Server для доступа к механизму Jet, который открывает рабочий лист Excel, будет использовать поставщика OLE DB. Функции `OpenDataSource()` мы передаем только имя сервера в четырехкомпонентной форме; при этом, как и в случае с Access, имя базы данных и владельца опускаем:

```
UPDATE OpenDataSource(
     'Microsoft.Jet.OLEDB.4.0',
     'Data Source=C:\SQLServerBible\CHA1_Schedule.xls;
     User ID=Admin;Password=;Extended properties=Excel 5.0'
     )...Tour
  SET [Base Camp] = 'Ashville'
  WHERE Tour = 'Gauley River Rafting';
```

На рис. 15.6 показан план выполнения распределенного запроса `UPDATE`. Он начинается справа с блока `Remote Scan`, который возвращает все 19 строк именованного диапазона Excel. После этого данные обрабатываются в SQL Server. Логическая операция `Remote Update` сводится к тому, что распределенный запрос обновления на самом деле изменяет значения всего в двух строках.

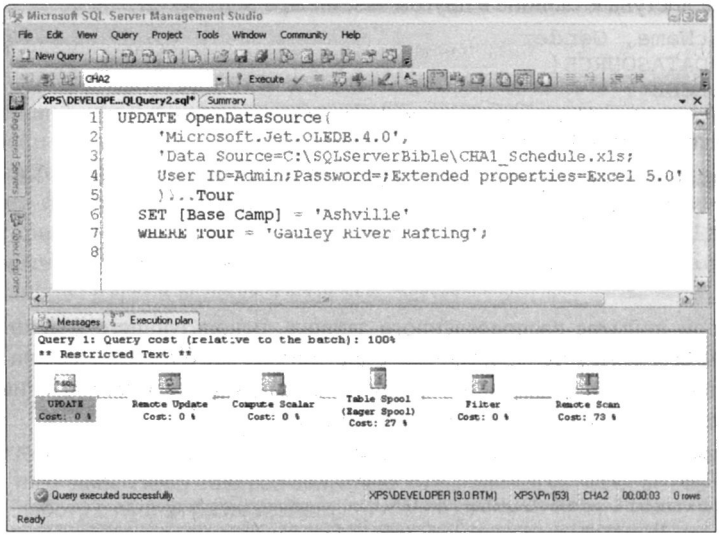


Рис. 15.6. План выполнения распределенного запроса обновления, использующего функцию `OpenDataSource()`

Чтобы завершить пример, следующий запрос считывает тот же рабочий лист Excel и проверяет, действительно ли имело место обновление. И снова функция `OpenDataSource()`, единственная в распределенном запросе, указывает на внешний сервер:

```
SELECT *
  FROM OpenDataSource(
    'Microsoft.Jet.OLEDB.4.0',
    'Data Source=C:\SQLServerBible\CHA1_Schedule.xls;
    User ID=Admin;Password=;Extended properties=Excel 5.0'
         )...Tour
  WHERE Tour = 'Gauley River Rafting';
```

Результат выполнения запроса будет следующим:

```
Base Camp       Tour
--------------- ----------------------
Ashville        Gauley River Rafting
Ashville        Gauley River Rafting
```

Сквозные распределенные запросы

Сквозные запросы выполняются на внешнем источнике данных, после чего результат передается на SQL Server. Основная причина использования таких запросов заключается в сокращении объема данных, передаваемых с сервера (внешнего источника данных) и клиента (SQL Server). Вместо передачи миллиона строк на SQL Server, чтобы использовать только двадцать пять из них, может оказаться выгоднее извлечь только эти строки из внешнего источника данных.

В то же время сквозные запросы должны использовать синтаксис внешнего сервера. Если источником данных является база данных Oracle, то в сквозном запросе должен использоваться язык PL/SQL; если база данных Access — то Access SQL.

В случае сквозного запроса, модифицирующего данные, сам тип удаленных данных определяет, будет ли обновление выполняться локально или удаленно.

- Если обновляются данные на другом экземпляре SQL Server, то операция будет выполняться именно на нем.
- Если данные обновляются на сервере, отличном от SQL Server, то поставщик определяет, где именно будут обрабатываться данные. Часто передаваемые запросы просто отбирают корректные строки. Эти строки передаются на SQL Server, обновляются на нем, а затем возвращаются удаленному источнику данных для обновления.

Существуют две формы локальных распределенных запросов: одна для связанных серверов и одна для внешних источников данных, определяемых в запросе. Также существуют две формы сквозных распределенных запросов. В одном случае функция `OpenQuery()` использует уже подключенный сервер; во втором — функция `OpenRowSet()` определяет связь непосредственно в запросе.

Использование четырехкомпонентного имени

Если распределенный запрос осуществляет доступ к другому экземпляру SQL Server, четырехкомпонентное имя называют *гибридным методом распределенных запросов*. В зависимости от предложений `FROM` и `WHERE`, SQL Server будет пытаться передать как можно большую часть запроса удаленному серверу, чтобы увеличить производительность.

При создании сложных распределенных запросов, использующих четырехкомпонентное имя, довольно сложно предсказать заранее, какая часть запроса будет передана. Мне встречались серверы, которые, в зависимости от предложения `WHERE`, либо передавали весь запрос, либо каждая таблица передавала свой запрос, либо запрос передавала только одна таблица.



Из четырех представленных методов распределенных запросов лучшими являются использование четырехкомпонентного имени и функции `OpenQuery()` соответственно. Оба метода используют административную настройку подключения к удаленному серверу, что делает запрос более надежным, если конфигурация сервера изменяется.

Выбор между этими двумя методами зависит от общего объема данных, их извлекаемого количества и производительности сервера. Рекомендую вам протестировать оба метода и сравнить их планы выполнения, после чего выбрать тот, который лучше проявляет себя в вашей ситуации. Если оба метода показывают равную производительность, то используйте метод четырехкомпонентного имени, чтобы сервер имел возможность автоматически оптимизировать распределенный запрос.

Функция `OpenQuery()`

Среди передаваемых запросов функция `OpenQuery()` задействует ресурсы уже подключенного сервера, поэтому с ней проще обращаться. Она также воспринимает изменения в конфигурации серверов без изменения кода.

Функция `OpenQuery()` используется в языке SQL DML в качестве таблицы. Она принимает два аргумента: имя связываемого сервера и сам передаваемый запрос. В следующем примере функция `OpenQuery()` используется для извлечения данных из рабочей книги Excel `CHA1_Schedule`:

```
SELECT *
  FROM OPENQUERY(CHA1_Schedule,
    'SELECT * FROM Tour WHERE Tour = "Gauley River Rafting"');
```

Результат выполнения запроса:

```
Tour                 Base Camp
-------------------- ---------
Gauley River Rafting   Ashville
Gauley River Rafting   Ashville
```

Как показано на рис. 15.7, передаваемый запрос, использующий функцию `OpenQuery()`, практически не требует обработки сервером — ему возвращаются ровно две строки. Предложение `WHERE` обрабатывается механизмом Jet при извлечении данных из рабочего листа Excel.

В следующем примере функция `OpenQuery()` дает указание механизму Jet, чтобы тот извлек только две строки, требующие обновления. Реальная инструкция `UPDATE` выполняется на сервере, а результат возвращается внешнему источнику данных. В результате передаваемый запрос выполняет в инструкции `UPDATE` только часть функции `SELECT`:

```
UPDATE OPENQUERY(CHA1_Schedule,
    'SELECT * FROM Tour WHERE Tour = "Gauley River Rafting"')
  SET [Base Camp] = 'Ashville'
  WHERE Tour = 'Gauley River Rafting';
```

Функция `OpenRowSet()`

Эта функция является двойником функции `OpenDataSet()`. Обе требуют, чтобы в распределенном запросе был полностью определен удаленный источник данных. Функция `OpenRowSet()` имеет дополнительный аргумент, определяющий передаваемый запрос:

```
SELECT ContactFirstName, ContactLastName
  FROM OPENROWSET ('Microsoft.Jet.OLEDB.4.0',
  'C:\SQLServerBible\CHA1_Customers.mdb'; 'Admin';'',
  'SELECT * FROM Customers WHERE CustomerID = 1');
```

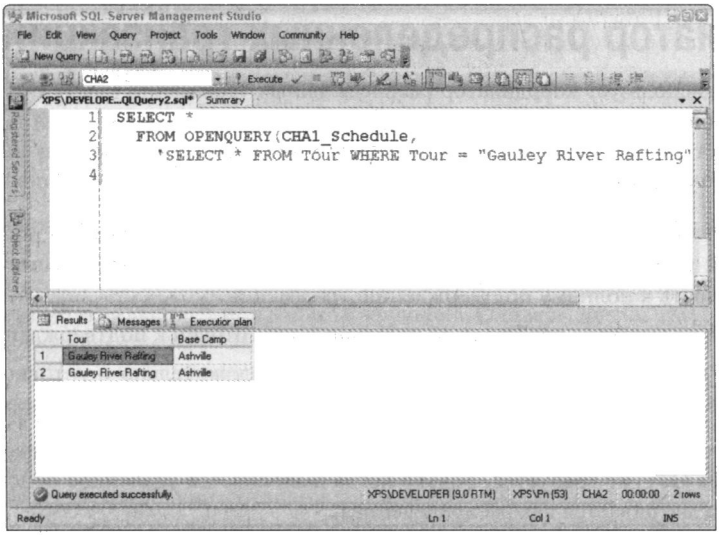


Рис. 15.7. Распределенный запрос, использующий функцию OpenQuery(), возвращает только две строки, отобранные предложением WHERE

Результат запроса следующий:

```
ContactFirstName     ContactLastName
-------------------- ---------------
Tom                  Mercer
```

Чтобы выполнить обновление с помощью функции OpenRowSet(), вставьте ее на место модифицируемой таблицы. В следующем примере мы изменим фамилию заказчика в базе данных Access. Предложение WHERE инструкции UPDATE обрабатывается передаваемой частою функции OpenRowSet():

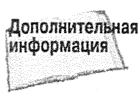
```
UPDATE OPENROWSET ('Microsoft.Jet.OLEDB.4.0',
  'C:\SQLServerBible\CHA1_Customers.mdb'; 'Admin';'',
  'SELECT * FROM Customers WHERE CustomerID = 1')
  SET ContactLastName = 'Wilson';
```



Операции массового заполнения поддерживаются функцией OpenRowSet(), и это существенно повышает их производительность.

Распределенные транзакции

Транзакции являются ключевым элементом поддержания целостности данных. Если логическая единица работы содержит изменение данных вне локального сервера, то стандартная транзакция не сможет обеспечить атомарность операции. Если в середине транзакции случится ошибка, то должен существовать механизм, способный отменить выполненную часть работы. В противном случае будет записана частичная транзакция, и база данных останется в противоречивом состоянии.



В главе 51 мы подробно обсудим свойства ACID баз данных и транзакций.

Координатор распределенных транзакций

SQL Server использует координатор распределенных транзакций (далее DTC) для обслуживания транзакций, затрагивающих несколько серверов, — их подтверждения и отката. Служба DTC использует двухфазную схему подтверждения многосерверных транзакций. Это подразумевает, что доступны оба сервера, при этом служба DTC выполняет следующие действия.

1. Каждый из серверов отправляет сообщение о готовности к подтверждению.
2. Каждый из серверов выполняет первый этап подтверждения, гарантируя тем самым способность к полному подтверждению транзакции.
3. Каждый из серверов сообщает об окончании подготовки к подтверждению.
4. Только после того как все серверы ответили положительно о готовности к подтверждению, сообщение о реальном подтверждении транзакции отправляется всем им.

Если логическая единица работы содержит только операции чтения с серверов, то использование службы DTC не требуется. Только когда выполняется удаленное обновление данных, транзакция рассматривается как распределенная.

Координатор распределенных транзакций является отдельной службой SQL Server, которая запускается и останавливается с помощью SQL Server Service Manager.

Только один экземпляр этой службы запускается на одном сервере, независимо от количества установленных и запущенных на нем экземпляров SQL Server. Фактическое имя службы — `msdtc.exe`, потребляемый объем памяти — 2,5 Мбайт.

Служба DTC должна быть запущена, когда инициируется распределенная транзакция, в противном случае последняя завершится ошибкой.

Создание распределенных транзакций

Распределенные транзакции аналогичны локальным, но имеют некоторые расширения в синтаксисе:

```
SET xact_abort on;
BEGIN DISTRIBUTED TRANSACTION;
```

В случае ошибки параметр подключения `xact_abort` приведет к откату текущей транзакции, а не только текущей инструкции T-SQL. Параметр `xact_abort` необходим, чтобы любая распределенная транзакция могла подключиться к удаленному экземпляру SQL Server, равно как и к большинству других соединений OLE DB.

Инструкция `BEGIN DISTRIBUTED TRANSACTION`, которая определяет, доступна ли служба DTC, строго не требуется. Даже если транзакция начинается инструкцией `BEGIN TRAN`, она все равно будет расширена до распределенной, и наличие службы DTC будет проверено, как только будет выполнен первый распределенный запрос. В то же время считается хорошей практикой использовать полную форму `BEGIN DISTRIBUTED TRANSACTION`, чтобы наличие DTC проверялось в самом начале транзакции. Если служба DTC не запущена, то автоматически выдается сообщение об ошибке:

```
Server: Msg 8501, Level 16, State 3, Line 7
MSDTC on server 'XPS' is unavailable.
```

Следующий пример демонстрирует распределенную транзакцию между локальным сервером и его вторым экземпляром:

```
USE Family;
SET xact_abort on;
```

```
BEGIN DISTRIBUTED TRANSACTION;

  UPDATE Person
    SET LastName = 'Johnson2'
    WHERE PersonID = 10;


  UPDATE [Noli\SQL2].Family.dbo.Person
    SET LastName = 'Johnson2'
    WHERE PersonID = 10;
COMMIT TRANSACTION;
```

Обычно при откате вложенных транзакций SQL Server откатываются все транзакции, находящиеся в режиме ожидания, т.е. до самой внешней. Однако служба DTC поступает по-другому — она откатывает только текущую транзакцию.

Мониторинг распределенных транзакций

Работу координатора распределенных транзакций можно просмотреть в самой операционной системе в качестве отдельно устанавливаемой службы сервера Windows Server 2003. Для этого нужно в главном системном меню Start выбрать пункт Control Panel⇨Administrative Tools⇨Component Services. Служба компонентов отображает список текущих ожидающих распределенных транзакций (рис. 15.8), а также статистику службы DTC (рис. 15.9).

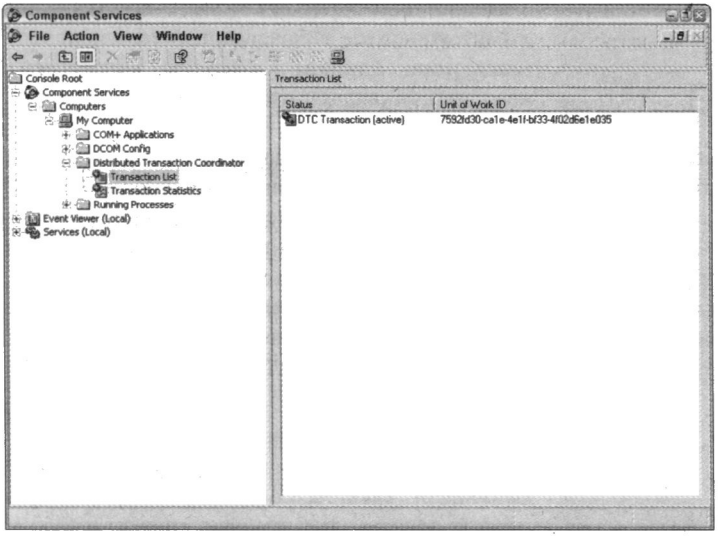


Рис. 15.8. Служба компонентов отображает список текущих транзакций DTC

Если некоторая распределенная транзакция столкнулась с трудностями, она, что наиболее вероятно, будет преждевременно отменена. Если же транзакция помечена как `In Doubt`, то ручное ее подтверждение или откат с помощью контекстного меню в службе компонентов могут разрешить ситуацию.

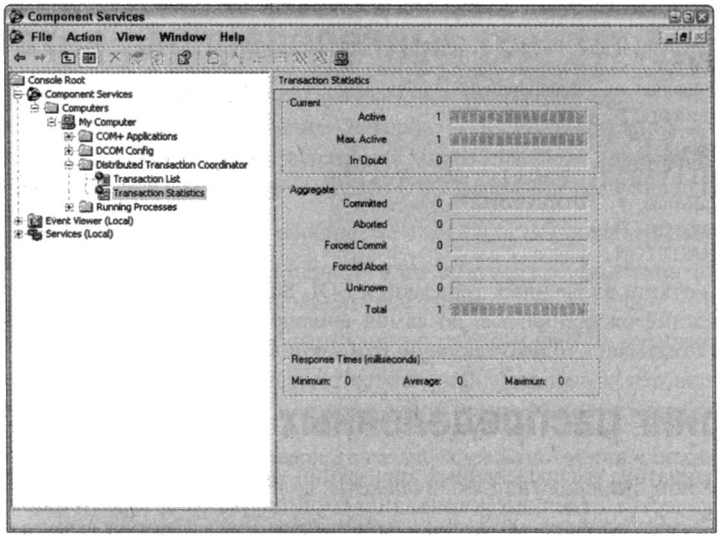


Рис. 15.9. В службе компонентов можно увидеть текущую и накопительную статистику о распределенных транзакциях. При перезапуске службы DTC статистика полностью очищается

Резюме

Корпоративные данных имеют тенденцию размещаться на множестве платформ и компьютеров. Способность SQL Server брать на вооружение возможности OLE DB и ODBC для выполнения распределенных запросов является ключевым фактором успеха множества проектов баз данных. Поэтому знание методов создания распределенных запросов является необходимым компонентом в арсенале разработчика баз данных.

Кроме перемещения данных с помощью распределенных запросов, инструкция `SELECT` незаменима при изменении данных. В следующей главе мы рассмотрим действия, выполняемые этой инструкцией при добавлении команд `INSERT`, `UPDATE` и `DELETE`.

Модификация данных

ГЛАВА

16

В этой главе...

Вставка данных из выражений, других результирующих наборов данных и хранимых процедур

Обновление данных

Удаление данных

Предупреждение и решение сложных проблем модификации данных

Все изменяется. Жизнь не может стоять на месте. Поскольку основной задачей базы данных является достоверное отражение реальности, то и ее данные должны изменяться вместе с реальным миром. Для программистов SQL это выливается в использование инструкций языка манипулирования данными DML. Эти операции не ограничиваются всего одной строкой данных. Язык SQL предполагает мышление в терминах наборов данных. Процесс модификации данных с помощью SQL предполагает использование всех допустимых возможностей извлечения данных — мощной инструкции SELECT, объединений, полнотекстового поиска, подзапросов и представлений.



На самом деле инструкции INSERT, UPDATE и DELETE являются всего лишь надстройкой над всеобъемлющей инструкцией SELECT. В основе любой операции модификации данных лежит полный потенциал инструкции SELECT. Даже при модификации данных вы должны мыслить в категориях наборов данных, а не только одной строки.

Вся эта глава посвящена вопросу модификации данных в SQL Server с помощью инструкций INSERT, UPDATE и DELETE. Модификация данных поднимает ряд вопросов, которые следует рассмотреть отдельно. Вставка первичных ключей требует использования специальных методов, а ограничения таблиц могут конфликтовать с операциями модификации данных. Ссылочная целостность может потребовать каскадного удаления данных из нескольких таблиц с помощью инструкции DELETE. Эта глава поможет вам понять эти и другие концепции, а также предложит методы решения подобных задач. Так как эти вопросы одновременно касаются инструкций INSERT, UPDATE и, некоторым образом, DELETE, они вынесены за пределы разделов, посвященных каждой из этих инструкций.

Дополнительная информация

В вопросах модификации данных критичными являются требования ACID (атомарности, целостности, изолированности и живучести). Для многих баз данных вполне достаточно принятого по умолчанию в SQL Server управления транзакциями. Однако излишние блокировки, связанные с ними, являются одним из четырех основных факторов снижения производительности. В главе 51 мы углубимся в архитектуру SQL Server, и вы поймете, как сделать так, чтобы операции модификации данных в транзакциях удовлетворяли требованиям ACID. В этой же главе мы рассмотрим механизмы блокировки данных в SQL Server.

Инструкции модификации данных могут выполняться в SQL Server с помощью различных интерфейсов. Эта глава в основном будет посвящена вопросам использования команд `INSERT`, `UPDATE` и `DELETE` для предоставления серверу инструкций относительно изменения данных программным путем.

В утилите Management Studio пользователю предлагаются два альтернативных интерфейса выполнения инструкций SQL: конструктор запросов (Query Designer) и редактор запросов (Query Editor). Последний, несмотря на отсутствие визуального представления столбцов и объединений, имеет гораздо более богатый набор средств работы с инструкциями T-SQL. С другой стороны, конструктор запросов предоставляет вам два способа проектирования инструкций манипулирования данными — визуально и с помощью программного кода (рис. 16.1). Любой из этих интерфейсов уместен для изучения команд модификации данных, однако так как редактор запросов имеет преимущество в использовании программного кода, я бы порекомендовал в ходе изучения настоящей главы использовать именно его.

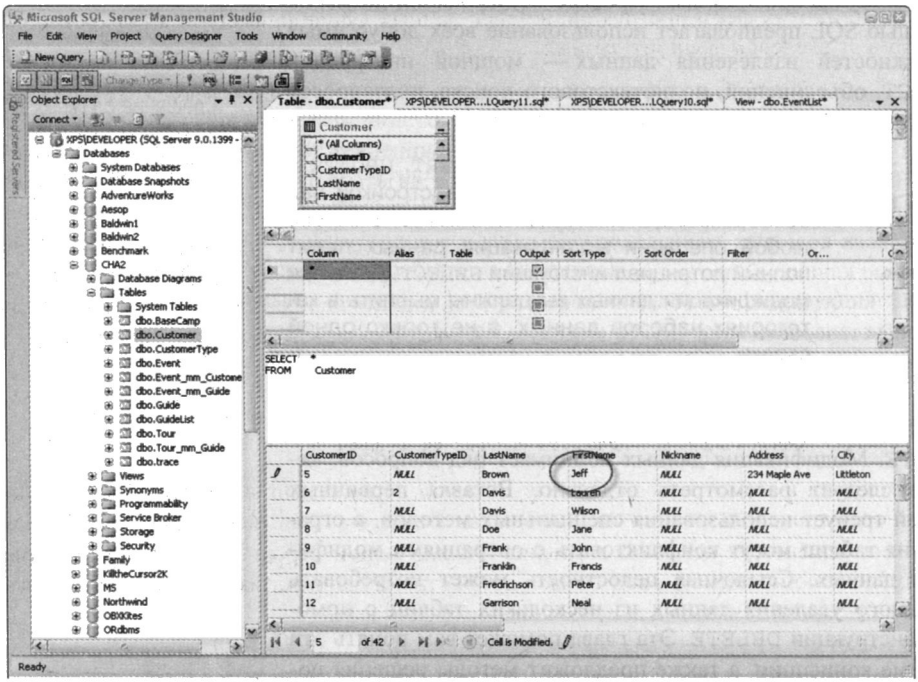


Рис. 16.1. Конструктор запросов утилиты Management Studio отлично приспособлен для задач модификации данных. В настоящем примере имя Jeff изменено непосредственно на панели результатов

Дополнительная информация

Подробнее об использовании конструктора запросов утилиты Management Studio см. в главе 6.

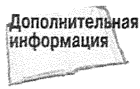
Вставка данных

Язык SQL предлагает четыре формы инструкций INSERT и SELECT INTO как основные методы вставки данных (табл. 16.1). В то время как простые методы реализуют вставку всего одной строки данных, более сложные получают результаты от вложенных инструкций SELECT и создают из результатов таблицы.

Таблица 16.1. Формы операций вставки

Форма вставки	Описание
INSERT/VALUES	Вставляет одну строку значений. Обычно используется для поддержки интерактивного ввода данных пользователем
INSERT/SELECT	Вставляет в таблицу результирующий набор данных вложенного подзапроса
INSERT/EXEC	Вставляет в таблицу результаты хранимой процедуры. Обычно используется в сложных задачах манипулирования данными
INSERT DEFAULT	Создает новую строку с применением всех значений, заданных по умолчанию. Обычно используется для вставки общепринятых полей строк данных
SELECT INTO	Создает новую таблицу из результирующего набора данных инструкции **SELECT**

Каждую из этих форм инструкции вставки данных лучше применять к конкретной задаче или форме извлечения внешних данных.



SQL Server сопровождает инструкции INSERT великим множеством средств преобразования данных и манипулирования ими. Популярные мастера *Bulk Copy Wizard* и *Copy Database Wizard* будут представлены в главе 35. Последний из них создает пакет службы интеграции. О взаимодействии практически любых множеств данных любого размера речь пойдет в главе 42.

При вставке новых данных для идентификации новых строк должны быть сгенерированы первичные ключи. Несмотря на то, что как столбцы идентичности, так и глобальные универсальные идентификаторы (GUID) являются отличными кандидатами на эту роль, каждый из них требует особого подхода. В этом разделе вы узнаете, как создавать значения столбцов идентичности и GUID.

Вставка одной строки значений

Простейший и самый непосредственный метод вставки данных заключается в использовании инструкции INSERT...VALUES. Так как эта форма принимает только один набор значений, она ограничена вставкой в таблицу только одной строки. Интерфейс пользователя имеет тенденцию принимать ввод только одной строки данных, так что этот метод считается наиболее предпочтительным для использования с формами.

```
INSERT [INTO] владелец.таблица [(столбец,...)]
VALUES (значение,...)
```

Создать инструкцию INSERT...VALUES совсем не сложно, но в то же время существует несколько вариантов. Ключевое слово INTO является необязательным и поэтому часто игнорируется. Это ключевое слово проверяет правильность столбцов в списке инструкции INSERT, а также соответствие типа перечисляемых значений типу вставляемых столбцов.

Когда значения вставляются в новую строку, каждое значение соответствует некоторому столбцу. Столбцы в списке могут перечисляться в любом порядке (не обязательно совпадающем с порядком столбцов в таблице) — главное, чтобы и значения имели тот же порядок.



В файле CH `16-Modifying Data.sql`, который можно загрузить с сайта `www.SQLServerBible.com`, содержатся все примеры программного кода, приводимые в настоящей главе. Дополнительные примеры инструкций модификации данных вы можете найти в сценариях заполнения учебных баз данных, а также в хранимых процедурах базы данных OBXKites.

Следующие инструкции `INSERT` обращаются к столбцам в разном порядке:

```
USE CHA2
INSERT INTO dbo.Guide (LastName, FirstName, Qualifications)
  VALUES ('Smith', 'Dan', 'Diver, Whitewater Rafting')
INSERT INTO dbo.Guide (FirstName, LastName, Qualifications)
  VALUES ('Jeff', 'Davis', 'Marine Biologist, Diver')
INSERT INTO dbo.Guide (FirstName, LastName)
  VALUES ('Tammie', 'Commer')
```

С помощью следующей инструкции `SELECT` мы проверим правильность вставки:

```
SELECT * FROM dbo.Guide
```

Давайте посмотрим на результат (полученный вами результат может отличаться от представленного, поскольку в базе данных могут содержаться другие данные):

```
GuideID  LastName   FirstName   Qualifications
-------- ---------- ----------- ----------------------------
1        Smith      Dan         Diver, Whitewater Rafting
2        Davis      Jeff        Marine Biologist, Diver
3        Commer     Tammie      NULL
```

Не все столбцы таблицы обязательно должны быть перечислены в инструкции `INSERT`. Третья инструкция приведенного выше примера не упоминала столбец квалификации, тем не менее она была выполнена, а в пропущенный столбец было вставлено пустое значение.

Если бы столбец `Qualification` имел значение по умолчанию, оно было бы занесено в него вместо пустого. Если бы в ограничениях столбца был установлен запрет пустых значений, то эта инструкция не была бы выполнена. (Более подробно о вставке пустых значений и значений по умолчанию мы поговорим в разделе "Потенциальные препятствия на пути модификации данных".)

Потенциально можно форсировать в инструкции `INSERT` вставку значений по умолчанию, даже не зная об их существовании. Для этого в списке столбцов/значений используют ключевое слово `DEFAULT`; при этом SQL Server запоминает указанное значение. Такой прием часто используется на практике, поскольку это позволяет задокументировать намерения, а не полагаться на наличие значений по умолчанию.

Явное перечисление столбцов является правильным подходом, поскольку позволяет предупредить вероятные изменения схемы таблицы (а также задокументировать саму вставку). В то же время список столбцов в инструкции `INSERT` не является обязательным. Если он не указан явно, значения вставляются в том порядке, в котором столбцы перечислены в определении таблицы (при этом столбец идентичности игнорируется). Очень критично, чтобы все столбцы таблицы получили корректные значения. Пропуск любого столбца в списке значений приведет к срыву всей операции вставки.

Ранее вы узнали, что когда столбцы явно перечислены в списке инструкции `INSERT`, столбец идентичности не может получить значение. Аналогичным образом, столбец идентичности игнорируется в списке значений, когда столбцы явно не указаны. Все остальные значения сохраняют тот же порядок. Продемонстрируем этот факт на примере таблицы `Guide`:

```
INSERT Guide
  VALUES ('Jones', 'Lauren',
      'First Aid, Rescue/Extraction', '25/6/59', '15/4/01')
```

Чтобы посмотреть на результат операции вставки, извлечем данные из таблицы Guide с помощью инструкции SELECT:

```
GuideID  LastName   FirstName   Qualifications
-------- ---------- ----------- ------------------------------
1        Smith      Dan         Diver, Whitewater Rafting
2        Davis      Jeff        Marine Biologist, Diver
3        Commer     Tammie      NULL
4        Jones      Lauren      First Aid, Rescue/Extraction
```

До сих пор значения в примерах представляли собой константы, однако они могут являться и результатами вычисления выражений. Это очень ценно, особенно когда данные требуют преобразований (вычислений, изменений, конкатенации и т.п.).

```
INSERT dbo.Guide (FirstName, LastName, Qualifications)
  VALUES ('Greg', 'Wilson',
     'Rock Climbing' + ',' + 'First Aid')
```

Следующая инструкция SELECT проверяет вставку данных об инструкторе Greg:

```
Select * FROM dbo.Guide
```

Результат будет следующим:

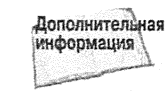
```
GuideID  LastName   FirstName   Qualifications
-------- ---------- ----------- --------------------------
1        Smith      Dan         Diver, Whitewater Rafting
2        Davis      Jeff        Marine Biologist, Diver
3        Commer     Tammie      NULL
4        Jones      Lauren      First Aid, Rescue/Extraction
5        Wilson     Greg        Rock Climbing, First Aid
(5 row(s) affected)
```

Когда данные вставляются в базу данных, они обычно вводятся пользователем в некоторой форме, и в этом случае метод INSERT...VALUES является наиболее приемлемым. Однако этот метод нельзя назвать динамичным. Если данные уже существуют в базе данных, то более эффективно будет использовать форму INSERT...SELECT.

Вставка результирующего набора данных инструкции SELECT

Данные можно переместить из результирующего набора данных в таблицу с помощью инструкции INSERT...SELECT. Реальная сила этого метода, а также его гибкость заключаются в том, что сама инструкция SELECT может извлечь данные практически из любого места и адаптировать их к текущим потребностям. Так как инструкция SELECT может вернуть бесконечное число строк, все они могут быть вставлены в таблицу. Синтаксис этой инструкции следующий.

```
INSERT [INTO] владелец.таблица
  SELECT столбцы
    FROM источники_данных
    [WHERE условия]
```



Исчерпывающую дискуссию о предложении SELECT этой инструкции вы найдете в главе 7 и последующих главах части II.

Как и в инструкции INSERT...VALUES, столбцы должны быть упорядочены и иметь соответствующие типы данных. Если необязательный список столбцов опускается, все столбцы должны заполняться в том порядке, в котором объявлены при создании таблицы (за исключением столбца идентичности).

В следующем примере мы используем базы данных OBXKites. В нем мы извлечем фамилии всех экскурсоводов из таблицы Guide базы данных Cape Hatteras Adventures и вставим их в список клиентов (таблицу Contact) базы данных OBXKites. Имена столбцов извлекаются из таблицы Guide, а название компании мы представляем строковой константой (обратите внимание на то, что при обращении к таблице Guide используется трехчастное имя *база_данных.владелец.таблица*).

```
USE OBXKites
-- Используем чистую копию базы данных OBXKites, до заполнения

INSERT dbo.Contact (FirstName, ContactCode, LastName, CompanyName)
  SELECT FirstName, LastName, GuideID, 'Cape Hatteras Adv.'
      FROM CHA2.dbo.Guide
```

Проверим вставку:

```
SELECT FirstName AS First, LastName AS Last, CompanyName
  FROM dbo.Contact
```

Будет получен следующий результат:

```
First      Last       CompanyName
---------- ---------- --------------------
Dan        Smith      Cape Hatteras Adv.
Jeff       Davis      Cape Hatteras Adv.
Tammie     Commer     Cape Hatteras Adv.
Lauren     Jones      Cape Hatteras Adv.
Greg       Wilson     Cape Hatteras Adv.
(5 row(s) affected)
```

Ключевым моментом в использовании инструкции INSERT...SELECT является извлечение корректного результирующего набора данных. Неплохо до выполнения вставки запустить инструкцию SELECT самостоятельно и протестировать ее результаты. Как говорится, семь раз отмерь, один отрежь.

Вставка результирующего набора данных из хранимой процедуры

Форма INSERT...EXEC инструкции вставки использует результаты выполнения хранимой процедуры для их вставки в таблицу. В данном случае можно использовать все возможности языка T-SQL. Базовая функция вставки является такой же, как и во всех ее остальных формах. Порядок столбцов в списке инструкции INSERT и в результатах хранимой процедуры должен быть одинаковым. Базовый синтаксис этой инструкции:

```
INSERT [INTO] владелец.таблица [(столбцы)]
EXEC хранимая_процедура параметры
```

Однако будьте бдительны, поскольку хранимая процедура может вернуть несколько наборов данных, и в этом случае инструкция INSERT попытается выполнить вставку из каждого из них. Таким образом, порядок столбцов должен соответствовать *каждому* из возвращаемых хранимой процедурой наборов данных.



Подробно о программировании хранимых процедур вы узнаете в главе 21.

В следующем примере создается хранимая процедура, возвращающая имена и фамилии сотрудников из баз данных Cape Hatteras Adventures и Northwind (последняя входит в комплект поставки версии SQL Server 2000). После этого создается таблица, в которую будут помещены результирующие наборы данных. После того как будут созданы и хранимая процедура, и место помещения данных, выполняется инструкция INSERT...EXEC:

```
Use CHA2
CREATE PROC ListGuides
AS
  SET NOCOUNT ON
-- Результирующий набор данных 1
  SELECT FirstName, LastName
  FROM dbo.Guide
-- Результирующий набор данных 2
  SELECT FirstName, LastName
  FROM northwind.dbo.employees
  RETURN
```

Проверим результаты выполнения хранимой процедуры:

```
Exec ListGuides

FirstName    LastName
------------ ------------
Dan          Smith
Jeff         Davis
Tammie       Commer
Lauren       Jones
Wilson       Greg

FirstName  LastName
---------- -------------
Nancy      Davolio
Andrew     Fuller
Janet      Leverling
Margaret   Peacock
Steven     Buchanan
Michael    Suyama
Robert     King
Laura      Callahan
Anne       Dodsworth
```

Следующая инструкция DDL создает таблицу, соответствующую структуре результирующих наборов данных хранимой процедуры:

```
CREATE TABLE dbo.GuideSample
  (FirstName VARCHAR(50),
   LastName VARCHAR(50) )
```

Итак, у нас все готово для выполнения инструкции вставки:

```
INSERT dbo.GuideSample (FirstName, LastName)
  EXEC ListGuides
```

Теперь проверим содержимое новой таблицы с помощью инструкции SELECT:

```
SELECT * FROM dbo.GuideSample
```

Будет получен следующий результат:

```
FirstName  LastName
---------- ------------
Dan        Smith
Jeff       Davis
Tammie     Commer
Lauren     Jones
Wilson     Greg
Nancy      Davolio
Andrew     Fuller
Janet      Leverling
Margaret   Peacock
Steven     Buchanan
Michael    Suyama
Robert     King
Laura      Callahan
Anne       Dodsworth
```

Создание инструкции INSERT...EXEC требует гораздо больше работы, чем INSERT... VALUES или INSERT...SELECT. В то же время хранимые процедуры могут содержать сложные логические схемы, и поэтому форма INSERT...EXEC является самой мощной из трех.



Формы INSERT...EXEC и SELECT...INTO не вставляют данные в табличные переменные. О табличных переменных мы подробно поговорим в главе 18.

Создание строки со значениями по умолчанию

Язык SQL имеет особую форму инструкции INSERT, которая создает новую строку, содержащую только значения столбцов по умолчанию. Единственным параметром такой инструкции является имя таблицы, при этом значения и имена столбцов не требуются и не принимаются во внимание. Синтаксис такой инструкции простой:

```
INSERT владелец.таблица DEFAULT VALUES
```

Лично я никогда не использовал эту форму инструкции INSERT на практике. Тем не менее, если вам когда-либо потребуется наполнить таблицу массой столбцов, содержащих значения, принятые по умолчанию, форма INSERT...DEFAULT может оказаться полезной.

Создание таблицы в процессе вставки данных

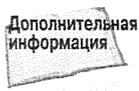
Последний метод вставки данных использует одну из вариаций инструкции SELECT. Используемый в ней параметр INTO принимает результирующий набор данных от инструкции SELECT и создает на его основе новую таблицу. Форму SELECT...INTO часто используют в операциях преобразования данных и в утилитах, которые должны работать с массой различных структур таблиц. Полный синтаксис этой формы включает все параметры инструкции SELECT, поэтому здесь мы приведем его усеченную версию:

```
SELECT столбцы
  INTO новая_таблица
  FROM источники_данных
  [WHERE условия]
```

Структура данных созданной таблицы может не соответствовать в точности исходной, поскольку она основана на комбинации столбцов исходной таблицы и результирующего

набора данных инструкции SELECT. Также может измениться длина строк и чисел. Если инструкция SELECT...INTO берет данные всего из одной таблицы и не содержит функций преобразования типов данных, то велика вероятность того, что столбцы таблицы и настройка пустых значений останутся нетронутыми. В то же время ограничения, ключи и индексы будут в любом случае утеряны.

Инструкцию SELECT...INTO можно рассматривать как операцию массового заполнения, аналогичную BULK INSERT и BULK COPY. Массовые операции позволяют серверу быстро перемещать данные в таблицы в обход процесса протоколирования транзакций (в принципе, все зависит от используемой модели восстановления данных). Параметры базы данных и модель восстановления влияют на инструкцию SELECT...INTO и прочие массовые операции. Если модель восстановления отличается от полной модели, то инструкция SELECT...INTO заноситься в журнал транзакций не будет.



Подробно об операциях BULK INSERT и BULK COPY вы узнаете в главе 19; а о моделях восстановления — в главе 36.

В следующем примере продемонстрировано использование инструкции SELECT...INTO для создания новой таблицы GuideList на основе извлечения данных из таблицы Guide (результаты операции несколько усечены):

```
USE CHA2
-- установка режима массового заполнения
Alter DATABASE CHA2 SET RECOVERY FULL
SP_DBOPTION 'CHA2', 'select into/bulkcopy', 'TRUE'
-- инструкция SELECT...INTO
SELECT *
  INTO dbo.GuideList
  FROM dbo.Guide
  ORDER BY Lastname, FirstName
```

Команда sp_help поможет нам увидеть структуру новой таблицы (результаты также несколько усечены):

```
sp_help GuideList
```

Будет получен следующий результат (некоторые строки опущены):

```
Name          Owner    Type          Created_datetime
------------ -------- ------------ -----------------------
GuideList     dbo      user table   2001-08-01 16:30:02.937

Column_name        Type        Length    Prec  Scale Nullable
------------------ ----------- --------- ----- ----- --------
GuideID            int         4         10    0     no
LastName           varchar     50                    no
FirstName          varchar     50                    no
Qualifications     varchar     2048                  yes
DateOfBirth        datetime    8                     yes
DateHire           datetime    8                     yes

Identity         Seed     Increment   Not For Replication
---------------- -------- ----------- -----------------------
GuideID          1        1           0

RowGuidCol
```

```
No rowguidcol column defined.

Data_located_on_filegroup
--------------------------
PRIMARY

The object does not have any indexes.
No constraints have been defined for this object.
No foreign keys reference this table.
No views with schema binding reference this table.
```

Следующая инструкция вставляет новую строку в таблицу, созданную инструкцией SELECT...INTO для проверки работоспособности столбца идентичности:

```
INSERT Guidelist (LastName, FirstName, Qualifications)
  VALUES('Nielsen', 'Paul', 'trainer')
```

Теперь посмотрим на данные, вставленные инструкцией SELECT...INTO и только что добавленные инструкцией INSERT...VALUES:

```
SELECT GuideID, LastName, FirstName
  FROM dbo.GuideList

GuideID     LastName     FirstName
----------- ------------ ------------
12          Nielsen      Paul
7           Atlas        Sue
11          Bistier      Arnold
3           Commer       Tammie
2           Davis        Jeff
10          Fletcher     Bill
5           Greg         Wilson
4           Jones        Lauren
1           Smith        Dan
```

Руководство по проектированию стиля данных

Существуют потенциальные проблемы, связанные с данными, которые выходят за рамки их типов, ограничений и допустимости пустых значений. Подобно тому, как средства проверки орфографии и грамматики программы Microsoft Word могут выявить очевидные ошибки (правда, не обратят внимание на плохой литературный стиль), база данных также может защитить от больших логических ошибок. Издатели в этом процессе руководствуются справочниками стилей и методическими рекомендациями. Например, скажите, как правильно назвать корпорацию Microsoft в книге: MS, Microsoft Corp. или Microsoft Corporation? Ответ на этот вопрос зависит от выбранного стиля.

Базы данных могут только выиграть от правильно выбранного единого стиля форматирования данных, а этот стиль зависит от принятых в вашей организации правил. Скажем, следует ли заключать код города в круглые скобки? А указывать код выхода на городскую линию с ведомственной АТС?

Чтобы начать разработку стиля, нужно провести некоторое время, всматриваясь в данные в поиске существующих стилевых несоответствий. После этого попытайтесь их согласовать и выработать единый стиль. В выработке единого стиля нет ничего магического — это просто один из инструментов приведения данных в соответствие друг другу.

Инструкция SELECT...INTO может выполнять множество полезных функций.

- Если из таблицы не извлекается ни одной строки, то инструкция SELECT...INTO создаст новую таблицу только со схемой данных.
- Если инструкция SELECT реорганизует столбцы или содержит функцию cast(), то новая таблица будет содержать данные с модифицированной схемой.
- В комбинации с оператором UNION инструкция SELECT...INTO может комбинировать по вертикали данные из нескольких таблиц. При этом оператор INTO должен находиться в первой инструкции SELECT объединения.
- Инструкция SELECT...INTO особенно полезна для денормализации таблиц. Она может комбинировать данные из разных таблиц и помещать их в одну.



При использовании инструкции SELECT...INTO нельзя попадаться в одну ловушку — эта инструкция не способна заменить объединения и представления. Когда создается новая таблица, она представляет собой всего лишь мгновенный снимок данных (проще говоря, вторую их копию). Базы данных, содержащие многочисленные копии старых наборов данных, являются источником постоянных проблем. Если вам нужно денормализовать данные для одноразового анализа или для передачи некоторому пользователю, то создание представления является куда лучшей альтернативой.

Обновление данных

Скажу без преувеличения, что инструкция UPDATE является исключительно мощным инструментом. То, что раньше занимало десятки программных строк и несколько вложенных циклов, теперь можно реализовать с помощью всего одной инструкции. SQL нельзя назвать в полной мере настоящим командным языком — он является декларативным. Код SQL всего лишь ставит задачу оптимизатору запросов. После этого оптимизатор разрабатывает наилучший план получения ответа на заданный вопрос и выполняет сформированную задачу. При этом он оценивает, сколько использовать таблиц и в каком порядке, как их объединять и какие индексы использовать. Он учитывает статистику заполнения таблиц и их размеры; производительность процессора и дисковой системы, емкость памяти. Только на основе всей этой информации он способен сформировать план выполнения запроса. Написание программ, обновляющих последовательно строки, никогда не позволило бы добиться такого уровня оптимизации.

Обновление одной таблицы

Инструкция UPDATE языка SQL предельно проста. Она способна изменить значение всего одной ячейки, а также всех столбцов всех строк таблицы. Однако предложение FROM позволяет таблице стать частью составного источника данных и использовать всю силу инструкции SELECT.

Вот как работает инструкция UPDATE:

```
UPDATE dbo.таблица
  SET столбец = значение/выражение/столбец,
     столбец = значение...
  [FROM источники_данных]
  [WHERE условия]
```

Инструкция UPDATE может изменить значения множества строк, но только одной таблицы. Ключевое слово SET используется для преобразования данных в новые значения. Эти новые значения могут быть константами, переменными, выражениями и даже столбцами другого источника данных, имеющегося в предложении FROM инструкции UPDATE.



Полный список возможностей выражений см. в главе 8.

Предложение WHERE жизненно важно для любой инструкции UPDATE, поскольку в его отсутствие обновляется вся таблица. Если предложение WHERE присутствует, то обновляются только те строки, которые им не отфильтрованы. Проверяйте и еще раз проверяйте предложение WHERE. Не считайте меня занудным, но я еще раз повторю: "Семь раз отмерь, один отрежь".

В следующем примере инструкция UPDATE выполняет типичную операцию из реальной жизни — она изменяет значение одного столбца одной строки. Лучше всего отфильтровать одну строку с помощью первичного ключа:

```
USE CHA2
UPDATE dbo.Guide
  SET Qualifications = 'Spelunking, Cave Diving,
     First Aid, Navigation'
  Where GuideID = 6
```

Проверим результаты обновления:

```
SELECT GuideID, LastName, Qualifications
  FROM dbo.Guide
  WHERE GuideID = 6
```

Будет получен следующий результат:

```
GuideID     LastName   Qualifications
----------- ---------- ----------------------------
6           Bistier    Spelunking, Cave Diving,
                         First Aid, Navigation
```

Выполнение глобального поиска и замены

Очистка базы данных от "мусора" представляет собой типичную задачу ее администратора. К счастью, SQL содержит функцию replace(), которая в комбинации с инструкцией UPDATE может помочь в глобальном поиске и замене.

В следующем примере, который ссылается на учебную базу данных Family, мы будем искать вхождения двойной буквы l (т.е. "ll") в фамилии и заменять их на "qua":

```
Use Family
Update Person
  Set LastName = Replace(Lastname, 'll', 'qua')
```

Теперь проверим результат выполнения инструкции, извлекая фамилии (результирующий список сокращен):

```
Select lastname from Person

lastname
---------------
Ha**qua**oway
Ha**qua**oway
Mi**qua**er
Mi**qua**er
Ha**qua**oway
...
```

Ссылка на множество таблиц при обновлении данных

Более мощной функцией инструкции UPDATE является присвоение столбцу результата выражения, которое ссылается на тот же или другие столбцы или даже на другие таблицы.

Хотя выражения доступны и при работе с одной таблицей, чаще всего в них приходится ссылаться на данные, находящиеся вне обновляемой таблицы. Необязательное предложение FROM позволяет создавать объединения между обновляемой таблицей и другими источниками данных. Обновляться может только одна таблица, однако будучи объединенной с другими, она может получать данные из их столбцов и использовать в выражениях обновления.

Одним из способов визуализации объединений, создаваемых в предложении FROM, является слияние всех таблиц в новый единый сверхширокий результирующий набор данных. После этого оставшаяся часть инструкции SQL работает уже с этим новым множеством. Несмотря на то что именно так и работает предложение FROM, реальная инструкция UPDATE работает не с этим множеством, а только с таблицей, указанной сразу после ключевого слова UPDATE.



Синтаксис инструкции UPDATE FROM входит в расширения T-SQL и не содержится в стандарте ANSI SQL-92. Если в будущем планируется перенос базы данных на другую платформу, то для обновления корректных строк используйте подзапросы.

Рассмотрим пример из реальной жизни. Предположим, что все сотрудники вскоре должны получить повышение зарплаты, размер которой зависит от подразделения, времени нахождения в текущей должности, производственных результатов и срока работы в компании (согласен, в реальной жизни такое вряд ли возможно). Если процент для каждого отдела хранится в таблице Department, то всего одна инструкция UPDATE позволяет объединить ее с таблицей Employee и учесть коэффициент размера премии. Предположим, что формула начислений следующая:

```
2+(((стаж_в_компании*0,1)+(месяцев_на_должности*0,02)
+ ((рейтинг_производительности*0,5) если больше 2))
* коэффициент_отдела)
```

В предлагаемом ниже сценарии создается несколько таблиц, которые заполняются данными, после этого тестируется формула и в результате выполняется обновление.

```
USE Tempdb
CREATE TABLE dbo.Dept (
  DeptID INT IDENTITY NOT NULL PRIMARY KEY NONCLUSTERED,
  DeptName VARCHAR(50) NOT NULL,
  RaiseFactor NUMERIC(4,2)
  )
  ON [Primary]
go
Create TABLE dbo.Employee (
  EmployeeID INT IDENTITY NOT NULL PRIMARY KEY NONCLUSTERED,
  DeptID INT FOREIGN KEY REFERENCES Dept,
  LastName VARCHAR(50) NOT NULL,
  FirstName VARCHAR(50) NOT NULL,
  Salary INT,
  PerformanceRating NUMERIC(4,2),
  DateHire DATETIME,
  DatePosition DATETIME
  )
  ON [Primary]
```

```
go
-- создание данных примера
INSERT dbo.Dept VALUES ('Engineering', 1.2)
INSERT dbo.Dept VALUES ('Sales',.8)
INSERT dbo.Dept VALUES ('IT',2.5)
INSERT dbo.Dept VALUES ('Manufacturing',1.0)
go
INSERT dbo.Employee
  VALUES( 1,'Smith', 'Sam',54000, 2.0, '1/1/97', '1/4/2001' )
INSERT dbo.Employee
  VALUES( 1,'Nelson','Slim',78000,1.5, '1/9/88', '1/1/2000' )
INSERT dbo.Employee
  VALUES( 2,'Ball', 'Sally',45000,3.5,' '1/8/99', '1/1/2001' )
INSERT dbo.Employee
  VALUES( 2, 'Kelly', 'Jeff',85000,2.4, '1/10/83', '1/9/1998' )
INSERT dbo.Employee
  VALUES( 3, 'Guelzow', 'Jo',120000,4.0, '1/7/95', '1/6/2001' )
INSERT dbo.Employee
  VALUES( 3, 'Anderson', 'Missy',95000,1.8, '1/2/99', '1/9/97' )
INSERT dbo.Employee
  VALUES( 4,'Reagan', 'Frank',75000,2.9, '1/4/00', '1/4/2000' )
INSERT dbo.Employee
  VALUES( 4, 'Adams', 'Hank',34000,3.2, '1/9/98', '1/9/1998' )
```

Предполагая, что датой повышения зарплаты является 1/5/2002, следующий запрос протестирует данные примера:

```
SELECT LastName, Salary,
  DateDiff(yy, DateHire, '1/5/2002') as YearsCo,
  DateDiff(mm, DatePosition, '1/5/2002') as MonthPosition,
  CASE
    WHEN Employee.PerformanceRating >= 2
      THEN Employee.PerformanceRating
    ELSE 0
  END as Performance,
  Dept.RaiseFactor AS 'Dept'
  FROM dbo.Employee
  JOIN dbo.Dept
    ON Employee.DeptID = Dept.DeptID
```

Должен быть получен следующий результат:

```
LastName  Salary  YearsCo MonthPosition Performance Dept
--------- ------- ------- ------------- ----------- ------
Smith     54000   5       13            2.00        1.20
Nelson    78000   14      28            .00         1.20
Ball      45000   3       16            3.50        .80
Kelly     85000   19      44            2.40        .80
Guelzow   120000  7       11            4.00        2.50
Anderson  95000   3       56            .00         2.50
Reagan    75000   2       25            2.90        1.00
Adams     34000   4       44            3.20        1.00
```

На основании заполненных данных выполним запрос, вычисляющий коэффициент повышения зарплаты:

```
SELECT LastName,
  (2 + (((DateDiff(yy, DateHire, '1/5/2002') * .1)
   + (DateDiff(mm, DatePosition, '1/5/2002') * .02)
   + (CASE
       WHEN Employee.PerformanceRating >= 2
```

```
       THEN Employee.PerformanceRating
      ELSE 0
     END * .5 ))
  * Dept.RaiseFactor))/100 as EmpRaise
  FROM dbo.Employee
  JOIN dbo.Dept
    ON Employee.DeptID = Dept.DeptID
```

Будет получен следующий результат:

```
LastName       EmpRaise
-------------- ---------------
Smith          .041120000
Nelson         .043520000
Ball           .038960000
Kelly          .051840000
Guelzow        .093000000
Anderson       .055500000
Reagan         .041500000
Adams          .048800000
```

Итак, данные внесены, и формулы проверены. Теперь пришло время выполнить инструкцию UPDATE и скорректировать зарплату сотрудников:

```
UPDATE Employee SET Salary = Salary * (1 +
  (2 + (((DateDiff(yy, DateHire, '1/5/2002') * .1)
   + (DateDiff(mm, DatePosition, '1/5/2002') * .02)
   + (CASE
       WHEN Employee.PerformanceRating >= 2
         THEN Employee.PerformanceRating
       ELSE 0
      END * .5 ))
  * Dept.RaiseFactor))/100 )
  FROM dbo.Employee
  JOIN dbo.Dept
    ON Employee.DeptID = Dept.DeptID
```

Следующая инструкция SELECT позволит нам увидеть плоды своего труда:

```
SELECT FirstName, LastName, Salary
  FROM Employee
```

А вот и сам результат:

```
FirstName   LastName      Salary
----------- ------------ -----------
Sam         Smith        56220
Slim        Nelson       81394
Sally       Ball         46753
Jeff        Kelly        89406
Dave        Guelzow      131160
Missy       Anderson     100272
Frank       Reagan       78112
Hank        Adams        35659
```

В завершение нашего примера очистим созданные учебные таблицы:

```
DROP TABLE dbo.Employee
DROP TABLE dbo.Dept
```

В приведенном выше примере были сведены воедино операции, описанные в нескольких предыдущих главах, в том числе создание и удаление таблиц, выражения CASE, объединения, скалярные функции работы с датами, не говоря уже об операциях вставки и обновления, описанных в настоящей главе. Пример был достаточно длинным, поскольку продемонстрировал

больше, нежели одну инструкцию UPDATE. В нем был показан типовой процесс проектирования сложных обновлений, который включает несколько этапов.

1. **Проверка доступных данных**. Первая инструкция SELECT объединила таблицы Employee и Dept и вывела список всех столбцов, необходимых формуле.
2. **Тестирование формулы**. Вторая инструкция SELECT базировалась на первой и применила формулу к нужным столбцам. На основании полученных данных можно вручную проверить результаты по избранным позициям и таким образом проверить формулу.
3. **Выполнение обновления**. Как только формула создана и проверена, ее можно подставить в инструкцию UPDATE, которую, в свою очередь, следует выполнить.

Инструкцию UPDATE нельзя не назвать мощной. Я заменил исключительно сложные наборы данных и вложенные циклы, выполняющиеся медленно и к тому же подверженные ошибкам, всего одной инструкцией UPDATE с грамотно созданным объединением. В результате время выполнения сократилось с нескольких минут до считанных секунд. В то же время я нисколько не умаляю достоинства подхода к обновлению с помощью наборов, а не строк данных.

Удаление данных

Инструкция DELETE очень опасна. В своей простейшей форме она удаляет все строки таблицы. Так как эта инструкция работает с целыми строками, ей не требуется явного указания столбцов. Первое предложение FROM не является обязательным, равно как и второе предложение WHERE. Но несмотря на то, что предложение WHERE не обязательно, оно прежде всех остальных заботится о том, какие именно строки будут удаляться из таблицы. Вот сокращенный синтаксис инструкции DELETE:

```
DELETE [FROM] владелец.таблица
[FROM источники_данных]
[WHERE условия]
```

Обратите внимание на то, что все в этой инструкции является необязательным, за исключением разве что самого ключевого слова DELETE и имени таблицы. Следующая инструкция удалит все данные из таблицы Product — никаких вопросов она не задаст и никакого второго шанса не предоставит:

```
DELETE
  FROM OBXKites.dbo.Product
```

SQL Server не имеет встроенной команды отмены операции. Как только транзакция завершена, все изменения становятся свершившимся фактом. Именно поэтому в инструкции DELETE такое большое значение имеет предложение WHERE.

До сих пор самым распространенным местом применения инструкции DELETE было удаление одной строки. Механизмом выбора этой строки обычно служил первичный ключ:

```
USE OBXKites
DELETE FROM dbo.Product
  WHERE ProductID = 'DB8D8D60-76F4-46C3-90E6-A8648F63C0F0'
```

Ссылка при удалении на множество таблиц

Инструкция UPDATE использует предложение FROM для объединения обновляемой таблицы с другими для более гибкого отбора строк. Инструкция DELETE использует ту же методику. В то же время немного странный вид ей придает первое необязательное предложение

FROM. Чтобы сделать текст программы более понятным и последовательным, я бы рекомендовал опускать это предложение.

Например, следующая инструкция DELETE игнорирует первое предложение FROM и использует только второе для объединения таблицы Product с таблицей ProductCategory, чтобы предложение WHERE имело возможность отфильтровать нужные строки по полю названия категории (ProductCategoryName). Предлагаемый в качестве примера запрос удаляет все строки, относящиеся к видео, из таблицы Product:

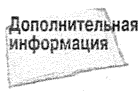
```
DELETE Product
  FROM dbo.Product
  JOIN ProductCategory
    ON Procduct.ProductCategoryID
       = ProductCategory.ProductCategoryID
  WHERE ProductcategoryName = 'Video'
```



Как и в предложении FROM инструкции UPDATE, второе предложение FROM инструкции DELETE не относится к стандарту ANSI SQL. Если для вашего проекта имеют значение вопросы переносимости, то для ссылок на дополнительные таблицы используйте подзапросы.

Каскадные удаления

Ссылочная целостность подразумевает тот факт, что ни одна вторичная строка не может ссылаться на первичную, которой на самом деле не существует. Это значит, что любая попытка удаления первичной строки завершится ошибкой, если какое-либо значение внешнего ключа ссылается на нее.



Подробно вопрос ссылочной целостности и ее использования разбирается в главах 2 и 17.

Механизм поддержания ссылочной целостности будет блокировать операции, которые ее нарушают. Единственным способом обхода этого ограничения является изначальное удаление вторичных строк, а затем уже первичной, на которую они ссылаются. Такая технология носит название *каскадного удаления*. В крупных базах данных такой каскад может содержать множество уровней.

Реализация механизма каскадного удаления вручную может быть очень трудоемкой работой. Так как ограничения внешнего ключа проверяются до выполнения триггеров, триггеры каскадного удаления не работают с механизмом декларативной ссылочной целостности (DRI). Таким образом, эти триггеры должны не только обслуживать каскадные удаления, но и выполнять проверку ссылочной целостности.

К счастью, в версии SQL Server 2000 каскадные удаления предложены как функция внешнего ключа. Каскадные удаления можно включить в Management Studio (рис. 16.2), а также с помощью кода T-SQL.

В примере сценария, который создает учебную базу данных Cape Hatteras Adventures версии 2 (CHA2_Create.sql), содержится отличный пример настройки каскадного удаления для поддержания ссылочной целостности. В данном случае, если удаляется событие или маршрут, удаляются также и соответствующие строки связывающей таблицы, реализующей отношение "многие ко многим". Параметр ON DELETE CASCADE внешнего ключа определяет такой режим удаления:

```
CREATE TABLE dbo.Event_mm_Guide (
  EventGuideID
```

```
  INT IDENTITY NOT NULL PRIMARY KEY NONCLUSTERED,
EventID
  INT NOT NULL
  FOREIGN KEY REFERENCES dbo.Event ON DELETE CASCADE,
GuideID
  INT NOT NULL
  FOREIGN KEY REFERENCES dbo.Guide ON DELETE CASCADE,
LastName
  VARCHAR(50) NOT NULL,
)
ON [Primary]
```

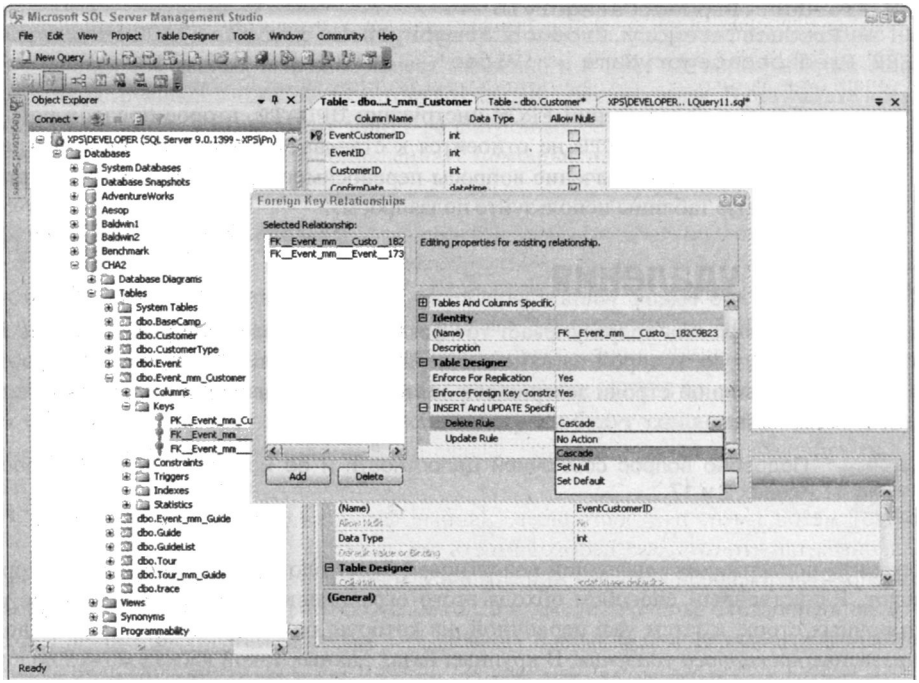


Рис. 16.2. Настройка в Management Studio внешних ключей для каскадного удаления

Следует особо отметить, что каскадные удаления и даже ссылочная целостность подходят не ко всем отношениям. Все зависит от постоянства вторичных строк. Если удаление первичной строки делает вторичную бессмысленной, то каскадное удаление имеет смысл. В то же время, если вторичная строка сохраняет свое назначение даже после удаления первичной, каскадное удаление и средства поддержания ссылочной целостности только разрушат связь базы данных с реальностью.

В качестве примера такой ситуации можно привести операцию удаления тура из базы данных Cape Hatteras Adventures. В этом случае все события, связанные с туром, теряют смысл, равно как и относящиеся к нему строки связующих таблиц (между событием и заказчиком и между событием и гидом).

Любой тур должен иметь базовый лагерь, так что для таблицы `Tour` требуется поддержание ссылочной целостности. В то же время, если удаляется какой-либо базовый лагерь, тур, начинающийся в нем, должен остаться доступным (он может быть назначен другому базовому лагерю). Таким образом, каскадное удаление от базового лагеря к туру не обосновано.

Если существует поддержка ссылочной целостности, а каскадные удаления отключены, то базовый лагерь не может быть удален, пока все его туры либо не будут удалены, либо не будут назначены другому лагерю.

Альтернативы физическому удалению данных

Многие разработчики стараются избегать удаления данных. Вместо этого они создают систему удаления данных из пользовательских представлений, оставляя их в исходных таблицах. Это можно реализовать несколькими способами.

- В строке может существовать логический битовый флажок, указывающий на то, что строка удалена. Такой подход упрощает удаление и восстановление одной строки — для этого нужно всего лишь изменить одно битовое значение. Но поскольку реляционная база данных содержит множество связанных между собою таблиц, работы оказывается больше, чем кажется на первый взгляд. Все запросы должны проверять наличие флага логического удаления и фильтровать соответствующим образом строки. К тому же, так как строки остаются физически существовать в SQL Server, а система поддержания ссылочной целостности сервера вообще не знает о существовании флага удаления, может потребоваться поддержание целостности на программном уровне. Ее сложность зависит от того, насколько далеко вы хотите распространить логическое удаление. Этот метод ускоряет операции логического удаления, однако сильно замедляет выполнение отбора записей. Каскадное логическое удаление может стать очень сложным, а восстановление записей — вообще превратиться в кошмар.
- Еще одна альтернатива сводится к физическому архивированию удаленных строк в другую таблицу или базу данных. Этот метод хорошо реализуется с помощью хранимых процедур, которые вставляют удаленные строки в архив, а затем удаляют их из основной базы данных.

  Этот метод имеет ряд достоинств. Данные физически удаляются из основной базы данных, так что отпадает необходимость искусственно изменять инструкции SELECT. Используя распределенные представления или федеративную схему базы данных, можно упростить включение в запросы отбора информации из нескольких баз данных. Физическое удаление данных позволяет без помех работать системе поддержания ссылочной целостности. К этому следует добавить, что основная база данных не засоряется ненужной информацией. Извлечение архивных данных остается при этом простым, хотя на администратора поддержка архива возлагает дополнительную нагрузку.

Дополнительная информация

В главе 53 вы больше узнаете о распределенных представлениях, а вся часть V будет посвящена стратегиям поддержки хранилищ данных и доставки информации из них конечным пользователям.

- Наиболее совершенной альтернативой удаления строк является поддержание полного аудита операций модификации данных. Это позволит не только отследить полную цепочку обновлений, но и восстановить удаленные строки. В то же время ведение аудита влечет появление накладных расходов, связанных со сложностью, производительностью и объемом хранимых данных.

Дополнительная информация

В главе 24 описано, как создать триггеры, выполняющие каскадные удаления, поддерживающие ссылочную целостность, создающие журналы аудита, архивирующие данные и выполняющие логическое удаление строк.

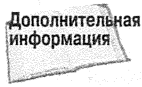
Возвращение модифицированных данных

SQL Server 2005 может при необходимости возвращать модифицированные данные для последующего использования. Это может пригодиться, когда с этими данными должна быть проведена дополнительная работа. Также эти данные могут потребоваться клиентскому приложению для уменьшения нагрузки на сервер.



Возможность немедленно возвращать результаты без необходимости привлечения других инструкций может на первый взгляд показаться несущественной. В то же время эта новая функция T-SQL, скорее всего, станет фаворитом при решении реальных практических задач.

Предложение OUTPUT может получить доступ к вставляемым и удаляемым виртуальным таблицам, извлекая из них возвращаемые данные. Представления вставляемых и удаляемых таблиц до и после выполнения транзакции при обычных условиях используются только триггерами. Виртуальная таблица удаления содержит все старые данные, а виртуальная таблица вставки — только что вставленные или обновленные данные. Предложение OUTPUT способно извлечь столбцы из этих таблиц — все или только избранные.



Более подробно о виртуальных таблицах удаления и вставки мы поговорим в главе 23.

Возвращение данных из операции вставки

Инструкция INSERT открывает доступ к виртуальной таблице вставки. Следующий пример, взятый из приведенного выше материала главы, был отредактирован для вставки предложения OUTPUT. Виртуальная таблица Inserted содержит вставленные данные и возвращает их:

```
USE CHA2;
INSERT dbo.Guidelist (LastName, FirstName, Qualifications)
OUTPUT Inserted.*
VALUES('Nielsen', 'Paul','trainer')
```

Результат выполнения пакета следующий:

```
GuideID LastName FirstName Qualifications DateOfBirth DateHire
------- -------- --------- -------------- ----------- ----------
7       Nielsen  Paul      trainer        NULL        NULL
```

Возвращение данных из операции обновления

Предложение OUTPUT также работает и с инструкцией UPDATE и может вернуть снимок данных до и после транзакции. В приводимом примере виртуальная таблица удаления используется для снятия исходных данных, а виртуальная таблица вставки содержит вставляемые значения. Возвращаются только старое и новое значения столбца Qualifications:

```
USE CHA2;
UPDATE dbo.Guide
  SET Qualifications = 'Scuba'
    OUTPUT Deleted.Qualifications as OldQuals,
       Inserted.Qualifications as NewQuals
  WHERE GuideID = 3
```

Результат выполнения инструкции следующий:

```
OldQuals NewQuals
-------- --------
NULL     Scuba
```

Возвращение данных из инструкции удаления

При удалении данных только виртуальная таблица удаления содержит сколько-нибудь полезные данные:

```
DELETE dbo.Guide
  OUTPUT Deleted.GuideID, Deleted.LastName, Deleted.FirstName
  WHERE GuideID = 3
```

Результат выполнения инструкции следующий:

```
GuideID LastName FirstName
------- -------- -------------
3       Wilson   Sam
```

Возвращение данных в переменной @Table

Программистам T-SQL предложение OUTPUT может вернуть данные для последующего использования в пакете или хранимой процедуре. В то же время эти данные можно поместить и в табличную переменную. Несмотря на то что синтаксис вам может показаться похожим на инструкцию INSERT...INTO, на самом деле эти операции функционируют совсем по-разному.

В приведенном ниже примере предложение OUTPUT передает свой результат табличной переменной @DeletedGuides.

```
DECLARE @DeletedGuides TABLE (
  GuideID INT,
  LastName VARCHAR(50),
  FirstName VARCHAR(50)
  );
DELETE dbo.Guide
  OUTPUT Deleted.GuideID, Deleted.LastName, Deleted.FirstName
  INTO @DeletedGuides
  WHERE GuideID = 2
```

Промежуточный результат будет следующим:

```
(1 row(s) affected)
```

Продолжаем выполнение пакета:

```
SELECT * FROM @DeletedGuides
```

Окончательный результат:

```
(1 row(s) affected)
GuideID     LastName   FirstName
----------- ---------- ----------------
2           Frank      Ken
```

Потенциальные препятствия на пути модификации данных

Даже допуская, что логика операций абсолютно корректна, а инструкции модификации данных на самом деле изменяют корректные значения, в любой момент все может пойти под откос. В этом разделе мы исследуем несколько типов потенциальных проблем и найдем пути, как их избежать.

Как показано в табл. 16.2, инструкции INSERT и UPDATE имеют больше подводных камней, чем инструкция DELETE, так как вставляют в таблицу данные, которые должны следовать множеству определенных правил. Инструкция DELETE только удаляет данные, поэтому и число возможных ловушек невелико.

Таблица 16.2. Потенциальные проблемы модификации данных

Потенциальная проблема	Инструкция INSERT	Инструкция UPDATE	Инструкция DELETE
Тип и длина данных	X	X	
Первичный ключ	X	X	
Внешний ключ	X	X	X
Уникальный индекс	X	X	
Запрет пустых значений и отсутствие умолчаний	X	X	
Проверка ограничений	X	X	
Триггер INSTEAD OF	X	X	X
Триггер AFTER	X	X	X
Необновляемые представления	X	X	X
Представления с параметром CHECK	X	X	
Безопасность	X	X	X

Проблема типа и длины данных

Тип и длина столбцов данных могут влиять на результаты инструкций INSERT и UPDATE.

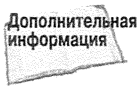
Одной из первых проверок, которые должны проходить новые данные, является проверка на их тип и длину. Часто ошибка типа данных вызвана отсутствием кавычек или, наоборот, их наличием. SQL Server выполняет неявное (т.е. автоматическое) преобразование данных. В то же время преобразования, которые автоматически выполняются в других языках программирования, могут не сработать в SQL Server. Приведем показательный пример:

```
USE OBXKites
INSERT Price (ProductID, Price, EffectiveDate)
  Values ('DB8D8D60-76F4-46C3-90E6-A8648F63C0F0',
     '15.00', 6/25/2002 )

Server: Msg 260, Level 16, State 1, Line 1
Disallowed implicit conversion from data type varchar
```

```
to data type money, table 'OBXKites.dbo.Price',
column 'Price'.
Use the CONVERT function to run this query.
```

Проблема в приведенном примере заключалась в излишних кавычках, обрамляющих новое значение цены. СУБД SQL Server не способна автоматически преобразовать строковое значение в числовое. Если вам необходимо такое преобразование, воспользуйтесь функцией `Convert()` или `Cast()`.



Подробно о типах данных и таблицах речь пойдет в главе 17. О преобразовании типов данных и связанных с этой операцией функциях см. в главе 8.

Проблемы первичного ключа

Первичные ключи могут влиять на инструкции `INSERT` и `UPDATE`.

Первичные ключи по определению должны быть уникальными. Попытка вставить значение первичного ключа, которое уже используется, вызовет ошибку.

С технической точки зрения обновление первичного ключа уже существующим значением также вызовет ошибку. Однако если вы следуете практическим указаниям, то будете использовать автоматически генерируемые первичные ключи, которые бессмысленны для обычного человека и которые нет никакого смысла обновлять.

Обновление первичного ключа может также разрушить ссылочную целостность базы данных, в результате чего инструкция `UPDATE` не будет выполнена. Однако в данном случае причиной сбоя будет не сам первичный ключ, а ограничение внешнего ключа, ссылающегося на него.



Подробно о создании первичных ключей см. в главе 2. К этой теме мы также вернемся в главе 17.

Одним из конкретных вопросов, связанных со вставкой, является создание первичных ключей для новый строк. SQL Server предлагает два отличных способа генерирования первичных ключей: *столбцы идентичности* и *глобальные универсальные идентификаторы* (*GUID*). Каждый из этих методов имеет свои достоинства и недостатки, так что к их выбору следует подходить внимательно.

Столбцы идентичности представляют собой генерируемые сервером последовательные целые числа. Они генерируются в ходе выполнения операций вставки, и сама инструкция `INSERT` никак не может повлиять на этот процесс, даже если будет содержать значение для столбца идентичности.

На самом деле тот факт, что столбцы идентичности отвергают поставляемые им значения, может создать серьезные проблемы, особенно если на такие же значения первичного ключа уже ссылаются вторичные таблицы. В учебной базе данных басен Эзопа первичные ключи "прошиты" в инструкции `INSERT` в сценариях заполнения, так как они заранее известны в операциях преобразования данных.

Решение проблемы сводится к использованию параметра базы данных `identity_insert`. Когда этот параметр установлен в значение `on`, защита столбца идентичности временно отключается, что позволяет вставлять в него свои данные. В базе данных параметр `identity_insert` может быть одновременно установлен в значение `on` только для одной таблицы. Следующий пакет инструкций SQL использует параметр `identity_insert` для заполнения первичного ключа.

```
USE CHA2
-- attempt to insert into an identity column
INSERT dbo.Guide (GuideID, FirstName, LastName)
  VALUES (10, 'Bill', 'Fletcher')
```

В результате будет получена ошибка:

```
Server: Msg 544, Level 16, State 1, Line 1
Cannot insert explicit value for identity column in table
'Guide' when IDENTITY_INSERT is set to OFF.
```

Теперь установим параметр `identity_insert` и попробуем выполнить другие вставки:

```
SET IDENTITY_INSERT Guide On
INSERT Guide (GuideID, FirstName, LastName)
  VALUES (100, 'Bill', 'Mays')
INSERT dbo.Guide (GuideID, FirstName, LastName)
  VALUES (101, 'Sue', 'Atlas')
```

Чтобы увидеть, какие значения теперь будут присвоены столбцу идентичности, снова отключим параметр `identity_insert` и выполним еще одну вставку:

```
SET IDENTITY_INSERT Guide Off
INSERT Guide ( FirstName, LastName)
VALUES ('Arnold', 'Bistier')
SELECT GuideID, FirstName, LastName
FROM dbo.Guide
```

Теперь посмотрим на результат:

```
GuideID FirstName  LastName
------- ---------- -------------
1       Dan        Smith
2       Jeff       Davis
3       Tammie     Commer
4       Lauren     Jones
5       Greg       Wilson
100     Bill       Mays
101     Sue        Atlas
102     Arnold     Bistier
```

Как было продемонстрировано, вставка вручную идентификатора с номером 101 привела к тому, что следующему было присвоено значение 102.

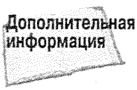
Еще одна проблема, связанная со столбцом идентичности, заключается в том, что достаточно тяжело узнать только что созданный идентификатор. Так как SQL Server генерирует это значение в ходе выполнения инструкции `INSERT`, программа, выполняющая вставку строки, его не знает. Сама инструкция `INSERT` проблем не вызывает — все выявляется тогда, когда сгенерированное значение приходится отображать в интерфейсе пользователя.

SQL Server предлагает три метода получения значения идентификатора.

- `@@identity`. Эта глобальная переменная возвращает последнее значение столбца идентичности, сгенерированное сервером для таблицы, подключения или набора данных. Однако если с момента вашей вставки произошла еще одна ошибка, то переменная `@@identity` вернет не нужное вам значение, а более позднее.
- `scope_identity()`. Эта системная функция возвращает последнее сгенерированное значение столбца идентичности в процессе выполнения текушей процедуры или пакета. Лично я рекомендую использовать именно этот метод получения последнего сгенерированного значения столбца идентичности.
- `ident_current(`*таблица*`)`. Эта функция возвращает последнее значение столбца идентичности, сгенерированное для указанной таблицы. Несмотря на кажущуюся схо-

жесть с функцией scope_identity(), данная функция не учитывает вставки значений идентичности, выполненные в других таблицах. Таким образом, вы получаете гарантированно корректный результат.

Глобальные уникальные идентификаторы GUID великолепно справляются с ролью первичных ключей. Их основное отличие от значений идентичности заключается в том, что они генерируются кодом SQL или с помощью значений столбца по умолчанию, а не во время выполнения самой вставки. Таким образом, разработчик получает больший контроль над созданием этих идентификаторов. Если значение вставляется в столбец, для которого определено значение по умолчанию, то проблемы не возникнут. Значение по умолчанию используется только тогда, когда в инструкции вставки для столбца явно не указано значение. Идентификаторы GUID можно использовать и по другим причинам, однако самой основной я считаю простоту их использования.



Более подробно вопросы создания первичных ключей будут рассмотрены в главе 17. Там же приведена сравнительная характеристика GUID и столбцов идентичности.

Глобальные универсальные идентификаторы создаются с помощью функции newid(). Если значение по умолчанию для первичного ключа задано как NewID(), то новый идентификатор генерируется для каждой новой строки. К тому же функция newid() может быть указана непосредственно в списке инструкции INSERT...VALUES. Эта функция будет выполняться даже в составе выражения инструкции INSERT...SELECT, работающей с группой строк. В составе хранимой процедуры или приложения результат выполнения этой функции может быть сохранен в переменной. Впоследствии эта переменная может быть использована в инструкции INSERT...VALUES для вставки ее значения в новую строку. Любой из этих вариантов хорошо справляется со своей работой, и в приложении их можно комбинировать.

Преимуществом предварительного определения GUID до выполнения инструкции INSERT...VALUES является то, что программа уже будет знать значение первичного ключа еще до операции вставки новой строки. Гибкость использования глобального универсального идентификатора заключается в том, что даже если структура программы не допускает его предварительного определения, то идентификатор, генерируемый непосредственно в инструкции вставки, справится со своей работой не хуже.

В следующем примере продемонстрированы различные методы генерирования GUID для первичного ключа при вставке новых строк в таблицу ProductCategory базы данных OBXKites. В первом примере мы проверим работу функции newid().

```
USE OBXKites
Select NewID()
```

Получаем следующий результат:

```
5CBB2800-5207-4323-A316-E963AACB6081
```

Следующие три запроса вставляют в новые строки GUID, используя при этом различные методы:

```
-- GUID вставляется по умолчанию
--(значением по умолчанию для столбца является функция NewID())
INSERT dbo.ProductCategory
(ProductCategoryID, ProductCategoryName)
VALUES (DEFAULT, 'Из умолчания')

-- GUID вставляется функцией
INSERT dbo.ProductCategory
(ProductCategoryID, ProductCategoryName)
VALUES (NewID(), 'Из функции')
```

```
-- GUID вставляется из переменной
DECLARE @NewGUID UniqueIdentifier
SET @NewGUID = NewID()
INSERT dbo.ProductCategory
(ProductCategoryID, ProductCategoryName)
VALUES (@NewGUID, 'Из переменной')
```

Для просмотра результатов вставки мы отфильтруем только те строки, которые начинаются со слова "Из":

```
SELECT ProductCategoryID, ProductCategoryName
  FROM dbo.ProductCategory
    WHERE ProductCategoryName LIKE 'Из %'
```

Будет получен следующий результат:

```
ProductCategoryID                    ProductCategoryName
------------------------------------ ----------------------
25894DA7-B5BB-435D-9540-6B9207C6CF8F Из умолчания
393414DC-8611-4460-8FD3-4657E4B49373 Из функции
FF868338-DF9A-4B8D-89B6-9C28293CA25F Из переменной
```

Следующая инструкция INSERT использует функцию newid() для вставки нескольких идентификаторов GUID:

```
INSERT dbo.ProductCategory
    (ProductCategoryID, ProductCategoryName)
  Select NewID(), LastName
    From CHA2.dbo.Guide
```

Приведенная инструкция SELECT запрашивает новые идентификаторы:

```
SELECT ProductCategoryID, ProductCategoryName
  FROM dbo.ProductCategory
```

Теперь посмотрим на результаты:

```
ProductCategoryID                    ProductCategoryName
------------------------------------ --------------------
1B2BBE15-B415-43ED-BCA2-293050B7EFE4 Kite
23FC5D45-8B60-4800-A505-D2F556F863C9 Accessory
3889671A-F2CD-4B79-8DCF-19F4F4703693 Video
...
5471F896-A414-432B-A579-0880757ED097 Fletcher
428F29B3-111B-4ECE-B6EB-E0913A9D34DC Atlas
E4B7D325-8122-48D7-A61B-A83E258D8729 Bistier
```

SQL Server предлагает двух отличных кандидатов на роль первичных ключей — столбцы идентичности и GUID. То, какой из них будет использоваться, зависит от нескольких факторов. В любом случае у вас есть несколько методов вставки новых строк. Кем бы вы ни были — программистом или администратором базы данных, — ситуация всегда будет у вас под контролем.

Проблемы внешних ключей

Внешние ключи могут влиять на выполнение инструкций INSERT, UPDATE и DELETE — они могут блокировать эти операции. Вставка новой строки вторичной таблицы, имеющей внешний ключ, не соответствующий ни одному первичному, может просто не выполниться.

В следующем примере значение идентификатора категории товара (ProductCategoryID) не существует в таблице ProductCategory. В результате ограничение внешнего ключа блокирует выполнение операции вставки:

```
-- Внешние ключи: проблемы вставки
INSERT Product (ProductID, Code,
ProductCategoryID, ProductName)
VALUES ('9562C1A5-4499-4626-BB33-E5E140ACD2AC',
'999'
'DB8D8D60-76F4-46C3-90E6-A8648F63C0F0',
'Basic Box Kite 21"')

Server: Msg 547, Level 16, State 1, Line 1
INSERT statement conflicted with COLUMN FOREIGN KEY
constraint 'FK__Product__Product__7B905C75'.
The conflict occurred in database 'OBXKites',
table 'ProductCategory', column 'ProductCategoryID'.
The statement has been terminated.
```

Обратите внимание на то, что, так как идентификатор GUID является уникальным, в разных системах он будет генерироваться по-разному.

Ограничения внешних ключей могут блокировать операции обновления как в первичных, так и во вторичных таблицах. Если обновляется первичный ключ, а внешний указывает на старое значение, то операция завершится по ошибке.

В следующем примере операция обновления блокируется по той причине, что предпринимается попытка обновления значения внешнего ключа вторичной таблицы значением, которое не существует в первичной:

```
-- Внешние ключи: проблемы обновления вторичной таблицы
UPDATE Product
SET ProductCategoryID =
'DB8D8D60-76F4-46C3-90E6-A8648F63C0F0'
WHERE ProductID = '67804443-7E7C-4769-A41C-3DD3CD3621D9'

Server: Msg 547, Level 16, State 1, Line 1
UPDATE statement conflicted with COLUMN FOREIGN KEY
Constraint 'FK__Product__Product__7B905C75'.
The conflict occurred in database 'OBXKites',
table 'ProductCategory', column 'ProductCategoryID'.
The statement has been terminated.
```

Обновление первичного ключа новым значением, если внешний ключ указывает на него, имеет тот же эффект, что и удаление строки первичной таблицы, если вторичная на нее ссылается. В обоих случаях операция завершится ошибкой, и вызвана она будет ограничениями внешнего ключа.

В следующем примере ошибка вызвана не таблицей `ProductCategory`, даже несмотря на то, что обновляется именно она; ошибку провоцирует таблица `Product`. Таблица `Product` имеет ограничения внешнего ключа, и одно из его значений теперь должно ссылаться на значение первичной таблицы, которого больше не существует.

```
-- Внешние ключи: проблемы обновления первичной таблицы
UPDATE ProductCategory
SET ProductCategoryID =
'DB8D8D60-76F4-46C3-90E6-A8648F63C0F0'
WHERE ProductCategoryID =
'1B2BBE15-B415-43ED-BCA2-293050B7EFE4'

Server: Msg 547, Level 16, State 1, Line 1
UPDATE statement conflicted with COLUMN REFERENCE constraint
'FK__Product__Product__7B905C75'. The conflict occurred
in database 'OBXKites', table 'Product',
column 'ProductCategoryID'.
The statement has been terminated.
```

Дополнительная информация

Подробно о роли ссылочной целостности при выборе внешних ключей см. в главе 2. Сам процесс создания внешних ключей подробно описан в главе 17.

Проблемы уникальных индексов

Уникальные индексы могут влиять на выполнение инструкций `INSERT` и `UPDATE`.

Если некоторый столбец имеет уникальный индекс (необязательно даже ключ), то попытка вставки нового значения, которое уже существует, провалится.

Как правило, завершится неудачей даже вся транзакция, содержащая операции вставки или обновления таких строк. В то же время существует параметр `ignore dup key`, позволяющий транзакциям выполняться, запрещая только те операции, которые приводят к дублированию строк.

Дополнительная информация

Более подробно об ограничениях уникальных индексов вы узнаете в главах 17 и 50.

Проблемы пустых значений и значений по умолчанию

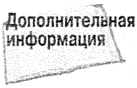
Допустимость пустых значений и значений по умолчанию столбцов могут оказывать влияние на выполнение инструкций `INSERT` и `UPDATE`.

Эти инструкции могут отправлять в таблицу одно из четырех возможных значений: данные, пустое значение, значение по умолчанию или вообще ничего. В свою очередь, столбец таблицы может быть сконфигурирован со значением по умолчанию и допустимостью пустых значений. В табл. 16.3 описаны результаты операций обновления и вставки с учетом различных конфигураций столбца. Например, если столбец имеет значение по умолчанию и допускает пустые значения, а операция обновления или вставки посылает пустое значение, в результате будет получена ошибка.

Таблица 16.3. Модификация данных, значения по умолчанию и пустые значения

Свойства столбца				
Значения по умолчанию	Нет	Нет	Есть	Есть
Допустимость пустых значений	Да	Нет	Да	Нет
Инструкция посылает			**Результат**	
данные	Данные	Данные	Данные	Данные
`null`	`null`	Ошибка	`null`	Ошибка
Значение по умолчанию	`null`	Ошибка	Значение по умолчанию	Значение по умолчанию
Ничего	`null`	Наиболее распространенная ошибка	Значение по умолчанию	Значение по умолчанию

Как вы увидели, самая распространенная ошибка происходит, когда в столбец ничего не посылается, а в нем не существует значения по умолчанию и не допустимы пустые значения.



Детальная информация о создании значений по умолчанию и конфигурировании пустых значений приведена в главе 17. О работе с пустыми значениями при извлечении данных см. в главе 8.

Проблемы ограничений проверки

Ограничения проверки могут оказать влияние на выполнение инструкций INSERT и UPDATE.

Каждый столбец таблицы может иметь несколько ограничений проверки. Как правило, это быстрые булевы операции, определяющие допустимость операции.

В следующем примере устанавливается ограничение, которое отсеивает лиц, не достигших на момент приема на работу 21-летнего возраста (следует отметить, что это ограничение уже применено при создании базы данных с помощью сценария `Create_CHA2.sql`):

```
USE CHA2
go
ALTER TABLE dbo.Guide ADD CONSTRAINT
  CK_Guide_Age21 CHECK (DateDiff(yy,DateOfBirth, DateHire)
    >= 21)
```

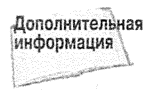
В следующем запросе в базу данных вставляется информация об экскурсоводе Мэри Джонсон. Так как она уже достигла 26-летнего возраста, вставка будет выполнена:

```
INSERT Guide(lastName, FirstName, Qualifications, DateOfBirth,
DateHire)
VALUES ('Johnson', 'Mary',
'E.R. Physician', '14/1/71', '1/6/97')
```

В противоположность этому Грег Франклин достиг только 19-летнего возраста, и ограничение проверки запретит вставку его данных:

```
INSERT Guide (lastName, FirstName,
Qualifications, DateOfBirth, DateHire)
VALUES ('Franklin', 'Greg',
'Guide', '12/12/83', '1/1/2002')

Server: Msg 547, Level 16, State 1, Line 1
INSERT statement conflicted with TABLE CHECK constraint
'CK_Guide_Age21'.
The conflict occurred in database 'CHA2', table 'Guide'.
The statement has been terminated.
```



Подробно об ограничениях проверки, их достоинствах и недостатках речь пойдет в главе 17.

Проблемы триггеров INSTEAD OF

Триггеры INSTEAD OF оказывают влияние на выполнение инструкций INSERT, UPDATE и DELETE.

Триггеры представляют собой особые хранимые процедуры, которые прикрепляются к таблице и вызываются при выполнении в ней каких-либо операций модификации данных. Существуют два типа триггеров: INSTEAD OF и AFTER. Отличаются они временем выполнения и отношением к операции модификации данных.

Триггеры INSTEAD OF выполняются *вместо* соответствующей операции; при этом сама инструкция отклоняется. Эти триггеры могут заменить собой операцию, проведя ее в корректной форме, также они могут выполнить какое-либо другое действие.

Проблема триггеров INSTEAD OF заключается в том, что они отчитываются о выполнении операции независимо от того, было ли что-либо записано в базу данных. В этом нет никакой ошибки, поскольку триггер все равно выполнялся, однако это вносит определенную путаницу в работу пользователей.

В следующем примере триггер InsteadOfDemo выполняется вместо инструкции вставки:

```
USE CHA2
go
CREATE TRIGGER InsteadOfDemo
ON Guide
INSTEAD OF INSERT
AS
  Print 'Пример триггера INSTEAD OF'
Return
```

Следующий запрос вставит указанную в триггере тестовую строку вместо фактически указанных в инструкции данных:

```
INSERT Guide(lastName, FirstName,
Qualifications, DateOfBirth, DateHire)
VALUES ('Jamison', 'Tom',
'Biologist, Adventurer', '14/1/56', '1/9/99')
```

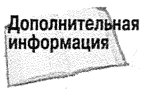
Будет получен следующий результат:

```
Пример триггера INSTEAD OF
(1 row(s) affected)
```

Инструкция INSERT должна была быть выполнена, но вставила ли она в таблицу строку? Проверим.

```
SELECT GuideID
  FROM Guide
  WHERE LastName = 'Jamison'

GuideID
-----------
(0 row(s) affected)
```



Создание триггеров будет описано в главе 23. Процесс выполнения транзакций модификации данных и время запуска триггеров рассмотрен в главе 51.

Следует отметить, что в приведенном примере триггер InsteadOfDemo перед своим созданием был удален.

Проблемы триггеров AFTER

Триггеры AFTER могут оказывать влияние на выполнение инструкций INSERT, UPDATE и DELETE.

Эти триггеры часто используют для осуществления сложной проверки вводимых данных. Если рассматриваемая операция вставки, обновления или удаления не будет одобрена триггером, она будет отменена. В программе также могут быть выполнены еще какие-либо операции. В то же время, если триггер явно не откатит операцию, она будет выполнена, как и

предполагалось изначально. В отличие от триггеров INSTEAD OF, триггеры AFTER отчитываются об ошибке, если инструкция была отменена.

В главе 51 мы обсудим все операции, неявно выполняемые в транзакциях, даже если они не содержит инструкции BEGIN TRANSACTION. Триггеры AFTER запускаются на выполнение сразу же после инструкции, но перед подтверждением операции. Таким образом, во время выполнения триггера AFTER транзакция считается открытой.

В следующем примере создается триггер AfterDemo, прикрепленный к таблице Guide. В этом триггере содержатся команды raiseerror и rollback transaction:

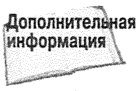
```
USE CHA2
CREATE TRIGGER AfterDemo
ON Guide
AFTER INSERT, UPDATE
AS
Print 'Пример триггера AFTER'
-- logic in a real trigger would decide what to do here
RAISERROR ('Пример ошибки', 16, 1 )
ROLLBACK TRAN
Return
```

В результате применения этого триггера к таблице Guide результат операции вставки будет следующим:

```
INSERT Guide(lastName, FirstName,
    Qualifications, DateOfBirth, DateHire)
  VALUES ('Harrison', 'Nancy',
    'Pilot, Sky Diver, Hang Glider,
       Emergency Paramedic', '25/6/69', '14/7/2000')

Пример триггера AFTER
Server: Msg 50000, Level 16, State 1,
Procedure AfterDemo, Line 7
Пример ошибки
```

Поиск строки с данными о Нэнси Харрисон ни к чему не приведет, поскольку транзакция была отменена.

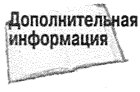


Более подробно триггеры AFTER будут описаны в главе 23. О дополнительных стратегиях создания триггеров вы узнаете в главе 24.

Следует обратить внимание, что программный код на компакт-диске, прилагаемом к книге, относящийся к настоящей главе, удаляет триггер AfterDemo, чтобы все остальные примеры могли работать.

Вычисляемые столбцы

С необновляемыми представлениями непосредственно связан вопрос вычисляемых столбцов — любая попытка записи данных в такой столбец обречена на неудачу.



Более подробно о создании вычисляемых столбцов вы узнаете в главе 17.

Проблемы необновляемых представлений

На возможность обновления представлений влияет несколько факторов. Самой распространенной причиной невозможности обновления является использование итоговых функций, группировок и объединений. Если представление содержит вложенные представления, хотя бы одно из которых является необновляемым, внешнее представление будет также необновляемым.

Представление vMedGuide, создаваемое в следующем примере, является необновляемым, так как предикат DISTINCT исключает дублирования, что, в свою очередь, не дает возможности серверу однозначно определить, какая именно строка обновляется:

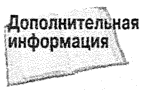
```
CREATE VIEW dbo.vMedGuide
AS
SELECT DISTINCT GuideID, LastName, Qualifications
  FROM dbo.Guide
  WHERE Qualifications LIKE '%Aid%'
  OR Qualifications LIKE '%medic%'
  OR Qualifications LIKE '%Physician%'
```

Для проверки возможности обновления представления выполним следующий запрос:

```
UPDATE dbo.vMedGuide
  SET Qualifications = 'E.R. Physician, Diver'
  WHERE GuideID = 1
```

Будет получен следующий результат:

```
Server: Msg 4404, Level 16, State 1, Line 1
View or function 'dbo.vMedGuide' is not updatable
because the definition contains the DISTINCT clause.
```

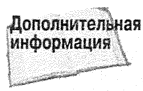


Подробно о создании представлений см. в главе 14. Там же вы найдете перечень причин, из-за которых невозможно обновлять представления.

Проблемы представлений с параметром проверки

Представления с параметром WITH CHECK OPTION влияют на выполнение инструкций INSERT и UPDATE.

Использование этого параметра может привести к появлению двух специфичных проблем. Одна из них, называемая *проблемой исчезающих столбцов*, возникает, когда обновленные столбцы больше не удовлетворяют критерию предложения WHERE. Эти столбцы никуда из базы данных не исчезают, просто представление перестает отображать их.



Подробно о параметре WITH CHECK OPTION и проблеме исчезающих столбцов см. в главе 14. Использование ролей системы безопасности будет рассмотрено в главе 40.

Добавление в представление параметра WITH CHECK OPTION может вызвать еще одну проблему. Он влияет на выполнение условия WHERE как для извлекаемых данных, так и для вставляемых и обновляемых. Если данные после операций вставки или обновления не способны отображаться в представлении, то параметр WITH CHECK OPTION может привести к невозможности выполнения соответствующей инструкции в целом.

В следующем примере в упомянутое уже ранее представление добавляется параметр WITH CHECK OPTION, после чего выполняются две попытки обновления данных. Первая из

них удовлетворяет требованиям предложения WHERE. Вторая операция пытается удалить строки из результирующего набора данных, который вернул представление, поэтому она завершается неудачей:

```
ALTER VIEW dbo.vMedGuide
AS
SELECT GuideID, LastName, Qualifications
  FROM dbo.Guide
  WHERE Qualifications LIKE '%Aid%'
  OR Qualifications LIKE '%medic%'
  OR Qualifications LIKE '%Physician%'
  WITH CHECK OPTION
```

Следующие запросы тестируют видоизмененное представление. Первый из них выполняется, поскольку содержит квалификацию Physician; второй завершается ошибкой:

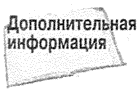
```
UPDATE dbo.vMedGuide
SET Qualifications = 'E.R. Physician, Diver'
WHERE GuideID = 1
UPDATE dbo.vMedGuide
SET Qualifications = 'Diver'
WHERE GuideID = 1

Server: Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target
view either specifies WITH CHECK OPTION or spans a view
that specifies WITH CHECK OPTION and one or more rows
resulting from the operation did not qualify
under the CHECK OPTION constraint.
The statement has been terminated.
```

Проблемы системы безопасности

Вопросы безопасности могут влиять на выполнение инструкций INSERT, UPDATE и DELETE.

К невозможности выполнения инструкции может привести множество настроек системы безопасности и ролей. Обычно вопросы безопасности не поднимаются на этапе создания программы. Однако, как только база данных запускается в промышленную эксплуатацию, защита данных часто выходит на первый план. Документирование настроек безопасности и ролей поможет вам решить большинство проблем, связанных с защитой данных.



Более подробно о системе безопасности и ролях мы поговорим в главе 40.



Большинство проблем модификации без труда могут предупредить программисты и администраторы баз данных, если они только понимают систему работы SQL Server, документируют базу данных, знакомы со схемой базы, с хранимыми процедурами и триггерами.

Резюме

Извлечение и модификация информации являются основными задачами приложений работы с базами данных. В этой главе мы рассмотрели инструкции языка DML `INSERT`, `UPDATE` и `DELETE`, а также вопросы их блокировки в базе данных.

На этом мы завершаем изучение инструкций DML, которым посвятили десяток глав. В следующих главах мы приступим к задачам программирования и проектирования баз данных в SQL Server с помощью T-SQL, CLR и новой архитектуры, ориентированной на службы.

Среда разработки SQL Server

Часть II была целиком посвящена пакетным запросам. В этой части мы продолжим изучение инструкции SELECT и рассмотрим вопросы управления потоком команд при создании серверных решений — от T-SQL до сборок .NET.

Начнется часть III с рассмотрения команд DDL (CREATE, ALTER и DROP); в последующих девяти главах мы еще больше углубимся в язык TRANSACT-SQL. А в последних главах этой части мы сведем воедино все элементы с уровнем абстракции данных.

Список технологий, реализованных в SQL Server, продолжает расти, и в завершающих главах данной части мы рассмотрим общеязыковую среду выполнения CLR, SQL Server Mobile, брокер служб, XML, XQuery, Web-службы и InfoPath.

ЧАСТЬ III

В этой части...

Глава 17
Реализация физической схемы базы данных

Глава 18
Программирование на языке Transact-SQL

Глава 19
Выполнение массовых операций

Глава 20
Курсоры

Глава 21
Создание хранимых процедур

Глава 22
Создание пользовательских функций

Глава 23
Реализация триггеров

Глава 24
Расширенные технологии T-SQL

Глава 25
Расширяемость с помощью уровня абстракции данных

Глава 26
Программирование для SQL Server Everywhere

Глава 27
Программирование сборок CLR в SQL Server

Глава 28
Создание запросов в брокере служб

Глава 29
Поддержка пользовательских типов данных

Глава 30
Программирование в ADO.NET 2.0

Глава 31
Использование XML, XPath и XQuery

Глава 32
Создание хранилищ данных SOA с помощью Web-служб

Глава 33
InfoPath и SQL Server 2005

ГЛАВА

17

Реализация физической схемы базы данных

В этой главе...

Создание файлов базы данных

Создание таблиц

Создание первичных и внешних ключей

Конфигурирование ограничений

Создание пользовательских столбцов данных

Документирование схемы базы данных

Триггеры DDL

Когда я служил в военно-морском флоте, то многому научился у капитана Миллера. Одна из его теорий заключалась в том, что любое приложение наполовину состоит из программного кода, а наполовину — из данных. За два с лишним десятка лет проектирования баз данных я более чем убедился в его правоте.

Как схема, так и сами данные часто более критичны для успешной реализации проекта, чем сама программа. Основные функции приложения разрабатываются на уровне схемы данных. Если схема данных поддерживает некоторую функцию, то программный код может воплотить ее в жизнь. В то же время, если функция не была спроектирована в таблицах, то и конечные формы, как бы хорошо они ни были запрограммированы, не смогут работать хорошо.

Логическая схема базы данных, которую мы рассматривали в главе 2, представляет собой в целом всего лишь академическое упражнение, нацеленное на понимание производственных требований. Логическая модель никогда не сохраняется и не обслуживает каких-либо данных. В противовес этому физическая схема является реальным хранилищем данных, которое должно удовлетворять одному из основных принципов информационной архитектуры, гласящему, что информация должна находиться в формате, удобном для повседневной эксплуатации и анализа. Команда, проектирующая физическую схему, должна заниматься не только вопросами целостности данных и удобства пользователей. Следует основательно подойти к вопросам производительности, переносимости, удобства выполнения запросов и обслуживания при реализации базы данных на конкретной платформе.

В этой главе мы обсудим вопросы проектирования физической схемы базы данных, после чего сосредоточим внимание на инструкциях CREATE, ALTER и DROP языка определения данных DDL. Именно эти три инструкции предназначены для проектирования физической схемы базы данных.



Все листинги, приведенные в настоящей главе, а также учебные базы данных доступны для загрузки на сайте `www.SQLServerBible.com`. Помимо этого вы можете найти расширенные запросы, используя представления каталогов, относящиеся к настоящей главе.

Проектирование физической схемы базы данных

При проектировании физической схемы следует начинать с чисто логической модели — хорошо разобраться в основных бизнес-правилах и задокументировать их. Только такой мозговой штурм поможет создать простую и гибкую модель, которая хорошо проявит себя в работе.

Реализация физической схемы базы данных включает в себя шесть компонентов.

- Создание файлов базы данных.
- Создание таблиц.
- Создание первичных и внешних ключей.
- Создание столбцов данных.
- Создание ограничений, гарантирующих целостность данных.
- Создание индексов (в принципе индексы могут быть без труда созданы и изменены уже после реализации физической схемы).

Преобразование логической схемы базы данных в физическую может включать несколько этапов.

- Преобразование сложного логического проекта в более простые и гибкие структуры таблиц.
- Преобразование составных первичных ключей в опирающиеся только на один столбец.
- Преобразование бизнес-логики в ограничения и триггеры.
- Преобразование логических отношений "многие ко многим" в пары физических отношений "один ко многим", использующих ассоциативные (или связывающие) таблицы.

Варианты проектирования физической схемы

Каждая команда проектировщиков создает физическую схему, исходя из логической, используя следующие возможные подходы.

- Логическая схема базы данных создается, а затем реализуется без оглядки на физическую схему.

  Такой подход является прямой дорогой к медленной и нерасширяемой схеме базы данных. Программный код будет исключительно тяжело писать. и все равно он вряд ли сможет преодолеть изъяны производительности, заложенные в проекте.

- Логическая схема базы данных создается при хорошем понимании требований бизнес-логики. Основываясь на логической модели, команда разработчиков создает физическую схему базы данных. Данный подход способен обеспечить создание быстрой и жизнеспособной схемы.

  Такой подход определяет создание схемы в два этапа, что требует привлечения относительно большого коллектива разработчиков. При этом одна группа программистов

собирает бизнес-правила, а вторая занимается их реализацией в физической схеме. Поверьте: наличие готовой логической схемы не заменит творческой работы команды специалистов при создании физической схемы.

- Третья комбинация методологий логического и физического проектирования объединяет весь процесс в один этап. При этом команда проектировщиков сразу создает физическую схему, основываясь на требованиях бизнес-логики. Этот метод хорошо себя проявит, если команда в полной мере понимает вопросы логического и физического моделирования, а также создания сложных запросов.

Ключевой задачей в проектировании физической схемы базы данных является рассмотрение нескольких моделей, каждая из которых удовлетворяет требованиям конечных пользователей и целостности данных. Каждый из проектов должен рассматриваться с точки зрения его простоты, быстродействия, гибкости и легкости обслуживания.

Корректировка модели данных

Ключ к простоте модели базы данных лежит в четком определении сущностей и их функциональности. Именно здесь на первый план выходит мозговой штурм, нацеленный на такую реорганизацию моделей данных, чтобы она стала как можно более простой и элегантной. В этой работе, как нигде, требуется опыт проектирования баз данных и работы с ними.

Часто решение сводится к рассмотрению данных с разных точек зрения и поиску в них общих элементов. Пользователи настолько плотно работают с данными, что редко способны идентифицировать корректные сущности. То, что пользователи видят как разные сущности, команда программистов может спроектировать как одну сущность с динамическими свойствами.

Комбинируя такой подход с методами проектирования данных, можно получить нормализованные базы данных с более высоким уровнем целостности данных, большей гибкостью, в то же время сократив общее количество таблиц.

Вопросы производительности

Нормализованная логическая модель базы данных, созданная без оглядки на оптимизацию физической схемы, никогда не даст хорошие результаты, так как при создании логической схемы редко рассматриваются вопросы производительности. Конкуренция блокировок, составные ключи, наличие множества объединений в наиболее часто используемых запросах, а также структуры таблиц, в которых тяжело выполнять обновления, — это лишь некоторые из проблем, которые может привнести в базу данных логический проект.

На производительность базы данных сильно влияет простота и сложность модели. Каждая новая сложность требует излишнего программирования, объединений и влечет за собой другие дополнительные сложности.

Одно из решений проблемы производительности заключается в правильном выборе первичных ключей. Логическая модель базы данных имеет тенденцию к наличию значимых составных первичных ключей. В то же время физическая схема только выиграет от разбиения этих ключей на использующие только один столбец (генерируемый компьютером). В одном из следующих разделов мы отдельно остановимся на этом вопросе.

Вопросы масштабируемости

Поддержка приложения на протяжении срока его службы требует гораздо больших затрат, чем само его создание. Таким образом, на этапе проектирования приложения нужно основательно заняться вопросами максимального облегчения обслуживания физической струк-

туры, программ и данных. Затраты на обслуживание могут в значительной мере сократить следующие методики.

- Использование последовательного соглашения об именах.
- Избежание создания сложных структур данных там, где с задачей могут справиться и более простые структуры.
- Использование сценариев, а не Management Studio.
- Избежание использования непереносимых расширений, не входящих в стандарт ANSI SQL, если, конечно, вы не собираетесь создавать весь конечный продукт на продуктах Microsoft.
- Поддержание целостности данных с помощью ограничений с самого начала разработки. Даже после короткого периода нарушения целостности данные могут сильно пострадать и потребовать чистки.
- Изначальная разработка ядра системы. Только после того, как оно заработает, можно добавлять дополнительные функции.
- Документирование не только того, *как* работает процедура, но и *почему* она работает именно так.

Ответственный подход к денормализации

Забавно, но функция проверки орфографии в моей программе Microsoft Word предлагает заменять слово "денормализация" словом "деморализация". Если речь идет о реляционной транзакционной базе данных, то мне к этому нечего добавить.

Денормализация, разрушающая нормальные формы, реализует технологию дублирования данных, чтобы упростить их извлечение.



О нормализации см. в главе 2.

Рассмотрим несколько примеров денормализации структуры данных. Если в таблицу заказов включить название клиента, то не будет потребности объединять эту таблицу с таблицей клиентов. Включение идентификатора клиента в таблицу строк заказа позволит также напрямую объединять ее с таблицей заказчиков, в обход таблицы заказов. Оба этих примера приводят к нарушению нормализации, поскольку добавляемые атрибуты не зависят от первичного ключа.

Некоторые разработчики регулярно денормализуют части базы данных в попытке увеличить производительность. Несмотря на то что количество объединений, необходимых в запросах, уменьшается, такая методика может существенно замедлить работу базы данных, так как для поддержания синхронизации дублирующихся данных потребуются дополнительные триггеры, а поддержание целостности данных усложнится.

Нормализация гарантирует целостность данных при их вводе и обновлении. Таким образом, рекомендации к денормализации некоторых частей базы данных зависят от цели, для которой создается эта база.

- Если данные используются для постоянных преобразований (т.е. на первое место выходит вопрос целостности данных) — не нужно заниматься денормализацией. Никогда не денормализуйте исходные данные.
- Денормализуйте итоговые данные, такие как балансы счетов или количество выданных на руки складских товаров, несмотря на то, что их можно вычислить с помощью таблицы складских транзакций. Эти данные можно получить с помощью триггера или специального вычисляемого столбца.

- Если данные не являются исходными и в основном предназначены для кубов OLAP или отчетов, то целостность данных не так важна. В этом случае денормализация будет лучшим способом повышения производительности.

Архитектура базы данных и то, какие таблицы для каких целей в ней использовать, являются движущей силой в принятии решения относительно денормализации части базы данных. Если база данных требует как OLTP, так и OLAP, то лучшим решением будет создание нескольких таблиц, которые дублируют данные для своих целей. Для OLTP нужны свои таблицы, отвечающие за поддержку данных, однако службе отчетности могут потребоваться аналогичные данные в собственной, расширенной таблице, из которой они будут извлекаться без необходимости привлечения множества объединений и блокировок. Здесь вся хитрость заключается в постоянном корректном заполнении денормализованных данных.

В одном проекте, над которым я работал, несколько аналитиков вводили и обрабатывали данные, которые публиковались ежеквартально. Опубликованная база данных оставалась статичной на протяжении всего квартала и использовалась исключительно для поисковых задач. Это отличный пример проекта, который должен одновременно отвечать требованиям OLTP и OLAP. В этом проекте для увеличения производительности поиска и отчетности была создана отдельная денормализованная база данных. В данном случае запускалась многочасовая процедура, которая денормализовала данные и заполняла базу данных OLAP. Как говорится, и волки сыты, и овцы целы.



Индексированные представления в своей основе представляют собой денормализованные кластеризованные индексы. Тему индексированных представлений мы рассмотрим в главах 42 и 43. Там же вы найдете советы относительно создания денормализованных баз данных отчетности, а также хранилищ данных.

Создание базы данных

База данных является физическим контейнером для всех схем данных, а также для серверных программ. База данных SQL Server представляет собой одну логическую единицу, даже несмотря на то, что физически может размещаться в нескольких файлах.

Создание баз данных является одним из тех вопросов, в которых SQL Server требует совсем немного административной работы. В то же время вы можете принять решение довести файлы базы данных до идеала, применив более сложную технологию.

Команда DDL `CREATE`

Операция создания базы данных с использованием параметров, заданных по умолчанию, крайне проста. Например, следующая команда DDL взята из учебной базы данных Cape Hatteras Adventures:

```
CREATE DATABASE CHA2
```

Инструкция `CREATE` создает файл данных с заданным именем и расширением `.mdf`, а также файл журнала транзакций с расширением `.ldf`.

Естественно, команда `CREATE` поддерживает множество параметров. По умолчанию приняты следующие.

- **Порядок символов по умолчанию** — принятый на сервере.
- **Исходный размер**. С помощью инструкции создается файл объемом порядка 2 Мбайт; с помощью Object Explorer создается файл объемом 3 Мбайт. Файл журнала транзакций

при использовании кода имеет исходный объем 0,5 Мбайт; при создании в Object Explorer — 1 Мбайт.

- **Размещение**. Файлы данных и журнала транзакций по умолчанию размещаются в каталоге, принятом в SQL Server по умолчанию.

Несмотря на то что эти значения, принятые по умолчанию, могут оказаться приемлемыми для учебных и тестовых баз данных, они совершенно не адекватны для реальных промышленных баз. Вам будут предложены более практичные альтернативы.

Создание баз данных в окне Object Explorer требует только ввода ее имени в форме, показанной на рис. 17.1. Данное окно открывается после щелчка правой кнопкой мыши на узле `Database` и выбора в контекстном меню пункта New Database.

Дополнительная информация

Место размещения файлов данных и журнала транзакций определяется в окне параметров сервера. Более подробно этот вопрос мы рассмотрим в главе 34.

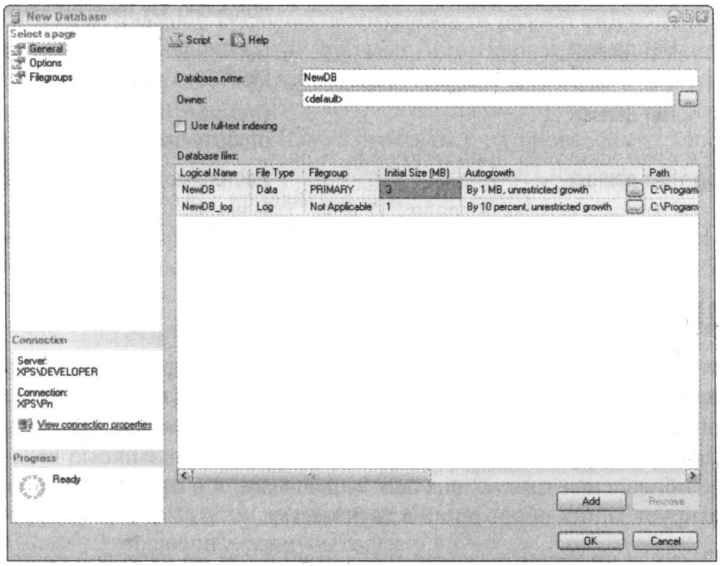


Рис. 17.1. Проще всего создать новую базу данных с помощью ввода имени на странице свойств базы данных

Окно Database Properties состоит их нескольких страниц: General, Options и Filegroups. Для существующих баз данных в это окно добавляются страницы Files, Permissions, Extended Properties, Mirroring и Log Shipping (на рис. 17.1 они не показаны). Все страницы окна Database Properties описаны в табл. 17.1.

Таблица 17.1. Страницы окна свойств базы данных

Страница	Новая база данных	Существующая база данных
General	Создание новой базы данных, настройка имени, владельца, упорядочения, модели восстановления, полнотекстовой индексации и свойств файла данных	Просмотр (только чтение) общих свойств: имени, даты последнего резервирования, размера и упорядочения

Окончание табл. 17.1

Страница	Новая база данных	Существующая база данных
Files	Нет данных	Просмотр и изменение владельца базы данных, упорядочения, модели восстановления, полнотекстовой индексации и файлов базы данных
Filegroups	Просмотр и изменение информации о файловых группах	Просмотр информации о файловых группах
Options	Просмотр и модификация параметров базы данных, таких как автосжатие, настройки ANSI, метод верификации страниц и монопольного доступа	Просмотр и модификация параметров базы данных, таких как автосжатие, настройки ANSI, метод верификации страниц и монопольного доступа
Permissions	Нет данных	Просмотр и модификация серверных ролей, пользователей и разрешений. Более подробно эту тему мы затронем в главе 40
Extended Properties	Нет данных	Просмотр и модификация расширенных свойств
Mirroring	Нет данных	Просмотр и конфигурирование зеркального отображения базы данных (см. главу 52)
Transaction Log Shipping	Нет данных	Настройка доставки журнала транзакций

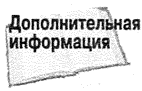
Концепции файлов базы данных

База данных состоит из двух файлов (точнее, из двух наборов файлов): файла данных и журнала транзакций. Файл данных содержит системные и пользовательские таблицы, индексы, представления, хранимые процедуры, пользовательские функции, триггеры и разрешения системы безопасности. Журнал транзакций с последовательной записью является "центром" SQL Server. Все обновления данных вначале записываются в журнал транзакций и проверяются, что гарантирует запись обновлений в двух местах.



Никогда не храните журнал транзакций в той же дисковой системе, что и файл данных. Из соображений целостности транзакций и возможности восстановления базы данных необходимо свести вероятность одновременного повреждения файла данных и журнала транзакций к минимуму.

Журнал транзакций содержит не только пользовательские записи, но и индексные записи, разбиения страниц, реорганизации таблиц и многое другое. После одного интенсивного теста обновлений я проверил журнал с помощью программы Log Explorer компании Lumigent и был немало удивлен, когда обнаружил, что объем записей о деятельности системы занимает в нем около 80%. Так как файл транзакций содержит не только текущую информацию, но и все обновления файла данных, он, как правило, все время разрастается.



Администрирование журнала транзакций включает в себя операции резервирования как части общего плана восстановления (см. главу 36). То, как сервер использует журнал в транзакциях, мы обсудим в главе 51.

Автоматизация роста размера файла

До выхода версии SQL Server 7 приходилось вручную управлять объемом файлов данных, чтобы иметь возможность добавления данных. К счастью, в новых версиях SQL Server этот процесс автоматизирован; при этом вы можете установить следующие параметры.

- **Enable Autogrowth**. По мере того как база наполняется данными, размер файла должен увеличиваться. Если автоматическое увеличение размера не включено, администратор должен изменять размеры файла вручную. Если же этот параметр установлен, то эту задачу берет на себя SQL Server, учитывая следующие параметры.
  - **File growth in megabytes**. Когда файл данных требует расширения, он автоматически увеличивается на заданное количество мегабайт. Увеличивать размер на фиксированное число мегабайт имеет смысл в крупных базах данных. Если увеличение объема файла предсказуемо, то лучший способ — установить этот параметр в значение, приблизительно равное еженедельному увеличению объема данных.
  - **File grouth by percent**. В случае необходимости размер файла данных автоматически увеличивается на заданный процент. Этот вариант лучше использовать в небольших базах данных. В крупных базах данных установка этого параметра может привести к слишком большому увеличению занимаемого файлом места на диске за одну операцию, что может сильно отразиться на производительности. Например, добавление 10% размера к файлу объемом в 5 Гбайт приведет к записи 500 Мбайт информации, что может продлиться довольно долго.
- **Maximum file size**. Установка этого параметра не позволит переполниться дисковой подсистеме; в противном случае переполнение могло бы привести к сбою операционной системы.

На рис. 17.2 показано окно конфигурирования файла данных.

Максимально возможный объем файла данных составляет 16 Тбайт, в то же время объем файла журнала транзакций ограничен 2 Тбайт. Однако это нисколько не ограничивает размер базы данных, так как она может состоять из нескольких файлов.

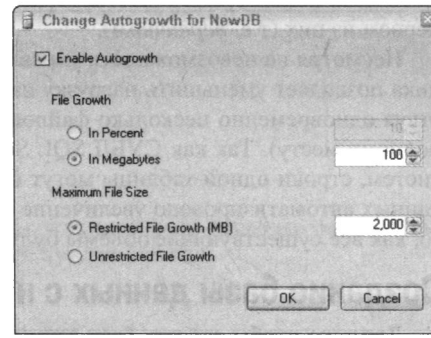


Рис. 17.2. В этом окне утилиты Management Studio сконфигурировано увеличение файла данных на 100 Мбайт и максимальный его объем в 2 Гбайт

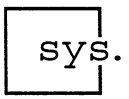
Установить автоматическое увеличение объема файла можно и программным путем, в инструкции `CREATE DETABASE`. Этот объем может быть установлен в килобайтах (Кбайт), мегабайтах (Мбайт), гигабайтах (Гбайт) и терабайтах (Тбайт). По умолчанию приняты мегабайты. Как и в случае использования утилиты Management Studio, этот объем может быть определен в абсолютных и процентных единицах. В следующем примере создается база данных `NewDB` с изначальным объемом файла данных в 10 Мбайт, максимальным объемом в 2 Гбайт и приращением в 10 Мбайт. Файл журнала транзакций имеет исходный размер 5 Мбайт, максимальный — 1 Гбайт и шаг приращения — 10%.

```
CREATE DATABASE NewDB
ON
PRIMARY
  (NAME = NewDB,
    FILENAME = 'c:\SQLData\NewDB.mdf',
```

```
      SIZE = 10MB,
      MAXSIZE = 2Gb,
      FILEGROWTH = 20)
LOG ON
  (NAME = NewDBLog,
     FILENAME = 'd:\SQLLog\NewDBLog.ldf',
      SIZE = 5MB,
      MAXSIZE = 1Gb,
      FILEGROWTH = 10%);
```

Если автоматическое увеличение объема не включено, то потребуется вручную корректировать объем файла при необходимости добавления в них данных. В этом случае можно также воспользоваться диалоговым окном параметров базы данных утилиты Management Studio и инструкцией `ALTER DATABASE` с параметром `MODIFY FILE`. Следующая инструкция изменяет объем файла данных базы `NewDB` до 25 Мбайт:

```
ALTER DATABASE NewDB
  MODIFY FILE
    (Name = NewDB,
    SIZE = 25MB,
    MAXSIZE = 2Gb,
    FILEGROWTH = 0);
```



Чтобы перечислить базы данных программным путем, воспользуйтесь представлением каталога `sysdatabases`.

Использование множества файлов

Как данные, так и журнал транзакций могут храниться во множестве файлов. Это позволит увеличить быстродействие и открыть новые перспективы для увеличения объема файлов. Любой дополнительный (или *вторичный*) файл данных имеет по умолчанию расширение `.ndf`. Если база использует множество файлов данных, то системные таблицы будут храниться в первом из них (т.е. *первичном*).

Несмотря на невозможность управления местом размещения таблиц и индексов, эта методика позволяет уменьшить нагрузку на каждую из используемых дисковых подсистем, задействуя одновременно несколько файлов (пропорционально имеющемуся у них в наличии свободному месту). Так как СУБД SQL Server сама управляет рабочей загрузкой дисковых подсистем, строки одной таблицы могут быть распределены по нескольким файлам. Если в базе данных автоматизировано увеличение объема файлов, он будет выполняться только после того, как все существующие объемы будут заполнены.

Создание базы данных с несколькими файлами

Для того чтобы создать базу данных с несколькими файлами в Management Studio, добавляйте имена файлов в таблицу Database files на странице Files диалогового окна параметров базы данных (рис. 17.3).

При программном создании базы данных с несколькими файлами добавьте их место размещения в инструкцию `CREATE DATABASE`.

```
CREATE DATABASE NewDB
ON
PRIMARY
  (NAME = NewDB,
    FILENAME = 'e:\SQLData\NewDB.mdf'),
```

```
  (NAME = NewDB2,
    FILENAME = 'f:\SQLData\NewDB2.ndf')
LOG ON
  (NAME = NewDBLog,
    FILENAME = 'g:\SQLLog\NewDBLog.ldf'),
  (NAME = NewDBLog2,
    FILENAME = 'h:\SQLLog\NewDBLog2.ldf');
```

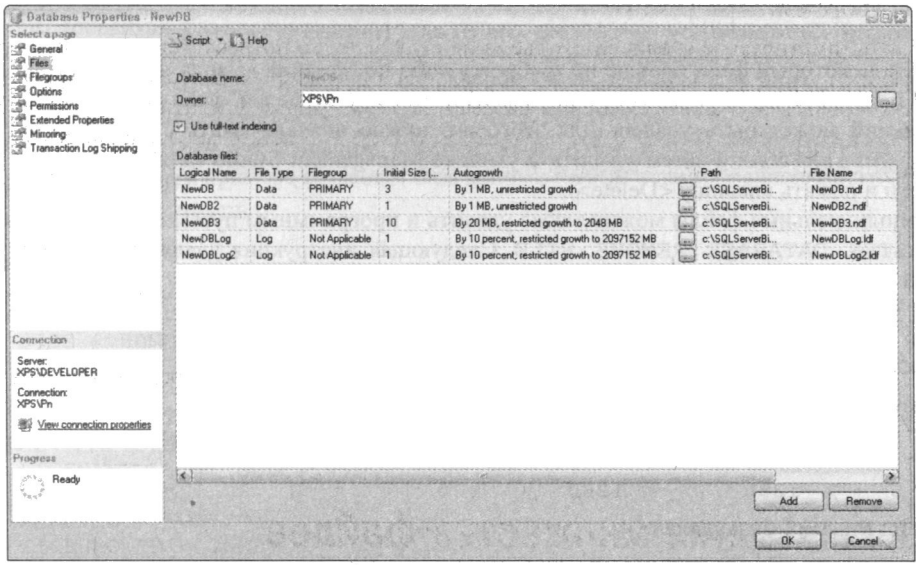
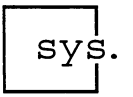


Рис. 17.3. Создание базы данных с несколькими файлами в Management Studio

Результат будет следующим:

```
The CREATE DATABASE process is allocating
  0.63 MB on disk 'NewDB'.
The CREATE DATABASE process is allocating
  1.00 MB on disk 'NewDB2'.
The CREATE DATABASE process is allocating
  1.00 MB on disk 'NewDBLog'.
The CREATE DATABASE process is allocating
  1.00 MB on disk 'NewDBLog2'.
```



Для перечисления файлов, используемых текущей базой данных, можете выполнить запрос к представлению каталога `sysdatabases`.

Изменение файлов текущей базы данных

Количество файлов, используемых текущей базой данных, можно легко изменить. Если данные заполнили устройство, вы можете добавить новый файл в таблице Database Files (рис. 17.3) так же, как и при создании базы данных.

В программном коде можно выполнить ту же операцию с помощью инструкции `ALTER DATABASE` с параметром `ADD FILE`. Синтаксис этой инструкции аналогичен использовавшейся при создании базы данных. В следующем примере к базе данных `NewDB` добавляется третий файл:

```
ALTER DATABASE NewDB
  ADD FILE
   (NAME = NewDB3,
    FILENAME = 'i:\SQLData\NewDB3.ndf',
    SIZE = 10MB,
    MAXSIZE = 2Gb,
    FILEGROWTH = 20);
```

Результат выполнения инструкции:

```
Extending database by 10.00 MB on disk 'NewDB3'.
```

Если некоторый файл больше не требуется либо по причине вывода из эксплуатации дисковой подсиситемы, либо из-за ее использования для других целей, файл данных или журнала транзакций может быть удален. Для этого его нужно вначале сжать с помощью функции `DBCC ShrinkFile`, а затем удалить в утилите Management Studio (для этого нужно выделить его и нажать клавишу <Delete>).

Дополнительные файлы можно также удалить и программным путем с помощью инструкции `ALTER DATABASE REMOVE FILE`. Следующая инструкция удаляет созданный ранее третий файл базы данных `NewDB`:

```
DBCC SHRINKFILE (NewDB3, EMPTYFILE)
ALTER DATABASE NewDB
  REMOVE FILE NewDB3;
```

Результат выполнения пакета следующий:

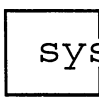
```
DbId FileId CurrentSize MinimumSize UsedPages EstimatedPages
---- ------ ----------- ----------- --------- --------------
12   5      1280        1280        0         0
```

Файл `NewDB3` был удален.

Планирование нескольких файловых групп

Файловая группа представляет собой расширенный механизм организации объектов базы данных. По умолчанию база данных имеет всего одну файловую группу — *первичную*. При конфигурировании базы данных для использования нескольких файловых групп конкретные объекты (таблицы, индексы и т.п.) могут создаваться в заданных группах. Эта методика может поддерживать две основные стратегии.

- Использование нескольких файловых групп способно повысить производительность за счет разделения таблиц и индексов на разные дисковые подсистемы.
- Использование нескольких файловых групп может помочь организовать план резервирования и восстановления, помещая статические данные в одну файловую группу и активизированные данные — в другую.



Проще всего определить состав файлов базы данных и их размеры следующим образом: выполнить запрос к представлению каталога `sysdatabase_files`.



При планировании файлов и файловых групп можно выиграть в производительности от помещения каждого файла в отдельную дисковую подсистему. Лично я настоятельно рекомендую распределять задачи ввода-вывода по нескольким дисковым приводам.

Однако перед тем как назначать дополнительные дисковые подсистемы файлам данных, обязательно удовлетворите потребности журнала транзакций и временной базы данных. В идеальном случае лучше иметь пять дисковых подсистем (для Windows, приложений и файла подкачки; для временной базы данных tempdb и для ее журнала; а также для файла данных и журнала транзакций), прежде чем создавать дополнительные файлы и файловые группы.

Создание базы данных с файловыми группами

Чтобы добавить к базе данных файловую группу в Management Studio, откройте страницу параметров базы данных в Object Explorer. На странице Filegroups создайте дополнительные файловые группы. После этого на странице Files создавайте новые файлы и выбирайте для них в комбинированном списке конкретные файловые группы.

Чтобы выполнить ту же операцию программным путем, используйте параметр FILEGROUPS. В следующем примере создается база данных NewDB с двумя файловыми группами:

```
CREATE DATABASE NewDB
ON
PRIMARY
  (NAME = NewDB,
     FILENAME = 'd:\SQLData\NewDB.mdf',
     SIZE = 50MB,
     MAXSIZE = 5Gb,
     FILEGROWTH = 25MB),
FILEGROUP GroupTwo
  (NAME = NewDBGroup2,
     FILENAME = 'e:\SQLData\NewDBTwo.ndf',
     SIZE = 50MB,
     MAXSIZE = 5Gb,
     FILEGROWTH = 25MB)
LOG ON
  (NAME = NewDBLog,
     FILENAME = 'f:\SQLLog\NewDBLog.ndf',
     SIZE = 100MB,
     MAXSIZE = 25Gb,
     FILEGROWTH = 25MB);
```

Модификация файловых групп

Файловые группы можно изменять точно так же, как и файлы. В Management Studio можно создавать новые файловые группы, добавлять и удалять в них файлы, а также удалять пустые группы. Однако удалить файловую группу несколько сложнее, чем сжать файл. Если в файловой группе содержатся какие-либо данные, то сжатие файла приведет к перемещению его данных в другой файл той же группы. Перед удалением файловой группы нужно удалить из нее таблицы и индексы.

С помощью редактора запросов и кода T-SQL можно создавать и удалять файловые группы. Для этого используются инструкции ALTER DATABASE ADD FILEGROUP и ALTER DATABASE REMOVE FILEGROUP. Они имеют практически такой же синтаксис, как инструкции добавления и удаления файлов.

Удаление базы данных

База данных может быть удалена с сервера с помощью выделения ее в Object Explorer и выбора в контекстном меню команды Delete.

Для программного удаления базы данных используйте инструкцию DROP DATABASE:

```
DROP DATABASE NewDB
```

Создание таблиц

Подобно всем реляционным СУБД, SQL Server является таблично ориентированной. После создания базы данных следующим действием является наполнение ее таблицами. База данных SQL Server может содержать до 2147483647 объектов, включая таблицы, так что вы можете создавать столько таблиц, сколько заблагорассудится.

Создание таблиц в Management Studio

Если вам нравится работать в графической среде, то воспользуйтесь утилитой Management Studio, которая предлагает две рабочие области для создания и модификации таблиц.

- Конструктор таблиц (Table Designer) (рис. 17.4) перечисляет столбцы по вертикали, размещая свойства столбцов в нижней части окна.
- Конструктор баз данных (Database Designer) (рис. 17.5) — более гибкий инструмент, чем конструктор таблиц. В нем могут отображаться ограничения внешних ключей как связи с другими таблицами.

Дополнительная информация

О работе с этими инструментами см. в главе 6.

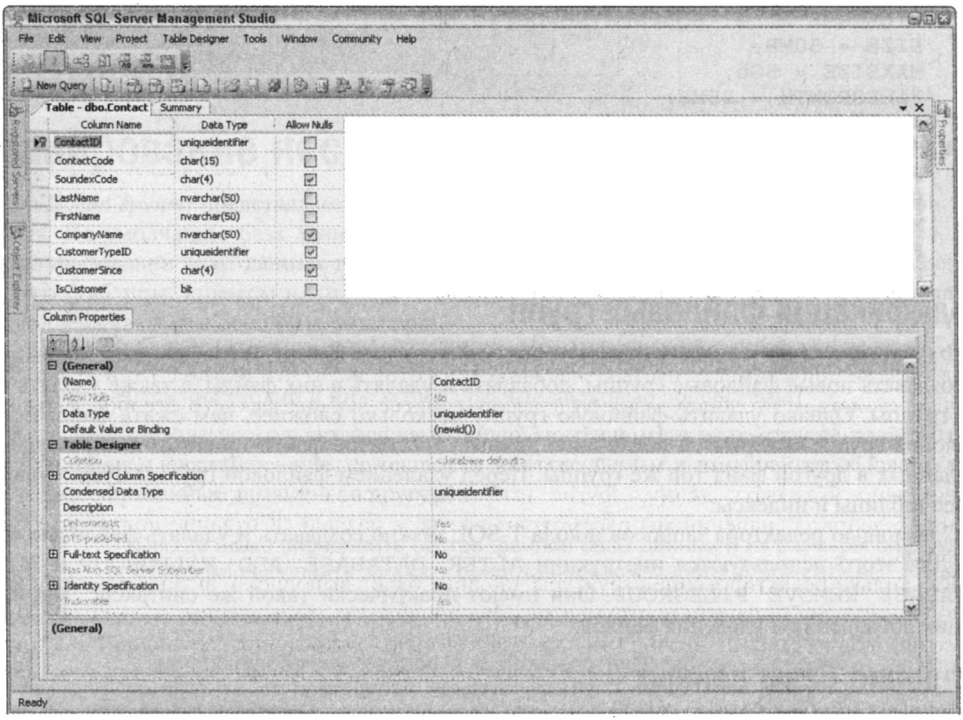


Рис. 17.4. Создание таблицы `Contact` *учебной базы данных OBXKites с помощью конструктора таблиц утилиты Management Studio*

В каждом из этих инструментов графически представлена структура таблицы. Как только проектирование таблицы будет завершено, Management Studio генерирует сценарий, который применяет изменения к базе данных. Если вы модифицируете существующую таблицу, то часто данные изначально сохраняются во временной таблице, затем удаляются некоторые элементы, после чего создается новая таблица, в которую вставляются данные.

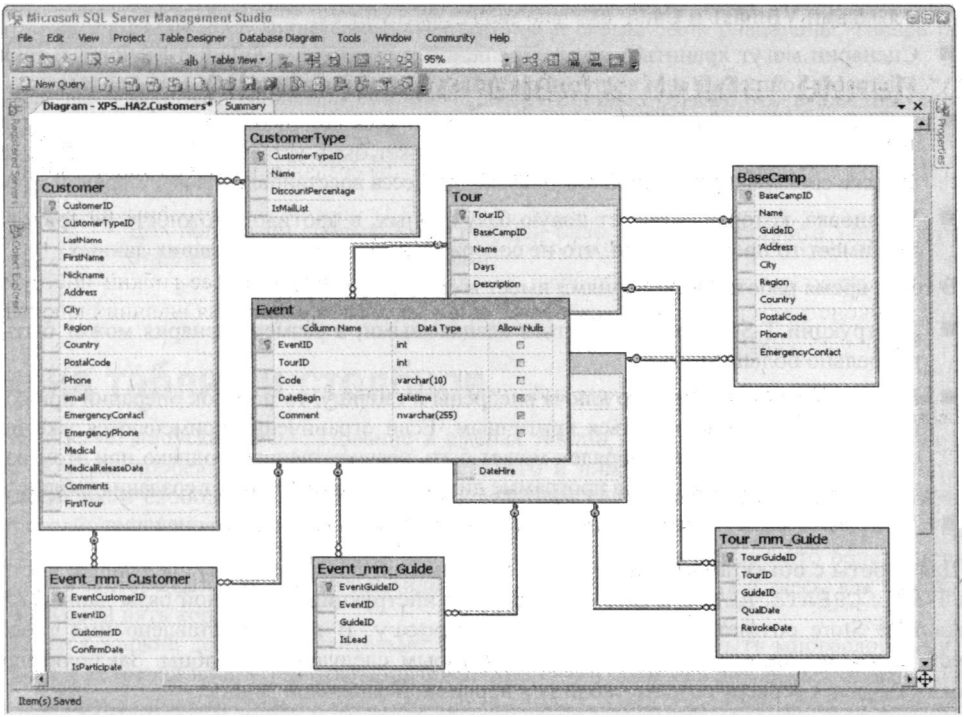
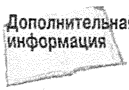


Рис. 17.5. Создание таблицы `Customer` в учебной базе данных CHA2 с помощью конструктора баз данных утилиты Management Studio

Конструктор таблиц отображает только имена столбцов и их тип данных (вместе с длиной), а также допустимость пустых значений. Несмотря на то что именно это и является основным свойством столбцов, меня раздражает процесс выбора каждого столбца только для того, чтобы просмотреть или изменить его остальные свойства.

Каждый из типов данных мы рассмотрим отдельно далее в этой главе. Для одних типов нужно явно указывать длину, другие имеют фиксированную длину. Вопросы пустых (неопределенных) значений мы рассмотрим позже, в разделе "Создание пользовательских столбцов".

Как только редактирование структуры таблицы будет выполнено, активизируется кнопка Save Change Script панели инструментов. Она приводит к отображению кода, который конструктор таблиц должен выполнить для фактического сохранения изменений. К тому же кнопка Save Change Script позволяет сохранить сценарий в файле `.sql`, чтобы его можно было повторить на другом сервере.



Подробно об использовании конструкторов таблиц и баз данных см. в главе 6.

Работа со сценариями SQL

Если вы создаете базу данных для массового тиражирования, то преимущество реализации схемы в сценариях становится очевидным.

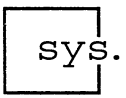
- Код сценария находится в одном месте. Работа со сценариями SQL аналогична работе с языками VB.NET и C#.
- Сценарии могут храниться в узлах решений и проектов в Solution Explorer, а также в Microsoft SourceSafe или в другой системе управления изменениями.
- Если базовый сценарий базы данных содержит весь код, необходимый для ее создания, то самая современная версия базы данных может быть установлена без необходимости запуска сценариев изменений или запуска процесса восстановления из резервной копии.
- Установка, которая освежает новую базу данных, в противоположность резервной копии имеет то преимущество, что не содержит каких-либо устаревших данных.

В то же время работа со сценариями имеет и свои недостатки.

- Инструкции T-SQL могут оказаться незнакомыми, а размер сценария может быть запредельно большим.
- Если ограничения внешнего ключа внедрены в таблицу, то порядок операций при создании таблицы может оказаться критичным. Если ограничения применяются уже после создания таблиц, то этот порядок может быть несущественным, однако при этом создание внешних ключей будет в программе дистанционно отдалено от создания таблиц.
- Диаграммы базы данных Management Studio не являются частью сценария.

Для работы с объектами (в том числе и с таблицами) предназначены следующие инструкции T-SQL: `CREATE`. `ALTER` и `DROP`. Следующая инструкция из учебной базы данных Outer Banks Kite Store создает таблицу `ProductCategory`. В ней представлено имя таблицы (вместе с именем ее владельца — `dbo`), за которым следуют ее столбцы. Заключительная строка указывает серверу создать таблицу в первичной файловой группе:

```
CREATE TABLE dbo.ProductCategory (
  ProductCategoryID UNIQUEIDENTIFIER NOT NULL
    ROWGUIDCOL DEFAULT (NEWID()) PRIMARY KEY NONCLUSTERED,
  ProductCategoryName NVARCHAR(50) NOT NULL,
  ProductCategoryDescription NVARCHAR(100) NULL
  )
  ON [Primary];
```



Для получения списка таблиц текущей базы данных выполните запрос к представлению каталога `sysobject`, задав условие фильтра `type_desc='USER_TABLE'`.



С расширенными примерами создания баз данных и таблиц с помощью сценариев можно ознакомиться на сайте книги по адресу `www.SQLServerBible.com`.

Схемы

Схема представляет собой объект, который служит исключительно для того, чтобы владеть другими объектами базы данных. Она сегментирует крупные базы данных на управляемые модули, а также может реализовывать сегментированную стратегию защиты данных.

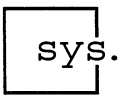
Обычно (и по умолчанию) объекты принадлежат схеме dbo. Имя схемы является третьей частью четырехкомпонентного имени:

```
Сервер.база_данных.схема.объект
```



В предыдущих версиях SQL Server объекты принадлежали пользователям. Также объекты могли принадлежать объектам схемы, что в принципе то же самое, что принадлежность пользователям, однако называется по-другому. В SQL Server 2005 концепции пользователей и схемы были разделены. Теперь пользователи больше не могут владеть объектами.

Если вы используете пользовательские схемы, отличные от dbo, то в каждом запросе должны указывать их в явном виде. Хотя следует заметить, что двухкомпонентное имя повышает производительность, к тому же вряд ли кому приносит особое удовольствие постоянный ввод имени схемы.



Для получения списков схем текущей базы данных выполните запрос к представлению каталога sysschemas.

Имена таблиц и столбцов

SQL Server допускает использование в именах таблиц и столбцов до 128 символов Unicode, включая пробелы, а также символы верхнего и нижнего регистров. Естественно, все прелести такой свободы перекрываются впоследствии необходимостью ввода бесконечных имен столбцов и заключения их в квадратные скобки при наличии пробелов. Еще более опасно обсуждать соглашения об именах с программистами — это все равно что болеть за свою команду в секторе фанатов другой команды. В то же время разрешите и здесь, как говорится, вставить свои пять копеек.

Ведутся жаркие дебаты относительно того, должны ли имена быть многословными. На форумах я встречал на эту тему обоснованные претензии, вызывавших целый шквал словесных атак других пользователей.

Лагерь сторонников многословных имен верит, что таблица содержит набор строк и потому должна именоваться описательным именем во множественном числе. Обоснование этой позиции звучит следующим образом: "Таблица заказчиков представляет собой их множество. Множества состоят из массы элементов, поэтому и имя таблицы должно отражать этот факт. Если бы таблица содержала данные всего об одном заказчике, то в ней бы просто не было смысла".

В то же время в моем окружении примерно три четверти программистов используют в названиях таблиц существительное в единственном числе. С их точки зрения, название Customer для таблицы заказчиков звучит лучше, чем Customers.

Думаю, большинство читателей согласятся со мной, что последовательность в именовании гораздо важнее соглашения об именах.



Последовательность для программистов — священна. Целью соглашений об именах, ограничениях, о ссылочной целостности, реляционной структуре и даже о типах данных является привнесение порядка в данные, которые моделируют реальность. Если при проектировании базы данных вы задаете себе вопрос, какой из вариантов более последовательный, значит, вы на правильном пути к решению задачи.

Лично мне кажется, что программисты используют соглашения об именах, чтобы оградить себя от хаотичных конструкций, с которыми приходилось работать в прошлом. Достаточно проработав с "плоскими" файлами и их именами во множественном числе, я склоняюсь к использованию единственного числа.

Если база данных достаточно велика и таблицы в ней логически сгруппированы, я добавляю к их именам двух- или трехбуквенную аббревиатуру, облегчающую работу с ней. Мне приходилось встречать и системы нумерации модулей и таблиц, и я не рекомендую их использовать. К примеру, для таблицы складских запасов отлично подходит имя `InvItem`; в то же время имя `0207_Item` похоже на шифровку.

Еще один вопрос связан с использованием символов подчеркивания при формировании имен. Например, некоторые магазины компьютерной техники для таблицы строк заказов используют имя `ORDER_DETAIL`. Я избегаю использования символов подчеркивания, используя их только для связывающих таблиц, разрешающих отношения "многие ко многим". Исследования показали, что комбинированные слова типа `OrderDetail` легче читать, чем комбинированные слова, набранные исключительно в верхнем или нижнем регистре.

Ниже приведены соглашения, которые я использую при проектировании баз данных.

- В именах таблиц используется единственное число без чисел, но с префиксом.
- В связывающих таблицах, разрешающих отношения "многие ко многим", используются имена типа *таблица*_mm_*таблица*.
- В именах для разделения слов используется смешанный регистр символов без подчеркиваний и пробелов (например, `MixedCase`).
- Для первичных ключей используется имя таблицы, дополненное символами `ID`. Например, для первичного ключа таблицы `Customer` уместно использовать имя `CustomerID`.

  Внешние ключи имеют те же имена, что и первичные, с которыми они связаны, за исключением рекурсивных отношений. Например, в учебной базе данных Family внешний ключ `MotherID` ссылается на первичный ключ `PersonID`, а в базе данных списка материалов на один первичный ключ рекурсивно ссылается несколько внешних (`BillofMaterials.MaterialID` и `BillofMaterials.SourceMaterialID` ссылаются на первичный ключ `Material.MaterialID`).
- Избегайте неуместных аббревиатур.
- Будьте последовательны при именовании объектов всех баз данных. Например, для полей имени и фамилии всегда используйте названия `FirstName` и `LastName`.

Файловые группы

В отличие от столбцов, единственной информацией, которую следует указать при создании таблиц, является их имя. В то же время таблицы можно создавать в конкретных файловых группах, если в базе данных существует несколько таких групп.

В организационных целях база данных OBXKites использует две файловые группы. Все данные, которые изменяются на регулярной основе, вынесены в файловую группу `primary`. Эта файловая группа часто резервируется. Данные, которые изменяются достаточно редко (например, классификатор приоритетов заказов), вынесены в файловую группу `static`:

```
CREATE TABLE OrderPriority (
  OrderPriorityID UNIQUEIDENTIFIER NOT NULL
    ROWGUIDCOL DEFAULT (NEWID()) PRIMARY KEY NONCLUSTERED,
  OrderPriorityName NVARCHAR (15) NOT NULL,
  OrderPriorityCode NVARCHAR (15) NOT NULL,
  Priority INT NOT NULL
  )
  ON [Static];
```

Создание первичных ключей

Первичные и внешние ключи являются теми нитями, которые связывают отдельные таблицы в единую реляционную базу данных. Я трактую эти столбцы как доменные, в отличие от пользовательских, которые хранят данные. Проектирование этих ключей имеет критическое влияние на производительность и полезность физической базы данных.

Схема базы данных должна быть трансформирована из теоретической логической структуры в физическую, и в основе последней часто лежат первичные и внешние ключи. После того как база данных будет запущена в производственных условиях, ключи уже сложно изменить. Правильный выбор первичных ключей на этапе проектирования — это то, за что стоит бороться.

Первичные ключи

Реляционная база данных зависит от первичных ключей. Это краеугольный камень физической схемы базы данных. В среде разработчиков не утихают споры относительно использования естественных (понятных пользователям) и суррогатных (автоматически генерируемых) первичных ключей.

На физическом уровне первичные ключи служат двум целям:

- для уникальной идентификации строк;
- как объект ссылки для внешних ключей.

В SQL Server первичные и внешние ключи реализованы как ограничения. Задачей данных ограничений является гарантирование удовлетворения новыми данными определенных критериев или блокирование операций модификации данных.

Ограничение первичного ключа является комбинацией требований уникальности (что подразумевает и запрет пустых значений) и использования уникального индекса (кластеризованного или нет).

Создание первичных ключей

В программе столбец может быть назначен первичным ключом двумя способами.

- С помощью объявления ограничения первичного ключа в инструкции `CREATE TABLE`. Следующий пример из учебной базы данных Cape Hatteras Adventures использует этот способ для создания таблицы `Guide` и назначения поля `GuideID` первичным ключом с некластеризованным индексом:

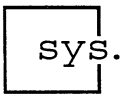
```
CREATE TABLE dbo.Guide (
  GuideID INT IDENTITY NOT NULL PRIMARY KEY NONCLUSTERED,
  LastName VARCHAR(50) NOT NULL,
  FirstName VARCHAR(50) NOT NULL,
  Qualifications VARCHAR(2048) NULL,
  DateOfBirth DATETIME NULL,
  DateHire DATETIME NULL
  )
  ON [Primary];
```

- С помощью объявления первичного ключа после создания таблицы. Для этого используется инструкция `ALTER TABLE`. Предполагая, что первичный ключ пока не назначен для таблицы `Guide`, следующая инструкция позволит это сделать:

```
ALTER TABLE dbo.Guide ADD CONSTRAINT
PK_Guide PRIMARY KEY NONCLUSTERED(GuideID)
ON [PRIMARY];
```



Метод индексации первичного ключа (кластеризованный и некластеризованный) является одним из самых важных соглашений проектирования физической схемы базы данных. В главе 50 мы углубимся в детали индексных страниц; там же вы узнаете и об основных стратегиях индексации первичных ключей.



Для того чтобы получить список первичных ключей текущей базы данных, выполните запрос к представлениям каталогов `sysobjects` и `syskey_constraints`.

Естественные первичные ключи

Естественные ключи отражают то, как объект идентифицируется в реальном мире. Фамилии людей, номера автомобилей и паспортов, а также адреса являются примерами естественных ключей.

Естественные ключи имеют свои достоинства и недостатки.

- Естественные ключи легко распознаются людьми. По ним пользователи могут легко найти нужные данные. Недостатком является то, что люди обычно хотят придать первичному ключу некоторый смысл и вносят в него определенные символы.

  Люди имеют склонность изменять то, что сами создают, а модификация значений первичного ключа может вызвать проблемы. Если вы решили использовать естественный первичный ключ, то обязательно каскадируйте обновления всех внешних ключей, ссылающихся на него. Только так вы сможете гарантировать ссылочную целостность базы данных при внесении изменений в первичный ключ.

- Естественные ключи размножают значения первичного ключа во все поколения внешних ключей. К тому же составные внешние ключи имеют склонность разрастаться, если включают в себя значения всех столбцов первичных ключей.

  Преимуществом является то, что можно создать объединение самой нижней в иерархии вторичной таблицы непосредственно с первичной, исключив из цепочки все промежуточные. Недостатком является громоздкость таких ключей, при этом большинство объединений должно использовать несколько столбцов.

- Естественные ключи обычно не имеют какой-либо определенной организации. Это отрицательно сказывается на производительности при вставке информации в середину отсортированных данных, что вызвано необходимостью разбиения страниц.

Использование столбцов идентичности в качестве суррогатных первичных ключей

Суррогатные ключи заполняются компьютером и, как правило, не имеют для людей никакого смыслового значения. В SQL Server суррогатные ключи представляют собой столбцы идентичности или глобальные уникальные идентификаторы.

До сих пор самым распространенным методом создания первичных ключей является использование *столбца идентичности*. Подобно столбцу автонумерации или последовательности в других базах данных, столбец идентичности при вставке в таблицу новых строк генерирует последовательность целых чисел. При желании вы можете назначить для этой последовательности изначальное число и интервал.

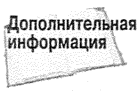
- Столбцы идентичности имеют два основных достоинства. Целые числа легче распознать на глаз и отредактировать, чем глобальные уникальные идентификаторы (GUID).
- Целые числа невелики и быстро обрабатываются. Мое неформальное тестирование показало, что использование целых чисел на 10% повышает производительность по

сравнению с GUID. В других публикациях я встречал значения ускорения от 10 до 33%. Однако такой скачок производительности достигается только при обработке тысяч строк. Если инструкция SELECT отбирает из большой таблицы за одну операцию только несколько строк, то вы вряд ли заметите повышение производительности.

Отрицательные стороны использования столбцов идентичности приведены ниже.

- Их уникальность распространяется только на одну таблицу — в разных таблицах будут использованы одни и те же значения. Мне встречались программы, в которых были некорректно объединены таблицы, в результате чего возвращался результирующий набор данных, содержащий несоответствующие данные двух таблиц.
- Модели со столбцами идентичности склонны использовать во всех таблицах суррогатные ключи. Это выливается в использование составных первичных ключей, созданных множеством внешних. Несмотря на то что при этом создаются маленькие и быстрые первичные ключи, для перемещения по схеме данных требуется большее число объединений.

Значения столбцов идентичности генерируются ядром базы данных при вставке строки. Попытка вручную вставить значение в столбец идентичности приведет к ошибке, если, конечно, для параметра SET INSERT_IDENTITY не установлено значение True.



Подробно о проблемах модификации данных в таблицах со столбцами идентичности см. в главе 16.

В следующем примере, взятом из базы данных Cape Hatteras Adventures, создается таблица, которая использует столбец идентичности для своего первичного ключа (листинг немного сокращен):

```
CREATE TABLE dbo.Event (
  EventID INT IDENTITY NOT NULL PRIMARY KEY NONCLUSTERED,
  TourID INT NOT NULL FOREIGN KEY REFERENCES dbo.Tour,
  EventCode VARCHAR(10) NOT NULL,
  DateBegin DATETIME NULL,
  Comment NVARCHAR(255)
  )
  ON [Primary];
```

Установка столбца (или столбцов) в качестве первичного ключа в Management Studio не сложнее выделения этого столбца и щелчка на соответствующей кнопке панели инструментов. Для создания составного первичного ключа выберите все задействованные в нем столбцы и щелкните на кнопке панели инструментов.



Чтобы ознакомиться с обоими методами создания суррогатных ключей, обратитесь к учебным базам данных Family, Cape Hatteras Adventures и Material Specification. Вышеназванные базы используют столбцы идентичности, а база данных Outer Banks Kite Store использует глобальные уникальные идентификаторы. Все фрагменты программ, приведенные в настоящей главе, можно загрузить с сайта книги по адресу www.SQLServerBible.com.

Использование уникальных идентификаторов в качестве суррогатных первичных ключей

Тип данных uniqueidentifier в SQL Server является двойником глобального уникального идентификатора (GUID), используемого в среде .NET. Это 16-байтовое шестнадцатеричное число, которое уникально для каждой таблицы, базы данных и сервера на всей планете. Хотя в пределах одного столбца уникальны и значения столбца идентичности, и значения

GUID, у последнего область уникальности гораздо шире, чем у первого. Нарушая правила стилистики, можно сказать, что GUID более уникален, чем столбцы идентичности.

Идентификаторы GUID имеют ряд существенных достоинств.

- База данных, использующая GUID в качестве первичных ключей, может быть легко реплицирована без каких-либо осложнений. Репликация добавит уникальные идентификаторы во все таблицы без столбца с типом `uniqueidentifier`. Несмотря на то что это сделает столбцы глобально уникальными для задач репликации, код приложения будет продолжать идентифицировать строки только по целочисленным первичным ключам. Таким образом, слияние реплицированных строк в других серверах приведет к ошибке. так как будут встречаться дублирующиеся значения первичного ключа.
- Использование GUID отстраняет пользователя от смыслового значения первичного ключа.
- Использование GUID поможет избежать ошибок, вызванных некорректным объединением таблиц, но, независимо от этого, данные будут возвращаться. Причина этого лежит в том, что даже несоответствующие строки в ключевых столбцах будут использовать целочисленные значения.
- Число глобальных уникальных идентификаторов бесконечно. Таблицы, созданные на основе столбца идентичности, могут содержать только 2147483648 строк. Естественно, тип данных можно установить в `bigint` или `numeric`, однако это лишит вас всех прелестей столбца идентичности.
- Поскольку идентификаторы GUID могут генерироваться либо как значения столбца по умолчанию, либо в выражениях инструкции `SELECT`, их программирование существенно упрощается по сравнению с программированием столбцов идентичности. Использование GUID позволит обойти стороной все проблемы, возникающие при использовании столбца идентичности.

Недостатки использования идентификаторов GUID в основном связаны с производительностью.

- Идентификаторы GUID существенно крупнее по сравнению с целыми числами, поэтому на странице их помещается меньше. В результате для извлечения одного и того же количества строк придется считывать большее количество страниц.
- Уникальные идентификаторы, генерируемые функцией `NewID()`, подобно естественным ключам, по своей сути случайны. Таким образом, операции вставки данных будут вызывать разбиения страниц, что негативно скажется на общей производительности. В то же время естественные ключи имеют естественное распределение, поэтому проблема разбиения страниц в естественных ключах встречается реже.

Таблица `Product` в базе данных Outer Banks Kite Store использует в качестве первичного ключа глобальный уникальный идентификатор. В следующем примере тип данных столбца `ProductID` устанавливается как `uniqueidentifier`. В нем запрещаются пустые значения, а свойство `rowguidcol` устанавливается в значение `True`, что позволяет использовать этот столбец в репликациях. В качестве значения по умолчанию используется новый генерируемый идентификатор. Это поле используется в качестве первичного ключа, и для него создается некластеризованный индекс:

```
CREATE TABLE dbo.Product (
  ProductID UNIQUEIDENTIFIER NOT NULL
    ROWGUIDCOL DEFAULT (NEWSEQUNTIALID())
    PRIMARY KEY CLUSTERED,
  ProductCategoryID UNIQUEIDENTIFIER NOT NULL
    FOREIGN KEY REFERENCES dbo.ProductCategory,
```

```
ProductCode CHAR(15) NOT NULL,
ProductName NVARCHAR(50) NOT NULL,
ProductDescription NVARCHAR(100) NULL,
ActiveDate DATETIME NOT NULL DEFAULT GETDATE(),
DiscountinueDate DATETIME NULL
)
ON [Static];
```

Существуют два основных метода генерирования уникальных идентификаторов (на самом деле все они генерируются операционной системой Windows) и множество мест, где эта генерация может выполняться.

- Функция `NewID()` генерирует уникальный идентификатор, используя несколько факторов, в том числе код NIC, адрес MAC, внутренний идентификатор процессора и текущий момент в часах компьютера. Последние шесть байтов занимает номер узла сетевой карты.

  Универсальная функция `NewID()` может использоваться в качестве значения по умолчанию для столбца, передаваться в инструкцию `INSERT`, а также выполняться как функция в любом выражении.

- Функция `NewsequentionalID()` аналогична функции `NewID()`, однако в отличие от последнего гарантирует, что каждое следующее значение уникального идентификатора больше всех остальных в таблице.

  Функция `NewsequentionalID()` может использоваться только в качестве значения по умолчанию для столбца. И это имеет определенный смысл, так как генерируемое значение зависит от наибольшего идентификатора в конкретной таблице.



Функция `NewsequentionalID()` появилась в версии SQL Server 2005. Теперь уникальные идентификаторы могут быть кластеризованы без угрозы возникновения проблемы разбиения страниц.



По моему мнению, в свете практических выгод и рисков естественных и суррогатных ключей некоторые идентификаторы настолько неизменны, что могут без проблем использоваться в качестве первичных ключей. Однако если возникает хоть малейшая вероятность изменения идентификатора, защищайте данные с помощью суррогатных первичных ключей. Если данные не будут реплицироваться, рекомендую использовать столбцы идентичности; в противном случае наилучшим решением будет использование идентификаторов GUID.

На конференциях, посвященных SQL Server, я веду дискуссионные группы "Решения моделирования ключей данных: за и против". посвященные рассмотренным в настоящем разделе вопросам. Предлагаю вам принять участие в этих диспутах, чтобы обсудить все возможные варианты решения задачи.

Создание внешних ключей

Вторичная таблица, которая связана с первичной, использует внешний ключ для указания на первичный ключ первичной таблицы. Требования *ссылочной целостности* подразумевают, что каждый внешний ключ обязательно указывает на первичный. Поддержание ссылочной ценности жизненно важно в реляционных базах данных. База данных должна начинать и завершать каждую транзакцию в непротиворечивом состоянии. Эта целостность должна расширяться и на ссылки внешних ключей.

Дополнительная информация

О целостности баз данных и принципах ACID вы можете подробно прочитать в главах 1 и 51.

Таблицы в SQL Server могут поддерживать до 253 ограничений внешних ключей. Внешние ключи могут ссылаться на первичные ключи, уникальные ограничения и уникальные индексы любой таблицы, за исключением, естественно, временных.

Бытует распространенное заблуждение, что поддержание ссылочной целостности является уделом исключительно первичных ключей. Поскольку именно внешний ключ должен ссылаться на допустимое значение первичного ключа, то это ограничение является аспектом внешнего, а не первичного ключа.

Декларативная ссылочная целостность

Декларативная ссылочная целостность (DRI) в SQL Server может гарантировать ссылочную целостность без необходимости написания пользователем триггеров или процедур. Она обслуживается самим ядром базы данных и практически не загружает ресурсы компьютера (в отличие от триггера).

В SQL Server декларативная ссылочная целостность поддерживается с помощью ограничений внешних ключей. Вы можете открыть форму Foreign Key Relationships, показанную на рис. 17.6, чтобы создать или изменить ограничения внешних ключей. Для этого выполните следующее.

- В конструкторе баз данных выделите столбец первичного ключа и перетащите его к столбцу внешнего ключа. Это приведет к открытию диалогового окна Foreign Key Relationships.
- В конструкторе таблиц щелкните на кнопке Relationships панели инструментов или выберите в меню команду Table⇨Designer⇨Relationships. В качестве альтернативы можете выбрать в конструкторе баз данных вторую таблицу (с внешним ключом), а затем щелкнуть на кнопке Relationships панели инструментов или выбрать одноименный пункт из контекстного меню этой таблицы.

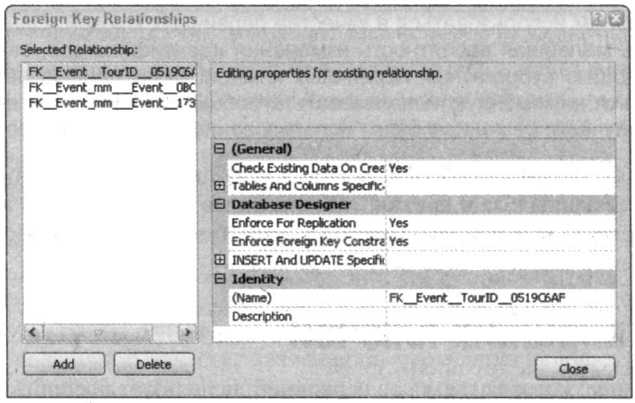


Рис. 17.6. Форма Foreign Key Relationships утилиты Management Studio используется для создания и модификации декларативной ссылочной целостности

В окне Foreign Key Relationships существует несколько параметров, позволяющих определить режим работы внешнего ключа.

- Enforce for Replications (поддерживать репликации).
- Enforce Foreign Key Constraint (поддерживать ограничения внешнего ключа).
- Delete Rule (каскадное удаление мы рассмотрим немного позже в этой главе).
- Update Rule (обновить правило).

В коде T-SQL вы можете объявить ограничения внешнего ключа либо при создании таблиц, либо после этого. После определения столбца фраза `FOREIGN KEY REFERENCE`, за которой следует имя таблицы (и при необходимости — столбца), позволяет создать внешний ключ, например:

```
ForeignKeyColumn FOREIGN KEY REFERENCES PrimaryTable(PKID)
```

В следующем примере, взятом из учебной базы данных CHA, создается связующая таблица `tour_mm_guide`, разрешающая отношение "многие ко многим". Эта таблица имеет два внешних ключа: по одному для таблиц `Tour` и `Guide`. Внешний ключ `TourID` явно ссылается на столбец первичного ключа, а внешний ключ `GuideID` указывает на таблицу и использует первичный ключ по умолчанию:

```
CREATE TABLE dbo.Tour_mm_Guide (
  TourGuideID INT
    IDENTITY
    NOT NULL
    PRIMARY KEY NONCLUSTERED,
  TourID INT
    NOT NULL
    FOREIGN KEY REFERENCES dbo.Tour(TourID)
    ON DELETE CASCADE,
  GuideID INT
    NOT NULL
    FOREIGN KEY REFERENCES dbo.Guide
    ON DELETE CASCADE,
  QualDate DATETIME NOT NULL,
  RevokeDate DATETIME NULL
  )
  ON [Primary];
```

Некоторые проектировщики баз данных предпочитают включать ограничения внешних ключей в определения таблиц, другие добавляют их уже после создания таблиц. Если таблица уже существует, то вы можете создать ограничение первичного ключа, используя инструкцию `ALTER TABLE ADD CONSTRAINT`:

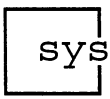
```
ALTER TABLE SecondaryTableName
  ADD CONSTRAINT ConstraintName
    FOREIGN KEY (ForeignKeyColumns)
    REFERENCES dbo.PrimaryTable (PrimaryKeyColumnName);
```

Таблица `Person` базы данных `Family` должна использовать этот метод, поскольку использует рекурсивное отношение. Внешний ключ не может быть создан раньше соответствующего первичного ключа. Так как рекурсивное отношение ссылается на ту же таблицу, она должна быть создана раньше внешнего ключа.

В следующем примере, скопированном из файла `family_create.sql`, создается таблица `Person`, после чего настраиваются внешние ключи `MotherID` и `FatherID`:

```
CREATE TABLE dbo.Person (
  PersonID INT NOT NULL PRIMARY KEY NONCLUSTERED,
  LastName VARCHAR(15) NOT NULL,
  FirstName VARCHAR(15) NOT NULL,
  SrJr VARCHAR(3) NULL,
```

```
  MaidenName VARCHAR(15) NULL,
  Gender CHAR(1) NOT NULL,
  FatherID INT NULL,
  MotherID INT NULL,
  DateOfBirth DATETIME NULL,
  DateOfDeath DATETIME NULL
  );
go
  ALTER TABLE dbo.Person
    ADD CONSTRAINT FK_Person_Father
      FOREIGN KEY(FatherID) REFERENCES dbo.Person (PersonID);
  ALTER TABLE dbo.Person
    ADD CONSTRAINT FK_Person_Mother
      FOREIGN KEY(MotherID) REFERENCES dbo.Person (PersonID);
```



Для получения списка внешних ключей текущей базы данных программным путем выполните запрос к представлению каталога `sysforeign_key_columns`.

Дополнительные внешние ключи

Между обязательными и дополнительными внешними ключами существует важное отличие. Некоторые отношения требуют наличия внешнего ключа (например, запись таблицы строк заказов должна ссылаться на строку самого заказа в другой таблице), в то время как другие отношения не требуют наличия значения. В последнем случае данные допустимы как при наличии внешнего ключа, так и при его отсутствии. Все определяется логической моделью.

На физическом уровне различие заключено в допустимости пустых значений во внешних ключах. Если внешний ключ обязателен, то столбец не должен допускать пустые значения; необязательные внешние ключи их допускают. Для полной своей реализации некоторые сложные отношения могут потребовать наличия ограничений проверки или триггеров.

Самое распространенное определение ссылочной целостности сводится к отсутствию "висячих" строк и датируется еще теми днями, когда первичные таблицы назывались *родительскими*, а вторичные — *дочерними*. Необязательные внешние ключи являются исключением из этого правила. Для них можно сформулировать следующее определение: "Висячие строки допустимы, однако если существует родитель, он должен быть определен".

Каскадные удаления и обновления

Сложности, создаваемые ссылочной целостностью, сводятся к недопустимости удаления или модификации строки первичной таблицы до тех пор, пока существуют строки вторичной таблицы, ссылающиеся на нее (т.е. строку). Если последние остаются после удаления строки первичной таблицы, то нарушаются требования ссылочной целостности.

Решение этой проблемы сводится к модификации строк вторичной таблицы как части транзакции, изменяющей строки первичной таблицы. Декларативная ссылочная целостность может выполнить эту работу вместо вас. Для строк вторичной таблицы возможны четыре варианта действий, выбираемых в параметрах Delete Rule и Update Rule формы Foreign Key Relationships. Параметр Update Rule имеет смысл только для естественных первичных ключей.

- **No Action**. Строки вторичной таблицы не изменяются. Их присутствие будет предохранять строки первичной таблицы от удаления и модификации.

  Используйте этот вариант, когда вторичные строки сопровождают значения первичных строк. Вы не сможете изменить или удалить первичные строки, если существуют

связанные с ними вторичные. Например, если для какого-то клиента выписан счет-фактура, вы не сможете его удалить.

- **Cascade**. Операции удаления и модификации, выполняемые на первичных строках, выполняются также и на вторичных.

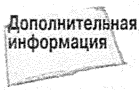
  Используйте этот вариант, когда вторичные данные теряют смысл в отсутствие первичных. Например, если удаляется заказ с номером 123, все его строки будут также удаляться. Если номер заказа изменяется с 123 на 456, то его строки также должны изменить номер заказа на это значение (предполагается, что используется естественный первичный ключ).

- **Set Null**. Этот вариант оставляет вторичные строки нетронутыми, однако изменяет значение вторичного ключа на пустое (разумеется, предполагается допустимость наличия пустых значений).

  Используйте этот вариант, если хотите разрешить удаление первичных строк без влияния на существование вторичных. Например, если удаляется запись о школьном классе, строки с данными об учениках не должны удаляться, поскольку их существование в базе данных самодостаточно.

- **Set Default**. Первичные строки могут удаляться и изменяться, при этом значения внешнего ключа изменяются на значения по умолчанию, заданные для данного столбца.

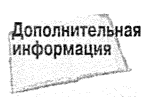
  Этот вариант аналогичен варианту `Set Null`, за исключением того, что на этот раз внешнему ключу присваивается определенное значение. В схемах, использующих суррогатные пустые значения (например, пустые строки), установка значений столбца по умолчанию и определение правила удаления как `Set Default` приведет к занесению во внешний ключ пустой строки при удалении строки первичного ключа.



О каскадных удалениях, а также о сопровождающих их проблемах см. в разделе "Удаление данных" главы 16.

В коде T-SQL добавление параметра `ON DELETE CASCADE` в ограничение внешнего ключа форсирует каскадные операции. В следующем примере, взятом из базы данных OBXKites, таблица строк заказа `OrderDetail` использует параметр каскадного удаления в ограничении внешнего ключа `OrderID`:

```
CREATE TABLE dbo.OrderDetail (
  OrderDetailID UNIQUEIDENTIFIER
    NOT NULL
    ROWGUIDCOL
    DEFAULT (NEWID())
    PRIMARY KEY NONCLUSTERED,
  OrderID UNIQUEIDENTIFIER
    NOT NULL
    FOREIGN KEY REFERENCES dbo.[Order]
      ON DELETE CASCADE,
  ProductID UNIQUEIDENTIFIER
    NULL
    FOREIGN KEY REFERENCES dbo.Product,
```



В главе 24, посвященной расширенным решениям на языке T-SQL, будет показано, как создать триггеры, обеспечивающие ссылочную целостность, и каскадные удаления в нестандартных схемах данных в пределах одной или нескольких баз данных.

Создание пользовательских столбцов данных

Эти столбцы хранят данные пользователя. Формально их можно разделить на две категории: используемые для идентификации людей, мест, событий или действий и используемые для дополнительного описания людей, мест, событий и действий.

Таблицы SQL Server могут содержать до 1024 столбцов; в то же время считается, что в таблице хорошо сформированной реляционной базы данных их число не должно превышать двадцати пяти.

Столбцы данных создаются в процессе определения таблицы и перечисляются в инструкции CREATE TABLE. Столбцы перечисляются в круглых скобках, при этом указывается имя столбца, используемый в нем тип данных, а также дополнительные атрибуты, ограничения, допустимость пустых значений и значения по умолчанию. Общий синтаксис следующий:

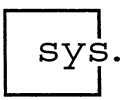
```
CREATE TABLE имя_таблицы (
имя_столбца тип_данных атрибуты,
имя_столбца тип_данных атрибуты
);
```

Столбцы данных могут добавляться и в существующие таблицы с помощью инструкции ALTER TABLE ADD COLUMN:

```
ALTER TABLE имя_таблицы
  ADD имя_столбца тип_данных атрибуты;
```

Существующие столбцы могут быть изменены с помощью инструкции ALTER TAQBLE ALTER COLUMN:

```
ALTER TABLE имя_таблицы
  ALTER COLUMN имя_столбца
    новый_тип_данных атрибуты;
```



Для того чтобы получить список столбцов текущей базы данных, выполните запрос к представлениям каталогов `sysobjects` и `syscolumns`.

Типы данных столбцов

Тип данных столбца служит для двух целей.

- Он обеспечивает первый уровень целостности данных. Символьные данные не могут быть занесены в столбцы типов даты-времени и числового типа. На практике мне встречались базы данных, все столбцы которых имели тип `nvarchar` для облегчения ввода данных. Однако такой подход нельзя назвать правильным. Тип данных играет важную роль как инструмент проверки допустимости данных, и его лучше не обходить вниманием.
- Он определяет объем дискового пространства, выделяемого для столбца.

Символьные типы данных

SQL Server поддерживает несколько символьных типов данных (табл. 17.2).

Типы данных, использующие таблицу Unicode, особенно полезны для хранения текста, написанного на нескольких языках. Однако за это приходится расплачиваться удвоением занимаемого пространства. Некоторые разработчики используют для всех своих символьных

столбцов тип nvarchar, в то время как другие не хотят его использовать ни за какие деньги. Лично я советую использовать данные Unicode только для текста на иностранных языках; в остальных случаях лучше использовать типы char, varchar и text.

Таблица 17.2. Символьные типы данных

Тип данных	Описание	Размер в байтах
Char (n)	Символьные данные фиксированной длины. Могут содержать до 8000 символов и использовать принятый по умолчанию порядок и набор символов	Заданная длина, умноженная на 1 байт
Nchar(n)	Символьные данные фиксированной длины в таблице Unicode	Заданная длина, умноженная на 2 байта
Varchar(n)	Символьные данные переменной длины. Могут содержать до 8000 символов и использовать принятый по умолчанию порядок и набор символов	По 1 байту на символ
varchar(max)	Символьные данные переменной длины. Могут содержать до 2 Гбайт информации и использовать принятый по умолчанию порядок и набор символов	По 1 байту на символ
nvarchar(n)	Символьные данные переменной длины, хранящие до 8000 символов при использовании порядка, принятого по умолчанию	По 2 байта на символ
nvarchar(max)	Символьные данные переменной длины, хранящие до 2 Гбайт. Используется таблица Unicode и порядок, принятый по умолчанию	По 2 байта на символ
text	Символьные данные переменной длины, содержащие до 2147483647 символов	По 1 байту на символ
ntext	Символьные данные в таблице Unicode, содержащие до 1073741823 символов	По 2 байта на символ
sysname	Пользовательский тип данных, используемый для имен таблиц и столбцов, — эквивалент типа nvarchar(128)	По 2 байта на символ

Числовые типы данных

SQL Server поддерживает несколько числовых типов данных (табл. 17.3).

Таблица 17.3. Числовые типы данных

Тип данных	Описание	Размер в байтах
bit	1 или 0	1 бит
tinyint	Целые числа от 0 до 255	1 байт
smallint	Целые числа от -32768 до 32767	2 байта
int	Целые числа от -2147483648 до 2147483647	4 байта
bigint	Целые числа от -2^63 до 2^63	8 байтов
decimal или numeric	Числа с фиксированной точностью до 10^38+1	Варьируется в зависимости от длины
money	Числа от -2^63 до 2^63 с точностью до одной десятитысячной	8 байтов

Тип данных	Описание	Размер в байтах
`smallmoney`	Числа от -214748,3648 до 214748,3647 с точностью до одной десятитысячной	4 байта
`float`	Числа с плавающей запятой от -1,79E+308 до 1,79E+308	4 байта или 8 байтов в зависимости от точности
`real`	Числа с плавающей запятой с точностью до 24 знаков	4 байта



При работе с денежными значениями особо тщательно выбирайте тип данных. Использование для них типов `float` и `real` может привести к ошибкам округления. Типы данных `money` и `smallmoney` имеют фиксированную точность в четыре знака (т.е. до одной сотой копейки). Для некоторых монетарных значений клиенту может понадобиться точность до одной копейки. В этом случае более подходящим окажется тип `decimal`.

Типы данных даты-времени

SQL Server хранит в одном столбце дату и время и использует для этого типы данных `datetime` и `smalldatetime` (табл. 17.4). Основным раличием между этими двумя типами является точность и учет столетия. Если в столбце должна храниться только дата и при этом не должен учитываться период до XX века, то вполне подойдет тип `smalldatetime`. Если требуется дополнительно хранить значение времени, точности типа `smalldatetime` может не хватить.

Таблица 17.4. Типы данных даты-времени

Тип данных	Описание	Размер в байтах
`datetime`	Значения даты и времени от 1 января 1753 года до 31 декабря 9999 года с точностью до трех миллисекунд	8 байтов
`smalldatetime`	Значения даты и времени с 1 января 1900 года до 6 июня 2079 года с точностью до одной минуты	4 байта

Юлианский календарь вступил в силу с 1 января 1753 года. Так как СУБД SQL Server не хочет разбираться, какие нации и религии и как использовали даты до 1753 года, она вообще исключает их из рассмотрения. Хотя в общем случае это не составляет особой проблемы, некоторые исторические и генеалогические базы данных требуют использования и более ранних дат. В качестве обходного маневра я рекомендую использовать для таких данных столбец на основе типа `char`, применяя при этом триггер или хранимую процедуру для проверки форматирования и допустимости даты при вводе.



Некоторые программисты (не являющиеся администраторами баз данных) выбирают для столбцов дат символьные типы данных, однако это может привести к путанице при выполнении преобразований. Используйте для фильтрации корректных дат функцию `IsDate()`.

Прочие типы данных

Прочие типы данных перечислены и описаны в табл. 17.5. Они способны покрыть потребности двоичных объектов и вариантных данных.

Таблица 17.5. Прочие типы данных

Тип данных	Описание	Размер в байтах
`timestamp`	Уникальное в пределах базы данных случайное значение, генерируемое при каждом обновлении	8 байтов
`uniqueidentifier`	Генерируемое системой 16-байтовое значение	16 байтов
`binary(n)`	Данные фиксированной длины вплоть до 8000 байтов	Определяется длиной
`binary(max)`	Данные фиксированной длины вплоть до 8000 байтов	Определяется длиной
`varbinary`	Двоичные данные переменной длины вплоть до 8000 байтов	Число используемых байтов
`image`	Двоичные данные переменной длины вплоть до 2147483647 байтов	Число используемых байтов
`sql_variant`	Может хранить любые типы данных длиной до 2147483647 байтов	

Тип данных `timestamp`, ранее известный как `rowversion`, может пригодиться для поиска потерянных обновлений (подробно о нем речь пойдет в главе 51). Тип `uniqueidentifier` пригодится для создания первичных ключей, особенно в реплицируемых базах данных. Более подробно об этом типе см. выше, в разделе "Создание первичных ключей".

Вычисляемые столбцы

Вычисляемые столбцы имеют в таблицах особую ценность, поскольку содержат результат предопределенного выражения так же, как и представления (т.е. сохраненные инструкции `SELECT`), однако без излишней нагрузки на систему. Такие столбцы не хранят реальных данных — они их вычисляют в момент выполнения запроса.

Вычисляемые столбцы также положительно влияют на целостность данных. Они вычисляют значения на уровне таблицы, не доверяя эту операцию каждому конкретному пользователю. По ним могут даже создаваться индексы.

Синтаксис вычисляемых столбцов противоположный присвоению псевдонима:

Имя_столбца `AS` *выражение*

Таблица `OrderDetail` учебной базы данных OBXKites содержит вычисляемый столбец для расширенной цены, что и продемонстрировано далее в ее определении:

```
CREATE TABLE dbo.OrderDetail (
...
  Quantity NUMERIC(7,2) NOT NULL,
  UnitPrice MONEY NOT NULL,
  ExtendedPrice AS Quantity * UnitPrice Persisted,
...
  )
  ON [Primary];
Go
```



В версии SQL Server 2005 вычисляемые столбцы могут оставаться на диске. Это значительно повышает производительность базы данных при выполнении поиска.

Ограничения и значения столбцов по умолчанию

База данных всегда настолько хороша, насколько хороши ее данные. Ограничения представляют собой высокоскоростную проверку допустимости значений или правил бизнес-логики, выполняемую на уровне ядра базы данных. Кроме проверки типа данных, SQL Server содержит пять типов ограничений.

- **Ограничения первичного ключа**. Гарантируют уникальность первичного ключа и отсутствие в нем пустых значений.
- **Ограничения внешнего ключа**. Гарантируют указание значения на допустимый ключ.
- **Допустимость пустых значений**. Проверяют наличие в столбце пустых значений, если они недопустимы.
- **Ограничения проверки**. Эти булевы ограничения задаются пользователем.
- **Ограничения на уникальность**. Гарантируют уникальность значений.

SQL Server также предлагает использование параметров столбцов.

- **Значение по умолчанию**. Если инструкция `INSERT` не вставляет в столбец никакого значения, в него подставляется значение по умолчанию.

Значения столбца, заданные по умолчанию, на одной из страниц Books Online рассматриваются как один из типов ограничений, однако на другой странице в списке ограничений они не упомянуты. Я считаю это свойством столбца, так как на самом деле никаких проверок ввода данных пользователем не проводится; также не форсируется поддержание целостности данных. В то же время значения, заданные по умолчанию, являются ценным параметром столбца.

Допустимость пустых значений

Пустое значение можно расценивать как неизвестное. Как правило, пустые значения появляются при неполном вводе пользователем данных в строку.



О том, как определять, выявлять и обрабатывать пустые значения, см. в главе 8.

Допустимы ли в столбце пустые значения, определяется с помощью атрибута столбца `NULL` или `NOT NULL`.

По умолчанию в новых столбцах SQL Server предполагает недопустимость пустых значений, однако этот режим работы можно изменить с помощью свойства подключения `ansi_null_dflt_on`. Стандарт ANSI по умолчанию предполагает допустимость пустых значений, если, конечно, в определении столбца не было явно указано `NOT NULL`.



Так как по умолчанию SQL Server и стандарт ANSI предполагают противоположные режимы допустимости пустых значений, лучше не полагаться на случай и при определении столбца явно указывать параметр `NULL` или `NOT NULL`.

В следующем примере продемонстрированы режимы работы с пустыми значениями ANSI и SQL Server, принятые по умолчанию. В первом примере демонстрируется работа SQL Server. Вначале общий параметр `ansi null` устанавливается в ложное значение `false`, а параметр подключения `ansi_null_dflt_off` — в значение `on`:

```
USE TempDB;
EXEC sp_dboption 'TempDB', ANSI_NULL_DEFAULT, 'false';
SET ANSI_NULL_DFLT_OFF ON;
```

Далее создается таблица `NullTest` без явного указания допустимости пустых значений:

```
CREATE TABLE NullTest(
  PK INT IDENTITY,
  One VARCHAR(50)
  );
```

Затем мы пытаемся вставить в таблицу пустое значение:

```
INSERT NullTest(One)
  VALUES (NULL);
```

и в результате получаем ошибку:

```
Server: Msg 515, Level 16, State 2, Line 1
Cannot insert the value NULL into column 'One',
table 'TempDB.dbo.NullTest';
column does not allow nulls. INSERT fails.
The statement has been terminated.
```

Поскольку режим допустимости пустых значений был установлен в принятый по умолчанию в SQL Server на момент создания таблицы, пустое значение таблица не приняла. Во втором примере мы перестроим эту таблицу с использованием стандартов ANSI:

```
EXEC sp_dboption 'TempDB', ANSI_NULL_DEFAULT, 'true';
SET ANSI_NULL_DFLT_ON ON;

DROP TABLE NullTest;

CREATE TABLE NullTest(
  PK INT IDENTITY,
  One VARCHAR(50)
  );
```

Теперь мы пытаемся вставить в нее пустое значение:

```
INSERT NullTest(One)
  VALUES (NULL);
```

и получаем положительный результат:

```
(1 row(s) affected)
```

Ограничения на уникальность

Ограничение на уникальность аналогично уникальному индексу или ограничению первичного ключа. Оно гарантирует то, что в разных строках столбца не будут содержаться одинаковые значения. Этот параметр чаще всего используется, когда в таблице нужно обеспечить отсутствие дублирующихся строк (например, в списке сотрудников уникальным должен быть идентификационный код, а в списке клиентов — ОКПО).

В утилите Management Studio ограничение на уникальность устанавливается во вкладке Index диалогового окна свойств таблицы. Этот процесс идентичен установке индекса, но с одним отличием: в этом случае вместо индекса выбирается ограничение.

В программном коде ограничение на уникальность можно установить, установив в определении столбца параметр `UNIQUE`. Приведем пример:

```
CREATE TABLE Employee (
  EmployeeID INT PRIMARY KEY NONCLUSTERED,
  EmployeeNumber CHAR(8)UNIQUE,
  LastName NVARCHAR(35),
```

```
  FirstName NVARCHAR(35)
  );
Insert Employee (EmployeeID, EmployeeNumber, LastName, FirstName)
  Values( 1, '1', 'Wilson', 'Bob');
Insert Employee (EmployeeID, EmployeeNumber, LastName, FirstName);
  Values( 2, '1', 'Smith', 'Joe');
```

В результате выполнения этого пакета инструкций будет получена ошибка:

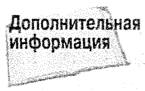
```
Server: Msg 2627, Level 14, State 2, Line 1
Violation of UNIQUE KEY constraint 'UQ__Employee__68487DD7'.
Cannot insert duplicate key in object 'Employee'.
The statement has been terminated.
```

Чтобы добавить ограничение на уникальность в уже существующую таблицу, можно использовать инструкцию ALTER TABLE:

```
ALTER TABLE Employee
  ADD CONSTRAINT EmpNumUnique
    UNIQUE (EmployeeNumber);
```

Ограничения проверки

Ограничения проверки являются проверкой целостности данных на уровне строк. Как правило, это небольшая формула, которая возвращает булево значение `true` или `false`. Ограничения проверки имеют доступ ко всем данным текущей строки — они не могут обратиться к другим строкам или выполнить поиск. В ограничения проверки могут быть включены скалярные функции, о которых мы говорили в главе 8.



Ограничения проверки могут содержать пользовательские скалярные функции (о них речь пойдет в главе 22), которые могут выполнять последовательность инструкций T-SQL. В результате вызов такой функции в ограничении проверки открывает почти неограниченные возможности, в том числе и возможность сложного поиска. В то же время сложные проверки правил бизнес-логики чаще всего выполняются в триггерах `after`.

Ограничения проверки полезны для поддержания общих правил допустимости данных, а также простых правил бизнес-логики. В качестве примеров можно привести проверку превышения датой увольнения даты приема на работу (или даты рождения плюс 18 лет).



Ограничения проверки на порядок быстрее табличных триггеров. Если некоторое правило можно проверить с помощью такого ограничения, то используйте его, а не триггер.

В следующем примере ограничение обеспечивает превышение табельным номером сотрудника (поле `EmployeeNumber`) значения 1:

```
Drop Table Employee
CREATE TABLE Employee (
  EmployeeID INT PRIMARY KEY NONCLUSTERED,
  EmployeeNumber CHAR(8) CHECK (EmployeeNumber > '1'),
  LastName NVARCHAR(35),
  FirstName NVARCHAR(35)
  );
Insert Employee (EmployeeID, EmployeeNumber, LastName, FirstName)
   Values( 2, '1', 'Smith', 'Joe');
```

Выполнение этого пакета инструкций приведет к ошибке:

```
Server: Msg 547, Level 16, State 1, Line 1
INSERT statement conflicted with COLUMN CHECK constraint
'CK__Employee__Employ__5FB337D6'.
The conflict occurred in database 'tempdb',
table 'Employee', column 'EmployeeNumber'.
The statement has been terminated.
```

Для добавления ограничения проверки в существующую таблицу используют инструкцию ALTER TABLE:

```
ALTER TABLE Employee
  ADD CONSTRAINT NoHireSmith
    CHECK (Lastname <> 'SMITH');
```

Значения по умолчанию

Это значение SQL Server вставляет в таблицу, если оно не было определено в инструкции INSERT в явном виде. Значения по умолчанию приобретают особое значение, когда в столбце недопустимы пустые значения. В этом случае, если не вставить в такой столбец значение, а значения по умолчанию в нем не определены, вся операция вставки будет отвергнута.

Значение по умолчанию может быть одним из следующих.

- Допустимое статическое числовое или символьное значение, такое как `123` или `local`.
- Скалярная системная функция, такая как `GetDate()` или `NewID()`.
- Скалярная функция, определенная пользователем.
- Пустое значение.

Тип значения по умолчанию должен быть совместим с типом данных столбца.

Если таблица создается в Management Studio, то значение по умолчанию определяется как одно из свойств столбца.

В программном коде значение по умолчанию добавляется как один из параметров определения столбца при создании таблицы или уже впоследствии, с помощью инструкции ALTER TABLE CREATE CONSTRAINT.

Следующий пример (с сокращениями) взят из определения таблицы Product учебной базы данных OBXKites. Значением столбца по умолчанию ActiveDate является текущая дата:

```
CREATE TABLE dbo.Product (
...
  ActiveDate DATETIME NOT NULL DEFAULT GETDATE(),
...
  );
```

Значения по умолчанию можно устанавливать и после создания таблицы. В следующем примере для определения текущих ограничений запускается хранимая процедура sp_help, затем ограничение удаляется и впоследствии заново устанавливается с помощью инструкции ALTER TABLE:

```
sp_help Product;
```

Результат будет следующим:

```
constraint_type           constraint_name
------------------------- ------------------
DEFAULT on column ActiveDate
                          DF__Product__ActiveD__7F60ED59
```

Теперь инструкция ALTER TABLE удаляет существующее значение по умолчанию:

```
ALTER TABLE Product
  DROP CONSTRAINT DF__Product__ActiveD__7F60ED59
```

после чего мы снова устанавливаем его:

```
ALTER TABLE Product
  ADD CONSTRAINT ActiveDefault
  DEFAULT GetDate() FOR ActiveDate;
```

Каталог данных

Несмотря на то что в SQL Server формально отсутствует каталог данных, ему на замену могут прийти пользовательские типы данных. Пользовательские типы представляют собой именованные объекты, обладающие следующими дополнительными свойствами.

- Определены тип данных и длина.
- Определена допустимость пустых значений.
- Предопределены правила, которые могут быть применены к данному типу данных.
- Предопределены значения по умолчанию, применимые к типу данных.

В сильно нормализованных базах данных, в которых не существует сходных данных в разных таблицах, концепция каталога данных может показаться неуместной. В то же время хороший стандарт типа данных в обычном магазине может оказаться очень полезным. Например, если в каждой из баз данных перечислены одни и те же сотрудники, программирование всех уровней облегчается и позволит избежать ошибок. Чтобы создать каталог данных из правил, значений по умолчанию и пользовательских типов данных, а затем применить все это ко множеству баз данных, лучше всего создать отдельный сценарий `DataCatalog.sql`, а затем запускать его в каждой базе или поместить в базу данных модели.

Правила, определяемые пользователем

Правило аналогично ограничению проверки, за исключением того, что создается независимо и только затем применяется к столбцу. Правило состоит только из имени и булева выражения. Булево выражение может ссылаться на данные с помощью символа @, за которым следует имя столбца.

В следующем примере продемонстрировано создание правила, которое проверяет дни рождения и гарантирует, что все они в прошлом:

```
-- Определяемое пользователем правило
CREATE RULE BirthdateRule AS @Birthdate <= Getdate();
```

Чтобы применить это правило к столбцу таблицы или пользовательскому типу данных, используется хранимая процедура `sp_bindrule`. Первым аргументом этой процедуры является имя правила, а вторым — объект, к которому оно применяется. В следующем примере вышеописанное правило `BirthdayRule` применяется к столбцу `BirthDate` таблицы `Person`:

```
EXEC sp_bindrule
  @rulename = 'BirthdateRule',
  @objname = 'Person.Birthdate';
```



Правила рассматриваются как средство обратной совместимости версий SQL Server. Их использование не рекомендовано компанией Microsoft. На смену им пришли ограничения проверки, помещаемые непосредственно в столбцы. Правила можно рассматривать скорее с точки зрения практики программирования, а не использования. Если в вашей базе данных пока еще остались правила, то замените их ограничениями проверки.

В утилите Management Studio правила создаются и применяются в узле Rules каждой из баз данных. В то же время большинство разработчиков, использующих правила, выносят их во внешний сценарий.

Значения по умолчанию, определяемые пользователем

Значения по умолчанию легко создать непосредственно в определении таблицы, хотя, подобно правилам, они продолжают существование исключительно из соображений обратной совместимости. Объекты значений по умолчанию представляют собой именованные значения, которые могут быть применены ко множеству таблиц. Значения по умолчанию могут создаваться и применяться к столбцам в Management Studio в узле Defaults каждой из баз данных.

В следующем примере создается значение по умолчанию, равное текущей дате, после чего оно применяется к столбцу Hiredate:

```
CREATE DEFAULT HireDefault AS Getdate()
go
sp_bindefault 'HireDefault', 'Contact.Hiredate';
```

Пользовательские типы данных

Пользовательские типы данных назначают имена системным типам и применяют к ним ограничения допустимости пустых значений. Именованные пользовательские типы данных затем могут использоваться так же, как и системные, в любом определении таблицы.

Тип данных SysName разработан компанией Microsoft специально для хранения системных имен (таблиц, столбцов) в столбцах.

В утилите Management Studio пользовательские типы данных создаются в узле User-Defined Data Type каждой из баз данных. Для их создания также может использоваться системная хранимая процедура sp_addtype, которой в качестве аргументов передаются имя, тип данных и параметр допустимости пустых значений. В следующем примере создается пользовательский тип данных, к которому применяются значение по умолчанию и правило, а затем этот тип данных используется при создании таблицы:

```
EXEC sp_addtype
  @typename = Birthdate,
  @phystype = SmallDateTime,
  @nulltype = 'NOT NULL';
go
EXEC sp_bindefault
  @defname = 'BirthdateDefault',
  @objname = 'Birthdate',
  @futureonly = 'futureonly';
EXEC sp_bindrule
  @rulename = 'BirthdateRule',
  @objname = 'Person.Birthdate';
```

Триггеры DDL

Триггером называется программный код, который выполняется в результате некоторого действия. Триггеры DML запускаются на выполнение в результате выполнения инструкций DML — INSERT, UPDATE и DELETE. Триггеры DDL выполняются в результате инструкций CREATE, ALTER и DROP. Триггеры DML отвечают на изменения данных, а триггеры DDL — на изменения схемы. Триггерам DDL будет посвящена отдельная глава; в этом разделе мы всего лишь коснемся темы и приведем несколько примеров.



Триггеры DDL появились в версии SQL Server 2005 и предназначены для аудита изменений на уровне схемы.

Триггеры DDL могут отвечать на изменения уровня либо схемы базы данных, либо сервера. Так как триггеры могут реагировать на множество типов событий и команд, команды, вызвавшие триггер, передаются ему в формате XML с помощью функции `EventData()`.



Лучше узнать о триггерах DML и областях их применения вы сможете в главе 23.

Создание и изменение триггеров DDL

Триггеры DDL создаются и изменяются с использованием синтаксиса, аналогичного триггерам DML. Область применения триггера задается в предложении `ON` либо параметром `ALL SERVER`, либо `DATABASE` (именно "DATABASE", а не имя базы данных).

```
CREATE | ALTER TRIGGER TriggerName
ON ALL SERVER | DATABASE
{WITH параметр}
FOR | AFTER тип_события or группа_событий,...n
AS
программный код
```

Аргументами могут выступать ключевые слова `encryption` и `execute as` — они оба доказали свою востребованность в триггерах аудита системного уровня. Существуют десятки событий и их групп — по одному для каждого типа операции DDL, которая может быть выполнена на сервере или в базе данных. Несколько наиболее интересных событий перечислены в этом разделе, с остальными вы можете ознакомиться в утилите Books Online.

Триггеры DDL уровня сервера могут отвечать на следующие события.

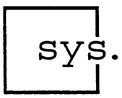
- `Alter Authorization Server` (изменения в сервере авторизации).
- `Create|Alter|Drop Database` (создание, изменение и удаление базы данных).
- `Create|Alter|Drop Login` (создание, изменение и удаление регистрационной записи).
- `Create|Drop Endpoint` (создание и удаление концевой точки).
- `Create|Alter|Drop Server Access` (создание, изменение и удаление уровня доступа к серверу).

Триггеры DDL уровня базы данных отвечают на следующие события.

- `Create|Alter|Drop Application Role` (создание, изменение и удаление роли приложения).
- `Grant|Deny|Revoke Database` (предоставление, отзыв и запрет доступа к базе данных).
- `Create|Drop Event Notification` (создание и удаление уведомления о событии).
- `Create|Alter|Drop Function` (создание, изменение и удаление функции).
- `Create|Alter|Drop Index` (создание, изменение и удаление индекса).
- `Create|Alter|Drop Procedure` (создание, изменение и удаление процедуры).
- `Create|Alter|Drop Role` (создание, изменение и удаление роли).
- `Create|Alter|Drop Schema` (создание, изменение и удаление схемы).

- `Create|Alter|Drop Synonym` (создание, изменение и удаление синонима).
- `Create|Alter|Drop Table` (создание, изменение и удаление таблицы).
- `Create|Alter|Drop Trigger` (создание, изменение и удаление триггера).
- `Create|Alter|Drop User` (создание, изменение и удаление пользователя).
- `Create|Alter|Drop View` (создание, изменение и удаление представления).
- `Create|Alter|Drop XML Schema Collection` (создание, изменение и удаление коллекции схем XML).

В утилите Management Studio триггеры уровня баз данных перечислены в узле `Programmability` окна Object Explorer. Триггеры баз данных могут быть запрограммированы, но не так легко изменены, как другие программные объекты, такие как хранимые процедуры. Триггеры уровня сервера перечисляются в узле `Server Objects` окна Object Explorer.



Получить список триггеров DDL уровня базы данных можно программным путем, выполнив запрос к представлению каталога `systriggers` или `sysevents`. Список серверных триггеров можно получить в представлениях `sysserver_triggers` и `sysserver_trigger_events`.

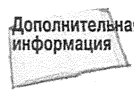
Для удаление триггера DDL используется инструкция `DROP TRIGGER`:

```
DROP TRIGGER имя_триггера
```

Функция `EventData()`

Триггеры DDL могут отвечать на множество различных событий, и поэтому им нужен некоторый метод получения данных о том событии, которое их инициировало. Триггеры DML имеют логические таблицы вставки и удаления; триггеры DDL имеют функцию `EventData()`. Эта функция возвращает данные о событии в формате XML. Схема XML варьируется в зависимости от типа события. Следует обратить внимание на то, что части схемы XML чувствительны к регистру символов.

Используя функцию `EventData()` для заполнения переменной XML, триггер может использовать запросы XQuery для исследования значений.



Более подробно об XML и запросах XQuery вы узнаете в главе 31.

В следующем примере триггер DDL извлекает информацию об изменениях в таблицах базы данных CHA2 и записывает эти изменения в таблицу аудита:

```
USE CHA2;
CREATE TABLE SchemaAudit (
  AuditDate DATETIME NOT NULL,
  UserName VARCHAR(50) NOT NULL,
  Object VARCHAR(50) NOT NULL,
  DDLStatement VARCHAR(max) NOT NULL
  )
go
CREATE TRIGGER SchemaAudit
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
DECLARE @EventData XML
SET @EventData = EventData()
```

```
INSERT SchemaAudit (AuditDate, UserName, Object, DDLStatement)
SELECT
  GetDate(),
  @EventData.value('data(/EVENT_INSTANCE/UserName)[1]',
    'SYSNAME'),
  @EventData.value('data(/EVENT_INSTANCE/ObjectName)[1]',
    'VARCHAR(50)'),
@EventData.value('data(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
   'VARCHAR(max)')
GO
CREATE TABLE Test (
  PK INT NOT NULL
  )
GO
DROP TABLE Test
GO
SELECT * FROM SchemaAudit
```

Будет получен следующий результат:

```
AuditDate                UserName Object DDLStatement
------------------------ -------- ------ ---------------------
2006-03-07 13:07:24.437  dbo      Test   CREATE TABLE Test (PK INT NOT NULL)
2006-03-07 13:07:24.450  dbo      Test   DROP TABLE Test
```

Включение и отключение триггеров DDL

Триггеры DDL могут быть включены и отключены. И это хорошо, поскольку администраторы баз данных должны иметь простую методику отключения триггеров DDL, которые откатывают любые изменения в схеме. Следующий программный код отключает, а затем снова включает триггер SchemaAudit:

```
DISABLE TRIGGER SchemaAudit
ON Family;

ENABLE TRIGGER SchemaAudit
ON Family;
```

Резюме

Логическая схема базы данных часто требует правок, чтобы служить в качестве физической схемы. Практически невозможно, чтобы физическая схема базы данных имела форму логической и стала сразу работать без применения ограничений, ключей и типов данных. Зная возможности определения таблиц в SQL Server, вы можете реализовать некоторые функции проекта на уровне серверных ограничений, а не в коде T-SQL триггера или хранимой процедуры.

Настоящая глава представила вам программный код создания физической схемы базы данных. В следующих главах мы продолжим тему программирования. При этом первые несколько глав будут посвящены языку T-SQL; в остальных мы рассмотрим платформу .NET и другие технологии программирования.

Программирование на языке Transact-SQL

ГЛАВА

18

В этой главе...

Основы T-SQL и пакетов

Работа с локальными переменными

Управление потоком команд

Просмотр объектов SQL Server в программе

Работа с временными таблицами и табличными переменными

Создание динамических запросов

Использование инструкций SELECT с многочисленными присвоениями

Предупреждение и устранение ошибок

Стандартные инструкции языка манипулирования данными DML предназначены только для извлечения и изменения данных. Этот язык лишен средств разработки процедур и алгоритмов, а также команд настройки сервера и управления им. В качестве компенсации любая конкретная реализация языка SQL дополняет его своими расширениями.

Язык Transact-SQL, больше известный как T-SQL, является реализацией стандарта SQL, дополненной набором расширений Microsoft. Основной задачей языка T-SQL является обеспечения программиста набором средств проектирования транзакционных баз данных.

Язык T-SQL часто ассоциируют с хранимыми процедурами, но на самом деле он представляет нечто большее. В приложениях "клиент/сервер" он может быть задействован несколькими способами.

- T-SQL может использоваться в выражениях как часть команд DML INSERT, UPDATE и DELETE, порожденных клиентским процессом.
- T-SQL может использоваться в программных блоках, направляемых от клиента к серверу.
- Функции T-SQL могут использоваться в качестве выражений в ограничениях проверки.
- Код T-SQL может использоваться в хранимых процедурах, функциях и триггерах сервера баз данных.

Вопросы программирования в настоящей книге мы начали рассматривать с главы 7, и начали с инструкций DML, которые лежат в самом сердце языка T-SQL. В этой главе мы всего лишь рассмотрим программные элементы, необходимые для создания серверных приложений. Программные средства, рассматриваемые в настоящей главе, являются основой создания хранимых процедур, пользовательских функций и триггеров.



Несмотря на то что версия SQL Server 2005 внесла в систему элементы общеязыковой среды выполнения CLR и платформы .NET, язык T-SQL остался лучшим средством доступа к базе данных.

Основы Transact-SQL

Язык T-SQL предназначен для управления наборами данных. По этой причине он не обладает некоторыми характерными чертами традиционных языков, которые необходимы для программирования приложений. Если вы уже давно занимаетесь созданием приложений, то наверняка противопоставите мышление программирования в T-SQL и в других языках, таких как VB, C# и Java.

Пакеты T-SQL

Запросом называют одну инструкцию T-SQL, а *пакетом* — их набор. Вся последовательность инструкций пакета отправляется серверу из клиентских приложений как одна цельная единица.

SQL Server рассматривает весь пакет как рабочую единицу. Наличие ошибки хотя бы в одной инструкции приведет к невозможности выполнения всего пакета. В то же время грамматический разбор не проверяет имена объектов и схем, так как сама схема может измениться в процессе выполнения инструкции.

Прерывание выполнения пакета

Файл сценария SQL и окно анализатора запросов (Query Analyzer) может содержать несколько пакетов. В данном случае все пакеты разделяют ключевые слова терминаторов. По умолчанию этим ключевым словом является `GO`, и оно должно быть единственным в строке. Все другие символы (даже комментарии) нейтрализуют разделитель пакета.

Разделитель пакетов на самом деле является функцией Management Studio, а не самого сервера. Его можно изменить на странице Query Execution диалогового окна свойств программы, но я не рекомендовал бы этого делать (по крайней мере, друзьям).

Инструкции DDL

Некоторые инструкции DDL языка T-SQL, такие как `Create Procedure`, обязательно должны быть первыми инструкциями пакета. Очень длинные сценарии, которые создают множество объектов, часто требуют наличия нескольких разделителей пакетов. Так как SQL Server отдельно разбирает синтаксис по пакетам, такое наличие множества разделителей помогает локализовать ошибки.

Переключение между базами данных

В интерактивном режиме работы текущая база данных всегда отображается на панели инструментов, и в любой момент может быть изменена. В программном коде текущая база определяется с помощью ключевого слова `USE`. Это ключевое слово в пакете указывает, с какой именно базой данных будет выполняться работа, начиная с текущей точки:

```
USE CHA2
```

На практике рекомендуется явно определять текущую базу данных с помощью команды `USE` — незачем надеяться на пользователя.

Выполнение пакетов

Пакет может быть выполнен несколькими способами.

- Сценарий SQL в полном объеме (т.е. все входящие в него пакеты) может быть выполнен путем открытия файла `.sql` в редакторе SQL утилиты Manage ment Stu dio и нажатия клавиши <F5> (или щелчка на кнопке !Execute панели инструментов, или выбора в меню пункта Query⇨Execute). (Я настроил свою операционную систему Windows так, чтобы при двойном щелчке на файле `.sql` автоматически запускался анализатор запросов.)
- В редакторе SQL утилиты Management Studio могут быть выполнены и отдельные инструкции SQL. Для этого их нужно выделить и нажать клавишу <F5> (или щелкнуть на кнопке !Execute панели инструментов, или выбрать в меню пункт Query⇨Execute).
- В приложении пакет T-SQL можно выполнить с помощью ADO или ODBC.
- Сценарий T-SQL может быть выполнен с помощью утилиты командной строки SQLCmd с передачей ей имени файла `.sql` в качестве параметра.
- Утилита `SQLCmd` имеет несколько параметров и может быть легко сконфигурирована практически для любых нужд.



Подробно об утилите SQLCmd см. в главе 6, посвященной Management Studio.

Выполнение хранимой процедуры

В пакете SQL хранимая процедура вызывается с помощью ключевого слова `exec`. При этом следует придерживаться ряда правил. Так как разрывы строк для SQL Server не имеют смысла, то команда `exec` означает окончание предыдущей инструкции.

Если вызов хранимой процедуры находится в первой строке пакета (или вообще является единственной инструкцией), то в нем не обязательно указывать ключевое слово `exec`. В то же время вставка этого ключевого слова не приведет к ошибке и к тому же поможет избежать проблем в будущем, если текст пакета будет изменен.

Следующие два вызова системной хранимой процедуры демонстрируют использование команды `exec` в пакете:

```
Sp_help;
EXEC sp_help;
```

В этом разделе мы рассмотрели только использование команды `exec` в пакете. Более подробная информация о творческом использовании ключевого слова `exec` содержится в разделе "Динамический SQL".

Форматирование в T-SQL

На протяжении всей этой книги программный код отформатирован для улучшения наглядности; в этом разделе мы рассмотрим ключевые моменты форматирования.

Завершение инструкции

Стандарт ANSI SQL требует помещения в конце каждой инструкции точки с запятой. В то же время при программировании на языке T-SQL точка с запятой не обязательна. При этом следует руководствоваться несколькими правилами.

- Не помещайте ее после оператора `try end`.
- Не помещайте ее после условия `if`.
- Обязательно помещайте ее после общетабличных выражений CTE.



Для лучшего восприятия программного кода все же рекомендуется использовать точки с запятой. В будущих версиях SQL Server их использование может стать обязательным, что может потребовать много дополнительной работы.

Продление строк

Инструкции T-SQL по своей природе имеют свойство быть длинными. Некоторые запросы последней главы с многочисленными объединениями и подзапросами занимают целую страницу. Лично мне нравится, что в T-SQL игнорируются символы пробелов и конца строки. Это значит, что длинная инструкция может быть продолжена на следующей строке без необходимости наличия какого-либо специального символа. Это свойство позволяет в значительной мере повысить читаемость программного кода.

В других реализациях SQL, таких как Access, для завершения инструкции требуется наличие точки с запятой. SQL Server допускает ее использование, но считает это необязательным.

Комментарии

Язык T-SQL допускает использование в одном пакете комментариев двух стилей: ANCI и языка C. Первый из них начинается с двух дефисов и заканчивается в конце строки:

```
-- Это комментарий стиля ANSI
```

Также комментарии стиля ANSI могут вставляться в конце строки инструкции:

```
Select FirstName, LastName - извлекаемые столбцы
FROM Persons - исходная таблица
Where LastName Like 'Hal%'; -- ограничение на строки
```

Редактор SQL может применять и удалять комментарии во всех выделенных строках. Для этого нужно соответственно выбрать команду меню Edit⇨Advanced⇨Comment Out (<Ctrl+C> или <Ctrl+K>) или Edit⇨Advanced⇨Remove Comments (<Ctrl+K> или <Ctrl+U>).

Комментарии стиля языка C начинаются с косой черты и звездочки (`/*`) и заканчиваются теми же символами в обратной последовательности. Этот тип комментариев лучше использовать для комментирования блоков строк, таких как заголовки или большие тестовые запросы.

```
/*
Триггер вставки таблицы Order
Пол Нильсен
версия 1.0 21 июля 2006 года
логика: и т.д.
версия 1.1: 31 июля 2006 года, добавлено то-то и то-то
*/
```

Одним из главных достоинств комментариев стиля C является то, что многострочные запросы в них можно выполнять, даже не раскомментируя.

Отладка T-SQL

Когда редактор SQL обнаруживает ошибку, он отображает ее характер и номер строки в пакете. Дважды щелкнув на ошибке, можно сразу же переместиться к соответствующей строке.

Довольно часто ошибка расположена не в том слове, которое указано в сообщении, — все зависит от того, как разбиралась соответствующая инструкция. Обычно фактически ошибка расположена непосредственно до или после указанного в сообщении места — в любом случае в сообщении место указано достаточно точно.

SQL Server предлагает несколько команд, облегчающих отладку пакетов. В частности, команда `print` отправляет сообщение без генерации результирующего набора данных. Лично я считаю команду `print` особо ценной для отслеживания хода выполнения пакета. Когда анализатор запросов находится в режиме сетки, выполните следующий пакет:

```
Select 3;
Print 6;
```

Результирующий набор данных отобразится в сетке и будет состоять из одной строки. В то же время во вкладке Messages отобразится следующий результат:

```
(1 row(s) affected)
6
```

Иногда полезно приостанавливать выполнение программы, чтобы увидеть блокировки или содержимое объектов. Команда `pause` позволяет приостановить выполнение пакета на заданное время. Например, при выполнении следующего кода его вторая строка вывода отобразится после двухсекундной паузы:

```
Print 'Начало';
waitfor delay '00:00:02';
Print 'Конец';
```

Результат выполнения кода:

```
Начало
Конец
```



Ключевым моментом является то, что в утилиту Management Studio версии SQL Server 2005 не включен отладчик языка T-SQL, — он присутствует в пакете Visual Studio 2005. Если в будущем отладчик будет включен в какой-либо пакет обновлений, я сообщу об этом на сайте `www.SQLServerBible.com`.

Переменные

Любой язык требует для временного хранения значений в памяти наличия переменных. Переменные T-SQL создаются с помощью команды `declare`, за которой следуют имя переменной и ее тип. Используемые для переменных типы данных в точности совпадают с существующими в таблицах. К этому можно добавить табличный тип и тип `SQLVariant`. В одной команде `declare` через запятую может быть перечислено несколько переменных.

Значения по умолчанию и область определения переменных

Область определения переменных (т.е. срок их жизни) распространяется только на текущий пакет. По умолчанию только что созданные переменные содержат пустые значения `null` и до включения в выражения должны быть инициализированы.

В следующем сценарии создаются две тестовые переменные, при этом продемонстрированы их область определения и значения по умолчанию. Весь сценарий является одним исполняемым файлом, хотя с технической точки зрения состоит из двух пакетов (разделенных командой `GO`). Сразу после сценария продемонстрированы три его инструкции `SELECT`:

```
DECLARE @Test INT,
                    @TestTwo NVARCHAR(25);
SELECT @Test, @TestTwo;
SET @Test = 1;
SET @TestTwo = 'значение';
SELECT @Test, @TestTwo ;
Go
SELECT @Test as BatchTwo, @TestTwo;

----------- -------------------------
NULL        NULL
(1 row(s) affected)

Value
----------- ---------------
1           значение
(1 row(s) affected)
Msg 137, Level 15, State 2, Line 2
Must declare the scalar variable "@Test".
```

Первая инструкция SELECT возвращает два пустых значения. После того как переменные были инициализированы, они возвращают присвоенные им значения. После завершения пакета результатом следующей инструкции SELECT является сообщение об ошибке №137.

Эти переменные имеют локальную область определения, которая не распространяется на другие пакеты и хранимые процедуры.

Использование команд SET и SELECT

Команды SET и SELECT могут использоваться для присвоения значений переменным. Основным отличием между ними является то, что команда SELECT может извлекать информацию из источника данных (т.е. таблицы, подзапроса, представления, и т.п.) и включать в себя другие инструкции SELECT, в то время как команда SET ограничена извлечением данных из выражений. Как одна, так и другая команда может содержать функции. Используйте более простую команду SET, когда требуется присвоить переменной результат функции или константу и не требуется рассматривать какой-либо источник данных.

Инструкция SELECT может извлекать значения из множества столбцов. Значение каждого из столбцов может быть присвоено переменной. Если инструкция SELECT извлекает несколько строк, то переменным присваиваются значения столбцов последней из них. Следующая инструкция SELECT извлекает 32 строки, упорядоченные по полю идентификатора личности. В то же время переменные возвращают код и фамилию только последнего человека в списке:

```
USE Family;
Declare @TempID INT,
@TempLastName VARCHAR(25);
SET @TempID = 99;
SELECT
@TempID = PersonID,
@TempLastName = LastName
FROM Person
ORDER BY PersonID;
SELECT @TempID, @TempLastName;
```

Результат выполнения пакета:

```
----------- -------------------------
32 @code last:Campbell
```



В приведенном выше примере присутствует довольно распространенная ошибка. Никогда не используйте инструкцию SELECT для заполнения переменных, если не уверены, что результирующий набор данных будет состоять всего из одной строки. В противном случае вам придется довольствоваться только последней строкой данных.

Если инструкция SELECT не возвращает ни одной строки, то на переменные не оказывается никакого влияния. Следующий запрос не возвращает значений, поскольку записи с идентификатором 100 в таблице Person не существует. По этой причине переменной @TempIDvariable присваивается значение последней существующей строки, при этом переменная фамилии сохраняет изначальное пустое значение:

```
Declare @TempID INT,
        @TempLastName VARCHAR(25);
SET @TempID = 99;
SELECT @TempID = PersonID,
       @TempLastName = LastName
  FROM Person
  WHERE PersonID = 100
  ORDER BY PersonID;
SELECT @TempID, @TempLastName;

----------- -------------------------
99 @code last:NULL
```

Условный отбор

Следующая инструкция SELECT содержит предложение WHERE, и ее синтаксис правильный, хотя для некоторых может выглядеть непривычно:

```
SELECT @переменная = выражение WHERE булево_выражение;
```

В данном случае предложение WHERE функционирует как условный оператор if. Если булево выражение истинно, то переменной присваивается значение, в противном случае инструкция SELECT все равно выполняется, но значение переменной не изменяется.

Использование переменных в запросах SQL

Одним из моих любимых свойств языка T-SQL является то, что переменные могут использоваться в запросах без необходимости создания сложных динамических строк, встраивающих переменные в программный код. Динамический SQL продолжает свое существование, но одиночное значение можно изменить проще — с помощью переменной.

Везде, где в запросе может использоваться выражение, может использоваться и переменная. В следующем примере продемонстрировано использование переменной в предложении WHERE:

```
USE OBXKites;
DECLARE @ProductCode CHAR(10);
SET @Code = '1001';
SELECT ProductName
FROM Product
WHERE Code = @ProductCode;
```

Будет получен следующий результат:

```
Name
--------------------------------------------------
Basic Box Kite 21 inch
```

Переменные с множественным присвоением

Переменные с множественным присвоением — это впечатляющий метод, позволяющий добавлять переменную к самой себе с помощью инструкции SELECT и подзапроса.

В этом разделе будет продемонстрирован ряд примеров из реальной жизни. Так как это несколько необычное использование инструкции SELECT, разрешите привести ее базовую форму:

```
SELECT @переменная = @переменная + d.столбец
FROM (управляемая_таблица) as d;
```

К переменной прибавляется каждая строка управляемой таблицы, преобразуя вертикальный столбец в горизонтальный список.

Этот тип извлечения данных довольно часто используется на практике. Иногда вертикальный список значений лучше преобразовать в разделенный запятыми горизонтальный список, растягивая подзаголовок на несколько дюймов. Короткие горизонтальные списки легче воспринимаются человеком, а также экономят пространство отчета.

В следующем примере создается список дат событий тура Outer Banks Lighthouses, предлагаемого в учебной базе данных Cape HatterasAdventures:

```
USE CHA2;
DECLARE
   @EventDates VARCHAR(1024);
SET @EventDates = '';
  SELECT @EventDates = @EventDates
  + CONVERT(VARCHAR(15), a.d,107 ) + '; '
        FROM (select DateBegin as [d]
                     from Event
                     join Tour
                        on Event.TourID = Tour.TourID
   WHERE Tour.[Name] = 'Outer Banks Lighthouses') as a;

SELECT Left(@EventDates, Len(@EventDates)-1)
AS 'Outer Banks Lighthouses Events';
```

Результат выполнения пакета:

```
Outer Banks Lighthouses Events
------------------------------------------------------
Feb 02, 2001; Jun 06, 2001; Jul 03, 2001; Aug 17, 2001;
Oct 03, 2001; Nov 16, 2001
```

Проблема использования переменных с многочисленными присвоениями заключается в том, что не гарантируется правильный порядок денормализованных данных. Так как этот прием не документирован и рассматривается как трюк, он не приветствуется сообществом пользователей SQL Server. Тем не менее он может оказаться полезным для решения некоторых задач (я предпочитаю его курсору).

Управление выполнением процедур

На первый взгляд кажется, что язык T-SQL практически полностью лишен средств управления потоком команд. Однако несмотря на относительную бедность этих средств, их вполне достаточно. Булевы расширения управления данными `exists`, `in` и `case` с успехом компенсируют ограниченность операторов `if` и `while`.

Оператор `if`

Да, это еще тот оператор `if`, который использовал ваш дедушка. В то же время этот оператор в языке T-SQL еще сильнее ограничен — всего одной инструкцией. Один оператор `if` — одна команда. К тому же его не ограничивают ни `then`, ни `end if`. Общий синтаксис этого оператора следующий:

```
IF условие
   инструкция
```

В приведенном ниже примере условие `if` не выполняется, что предотвращает выполнение следующего за ним оператора.

```
IF 1 = 0
  PRINT 'Первая строка';
PRINT 'Вторая строка';
```

Результат будет следующим:

```
Вторая строка
```



За оператором `if` не следует точка с запятой — если она будет присутствовать, то это будет рассматриваться как ошибка. На самом деле оператор `if` является всего лишь префиксом к следующему, и они оба рассматриваются как одна инструкция.

`Begin...End`

Команду `If`, позволяющую выполнить всего одну инструкцию, нельзя назвать особо полезной. Однако можно создать блок инструкций, ограничив его командами `begin` и `end`, как в следующем примере:

```
IF условие
  Begin;
    несколько инструкций;
  End;
```

Вспоминается, как однажды я в течение нескольких часов безуспешно пытался найти ошибку в хранимой процедуре. Она возникала в любой случае, что бы я ни пытался делать. В конце концов оказалось, что я забыл указать ключевые слова `begin` и `end`, и ошибка генерировалась независимо от выполнения условия. Такую ошибку очень просто допустить, но очень сложно локализовать.

`If Exists()`

Несмотря на то что оператор `if` выглядит ограниченным, его предложение условия может включать в себя мощные функции, подобно предложению `WHERE`. Это выражения `if exists()` и `if...in()`.

Выражение `if exists()` использует в качестве условия наличие какой-либо строки, возвращенной инструкцией `SELECT`. Так как ищутся любые строки, список столбцов в инструкции `SELECT` можно заменить звездочкой. Этот метод работает быстрее, чем проверка условия `@@rowcount>0`, потому что не требуется подсчет общего количества строк. Как только хотя бы одна строка удовлетворяет условию `if exists()`, запрос может продолжать выполнение.

В следующем примере выражение `if exists()` используется для проверки заказов, если хотя бы один из них открыт:

```
USE OBXKITES;
IF EXISTS(SELECT * FROM [ORDER] WHERE Closed = 0)
BEGIN;
Print 'Обработка заказов';
END;
```

На самом деле для данного выражения не существует различий между отбором всех столбцов или только избранных. Однако на практике отбор всех столбцов позволяет функции выбрать наилучший из них (возможно, проиндексированный) и выполниться немного быстрее.

If...Else

Необязательная команда `else` позволяет задать инструкцию, которая будет выполнена в случае, если условие `if` не будет выполнено. Подобно `if`, оператор `else` управляет только непосредственно следующей за ним командой или блоком `begin...end`:

```
IF условие
одна строка или блок begin...end;
ELSE
одна строка или блок begin...end;
```

While

Команда `while` используется для циклического выполнения инструкций, пока некоторое условие является истинным. Подобно команде `if`, ее действие распространяется только на непосредственно следующую за ней инструкцию либо на блок `begin...end`.

Некоторые методы создания циклов отличаются временем выполнения проверки условия. Оператор `while` языка T-SQL работает следующим образом.

1. Команда `while` проверяет выполнение условия. Если оно оказывается истинным, выполняется следующая инструкция или их блок; в противном случае они не выполняются.
2. После завершения выполнения инструкции или их блока управление снова передается оператору `while`.

Следующий короткий сценарий демонстрирует использование оператора `while` для создания цикла:

```
Declare @Temp Int;
Set @Temp = 0;
While @Temp <3;
  Begin;
    Print 'тестируемое условие' + Str(@Temp);
    Set @Temp = @Temp + 1;
  End;
```

Будет получен следующий результат:

```
тестируемое условие       0
тестируемое условие       1
тестируемое условие       2
```

Команды `continue` и `break` расширяют возможности создания циклов. Первая из них позволяет немедленно вернуться к команде `while`. Далее условие проверяется в обычном порядке.

Команда `break` позволяет немедленно выйти из цикла и продолжить выполнение сценария так, будто условие `while` не было выполнено. В следующем примере (не предназначенном для фактического запуска) продемонстрировано использование команды `break`:

```
CREATE PROCEDURE MyLife()
AS
WHILE Not @@Eyes2blurry = 1
  BEGIN
    EXEC Eat
    INSERT INTO Book(Words)
      FROM Brain(Words)
      WHERE Brain.Thoughts
         IN('Make sense', 'Good Code', 'BestPractice')
    IF @StarTrekNextGen = 'On'
        BREAK
  END
```

Goto

Прежде чем ассоциировать команду `Goto` с программами на языке BASIC конца 1970-х годов, напоминающими спагетти, следует сказать, что в данном случае команда `goto` предназначена для перехода к метке только в пределах одного пакета или процедуры и редко используется для чего-то другого, кроме как обработки ошибок.

Сама метка имеет вид идентификатора с помещенным в конце символом двоеточия:

```
Имя_метки:;
```

В следующем примере команда `goto` используется для перехода к метке `errorhandler:`, в обход команды `print`:

```
GOTO ErrorHandler;
Print 'more code';
ErrorHandler:;
Print 'Logging the error';
```

Результат выполнения процедуры:

```
Logging the error
```

Изучение SQL Server программным путем

Одним из главных достоинств SQL Server является его прекрасный интерфейс пользователя, позволяющий проектировать и администрировать базы данных. Утилита Management Studio является отличным инструментом для графического исследования базы данных. С другой стороны, программный код T-SQL позволяет глубже заглянуть в "недра" сервера.

sp_help

Хранимая процедура `sp_help` и ее вариации возвращают информацию о сервере, его объектах, подключениях и многом другом. В своей базовой форме эта процедура перечисляет все доступные объекты текущей базы данных, в то время как другие вариации позволяют более пристально всмотреться в отдельные объекты и подключения.

При добавлении в качестве параметра имени объекта эта процедура возвращает информацию о заданном объекте. Приведем пример:

```
USE OBXKites;
sp_help price;
```

В результате будет возвращено семь наборов данных с информацией о таблице `Price`, в том числе:

- имя, дата создания и владелец;
- столбцы;
- столбцы идентичности;
- столбцы глобальных универсальных идентификаторов;
- место размещения файловой группы;
- индексы;
- ограничения.

Глобальные переменные

В большинстве языков программирования глобальной называется переменная, имеющая большую область определения. В T-SQL все не так. Глобальные переменные можно назвать *системными*. Они доступны только для чтения, и являются своеобразными окнами в состояние системы в текущем подключении и/или пакете.

Глобальные переменные создать нельзя. Существует фиксированный набор глобальных переменных, и все они начинаются с двух символов @@. Все глобальные переменные перечислены в табл. 18.1, однако чаще остальных на практике используются `@@Error`, `@@Identity`, `@@NestLevel` и `@@ServerName`.

Таблица 18.1. Глобальные переменные

Глобальная переменная	Возвращает	Область определения
`@@Connections`	Общее число попыток подключения с момента запуска SQL Server	Сервер
`@@CPU_Busy`	Общее число миллисекунд загрузки процессора с момента запуска SQL Server	Сервер
`@@Cursor_Rows`	Количество строк, возвращенных последним открытым курсором	Подключение
`@@DateFirst`	День, считающийся первым днем недели (1 представляет понедельник, 2 — вторник и т.д.)	Подключение
`@@DBTS`	Текущее значение штампа времени в базе данных	База данных
`@@Error`	Код ошибки последней выполненной инструкции T-SQL	Подключение
`@@Fetch_Status`	Состояние строки из последней команды `cursor fetch`	Подключение
`@@Identity`	Последнее значение идентичности, сгенерированное в текущем подключении	Подключение
`@@Idle`	Общее количество миллисекунд, которые SQL Server простаивал с момента запуска	Сервер
`@@IO_Busy`	Общее количество миллисекунд, потраченных сервером на операции ввода-вывода	Сервер
`@@LangID`	Идентификатор языка, используемый для текущего подключения	Подключение
`@@Language`	Название языка, используемого в текущем подключении	Подключение
`@@Lock_Timeout`	Время ожидания блокировки, используемое в текущем подключении	Подключение
`@@Max_Connections`	Текущее максимальное количество одновременных подключений к SQL Server	Сервер

Окончание табл. 18.1

Глобальная переменная	Возвращает	Область определения
`@@Max_Precision`	Настройка максимальной точности числовых значений	Сервер
`@@Nestlevel`	Текущее число вложенных хранимых процедур	Подключение
`@@Options`	Двоичное представление всех текущих параметров подключения	Подключение
`@@Pack_Received`	Общее количество сетевых пакетов, полученных сервером с момента запуска	Сервер
`@@Pack_Sent`	Общее количество сетевых пакетов, отправленных сервером с момента запуска	Сервер
`@@Packet_Errors`	Общее число ошибок сетевых пакетов, распознанных сервером с момента запуска	Сервер

`@@ProcID`	Идентификатор текущей хранимой процедуры. Может использоваться функцией `SysObject` для получения имени процедуры:
	`SELECT Name`
	`FROM SysObject`
	`WHERE id=@@ProcID`
		Подключение

`@@RemServer`	Имя подключаемого сервера при запуске удаленных хранимых процедур	Подключение
`@@RowCount`	Количество строк, возвращенных последней инструкцией T-SQL	Подключение
`@@ServerName`	Имя текущего сервера	Сервер
`@@ServiceName`	Имя службы SQL Server в системе Windows	Сервер
`@@SPID`	Идентификатор серверного процесса текущего подключения	Подключение
`@@TextSize`	Текущий максимальный размер особо больших данных (типов `text`, `ntext` и `image`)	Сервер
`@@TimeTicks`	Количество миллисекунд в одном такте	Сервер
`@@Total_Errors`	Общее количество дисковых ошибок, зарегистрированных сервером с момента запуска	Сервер
`@@Total_Read`	Общее количество операций чтения с диска с момента запуска сервера	Сервер
`@@Total_Write`	Общее число операций записи на диск с момента запуска сервера	Сервер
`@@TranCount`	Общее число активизированных транзакций в текущем подключении	Подключение
`@@Version`	Редакция, версия и установленный пакет обновлений SQL Server	Сервер

Временные таблицы и табличные переменные

Временные таблицы и табличные переменные играют роль, которая коренным образом отличается от стандартных таблиц. Благодаря своей временной природе эти объекты полезны в качестве средства перемещения данных между другими объектами или как спасательный круг, предназначенный для выполнения исключительно срочной работы.

Локальные временные таблицы

Локальные таблицы создаются точно так же, как и обычные, за исключением того, что их имя должно начинаться с символа решетки (#). Временные таблицы фактически создаются на диске в базе данных tempdb:

```
CREATE TABLE #ProductTemp (
  ProductID INT PRIMARY KEY
  );
```

Срок жизни временных таблиц невелик. Когда завершаются создавшие их пакет или хранимая процедура, временные таблицы удаляются. Если временная таблица создается в ходе интерактивной сессии (например, в окне Query Analyzer), то она выживает только до конца текущей сессии. Естественно, в пакете эти таблицы можно удалить и обычным образом.

Область определения временной таблицы также ограничена — ее может видеть только создавшее ее подключение. Даже если тысяча пользователей создаст временные таблицы с одним и тем же именем, каждый из них будет видеть только свою. Причина этого заключается в том, что физически временные таблицы создаются в базе данных tempdb с именем, состоящим из указанного пользователем, и идентификатора подключения. Имена большинства объектов могут иметь до 128 символов, но временные таблицы ограничены только сто шестнадцатью, поскольку последние 12 символов используются для идентификации подключения. Продемонстрируем это на примере. В предлагаемом сценарии создается временная таблица, а затем ее имя извлекается из таблицы sysObjects:

```
SELECT Name
  FROM TempDB.dbo.SysObjects
  WHERE Name Like '#Pro%';
```

Результат показан в сокращенном виде, так как на самом деле реальное значение имеет длину в 128 символов:

```
Name
------------------------------------------------------------
#ProductTemp_____________________________________00000000002D
```

Несмотря на присваиваемое сервером длинное имя, в запросах используется только его часть, заданная пользователем.

Глобальные временные таблицы

Глобальные временные таблицы аналогичны локальным, но имеют более широкую область определения. К глобальным временным таблицам могут обращаться все пользователи, а срок их жизни продлевается до окончания сессии работы последнего пользователя, обращающегося к ним.

Глобальные временные таблицы имеют имя, начинающееся с двух символов решетки (##). В следующем примере проверяется наличие глобальных временных таблиц, и при получении отрицательного результата таковая создается:

```
IF NOT EXISTS(
  SELECT * FROM Tempdb.dbo.Sysobjects
    WHERE Name = '##TempWork')
  CREATE TABLE ##TempWork(
    PK INT,
    Col1 INT
  );
```

Временные таблицы чаще всего используются для текущей работы. В качестве альтернативы можно создать обычную пользовательскую таблицу в базе данных tempdb. SQL Server очищает и перестраивает эту базу данных при каждом запуске, удаляя и альтернативные таблицы.

Табличные переменные

Табличные переменные аналогичны временным таблицам — на самом деле и те и другие находятся в базе данных tempdb. Главное отличие, кроме синтаксиса, заключается в том, что табличные переменные имеют ту же область определения, что и локальные. Они видимы только в пакете или хранимой процедуре, в которых созданы. Табличные переменные имеют и ряд дополнительных ограничений.

- Они не могут быть созданы с использованием синтаксиса SELECT * INTO или INSERT INTO.
- Табличные переменные не могут быть созданы внутри функций.
- В табличных переменных недопустимы ограничения проверки и внешние ключи. В то же время в них разрешены первичные ключи, значения по умолчанию, пустые значения и ограничения на уникальность.
- Табличные переменные не могут иметь зависимых объектов, таких как триггеры и внешние ключи.

Табличные переменные объявляются как обычные, а не создаются с помощью инструкций SQL DDL. Когда на табличную переменную ссылается запрос, обращение к ней происходит как к таблице, однако именуется она как переменная. Следующий сценарий должен быть выполнен как единый пакет или завершиться ошибкой:

```
DECLARE @WorkTable TABLE (
  PK INT PRIMARY KEY,
  Col1 INT NOT NULL);
INSERT INTO @WorkTable (PK, Col1)
   VALUES ( 1, 101);
SELECT PK, Col1
   FROM @WorkTable;
```

Будет получен следующий результат:

```
PK          Col1
----------- -----------
1           101
```

Динамический SQL

Термин *динамический SQL* имеет два противоречивых определения. Некоторые говорят, что он подразумевает запрос, выполняемый клиентом, а не хранимой процедурой. Точнее бы было сказать, что он описывает любую инструкцию SQL DML, динамически генерируемую в процессе выполнения программы.

Динамический SQL особенно полезен в следующих ситуациях.

- Для сборки предложения WHERE из множества возможных критериев.
- Для сборки предложения FROM, включающего только те таблицы и объединения, которые необходимы предложению WHERE.
- Для создания динамического предложения ORDER BY, по-разному упорядочивающего данные в зависимости от потребностей пользователя.

Выполнение инструкций динамического SQL

Команда execute (или сокращенно exec) создает новый экземпляр пакета, как если бы выполняемый код был хранимой процедурой. Несмотря на то что команда execute обычно используется для вызова хранимых процедур, ее также можно использовать для запуска пакетов и запросов T-SQL:

```
EXEC[UTE] ('пакет_T-SQL)
WITH RECOMPILE;
```

Параметр WITH RECOMPILE заставляет SQL Server перекомпилировать пакет и создать новый план выполнения запроса. Если строка T-SQL и ее параметры подвержены сильным изменениям, то параметр WITH RECOMPILE позволит избежать выполнения ошибочного плана. В то же время, если строка T-SQL не подвержена изменениям, ненужный процесс перекомпиляции только замедлит выполнение запроса. Большинство процедур динамического SQL создают в корне отличающиеся запросы SQL, так что параметр WITH RECOMPILE в данном случае обычно уместен.

Например, следующая команда exec выполняет обычную инструкцию SELECT:

```
USE Family;
EXEC ('Select LastName from Person Where PersonID = 12');
```

Будет получен следующий результат:

```
LastName
---------------
Halloway
```



При работе с командой exec следует учитывать контекст безопасности исполняемого кода. В версии SQL Server 2005 введен синтаксис execute as для явного задания контекста безопасности. Подробно контекст безопасности и синтаксис команды execute as рассматриваются в главе 40.



Новинкой в команде execute является возможность выполнения кода на связанном, а не только на локальном сервере. Следующий код передается на удаленный сервер, а результаты возвращаются на локальный:

```
EXEC[UTE] (код) AT имя_связанного_сервера
```

sp_executeSQL

Новый метод выполнения динамического SQL заключается в использовании системной хранимой процедуры sp_executeSQL. Этот метод больше подходит для сложных запросов, чем обычная команда execute. Я заметил, что в некоторых ситуациях команда exceute не справлялась с выполнением динамических запросов, в то время как процедура sp_executeSQL работала безукоризненно.

```
EXEC Sp_ExecuteSQL
'запрос T-SQL',
определение_параметров,
параметр, параметр...
```

В строке '*запрос T-SQL*' не допускается конкатенация строк, поэтому параметры не указываются. Сам запрос и определения должны использовать таблицу символов Unicode.

Параметры хранимой процедуры предназначены для оптимизации. Если запрос T-SQL при каждом запуске использует одни и те же параметры, то они могут быть переданы хранимой процедуре sp_executeSQL, чтобы сформировать план выполнения запроса. В следую-

щем примере выполняется тот же запрос к таблице Person базы данных Family, но в данном случае использованы параметры (буква N необходима, так как должны использоваться строки в кодировке Unicode):

```
EXEC sp_executeSQL
  N'Select LastName
     From Person
     Where PersonID = @PersonSelect',
  N'@PersonSelect INT',
  @PersonSelect = 12;
```

Результат будет получен тот же:

```
LastName
----------
Halloway
```

Создание динамического кода SQL

Создание динамических строк кода SQL обычно предполагает комбинирование выражения SELECT COLUMNS с более гибкими предложениями FROM и WHERE.

После создания строки запроса он может быть выполнен с помощью все той же команды exec. Предлагаемый пример запроса использует предложения FROM и WHERE, основываясь на требованиях пользователя.

В примере битовая переменная NeedsAnd отслеживает потребности и с помощью связки AND объединяет их в предложении WHERE. Если задана категория товара, то начальная часть инструкции SELECT включает в себя все необходимые объединения таблицы ProductCategory. Предложение WHERE обследует все возможные критерии пользователя. Если пользователь задал для столбца некоторый критерий, то он вместе с этим критерием добавляется в строку @SQLWhere.

В динамических запросах, используемых в реальных практических задачах, иногда содержатся десятки сложных критериев и параметров. В настоящем примере мы используем всего три столбца:

```
USE OBXKites;

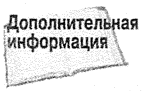
DECLARE
  @SQL NVARCHAR(1024),
  @SQLWhere NVARCHAR(1024),
  @NeedsAnd BIT,
-- Пользовательские параметры
  @ProductName VARCHAR(50),
  @ProductCode VARCHAR(10),
  @ProductCategory VARCHAR(50);
-- Инициализация переменных
SET @NeedsAnd = 0;
SET @SQLWhere = '';
-- Имитация требований пользователя
SET @ProductName = NULL;
SET @ProductCode = 1001;
SET @ProductCategory = NULL;
-- Сборка динамического запроса
-- Настройка начальной части инструкции
IF @ProductCategory IS NULL
  SET @SQL = 'Select ProductName from Product'
ELSE
```

```
  SET @SQL = 'Select ProductName
        from Product
        Join ProductCategory
           on Product.ProductCategoryID
              = ProductCategory.ProductCategoryID';

-- Сборка динамического предложения WHERE
IF @ProductName IS NOT NULL
  BEGIN;
    SET @SQLWhere = 'ProductName = ' + @ProductName;
    SET @NeedsAnd = 1;
  END;
IF @ProductCode IS NOT NULL
  BEGIN;
    IF @NeedsAnd = 1
      SET @SQLWhere = @SQLWhere + ' and ';
    SET @SQLWhere = 'Code = ' + @ProductCode;
    SET @NeedsAnd = 1;
  END;
IF @ProductCategory IS NOT NULL
  BEGIN;
    IF @NeedsAnd = 1
      SET @SQLWhere = @SQLWhere + ' and ';
    SET @SQLWhere = 'ProductCategory = ' + @ProductCategory;
    SET @NeedsAnd = 1;
  END;
-- Окончательная сборка запроса SELECT
IF @SQLWhere <> ''
SET @SQL = @SQL + ' WHERE ' + @SQLWhere
Print @SQL
EXEC sp_executeSQL @SQL
WITH RECOMPILE
```

В результатах показан как текст сформированного динамического запроса, так и полученный в результате его выполнения набор данных:

```
Select Name from Product where Code = 1001;
Name
---------------------------------
Basic Box Kite 21 inch
```



Метод динамического аудита использует сложный запрос SQL в хранимой процедуре. Этот метод мы подробно рассмотрим в главе 24.

Обработка ошибок

Все полноценные языки программирования предлагают какой-либо метод перехвата, протоколирования и обработки ошибок. В этом отношении языку T-SQL было нечем похвастаться, однако теперь прогресс налицо. В целом обработка ошибок работает хорошо (за исключением нескольких замечаний), но существует несколько фатальных ошибок, которые приводят к прекращению выполнения кода, не предоставляя вам никакой возможности их тестирования и обработки.



Команда `Try...Catch` привнесла в SQL Server средства обработки ошибок XXI века. После перевода своих баз данных в среду SQL Server 2005 в первую очередь перестройте обработку ошибок.

Try...Catch

Выражение Try...Catch является стандартным методом перехвата и обработки ошибок, который программисты используют уже долгие годы. Основная идея заключается в том, что выполняется попытка выполнить фрагмент программы, и если при этом возникают какие-либо ошибки, то они обрабатываются в секции Catch. Общая структура выражения такова:

```
BEGIN TRY;
<программа>;
END TRY
BEGIN CATCH;
<программа>;
END CATCH;
```

Компилятор T-SQL трактует комбинацию try...begin catch как единую команду. Как и в любой другой инструкции, наличие терминатора go или точки с запятой между этими двумя командами приведет к необрабатываемой ошибке. Инструкция begin catch должна следовать непосредственно за end try.

Если при выполнении секции try происходит какая-либо ошибка, то управление немедленно предается секции catch. Если секция try выполняется без ошибок, то блок catch вообще не выполняется. Приведем пример:

```
BEGIN TRY;
  SELECT 'Первая попытка';
  RAISERROR('Имитация ошибки', 16, 1);
  Select 'Вторая попытка';
END TRY
BEGIN CATCH
  SELECT 'Секция обработки ошибки';
END CATCH;
SELECT 'Третья попытка';
```

Будет получен следующий результат:

```
-------
Первая попытка
-------
Секция обработки ошибок
-------
Третья попытка
(1 row(s) affected)
```

В этом примере SQL Server выполняет секцию try, пока не встречает функцию raiserror, имитирующую ошибку. После этого управление передается в секцию catch. Следом за блоком catch выполняется следующая по порядку инструкция, выводящая сообщение о третьей попытке.

Если в блоке try происходит ошибка и управление передается в секцию catch, туда же попадает и информация об ошибке. Эту информацию можно извлечь с помощью функций, перечисленных в табл. 18.2. Эти функции были специально созданы для секции catch — вне этого блока они всегда возвращают пустое значение null.

Таблица 18.2. Функции перехвата

Функция	Что возвращает
Error_Message()	Текст сообщения об ошибке
Error_Number()	Номер ошибки

Функция	Что возвращает
`Error_Procedure()`	Имя хранимой процедуры или триггера, в которых возникла ошибка
`Error_Severity()`	Опасность ошибки
`Error_State()`	Состояние ошибки

Функции перехвата извлекают информацию об ошибке, вызвавшую переход в блок `catch`. Они могут вызываться множество раз, но все равно хранят информацию об ошибке. В блоке `catch` следующая инструкция `SELECT` может извлечь полную информацию об ошибке:

```
SELECT
  ERROR_MESSAGE() AS [Message],
  ERROR_PROCEDURE() AS [Procedure],
  ERROR_LINE() AS Line,
  ERROR_NUMBER() AS Number,
  ERROR_SEVERITY() AS Severity,
  ERROR_STATE() AS State;
```

Результат выполнения инструкции:

```
Message           Procedure  Line  Number    Severity   State
----------------- ---------- ----- --------- ---------- ---------
Имитация ошибки   NULL       4     50000     16         1
```

Полученные данные могут впоследствии быть запротоколированы в журнале ошибок.

Старая глобальная переменная @@Error

Исторически сложилось так, что обработка ошибок в T-SQL всегда хромала на обе ноги. Основная информация хранилась в глобальных переменных `@@Error` и `@@rowcount`. Она содержала состояние выполнения предыдущей инструкции; нулевое значение соответствовало отсутствию ошибок.

Сложность заключалась в том, что, в отличие от других языков, информация о последней ошибке не хранилась до тех пор, пока не появлялась следующая. Значение переменной `@@Error` обновлялось после выполнения каждой инструкции, даже после операции тестирования ее значения.

В следующем примере мы попытаемся обновить значение первичного ключа на уже существующее. При этом будут нарушены ограничения внешнего ключа и будет сгенерирована ошибка. Две команды `print` демонстрируют, что значение переменной `@@Error` изменяется с каждой инструкцией. Первая команда `print` отображает информацию об успехе или ошибке обновления; вторая команда `print` отображает успех или ошибку выполнения первой:

```
USE Family;
UPDATE Person
  SET PersonID = 1
  Where PersonID = 2;
Print @@Error;
Print @@Error;
```

Результат выполнения пакета:

```
Server: Msg 547, Level 16, State 1, Line 1
UPDATE statement conflicted with COLUMN REFERENCE constraint
'FK__Marriage__Husband__7B905C75'. The conflict occurred in
database 'Family', table 'Marriage', column 'HusbandID'.
```

```
The statement has been terminated.
547
0
```

Решение проблемы сохранения значения последней ошибки заключается в использовании локальной переменной. Этот метод позволяет сохранить статус ошибки до тех пор, пока она не будет правильно обработана. В следующем примере такой переменной является `@err`:

```
USE Family;
DECLARE @err INT;

UPDATE Person
  SET PersonID = 1
  Where PersonID = 2
SET @err = @@Error;

IF @err <> 0
Begin
-- код обработки ошибки
Print @err
End;
```

В результате на печать будет выведено следующее сообщение:

```
Msg 547, Level 16, State 1, Line 1
UPDATE statement conflicted with COLUMN REFERENCE constraint
'FK__Marriage__Husban__7B905C75'. The conflict occurred in database
'Family', table 'Marriage', column 'HusbandID'.
The statement has been terminated.
547
```

Глобальная переменная @@RowCount

После проверки успеха выполнения запроса нужно найти количество выполненных строк программы, чтобы локализовать ошибку. Даже если ошибка не была сгенерирована, все равно существует вероятность того, что данные не соответствуют и операция не была выполнена. Глобальная переменная `@@RowCount` позволяет проверить эффективность запроса.

Вопросы обновления значений, характерные для переменной `@@Error`, не обходят стороной и переменную `@@RowCount`. В то же время нет никакого смысла хранить нулевое значение.

В следующем пакете переменная `@@RowCount` используется для проверки числа обновленных строк. Отсутствие результата искусственно вызывается некорректным условием предложения `WHERE`. В таблице не существует личности с идентификатором 100, а переменная `@@RowCount` используется для локализации причины ошибки:

```
USE FAMILY;
UPDATE Person
  SET LastName = 'Johnson'
  WHERE PersonID = 100;
IF @@RowCount = 0
Begin
  -- код обработки ошибки
  Print 'Не обработано ни одной строки'
End;
```

Результат выполнения пакета будет следующим:

```
Не обработано ни одной строки
```

Raiserror

Чтобы вернуть произвольное сообщение об ошибке в вызывающую процедуру или клиентское приложение, используют команду `raiserror`. Существуют две формы этой команды: старая упрощенная и рекомендуемая полная.

Простая форма команды Raiserror

Простая форма команды `Raiserror`, существующая со времен Sybase, передает только номер ошибки и сообщение. Уровень опасности всегда устанавливается в 16 — определяемый пользователем:

```
RAISERROR номер_ошибки, сообщение;
```

Например, следующий программный код возвращает простое сообщение об ошибке:

```
RAISERROR 5551212 'Невозможно обновить клиента'
```

Результат выполнения команды:

```
Msg 5551212, Level 16, State 1, Line 1
'Невозможно обновить клиента'
```

Полная форма команды Raiserror

Усовершенствованная форма команды `Raiserror` вобрала в себя четыре новые функции.

- Указывает уровень опасности ошибки.
- Позволяет динамически изменять сообщение.
- Использует хранимые сообщения уровня сервера.
- Позволяет протоколировать ошибки в журнал.

Синтаксис этой команды для Windows следующий.

```
RAISERROR (
 сообщение или номер, опасность, состояние,
 дополнительные_аргументы
) With Log;
```

Опасность ошибки

В системе Windows установлена система стандартных кодов опасности ошибок (табл. 18.3). Коды, не перечисленные в таблице, зарезервированы компанией Microsoft.

Таблица 18.3. Доступные коды опасности

Код опасности	Описание
10	Ошибка не сгенерирована, но возвращается сообщение, например с помощью команды `print`
11–13	Нет специального значения
14	Информационное сообщение
15	Предупреждение: что-то пошло не так
16	Критичная ошибка. Выполнение процедуры прекращено

Добавление в сообщение переменных параметров

Сообщение об ошибке может содержать фиксированную строку или номер ошибки. Для каждого из этих типов существуют дополнительные аргументы, которые подставляются на фиксированные места в сообщении. Доступны различные их типы, но мне кажутся самыми полезными `%s` — для строки и `%i` — для знакового целого числа. Приведу пример, использующий один строковый аргумент:

```
RAISERROR ('Невозможно обновить %s.', 14, 1, 'Customer');
```

Будет получен следующий результат:

```
Msg 50000, Level 14, State 1, Line 1
Невозможно обновить Customer.
```

Хранимые сообщения

Команда `raiserror` системы Windows также может извлечь сообщение из системного представления `sysmessages`. Номера сообщений 1–50000 зарезервированы компанией Microsoft, а более высокие доступны для пользователей. Основным преимуществом использования хранимых сообщений является их последовательность и пронумерованность.

Следует заметить, что в представлении `sysmessages` нумерация распространяется на весь сервер. Если два производителя баз данных используют пересекающиеся сообщения, то между базами данных не существует разделения. Это значит, что предпочтение не отдается ни одному из проектов. Второй вопрос заключается в том, что при переносе базы данных на другой сервер сообщения также следует перемещать.

Таблица `sysmessages` содержит столбцы идентификатора и текста сообщения, уровня опасности, а также признака протоколирования. В то же время уровень опасности команды `raiserror` подставляется вместо значения из таблицы `SysMessage`; таким образом, последнее нивелируется.

Для управления сообщениями в программном коде используется системная хранимая процедура `sp_addmessage`:

```
EXEC sp_addmessage 50001, 16, 'Невозможно обновить %s';
```

Проекты баз данных могут быть развернуты на разных языках, и необязательный параметр `@lang` используется для выбора языка сообщений об ошибках.

Если сообщение уже существует, то в вызов хранимой процедуры можно добавить параметр `replace`:

```
EXEC sp_addmessage 50001, 16,
'Все еще невозможно обновить %s', @Replace = 'Replace';
```

Чтобы просмотреть существующие пользовательские сообщения, следует выполнить запрос к системному представлению `sysmessages`:

```
SELECT *
  FROM sysmessages
  WHERE message_id > 50000;
```

Может быть получен следующий результат:

```
message_id language_id severity is_event_logged text
---------- ----------- -------- ---------------- -----------------
50001      1033        16       0  Все еще невозможно обновить %s
```

Для перемещения сообщений между серверами нужно выполнить одно из следующих действий.

- Сохранить сценарий, изначально использовавшийся для загрузки сообщений.

- Использовать следующий запрос для генерирования сценария, добавляющего сообщения.

```
SELECT 'EXEC sp_addmessage, '
+ Cast(message_id as VARCHAR(7))
+ ', ' + Cast(Severity as VARCHAR(2))
+ ', ''' + [text] + ''';'
FROM sysmessages
WHERE message_id > 50000;
```

Результат будет следующий:

```
------------------------------------------------------------
EXEC sp_addmessage, 50001, 16, 'Все еще невозможно обновить %s';
```

Для удаления сообщения используют системную хранимую процедуру `sp_dropmessage`, которой в качестве аргумента передают номер ошибки:

```
EXEC sp_dropmessage 50001;
```

Протоколирование ошибок

Еще одним преимуществом использования формы Windows команды `raiserror` является возможность протоколирования ошибок в журнал событий Windows NT и в журнал событий SQL Server. Недостатком журнала событий Windows является то, что ведется он отдельно на каждой рабочей станции. Несмотря на то что этот журнал является отличным местом для хранения ошибок подключения к серверу, для ошибок базы данных он явно не подходит.

Существуют два способа задания необходимости протоколирования событий.

- Если хранимые сообщения создаются с помощью задания параметра `@with_log='with_log'` или установки флажка Always log в процессе добавления нового сообщения в Management Studio, ошибки будут протоколироваться.
- Если в команде `raiserror` указан параметр `with log`, то будет протоколироваться только данное сообщение об ошибке.

Например, выполнение следующей команды `raiserror` приведет к записи ошибки невозможности обновления в журнал событий:

```
RAISERROR ('Невозможно обновить %s.', 14, 1, 'Customer')
  WITH LOG
```

Результат будет следующим:

```
Server: Msg 50000, Level 14, State 1, Line 1
Невозможно обновить Customer.
```

Чтобы просмотреть ошибки в журнале событий (рис. 18.1), выберите в меню Пуск системы Windows пункт Панель управления⇨Средства администрирования⇨Журнал событий (Control Panel⇨Administrative Tools⇨Event Viewer). Запустить просмотр журнала событий можно также из меню Все программы (Programs).

Журнал SQL Server

SQL Server поддерживает несколько файлов журналов. При каждом запуске SQL Server создает новый файл журнала. SQL Server постоянно поддерживает семь файлов журналов: шесть архивных и один текущий. Все эти журналы перечислены в окне Object Explorer утилиты Management Studio в узле Management⇨SQL Server Logs. Дважды щелкните на названии журнала, чтобы открыть окно просмотра, показанное на рис. 18.2. В просмотре вам помогут установка фильтра и функция поиска.

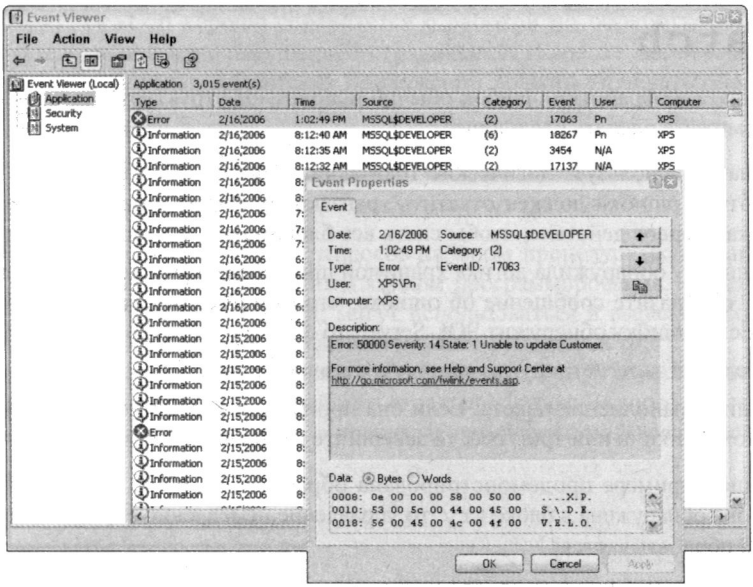


Рис. 18.1. Ошибка SQL Server в окне журнала событий Windows. Обратите внимание на то, что в данные об ошибке включены имена сервера и базы данных

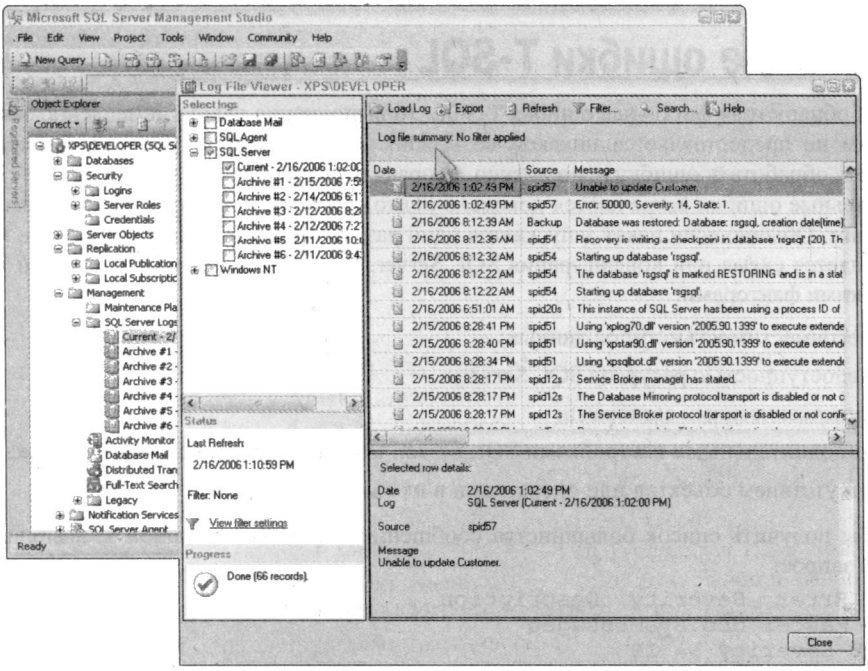


Рис. 18.2. Просмотр ошибок в журнале SQL Server в окне утилиты Management Studio

Блок Catch

Для перехвата и обработки ошибок обычно используют структуру Try...Catch. В блоке Catch вы можете сделать следующее.

1. Если пакет использует логические транзакции (begin tran...commit tran), то обработчик ошибки должен откатить транзакцию. Лично я рекомендую использовать откат как первое действие, чтобы снять все блокировки, установленные транзакцией.
2. Если ошибку обнаружила логика хранимой процедуры и она не является ошибкой SQL Server, отобразите сообщение об ошибке, чтобы информировать конечного пользователя. Если ошибку обнаружит SQL Server, то он сам откроет окно предупреждения.
3. При желании зарегистрируйте ошибку в отдельной таблице.
4. Завершите выполнение пакета. Если она произошла в хранимой процедуре, пользовательской функции или триггере, то завершите их выполнение командой return.

В следующем примере продемонстрирована обработка ошибок. Если ошибка связана с SQL Server и она обнаружена в блоке try, то управление немедленно передается в блок catch, в котором она обрабатывается:

```
Begin Try
-- Код T-SQL
End Try
Begin Catch
-- Код обработки ошибок
End Catch
```

Фатальные ошибки T-SQL

Если обнаружена фатальная ошибка T-SQL, то выполнение пакета немедленно завершается, и вам не предоставляется никакой возможности просмотреть содержимое переменной @@Error, обработать ошибку и, возможно, исправить ситуацию.

Фатальные ошибки встречаются довольно редко, поэтому больших проблем, как правило, не создают. Обычно если пакет запускается один раз, то он продолжит работу, если, конечно, не изменяется схема или конфигурация SQL Server. Чаще всего фатальные ошибки вызваны несколькими факторами:

- несовместимостью типов данных;
- недоступностью ресурсов SQL Server;
- синтаксическими ошибками;
- дополнительными настройками SQL Server, несовместимыми с конкретной задачей;
- отсутствием объектов или опечатками в их именах.

Чтобы получить список большинства сообщений о фатальных ошибках, выполните следующий запрос:

```
SELECT Error, Severity, Description
  FROM Master.dbo.SysMessages
  WHERE Severity >= 19
  ORDER BY Severity, Error
```

Структура `try...catch` хорошо справляется с работой по обработке повседневных ошибок пользователя, например с нарушением ограничений. И все же для обеспечения большей безопасности разработчики клиентских приложений должны включать обработку ошибок в свои программы.

Резюме

Язык T-SQL расширяет возможности запросов набором процедурных команд. Несмотря на то что T-SQL нельзя назвать полноценным языком программирования, он неплохо справляется с работой. Команды пакетов T-SQL могут использоваться в выражениях, быть сконцентрированы в хранимых процедурах, пользовательских функциях и триггерах.

В следующих двух главах мы продолжим изучение языка T-SQL и рассмотрим массовые операции и процессы, основанные на обработке отдельных строк. За ними последуют главы, посвященные организации пакетов T-SQL в хранимые процедуры, пользовательские функции и триггеры.

ГЛАВА

19

Выполнение массовых операций

В этой главе...

Массовая вставка из "плоских" файлов с разделителями

Оператор T-SQL `bulk insert`

Параметры массовой вставки

Утилита BCP

В работе часто требуется быстро выполнить копирование больших объемов информации. Это могут быть данные, наработанные за ночь, или "плоские" файлы с разделителями, которые еще нужно обработать. Если несколько сотен мегабайт информации должны переместиться в базы данных SQL Server за ограниченный промежуток времени, то на помощь придут массовые операции.

Популярность формата XML растет с каждым днем, однако еще быстрее увеличиваются размеры файлов этого формата. Теги XML добавляют в файл данных существенный объем, иногда увеличивая его размер вчетверо. Для очень больших файлов чаще используют текстовый формат с разделителями. Для вставки данных из этого наследия прошлого лучше всего использовать массовые операции.

В SQL Server массовые операции выполняются в обход журнала транзакций, направляя данные непосредственно в базу данных. Несмотря на то что это в значительной мере повышает скорость операции, игнорирование журнала транзакций может усложнить план восстановления данных. Все зависит от используемой модели восстановления.

- **Простая модель восстановления**. В журнал заносятся только текущие транзакции.
- **Модель восстановления с протоколированием массовых операций**. Отдельные транзакции массовых операций идут в обход журнала, однако впоследствии в него заносятся данные операции в целом.
- **Полная модель восстановления**. Массовые операции не заносятся в журнал транзакций, поэтому сам журнал теряет свою пригодность. Чтобы перезапустить возможность процесса восстановления вслед за массовой операцией, выполните полное резервирование данных, после чего перезапустите журнал транзакций.



Более детально о моделях восстановления и их настройке вы узнаете в главе 36. Особенности ведения журнала транзакций будут рассмотрены в главе 51.

С технической точки зрения синтаксис инструкции `SELECT INTO` можно также назвать массовой операцией, поскольку она также обходит стороной журнал транзакций. Она создает новую таблицу из результирующего набора данных инструкции `SELECT` (см. главу 16).

Операции массовой вставки являются одним этапом процесса ETL (эта аббревиатура буквально переводится как "извлечь, преобразовать, загрузить"). Несмотря на то что программирование этих процессов на языке T-SQL допустимо, серьезную конкуренцию ему составила служба интеграции SQL Server (Integration Service). Подробно разработка решений службы интеграции описана в главе 42.



Массовая вставка имеет исключительно высокое быстродействие, и я успешно использую ее в производственных процессах. Однако следует особо отметить, что в данном случае исходные данные должны быть предварительно проверены. Вариации типов данных, а также отсутствующие и нерегулярные столбцы могут вызвать серьезные проблемы.

Массовые операции можно выполнить с помощью утилиты командной строки `BCP`, инструкции `bulk insert` языка T-SQL, а также службы интеграции.

Команда `bulk insert`

Команда `bulk insert` может использоваться в сценариях T-SQL и хранимых процедурах для импорта данных в SQL Server. В параметрах этой команды указываются таблица, получающая данные, путь к исходному файлу и параметры.

Если вы хотите протестировать эту команду, воспользуйтесь файлом `Address.csv`, из которого выполняется загрузка данных в учебную базу данных Adventureworks. Этот файл уже может находиться на вашем жестком диске; также его можно загрузить с сайта MDSN. Его объем — 4 Мбайт, и в нем содержится 19614 строк адресных данных (честно говоря, этого слишком мало для операций ETL).

Следующий пакет поможет выполнить массовую вставку из файла `Address` в таблицу `AWAddress` базы данных Adventureworks:

```
Use Tempdb;

CREATE TABLE AWAddressStaging (
  ID INT,
  Address VARCHAR(500),
  City VARCHAR(500),
  Region VARCHAR(500),
  PostalCode VARCHAR(500),
  GUID VARCHAR(500),
  Updated DATETIME
  );

BULK INSERT AWAddressStaging
 FROM 'C:\Program Files\Microsoft SQL Server\90\Tools\Samples\
       AdventureWorks OLTP\Address.csv'
 WITH (FIRSTROW = 1,ROWTERMINATOR ='\n');
```

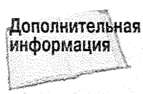
На моем ноутбуке Dell XPS эта операция массовой вставки заняла меньше половины секунды.

Первое, что следует узнать об операции `bulk insert`, — это то, что каждый столбец источника данных вставляется непосредственно в таблицу назначения, используя отображение "один к одному". Первый столбец источника вставляется в первый столбец таблицы приемника, второй — во второй и т.д. Если в таблице приемника слишком много столбцов, то лишние игнорируются. Если же в ней столбцов меньше, чем нужно, то из лишних данных источника получится настоящее месиво.



Так как операция `bulk insert` зависима от расположения столбцов таблиц источника и приемника, то лучше использовать между этими таблицами еще один уровень абстракции в виде представления. Если структуры таблиц источника и приемника не совпадают, скорректируйте представление, и вам не придется вносить глобальные правки в схему базы данных.

Можно также изначально вставить данные в отдельную таблицу, проверить их, затем выполнить остальные преобразования и объединить эту таблицу с таблицей назначения.



Команда `bulk insert` не поддерживает конкатенацию строк и переменных в параметре `from`. По этой причине, если вам нужно присоединить путь к имени файла, сгенерируйте динамическую инструкцию SQL для выполнения массовой вставки. О создании и выполнении динамических инструкций SQL см. в главе 18.

Параметры команды `bulk insert`

В практических задачах мне постоянно приходилось использовать некоторые параметры команды `bulk insert`.

- **FieldTerminator**. В этом параметре определяется символ, разделяющий отдельные столбцы файла источника данных. По умолчанию используется запятая, но я всегда заменял ее вертикальной чертой.
- **RowTerminator**. В этом параметре определяется символ, завершающий строку данных. По умолчанию используется `'\n'` — стандартный символ перевода строки. Однако файлы на мэйнфреймах и в других системах часто используют другие признаки конца строки. В этом случае откройте шестнадцатеричный редактор и посмотрите, какими кодами завершаются строки в файле, а затем определите конец строки в шестнадцатеричных кодах. К примеру, шестнадцатеричное значение `'0A'` программируется следующим образом:

  ```
  ROWTERMINATOR = '0x0A'
  ```

- **FirstRow**. Этот параметр вам пригодится, если исходный файл содержит заголовки столбцов. В этом случае задайте номер той строки, в которой начинаются реальные данные, заносимые в таблицу.
- **TabLock**. Этот параметр позволяет заблокировать всю таблицу для монопольного доступа, что защитит SQL Server от проблем блокирования страниц таблицы, используемых только для вставки. Использование этого параметра может значительно повысить производительность операции, но в то же время не даст возможности в процессе вставки обращаться к таблице другим пользователям. Если массовая вставка является частью процесса ETL, связанного с обособленной таблицей, то это не вызовет проблем. Но если массовая вставка выполняется в рабочую таблицу, из которой пользователи извлекают данные, то использование этого параметра не рекомендуется.
- **Rows per Batch**. Этот параметр указывает серверу выполнять в одном пакете вставку только заданного количества строк, а не всего файла. Настройка размера пакета может

положительно сказаться на производительности. Я рекомендовал бы начать со 100 строк, а затем поэкспериментировать, чтобы найти приемлемое значение.

- **MaxErrors**. Этот параметр определяет, вставка скольких строк должна пройти неудачно, прежде чем будет отменена вся операция. В зависимости от конкретных производственных требований вам может потребоваться установить значение этого параметра в нуль.



Параметр `Errorfile` указывает на файл, в котором будут накапливаться строки, отвергнутые операцией `bulk insert`. В производственных условиях лучше не пренебрегать этим действием.

Остальные параметры мне ни разу не пригодились на практике. Среди них `Check_Constraints`, `CodePage`, `DataFileType`, `Fire_Triggers`, `KeepIdentity`, `KeepNulls`, `Kilobytes_per_batch` и `Order`. Если данные вставляются изначально в обособленную таблицу и только потом объединяются с основной информацией после выполнения соответствующих преобразований, то использование этих параметров вряд ли уместно.



Операция массовой вставки обрабатывает столбцы в том порядке, в котором они находятся в файле с разделителями. Этот порядок должен в точности совпадать с таблицей-приемником. Массовая вставка в представление добавляет дополнительный уровень абстракции, поэтому в данном случае можно будет внести изменения в порядок столбцов позже.

При создании инструкций массовой вставки часто имеет смысл открывать исходный файл с помощью Excel и проверять данные. Сортировка данных в столбцах поможет выявить существующие аномалии форматирования.

Утилита BCP

Утилита BCP (эта аббревиатура расшифровывается как "программа массового копирования") — вариация командной строки операции массового копирования. Она отличается от операции массовой вставки тем, что может как импортировать, так и экспортировать данные. Эта утилита использует многие из параметров операции `bulk insert`. Ее базовый синтаксис следующий:

```
BCP таблица_назначения путь_к_файлу_данных параметры
```

Для таблицы назначения используйте полное четырехкомпонентное имя (*`сервер.база_данных.схема.объект`*). Если хотите ознакомиться с полным синтаксисом этой утилиты, наберите в командной строке **`BCP`**.

Так как эта программа является внешней, она требует авторизации для подключения к серверу. У вас есть два варианта действий: указать в командной строке параметр `-P` и запрограммировать пароль в пакетном файле или не указывать параметр `-P`, и тогда система сама запросит у вас ввод пароля. Лично мне не нравится ни один из этих вариантов.



Для простых операций ETL я предпочитаю использовать операцию T-SQL `bulk insert`. В сложных операциях, содержащих преобразования данных, выигрывает использование службы интеграции. Если говорить честно, то я крайне редко прибегаю к услугам утилиты командной строки `BCP`.

Резюме

Эту главу можно считать дополнением к предыдущей — в ней рассматривалась специфическая команда T-SQL. Массовые операции являются той рабочей лошадкой, которая позволяет импортировать громадные массивы данных, игнорируя журнал транзакций и направляя данные непосредственно в таблицы. Единственным их недостатком является то, что они усложняют план восстановления данных. Лучше всего выполнять операции массовой вставки с помощью команды `bulk insert` языка T-SQL или проектов службы интеграции.

В следующей главе мы рассмотрим еще одну специфическую функцию T-SQL, и это одна из моих любимых тем. Присоединяйтесь ко мне и откажитесь от использования курсоров!

Курсор

ГЛАВА

20

В этой главе...

Отбор данных с помощью курсора

Стратегическое использование или игнорирование курсора

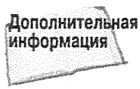
Переход от медленных курсоров к высокопроизводительным пакетным решениям

Язык SQL отлично справляется с обработкой наборов строк. Однако SQL вырос из коротких штанишек старых структур файлов ISAM, и построчная обработка данных осталась только в форме бесконечно медленного курсора.

Несмотря на то что существуют вполне оправданные причины использования курсоров, основной из них все же является то, что программисты со старым процедурным мышлением чувствуют себя с ними более комфортно, чем с реляционной алгеброй, основанной на множествах.

Курсоры SQL создают обманчивое впечатление легко настраиваемых. Когда программисты видят длинный список их параметров, они думают, что с их помощью можно добиться высокой производительности курсоров. Да и типы курсоров имеют такие привлекательные названия, как *курсор прямого доступа, динамический* и *ключевой*. Приведу цитату из MSDN: "СУБД Microsoft SQL Server 2000 реализовала курсор с оптимизированной производительностью, назвав его курсором прямого доступа". Вопрос 70-229 экзамена по проектированию баз данных SQL Server 2000 даже содержал вопрос, какой из курсоров можно образно назвать "пожарным". Не верьте всему этому. Курсоры не могут быть быстрыми в силу своей природы — они за один раз подбирают одну "крошку" данных.

Второй уровень теории оптимизации (см. главу 6) рассматривает вопросы создания высокопроизводительных систем. Он предполагает создание пакетных, а не итеративных решений. Несмотря на то что в этой главе мы рассмотрим именно итеративный проход по данным с помощью курсора, вы должны из нее вынести стратегический вывод, что нужно максимально ограничить использование курсоров, заменяя их пакетами запросов.



Курсоры SQL Server являются серверными, что отличает их от клиентских курсоров ADO. Эти курсоры размещаются в сервере перед тем, как какие-либо данные отправляются клиенту. Клиентские курсоры часто используются для прокрутки строк в наборах данных ADO при заполнении пользовательских форм. О курсорах ADO речь пойдет в главе 30.

Все о курсорах

По своей сути курсор является указателем на одну строку данных. Для прохождения по набору данных до его конца обычно используют цикл `while`. SQL Server поддерживает стандартный синтаксис ANSI SQL-92 и расширенный синтаксис T-SQL, предлагающий дополнительные возможности.

Пять этапов жизни курсора

Курсор создает результирующий набор данных на основе инструкции `SELECT`, а затем проходит по нему построчно. Пятью этапами жизни курсора являются следующие.

1. Объявление курсора определяет тип и режим его работы, а также описывает инструкцию `SELECT`, поставляющую ему данные. При объявлении курсора никакие данные не извлекаются. Это единственный случай, когда инструкция `declare` не требует амперсанда. Курсор SQL-92 объявляется с помощью инструкции `cursor for`:

   ```
   DECLARE имя_курсора CURSOR
     FOR инструкция_SELECT
     FOR параметры_курсора
   ```

   Расширенный курсор T-SQL объявляется аналогичным образом:

   ```
   DECLARE имя_курсора CURSOR параметры_курсора
   FOR инструкция_SELECT
   ```

2. При открытии курсора извлекаются данные, которыми он и заполняется:

   ```
   OPEN имя_курсора
   ```

3. Курсор перемещается к следующей строке и заполняет значениями ее столбцов локальные переменные (эти переменные должны быть предварительно объявлены):

   ```
   FETCH [направление] имя_курсора INTO @переменная1, @переменная2
   ```

   По умолчанию команда `FETCH` перемещает курсор к следующей строке (направление `NEXT`), однако при желании можно переместить курсор к предыдущей (`PRIOR`), первой (`FIRST`) и последней (`LAST`) строке. Также с помощью этой команды можно переместить курсор к строке с некоторым абсолютным номером (`ABSOLUTE n`) или сместить относительно текущей позиции на определенное расстояние (`RELATIVE n`). Проблема последнего подхода состоит в том, что в реляционной базе данных номер строки не имеет определенного смысла. Если в коде требуется перейти к конкретной строке, чтобы получить логический результат, то это должно быть заранее предусмотрено в модели базы данных.

4. Закрытие курсора снимает блокировку данных, но сохраняет инструкцию `SELECT`. Курсор впоследствии может быть открыт в той же точке:

   ```
   CLOSE имя_курсора
   ```

5. Демонтаж курсора высвобождает отведенную под него память и аннулирует его определение:

   ```
   DEALLOCATE имя_курсора
   ```

Существует пять основных команд, необходимых для обслуживания курсора. Добавьте к этому метод итеративного, циклического перемещения курсора, и можете считать задачу выполненной.

Управление курсором

Так как курсор проходит по набору данных построчно, код T-SQL требует команд постоянного смещения курсора. Для управления этим циклическим процессом язык T-SQL предлагает две глобальные переменные, содержащие информацию о состоянии курсора.

Переменная `@@cursor_rows` возвращает общее количество строк в курсоре. Если курсор заполняется асинхронно, то в этой переменной будет содержаться отрицательное значение.

Существенной для управления курсором является глобальная переменная `@@fetch_status`. Она отчитывается о состоянии курсора после последней команды `FETCH`. Это состояние важно для управления перемещением курсора и оценки, достиг ли он одного из концов набора данных. Возможные значения переменной `@@fetch_status` свидетельствуют о следующем.

- `0` — последняя операция `FETCH` успешно извлекла строку.
- `1` — последняя операция `FETCH` достигла конца набора данных.
- `2` — строка, к которой переместился курсор, оказалась недоступной; она была удалена.

Комбинирование переменной `@@fetch_status` с оператором `while` позволяет создать циклы, позволяющие успешно перемещаться по строкам набора данных.

Обычно в пакетах создается курсор, после чего выполняется первая команда `FETCH` и начинается цикл `while`, который продолжается до тех пор, пока не будет достигнут конец набора данных. В верхней части цикла проверяется значение переменной `@@fetch_status` для определения, создан ли курсор. В следующем примере продемонстрированы все пять этапов жизни курсора и управление итеративным процессом его перемещения с помощью цикла `while` и переменной `@@fetch_status`:

```
-- Этап 1
DECLARE cDetail CURSOR FAST_FORWARD
  FOR SELECT DetailID
  FROM Detail
  WHERE AdjAmount IS NULL
-- Этап 2
OPEN cDetail
-- Этап 3
FETCH cDetail INTO @cDetailID
  EXEC CalcAdjAmount
    @DetailID = @cDetailID,
    @AdjustedAmount = @SprocResult OUTPUT
  UPDATE Detail
    SET AdjAmount = @SprocResult
    WHERE DetailID = @cDetailID
  WHILE @@Fetch_Status = <>1
    BEGIN
      BEGIN
        EXEC CalcAdjAmount
          @DetailID = @cDetailID,
          @AdjustedAmount = @SprocResult OUTPUT
        UPDATE Detail
          SET AdjAmount = @SprocResult
          WHERE DetailID = @cDetailID
       END
-- Итеративный проход по курсору
     FETCH cDetail INTO @cDetailID -- fetch next
  END
```

```
-- Этап 4
CLOSE cDetail
-- Этап 5
DEALLOCATE cDetail
```

Обновление курсора

Поскольку курсор последовательно проходит по набору данных, SQL Server знает, какая строка является текущей. Указатель курсора может использоваться в предложениях WHERE инструкций SQL DML (SELECT, INSERT, UPDATE и DELETE), чтобы манипулировать корректными данными.

Параметр FOR UPDATE инструкции DECLARE позволяет выполнять обновления с помощью курсора. При этом если явно указаны какие-либо столбцы, именно они будут обновлены; если никаких столбцов не указано, будут обновлены все.

```
DECLARE cDetail CURSOR
  FOR SELECT DetailID
    FROM Detail
  WHERE AdjAmount IS NULL
  FOR UPDATE OF AdjAmount
```

В теле цикла, после того, как было выполнено перемещение к нужной строке, инструкция DML может включить курсор в предложение WHERE, используя синтаксис CURRENT OF. В следующем примере, взятом из сценария KilltheCursor.sql, выполняется ссылка на курсор cDetail:

```
UPDATE Detail
  SET AdjAmount = @SprocResult
  WHERE CURRENT OF cDetail
```

Область определения курсора

Так как курсоры имеют тенденцию использоваться в самых запутанных ситуациях, понимание области их определения исключительно важно. Эта область определяет, виден ли курсор только в пакете, в котором создан или может использоваться во всех вызываемых процедурах. Область определения курсора задается при его объявлении:

```
DECLARE имя_курсора CURSOR Local или Global
  FOR инструкция_SELECT
```

По умолчанию областью определения курсора является вся база данных. Это указывается в параметре CURSOR_DEFAULT:

```
ALTER DATABASE Family SET CURSOR_DEFAULT LOCAL
```

Для успешного выполнения процедур важно знать текущую область определения курсора. Это выполняется с помощью функции DATABASEPROPERTYEX():

```
SELECT DATABASEPROPERTYEX('Family', 'IsLocalCursorsDefault')
```

Результат выполнения функции следующий:

```
1
```



Кроме параметров global и for update курсоры имеют еще несколько, управляющих возможностью перемещения и обновления данных, — static, keyset, dynamic и optimistic. Мне не хочется тратить место в книге на описание этих параметров, так как лучшей стратегией все-таки остается максимальное ограничение сферы действия курсоров и переход на пакетные решения.

Курсоры и транзакции

По сравнению с пакетными решениями курсоры практически не влияют на блокировки. Это можно аргументировать тем, что пакетное решение блокирует таблицу в целом. В то же время курсор блокирует только конкретную строку, с которой работает. По этой причине, несмотря на то что работа курсора с таблицей в миллион строк будет продолжаться в пятнадцать раз дольше, по крайней мере, он не будет блокировать другие транзакции. Вы сами можете решить, вступать ли в дебаты по этому вопросу.

Компромисс заключается в использовании функции `RowNumber()` с пакетным решением для обновления больших групп строк.

Одной из методик, которая часто используется для повышения производительности курсоров, является объединение всего курсора в одну логическую транзакцию. В этом решении есть свои за и против. Несмотря на то что производительность повышается примерно на 50%, за это приходится расплачиваться блокировкой, устанавливаемой транзакцией.

Стратегии курсора

Ключевым моментом работы с курсорами является понимание того, когда использовать их, а когда искать решение с помощью пакетов. Если выполнить поиск совета на портале Google, он ответит примерно следующее: "Не используйте курсор, если хотя бы две недели безуспешно не поискали пакетное решение у знающих друзей и в группах новостей. Если по крайней мере три оппонента бросили от отчаяния поиск решения с помощью запросов, то можете приступать к написанию курсора".

В общем случае существует всего пять специфических ситуаций, которые решаются с помощью курсора. Ниже предложены стратегии, рекомендуемые в этих случаях.

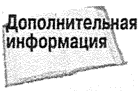
- **Сложная логика**, использующая различные формулы и имеющая свои исключения, довольно редко находит пакетные реализации. В этом случае, как правило, используют одно из двух следующих решений. Чаще всего эту логику вкладывают в циклическую работу с курсором. Еще одно решение заключается в создании хранимой процедуры, которая принимает идентификатор строки, обрабатывает ее, а затем возвращает вычисленное решение или сама обновляет значения.

  В данной ситуации рекомендуется вложить логику в управляемый данными запрос, использующий выражение `CASE`. В следующем разделе будет продемонстрирована эта методика работы.

- **Динамические программные итерации**, такие как генератор инструкций DDL. Для данного случая я не вижу лучшего решения, нежели курсор.

- **Денормализация списка**, которая предполагает преобразование вертикального списка значений в разделенную запятыми горизонтальную строку. Часто данные более понятны в виде строки, чем подотчета или подчиненного списка в несколько дюймов длиной. К тому же такой подход экономит пространство в отчете. Несмотря на то что чаще всего для решения такой задачи используют курсор, можно найти и пакетное решение (оно будет предложено далее в этой главе).

- **Создание перекрестного запроса**. Это одна из задач, которые традиционно в SQL Server относили к разряду особо сложных. Создание такого запроса требует использования целого набора выражений `CASE`. Для создания динамического перекрестного запроса рекомендуется использовать курсор.



В главе 15 продемонстрирован ряд методов создания перекрестных запросов. Среди этих методов использование традиционных выражений `CASE`, метод курсора и использование нового в SQL Server 2005 ключевого слова `pivot`.

- **Навигация по иерархическому дереву**. Эта задача может быть решена с помощью ряда пакетных методов, хотя программисты с процедурным подходом обычно рекурсивно проверяют каждый узел.



О запросах CTE и прочих высокопроизводительных решениях задачи обхода иерархического дерева см. в главе 12.

Сложные логические решения

Считается, что от курсора, используемого для решения сложных логических задач, избавиться труднее всего. Сложности возникают, когда в принятие решения вовлечено множество формул, различные величины и многочисленные исключения. В этом разделе мы начнем с рассмотрения решений таких задач с помощью курсора, а затем переработаем его в различные пакетные альтернативы.



Сценарий `Ch20KillTheCursor.sql` содержит инструкции DDL, предназначенные для создания учебной базы данных и ее таблиц. Этот сценарий генерирует случайные данные любой величины, а затем тестирует на них все методы, предлагаемые в настоящей главе, на предмет производительности. Текущую версию этого файла можно загрузить с сайта `www.SQLServerBible.com`.

В целом будет предложено семь решений, каждое из которых разрешает предложенную сложную логическую задачу. В конце изложения каждого из решений выполняется тест производительности относительно прогрессивно растущего набора данных.

Пример сложной логической задачи

Представим себе ситуацию оформления заказа, использующую множество формул и исключений. Ниже изложены бизнес-правила и исключения предлагаемого примера.

Переменные формулы:

- 1 — **обычная ситуация**. БазоваяСтавка*Количество*БазовыйМножительДействия;
- 2 — **прогрессивная ставка**. БазоваяСтавка*Количество*ПеременныйПрогрессивныйКоэффициент;
- 3 — **прототип**. Количество*БазовыйМножительДействия.

Исключения:

- если по заказу существует Executive OverRide, то игнорировать БазовыйМножительДействия;
- если транзакция выполняется в выходные, то умножить сумму на 2,5;
- постоянные клиенты получают скидку в 20% на базовую ставку;
- базовая ставка равна нулю, если выполняется благотворительный заказ.

Проанализируем, что мы имеем: три формулы и четыре исключения. Обычно этого достаточно, чтобы приступить к написанию курсора... но стоит ли?

Программирование логики

Базовая логика курсора рассматриваемого примера внедрена в хранимую процедуру и пользовательскую скалярную функцию. Из соображений экономии места в этом разделе будет описана только хранимая процедура, однако и в скалярной функции используется та же логика. Обе принимают в качестве аргумента идентификатор `DetailID` и вычисляют общую сумму.

Так как хранимая процедура оперирует одновременно только с одной строкой заказа, она заставляет нас использовать курсор, чтобы программный код выполнялся для каждой строки заказа.

В этой процедуре для обработки различных формул используются операторы `if`. Еще один набор операторов `if` обслуживает исключения. Несмотря на то что эта процедура корректно справляется с описанной задачей, она содержит заложенные в код значения для множителей исключений:

```
CREATE PROC CalcAdjAmount (
  @DetailID INT,
  @AdjustedAmount NUMERIC(7,2) OUTPUT
  )
AS
SET NoCount ON
-- Получение идентификатора
DECLARE
  @Formula SMALLINT,
  @AccRate NUMERIC (7,4),
  @IgnoreBaseMultiplier BIT,
  @TransDate INT,
  @ClientTypeID INT
SELECT @Formula = Formula
  FROM Detail
  JOIN ActionCode
    ON Detail.ActionCode = ActionCode.ActionCode
  WHERE DetailID = @DetailID
SET @IgnoreBaseMultiplier = 0
SELECT @IgnoreBaseMultiplier = ExecOverRide
  FROM [Order]
  JOIN Detail
    ON [Order].OrderID = Detail.OrderID
  WHERE DetailID = @DetailID
-- Обычная формула
IF @Formula = 1
BEGIN
  IF @IgnoreBaseMultiplier = 1
    SELECT @AdjustedAmount = BaseRate * Amount
      FROM Detail
      JOIN ActionCode
        ON Detail.ActionCode = ActionCode.ActionCode
      WHERE DetailID = @DetailID
  ELSE
    SELECT @AdjustedAmount = BaseRate * Amount * BaseMultiplier
      FROM Detail
      JOIN ActionCode
        ON Detail.ActionCode = ActionCode.ActionCode
      WHERE DetailID = @DetailID
  END
-- 2-Прогрессивная ставка: BaseRate * Amount * Acceleration Rate
  IF @Formula = 2
  BEGIN
```

```
    SELECT @AccRate = Value
      FROM dbo.Variable
      WHERE Name = 'AccRate'
    SELECT @AdjustedAmount = BaseRate * Amount * @AccRate
      FROM Detail
      JOIN ActionCode
        ON Detail.ActionCode = ActionCode.ActionCode
      WHERE DetailID = @DetailID
  END
-- 3-Прототип: Amount * ActionCode's BaseMultiplier
  IF @Formula = 3
  BEGIN
    IF @IgnoreBaseMultiplier = 1
      SELECT @AdjustedAmount = Amount
        FROM Detail
        JOIN ActionCode
          ON Detail.ActionCode = ActionCode.ActionCode
        WHERE DetailID = @DetailID
    ELSE
      SELECT @AdjustedAmount = Amount * BaseMultiplier
        FROM Detail
        JOIN ActionCode
          ON Detail.ActionCode = ActionCode.ActionCode
        WHERE DetailID = @DetailID
 END
-- Исключение: Надбавка за выходные дни
SELECT @TransDate = DatePart(dw,TransDate),
  @ClientTypeID =      ClientTypeID
  FROM [Order]
  JOIN Detail
    ON [Order].OrderID = Detail.OrderID
  JOIN Client
    ON Client.ClientID = [Order].OrderID
  WHERE DetailID = @DetailID
IF @TransDate = 1 OR @TransDate = 7
  SET @AdjustedAmount = @AdjustedAmount * 2.5
-- Исключение: Скидка постоянным клиентам
IF @ClientTypeID = 1
  SET @AdjustedAmount = @AdjustedAmount * .8
IF @ClientTypeID = 2
  SET @AdjustedAmount = 0
RETURN
```

Курсор SQL-92 с хранимой процедурой

Исходное решение использует традиционный метод — последовательный проход по всем строкам с вызовом для каждой из них хранимой процедуры и обновлением значений. Именно такой метод программирования должен был заменить SQL:

```
-- Этап 1
DECLARE cDetail CURSOR
  FOR SELECT DetailID
        FROM Detail
        WHERE AdjAmount IS NULL
  FOR READ ONLY
-- Этап 2
OPEN cDetail
```

```
— Этап 3
FETCH cDetail INTO @cDetailID -- prime the cursor
  EXEC CalcAdjAmount
    @DetailID = @cDetailID,
    @AdjustedAmount = @SprocResult OUTPUT
  UPDATE Detail
    SET AdjAmount = @SprocResult
    WHERE DetailID = @cDetailID
  WHILE @@Fetch_Status = 0
    BEGIN
      BEGIN
        EXEC CalcAdjAmount
          @DetailID = @cDetailID,
          @AdjustedAmount = @SprocResult OUTPUT
        UPDATE Detail
          SET AdjAmount = @SprocResult
          WHERE DetailID = @cDetailID
      END
      FETCH cDetail INTO @cDetailID -- fetch next
    END
-- Этап 4
CLOSE cDetail
-- Этап 5
DEALLOCATE cDetail
```

Курсор прямого доступа с хранимой процедурой

Второе итеративное решение использует так называемый "высокопроизводительный" курсор T-SQL. Во всем остальном это решение совпадает с предложенным в предыдущем разделе:

```
-- Этап 1
DECLARE cDetail CURSOR FAST_FORWARD READ_ONLY
  FOR SELECT DetailID
        FROM Detail
  WHERE AdjAmount IS NULL
 -- Этап 2
OPEN cDetail
-- Этап 3
FETCH cDetail INTO @cDetailID - подготовка курсора
  EXEC CalcAdjAmount
    @DetailID = @cDetailID,
    @AdjustedAmount = @SprocResult OUTPUT
  UPDATE Detail
    SET AdjAmount = @SprocResult
    WHERE DetailID = @cDetailID
  WHILE @@Fetch_Status = 0
    BEGIN
      BEGIN
        EXEC CalcAdjAmount
          @DetailID = @cDetailID,
          @AdjustedAmount = @SprocResult OUTPUT
        UPDATE Detail
          SET AdjAmount = @SprocResult
          WHERE DetailID = @cDetailID
      END
      -- Этап 3
      FETCH cDetail INTO @cDetailID - переход к следующей строке
```

```
    END
-- Этап 4
CLOSE cDetail
-- Этап 5
DEALLOCATE cDetail
```

Курсор прямого действия и пользовательская функция

Это решение использует быстрый курсор прямого действия для прохождения по данным, вызывая в каждой строке пользовательскую функцию, результат которой используется для обновления данных:

```
-- Этап 1
DECLARE cDetail CURSOR FAST_FORWARD READ_ONLY
  FOR SELECT DetailID
        FROM Detail
        WHERE AdjAmount IS NULL
 -- Этап 2
OPEN cDetail
-- Этап 3
FETCH cDetail INTO @cDetailID - подготовка курсора
  UPDATE Detail
    SET AdjAmount = dbo.fCalcAdjAmount(@cDetailID)
    WHERE DetailID = @cDetailID
  WHILE @@Fetch_Status = 0
    BEGIN
      UPDATE Detail
        SET AdjAmount = dbo.fCalcAdjAmount(@cDetailID)
        WHERE DetailID = @cDetailID
    END
    -- Этап 3
    FETCH cDetail INTO @cDetailID - переход к следующей строке
-- Этап 4
CLOSE cDetail
-- Этап 5
DEALLOCATE cDetail
```

Курсор обновления с хранимой процедурой

Решение, использующее курсор обновления, реализует ту же логику, что и предыдущее. Основное отличие состоит в том, что сам курсор используется для выбора корректной строки для инструкции UPDATE. Этот курсор также вызывает хранимую процедуру для каждой строки:

```
-- Этап 1
DECLARE cDetail CURSOR FAST_FORWARD READ_ONLY
  FOR SELECT DetailID
        FROM Detail
  WHERE AdjAmount IS NULL
 FOR Update of AdjAmount
-- Этап 2
OPEN cDetail
-- Этап 3
FETCH cDetail INTO @cDetailID - подготовка курсора
  EXEC CalcAdjAmount
```

```
    @DetailID = @cDetailID,
    @AdjustedAmount = @SprocResult OUTPUT
  UPDATE Detail
    SET AdjAmount = @SprocResult
    WHERE Current of cDetail
  WHILE @@Fetch_Status = 0
    BEGIN
      BEGIN
        EXEC CalcAdjAmount
          @DetailID = @cDetailID,
          @AdjustedAmount = @SprocResult OUTPUT
        UPDATE Detail
          SET AdjAmount = @SprocResult
          WHERE Current of cDetail
      END
      FETCH cDetail INTO @cDetailID - переход к следующей строке
    END
-- Этап 4
CLOSE cDetail
-- Этап 5
DEALLOCATE cDetail
```

Запрос обновления с пользовательской функцией

Решение, использующее запрос обновления, — крайне простое, но может на первый взгляд показаться пугающим. В этом решении вся логика скрыта в пользовательской функции. Несмотря на то что эта функция все равно вызывается для каждой строки, внедрение функции в инструкцию UPDATE имеет свои преимущества. Исследование плана выполнения запроса показало, что оптимизатор внедрил логику функции в план запроса и сгенерировал великолепное пакетное решение:

```
UPDATE dbo.Detail
  SET AdjAmount = dbo.fCalcAdjAmount(DetailID)
  WHERE AdjAmount IS NULL
```

Использование множества запросов

Шестое по счету решение использует отдельные запросы для каждой формулы и исключения. Предложение WHERE в запросе отсекает только те строки, которые соответствуют требованиям формулы или исключения.

В этом решении присутствует компонент, управляемый данными. Прогрессивный коэффициент извлекается из таблицы Variable с помощью скалярного подзапроса, а исключения обрабатываются с помощью управляемых данными объединений с таблицами ClientType и DayOfWeekMultiplyer:

```
UPDATE dbo.Detail
  SET AdjAmount = BaseRate * Amount
  FROM Detail
    JOIN ActionCode
      ON Detail.ActionCode = ActionCode.ActionCode
    JOIN [Order]
      ON [Order].OrderID = Detail.OrderID
  WHERE (Formula = 1 OR Formula = 3 )AND ExecOverRide = 1
    AND AdjAmount IS NULL
```

```
UPDATE dbo.Detail
  SET AdjAmount = BaseRate * Amount * BaseMultiplier
  FROM Detail
    JOIN ActionCode
      ON Detail.ActionCode = ActionCode.ActionCode
    JOIN [Order]
      ON [Order].OrderID = Detail.OrderID
    WHERE Formula = 1 AND ExecOverRide = 0
       AND AdjAmount IS NULL

-- 2-Accelerated BaseRate * Amount * Acceleration Rate
UPDATE dbo.Detail
  SET AdjAmount = BaseRate * Amount * (SELECT Value
                              FROM dbo.Variable
                              WHERE Name = 'AccRate')
    FROM Detail
      JOIN ActionCode
        ON Detail.ActionCode = ActionCode.ActionCode
      JOIN [Order]
        ON [Order].OrderID = Detail.OrderID
    WHERE Formula = 2
      AND AdjAmount IS NULL

-- 3-Prototype Amount * ActionCode's BaseMultiplier
UPDATE dbo.Detail
  SET AdjAmount = Amount * BaseMultiplier
    FROM Detail
     JOIN ActionCode
      ON Detail.ActionCode = ActionCode.ActionCode
     JOIN [Order]
      ON [Order].OrderID = Detail.OrderID
    WHERE Formula = 3 AND ExecOverRide = 0
      AND AdjAmount IS NULL

-- Исключения
-- Корректировка выходных дней
UPDATE dbo.Detail
  SET AdjAmount = AdjAmount * Multiplier
  FROM Detail
    JOIN [Order]
      ON [Order].OrderID = Detail.OrderID
    JOIN DayOfWeekMultiplier DWM
      ON CAST(DatePart(dw,[Order].TransDate) as SMALLINT) =
DWM.DayOfWeek

-- Корректировка по клиентам
UPDATE dbo.Detail
  SET AdjAmount = AdjAmount * Multiplier
  FROM Detail
    JOIN [Order]
      ON [Order].OrderID = Detail.OrderID
    JOIN Client
      ON [Order].ClientID = Client.ClientID
    Join ClientType
      ON Client.ClientTypeID = ClientType.ClientTypeID
```

Запросы с выражением CASE

Последнее решение использует выражение CASE и управляемые данными значения для разрешения сложности с помощью всего одного запроса. Главенствующая роль выражения CASE сводится к тому, что оно способно внедрить гибкую логику в один запрос.

Управляемые данными значения и формулы также внедрены в запрос с помощью объединений с корректными значениями таблиц классификаторов. Конструкции, управляемые данными, также снижают затраты на эксплуатацию систем, поскольку в них можно изменить значения без внесения правок в программный код.

В данном примере выражение CASE выбирает корректную формулу, основываясь на значениях таблицы ActionCode. Использование этих значений в примере жестко прошито в инструкции, однако при желании этот процесс можно сделать также управляемым данными.

Как и в решении с множеством запросов, прогрессивный коэффициент и исключения управляются данными:

```
UPDATE dbo.Detail
SET AdjAmount = DWM.Multiplier * ClientType.Multiplier *
  CASE
    WHEN ActionCode.Formula = 1 AND ExecOverRide = 0
      THEN BaseRate * Amount * BaseMultiplier
    WHEN (ActionCode.Formula = 1 OR ActionCode.Formula = 3 )
          AND ExecOverRide = 1
      THEN BaseRate * Amount
    WHEN ActionCode.Formula = 2
      THEN BaseRate * Amount * (SELECT Value
                                FROM dbo.Variable
                                WHERE Name = 'AccRate')
    WHEN (Formula = 3 AND ExecOverRide = 0)
      THEN Amount * BaseMultiplier
  END
FROM Detail
  JOIN ActionCode
    ON Detail.ActionCode = ActionCode.ActionCode
  JOIN [Order]
    ON [Order].OrderID = Detail.OrderID
  JOIN Client
    ON [Order].ClientID = Client.ClientID
  Join ClientType
    ON Client.ClientTypeID = ClientType.ClientTypeID
  JOIN DayOfWeekMultiplier DWM
    ON CAST(DatePart(dw,[Order].TransDate) as SMALLINT) = DWM.DayOfWeek
WHERE AdjAmount IS NULL
```

Анализ производительности

Для тестирования производительности предложенных семи методов с помощью сценария KilltheCursor.sql таблицы были заполнены данными, и каждый метод был выполнен три раза. Запуск десяти итераций позволил оценить производительность и масштабируемость решений (рис. 20.1).

Самую низкую производительность показало решение, использующее курсор обновления; ненамного его опередили остальные три решения. использующие курсоры. Кривые их графиков резко устремляются вверх. Это значит, что время выполнения растет быстрее, чем объем данных, что означает плохую масштабируемость.

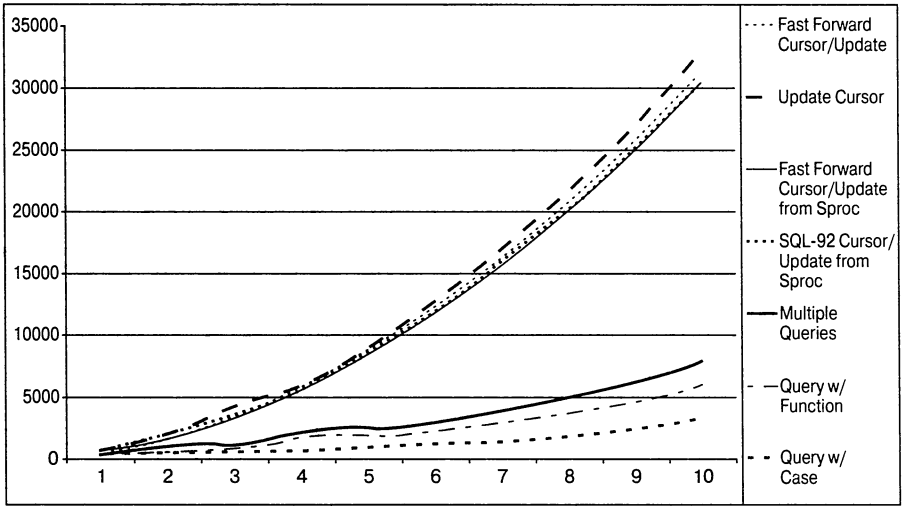


Рис. 20.1. Как видно по результатам тестирования производительности, курсоры практически не масштабируемы

Два решения, использующие пользовательские функции, неплохо проявили себя с курсорами. Они масштабируются, показывая линейное падение производительности, в то же время те же функции, используемые в запросах, проявили себя исключительно хорошо.

Особо выделяется на графике пунктирная линия, соответствующая запросу с выражением CASE. Несмотря на то что программный код кажется громоздким и медленным, он показал наилучшую производительность и масштабируемость — это решение даже обогнало все остальные, использующие запросы. Спрашивается, почему? Да потому, что из всех решений именно запросы с выражениями CASE передают наибольшее управление оптимизатору запросов SQL Server.

Все проведенные тесты показали, что курсоры работают как минимум в десять раз медленнее пакетов инструкций, к тому же весьма плохо масштабируются. При некоторой оптимизации разница в производительности может достичь и больших величин. На самом деле на практике я встречал курсор, который работал семь часов, в то время как переработанное в пакет решение справилось с задачей всего за три минуты. Внедрение сложной логики в курсор может показаться заманчивым, однако платить за это придется колоссальными потерями производительности.

Пример денормализации списка

Второй пример курсора в действии решает задачу денормализации списка. Нам требуется получить из дат туров Outer Banks Lighthouses разделенный запятыми список значений в виде одной строки.

Курсор проходит по всем датам туров, при этом в цикле WHILE все даты добавляются в локальную переменную @EventDates. Битовая локальная переменная @SemiColon определяет, нужен ли между датами разделитель в виде точки с запятой. В конце пакета инструкция SELECT возвращает денормализованный список дат.

```
USE CHA2
DECLARE
  @EventDates VARCHAR(1024),
```

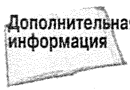
```
  @EventDate DATETIME,
  @SemiColon BIT
SET @Semicolon = 0
SET @EventDates = ''
DECLARE cEvent CURSOR FAST_FORWARD
  FOR SELECT DateBegin
        FROM Event
        JOIN Tour
          ON Event.TourID = Tour.TourID
        WHERE Tour.[Name] = 'Outer Banks Lighthouses'
  OPEN cEvent
  FETCH cEvent INTO @EventDate - Подготовка курсора
  WHILE @@Fetch_Status = 0
    BEGIN
      IF @Semicolon = 1
         SET @EventDates
              = @EventDates + '; '
              + Convert(VARCHAR(15), @EventDate, 107 )
      ELSE
        BEGIN
          SET @EventDates
               = Convert(VARCHAR(15), @EventDate,107 )
          SET @SEMICOLON = 1
        END
        FETCH cEvent INTO @EventDate - к следующей строке
      END
  CLOSE cEvent
DEALLOCATE cEvent

SELECT @EventDates
```

Будет получен следующий результат:

```
------------------------------------------------------------
Feb 02, 2001; Jun 06, 2001; Jul 03, 2001; Aug 17, 2001;
Oct 03, 2001; Nov 16, 2001
```

Курсор с легкостью справился с задачей денормализации списка, и в будущем он может оказаться вам полезным. В то же время пакетное решение, использующее технологию переменной со множеством присвоений и продемонстрированное в следующем примере, работает в десяток раз быстрее и может быть использовано для более интенсивных задач.



Подробно о переменных с множеством присвоений см. в главе 18.

В предлагаемом примере внутренний подзапрос возвращает список внешнему запросу, который, в свою очередь, добавляет каждый полученный результат в переменную `@EventDates`:

```
USE CHA2
DECLARE
  @EventDates VARCHAR(1024)
SET @EventDates = ''

SELECT @EventDates = @EventDates
  + CONVERT(VARCHAR(15), a.d,107 ) + ';'
      FROM (select DateBegin as [d]
              from Event
              join Tour
                on Event.TourID = Tour.TourID
```

```
            WHERE Tour.[Name] = 'Outer Banks Lighthouses') as a
SELECT Left(@EventDates, Len(@EventDates)-1)
  AS 'Outer Banks Lighthouses Events'
```

Будет получен следующий результат:

```
Outer Banks Lighthouses Events
------------------------------------------------------------
Feb 02, 2001; Jun 06, 2001; Jul 03, 2001; Aug 17, 2001;
Oct 03, 2001; Nov 16, 2001
```

Резюме

Когда оптимизация увеличивает производительность в десятки раз (от часов выполнения — в минуты, от минут — в секунды), тогда выполненная работа приносит удовлетворение. Не существует лучшего способа оптимизации хранимой процедуры, чем заменить ненужный курсор. Если вы ищете какой-то очень низкопроизводительный продукт, тогда курсор — именно то, что вам нужно.

Разрешите процитировать моего друга Билла Вона: "Курсор придумал дьявол!" К этому мне нечего добавить. Любой курсор работает до безумия медленно. Ненужные курсоры входят в мой список пяти самых важных проблем производительности SQL Server. Лучший способ настроить курсор — это полностью заменить его элегантным пакетным запросом.

В следующей главе мы возьмем на вооружение все, о чем узнали в предыдущих главах, и будем встраивать пакеты в хранимые процедуры.

Создание хранимых процедур

ГЛАВА

21

В этой главе...

Создание хранимых процедур и управление ими

Передача данных в хранимые процедуры и получение их из них

Использование хранимых процедур в запросах

Выполнение хранимых процедур на связанных серверах

Сценарий ввода заказов, основанный на хранимых процедурах

Основной целью разработок в архитектуре "клиент/сервер" является перенос обработки как можно ближе к данным. Перенос обработки из клиентских приложений на сервер уменьшает объем сетевых потоков, улучшает производительность и облегчает задачи поддержания целостности данных.

Одним из самых популярных методов переноса обработки ближе к данным является создание *хранимых процедур*. В хранимых процедурах нет ничего сложного. В них используются те же запросы и пакеты инструкций T-SQL. Точно так же, как запрос может быть сохранен в представлении, пакет инструкций может быть сохранен под именем хранимой процедуры, одновременно проходя процесс компиляции.

В проектах с архитектурой "клиент/сервер" программный код может быть создан в одном из нескольких мест. Одним из основных отличий между местами хранения является близость к обрабатываемым данным. В пространстве, разделяющем понятия "близко к данным" и "отдельно от данных", хранимые процедуры объединяют достоинства серверных программ с возможностью программирования с рабочих станций (рис. 21.1).



Рис. 21.1. Чем ближе в пространстве обработки данных программа находится к данным, тем лучше

Как серверные программы хранимые процедуры имеют ряд преимуществ.

- Хранимые процедуры хранятся в компилированном виде, поэтому выполняются быстрее, чем пакеты или запросы.
- Выполнение обработки данных на сервере, а не на рабочей станции, значительно снижает нагрузку на локальную сеть.
- Хранимые процедуры имеют модульный вид, поэтому их легко внедрять и изменять. Если клиентское приложение вызывает хранимую процедуру для выполнения некоторой операции, то модификация процедуры в одном месте влияет на ее выполнение у всех пользователей.
- Хранимые процедуры можно рассматривать как важный компонент системы безопасности базы данных. Если все клиенты осуществляют доступ к данным с помощью хранимых процедур, то прямой доступ к таблицам может быть запрещен, и все действия пользователей будут находиться под контролем.

Чтобы создать эффективную хранимую процедуру, не начинайте чтение книги с этой главы. Действительно хорошая хранимая процедура основывается на эффективном пакете (см. главу 18), содержащем производительные запросы SQL (см. главы 7–16). В этой главе описывается только, как взять в руки пакет инструкций и оформить его как хранимую процедуру.

Управление хранимыми процедурами

Реальное управление хранимыми процедурами легко сравнить с заложенной в них логикой. Если вы знаете основные факты и синтаксис, то управление хранимыми процедурами не составит для вас никакого труда.

Инструкции CREATE, ALTER и DROP

Хранимые процедуры управляются посредством инструкций языка определения данных (DDL) `CREATE`, `ALTER` и `DROP`. Инструкция `CREATE` должна быть первой в пакете; терминатор пакета завершает создание хранимой процедуры. В следующем примере создается простая хранимая процедура, которая извлекает данные из таблицы `ProductCategory` базы данных OBXKites:

```
USE OBXKites;
go

CREATE PROCEDURE CategoryList
AS
SELECT ProductCategoryName, ProductCategoryDescription
  FROM dbo.ProductCategory;
RETURN;
```

На протяжении всей этой главы мы будем добавлять в хранимую процедуру `CategoryList` новые функции.

Инструкция `DROP` удаляет хранимую процедуру из базы данных. Инструкция `ALTER` изменяет содержимое всей хранимой процедуры. Для внесения изменений предпочтительнее использовать инструкцию `ALTER`, а не комбинацию инструкций удаления и создания, так как последний метод удаляет все разрешения.

Хранимыми процедурами можно управлять из окна Object Explorer, однако я настоятельно рекомендую использовать для этого сценарии (файлы `.sql`), которые можно организовать в систему управления версиями.

Возвращение набора записей

Если хранимая процедура является сохраненным пакетом, тогда то, что может сделать пакет, может сделать и хранимая процедура. Как пакет может вернуть набор записей из запроса SELECT, так и хранимая процедура может вернуть набор записей из запроса.

Возвращаясь к хранимой процедуре, созданной в предыдущем разделе, можно отметить, что при ее выполнении запрос, содержащийся в ней, вернет все строки из таблицы productcategory:

```
EXEC CategoryList;b
```

Результат выполнения процедуры (сокращенный):

```
ProductCategoryName    ProductCategoryDescription
---------------------- ---------------------------
Accessory              kite flying accessories
Book                   Outer Banks books
Clothing               OBX t-shirts, hats, jackets
...
```

Компиляция хранимых процедур

Компиляция хранимых процедур выполняется автоматически при первом их запуске, после чего скомпилированный код сохраняется в памяти (точнее, SQL Server создает план выполнения запросов и программного кода хранимой процедуры, после чего тот сохраняется в памяти).

Для отслеживания скомпилированных объектов SQL Server использует таблицу Master.dbo.SysCacheObjects. Чтобы просмотреть скомпилированный код, запустите на выполнение следующий запрос:

```
SELECT cast(C.sql as Char(35)) as StoredProcedure, cacheobjtype,
       usecounts as Count
  FROM Master.dbo.SysCacheObjects C
  JOIN Master.dbo.SysDatabases D
    ON C.dbid = C.dbid
  WHERE D.Name = DB_Name()
    AND ObjType = 'Proc'
  ORDER BY StoredProcedure;
```

Результат выполнения запроса (сокращенный):

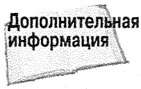
```
StoredProcedure                    cacheobjtype    Count  ObjType
---------------------------------  --------------- ------ --------
CREATE PROCEDURE [dbo].[CleanBatchR Compiled Plan   2      Proc
CREATE PROCEDURE [dbo].[CleanEventR Compiled Plan   1      Proc
CREATE PROCEDURE [dbo].[CleanExpire Compiled Plan   2      Proc
CREATE PROCEDURE [dbo].[CleanExpire Compiled Plan   2      Proc
...
```

План выполнения запроса хранимой процедуры может устареть, пока находится в памяти. Если радикально изменяется распределение данных, добавляются или удаляются индексы, то перекомпиляция хранимой процедуры может значительно повысить производительность. Чтобы вручную инициировать процесс перекомпиляции хранимой процедуры, используется системная процедура sp_recompile. Она помечает хранимую процедуру или триггер, чтобы при следующем ее запуске она автоматически была перекомпилирована:

```
EXEC sp_recompile CategoryList;
```

Результат выполнения будет следующим:

```
Object 'CategoryList' was successfully marked
for recompilation.
```



Более подробно о планах выполнения запросов и методах их хранения в памяти речь пойдет в главе 50.

Шифрование хранимых процедур

Когда создается хранимая процедура, ее текст сохраняется в таблице `SysComments`. Этот текст сохраняется не для ее выполнения, а для последующей возможной модификации.

Системная хранимая процедура `sp_helptext` может извлечь исходный текст хранимой процедуры из таблицы `SysComments`, например:

```
sp_helptext CategoryList;
```

Результат будет следующим:

```
Text
----------------------------------
CREATE PROCEDURE CategoryList
AS
SELECT *
  FROM dbo.ProductCategory
```

Если хранимая процедура создается с параметром `WITH ENCRYPTION`, то ее текст в таблице `SysComments` невозможно прочитать. Как правило, сторонние производители программного обеспечения шифруют свои хранимые процедуры. Следующая инструкция `ALTER` сохранит процедуру `CategoryList` с параметром `WITH ENCRYPTION`, после чего мы попытаемся прочитать ее исходный текст:

```
ALTER PROCEDURE CategoryList
WITH ENCRYPTION
AS
SELECT *
  FROM dbo.ProductCategory;

sp_helptext CategoryList;
```

В результате получим следующее сообщение:

```
The text for object 'CategoryList' is encrypted.
```

Системные хранимые процедуры

Основной синтаксис языка SQL содержит всего десять инструкций: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `ALTER`, `DROP`, `GRANT`, `REVOKE` и `DENY`. В то же время компания Microsoft заложила в базу данных `master` массу хранимых процедур, позволяющих выполнить сотни задач. Чтобы эти процедуры стали доступны всем базам данных, нужно знать правила управления их областью видимости. Любая хранимая процедура, начинающаяся с символов `sp_` и находящаяся в базе данных `master`, может быть выполнена в любой базе данных. Если возникает конфликт между именами системной хранимой процедуры и другой, содержащейся в локальной базе данных, то выполняется последняя.



При создании хранимых процедур руководствуйтесь общепринятыми соглашениями об именах. Не начинайте имена создаваемых хранимых процедур с символов `sp_`, иначе вы потенциально становитесь жертвой конфликтов имен и недоразумений. Я начинаю имена своих хранимых процедур с символа `p`.

Передача данных в хранимые процедуры

Хранимая процедура становится только более ценной, если использует параметры. Ранее созданная хранимая процедура `CategoryList` возвращает названия всех категорий товаров. В то же время процедуре, извлекающей конкретную строку таблицы, нужно передавать некоторый параметр (например, идентификатор товара).

Хранимые процедуры SQL Server могут иметь массу входных и выходных параметров (точнее говоря, вплоть до 2100 единиц).

Входные параметры

В инструкции `CREATE PROCEDURE` можно перечислить параметры, передаваемые хранимой процедуре, указав их непосредственно после ее имени. Каждый из параметров должен начинаться с символа `@`. Для хранимой процедуры он является локальной переменной. Как и все локальные переменные, параметры должны объявляться с допустимыми типами данных. При вызове хранимой процедуры должны также указываться все параметры (если, конечно, некоторые из них не имеют значений, заданных по умолчанию).

В следующем примере создается хранимая процедура, возвращающая одну категорию товаров. Параметр `@CategoryName` может принимать текстовую строку в таблице Unicode длиной до 35 символов. Передаваемое значение в самой хранимой процедуре присваивается локальной переменной и используется в предложении `WHERE`:

```
USE OBXKites;

go
CREATE PROCEDURE CategoryGet
   (@CategoryName NVARCHAR(35))
AS
SELECT ProductCategoryName, ProductCategoryDescription
  FROM dbo.ProductCategory
  WHERE ProductCategoryName = @CategoryName;
```

При выполнении следующего программного кода хранимой процедуре в качестве параметра передается литерал `'Kite'`, который подставляется в качестве локальной переменной в предложение `WHERE` запроса `SELECT`:

```
EXEC CategoryGet 'Kite';
```

Результат выполнения хранимой процедуры:

```
ProductCategoryName     ProductCategoryDescription
----------------------- ------------------------------
Kite                    a variety of kites, from simple
                        stunt, to Chinese, to novelty
```

Если хранимой процедуре передается множество параметров, то они должны сохранять порядок, указанный в определении. Можно также передавать параметры в любом порядке, но при этом указывать их имена. Если эти два метода смешиваются, то после первого явного указания имени параметра все остальные должны использовать тот же метод.

В следующих трех примерах продемонстрированы вызовы хранимых процедур и передача им параметров с использованием исходного порядка и имен:

```
EXEC StoredProcedure
  @Parameter1 = n,
  @Parameter2 = 'n';

EXEC StoredProcedure n, 'n';

EXEC StoredProcedure n, @Parameter2 = 'n';
```

Значения параметров, заданные по умолчанию

При вызове хранимой процедуры вы обязаны указать все ее параметры, кроме тех, для которых были определены значения по умолчанию. В определении хранимой процедуры значения по умолчанию определяются добавлением к имени параметра знака равенства и значения:

```
CREATE PROCEDURE имя_процедуры (
  @переменная тип_данных = значение_по_умолчанию
  );
```

В следующем примере, извлеченном из учебной базы данных OBXKites, продемонстрировано использование значений по умолчанию параметров хранимой процедуры. Если в эту процедуру передается название некоторой категории, то она возвращает данные только о ней. Если же ей ничего не передается, то в предложении WHERE используется значение по умолчанию — NULL, что позволяет вернуть все категории товаров:

```
CREATE PROCEDURE pProductCategory_Fetch2(
  @Search NVARCHAR(50) = NULL
)
-- Если @Search = null, возвращаются все категории товаров
-- Если в @Search передается значение, фильтруются названия
AS
  SET NOCOUNT ON;
  SELECT ProductCategoryName, ProductCategoryDescription
    FROM dbo.ProductCategory
    WHERE ProductCategoryName = @Search
      OR @Search IS NULL;
IF @@RowCount = 0
  RAISERROR(
    'Категория ''%s'' не найдена. ',14,1,@Search);
```

В первом вызове мы передадим хранимой процедуре имя категории:

```
EXEC pProductCategory_Fetch 'OBX';
```

Будет получен следующий результат:

```
ProductCategoryName     ProductCategoryDescription
----------------------- ---------------------------------
OBX                     OBX stuff
```

Если процедура pProductCategory_Fetch выполняется без явного указания параметра, то переменной @Search присваивается значение по умолчанию — null, что делает истинным выполнение условия предложения WHERE для всех строк таблицы ProductCategory.

```
EXEC pProductCategory_Fetch;

ProductCategoryName     ProductCategoryDescription
----------------------- ---------------------------------
Accessory               kite flying accessories
Book                    Outer Banks books
```

```
Clothing                OBX t-shirts, hats, jackets
Kite                    a variety of kites, from simple
                        stunt, to Chinese, to novelty
Material                Kite construction material
OBX                     OBX stuff
Toy                     Kids stuff
Video                   stunt kite contexts and lessons,
                        and Outer Banks videos
```

Получение данных из хранимой процедуры

SQL Server предлагает четыре способа получения данных из хранимой процедуры. Пакет может вернуть данные из инструкции `SELECT` или команды `raiseerror`. Хранимые процедуры используют эти методы пакетов и добавляют к ним два собственных: выходные переменные и команду `output`.

Выходные параметры

Выходные параметры позволяют хранимой процедуре возвращать данные вызывающей клиентской программе. При этом указание ключевого слова `output` обязательно как при определении процедуры, так и при ее вызове. В самой хранимой процедуре выходные параметры являются локальными переменными. В вызывающей процедуре или пакете выходные переменные должны быть предварительно определены, чтобы получить результирующие значения. Когда выполнение хранимой процедуры завершается, текущее значение параметра передается локальной переменной вызывающей программы.

Несмотря на то что выходные параметры обычно используются только для вывода значений, на самом деле они имеют двойное назначение.



Выходные параметры обычно используют, когда из хранимой процедуры требуется получить одну информационную единицу, а не набор данных. Для возвращения одной строки информации гораздо выгоднее в смысле быстродействия использовать выходные параметры, чем подготавливать набор данных.

В следующем примере выходной параметр используется для возвращения названия конкретного товара из таблицы `Product` базы данных OBXKites.

1. В пакете определяется локальная переменная `@ProdName`, получающая значение выходного параметра.
2. В пакете вызывается хранимая процедура с указанием имени выходного параметра `@ProdName`.
3. В хранимой процедуре локальная переменная `@ProdName`, соответствующая переданному выходному параметру, создается в заголовке; ее изначальное значение равно `null`.

   Теперь следует подготовить данные для выходного параметра `@ProductName`.
4. Инструкция `SELECT` в теле хранимой процедуры присваивает переменной `@ProductName` значение `Basic Box Kite 21 inch`, соответствующее товару с кодом `1001`.

5. Хранимая процедура завершает выполнение, и управление передается вызывающему пакету; при этом полученное значение `@ProductName` передается его одноименной локальной переменной.

6. Вызывающий пакет с помощью команды `print` выводит полученное значение на экран.

Ниже приведен текст хранимой процедуры.

```
USE OBXKites;
go
CREATE PROC GetProductName (
  @ProductCode CHAR(10),
  @ProductName VARCHAR(25) OUTPUT )
AS
SELECT @ProductName = ProductName
  FROM dbo.Product
  WHERE Code = @ProductCode;
```

А это текст вызывающего пакета:

```
USE OBXKITES;
DECLARE @ProdName VARCHAR(25);
EXEC GetProductName '1001', @ProdName OUTPUT;
PRINT @ProdName;
```

Будет получен следующий результат:

```
Basic Box Kite 21 Inch
```

Позже в этой главе будет приведен более сложный пример, взятый из базы данных OBXKites. В нем выходной параметр будет использован для передачи номера заказа из одной хранимой процедуры в другую.

Использование команды RETURN

Команда `return` позволяет завершить выполнение процедуры в ее середине и вернуть значение вызывающему пакету или клиенту. С технической точки зрения эта команда может использоваться при работе с любым пакетом, однако значение она может вернуть только из хранимой процедуры или функции.

Возвращаемое значение `0` указывает на успешное выполнение процедуры и установлено по умолчанию. Компания Microsoft зарезервировала значения от –99 до –1 для служебного пользования. Разработчикам для возвращения состояния ошибки пользователю рекомендуется использовать значения –100 и меньше.



Используйте возвращаемое значение для индикации успешного или аварийного завершения процедуры. Старайтесь не использовать его для возвращения реальных данных. Если вам нужно получить не набор данных, а всего одно значение, лучше воспользуйтесь выходным параметром.

При вызове хранимой процедуры, если ожидается выходное значение, команда `exec` должна использовать целочисленную переменную:

```
EXEC @локальная_переменная = имя_хранимой_процедуры;
```

В следующем примере хранимая процедура возвращает индикатор успешного или аварийного завершения в зависимости от значения входного параметра:

```
CREATE PROC IsItOK (
@OK VARCHAR(10) )
AS
```

```
IF @OK = 'OK'
  RETURN 0
ELSE
  RETURN -100;
```

А вот пример вызывающего пакета:

```
DECLARE @ReturnCode INT;
EXEC @ReturnCode = IsITOK 'OK';
PRINT @ReturnCode;
EXEC @ReturnCode = IsItOK 'NotOK';
PRINT @ReturnCode;
```

Результаты выполнения пакета будут следующими:

```
0
-100
```

Маршруты и область определения возвращаемых данных

Любая хранимая процедура имеет четыре способа возвращения данных (`SELECT`, `raiseerror`, `return` и выходные параметры). Решение относительно использования одного из этих методов принимается на основании количества и назначения возвращаемых данных, а также области их определения. Область определения возможных четырех методов следующая.

- Команда `return` и выходные параметры передают значения локальным переменным вызывающего пакета.
- Команда `raiserror` и результирующий набор данных передаются клиентскому приложению. Сама вызывающая процедура или пакет ничего не знает об этих значениях.

Если пакет, вызывающий хранимую процедуру А, которая, в свою очередь, вызывает хранимую процедуру Б, выполняется в анализаторе запросов, то хранимая процедура А не увидит набора данных или ошибки `raiserror`, возвращенных процедурой Б (рис. 21.2).

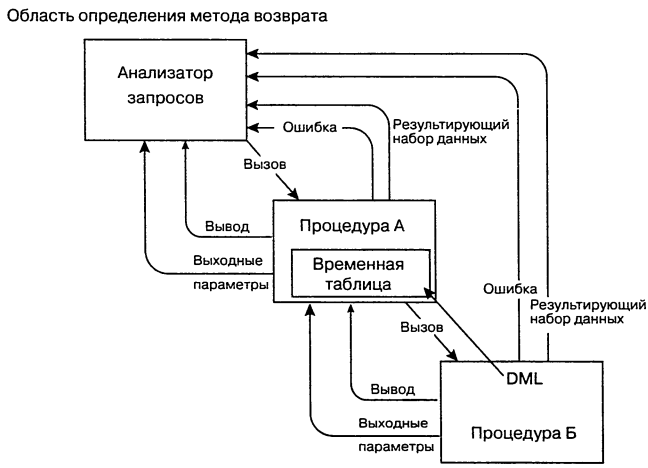


Рис. 21.2. Маршруты и область определения возвращаемых данных зависят от используемого метода

Если некоторой хранимой процедуре нужно работать с набором данных, возвращаемым другой хранимой процедурой, которую она вызывает, должна использоваться временная таблица. Если такая таблица будет создана вызывающей процедурой, то ее область определения распространится на нее.

На рис. 21.2 процедура Б может выполнить любые инструкции DML относительно любой временной таблицы, созданной процедурой А. После завершения процедуры Б эти данные станут доступны процедуре А.



С любым возвращаемым набором данных SQL Server по умолчанию будет посылать сообщение о количестве обработанных или возвращенных строк. Это всего лишь небольшой нюанс, но при неформальном тестировании я обнаружил, что это замедляет выполнение запроса на 17%.
Исходя из этого, выработайте привычку начинать любую хранимую процедуру следующими строками:

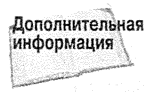
```
AS
SET NoCount ON
```

Более полную информацию о конфигурировании подключений вы получите в главе 34.

Использование хранимых процедур в запросах

Хранимые процедуры обычно запускаются на выполнение с помощью команды `exec` из пакета или клиентского приложения. В то же время их можно вызывать и в предложении `FROM` запроса, используя при этом функцию `openquery()`.

Функция `Openquery()` посылает сквозной запрос внешнему источнику данных для удаленного выполнения. Если в функции `openquery()` указана хранимая процедура, то ее результаты передаются на локальный сервер.



Подробно о функции `openquery()` см. в главе 15.

Так как результирующий набор данных хранимой процедуры возвращается функцией и используется как источник данных в предложении `FROM` инструкции `SELECT`, предложение `WHERE` может сократить объем вывода хранимой процедуры.

Несмотря на то что этот прием позволяет использовать хранимые процедуры в инструкциях `SELECT`, он не так оптимизирован, как техника передачи любых ограничений для обработки хранимой процедурой. Единственным достоинством функции `openquery()` является то, что она позволяет вызывать сложную хранимую процедуру из любого запроса.

В следующем примере предполагается, что было установлено подключение к связанному серверу `NOLI`:

```
SELECT * FROM OpenQuery(
NOLI
'EXEC OBXKites.dbo.pProductCategory_Fetch')
WHERE ProductCategoryDescription Like '%stuff%'
```

Результат выполнения запроса:

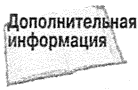
```
ProductCategoryName    ProductCategoryDescription
---------------------- --------------------------
OBX                    OBX stuff
Toy                    Kids stuff
```



Если вам нужно вызвать сложный программный код из инструкции SELECT, то использование функции openquery() для вызова хранимой процедуры сработает, однако синтаксис будет слишком сложный. Лучше для этих целей создать выражение CASE или пользовательскую функцию.

Выполнение удаленных хранимых процедур

Существуют два метода вызова хранимых процедур, размещенных на другом сервере: ссылка по четырехкомпонентному имени и распределенный запрос. Оба метода требуют, чтобы удаленный сервер был связан с текущим. Хранимые процедуры могут только вызываться удаленно — создаваться удаленно они не могут.



О том, как установить защищенные подключения к удаленным серверам, см. в главе 15.

Вызов удаленной хранимой процедуры с помощью четырехкомпонентного имени выглядит следующим образом:

сервер.база_данных.схема.имя_процедуры

В следующем примере в базе данных OBXKites во втором экземпляре сервера NOLI (рассматриваемом как удаленный) создается новая категория товаров:

```
EXEC [Noli\SQL2].OBXKites.dbo.pProductCategory_AddNew
'Food', 'Eatables'
```

В качестве альтернативы для вызова удаленной хранимой процедуры может использоваться функция OpenQuery():

```
OpenQuery(имя_связанного_сервера, 'exec хранимая_процедура')
```

В следующем примере на выполнение запускается хранимая процедура pCustomerType_Fetch, размещенная в базе данных по умолчанию для текущего регистрационного имени пользователя, использованного для подключения к серверу NOLI\SQL2. Если база данных по умолчанию указана некорректно, то для обращения к корректной базе будет использовано трехчастное имя:

```
SELECT CustomerTypeName, DiscountPercent, [Default]
  FROM OPENQUERY(
  [Noli\SQL2], 'OBXKites.dbo.pCustomerType_Fetch')
```

Результат выполнения запроса:

```
CustomerTypeName     DiscountPercent Default
-------------------- --------------- -------
Preferred            10              0
Retail               00              1
Wholesale            15              0
```

Как и в случае с другими распределенными запросами, если транзакция обновляет данные более чем на одном сервере, должна быть запущена служба координатора распределенных запросов.

Завершенная хранимая процедура

В этом разделе будет представлен полный сценарий с хранимыми процедурами из учебной базы данных OBXKites. В нем для добавления заказов в базу данных будут использоваться три хранимые процедуры: `pGetPrice`, `pOrder_AddNew` и `pOrder_AddItem`. В этом сценарии будет продемонстрировано множество средств T-SQL и хранимых процедур. При этом в начале каждого из соответствующих подразделов будет описано назначение и содержание процедуры, которой он посвящен.



Полный текст рассматриваемых хранимых процедур и вызывающих их пакетов вы можете найти в файлах `OBXKites_Create.sql` и `OBXKites_Populate.sql`.

Хранимая процедура `pGetPrice`

В хранимой процедуре `pGetPrice` продемонстрировано использование значений параметров по умолчанию, выходных параметров, обработки ошибок и взаимоблокировок. На входе она принимает код товара, необязательную дату и необязательный код клиента. Используя эту информацию, процедура определяет корректную цену и возвращает ее в выходной переменной.

Каждый клиент может быть отнесен к определенной категории, а каждая из категорий может иметь определенную величину скидки. Если в процедуру передается код клиента, то в процедуре в первую очередь вычисляется установленная для него скидка; в противном случае скидка считается нулевой.

База данных OBXKites содержит таблицу прайс-листа, в которой каждый продукт может иметь разные цены в разные дни. Этот метод позволяет сохранить историю ценообразования и вводить цены заранее. Если функция `pGetPrice` запускается с пустым значением даты, то используется текущая дата. При этом для определения даты установки цены берется в расчет максимальная дата, не превышающая текущую (или заданную). Как только дата ценообразования будет определена, найти корректную цену просто. Для вычисления даты ценообразования в процедуре `pGetPrice` используется подзапрос.

Ошибки обрабатываются в блоке `Catch`. Если ошибка вызвана превышением времени ожидания блокировки или взаимоблокировкой, то обработчик ожидает еще четверть секунды, после чего переходит к метке `LockTimeOutRetry`, расположенной в начале процедуры, чтобы повторить попытку блокировки снова и успешно выполнить процедуру. Если после пяти попыток блокировка так и не была установлена, обработчик сообщает об ошибке и выходит из процедуры. Вот полный текст хранимой процедуры `pGetPrice`:

```
CREATE PROCEDURE pGetPrice(
  @Code CHAR(10),
  @PriceDate DATETIME = NULL,
  @ContactCode CHAR(15) = NULL,
  @CurrPrice MONEY OUTPUT
  )
AS
-- Возвращает активную цену для заданной даты или текущего дня
-- Тип заказчика определяет величину скидки
-- Выходной параметр, @CurrPrice, будет содержать активную цену
-- Пример кода для вызова этой хранимой процедуры:
-- Declare @Price money
-- EXEC GetPrice '1006', NULL, @Price OUTPUT
```

```
-- Select @Price
SET NOCOUNT ON
DECLARE
  @DiscountPercent NUMERIC (4,2),
  @Err INT,
  @ErrCounter INT
SET @ErrCounter = 0
SET @CurrPrice = NULL

LockTimeOutRetry:
BEGIN TRY
IF @PriceDate IS NULL
  SET @PriceDate = GETDATE()
-- устанавливаем скидку;
-- если код клиента отсутствует, она равна нулю
SELECT @DiscountPercent = CustomerType.DiscountPercent
  FROM dbo.Contact
  JOIN dbo.CustomerType
    ON contact.CustomerTypeID = CustomerType.CustomerTypeID
  WHERE ContactCode = @ContactCode
IF @DiscountPercent IS NULL
  SET @DiscountPercent = 0
SELECT @CurrPrice = Price * (1-@DiscountPercent)
  FROM dbo.Price
  JOIN dbo.Product
    ON Price.ProductID = Product.ProductID
  WHERE Code = @Code
        AND EffectiveDate =
               (SELECT MAX(EffectiveDate)
                  FROM dbo.Price
                  JOIN dbo.Product
                     ON Price.ProductID = Product.ProductID
                  WHERE Code = @Code
                        AND EffectiveDate <= @PriceDate)
IF @CurrPrice IS NULL
  BEGIN
     RAISERROR('Code: ''%s'' has no established price.',15,1, @Code)
     RETURN -100
  END
END TRY
BEGIN CATCH
  SET @Err = @@ERROR
  IF (@Err = 1222 OR @Err = 1205) AND @ErrCounter = 5
    BEGIN
       RAISERROR ('Unable to Lock Data after five attempts.', 16,1)
       RETURN -100
    END
  IF @Err = 1222 OR @Err = 1205 -- Lock Timeout / Deadlock
    BEGIN
       WAITFOR DELAY '00:00:00.25'
       SET @ErrCounter = @ErrCounter + 1
      GOTO LockTimeOutRetry
    END
-- неизвестная ошибка
RAISERROR (@err, 16,1) WITH LOG
RETURN -100
END CATCH
```

Хранимая процедура pOrder_AddNew

Данные заказа содержатся в двух таблицах: [Order] и OrderDetail. Первая из них содержит информацию заголовка, а вторая — список товаров. Открытие заказа предполагает сбор и проверку информации заголовка, генерацию номера заказа (OrderNumber) и вставку строки в таблицу [Order].

Хранимая процедура pOrder_AddNew принимает в качестве параметров код заказчика, код работника, ответственного за прием заказа, местонахождение пункта продажи и дату приема заказа.

Продажи иногда осуществляются задним числом, так что предполагать, что датой заказа будет текущая — бессмысленно. Если дата заказа не передается в процедуру, она устанавливается в инструкции INSERT в текущую (так как параметр принимает значение по умолчанию — null).

Код заказчика также является необязательным параметром. Если он не передается в процедуру, принимается значение по умолчанию — 0 и затем преобразуется в пустое значение идентификатора клиента (ContactID). Схема базы данных допускает пустые значения идентификаторов заказчиков, предполагая, что некоторые из них захотят остаться анонимными.

Все записи может распознать человек, однако база данных для репликации использует глобальные универсальные идентификаторы GUID для заказчиков, местоположений и сотрудников и проверяет их в таблицах классификаторов.

Когда все коды будут проверены, процедура находит номер последнего заказа и увеличивает его на единицу. Заказ вставляется в строку таблицы в той же транзакции, в которой определяется его номер. Во избежание дублирования номеров заказов уровень изоляции транзакции устанавливается в serialized. (В реальных приложениях обычно используются более сложные методы генерации номера заказа, которые позволяют избежать дубликатов в пространстве всех точек продаж.)

Используемая схема обработки ошибок сходна с процедурой pGetPrice. Единственное отличие заключается в том, что перед обработкой ошибки транзакция откатывается.

В заключение процедура возвращает номер заказа в выходной переменной, позволяя вызывающему пакету вставить его в таблицу и приступить к формированию строк заказа. Ниже приведен код создания хранимой процедуры pOrder_AddNew.

```
CREATE PROC pOrder_AddNew (
  @ContactCode CHAR(15) = 0,
-- по умолчанию заказчик анонимный
  @EmployeeCode CHAR(15),
  @LocationCode CHAR(15),
  @OrderDate DATETIME = NULL,
  @OrderNumber INT OUTPUT
  )
AS
-- Логика:
-- Если номер заказчика указан, проверяем его
SET NOCOUNT ON
DECLARE
  @ContactID UNIQUEIDENTIFIER,
  @OrderID UNIQUEIDENTIFIER,
  @LocationID UNIQUEIDENTIFIER,
  @EmployeeID UNIQUEIDENTIFIER,
  @Err INT,
  @ErrCounter INT
SET @ErrCounter = 0
```

```
LockTimeOutRetry:
-- Находим идентификатор заказчика
IF @ContactCode = 0
  SET @ContactID = NULL
ELSE
 BEGIN
   SELECT @ContactID = ContactID
     FROM dbo.Contact
     WHERE ContactCode = @ContactCode
   SET @Err = @@ERROR
     IF @Err <> 0 GOTO ErrorHandler
   IF @ContactID IS NULL
     BEGIN - указанный заказчик не найден
       RAISERROR(
         'Код заказчика: ''%s не найден',15,1, @ContactCode)
       RETURN -100
     END
   END

-- Находим идентификатор местоположения
SELECT @LocationID = LocationID
  FROM dbo.Location
  WHERE LocationCode = @LocationCode
SET @Err = @@ERROR
  IF @Err <> 0 GOTO ErrorHandler
IF @LocationID IS NULL
  BEGIN - Местоположение не найдено
    RAISERROR(
      'Код региона: ''%s'' не найден',15,1, @LocationCode)
    RETURN -100
  END
IF EXISTS(SELECT *
            FROM dbo.Location
            WHERE LocationID = @LocationID
              AND IsRetail = 0)
  BEGIN - Нет точек продаж
     RAISERROR(
      'Регион: ''%s'' не имеет точек продажи',15,1,
         @LocationCode)
     RETURN -100
  END

-- Находим идентификатор сотрудника
SELECT @EmployeeID = ContactID
  FROM dbo.Contact
  WHERE ContactCode = @EmployeeCode
SET @Err = @@ERROR
  IF @Err <> 0 GOTO ErrorHandler
IF @EmployeeCode IS NULL
  BEGIN - Сотрудник не найден
    RAISERROR(
     'Код сотрудника: ''%s'' не найден',15,1, @EmployeeCode)
    RETURN -100
  END

-- Генерация номера заказа
SET @OrderID = NEWID()
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

```
BEGIN TRANSACTION
  SELECT @OrderNumber = Max(OrderNumber) + 1
    FROM [Order]
  SET @OrderNumber = ISNULL(@OrderNumber, 1)
  SET @Err = @@ERROR
  IF @Err <> 0
    BEGIN
      ROLLBACK TRANSACTION
      GOTO ErrorHandler
    END
-- Все OK - выполняем вставку
INSERT dbo.[Order] (
    OrderID, ContactID, OrderNumber,
    EmployeeID, LocationID, OrderDate )
  VALUES (
    @OrderID, @ContactID,@OrderNumber,
    @EmployeeID, @LocationID, ISNULL(@OrderDate,GETDATE()))
IF @Err <> 0
  BEGIN
    ROLLBACK TRANSACTION
    GOTO ErrorHandler
  END
COMMIT TRANSACTION

RETURN - номер заказа @OrderNumber установлен
ErrorHandler:
IF (@Err = 1222 OR @Err = 1205) AND @ErrCounter = 5
  BEGIN
    RAISERROR ('Unable to Lock Data after five attempts.', 16,1)
    RETURN -100
  END
IF @Err = 1222 OR @Err = 1205 -- Отсутствие блокировки
                              -- или взаимоблокировка
  BEGIN
    WAITFOR DELAY '00:00:00.25'
    SET @ErrCounter = @ErrCounter + 1
    GOTO LockTimeOutRetry
  END
-- иначе -- неизвестная ошибка
RAISERROR (@err, 16,1) WITH LOG
RETURN -100
```

Хранимая процедура `pOrder_AddItem`

Когда строка заказа вставлена в таблицу, третья хранимая процедура вставляет в заказ строки товаров. Процедура продажи должна быть достаточно гибкой, поэтому в данной процедуре предусмотрено множество значений по умолчанию. Единственными обязательными параметрами являются номер заказа и количество. Товар может определяться или названием, или кодом. Цена за единицу товара извлекается с помощью ранее описанной хранимой процедуры `pGetPrice`. В заключение, если требуется доставка товара заказчику, вводятся ее дата и комментарий. Если информация о доставке отсутствует, процедура предполагает, что доставлять товар нужно немедленно.

Как и другие хранимые процедуры, `pOrder_AddItem` начинается с проверки всех параметров и вставки ассоциированных идентификаторов в таблицу `OrderDetail`.

В этой процедуре цена товара определяется процедурой pGetPrice, если в параметре она пуста. (В главе 22 мы создадим функцию fGetPrice.) С целью сравнения в данной процедуре также использован вызов этой функции, но эта часть текста закомментирована. Ниже приведен текст хранимой процедуры pOrder_AddItem.

```
CREATE PROCEDURE pOrder_AddItem(
  @OrderNumber CHAR(15),
  @Code CHAR(15) = 0, -- по умолчанию товар ищется на складе
  @NonStockProduct NVARCHAR(256) = NULL,
  @Quantity NUMERIC(7,2),
  @UnitPrice MONEY = 0, -- по умолчанию цена определяется
                        -- по прайс-листу
  @ShipRequestDate DATETIME = NULL, -- по умолчанию доставка
                                    -- немедленная
  @ShipComment NVARCHAR(256) = NULL -- комментарий не обязателен
  )
AS
DECLARE
  @OrderID UNIQUEIDENTIFIER,
  @ProductID UNIQUEIDENTIFIER,
  @ContactCode CHAR(15),
  @PriceDate DATETIME,
  @Err INT,
  @ErrCounter INT
SET @ErrCounter = 0

LockTimeOutRetry:
-- Получаем идентификатор заказа
SELECT @OrderID = OrderID
  FROM dbo.[Order]
  WHERE OrderNumber = @OrderNumber
  SET @Err = @@ERROR
    IF @Err <> 0 GOTO ErrorHandler

-- Получаем идентификатор товара
SELECT @ProductID = ProductID
  FROM Product
  WHERE Code = @Code
  SET @Err = @@ERROR
    IF @Err <> 0 GOTO ErrorHandler

--- Получаем код клиента и дату установки прайс-листа
SELECT @ContactCode = ContactCode, @PriceDate = OrderDate
  FROM dbo.[Order]
  LEFT JOIN Contact
    ON [Order].ContactID = Contact.ContactID
SET @Err = @@ERROR
  IF @Err <> 0 GOTO ErrorHandler

-- Получаем цену за единицу товара
IF @UnitPrice IS NULL
  EXEC pGetPrice
     @Code, @PriceDate, @ContactCode, @UnitPrice OUTPUT
-- Альтернативный вызов функции fGetPrice
-- SET @UnitPrice = dbo.fGetPrice (
-- @Code,@PriceDate, @ContactCode)
SET @Err = @@ERROR
  IF @Err <> 0 GOTO ErrorHandler
```

```
  IF @UnitPrice IS NULL
    BEGIN
      RAISERROR(
        'Для кода товара: ''%s'' цена не установлена. ',15,1,
         @Code)
      RETURN -1
    END

-- Устанавливаем дату доставки
IF @ShipRequestDate IS NULL
  SET @ShipRequestDate = @PriceDate
-- Выполняем вставку строки
INSERT OrderDetail(
    OrderID, ProductID, NonStockProduct, Quantity,
    UnitPrice, ShipRequestDate, ShipComment)
  VALUES (
    @OrderID, @ProductID, @NonStockProduct, @Quantity,
    @UnitPrice, @ShipRequestDate, @ShipComment)
SET @Err = @@ERROR
  IF @Err <> 0 GOTO ErrorHandler
RETURN 0

ErrorHandler:
IF (@Err = 1222 OR @Err = 1205) AND @ErrCounter = 5
  BEGIN
    RAISERROR (
     'Невозможно блокировать данные за пять попыток.', 16,1)
    RETURN -100
  END
-- Превышено время ожидания блокировки или взаимоблокировки
IF @Err = 1222 OR @Err = 1205
  BEGIN
    WAITFOR DELAY '00:00:00.25'
    SET @ErrCounter = @ErrCounter + 1
    GOTO LockTimeOutRetry
  END
else - неизвестная ошибка
  RAISERROR (@err, 16,1) WITH LOG
  RETURN -100
```

Добавление заказа

Чтобы вы смогли увидеть созданные хранимые процедуры в действии, в следующем пакете, взятом из сценария `OBXKites_Populate.sql`, демонстрируется создание двух заказов. Этот пакет может быть реально послан на SQL Server для вставки в базу данных OBXKites двух заказов.

Хранимая процедура `pOrder_AddNew` создает строку нового заказа и возвращает вызывающему пакету его номер. После этого пакет создает строки заказа, передавая этот номер хранимой процедуре `pOrder_AddItem`. Локальная переменная пакета `@OrderNumber` используется для получения номера заказа из процедуры `pOrder_AddNew` и передачи его в каждый вызов процедуры `pOrder_AddItem`.

При формировании первого заказа в явном виде указаны имена параметров; при формировании второго заказа параметры перечисляются в установленном в определении процедуры порядке.

```
DECLARE @OrderNumber INT

--Первый заказ
EXEC pOrder_AddNew
  @ContactCode = '101',
  @EmployeeCode = '120',
  @LocationCode = 'CH',
  @OrderDate=NULL,
  @OrderNumber = @OrderNumber output
EXEC pOrder_SetPriority @OrderNumber, '1'
EXEC pOrder_AddItem
  @OrderNumber = @OrderNumber,
  @Code = '1002',
  @NonStockProduct = NULL,
  @Quantity = 12,
  @UnitPrice = NULL,
  @ShipRequestDate = '11/15/01',
  @ShipComment = NULL

-- Второй заказ
EXEC pOrder_AddNew
  '101', '120', 'CH', NULL, @OrderNumber output
EXEC pOrder_AddItem
  @OrderNumber, '1002', NULL, 3, NULL, NULL, NULL
EXEC pOrder_AddItem
  @OrderNumber, '1003', NULL, 5, NULL, NULL, NULL
EXEC pOrder_AddItem
  @OrderNumber, '1004', NULL, 2, NULL, NULL, NULL
EXEC pOrder_AddItem
  @OrderNumber, '1044', NULL, 1, NULL, NULL, NULL
```

Резюме

Использование хранимых процедур является одним из способов сохранения и оптимизации пакетов. Хранимые процедуры компилируются и сохраняются в памяти при первом запуске. Не существует более быстрого метода выполнения инструкций SQL и не существует программных объектов, находящихся ближе к данным. Подобно пакетам, хранимые процедуры могут вернуть набор данных, воспользовавшись инструкцией SELECT.

В следующей главе мы рассмотрим пользовательские функции, которые комбинируют достоинства хранимых процедур с переносимостью представлений.

ГЛАВА

22

Создание пользовательских функций

В этой главе...

Создание скалярных функций

Замена представлений внедренными табличными функциями

Использование сложного программного кода в табличных функциях для получения результирующего набора данных

Пользовательские функции были представлены в версии SQL Server 2000, но поначалу были как-то вяло восприняты сообществом пользователей SQL Server.

Затем сообщество открыло для себя, что с помощью пользовательских функций можно внедрить в запросы обработку сложной логики и таким образом решать задачи, для которых прежде требовалось использование курсора. В результате пользовательские функции стали излюбленным инструментом всех серьезных разработчиков приложений для SQL Server.

Достоинства пользовательских функций перечислены ниже.

- С их помощью можно внедрить в запросы сложную логику.
- Создавая новые функции, можно проектировать сложные выражения.
- Эти функции обладают всеми достоинствами представления, поскольку могут использоваться в предложении `FROM` инструкции `SELECT` и в выражениях и могут быть задействованы в схеме. К тому же пользовательские функции могут принимать параметры, в то время как представления — нет.
- Они обладают достоинствами хранимых процедур, так как могут быть скомпилированы и оптимизированы таким же способом.

Главным аргументом противников пользовательских функций является вопрос переносимости.

Пользовательские функции привязаны к SQL Server, и любую базу данных, использующую множество таких функций, будет сложно или даже невозможно перенести на другую платформу СУБД без существенной переработки. Эта задача усложняется тем, что также должны быть переписаны и все инструкции `SELECT`, в которые внедрены пользовательские

функции. Если в будущем планируется развертывание базы данных на других платформах, то лучше заменить все пользовательские функции представлениями или хранимыми процедурами.

Пользовательские функции подразделяются на три типа (рис. 2.1).

- Скалярные, возвращающие одно значение.
- Внедренные табличные, аналогичные представлениям.
- Сложные табличные, создающие в программном коде результирующий набор данных.

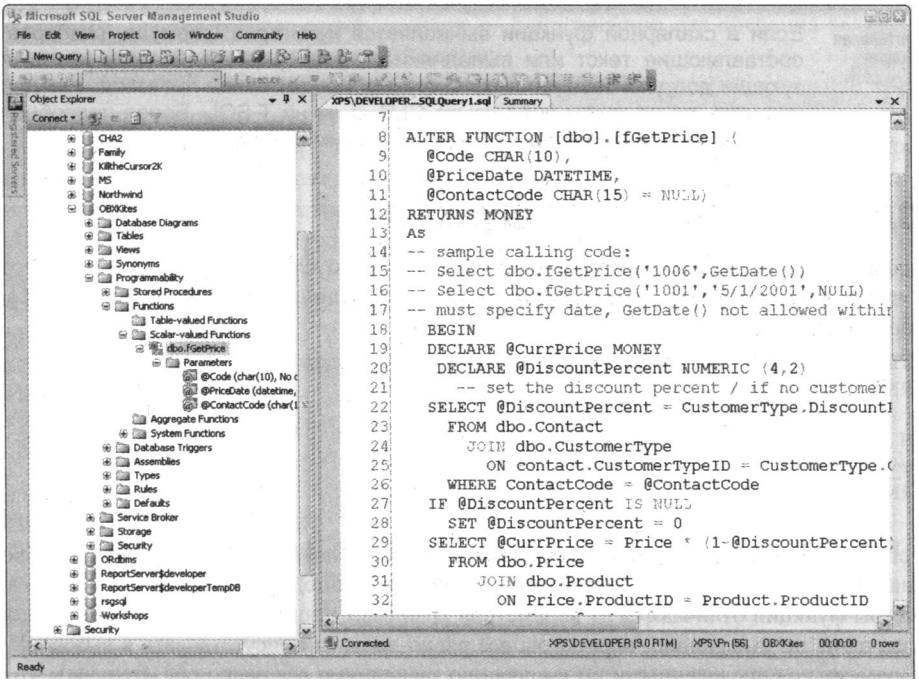


Рис. 22.1. В окне **Object Explorer** *утилиты Management Studio перечислены все пользовательские функции, определенные в базе данных, с разделением на скалярные и табличные*



Пользовательские функции в версии SQL Server 2005 были улучшены в двух отношениях. Во-первых, несмотря на то, что код CLR нельзя назвать лучшим инструментом для серверных программ, работающих с данными, в некоторых пользовательских функциях лучше использовать все-таки его. Во-вторых, табличные пользовательские функции могут принимать параметры в виде строк, аналогично тому, как коррелированные подзапросы работают с внешними запросами. Это позволяет разрешить некоторые сложные проблемы.

Скалярные функции

Скалярными называют те функции, которые возвращают одно значение. Эти функции могут принимать множество параметров, выполнять вычисления, но в результате выдают одно значение. Эти функции могут использоваться в любых выражениях, даже участвующих в ограничениях проверки. Значение возвращается функцией с помощью оператора `return` — эта команда должна завершать скалярную функцию.

Скалярные функции должны быть *детерминированными*. Это значит, что для одних и тех же входных параметров они должны генерировать одно и то же выходное значение. По этой причине некоторые встроенные функции (например, `getdate()`, `rasd()` и `newid()`) недопустимы в скалярных пользовательских функциях.

В скалярных пользовательских функциях не допускаются операции обновления базы данных, но в то же время они могут работать с локальными временными таблицами. Они не могут возвращать данные BLOB (двоичные большие объекты) таких типов, как `text`, `image` и `ntext`, равно как табличные переменные и курсоры.



Если в скалярной функции вычисляется некоторая формула, разбирается на составляющие текст или выполняются некоторые другие задачи, не использующие доступ к базе данных, то функция будет более производительной, если написать ее на каком-либо языке CLR, а не на T-SQL. Более подробно о программировании на языках CLR мы поговорим в главе 27.

Создание скалярных функций

Скалярные функции создаются, изменяются и удаляются с помощью тех же инструкций DDL, что и другие объекты, хотя синтаксис немного отличается, чтобы определить возвращаемое значение:

```
CREATE FUNCTION имя_функции (входные_параметры)
RETURNS тип_данных
AS
BEGIN
текст_функции
RETURN выражение
END
```

В списке входных параметров должны быть указаны типы данных и, в случае необходимости, значения по умолчанию, аналогично хранимым процедурам (*параметр=умолчание*). Параметры функций отличаются от параметров хранимых процедур тем, что даже если определены значения по умолчанию, параметры все равно должны присутствовать в вызове функции (т.е. параметры с определенными по умолчанию значениями все равно обязательны). Чтобы запросить значение по умолчанию при вызове функции, ей передается ключевое слово `default`.

Считается хорошей практикой создавать для параметров функции по умолчанию, поскольку они предохраняют функцию от генерирования ошибки, если она используется в выражении, а необходимый параметр ей не предоставлен; или если инструкция `SELECT` не отобрала в функции ни одной строки.

Следующая скалярная функция выполняет простую арифметическую операцию; ее второй параметр имеет значение по умолчанию:

```
CREATE FUNCTION dbo.Multiply (@A INT, @B INT = 3)
RETURNS INT
AS
BEGIN
  RETURN @A * @B
End
go
SELECT dbo.Multiply (3,4)
SELECT dbo.Multiply (7, DEFAULT)
```

Результат выполнения пакета будет следующим:

```
---------
12
21
```



Системные функции, разработанные компанией Microsoft, хранятся в базе данных `master` и должны вызываться с префиксом в виде двух двоеточий (`::`*имя_функции*).

В следующем примере представлена более сложная скалярная функция `fGetPrice`, функционально аналогичная представленной в предыдущей главе хранимой процедуре `pGetPrice`, которая возвращала единственный результат в своем выходном параметре. Эта хранимая процедура возвращала корректную цену для заданной даты и детерминированной скидки клиента. Так как возвращаемое значение — единственное, эта операция является кандидатом на реализацию в виде скалярной функции.

Эта функция имеет такое же командное наполнение, как и приведенная выше хранимая процедура, за исключением того, что результат возвращается не с помощью выходного параметра `@CurrPrice`, а в операторе `return`. Параметр кода клиента имеет значение по умолчанию `null`. Еще одним отличием является то, что хранимая процедура использовала значение по умолчанию `null` для параметра даты, подставляя вместо него текущую дату с помощью функции `getdate()`. Пользовательские функции являются детерминированными и не могут использовать функции типа `getdate()`, поэтому параметр даты в данном случае является обязательным. Ниже приведен текст скалярной функции `fGetPrice()`.

```
CREATE FUNCTION fGetPrice (
    @Code CHAR(10),
    @PriceDate DATETIME,
    @ContactCode CHAR(15) = NULL)
RETURNS MONEY
As
BEGIN
  DECLARE @CurrPrice MONEY
  DECLARE @DiscountPercent NUMERIC (4,2)
-- устанавливаем процент скидки
-- если код заказчика не передан, скидка будет нулевой
  SELECT @DiscountPercent = CustomerType.DiscountPercent
    FROM dbo.Contact
      JOIN dbo.CustomerType
        ON contact.CustomerTypeID =
           CustomerType.CustomerTypeID
    WHERE ContactCode = @ContactCode
  IF @DiscountPercent IS NULL
    SET @DiscountPercent = 0
  SELECT @CurrPrice = Price * (1-@DiscountPercent)
    FROM dbo.Price
      JOIN dbo.Product
        ON Price.ProductID = Product.ProductID
    WHERE Code = @Code
      AND EffectiveDate =
         (SELECT MAX(EffectiveDate)
            FROM dbo.Price
              JOIN dbo.Product
                ON Price.ProductID = Product.ProductID
            WHERE Code = @Code
               AND EffectiveDate <= @PriceDate)
  RETURN @CurrPrice
END
```

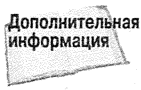
Вызов скалярных функций

Скалярные функции могут использоваться в любом месте выражений, где допустимо одно значение. Пользовательские скалярные функции должны всегда вызываться с помощью двухкомпонентного имени (*владелец.имя*). В следующем примере продемонстрирован вызов функции `fGetPrice()` из базы данных OBXKites:

```
USE OBXKites
SELECT dbo.fGetPrice('1006',GetDate(),DEFAULT)
SELECT dbo.fGetPrice('1001','5/1/2001',NULL)
```

Результат выполнения пакета:

```
----------------------
125.9500
----------------------
14.9500
```



Функция `dbo.GenColUpdate` является пользовательской скалярной функцией, используемой при аудите, и будет рассмотрена в главе 24. Пользовательская функция `dbo.TitleCase` была создана в главе 8.

Создание функций со связанной схемой

Все три типа пользовательских функций могут быть созданы с использованием преимуществ связывания со схемой. Представления могут связываться со схемой, но эта функция недоступна для хранимых процедур. Связывание со схемой делает невозможным изменение и удаление всех объектов, от которых зависит функция. Если связанная со схемой функция обращается к таблице `TableA`, то к последней могут быть добавлены столбцы; в то же время не может быть изменен или удален ни один из существующих ее столбцов, равно как и вся таблица.

Для создания функции, связанной со схемой, вставьте параметр между ключевыми словами `returns` и `as`, как показано ниже.

```
CREATE FUNCTION FunctionName (Input Parameters)
RETURNS DataType
WITH SCHEMA BINDING
AS
BEGIN;
  инструкции;
  RETURNS выражение;
END;
```

Подключение к схеме не только предупреждает пользователя о том, что некоторая операция повлияет на объект, — оно отвергнет эти изменения. Для удаления связи со схемой, чтобы разрешить изменения, используется инструкция `ALTER`, отключающая это связывание.

Внедренные табличные функции

Второй тип пользовательских функций очень похож на представления. Оба обрамляют сохраняемую инструкцию `SELECT`. Внедренные табличные функции имеют все достоинства представлений, добавляя к ним использование параметров и возможность компиляции. Как и в представлениях, если инструкция `SELECT` обновляема, то обновляемой является и функция.

Создание внедренной табличной функции

Внедренная табличная функция не имеет в своем теле блока BEGIN...END — вместо этого возвращается результирующий набор данных инструкции SELECT в виде таблицы с заданным именем:

```
CREATE FUNCTION имя_функции (параметры)
RETURNS Table
AS
RETURN (инструкция_SELECT)
```

Следующая внедренная табличная функция является функциональным эквивалентом представления vEventView, созданного нами в главе 14.

```
USE CHA2
go
CREATE FUNCTION fEventList ()
RETURNS Table
AS
RETURN(
SELECT dbo.CustomerType.Name AS Customer,
    dbo.Customer.LastName, dbo.Customer.FirstName,
    dbo.Customer.Nickname,
    dbo.Event_mm_Customer.ConfirmDate, dbo.Event.Code,
    dbo.Event.DateBegin, dbo.Tour.Name AS Tour,
    dbo.BaseCamp.Name, dbo.Event.Comment
  FROM dbo.Tour
    INNER JOIN dbo.Event
      ON dbo.Tour.TourID = dbo.Event.TourID
    INNER JOIN dbo.Event_mm_Customer
      ON dbo.Event.EventID = dbo.Event_mm_Customer.EventID
    INNER JOIN dbo.Customer
      ON dbo.Event_mm_Customer.CustomerID
         = dbo.Customer.CustomerID
    LEFT OUTER JOIN dbo.CustomerType
      ON dbo.Customer.CustomerTypeID
         = dbo.CustomerType.CustomerTypeID
    INNER JOIN dbo.BaseCamp
      ON dbo.Tour.BaseCampID = dbo.BaseCamp.BaseCampID)
```

Вызов внедренной табличной функции

Для извлечения данных с помощью функции fEventList вызовите ее в предложении FROM инструкции SELECT:

```
SELECT LastName, Code, DateBegin
  FROM dbo.fEventList()
```

Результат выполнения инструкции (сокращенный):

```
LastName      Code        DateBegin
------------ ---------- -----------------------
Anderson      01-003      2001-03-16 00:00:00.000
Brown         01-003      2001-03-16 00:00:00.000
Frank         01-003      2001-03-16 00:00:00.000
...
```

Как и в хранимых процедурах, основная задержка в производительности возникает при первом вызове функции, когда она компилируется и сохраняется в памяти. Все дальнейшие вызовы срабатывают быстрее.

Сравнивая внедренные табличные функции с другими объектами SQL Server, можно сказать, что по производительности они стоят на одной ступеньке с хранимыми процедурами и на 5–10% быстрее представлений.

Использование параметров

Одним из преимуществ внедренных табличных функций по сравнению с представлениями является возможность первых включать параметры в предварительно скомпилированные инструкции SELECT. Представления не могут иметь параметров и обычно ограничивают результат с помощью добавления предложения WHERE в инструкцию SELECT, вызывающую представление.

В следующих примерах сравниваются добавление ограничения в представление с использованием параметра функции. Описанное ниже представление возвращает текущий прайслист для всех товаров.

```
USE OBXKites
go

CREATE VIEW vPricelist
AS
SELECT, Price.Price
  FROM dbo.Price
    JOIN dbo.Product P
      ON Price.ProductID = P.ProductID
  WHERE EffectiveDate =
      (SELECT MAX(EffectiveDate)
         FROM dbo.Price
         WHERE ProductID = P.ProductID
            AND EffectiveDate <= GetDate())
```

Для ограничения этого списка ценой одного товара в вызывающую инструкцию добавляется предложение WHERE:

```
SELECT *
  FROM vPriceList
  WHERE = '1001'
```

Будет получен следующий результат:

```
Code     Price
-------- -------------
1001     14.9500
```

Из представления vPriceList SQL Server создает новую инструкцию SELECT, затем вызывает ограничение WHERE внешней инструкции и на их основании генерирует план выполнения запроса.

В противоположность этому функция позволяет передавать ограничение в качестве параметра в уже скомпилированную инструкцию SELECT:

```
CREATE FUNCTION dbo.fPriceList (
  @Code CHAR(10) = Null, @PriceDate DateTime)
RETURNS Table
AS
RETURN(
SELECT Code, Price.Price
  FROM dbo.Price
    JOIN dbo.Product P
      ON Price.ProductID = P.ProductID
  WHERE EffectiveDate =
      (SELECT MAX(EffectiveDate)
```

```
        FROM dbo.Price
        WHERE ProductID = P.ProductID
           AND EffectiveDate <= @PriceDate)
    AND (Code = @Code
     OR @Code IS NULL)
  )
```

Если функция вызывается с первым параметром по умолчанию, то возвращается прайслист для всех товаров на заданную дату:

```
SELECT * FROM dbo.fPriceList(DEFAULT, '2/20/2002')
```

Будет получен следующий результат:

```
Code       Price
---------- ---------
1047       6.9500
1049       12.9500
...
```

Если же в качестве первого параметра передается код товара, то скомпилированная инструкция SELECT в функции вернет только одну строку, соответствующую заданному товару:

```
SELECT * FROM dbo.fPriceList('1001', '2/20/2002')
```

Будет получен следующий результат:

```
Code       Price
---------- ---------
1001       14.9500
```

Коррелированные пользовательские функции

Во внедренных табличных функциях может использоваться введенная в SQL Server 2005 команда APPLY; при этом используются различные значения параметров для разных строк, обрабатываемых главным запросом. Однажды, работая с версией SQL Server 2000, мне пришлось потратить немало времени, чтобы обойти это ограничение, так что мне не остается ничего иного, кроме как поблагодарить разработчиков SQL Server 2005.

Существуют две формы команды APPLY; наиболее распространенной из них является CROSS APPLY. Это имя может несколько ввести в заблуждение, поскольку она работает скорее как внутреннее (INNER JOIN), а не перекрестное объединение (CROSS JOIN). Команда CROSS APPLY объединяет данные из основного запроса с табличным набором данных из пользовательской функции. Если функция не возвращает данных, то строка из основного запроса также не возвращается, как в следующем примере:

```
USE CHA2
go
CREATE FUNCTION fEventList2 (@CustomerID INT)
RETURNS Table
AS
RETURN(
SELECT dbo.CustomerType.Name AS Customer,
  dbo.Customer.LastName, dbo.Customer.FirstName,
  dbo.Customer.Nickname,
  dbo.Event_mm_Customer.ConfirmDate, dbo.Event.Code,
  dbo.Event.DateBegin, dbo.Tour.Name AS Tour,
  dbo.BaseCamp.Name, dbo.Event.Comment
  FROM dbo.Tour
    INNER JOIN dbo.Event
      ON dbo.Tour.TourID = dbo.Event.TourID
    INNER JOIN dbo.Event_mm_Customer
```

```
      ON dbo.Event.EventID = dbo.Event_mm_Customer.EventID
    INNER JOIN dbo.Customer
      ON dbo.Event_mm_Customer.CustomerID
         = dbo.Customer.CustomerID
    LEFT OUTER JOIN dbo.CustomerType
      ON dbo.Customer.CustomerTypeID
         = dbo.CustomerType.CustomerTypeID
    INNER JOIN dbo.BaseCamp
      ON dbo.Tour.BaseCampID = dbo.BaseCamp.BaseCampID
  WHERE Customer.CustomerID = @CustomerID
)

SELECT C.LastName, Code, DateBegin, Tour
 FROM Customer C
   CROSS APPLY fEventList2(C.CustomerID)
 ORDER BY C.LastName
```

Будет получен следующий результат:

```
LastName   Code    DateBegin               Tour
---------- ------ ----------------------- ------------------------
Anderson   01-003 2001-03-16 00:00:00.000 Amazon Trek
Anderson   01-006 2001-07-03 00:00:00.000 Bahamas Dive
Anderson   01-016 2001-11-16 00:00:00.000 Outer Banks Lighthouses
Andrews    01-015 2001-11-05 00:00:00.000 Amazon Trek
Andrews    01-012 2001-09-14 00:00:00.000 Gauley River Rafting
Andrews    01-014 2001-10-03 00:00:00.000 Outer Banks Lighthouses
Bettys     01-013 2001-09-15 00:00:00.000 Gauley River Rafting
Bettys     01-015 2001-11-05 00:00:00.000 Amazon Trek
...
```

Команда `OUTER APPLY` работает почти так же, как левое внешнее объединение. При таком использовании строки главного запроса включаются в результирующий набор данных, независимо от того, вернула ли внедренная табличная функция какие-либо данные.

Табличные функции с множеством инструкций

Пользовательские табличные функции с множеством инструкций комбинируют способность скалярных функций содержать сложный программный код со способностью внедренных табличных функций возвращать результирующий набор данных. Этот тип функций создает табличную переменную, а затем заполняет ее в теле функции. Сформированная таблица впоследствии возвращается функцией и может использоваться в инструкциях `SELECT`.

Главным преимуществом таких функций является то, что результирующий набор данных может формироваться в пакете инструкций, а затем напрямую использоваться в инструкциях `SELECT`. Это позволяет использовать этот тип функций вместо хранимых процедур.

Новая команда `APPLY` может использоваться с пользовательскими табличными функциями с множеством инструкций точно так же, как с внедренными.

Создание табличных функций с множеством инструкций

Синтаксис, используемый для создания табличных функций с множеством инструкций, практически такой же, как и для создания скалярных функций:

```
CREATE FUNCTION имя_функции (входные_параметры)
RETURNS @имя_таблицы TABLE (столбцы)
AS
BEGIN
  Программный код заполнения табличной переменной
  RETURN
END
```

Следующая процедура позволяет создать табличную пользовательскую функцию с множеством инструкций, которая возвращает основной результирующий набор данных.

1. В начале инструкции `CREATE FUNCTION` создается табличная переменная `@Price`.
2. В теле функции две инструкции `INSERT` заполняют переменную `@Price`.
3. После выполнения функции значение табличной переменной `@Price` передается во внешнюю процедуру как результат функции.

Функция `fPriceAvg` возвращает все цены из таблицы `Price` и среднее значение цены для каждого товара:

```
USE OBXKites
go
CREATE FUNCTION fPriceAvg()
RETURNS @Price TABLE
  (Code CHAR(10),
    EffectiveDate DATETIME,
    Price MONEY)
AS
  BEGIN
    INSERT @Price (Code, EffectiveDate, Price)
      SELECT Code, EffectiveDate, Price
        FROM Product
          JOIN Price
            ON Price.ProductID = Product.ProductID
    INSERT @Price (Code, EffectiveDate, Price)
      SELECT Code, Null, Avg(Price)
        FROM Product
          JOIN Price
            ON Price.ProductID = Product.ProductID
        GROUP BY Code
  RETURN
END
```

Вызов функции

Для выполнения функции к ней производится обращение в предложении `FROM` инструкции `SELECT`:

```
SELECT *
  FROM dbo.fPriceAvg()
```

Результат выполнения инструкции:

```
Code   EffectiveDate              Price
------ ------------------------- --------
1001   2001-05-01 00:00:00.000   14.9500
1001   2002-06-01 00:00:00.000   15.9500
1001   2002-07-20 00:00:00.000   17.9500
```

Резюме

Пользовательские функции расширяют возможности объектов SQL Server и обеспечивают максимальную гибкость выражений и инструкций SELECT, в то же время принося в жертву переносимость приложения.

Скалярные пользовательские функции возвращают одно значение и должны быть детерминированными. Внедренные табличные функции аналогичны представлениям и возвращают результат одной инструкции SELECT. Табличные функции с множеством инструкций используют программный код для заполнения табличной переменной, которая затем и возвращается.

Программный код T-SQL может быть заключен в оболочку хранимых процедур, пользовательских функций и триггеров. В следующей главе мы рассмотрим триггеры — особые процедуры T-SQL, которые автоматически запускаются на выполнение при возникновении событий уровня таблицы.

Реализация триггеров

ГЛАВА

23

В этой главе...

Создание триггеров INSTEAD OF и AFTER

Использование данных транзакций в триггерах

Интеграция множества триггеров

Создание триггера для удаления одной строки

Триггеры представляют собой специальные хранимые процедуры, прикрепленные к событиям таблиц. Их невозможно выполнить вручную; их запуск является откликом на события вставки, обновления и удаления данных из таблицы.

Пользователи не могут обойти триггеры, и если триггер не посылает сообщение, пользователи даже не догадываются о его существовании.

Разработка триггеров предполагает знакомство с рядом вопросов, связанных с SQL Server, в том числе с порядком выполнения транзакций, блокировками, языком T-SQL и хранимыми процедурами. Это является необходимым условием создания действенных триггеров. Триггеры состоят из нескольких уникальных элементов и требуют тщательного планирования. В то же время они обеспечивают гарантированное выполнение сложных правил бизнес-логики и проверки данных.

Некоторые администраторы баз данных избегают использования триггеров, поскольку те по своей природе привязаны к одной платформе; если база данных переносится на другую платформу, то бо́льшую их часть придется переписать. Также триггеры несколько снижают производительность приложений. В защиту триггеров можно сказать следующее. Если некоторое правило слишком сложное для ограничений, то оно может быть реализовано только в триггерах. Правила бизнес-логики, реализованные вне SQL Server, — скорее исключения, чем правило. Если триггер неграмотно написан, он будет оказывать существенное влияние на производительность. В то же время хорошо написанный триггер гарантирует обеспечение целостности данных и хорошей производительности.

Основы триггеров

Триггеры SQL Server запускаются один раз при выполнении какой-либо операции модификации данных. Такой режим работы отличается от Oracle, где триггеры могут выполняться как для каждой операции, так и для каждой отдельной обрабатываемой ею строки. Несмотря на то что на первый взгляд это

кажется ограничением, необходимость разработки пакетных триггеров обеспечивает прозрачную логику, равно как и достойную производительность.

Триггеры могут быть созданы для трех событий, соответствующих трем операциям модификации данных: вставке, обновлению и удалению.

В SQL Server 2005 существуют два типа триггеров транзакций: INSTED OF (триггер замены операции) и AFTER (триггер, выполняющийся сразу после операции). Эти типы отличаются своим назначением, временем выполнения и производимым эффектом. Все это детализировано в табл. 23.1.



Триггеры базы данных могут запускаться и командами языка определения данных DDL — CREATE, ALTER и DROP. Поскольку эти триггеры оперируют на уровне схемы базы данных, они рассмотрены в главе 17.

Таблица 23.1. Сравнение типов триггеров

	Триггер INSTEAD OF	Триггер AFTER
Инструкция DML	Автоматически откатывается	Выполняется, если триггер сам не откатит транзакцию
Время выполнения	Перед ограничениями первичного и внешнего ключей	После выполнения транзакции, но перед ее подтверждением
Количество возможных событий таблицы	Одно	Несколько
Можно ли применять к представлениям?	Да	Нет
Допустима ли вложенность?	В зависимости от параметров сервера	В зависимости от параметров сервера
Допустима ли рекурсивность?	Нет	В зависимости от параметров сервера

Порядок выполнения транзакций

Создание триггеров требует четкого понимания общего порядка выполнения транзакций, в противном случае может возникнуть конфликт между ограничениями и триггерами, что превратит разработку приложения и его отладку в кошмар.

Любая транзакция проходит несколько проверок и программных кодов в следующем порядке.

1. Проверка Identity Insert.
2. Ограничение допустимости пустых значений.
3. Проверка типа данных.
4. Выполнение триггера INSTEAD OF. Если такой триггер существует, то выполнение инструкции DML останавливается в этой точке. Триггеры INSTEAD OF не могут быть рекурсивными. Таким образом, если триггер, запущенный событием некоторой инструкции, снова выполняет ту же инструкцию (INSERT, UPDATE или DELETE), во второй раз его присутствие игнорируется.
5. Ограничение первичного ключа.

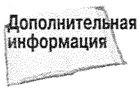
6. Ограничение проверки.
7. Ограничение внешнего ключа.
8. Выполнение инструкции DML и обновление журнала транзакций.
9. Выполнение триггера AFTER.
10. Подтверждение транзакции.
11. Запись в файл данных.

На основе знания порядка выполнения транзакций можно сделать несколько замечаний, касающихся проектирования триггеров.

- Триггер AFTER выполняется после всех ограничений. По этой причине он не может скорректировать данные, поэтому данные должны пройти все проверки ограничениями, включая ограничение внешнего ключа.
- Триггер INSTEAD OF может помочь обойти проблемы, связанные с внешним ключом, но не с допустимостью пустых значений, типом данных или столбцом идентичности.
- Триггер AFTER может принимать во внимание, что данные уже прошли все встроенные проверки целостности данных.
- Триггер AFTER выполняется перед подтверждением транзакции. Таким образом, если данные недопустимы, он имеет возможность откатить транзакцию.



В некоторых приложениях уведомления о событиях могут стать более приемлемым методом обработки событий изменения данных.



Триггеры могут быть написаны с использованием CLR и любого языка семейства .NET. Более подробно этот вопрос будет рассмотрен в главе 27.

Создание триггеров

Триггеры создаются и модифицируются с помощью стандартных команд языка DDL CREATE, ALTER и DROP следующим образом:

```
CREATE TRIGGER имя_триггера ON имя_таблицы
AFTER Insert, Update Delete
AS
программный_код_триггера
```

До выхода версии SQL Server 2000 в SQL Server существовали только триггеры AFTER. Поскольку не было острой необходимости разделять триггеры INSTEAD OF и AFTER, был сохранен старый синтаксис FOR INSERT, UPDATE или DELETE. Для обеспечения работоспособности старых триггеров AFTER они могут создаваться с использованием ключевого слова FOR вместо AFTER.

Подобно хранимым процедурам, триггеры могут создаваться с шифрованием. Однако это шифрование также легко взломать.

Несмотря на то что я настойчиво рекомендую создавать и изменять триггеры с помощью сценариев и управления версиями, вы можете просматривать и редактировать триггеры в окне Object Explorer утилиты Management Studio (рис. 23.1).

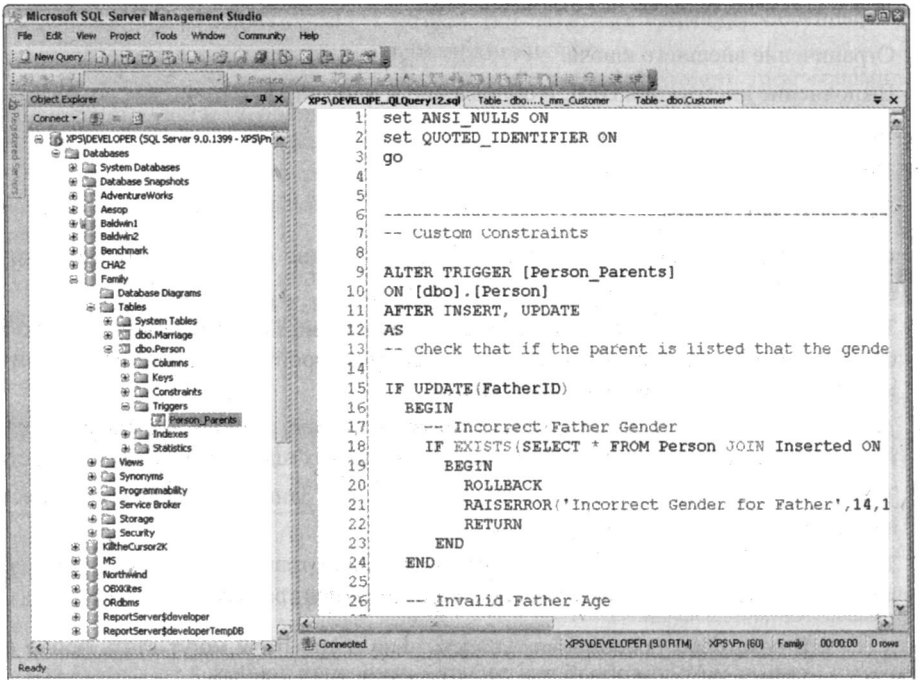


Рис. 23.1. В окне Object Explorer перечислены все триггеры каждой таблицы. С помощью контекстного меню пользователь может редактировать эти триггеры

Триггеры AFTER

Таблица может иметь несколько триггеров AFTER для каждого из трех своих событий (вставка, обновление и удаление). Триггеры AFTER применимы только к таблицам.

Традиционный триггер является триггером AFTER, который запускается после выполнения транзакции, но до ее подтверждения. Эти триггеры могут пригодиться для следующих операций:

- сложная проверка данных;
- поддержка сложных правил бизнес-логики;
- запись журнала аудита данных;
- обслуживание измененных столбцов данных;
- реализация пользовательских проверок целостности данных и каскадных удалений.



Используйте триггеры AFTER в транзакциях, которые с наибольшей вероятностью будут успешными, так как эти триггеры выполняются после завершения операции, но до ее подтверждения. По этой причине триггеры AFTER идеально подходят для проверки данных и поддержания сложных правил.

Когда изучается какой-либо язык программирования, обычно первой написанной программой является ввод на экран традиционной фразы “Hello, world”. При этом от вас больше ничего не требуется, кроме как скомпилировать программу и убедиться, что эта фраза действительно отображается на экране. Проводя аналогию, следующий триггер AFTER при выполнении выводит на экран слова “After Trigger”:

```
USE Family;
CREATE TRIGGER TriggerOne ON Person
AFTER Insert
AS
PRINT 'After Trigger';
```

Для тестирования данного триггера выполним операцию вставки строки:

```
INSERT Person(PersonID, LastName, FirstName, Gender)
  VALUES (50, 'Ebob', 'Bill','M');
```

Получим следующий результат:

```
After Trigger
(1 row(s) affected)
```

Как мы видим, инструкция вставки была выполнена, и тригер вывел на экран заложенное в него сообщение.

Триггеры INSTEAD OF

Триггер INSTEAD OF заменяет транзакцию (т.е. выполняется вместо нее). Это подобно тому, как если бы триггер автоматически откатил транзакцию, для которой был создан.

Каждая таблица ограничена возможностью использования только одного триггера подстановки для каждого из своих событий. В то же время триггеры INSTEAD OF могут применяться к представлениям точно так же, как к таблицам.

Не путайте триггеры INSTEAD OF с триггерами BEFORE или событиями BEFORE UPDATE. Это не одно и то же. Триггер BEFORE, если бы таковой существовал в SQL Server, не пересекался бы с транзакцией, если бы в самом триггере не была предусмотрена инструкция отката.

Триггеры INSTEAD OF особенно полезны, когда заранее известно, что инструкция DML, запустившая триггер, почти наверняка будет отменена, а вместо нее должна быть реализована некоторая логика. Приведем примеры.

- Инструкция DML пытается обновить необновляемое представление. В этом случае триггер INSTEAD OF обновляет таблицы, на которых построено данное представление.
- Инструкция DML пытается напрямую обновить таблицу складских остатков, что недопустимо. В этом случае триггер INSTEAD OF обновляет таблицу складских операций.
- Инструкция DML пытается удалить строку, а триггер INSTEAD OF вместо этого перемещает данную строку в архивную таблицу.

В следующем примере мы создадим и протестируем триггер замены операции вставки:

```
CREATE TRIGGER TriggerTwo ON Person
INSTEAD OF Insert
AS
PRINT 'Instead of Trigger'
go

INSERT Person(PersonID, LastName, FirstName, Gender)
VALUES (51, 'Ebob', '','M')
```

Будет получен следующий результат:

```
Instead of Trigger
(1 row(s) affected)
```

Как мы видим, в результате отображаются заложенные в триггер слова и отчет о том, что была обработана одна строка. Однако попытка извлечения строки с идентификатором 51 завершится неудачей:

```
SELECT LastName
  FROM Person
  WHERE PersonID = 51

LastName
---------------
(0 row(s) affected)
```

В данном случае инструкция INSERT отработала так, как будто одна строка была обработана, хотя эффект этой инструкции был блокирован триггером INSTEAD OF. Вместо вставки строки была выполнена операция вывода на экран заложенной в триггер фразы. Заметим, что созданный ранее триггер AFTER остался в силе, однако его сообщение не было выведено на экран.

Ограничения триггеров

Ввиду самой природы триггеров (программный код, прикрепленный к таблице) они имеют несколько ограничений. Следующие инструкции недопустимы в триггерах:

- CREATE, ALTER и DROP (создание, изменение и удаление таблицы);
- RECONFIGURE (изменение конфигурации);
- RESTORE (восстановление базы данных или журнала);
- DISK RESIZE (изменение размеров дискового пространства);
- DISK INIT (инициализация диска).

Отключение триггеров

Инструкция DML не может обойти триггер, однако системный администратор может его временно отключить. Это лучше комбинации операций удаления и воссоздания триггера, особенно если тот выполняет задачу модификации данных.

Для временного отключения триггера используйте инструкцию DDL ALTER TABLE с параметром DISABLE TRIGGER (для его включения выполняется та же инструкция, но с параметром ENABLE TRIGGER).

```
ALTER TABLE имя_таблицы DISABLE TRIGGER имя_триггера
ALTER TABLE имя_таблицы ENABLE TRIGGER имя_триггера
```

В следующем примере отключается триггер INSTEAD OF с названием TriggerOne таблицы Person:

```
ALTER TABLE Person
  DISABLE TRIGGER TriggerOne
```

Для просмотра состояния триггера (включен он или отключен) используется функция objectproperty(), которой передается идентификатор триггера и параметр ExecIsTriggerDisabled:

```
SELECT OBJECTPROPERTY(
  OBJECT_ID('TriggerOne'),'ExecIsTriggerDisabled')
```

Создание списка триггеров

Так как триггеры скрыты в структуре таблицы, для создания списка триггеров, существующих в базе данных, можно использовать следующий запрос. В приведенном примере также проверяется таблица sysobject на предмет объектов с типом tr, после чего эта таблица объеди-

няется с таблицей Trigger для возможности извлечения имени таблицы. В приведенном ниже примере для вызова функции objectproperty() в каждой строке использован коррелированный подзапрос, преобразующий числовое состояние триггера в словесное описание.

```
SELECT SubString(S2.Name,1,30) as [Table],
  SubString(S.Name, 1,30) as [Trigger],
  CASE (SELECT -- Correlated subquery
         OBJECTPROPERTY(OBJECT_ID(S.Name),
         'ExecIsTriggerDisabled'))
    WHEN 0 THEN 'Включен'
    WHEN 1 THEN 'Отключен'
  END AS Status
  FROM Sysobjects S
    JOIN Sysobjects S2
      ON S.parent_obj = S2.ID
  WHERE S.Type = 'TR'
  ORDER BY [Table], [Trigger]
```

Будет получен следующий результат:

```
Table          Trigger              Status
-------------- -------------------- --------
Person         Person_Parents       Включен
Person         TriggerOne           Отключен
Person         TriggerTwo           Включен
```

Триггеры и безопасность

Только пользователи, принадлежащие фиксированной серверной роли sysadmin или фиксированным ролям dbowner или ddladmin базы данных, имеют разрешение на создание, изменение, включение и отключение триггеров.

Программный код в триггере выполняется, предполагая разрешения владельца таблицы, к которой прикреплен триггер.

Работа с транзакциями

Запускать триггер могут инструкции DML INSERT, UPDATE и DELETE. Очень важно, чтобы триггер имел доступ к изменениям, выполненным инструкцией DML, чтобы проверить полученные значения или обработать результаты транзакции. SQL Server предлагает четыре способа проверки в теле триггера эффекта, произведенного инструкцией DML. Образы Inserted и Deleted содержат наборы данных до и после выполнения инструкции, а функции updated() и columns_updated() можно использовать для определения того, на какие столбцы оказала воздействие инструкция DML.

Определение состава обновленных столбцов

SQL Server предлагает два метода определения состава обновленных столбцов. Функция update() возвращает значение true для одного столбца, если на него повлияла транзакция DML:

```
IF UPDATE(ColumnName)
```

Следующая инструкция INSERT окажет влияние на все столбцы, после чего функция update() укажет на наличие обновлений в конкретном столбце:

```
ALTER TRIGGER TriggerOne ON Person
AFTER Insert, Update
AS
IF Update(LastName)
  PRINT 'Изменен столбец LastName'
ELSE
  PRINT 'Столбец LastName остался нетронутым'
```

Протестируем данный триггер, который проверяет наличие изменений в столбце `LastName`:

```
UPDATE Person
SET LastName = 'Johnson'
WHERE PersonID = 25
```

Будет получен следующий результат:

```
Изменен столбец LastName
```

Обычно функция `update()` используется для выполнения проверки данных по мере необходимости. Нет смысла заново проверять данные в столбце, если они не изменялись инструкцией DML. В то же время эта функция замечает именно наличие операции изменения столбца в инструкции DML, а не изменение самих данных. Например, если инструкция изменит значение `'abc'` на `'abc'` (т.е. фактически не изменит его), функция `update()` все равно отчитается о наличии изменений.

Функция `columns_updated()` возвращает битовую карту `varbinary` наличия изменений в столбцах таблицы. Если конкретный бит имеет значение `true`, это значит, что столбец был изменен. Результат функции `columns_updated()` можно сравнить с целочисленными или двоичными данными с помощью любой битовой операции, чтобы определить, был ли изменен конкретный столбец.

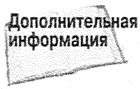
В документации говорится, что столбцы представлены битами в направлении слева направо, однако это не совсем так. На самом деле слева направо следуют байты, а в каждом байте биты следуют справа налево. Дополнительно усложняет ситуацию то, что данные типа `varbinary`, возвращаемые функцией `columns_updated()`, зависят от количества столбцов в таблице.

Следующая функция имитирует реальный режим работы функции `columns_updated()`. Функции передается номер тестируемого столбца и общее количество столбцов в таблице. В результате возвращается битовая маска для этого столбца:

```
CREATE FUNCTION dbo.GenColUpdated
  (@Col INT, @ColTotal INT)
RETURNS INT
AS
BEGIN
-- Copyright 2001 Пол Нильсен
-- Эта функция имитирует режим работы функции Columns_Updated()
DECLARE
  @ColByte INT,
  @ColTotalByte INT,
  @ColBit INT
-- Вычисление позиции байта
SET @ColTotalByte = 1 + ((@ColTotal-1) /8)
SET @ColByte = 1 + ((@Col-1)/8)
SET @ColBit = @col - ((@colByte-1) * 8)
RETURN Power(2, @colbit + ((@ColTotalByte-@ColByte) * 8)-1)
END
```

Эта функция используется в динамическом триггере (или хранимой процедуре) ведения журнала аудита; при этом выполняется операция побитового `and` (`&`) между результатами функций `columns_updated()` и `GenColUpdated()`. Если результат операции равен результату функции `GenColUpdated()`, значит, рассматриваемый столбец на самом деле был обновлен:

```
Set @Col_Updated = Columns_Updated()
...
Set @ColUpdatedTemp =dbo.GenColUpdated(@ColCounter,@ColTotal)
If (@Col_Updated & @ColUpdatedTemp) = @ColUpdatedTemp
```



Полный программный код динамического аудита приведен в главе 24. Сценарий `DynamicAudit.sql`, который можно найти на сайте книги `www.SQLServerBible.com`, применяет этот программный код к базе данных Northwind.

Логические таблицы `Inserted` и `Deleted`

SQL Server позволяет программному коду в теле триггера получить доступ к эффекту от транзакции, которая вызвала триггер. Логические таблицы `Inserted` и `Deleted` представляют собой образы данных, доступные только для чтения. Эти таблицы можно рассматривать как представления журнала транзакций.

Таблица `Deleted` содержит строки в состоянии до применения инструкции DML, а таблица `Inserted` — после применения (табл. 23.2).

Таблица 23.2. Таблицы `Inserted` и `Deleted`

Инструкция DML	Таблица `Inserted`	Таблица `Deleted`
`INSERT`	Вставленные строки	Пустая
`UPDATE`	Строки базы данных после обновления	Строки базы данных до обновления
`DELETE`	Пустая	Строки, подлежащие удалению

Таблицы `Inserted` и `Deleted` имеют очень ограниченную область определения. Их можно увидеть только внутри триггера; хранимые процедуры, вызываемые триггером, не имеют доступа к этим таблицам.

В следующем примере таблица `Inserted` используется для создания отчета о новых значениях в столбце `LastName`:

```
ALTER TRIGGER TriggerOne ON Person
AFTER Insert, Update
AS
SET NoCount ON
IF Update(LastName)
  SELECT 'Вы изменили значение столбца LastName на '
      + Inserted.LastName
    FROM Inserted
```

Если этот триггер прикрепить к таблице `Person` и обновить значение столбца `LastName` следующим образом:

```
UPDATE Person
  SET LastName = 'Johnson'
  WHERE PersonID = 32
```

то получим следующий результат:

```
-------------------------------------------------
Вы изменили значение столбца LastName на Johnson
(1 row(s) affected)
```

Создание триггеров, работающих со множеством строк

Многие триггеры, которые мне встречались на практике, не предусматривали возможность работы с многострочными инструкциями INSERT, UPDATE и DELETE. Они считывали значение из таблицы Inserted или Deleted и сохраняли его в локальной переменной для обработки или проверки. Такая методика позволяет обрабатывать только последнюю строку, обработанную инструкцией DML, что создает большую брешь в системе поддержания целостности данных. Мне также встречались базы данных, которые для прохода по измененным строкам использовали курсор. Именно такое неподобающее программирование и завоевало для триггеров плохую репутацию.



Так как SQL является средой, ориентированной на работу с множествами, любой триггер должен учитывать возможность изменения инструкцией DML множества строк. В такой ситуации лучше всего работать с таблицами Inserted и Deleted с помощью операций над множествами.

Объединение между таблицами Inserted и Deleted или между Inserted и исходной таблицей вернет полное множество строк, на которые оказала воздействие инструкция DML. В табл. 23.3 перечислены возможные комбинации объединений, которые можно использовать в триггерах, работающих с множеством строк.

Таблица 23.3. Предложения FROM для многострочных операций

Инструкция DML	Предложение FROM
INSERT	FROM Inserted
UPDATE	FROM Inserted JOIN Deleted ON Inserted.PK=Deleted.PK
INSERT, UPDATE	FROM Inserted LEFT OUTER JOIN Deleted ON Inserted.PK=Deleted.PK
DELETE	FROM Deleted

В следующем примере триггер просматривает таблицы Inserted и Deleted:

```
ALTER TRIGGER TriggerOne ON Person
AFTER Insert, Update
AS
SELECT D.LastName + ' изменен на ' + I.LastName
  FROM Inserted I
    JOIN Deleted D
      ON I.PersonID = D.PersonID;
GO

UPDATE Person
  SET LastName = 'Carter'
  WHERE LastName = 'Johnson'
```

Будет получен следующий результат:

```
Johnson изменен на Carter
Johnson изменен на Carter
(2 row(s) affected)
```

Следующий триггер, AFTER, извлеченный из учебной базы данных Family, поддерживает следующее правило: идентификатор отца FatherID не только должен указывать на существующее лицо, но к тому же данный человек должен быть мужчиной:

```
CREATE TRIGGER Person_Parents
ON Person
AFTER INSERT, UPDATE
AS
IF UPDATE(FatherID)
BEGIN
-- Неверный пол отца
IF EXISTS(
    SELECT *
      FROM Person
        JOIN Inserted
          ON Inserted.FatherID = Person.PersonID
      WHERE Person.Gender = 'F')
  BEGIN
    ROLLBACK
    RAISERROR('Incorrect Gender for Father',14,1)
    RETURN
  END
END
```

Взаимодействие триггеров

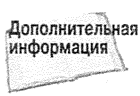
При отсутствии четкого плана база данных, использующая множество триггеров, может быстро стать неуправляемой, и разрешить такие проблемы будет достаточно сложно.

Организация триггеров

В версии SQL Server 6.5 каждое событие могло иметь только один триггер, а один триггер мог применяться только к одному событию. Стиль программирования, обязующий использовать только такие ограниченные триггеры, показал свою несостоятельность. В версии 7.0 SQL Server реализована возможность существования множества триггеров AFTER для одного события таблицы, при этом один триггер может применяться к нескольким событиям. Это позволило использовать более гибкий стиль разработки приложений.

Приняв участие в разработке баз данных, содержащих сотни триггеров, я бы порекомендовал организовывать триггеры не по событиям таблиц, а по задачам, выполняемым триггерами, например:

- проверка данных
- поддержка сложных правил бизнес-логики
- ведение журнала аудита
- изменение даты;
- обеспечение сложной защиты данных.



Все эти задачи будут детально рассмотрены в главе 24.

Вложенные триггеры

Термин *вложение триггеров* указывает на то, могут ли триггеры, вызываемые инструкциями DML, вызывать другие триггеры. Например, если параметр сервера `Nested Triggers` включен и некоторый триггер обновляет таблицу `A`, а эта таблица сама имеет триггер, то и он тоже будет вызван (рис. 23.2).

По умолчанию параметр `Nested Triggers` включен. Включить его явным образом можно с помощью следующих команд:

```
EXEC sp_configure 'Nested Triggers', 1
Reconfigure
```

Если база данных создается с большим объемом серверного программного кода, желательно, чтобы инструкция DML инициировала триггер, который вызывал хранимую процедуру, также инициирующую свой триггер, и т.д.

Триггеры в SQL Server ограничены 32 уровнями рекурсии. Из соображений безопасности рекомендуется проверять текущий уровень рекурсии в самом триггере — для этого предназначена функция `Trigger_NestLevel()`. Когда предел будет достигнут, SQL Server генерирует фатальную ошибку.

Рекурсивные триггеры

Рекурсивный триггер является уникальным типом вложенного триггера `AFTER`. Если триггер выполняет инструкцию DML, которая снова вызывает его самого, он называется рекурсивным (рис. 23.3). Если параметр базы данных `Recursive_Triggers` отключен, то рекурсивная итерация триггеров становится невозможной.

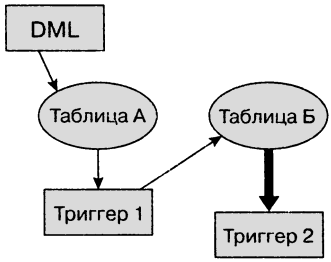


Рис. 23.2. Параметр конфигурации `Nested Triggers` позволяет инструкции DML в теле триггера инициировать еще один триггер

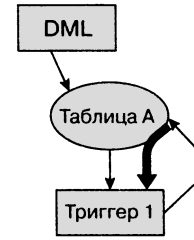


Рис. 23.3. Рекурсивный триггер ссылается сам на себя. Он выполняет инструкцию DML, которая запускает его повторно

Триггер считается рекурсивным только в том случае, когда он напрямую инициирует самого себя. Если в триггере выполняется хранимая процедура, которая обновляет таблицу триггера, такой рекурсивный вызов считается непрямым, и на него не распространяется действие параметра базы данных `Recursive_Triggers`.

Рекурсивные триггеры включаются с помощью инструкции `ALTER DTATBASE`:

```
ALTER DATABASE имя_базы_данных SET RECURSIVE_TRIGGERS ON | OFF
```

В качестве примера полезного рекурсивного триггера рассмотрим триггер `ModifiedData`. Он записывает текущие дату и время в изменяемый столбец обновляемой строки. Используя учебную базу данных OBXKites в качестве модели, добавим в таблицу товаров столбцы `Created` и `Modified`:

```
USE OBXKites
ALTER TABLE dbo.Product
  ADD
    Created DateTime Not Null DEFAULT GetDate(),
    Modified DateTime Not Null DEFAULT GetDate()
```

Представленный триггер вначале выводит свой уровень вложенности, используя функцию `TriggerNest()`. Это очень полезно в целях отладки во вложенных или рекурсивных триггерах. Первый оператор `if` предохраняет столбцы `Created` и `Modified` от ручного изменения пользователем. Если триггер инициируется пользователем, уровень вложенности будет равен единице.

Первый раз триггер инициируется инструкцией `UPDATE`. Все последующие запуски триггера завершаются оператором `Return`, поскольку уровень вложенности превышает единицу. Это позволяет избежать бесконечной рекурсии. Ниже приведен код DDL этого триггера.

```
CREATE TRIGGER Products_ModifiedDate ON dbo.Product
FOR UPDATE
AS
SET NoCount ON
PRINT Trigger_NestLevel()
If Trigger_NestLevel() > 1
Return
If (Update(Created) or Update(Modified))
AND Trigger_NestLevel() = 1
Begin
  Raiserror('Обновление отключено.', 16, 1)
  Rollback
  Return
End
-- Обновление даты изменения строки
UPDATE Product
  SET Modified = getdate()
  FROM Product
    JOIN Inserted
      ON Product.ProductID = Inserted.ProductID
```

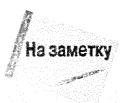
Для тестирования триггера выполним следующую инструкцию `UPDATE`, что приведет к изменению триггером значения столбца `Modified`. После этого инструкция `SELECT` возвращает дату и время из столбцов `Created` и `Modified`:

```
UPDATE PRODUCT
SET [Name] = 'Modified Trigger'
WHERE Code = '1002'

SELECT Code, Created, Modified
FROM Product
WHERE Code = '1002'
```

Будет получен следующий результат:

```
Code   Created                   Modified
------ ------------------------ ------------------------
1002   2002-02-18 09:48:31.700  2002-02-18 15:19:34.350
```



Рекурсивные триггеры необходимы в реплицируемых базах данных.

Триггеры INSTEAD OF и AFTER

Если таблица для одного и того же события имеет триггеры INSTEAD OF и AFTER, возможна следующая последовательность операций.

1. Инструкция DML инициирует транзакцию.
2. Вместо инструкции DML выполняется триггер INSTEAD OF.
3. Если триггер INSTEAD OF выполняет инструкцию DML для того же события таблицы, то процесс продолжается.
4. Выполняется триггер AFTER.

Множество триггеров AFTER

Если некоторое событие таблицы имеет несколько триггеров AFTER, то все они будут выполнены. Порядок выполнения этих триггеров менее важен, чем может показаться на первый взгляд.

Каждый из триггеров имеет возможность откатить транзакцию. Если транзакция откатывается, будет отменена вся работа, выполненная изначальной инструкцией и всеми уже выполненными триггерами.

Тем не менее существует возможность выполнить конкретный триггер в списке первым или последним. Я рекомендую это делать только в том случае, если у некоторого триггера велика вероятность отката транзакции. В этом случае его лучше поставить перед всеми остальными, чтобы повысить общую производительность. С логической же точки зрения порядок выполнения триггеров не имеет значения.

Для задания порядка выпонения триггеров используется системная хранимая процедура sp_settriggerorder. Они имеет следующий синтаксис:

```
sp_settriggerorder
  @triggername = 'имя_триггера',
  @order = 'first' или 'last' или 'none',
  @stmttype = 'INSERT' или 'UPDATE' или 'DELETE'
```

Эффект от установки порядка триггеров не обладает свойством кумулятивности. Например, если установить триггер A первым, а затем установить триггер Б первым, это не отодвинет триггер A на второе место. В данном случае он вернется в неупорядоченное состояние.

Резюме

Триггеры являются ключевой функцией в базах данных с архитектурой "клиент/сервер", позволяют разработчику создавать сложные правила бизнес-логики и гарантировать их строгое выполнение на уровне базы данных. SQL Server 2005 содержит два типа триггеров: AFTER и INSTEAD OF.

В предыдущих четырех главах были представлены технология программирования на языке T-SQL и способы упаковки инструкций в хранимые процедуры, пользовательские функции и триггеры. В следующей главе мы объединим все предложенные технологии для создания расширенного серверного программного кода.

Расширенные технологии T-SQL

ГЛАВА

24

В этой главе...

Методология создания серверного кода баз данных

Выполнение усложненной проверки данных

Реализация сложной ссылочной целостности

Реализация системы безопасности на уровне строк

Создание системы аудита данных

Реализация сложных транзакций

Логическое удаление и архивирование старых данных

При реализации логической схемы приходится внедрять в ограничения множество правил бизнес-логики и поддержания целостности. В то же время некоторые из этих правил могут оказаться слишком сложными для такой реализации. Такие правила можно внедрять в клиентские приложения, на средний уровень программного обеспечения или в сервер базы данных. Из этих трех возможных решений последнее имеет абсолютное превосходство. Нет никакой гарантии, что будущие пользователи будут осуществлять доступ к базе данных с помощью текущего клиентского программного обеспечения или объектов среднего уровня.

Реализация правил бизнес-логики на уровне сервера баз данных с помощью триггеров, хранимых процедур и функций, написанных на языке T-SQL, обладает теми же достоинствами, что и ограничения.

- Правила являются абсолютными и не могут игнорироваться никакими запросами DML (`SELECT`, `INSERT`, `UPDATE` или `DELETE`).
- Правила расположены настолько близко к данным, насколько это возможно, что повышает скорость обработки и снижает сетевые потоки.

Ограничения, триггеры и хранимые процедуры стоят на страже целостности данных.

В предыдущих главах мы обсуждали специфику создания серверного кода. Предложенная методология позволяет максимально приблизить обработку к самим данным. В настоящей главе мы сведем воедино все описанные ранее технологии и предложим методику проектирования баз данных с максимальным использованием серверного кода.

Методология проектирования баз данных с серверным кодом предполагает создание отдельных хранимых процедур для всех операций доступа к данным, используемых клиентским приложением, а также реализацию механизма поддержания правил бизнес-логики на сервере с помощью триггеров и хранимых процедур. Несмотря на то что данная методика существенно усложняет программирование, она имеет ряд несомненных преимуществ.

- Все операции доступа к данным работают согласованно.
- Весь программный код скомпилирован и оптимизирован.
- Обеспечивается усиленная защита данных.
- Все действия, даже операции чтения, можно отследить.
- Поддержка сложных правил и операции обработки удаляются из клиентских приложений.
- Вероятность ошибок запросов минимизирована.

Проверка сложных правил бизнес-логики

Чаще всего проверка данных осуществляется на уровне ограничений, однако ограничения проверки (а также пользовательские функции) ограничены пределами текущей строки.

Сложная проверка данных имеет две особенности.

- Проверка данных может требовать доступа не только к текущей строке, отвергая таким образом возможность реализации с помощью ограничений.
- Проверка данных может быть различной для разных клиентов, компаний, подразделений и т.п.; к тому же она может варьироваться со временем. Процедуры проверки данных могут подгружать требования из таблиц конфигурации, позволяя администратору базы данных изменять их, не затрагивая схему.

Сложную проверку данных лучше всего реализовывать с помощью триггеров, поскольку они — достаточно гибкие, мощные и на 100% оптимизированные программные элементы.

Основным приемом при проектировании таких триггеров является проверка наличия во вставляемой таблице каких-либо данных, не удовлетворяющих правилам бизнес-логики. Если такие данные обнаруживаются, транзакция откатывается. Если не осуществлять такую проверку, триггер не справится с возложенной на него задачей и пропустит в базу некорректные данные.

В качестве альтернативы можно сравнивать переменную `@@Rowcount` транзакции со значением функции с количеством корректных строк в таблице `Inserted`, однако проверка на наличие некорректных данных обычно проще и работает быстрее.

В качестве примера задачи, требующей сложной проверки данных, можно взять следующее правило для базы данных Cape Hatteras Adventures: *любой ответственный за событие экскурсовод должен быть квалифицирован для соответствующего тура.*

Экскурсоводы, ответственные за события, перечислены в таблице `Event_mm_Guide`, поэтому триггер должен проверять данные, вставляемые в эту таблицу.

Чтобы проверить наличие данных, нарушающих данное правило, инструкция `SELECT` объединяет таблицу `Inserted` триггера с таблицей `Event`, чтобы определить идентификатор тура, а затем с таблицей `Event_mm_Guide` — для проверки квалификации ответственного за событие. Обратите внимание на то, что в инструкции не требуется объединение с таблицей `Event_mm_Guide`, — все необходимые данные содержатся в таблице `Inserted`. В завершение инструкция `SELECT` отсекает строки тех экскурсоводов, кто еще не получил квалификацию или у кого она уже просрочена.

```
USE CHA2
CREATE TRIGGER LeadQualified ON Event_mm_Guide
AFTER INSERT, UPDATE
AS
SET NoCount ON
IF EXISTS(
  SELECT *
    FROM Inserted
```

```
     JOIN dbo.Event
       ON Inserted.EventID = Event.EventID
     LEFT JOIN dbo.Tour_mm_Guide
       ON Tour_mm_Guide.TourID = Event.TourID
           AND Inserted.GuideID = Tour_mm_Guide.GuideID
    WHERE
      Inserted.IsLead = 1
      AND
      (QualDate > Event.DateBegin
        OR
        RevokeDate IS NOT NULL
        OR
        QualDate IS NULL )
    )
BEGIN
RAISERROR('Ответственный не квалифицирован.',16,1)
ROLLBACK TRANSACTION
END
```

В следующих двух запросах мы протестируем созданную реализацию технологии сложной проверки, попытавшись назначить ответственными за событие как квалифицированного экскурсовода, так и неквалифицированного. В первом запросе мы попытаемся назначить ответственным за тур Gauley River Rafting экскурсовода Джона Джонсона:

```
INSERT Event_mm_Guide (EventID, GuideID, IsLead)
VALUES (10, 1, 1)
```

Получим следующий результат:

```
Ответственный не квалифицирован.
```

Если же назначить экскурсовода 5 класса Кена Франка ответственным за тур Gauley River, то триггер разрешит вставку:

```
INSERT Event_mm_Guide (EventID, GuideID, IsLead)
VALUES (10, 2, 1)

(1 row(s) affected)
```

Поддержка сложной ссылочной целостности

Реализация декларативной ссылочной целостности с помощью ограничений внешних ключей используется чаще всего, и это правильный подход.

Можно сказать, что перед тем как пытаться создавать сложный триггер поддержания ссылочной целостности, имеет смысл попытаться решить задачу стандартными и более простыми методами. Программный код триггера поддержания ссылочной целостности будет содержать серию объединений первичных и вторичных таблиц, чтобы найти такие значения внешнего ключа таблицы `Inserted`, которые не имеют соответствий в первичной таблице. В рассматриваемом ниже примере таблица `TableB` имеет внешний ключ, указывающий на таблицу `TaableA`. Отмечу, что рассматриваемый программный код не привязан ни к какой конкретной базе данных:

```
CREATE TRIGGER RICheck ON Tour
AFTER INSERT, UPDATE
AS
SET NoCount ON
IF Exists(SELECT *
            FROM Inserted
```

```
            LEFT OUTER JOIN BaseCamp
              ON Inserted.BaseCampID
                = BaseCamp.BaseCampID
           WHERE BaseCamp.BaseCampID IS NULL)
  BEGIN
   RAISERROR
   ('Несоответствующий внешний ключ: Tour.BaseCampID', 16, 1)
    ROLLBACK TRANSACTION
    RETURN
  END
```

Далее мы попробуем назначить тур Amazon Trek базовому лагерю с идентификатором 99. Триггер поддержания ссылочной целостности блокирует обновление данных:

```
UPDATE Tour
  SET BaseCampID = 99
  WHERE TourID = 1
```

Как и ожидалось, результатом будет ошибка:

```
Несоответствующий внешний ключ: Tour.BaseCampID
```

В то же время существуют достаточно редкие случаи, когда ссылочная целостность, необходимая базе данных, не может быть поддержана стандартными ограничениями внешнего ключа. Как правило, такие схемы баз данных содержат многочисленные отношения.

Примером сложной ссылочной целостности может послужить система MRP II, с которой мне приходилось работать. Эта система должна была заполнять строки накладной либо из счета-фактуры, либо из складской таблицы. Вначале мы пытались решить задачу с помощью двух внешних ключей таблицы `Allocation`: один из них указывал на таблицу `OrderDetail` (строки накладной) и обслуживался требованиями таблицы `Product` (товары). Второй ключ указывал на источник заполнения накладной и мог быть либо глобальным универсальным идентификатором строки (GUID) счета-фактуры, либо складским идентификатором GUID. Так как второй внешний ключ не мог однозначно указывать на строку конкретной таблицы, ограничение стандартного внешнего ключа не способно было справиться с задачей.

Чтобы реализовать поддержание такой сложной ссылочной целостности, в таблице `Allocation` был создан триггер. Этот триггер проверял правильность идентификатора строки счета-фактуры или складской позиции. Это было реализовано с помощью запроса различия, сравнивающего все строки таблицы `Inserted` с полями `SourceID` таблиц `Inventory` и `PurchaseOrderDetail`:

```
CREATE TRIGGER AllocationCheck ON Allocation
AFTER INSERT, UPDATE
AS
SET NoCount ON
-- Проверка некорректности складской позиции
IF Exists(SELECT *
            FROM Inserted I
             LEFT OUTER JOIN InventoryItem
              ON I.SourceID = InventoryItem.InventoryItemID
             LEFT OUTER JOIN PurchaseOrderDetail
              ON I.SourceID = PurchaseOrderDetail.PODID
            WHERE Inventory.InventoryID IS NULL
                    AND PurchaseOrderDetail.PODID IS NULL)
  BEGIN
    RAISERROR
     ('Неверный источник товара', 16, 1)
    ROLLBACK TRANSACTION
    RETURN
  END
```

В качестве альтернативы можно было создать два внешних ключа в таблице строк накладной (Allocation), один из которых указывал бы на складскую таблицу, а второй — на таблицу счетов-фактур. Это позволило бы проще решить ту же задачу. Проверка должна была бы сводиться к тому, чтобы только один из этих внешних ключей содержал пустое значение. С такой проверкой справилось бы следующее стандартное ограничение:

```
ALTER TABLE Allocation
  ADD CONSTRAINT AllocationSourceExclusive CHECK
    (PurchaseOrderID IS NULL AND InventoryID IS NOT NULL)
       OR
    (PurchaseOrderID IS NOT NULL AND InventoryID IS NULL)
```

Выбор одного из методов поддержания сложной ссылочной целостности зависит от извлечения корректной информации и привычного стиля работы самого разработчика. Оба метода требуют использования левых внешних объединений и функции coalesce().

Обеспечение защиты данных на уровне строк

СУБД SQL Server идеально подходит для обеспечения защиты информации по вертикали и в то же время лишена возможности динамически обеспечивать ее на уровне строк. Представления с параметром WITH CHECK OPTION могут обеспечить встроенную в программный код защиту. Однако реализация базы данных на таких представлениях приведет к уменьшению производительности и усложнению поддержки базы.

Базы данных промышленного масштаба часто содержат в строках конфиденциальные данные. Можно сформулировать четыре основных правила обеспечения их защиты.

- Данные о складской стоимости товаров, графике их выпуска и содержании в них разных материалов не должны выходить за пределы соответствующего производственного отдела. В то же время программы MRP обычно используют системы отслеживания этих данных в пределах всего предприятия (всех его подразделений и филиалов).
- Данные о сотрудниках должны быть доступны исключительно для отдела кадров и, возможно, для непосредственного начальника.
- Корпоративная торговая система должна допускать покупателей пиломатериалов к информации только о пиломатериалах, а покупателей компьютеров — к информации о компьютерах.
- Любое отделение банка должно быть способно прочитать файл данных любого клиента, но редактировать имеет право только файлы своих клиентов.

Решение этих и других задач защиты информации на уровне строки предполагает создание серверного программного кода, и на то есть ряд причин.

- Таблица системы безопасности может содержать список пользователей и их подразделений, а также их права на чтение и запись информации.
- Процедура системы безопасности проверяет права пользователей относительно запрашиваемых данных и возвращает разрешение или запрет.
- Процедура перехода между записями проверяет разрешение на возврат данных.
- Триггеры вызывают процедуру системы безопасности для проверки прав пользователя на выполнение конкретной инструкции DML на запрашиваемых строках.

Чтобы продемонстрировать описанную конструкцию, мы реализуем ее на примере учебной базы данных OBXKites. Каждый сотрудник из таблицы Contact получит разрешения на чтение, запись или администрирование данных о складских запасах и продажах соответствующего региона. В этой схеме безопасность данных может быть обеспечена с помощью хранимой процедуры, функции, учетной записи NT или триггера.

Таблица безопасности

Таблица безопасности (Security) выступает в роли ассоциативной между таблицами Contact и Location. В ней определены следующие уровни доступа:

- **0 или отсутствие записи** — запрет доступа;
- **1** — разрешение чтения;
- **2** — разрешение записи;
- **3** — разрешение администрирования.

В качестве альтернативы можно было бы использовать трехбитовый столбец, однако привилегии обладают свойством накопления, поэтому больше подходит целочисленный столбец.

Создание таблицы

Таблица безопасности имеет два логических внешних ключа. Внешний ключ, связанный с таблицей регионов, обслуживается стандартным ограничением внешнего ключа. В то же время ссылка на таблицу Contact должна ограничиваться строками, содержащими признак сотрудника, чтобы поддерживать требования ссылочной целостности. Назначение прав доступа бессмысленно при отсутствии ссылки на сотрудника или на регион, поэтому внешние ключи предусматривают каскадное удаление. Столбец прав пользователя имеет ограничение на диапазон значений (от нуля до трех) и еще одно ограничение, запрещающее пользователю иметь одновременно два кода доступа к одному и тому же региону:

```
USE OBXKites
CREATE TABLE dbo.Security (
  SecurityID UniqueIdentifier NOT NULL
  Primary Key NonClustered,
  ContactID UniqueIdentifier NOT NULL
    REFERENCES Contact ON DELETE CASCADE,
  LocationID UniqueIdentifier NOT NULL
    REFERENCES Location ON DELETE CASCADE,
  SecurityLevel INT NOT NULL DEFAULT 0
  )
```

Три следующие команды добавляют в таблицу Security ограничения:

```
CREATE TRIGGER ContactID_RI
ON dbo.Security
AFTER INSERT, UPDATE
AS
SET NoCount ON
IF EXISTS(SELECT *
            FROM Inserted
             LEFT OUTER JOIN dbo.Contact
               ON Inserted.ContactID = Contact.ContactID
            WHERE Contact.ContactID IS NULL
                OR Contact.IsEmployee = 0 )
  BEGIN
   RAISERROR
```

```
    ('Ограничение внешнего ключа: Security.ContactID', 16, 1)
   ROLLBACK TRANSACTION
   RETURN
  END
ALTER TABLE dbo.Security
  ADD CONSTRAINT ValidSecurityCode CHECK
    (SecurityLevel IN (0,1,2,3))
ALTER TABLE dbo.Security
  ADD CONSTRAINT ContactLocation UNIQUE
    (ContactID, LocationID)
```



Так как база данных OBXKites в качестве первичных ключей использует глобальные идентификаторы GUID, для ввода данных лучше использовать хранимые процедуры. Сценарий, относящийся к настоящей главе (`ch24-Advanced T-SQL Solutions.sql`) и находящийся на сайте книги (`www.SQLServerBible.com`), содержит хранимые процедуры, аналогичные приведенным в книге, и предназначенные для ввода данных в таблицу безопасности. В этом сценарии также содержатся и учебные данные.

Безопасность переходов

Для просмотра таблицы `Security` в первую очередь была создана хранимая процедура `pSecurity_Fetch`, которая может вернуть как все записи таблицы безопасности, так и относящиеся только к конкретному сотруднику или региону:

```
CREATE PROCEDURE pSecurity_Fetch(
   @LocationCode CHAR(15) = NULL,
   @ContactCode CHAR(15) = NULL )
AS
SET NoCount ON
SELECT Contact.ContactCode,
       Location.LocationCode,
       SecurityLevel
  FROM dbo.Security
    JOIN dbo.Contact
     ON Security.ContactID = Contact.ContactID
    JOIN dbo.Location
     ON Security.LocationID = Location.LocationID
  WHERE (Location.LocationCode = @LocationCode
        OR @LocationCode IS NULL)
        AND (Contact.ContactCode = @ContactCode
        OR @ContactCode IS NULL)
```

Назначение уровней безопасности

Разрешения системы безопасности устанавливаются путем занесения данных в таблицу `Security`, являющуюся связующей между таблицами контактов и регионов. Продолжая тему серверного программного кода, приведем хранимую процедуру, назначающую уровни системы безопасности парам "сотрудник–регион". В ней вы не найдете для себя ничего нового. Она принимает на входе коды контакта и региона, преобразует их в идентификаторы GUID, а затем выполняет инструкцию вставки:

```
CREATE PROCEDURE pSecurity_Assign(
    @ContactCode VARCHAR(15),
    @LocationCode VARCHAR(15),
    @SecurityLevel INT
    )
AS
```

```
SET NOCOUNT ON
DECLARE
  @ContactID UNIQUEIDENTIFIER,
  @LocationID UNIQUEIDENTIFIER
-- Получение идентификатора контакта
SELECT @ContactID = ContactID
  FROM dbo.Contact
  WHERE ContactCode = @ContactCode
IF @@ERROR <> 0 RETURN -100
IF @ContactID IS NULL
  BEGIN
    RAISERROR
     ('Контакт: ''%s'' не найден', 15,1,@ContactCode)
    RETURN -100
  END
-- Получение идентификатора региона
SELECT @LocationID = LocationID
  FROM dbo.Location
  WHERE LocationCode = @LocationCode
IF @@ERROR <> 0 RETURN -100
IF @LocationID IS NULL
  BEGIN
    RAISERROR
     ('Регион: ''%s'' не найден', 15,1,@LocationCode)
    RETURN -100
  END
-- Вставка
INSERT dbo.Security (ContactID,LocationID, SecurityLevel)
  VALUES (@ContactID, @LocationID, @SecurityLevel)
IF @@ERROR <> 0 RETURN -100
RETURN
```

После создания хранимых процедур pSecurity_Fetch и pSecurity_Assign мы добавим в следующем пакете тестовые данные. Первые два запроса возвращают данные, которые будут использованы для тестирования:

```
SELECT ContactCode
  FROM Contact
  WHERE IsEmployee = 1
```

Будет получен следующий результат:

```
ContactCode
-----------
118
120
119
```

Следующий запрос возвращает коды регионов:

```
SELECT LocationCode FROM Location
```

Результат выполнения запроса:

```
LocationCode
------------
CH
Clt
ElC
JR
KH
W
```

На основе этих данных следующие четыре вызова хранимых процедур назначат сотрудникам уровни доступа к данным:

```
EXEC pSecurity_Assign
  @ContactCode = 118,
  @LocationCode = CH,
  @SecurityLevel = 3

EXEC pSecurity_Assign
  @ContactCode = 118,
  @LocationCode = Clt,
  @SecurityLevel = 2

EXEC pSecurity_Assign
  @ContactCode = 118,
  @LocationCode = Elc,
  @SecurityLevel = 1

EXEC pSecurity_Assign
  @ContactCode = 120,
  @LocationCode = W,
  @SecurityLevel = 2
```

Следующие две команды тестируют вставленные данные с помощью процедуры pSecurity_Fetch. В первой процедуре тестируются установки системы безопасности для региона W:

```
EXEC pSecurity_Fetch @LocationCode = 'W'
```

Будет получен следующий результат:

```
ContactCode LocationCode    SecurityLevel
----------- --------------- -------------
120         W               3
```

Следующий пакет тестирует установки для сотрудника Dave Boston с кодом 118:

```
EXEC pSecurity_Fetch @ContactCode = '118'
```

Будет получен следующий результат:

```
ContactCode  LocationCode SecurityLevel
------------ ------------ -------------
118          Clt          2
118          CH           3
118          ElC          1
```

Схема системы безопасности на уровне строк использует ряд ограничений; следующая команда их тестирует.

Тест на уникальность:

```
EXEC pSecurity_Assign
  @ContactCode = 120,
  @LocationCode = W,
  @SecurityLevel = 2
```

Результат теста:

```
Server: Msg 2627, Level 14, State 2,
Procedure pSecurity_Assign, Line 35
Violation of UNIQUE KEY constraint 'ContactLocation'.
Cannot insert duplicate key in object 'Security'.
The statement has been terminated.
```

Тест на ограничение проверки корректности кода доступа:

```
EXEC pSecurity_Assign
  @ContactCode = 118,
  @LocationCode = W,
  @SecurityLevel = 5
```

Результат теста:

```
Server: Msg 547, Level 16, State 1,
Procedure pSecurity_Assign, Line 35
INSERT statement conflicted with COLUMN CHECK constraint
'ValidSecurityCode'. The conflict occurred in database
'OBXKites', table 'Security', column 'SecurityLevel'.
The statement has been terminated.
```

Тест триггера проверки правила бизнес-логики на доступность кода доступа только для сотрудников:

```
Select ContactCode FROM Contact WHERE IsEmployee = 0
EXEC pSecurity_Assign
  @ContactCode = 102,
  @LocationCode = W,
  @SecurityLevel = 3
```

Результат теста:

```
Ограничение внешнего ключа: Security.ContactID
```

Тестирование ограничения внешнего ключа контакта, который первый проверяется в хранимой процедуре:

```
EXEC pSecurity_Assign
  @ContactCode = 999,
  @LocationCode = W,
  @SecurityLevel = 3
```

Результат теста:

```
Server: Msg 50000, Level 15, State 1, Procedure pSecurity_Assign, Line
19
Контакт: '999' не найден
```

Тестирование ограничения внешнего ключа региона, используемого в хранимой процедуре:

```
EXEC pSecurity_Assign
  @ContactCode = 118,
  @LocationCode = RDBMS,
  @SecurityLevel = 3
```

Результат теста:

```
Server: Msg 50000, Level 15, State 1, Procedure pSecurity_Assign, Line
30
Регион: 'RDBMS' не найден
```

Обновление уровней доступа к данным

Предложенная ранее хранимая процедура `pSecurity_Assign` была способна добавлять новые записи в таблицу безопасности, но не могла обновлять уже существующие ее записи.

Следующая версия этой же процедуры проверяет, существует ли в таблице `Security` заданная комбинация кодов контакта и региона, а затем выполняет соответствующую операцию — обновление или вставку. С помощью одних и тех же параметров новой процедуры установки системы безопасности могут быть и созданы, и скорректированы:

```
ALTER PROCEDURE pSecurity_Assign(
  @ContactCode CHAR(15),
  @LocationCode CHAR(15),
  @SecurityLevel INT
  )
AS
SET NOCOUNT ON
DECLARE
  @ContactID UNIQUEIDENTIFIER,
  @LocationID UNIQUEIDENTIFIER
-- Получение идентификатора контакта
SELECT @ContactID = ContactID
  FROM dbo.Contact
  WHERE ContactCode = @ContactCode
IF @ContactID IS NULL
  BEGIN
    RAISERROR
     ('Контакт: ''%s'' не найден', 15,1,@ContactCode)
    RETURN -100
  END
-- Получение идентификатора региона
SELECT @LocationID = LocationID
  FROM dbo.Location
  WHERE LocationCode = @LocationCode
IF @LocationID IS NULL
  BEGIN
    RAISERROR
      ('Регион: ''%s'' не найден', 15,1,@LocationCode)
    RETURN -100
  END
-- Что выполнять: вставку или обновление?
IF EXISTS(SELECT *
  FROM dbo.Security
  WHERE ContactID = @ContactID
     AND LocationID = @LocationID)
-- Обновление
  BEGIN
    UPDATE dbo.Security
      SET SecurityLevel = @SecurityLevel
        WHERE ContactID = @ContactID
          AND LocationID = @LocationID
    IF @@ERROR <> 0 RETURN -100
  END
-- Вставка
ELSE
  BEGIN
    INSERT dbo.Security
      (ContactID,LocationID, SecurityLevel)
       VALUES (@ContactID, @LocationID, @SecurityLevel)
    IF @@ERROR <> 0 RETURN -100
  END
RETURN
```

Следующий сценарий тестирует способность новой процедуры изменять уровни доступа для контакта с кодом `120` и региона `W`:

```
EXEC pSecurity_Assign
  @ContactCode = 120,
  @LocationCode = W,
```

```
  @SecurityLevel = 2
EXEC pSecurity_Fetch
  @ContactCode = 120
```

Будет получен следующий результат:

```
ContactCode LocationCode SecurityLevel
----------- ------------ -------------
120         W            2
```

В следующем сценарии в таблицу `Security` вставляется новая запись и тут же модифицируется, после чего результат проверяется:

```
EXEC pSecurity_Assign
  @ContactCode = 120,
  @LocationCode = CH,
  @SecurityLevel = 1
EXEC pSecurity_Assign
  @ContactCode = 120,
  @LocationCode = W,
  @SecurityLevel = 3
EXEC pSecurity_Fetch
  @ContactCode = 120
```

Будет получен следующий результат:

```
ContactCode LocationCode  SecurityLevel
----------- ------------- -------------
120         W             3
120         CH            1
```

Хранимая процедура проверки полномочий

Хранимая процедура проверки прав доступа находится в центре системы безопасности строкового уровня. Она возвращает истинное или ложное значение в зависимости от наличия прав на затребованную операцию.

В этой процедуре извлекается допустимый уровень доступа пользователя, который затем сравнивается с необходимым для выполнения операции с данными указанного региона. Если уровень доступа достаточный, возвращается единица, в противном случае — нуль.

```
CREATE PROCEDURE p_SecurityCheck (
    @ContactCode CHAR(15),
    @LocationCode CHAR(15),
    @SecurityLevel INT,
    @Approved BIT OUTPUT )
AS
SET NoCount ON
DECLARE @ActualLevel INT
SELECT @ActualLevel = SecurityLevel
  FROM dbo.Security
  JOIN dbo.Contact
    ON Security.ContactID = Contact.ContactID
  JOIN dbo.Location
    ON Security.LocationID = Location.LocationID
  WHERE ContactCode = @ContactCode
    AND LocationCode = @LocationCode
IF @ActualLevel IS NULL
   OR
   @ActualLevel < @SecurityLevel
   OR
```

```
   @ActualLevel = 0
       SET @Approved = 0
ELSE
       SET @Approved = 1
       RETURN 0
```

В следующем пакете вызывается хранимая процедура `p_SecurityCheck` и используется локальная переменная `@OK` для получения выходного параметра. При тестировании попробуйте использовать несколько разных параметров (чтобы узнать состав допустимых значений, воспользуйтесь процедурой `pSecurity_Fetch`). В следующем примере проверяется, имеет ли пользователь с кодом `118` административные привилегии на складе в городе Шарлотт:

```
DECLARE @OK BIT
EXEC p_SecurityCheck
    @ContactCode = 118,
    @LocationCode = Clt,
    @SecurityLevel = 3,
    @Approved = @OK OUTPUT
SELECT @OK
```

Результатом будет значение нуль — признак отсутствия полномочий.

Функция проверки полномочий

В функции проверки полномочий реализована та же логика, что и в аналогичной хранимой процедуре. Основным преимуществом функции является то, что ее можно использовать непосредственно в выражении `if`, не создавая локальную переменную для хранения выходного параметра. На входе функция принимает те же параметры, что и хранимая процедура; отличается только способ возврата результата — с помощью оператора `RETURN`, а не выходного параметра.

```
CREATE FUNCTION dbo.fSecurityCheck (
    @ContactCode CHAR(15),
    @LocationCode CHAR(15),
    @SecurityLevel INT)
RETURNS BIT
BEGIN
 DECLARE @ActualLevel INT,
         @Approved BIT
 SELECT @ActualLevel = SecurityLevel
  FROM dbo.Security
    JOIN dbo.Contact
      ON Security.ContactID = Contact.ContactID
    JOIN dbo.Location
      ON Security.LocationID = Location.LocationID
  WHERE ContactCode = @ContactCode
        AND LocationCode = @LocationCode
 IF @ActualLevel IS NULL
     OR @ActualLevel < @SecurityLevel
     OR @ActualLevel = 0
         SET @Approved = 0
  ELSE
         SET @Approved = 1
  RETURN @Approved
END
```

В следующем пакете продемонстрирован вызов функции для проверки прав доступа в теле хранимой процедуры. Если функция возвращает нуль, значит, у пользователя нет достаточных прав, и выполнение процедуры прекращается:

```
-- Проверка в теле процедуры
IF dbo.fSecurityCheck( 118, 'Clt', 3) = 0
  BEGIN
    RAISERROR('Недостаточные привилегии', 16,1)
    ROLLBACK TRANSACTION
    RETURN -100
  END
```

Использование учетной записи NT

Некоторые приложения создаются так, чтобы пользователь выполнял в них регистрацию, а затем эти регистрационные данные передавались в процедуры. В то же время, если экземпляр SQL Server использует аутентификацию NT, процедуры системы безопасности могут использовать и ее.

Вместо того чтобы запрашивать код контакта в качестве параметра, процедура или функция может идентифицировать текущего пользователя с помощью функции `suser_sname()` и извлекать регистрационные данные NT. Регистрационное имя (доменное или локальное) может быть добавлено в таблицу `Contact`. В качестве альтернативы может быть создана вторичная таблица, хранящая для каждого пользователя множество регистрационных записей. Некоторые глобальные сети требуют регистрации в разных регионах под разными доменными именами, так что создание отдельной таблицы `ContactLogin` является отличной альтернативой.

Следующая функция модифицирована для извлечения регистрационных данных NT конкретного пользователя, содержащихся в таблице `ContactLogin`:

```
SELECT suser_sname()
```

Результат получается следующий:

```
NOLI\Paul
```

Следующая инструкция создает вторичную таблицу, хранящую регистрационные данные:

```
CREATE TABLE dbo.ContactLogin(
  ContactLogin UNIQUEIDENTIFIER
    PRIMARY KEY NONCLUSTERED DEFAULT NewId(),
  ContactID UniqueIdentifier NOT NULL
    REFERENCES dbo.Contact ON DELETE CASCADE,
  NTLogin VARCHAR(100) )
```

Когда таблица создана, всего одна инструкция `INSERT` может заполнить в ней данными одну строку. После этого мы проверим факт вставки данных.

```
INSERT CONTACTLOGIN (ContactID, NTLogin)
SELECT ContactID, 'NOLI\Paul'
  FROM dbo.Contact
  WHERE ContactCode = 118

SELECT ContactCode, NTLogin
  FROM dbo.Contact
    JOIN ContactLogin
      ON Contact.ContactID = ContactLogin.ContactID
```

Будет получен следующий результат:

```
ContactCode NTLogin
----------- ---------
118         Paul/NOLI
```

Функция проверки полномочий была модифицирована так, чтобы использовать таблицу `ContactLogin` и возвращать список, соответствующий одной регистрационной записи NT. Так как код контакта больше не требуется, в инструкции `SELECT` отсутствует ссылка на таблицу `Contact`, и таблица `Security` объединяется непосредственно с таблицей `ContactLogin`:

```
CREATE FUNCTION dbo.fSecurityCheckNT (
    @LocationCode CHAR(15),
    @SecurityLevel INT)
RETURNS BIT
BEGIN
  DECLARE @ActualLevel INT,
          @Approved BIT
  SELECT @ActualLevel = SecurityLevel
    FROM dbo.Security
      JOIN dbo.Location
        ON Security.LocationID = Location.LocationID
      JOIN dbo.ContactLogin
        ON Security.ContactID = ContactLogin.ContactID
    WHERE NTLogin = suser_sname()
          AND LocationCode = @LocationCode
  IF @ActualLevel IS NULL
     OR @ActualLevel < @SecurityLevel
     OR @ActualLevel = 0
        SET @Approved = 0
  ELSE
        SET @Approved = 1
  RETURN @Approved
END
```

Тестирование новой функции ничем не отличается от тестирования предыдущей, однако на этот раз в функцию не передается код контакта — регистрационные данные извлекаются непосредственно из операционной системы:

```
IF dbo.fSecurityCheckNT('Clt', 3) = 0
  BEGIN
    RAISERROR('Недостаточные привилегии', 16,1)
    ROLLBACK TRANSACTION
     RETURN -100
  END
```

Данная функция не вернула ошибку, и я смог завершить выполнение процедуры.

Триггер проверки полномочий

Хранимые процедуры и функции проверки полномочий хорошо справляются с работой, когда включены в другие хранимые процедуры, о которых мы говорили в начале главы. В то же время для реализации системы безопасности на уровне строк в базах данных, рассчитанных на доступ из представлений, разовых запросов или прямых инструкций DML, необходимы триггеры. Триггер способен предотвратить обновление, однако не способен контролировать операции чтения. Если вам требуется и это, реализуйте чтение исключительно с помощью хранимых процедур.

Следующий триггер аналогичен описанной ранее функции проверки полномочий. Отличие состоит в том, что он допускает наличие множества заказов из потенциально большого количества регионов. Существующие в нем объединения должны сравнивать строки таблицы `[Order]` и их регионы с допустимыми для данных регионов полномочиями пользователей. Объединение может непосредственно связывать таблицу `ContactLogin` с таблицей `Security`. Так как это триггер вставки и обновления, пользователь с любым уровнем доступа ниже второго для любого заказа будет отвергаться с генерацией ошибки системы безопасности. При этом инструкция `ROLLBACK TRANSACTION` будет откатывать исходную инструкцию DML, вызвавшую триггер:

```
CREATE TRIGGER OrderSecurity ON [Order]
AFTER INSERT, UPDATE
AS
IF EXISTS (
SELECT *
  FROM dbo.Security
    JOIN dbo.ContactLogin
      ON Security.ContactID = ContactLogin.ContactID
    JOIN Inserted
      ON Inserted.LocationID = Security.LocationID
  WHERE NTLogin = suser_sname()
     AND SecurityLevel < 2 )
BEGIN
  RAISERROR('Недостаточно полномочий', 16,1)
  ROLLBACK TRANSACTION
END
```

Аудит изменений данных

Введение аудита повышает уровень целостности данных в базе. Полноценный журнал аудита может содержать массу информации, в том числе приведенную ниже.

- Все изменения данных в строках с момента их изначальной вставки.
- Все изменения данных, внесенные конкретным пользователем за последнюю неделю.
- Все изменения данных, внесенные на конкретной рабочей станции во время обеденного перерыва.
- Все изменения данных, внесенные клиентскими приложениями, отличающимися от стандартного.

Ведение журнала аудита данных решает множество существенных проблем как для администраторов баз данных, так и для конечных пользователей. Однажды моя консультационная компания разработала систему управления юридическими документами для компании из списка Fortune 100. Эту базу данных постоянно должна была пополнять регуляторными законами юридическая фирма. Однажды эта фирма выбилась из графика и объяснила это тем, что не могла в течение двух недель вводить данные из-за проблем с программным обеспечением. Когда мы вывели список из более 70000 изменений данных, зафиксированных за эти две недели в журнале аудита данных, вопрос поиска виновного разрешился сам собой.

Мне приходилось встречать в публикациях методы аудита данных, добавляющие в таблицу несколько столбцов или дублирующие таблицы для сохранения последних изменений. Ни один из этих методов нельзя назвать приемлемым. Частичный аудит и аудит последних изменений не имеют практически никакого смысла. Журнал аудита должен постоянно хранить все изменения данных, в противном случае любой, кто поймет его систему действия, сможет внести еще одно изменение и тем самым навсегда стереть исходное значение.

Журнал аудита

Задачей таблицы (или журнала) аудита (`Audit`) является организация единого места хранения всех изменений в базе данных. Предлагаемая ниже таблица журнала аудита способна хранить все некрупные изменения (не BLOB), вносимые во все таблицы. Столбец `Operation` хранит значения, определяющие характер инструкции DML, ответственной за изменения (`I`, `U` или `D`):

```
CREATE TABLE dbo.Audit (
  AuditID UNIQUEIDENTIFIER RowGUIDCol NOT NULL
    CONSTRAINT DF_Audit_AuditID DEFAULT (NEWID())
    CONSTRAINT PK_Audit PRIMARY KEY NONCLUSTERED (AuditID),
  AuditDate DATETIME NOT NULL,
  SysUser VARCHAR(50) NOT NULL,
  Application VARCHAR(50) NOT NULL,
  TableName VARCHAR(50)NOT NULL,
  Operation CHAR(1) NOT NULL,
  PrimaryKey VARCHAR(50) NOT NULL,
  RowDescription VARCHAR(50) NULL,
  SecondaryRow VARCHAR (50) NULL,
  [Column] VARCHAR(50) NOT NULL,
  OldValue VARCHAR(50) NULL,
  NewValue VARCHAR(50) NULL
  )
```

В столбце `PrimaryKey` хранится указатель на измененную строку, а в столбце `RowDescription` — осмысленное описание этой строки. Эти два столбца позволяют создать объединение с исходной таблицей и получить непосредственное описание операции. Значение столбца `PrimaryKey` имеет первостепенную важность, так как позволяет быстро отыскать все изменения в некоторой строке, независимо от того, как описание изменится со временем.

Фиксированный триггер журнала аудита

Наиболее простой метод аудита данных использует триггеры для всех таблиц, проверяющие все столбцы с помощью функции `updated()` и записывающие найденные изменения в таблицу аудита.

Инструкция `INSERT` объединяет таблицы `Inserted` и `Deleted`, чтобы корректно обработать все многострочные вставки и обновления. Это объединение является левым внешним, так что операции вставки, содержащие строки только в таблице `Inserted`, будут все равно записаны в журнал аудита. Данное объединение также ограничено условием, гарантирующим, что если многострочное обновление затрагивает столбцы только нескольких строк из этого множества, то только эти изменения будут записаны в таблицу аудита.

В приведенном ниже примере триггера журнала аудита отслеживаются изменения в таблице `Product` базы данных OBXKites.

```
CREATE TRIGGER Product_Audit
ON dbo.Product
AFTER Insert, Update
NOT FOR REPLICATION
AS
DECLARE @Operation CHAR(1)
IF EXISTS(SELECT * FROM Deleted)
  SET @Operation = 'U'
ELSE
  SET @Operation = 'I'

IF UPDATE(ProductCategoryID)
  INSERT dbo.Audit
     (AuditDate, SysUser, Application, TableName, Operation,
      PrimaryKey, RowDescription, SecondaryRow, [Column],
      OldValue, NewValue)
      SELECT GetDate(), suser_sname(), APP_NAME(), 'Product',
            @Operation, Inserted.ProductID, Inserted.Code,
            NULL, 'ProductCategoryID',
```

```
            OPC.ProductCategoryName, NPC.ProductCategoryName
        FROM Inserted
          LEFT OUTER JOIN Deleted
            ON Inserted.ProductID = Deleted.ProductID
              AND Inserted.ProductCategoryID
                  <> Deleted.ProductCategoryID
          LEFT OUTER JOIN dbo.ProductCategory OPC
            ON Deleted.ProductCategoryID
               = OPC.ProductCategoryID
          JOIN dbo.ProductCategory NPC
            ON Inserted.ProductCategoryID
               = NPC.ProductCategoryID

IF UPDATE(Code)
  INSERT dbo.Audit
     (AuditDate, SysUser, Application, TableName, Operation,
      PrimaryKey, RowDescription, SecondaryRow, [Column],
      OldValue, NewValue)
      SELECT GetDate(), suser_sname(), APP_NAME(),
           'Product', @Operation, Inserted.ProductID,
            Inserted.Code, NULL, 'Code',
            Deleted.Code, Inserted.Code
        FROM Inserted
          LEFT OUTER JOIN Deleted
            ON Inserted.ProductID = Deleted.ProductID
               AND Inserted.Code <> Deleted.Code

IF UPDATE(ProductName)
  INSERT dbo.Audit
    (AuditDate, SysUser, Application, TableName, Operation,
     PrimaryKey, RowDescription, SecondaryRow, [Column],
     OldValue, NewValue)
     SELECT GetDate(), suser_sname(), APP_NAME(),
         'Product', @Operation,
          Inserted.ProductID, Inserted.Code, NULL, 'Name',
          Deleted.ProductName, Inserted.ProductName
       FROM Inserted
         LEFT OUTER JOIN Deleted
           ON Inserted.ProductID = Deleted.ProductID
              AND Inserted.ProductName <> Deleted.ProductName

IF UPDATE(ProductDescription)
  INSERT dbo.Audit
    (AuditDate, SysUser, Application, TableName, Operation,
     PrimaryKey, RowDescription, SecondaryRow, [Column],
     OldValue, NewValue)
     SELECT GetDate(), suser_sname(), APP_NAME(), 'Product',
          @Operation, Inserted.ProductID, Inserted.Code,
          NULL, 'ProductDescription',
          Deleted.ProductDescription,
          Inserted.ProductDescription
        FROM Inserted
          LEFT OUTER JOIN Deleted
            ON Inserted.ProductID = Deleted.ProductID
                AND Inserted.ProductDescription
                  <> Deleted.ProductDescription

IF UPDATE(ActiveDate)
```

```
  INSERT dbo.Audit
     (AuditDate, SysUser, Application, TableName, Operation,
      PrimaryKey, RowDescription, SecondaryRow, [Column],
      OldValue, NewValue)
      SELECT GetDate(), suser_sname(), APP_NAME(), 'Product',
           @Operation, Inserted.ProductID, Inserted.Code,
           NULL, 'ActiveDate',
           Deleted.ActiveDate, Inserted.ActiveDate
        FROM Inserted
          LEFT OUTER JOIN Deleted
            ON Inserted.ProductID = Deleted.ProductID
               AND Inserted.ActiveDate != Deleted.ActiveDate

IF UPDATE(DiscontinueDate)
  INSERT dbo.Audit
     (AuditDate, SysUser, Application, TableName, Operation,
      PrimaryKey, RowDescription, SecondaryRow, [Column],
      OldValue, NewValue)
      SELECT GetDate(), suser_sname(), APP_NAME(), 'Product',
           @Operation, Inserted.ProductID, Inserted.Code,
            NULL, 'DiscontinueDate',
            Deleted.DiscontinueDate, Inserted.DiscontinueDate
        FROM Inserted
          LEFT OUTER JOIN Deleted
            ON Inserted.ProductID = Deleted.ProductID
               AND Inserted.DiscontinueDate
                   != Deleted.DiscontinueDate
```

Если установить этот триггер аудита, то следующий пакет вставки и обновления данных о товарах, использующий две инструкции DML, а также созданные ранее хранимые процедуры, позволит его проверить. Первый тест использует хранимую процедуру pProduct_AddNew:

```
EXEC pProduct_AddNew 'Kite', 200, 'The MonstaKite',
'Man what a big Kite!'

SELECT TableName, RowDescription, [Column], NewValue
FROM dbo.Audit
```

Будет получен следующий результат:

```
TableName RowDescription
                Column                  NewValue
--------- ----- ---------------------- --------------------
Product   200   ProductCategoryID       Kite
Product   200   Code                    200
Product   200   Name                    The MonstaKite
Product   200   ProductDescription      Man what a big Kite!
Product   200   ActiveDate              Mar 1 2002 1:35PM
Product   200   DiscontinueDate         NULL
```

Триггеры являются отличным средством ведения журнала аудита, так как регистрируют все изменения, даже выполняемые прямым вводом инструкций DML. В следующем примере мы не будем использовать хранимые процедуры и выполним обычную инструкцию UPDATE. После этого в журнале аудита мы увидим как старое, так и новое значение:

```
UPDATE dbo.Product
  SET ProductDescription = 'Biggie Sized'
  WHERE Code = 200
```

Следующий запрос извлечет для нас историю столбца ProductDescription для товара с кодом 200:

```
SELECT AuditDate, OldValue, NewValue
  FROM dbo.Audit
  WHERE TableName = 'Product'
     AND RowDescription = '200'
     AND [Column] = 'ProductDescription'
```

Получим следующий результат:

```
AuditDate                  OldValue         NewValue
-------------------------- ---------------- -------------
2002-03-01 13:35:17.093    NULL             Man what a
                                            big Kite!
2002-03-01 15:10:49.257    Man what a       Biggie Sized
                           big Kite!
```

Выполнение отката операций с помощью журнала аудита

В полноценной системе аудита пользователь может увидеть все изменения конкретной строки, внесенные с момента ее создания. В этом списке пользователь может выбрать любую операцию и откатить ее назад. Как только выбрана некоторая строка журнала аудита, откат операции сводится к обычному выполнению инструкции UPDATE, использующей данные журнала аудита.

В следующем примере продемонстрирован откат изменения, зарегистрированного в журнале аудита. Хранимая процедура pAudit_RollBack принимает в качестве параметра первичный ключ таблицы Audit, после чего создает динамическую инструкцию UPDATE, содержащую корректные таблицу, строку, столбец и восстанавливаемое значение:

```
CREATE PROCEDURE pAudit_RollBack (
   @AuditID UNIQUEIDENTIFIER)
AS
SET NoCount ON

DECLARE
  @SQLString NVARCHAR(4000),
  @TableName NVARCHAR(50),
  @PrimaryKey NVARCHAR(50),
  @Column NVARCHAR(50),
  @NewValue NVARCHAR(50)

SELECT
  @TableName = TableName,
  @PrimaryKey = PrimaryKey,
  @Column = [Column],
  @NewValue = OldValue
  FROM dbo.Audit
  WHERE AuditID = @AuditID

SET @SQLString =
  'UPDATE ' + @TableName
  + ' SET ' + @Column + ' = ''' + @NewValue +''''
  + ' WHERE ' + @TableName + 'ID = ''' + @PrimaryKey + ''''
EXEC sp_executeSQL @SQLString
Return
```

Имея в своем распоряжении данную хранимую процедуру, следующий сценарий имитирует логику, необходимую для отката операции обновления. Как вы помните, исходным описанием

товара с кодом 200 было "Man what a big Kite"; во время тестирования триггера журнала аудита оно было изменено на "Biggie Sized". Следующий сценарий находит строку, соответствующую этому изменению, и передает идентификатор GUID процедуре pAudit_RollBack, которая, в свою очередь, откатывает данное обновление:

```
DECLARE @AuditRollBack UNIQUEIDENTIFIER

SELECT @AuditRollBack = AuditID
  FROM dbo.Audit
  WHERE TableName = 'Product'
     AND RowDescription = '200'
     AND OldValue = 'Man what a big Kite!'

EXEC pAudit_RollBack @AuditRollBack

SELECT ProductDescription
  FROM dbo.Product
  WHERE Code = 200
```

Результат выполнения пакета:

```
ProductDescription
--------------------------
Man what a big Kite!
```

Эта процедура откатила только одно конкретное изменение, однако ее можно модифицировать так, чтобы можно было произвести откат к заданной точке во времени. Для этого из журнала аудита следует извлечь историю строки и в цикле выполнить откаты всех операций в обратном порядке.

Сложности аудита

Кроме дополнительного времени, необходимого на программирование, ведение аудита влечет за собой и другие сложности.



Спроектируйте всю базу данных и проверьте корректность ее схемы еще до реализации системы аудита. Внести изменения в схему будет намного сложнее, когда триггеры журнала аудита уже будут в наличии.

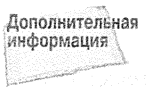
Аудит связанных данных

Наибольшую сложность представляет аудит связанных данных, таких как вторичные строки. Например, внесение изменений в строку заказа фактически изменяет весь заказ. Когда пользователь захочет просмотреть всю историю заказа, он должен увидеть все изменения, связанные с ним. Таким образом, внесение изменения в таблицу строк заказа должно быть записано и как изменение таблицы заказов, при этом номер измененной строки заказа должен быть отображен в столбце SecondaryRow.

Запись изменений, вносимых во внешние ключи, представляет собой еще один сложный аспект полноценного аудита. Пользователю вряд ли захочется видеть новый идентификатор GUID или значение идентичности. Если, к примеру, способ доставки товара изменился со "Slow Boat" на "Speedy Express", то триггер журнала аудита должен будет перейти по внешнему ключу и записать в журнал понятное пользователю значение. В приведенном ранее примере фиксированного триггера Product_Audit изменения, вносимые в столбец идентификатора категории товара, приводят к записи в журнал аудита названия этой категории.

Создаваемые и изменяемые даты

При использовании полноценной системы аудита из журнала можно извлечь даты создания и последнего изменения любой строки. На практике же в многопользовательской среде с большим потоком данных я настоятельно рекомендую выполнить денормализацию и вынести эти даты непосредственно в таблицу аудита.



В главе 23 был описан триггер, обновляющий созданные и обновленные столбцы и одновременно предотвращающий проблемы, связанные с рекурсией.

Аудит инструкций SELECT

Триггеры аудита данных ограничены инструкциями INSERT, UPDATE и DELETE. Для реализации аудита чтения данных используйте хранимую процедуру fetch. Используйте средства защиты информации SQL Server для ограничения доступа к таблицам; при этом создавайте систему таким образом, чтобы все операции чтения выполнялись с помощью хранимой процедуры или функции.

Аудит данных и вопросы безопасности

Еще одним вопросом, связанным с ведением журнала аудита, является обеспечение его защиты. Любой, кто имеет право чтения этого журнала, одновременно получает неявный доступ к информации всех обслуживаемых им таблиц. Если пользователи должны иметь возможность просматривать историю конкретных строк, используйте для доступа к данным аудита хранимую процедуру.

Аудит данных и производительность

Полноценный журнал аудита данных добавляет на систему определенную нагрузку. Любая вставка одной строки в таблицу с двадцатью столбцами повлечет за собой выполнение двадцати инструкций вставки в таблицу аудита. Чтобы несколько снизить дополнительную нагрузку, примите следующие меры.

- Ограничьте число индексов в таблице аудита.
- Размещайте таблицу аудита в отдельной файловой группе в отдельной дисковой подсистеме. Наличие отдельной файловой группы также упростит и выполнение резервного копирования.
- Используя фиксированные триггеры аудита, ограничьтесь только теми столбцами, которые требуют высокого уровня целостности данных.

Динамические триггеры и процедуры журнала аудита

Проведя несколько месяцев за написанием фиксированных триггеров аудита для проекта с сотнями таблиц, я однозначно пришел к решению использовать динамическую систему аудита.

Динамический журнал аудита требует небольших триггеров в каждой из таблиц. Эти триггеры всего лишь копируют таблицы Inserted и Deleted во временные, после чего передают несколько переменных хранимой процедуре, на которую возложена основная работа. Та проверяет двоичное значение Columns_Updated, после чего определяет правильное значение, используемое вместо пустого, а затем генерирует динамическую инструкцию SQL, выполняющую запись в журнал аудита.

Вот в чем состоит ключевой момент. Для выполнения динамической инструкции SQL требуется вызов системной хранимой процедуры sp_execSQL, которая функционирует во вложенном пакете T-SQL. Область определения таблиц Inserted и Deleted ограничена триггером — эти таблицы недоступны вызывающей процедуре или динамической инструкции SQL. Именно поэтому и создаются временные таблицы, которые уже и передают изменения в хранимую процедуру и впоследствии в динамический запрос SQL. Область определения временной таблицы включает в себя вызывающую хранимую процедуру и исполняемые модули, что и делает возможным создание динамического триггера аудита.

Естественно, любой метод имеет свои достоинства и недостатки. С положительной стороны метод "триггер/хранимая процедура/динамический SQL" чрезвычайно легко реализовать. Он способен работать практически с любыми таблицами. В то же время существуют и некоторые ограничения этой версии динамического триггера аудита.

- Для передачи значений таблиц Inserted и Deleted в хранимую процедуру он использует временные таблицы, поэтому его трудно назвать самым быстрым методом. К тому же временные таблицы создаются с помощью синтаксиса SELECT...INTO, что связано с дополнительными вопросами производительности и блокировки.
- Текущий динамический журнал аудита не отслеживает автоматически связанные данные и вторичные таблицы. Он также не способен вести аудит таблиц с составными первичными ключами.
- Он не ведет аудит таблиц со столбцами BLOB (типов image, text и ntext), поскольку их нельзя извлечь из таблиц Inserted и Deleted.

Исходя из этих ограничений, рекомендуется использовать метод динамического аудита только для таблиц, которые не обновляются часто, а также для баз данных, которые при использовании этого метода имеют нормальные показатели производительности. Для обеспечения высокой производительности лучше использовать метод фиксированных триггеров аудита, в которых "прошит" состав столбцов; в то же время этот метод предполагает намного больше программирования и сложности в сопровождении.

Принимая во внимание вышесказанное, приведем пример программного кода ведения динамического журнала аудита:

```
/*
Создание динамических журнала и триггера аудита.
Пол Нильсен
В этом примере к таблицам Customers и Products
базы данных Northwind добавляется
динамический триггер аудита.
Версия 1.1 - 6 августа 2001
*/
USE Northwind
-------------------------------------------
--Создание таблицы для хранения журнала аудита
IF Exists (SELECT * FROM sysobjects WHERE NAME = 'Audit')
  DROP TABLE Audit
Go

CREATE TABLE dbo.Audit (
  AuditID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL
    CONSTRAINT DF_Audit_AuditID DEFAULT (NEWID())
    CONSTRAINT PK_Audit PRIMARY KEY NONCLUSTERED (AuditID),
  AuditDate DATETIME NOT NULL,
  SysUser VARCHAR(50) NOT NULL,
  Application VARCHAR(50) NOT NULL,
```

```
  TableName VARCHAR(50)NOT NULL,
  Operation CHAR(1) NOT NULL,
  PrimaryKey VARCHAR(50) NOT NULL,
-- RowDescription VARCHAR(50) NULL,
  SecondaryRow VARCHAR(50) NULL,
  [Column] VARCHAR(50) NOT NULL,
  OldValue VARCHAR(50) NULL,
  NewValue VARCHAR(50) NULL
  )
GO

-------------------------------------------------------------
-- Создание функции для имитации значения Columns_Updated()
IF EXISTS (SELECT *
             FROM sysobjects
             WHERE NAME = 'GenColUpdated')
  DROP FUNCTION GenColUpdated
Go
CREATE FUNCTION dbo.GenColUpdated
    (@Col INT, @ColTotal INT)
RETURNS INT
AS
BEGIN
-- Эта функция имитирует Columns_Updated()
  DECLARE
    @ColByte INT,
    @ColTotalByte INT,
    @ColBit INT
-- Вычисление позиций байтов
  SET @ColTotalByte = 1 + ((@ColTotal-1) /8)
  SET @ColByte = 1 + ((@Col-1)/8)
  SET @ColBit = @col - ((@colByte-1) * 8)
-- генерация значения Columns_Updated() для данного столбца
  RETURN
  POWER(2, @colbit + ((@ColTotalByte-@ColByte) * 8)-1)
END
go

-------------------------------------------------
-- Создание хранимой процедуры динамического аудита
IF EXISTS (SELECT * FROM SysObjects WHERE NAME = 'pAudit')
  DROP PROC pAudit
Go
CREATE PROCEDURE pAudit (
    @Col_Updated VARBINARY(1028),
    @TableName VARCHAR(100),
    @PrimaryKey SYSNAME)
AS
SET NoCount ON
DECLARE
  @ColTotal INT,
  @ColCounter INT,
  @ColUpdatedTemp INT,
  @ColName SYSNAME,
  @BlankString CHAR(1),
  @SQLStr NVARCHAR(1000),
  @ColNull NVARCHAR(50),
  @SysUser NVARCHAR(100),
```

```
  @ColumnDataType INT,
  @IsUpdate BIT,
  @tempError INT
SET @SysUser = suser_sname()
SET @BlankString = ''
-- Инициализация переменных столбцов
SELECT @ColCounter = 0
SELECT @ColTotal = Count(*)
  FROM SysColumns
    JOIN SysObjects
      ON SysColumns.id = SysObjects.id
  WHERE SysObjects.name = @TableName
-- Установка флага IsUpdated
IF EXISTS(SELECT * FROM #tempDel)
  SELECT @IsUpdate = 1
ELSE
  SELECT @IsUpdate = 0
-- Обновления столбцов
WHILE ((SELECT @ColCounter) != @ColTotal)
   BEGIN
     SELECT @ColCounter = @ColCounter + 1
     SET @ColUpdatedTemp
         = dbo.GenColUpdated(@ColCounter,@ColTotal)
-- битовый AND между обновляемыми битами
-- и битом выбранного столбца
     IF (@Col_Updated & @ColUpdatedTemp) = @ColUpdatedTemp
       BEGIN
         SET @ColNull = null
         SELECT
            @ColName = SysColumns.[name],
-- получение имени столбца и типа данных
            @ColumnDataType = SysColumns.xtype
           FROM SysColumns
            JOIN SysObjects
              ON SysColumns.id = SysObjects.id
           WHERE SysObjects.[NAME] = @TableName
                 and SysColumns.ColID = @ColCounter
         IF @ColName NOT IN ('Created', 'Modified')
           BEGIN
-- текстовые столбцы
             IF @ColumnDataType IN
               ( 175, 239, 99, 231, 35, 231, 98, 167 )
                  SET @ColNull = ''''''
-- числовые и битовые столбцы
             ELSE IF @ColumnDataType IN
                  ( 106, 62, 56, 60, 108, 59, 52, 122, 104 )
                    SET @ColNull = '0'
-- столбцы дат
                  ELSE IF @ColumnDataType IN ( 61, 58 )
                         SET @ColNull = '''1/1/1980'''
-- столбцы уникальных идентификаторов
                       ELSE IF @ColumnDataType IN ( 36 )
                            SET @ColNull = ''''''
           IF @ColNull IS NOT NULL
             BEGIN
               IF @IsUpdate = 1
-- нужно скорректировать отступы
SET @SQLStr =
```

```
  ' Insert Audit(TableName, PrimaryKey, SysUser, [Column],'
  +' AuditDate, Application, OldValue, NewValue,Operation)'
  +' Select '''+ @TableName + ''',
  #tempIn.['+ @PrimaryKey + '],
  ''' + @SysUser + ''', ' +
  '''' + @ColName + ''', GetDate(), App_Name(),' +
  ' IsNull(convert(nvarchar(100),
  #tempDel.[' + @ColName + ']),''<null>''), ' +
  ' IsNull(convert(nvarchar(100),
  #tempIn.[' + @ColName + ']),''<null>''),''U''' +
  ' From #tempIn' +
  ' Join #tempDel' +
  ' On #tempIn.['+ @PrimaryKey + ']
  = #tempDel.['+ @PrimaryKey + ']' +
  ' AND isnull(#tempIn.' + @ColName + ',' + @ColNull + ')
  != isnull(#tempDel.' + @ColName + ',' + @ColNull + ')'
  + ' Where Not (#tempIn.[' + @ColName + '] Is Null
  and #tempDel.[' + @ColName + '] Is Null)'
ELSE -- Вставка
SET @SQLStr =
  ' Insert Audit(TableName, PrimaryKey, SysUser, [Column],'
  +' AuditDate, Application, OldValue, NewValue,Operation)'
  +' Select '''+ @TableName + ''',#tempIn.['+ @PrimaryKey
  + '], ''' + @SysUser + ''', ' +
  '''' + @ColName + ''', GetDate(), App_Name(),' +
  ' Null, ' +
  ' IsNull(convert(nvarchar(100),
  #tempIn.[' + @ColName +']),''<null>''),''I''' +
  ' From #tempIn' +
  ' Where Not (#tempIn.[' + @ColName + '] Is Null)'

EXEC sp_executesql @SQLStr
SET @TempError = @@Error
IF @TempError <> 0
  BEGIN
-- включайте откат, только если хотите,
-- чтобы ошибка при записи в журнал аудита
-- отменяла операцию модификации данных
-- Откат
    RAISERROR ('Ошибка записи в журнал аудита', 15, 1)
END
END END END END RETURN
Go

-------------------------------------------------------------
-- Примеры табличных триггеров
-- Они должны быть добавлены ко всем таблицам
-- с настройками имени таблицы и первичного ключа
-- Триггер для таблицы Products
IF EXISTS (SELECT *
             FROM sysobjects
             WHERE NAME = 'Products_Audit')
  DROP TRIGGER Products_Audit
Go
CREATE TRIGGER Products_Audit
  ON dbo.Products
  AFTER Insert, Update
  NOT FOR REPLICATION
```

```
AS
DECLARE
  @Col_Updated VARBINARY(1028),
  @TableName VARCHAR(100),
  @PrimaryKey SYSNAME
SET NoCount ON
-- Настройка данных для аудита
-- Установка имени таблицы
SET @TableName = 'Products'
-- Настройка столбца, идентифицирующего строку
SET @PrimaryKey = 'ProductID'
SET @Col_Updated = Columns_Updated()
SELECT * INTO #TempIn FROM Inserted
SELECT * INTO #TempDel FROM Deleted
-- Вызов хранимой процедуры аудита
EXEC pAudit @Col_Updated, @TableName, @PrimaryKey
Go

-----------------------------------------
-- Триггер для таблицы Customers
IF EXISTS (SELECT *
             FROM SysObjects
             WHERE [NAME] = 'Customers_Audit')
  DROP TRIGGER Customers_Audit
Go
CREATE TRIGGER Customers_Audit
  ON dbo.Customers
  AFTER Insert, Update
  NOT FOR REPLICATION
AS
DECLARE
  @Col_Updated VARBINARY(1028),
  @TableName VARCHAR(100),
  @PrimaryKey SYSNAME
SET NoCount ON
SET @TableName = 'Customers'
SET @PrimaryKey = 'CustomerID'
SET @Col_Updated = Columns_Updated()
SELECT * INTO #TempIn FROM Inserted
SELECT * INTO #TempDel FROM Deleted
EXEC pAudit @Col_Updated, @TableName, @PrimaryKey
```



В сценарии `DynamicAudit.sql`, который можно загрузить с сайта книги (`www.SQLServerBible.com`), содержится несколько примеров вставки и удаления, а также запросов извлечения данных из таблицы аудита, при объединении ее с таблицей товаров.

Обработка транзакций консолидации

Хранимые процедуры идеально подходят для работы с денормализованными консолидированными данными. Типичным примером такой задачи является складская система, которая записывает все операции в специальную таблицу, вычисляет остатки, а затем записывает полученные значения в главную складскую таблицу с целью повышения производительности.

Чтобы защитить целостность такой складской таблицы, с помощью триггеров должны поддерживаться следующие правила.

- Складская таблица не должна обновляться никаким процессом, кроме триггеров таблицы складских операций. Любая попытка непосредственной записи в складскую таблицу должна немедленно отражаться в таблице операций как корректировка вручную.
- При вставке любой записи в таблицу операций должны немедленно корректироваться остатки в складской таблице.
- Таблица складских операций не должна допускать обновления существующих строк.

Учебную базу данных OBXKites можно рассматривать как пример упрощенной складской системы. Чтобы продемонстрировать обработку транзакций консолидации, реализуем описанные выше правила с помощью следующих триггеров.

С целью тестирования мы вначале создадим пример складской позиции с помощью следующего сценария:

```
USE OBXKites

DECLARE
  @ProdID UniqueIdentifier,
  @LocationID UniqueIdentifier
SELECT @ProdID = ProductID
  FROM dbo.Product
  WHERE Code = 1001
SELECT @LocationID= LocationID
  FROM dbo.Location
  WHERE LocationCode = 'CH'

INSERT dbo.Inventory (ProductID, InventoryCode, LocationID)
  VALUES (@ProdID,'A1', @LocationID)

SELECT Product.Code, InventoryCode, QuantityOnHand
  FROM dbo.Inventory
    JOIN dbo.Product
      ON Inventory.ProductID = Product.ProductID
```

Получим следующий результат:

```
Code     InventoryCode   QuantityOnHand
-------- --------------- --------------
1001     A1              0
```

Триггер таблицы складских операций

Триггер складской операции использует итоговую функцию для подсчета текущих остатков для таблицы `Inventory`. Как только в таблицу `InventoryTransaction` вставляется какая-либо строка, этот триггер обновляет таблицу `Inventory`. Объединение между таблицей образов `Inserted` и `Inventory` позволяет триггеру обслуживать многострочные вставки:

```
CREATE TRIGGER InvTrans_Aggregate
  ON dbo.InventoryTransaction
  AFTER Insert
AS
UPDATE dbo.Inventory
  SET QuantityOnHand
       = Inventory.QuantityOnHand + Inserted.Value
  FROM dbo.Inventory
    JOIN Inserted
      ON Inventory.InventoryID = Inserted.InventoryID
Return
```

Следующий пакет тестирует триггер InvTrans_Aggregate, вставляя складскую операцию и обозревая результаты в обеих рассматриваемых таблицах:

```
INSERT InventoryTransaction (InventoryID, Value)
  SELECT InventoryID, 5
     FROM dbo.Inventory
     WHERE InventoryCode = 'A1'

INSERT InventoryTransaction (InventoryID, Value)
  SELECT InventoryID, -3
     FROM dbo.Inventory
     WHERE InventoryCode = 'A1'

INSERT InventoryTransaction (InventoryID, Value)
  SELECT InventoryID, 7
     FROM dbo.Inventory
     WHERE InventoryCode = 'A1'
```

Следующий запрос извлекает данные из таблицы InventoryTransaction:

```
SELECT InventoryCode, Value
  FROM dbo.InventoryTransaction
    JOIN dbo.Inventory
      ON Inventory.InventoryID
         = Inventorytransaction.InventoryID
```

Получим следующий результат:

```
InventoryCode   Value
--------------- ------
A1              5
A1              -3
A1              7
```

Триггер InvTrans_Aggregate должен был сформировать корректное значение остатков в таблице Inventory. Следующий запрос поможет нам проверить, все ли он сделал правильно:

```
SELECT Product.Code, InventoryCode, QuantityOnHand
  FROM dbo.Inventory
    JOIN dbo.Product
       ON Inventory.ProductID = Product.ProductID
```

Получим следующий результат:

```
Code     InventoryCode   QuantityOnHand
-------- --------------- --------------
1001     A1              9
```

Триггер складской таблицы

С количественными значениями таблицы Inventory, в принципе, нельзя работать напрямую. Любая корректировка количества товаров на складе должна проходить через таблицу складских операций. В то же время некоторым пользователям может понадобиться вручную внести коррективы в складские остатки. Лучшим выходом из данной ситуации является использование серверного программного кода для формирования корректной операции, независимо от метода, использованного пользователем. Таким образом, триггер складской таблицы должен перенаправлять непосредственные обновления таблицы Inventory в таблицу InventoryTransaction, в то же время позволяя триггеру InvTrans_Aggregate обновлять таблицу Inventory.

Естественно, лучше, чтобы триггер мог обслуживать многострочные обновления. Главной целью является отмена исходной инструкции UPDATE при одновременном сохранении достаточного количества данных для вставки операций в таблицу InventoryTransaction.

Откат такой операции UPDATE невозможен, поскольку это сотрет данные в таблицах образов Inserted и Deleted, равно как и во временных таблицах, созданных в триггере. В то же время значения не могут быть сохранены в локальных переменных, так как такая техника не позволит обслуживать многострочные операции.

Решение состоит в том, чтобы переписать еще не обновленные значения из таблицы Deleted назад в таблицу Inventory. Тогда разницу между полями QuantityOnHand таблиц Deleted и Inventory можно записать в таблицу InventoryTransaction как отдельную ручную операцию.

Данный триггер выполняется только в том случае, когда обновляется значение поля QuantityOnHand и триггер вызывается пользовательской инструкцией DML. Если же данное поле обновляется ранее рассмотренным триггером InvTrans_Agregate, то уровень вложенности (т.е. результат функции NestLevel()) будет больше единицы. Ниже приведен текст триггерного решения складской задачи со стороны таблицы Inventory.

```
CREATE TRIGGER Inventory_Aggregate
ON Inventory
AFTER UPDATE
AS
-- Перенаправление прямых обновлений
If Trigger_NestLevel() = 1 AND Update(QuantityOnHand)
  BEGIN
    UPDATE Inventory
      SET QuantityOnHand = Deleted.QuantityOnHand
      FROM Deleted
        JOIN dbo.Inventory
          ON Inventory.InventoryID = Deleted.InventoryID
    INSERT InventoryTransaction
      (Value, InventoryID)
      SELECT
          Inserted.QuantityOnHand - Inventory.QuantityOnHand,
          Inventory.InventoryID
        FROM dbo.Inventory
          JOIN Inserted
            ON Inventory.InventoryID = Inserted.InventoryID
END
```

Чтобы продемонстрировать работу триггера, следующая инструкция UPDATE попытается изменить складские остатки с 9 на 10. Новый триггер Inventory_Aggregate перехватит эту инструкцию и восстановит значение остатков 9, в то же время создавая новую операцию +1 в таблице InventoryTransaction. (Если таблица операций имеет столбцы типа операции и комментарии, то можно пометить эту операцию как ручную.) После этого триггер InvTrans_Aggregate таблицы InventoryTransaction увидит инструкцию INSERT и скорректирует поле QuantityOnHand таблицы Inventory в значение 10.

```
-- Тестирование триггера
Update dbo.Inventory
  SET QuantityOnHand = 10
  Where InventoryCode = 'A1'
```

Сформировав ручную операцию, с помощью следующего запроса проверим содержимое таблицы InventoryTransaction:

```
SELECT InventoryCode, Value
  FROM dbo.InventoryTransaction
```

```
    JOIN dbo.Inventory
      ON Inventory.InventoryID
         = Inventorytransaction.InventoryID
```

Как мы видим, ручное увеличение количества товара на единицу было успешно записано:

```
InventoryCode    Value
---------------- ------
A1               5
A1               -3
A1               7
A1               1
```

А теперь посмотрим, скорректировал ли триггер `InvTrans_Aggregate` значение остатков в таблице `Inventory`:

```
SELECT Product.Code, InventoryCode, QuantityOnHand
  FROM dbo.Inventory
    JOIN dbo.Product
      ON Inventory.ProductID = Product.ProductID
```

Получим следующий результат:

```
Code    InventoryCode    QuantityOnHand
------- ---------------- --------------
1001    A1               10
```

Логическое удаление данных

Для усиления поддержки целостности данных многие разработчики препятствуют физическому удалению информации — они заменяют его *логическим удалением*. Наиболее распространенным методом является создание специального битового столбца флага удаления. Когда пользователь удаляет некоторую строку из клиентского приложения, триггер маркирует ее как удаленную, устанавливая флаг в значение `true`. Флаг логического удаления может быть реализован несколькими способами.

- Его установку может выполнять само клиентское приложение.
- Эта задача может быть возложена на хранимую процедуру.
- Триггер `INSTEAD OF` может перехватывать инструкции `DELETE` и вместо физического удаления устанавливать флаг.



Флаг логического удаления не стоит рассматривать как какой-то продвинутый метод. Еще СУБД DBASE III использовала этот флаг для маркировки удаленных строк, а физическое удаление осуществляла только при операции сжатия файла.

Способность логического удаления является неплохим методом, применяемым в хорошо продуманных базах данных. Однако рекомендуется рассматривать этот метод при проектировании баз данных в последнюю очередь, поскольку сама реализация занимает массу времени, к тому же любое изменение схемы базы может разрушить систему логического удаления. В настоящем разделе мы рассмотрим систему логического удаления строк в одной таблице. При этом будут рассмотрены проблемы, связанные с ней, и предложены пути их обхода.

Триггеры логического удаления

Триггеры `INSTEAD OF` реализуют систему логического удаления на уровне таблицы и гарантируют ее постоянную функциональность. Эти триггеры преследуют две цели: преобра-

зование физического удаления в логическое и реализацию некоторого метода реального физического удаления строк.

Предлагаемый триггер позволяет пользователю sa физически удалить любую строку, таким образом предоставляя метод очистки базы данных. Триггер INSTEAD OF не имеет рекурсивного действия, так что содержащаяся в нем инструкция DELETE не приведет к повторному его вызову и будет выполнена. Первая команда в данном примере изменяет структуру таблицы Product, добавляя в нее битовый флаг IsDeleted:

```
ALTER TABLE Product
  ADD IsDeleted BIT NOT NULL DEFAULT 0

CREATE Trigger Product_LogicalDelete
On dbo.Product
INSTEAD OF Delete
AS
IF (suser_sname() = 'sa')
  BEGIN
    PRINT 'физическое удаление'
    DELETE FROM dbo.Product
      FROM dbo.Product
        JOIN Deleted
          ON Product.ProductID = Deleted.ProductID
  END
ELSE
  BEGIN
    PRINT 'логическое удаление'
    UPDATE Product
      SET IsDeleted = 1
      FROM dbo.Product
        JOIN Deleted
          ON Product.ProductID = Deleted.ProductID
  END
```

Для тестирования триггера логического удаления мы выполним следующий запрос, который удаляет строки из таблицы Product. Следует особо отметить, что я зарегистрировался в системе как Noli\Paul:

```
DELETE Product
  WHERE Code = '1053'
```

Получим следующий результат:

```
Логическое удаление
```

Теперь посмотрим на флаг логического удаления:

```
SELECT Code, IsDeleted
  FROM dbo.Product
  WHERE Code = 1053
```

Будет получен следующий результат:

```
Code            IsDeleted
--------------- ---------
1053            1
```

Подключившись заново как пользователь sa, я снова попытался выполнить ту же инструкцию DELETE:

```
DELETE Product
  WHERE Code = '1053'
```

Результат будет следующий:

```
Физическое удаление
(1 row(s) affected)
(1 row(s) affected)
```

Первая строка `(1 row(s) affected)` явилась результатом исходной инструкции `DELETE`. Несмотря на то что она была перехвачена триггером `INSTEAD OF` и исходное удаление игнорировалось, инструкция отрапортовала об успехе операции. Вторая строка `(1 row(s) affected)` стала результатом выполнения инструкции `DELETE` уже в триггере `Product_LogicalDelete`. На самом деле именно эта инструкция и удалила физически строку таблицы.

Восстановление логически удаленных строк

Прежде чем быть физически удаленной пользователем `sa`, строка может быть восстановлена установкой значения столбца `IsDeleted` в значение `false`. Если в базе данных реализован метод обеспечения безопасности данных на уровне строк, описанный ранее в этой главе, то триггер `AFTER UPDATE` может проверить наличие у пользователя административных привилегий на данную строку и восстановить значение столбца `IsDeleted` в `0`.

Фильтрация логически удаленных строк

Главной проблемой в реализации схемы логического удаления является то, что любая инструкция `SELECT` должна учитывать флаг `IsDelete`, в противном случае удаленные данные могут навредить результату операции. Лучшей гарантией того, что клиентские приложения извлекают только текущие и корректные данные, является использование для отбора данных хранимых процедур.

В то же время, когда пользователь пытается выполнить написанный вручную запрос, нет никакой гарантии того, что он учтет флаг `IsDeleted`. Здесь лучше предоставить ему только представления, а также табличные пользовательские функции для извлечения данных, при этом ограничив круг лиц, допущенных к ним, с помощью средств системы безопасности SQL Server.

Каскадное логическое удаление

Это именно тот случай, когда логическое удаление данных превращается в сплошной кошмар. Если первичная строка физически удаляется, то вторичная строка, не имеющая смысла без первичной, также должна быть удалена. Спрашивается, можно ли в логическом удалении также реализовать принцип каскадности?

Если логически удаляется заказ, то связанные с ним строки могут быть также логически удалены при следующих операциях чтения или записи. Оба этих метода имеют свои проблемы.

Реализация каскадного удаления во время чтения

Если заказ логически удален, то его строки должны быть исключены из любых вычислений, связанных с открытыми заказами. Одним из возможных методов реализации является объединение с таблицей заказов и включение в предложение `WHERE` поля `Order.IsDeleted`. Эта задача может стать достаточно сложной, если логическая связь распространяется на множество уровней. Из собственного опыта знаю, что реализация каскадного логического удаления в операциях чтения может напрочь убить производительность, так как число таблиц, объединений и предложений `WHERE` возрастает экспоненциально, чтобы покрыть всевозможные логические комбинации.

Реализация каскадного удаления во время записи

Если вторичные строки логически удаляются в процессе логического удаления первичных, то последующим операциям чтения нет никакой необходимости проверять первичную таблицу на предмет логически удаленных строк. Проблемы возникают тогда, когда вторичные или первичные строки были удалены самостоятельно.

Здесь можно использовать два флага, один из которых является маркером логического удаления строки, а второй — маркером каскадного логического удаления. Даже если сегодняшнее решение вызовет проблемы только в будущем. это все равно вызывает головную боль. Предположим, что заказ логически удален и логическая операция удаления направлена в таблицу строк заказа. Одна из строк заказа указывает на товар, который также логически удаляется. Впоследствии при восстановлении заказа все его строки будут также логически восстановлены, за исключением той, которая указывает на логически удаленный товар.

Существуют два возможных решения этой проблемы. Первое из них заключается в добавлении флага логического каскадного удаления для каждого отношения внешнего ключа таблицы. Это сделает программный код более запутанным (лично мне это решение не нравится).

Второе решение заключается в использовании единого флага каскадного удаления и создании более сложной системы восстановления, которая проверяет каждое из отношений первичной таблицы, прежде чем восстановить строку. Несмотря на то что этот метод предполагает очень много работы, это решение я считаю лучшим.

Логическое удаление и ссылочная целостность

Реализация полноценной системы логического удаления поднимает еще одну проблему — поддержания ссылочной целостности. Этот принцип будет нарушаться при любой попытке обращения к строке, которая была логически удалена. Механизму декларативной ссылочной целостности SQL Server не важно, была ли логически удалена строка, — ему важно, чтобы она физически присутствовала в таблице.

База данных с системой логического удаления требует сложного триггера поддержания ссылочной целостности, который проверяет не только существование значения первичного ключа главной таблицы, но и флаг `IsDeleted`.

Маркировка неактивности

Системы, в которых реализован механизм логического удаления, часто имеют еще один уровень, определяющий бездействие строки, например *флаг активности* или устарелости данных. Эти флаги позволяют пользователю маркировать строки как недействительные без необходимости удаления данных. Например, в лаборатории R&D, занимающейся разработкой новых материалов, исследователям не хочется постоянно сталкиваться с тысячами устаревших версий формул. В то же время они не хотят удалять и сами существующие данные. Маркировка формулы как недействительной позволяет скрыть ее, однако в случае необходимости — извлечь.

Архивирование данных

Старые данные часто не нужны в повседневной работе, поэтому их можно безболезненно заархивировать или переместить в другую базу данных. Проще всего заархивировать данные, переместив их в отдельную таблицу, имеющую ту же структуру, что и исходная. При этом таблица может находиться как в текущей базе данных, так и в другой.

Архивирование данных является отличной альтернативой логическому удалению. При этом совершенно исчезают проблемы поддержания ссылочной целостности и каскадных логических удалений.

Хранимая процедура может запросто реализовать вставку данных в архивную таблицу и удаление их из текущей. Приведем пример:

```
CREATE PROCEDURE pProduct_Archive (
    @Code CHAR(15) )
AS
SET NoCount ON
BEGIN TRANSACTION
  INSERT Product_Archive
    SELECT *
      FROM dbo.Product
      WHERE Code = @Code
  IF @@ERROR <> 0
    BEGIN
       ROLLBACK TRANSACTION
       RETURN
    END
  DELETE dbo.Product
    WHERE Code = @Code
  IF @@ERROR <> 0
    BEGIN
      ROLLBACK TRANSACTION
    END
  COMMIT TRANSACTION
RETURN
```

Хранимой процедуре, скорее всего, придется перемещать строки нескольких таблиц. К примеру, при архивировании заказов следует переместить строки таблицы `[Order]` и таблицы `OrderDetail`.

Резюме

В этой главе мы заставили T-SQL решить множество сложных вопросов. Сложную обработку данных и поддержание правил бизнес-логики лучше всего реализовать с помощью серверного программного кода. Только когда правило реализовано на сервере, гарантируется его стопроцентное корректное выполнение. Реализацию правил за пределами сервера, скорее всего, можно рассматривать как исключение. Серверный программный код идеально подходит для процедур `INSERT`, `UPDATE`, `DELETE` и `FETCH`, сложных бизнес-правил, поддержания сложной ссылочной целостности, ведения журналов аудита данных и логического удаления.

В следующей главе мы продолжим изучение темы программирования на T-SQL и рассмотрим решение еще более приближенных к практике задач.

ГЛАВА

25

Расширяемость с помощью уровня абстракции данных

В этой главе...

Гибкость баз данных

Укрепление баз данных

Хранимые процедуры `AddNew`, `Fetch`, `Update` и `Delete`

Существует довольно распространенная ситуация: схема базы данных проектировалась много лет или даже десятилетий назад, и несмотря на то, что все в организации хотят улучшить ее структуру, это настолько рискованно и влечет за собой такие затраты, что никто за это не берется. А причина одна — существует слишком много непосредственных подключений к таблицам данных. Самые большие затраты влечет за собой не создание, а поддержка программного обеспечения. Что же касается баз данных, то прямые подключения в динамических запросах SQL к таблицам поддерживать труднее всего. Как в принципе домино: любое изменение в схеме отражается на сбоях в программах, отчетах и пакетах DTS. Обычная инструкция `SELECT` практически берет в заложники базу данных, и любая надежда на расширение рассеивается, как дым. Главное достоинство SQL — простота создания запросов к данным — становится его слабым звеном. Ответ лежит в *уровнях абстракции*.

Уровни абстракции окружают нас повсюду. Любой интерфейс пользователя, скрывающий за своей простотой всю сложность обработки данных, представляет собой уровень абстракции. Хорошим примером уровня абстракции, с которым нам приходится сталкиваться ежедневно, являются стандартные элементы управления автомобилем. Привычный руль позволяет нам управлять машиной, не задумываясь о том, как он соединен с передними колесами. Педаль тормоза позволяет замедлить движение автомобиля, совершенно не вдаваясь в тонкости функционирования тормозной системы. То же можно сказать и о педали газа, рычаге переключения передач, фарах и т.д. Привычные элементы управления являются уровнем абстракции механической работы автомобиля.

Чтобы избежать создания хрупкой базы данных или исправить уже существующую, следует использовать уровень абстракции — единую точку доступа к данным, служащую ло-

гическим буфером обмена. Это своеобразный метод сокрытия данных, когда доступ к базе осуществляется посредством некоего промежуточного звена. База данных или серверный программный код может со временем измениться, в то же время доступ к данным останется постоянным. Грамотно спроектированный уровень абстракции данных и гибкая схема базы — вот два ключа для создания расширяемой базы данных.

Принципы информационной архитектуры, сформулированные в главе 1, поддерживают концепцию уровней абстракции данных. Главный принцип гласит, что *данные должны быть доступны как в настоящий момент, так и в будущем*. Если база данных должна выжить в будущем (а многие базы намного переживают свои исходные интерфейсы пользователя), она должна быть расширяемой. А лучший способ гарантировать расширяемость — это использовать уровень абстракции данных.

Знатоки уже утомились доказывать, что архитектуры "клиент/сервер" и многоуровневая не справляются с работой и что единственный ответ лежит в использовании архитектуры, ориентированной на службы (SOA). Мне кажется, что это не совсем так. Базы данных, созданные без использования уровня абстракции данных, могут действительно не работать, как нужно. В то же время при использовании уровня абстракции данных будут отлично работать все виды архитектур: и многоуровневая, и "клиент/сервер", и SOA.

Уровень абстракции данных может существовать как в базе данных (в виде хранимых процедур, представлений и функций), так и в приложении (с помощью использования .NET и ADO). В то же время не смешивайте эти два подхода.

Лично я предпочитаю создавать уровень абстракции в базе данных, и на это есть две причины. Во-первых, это позволяет расположить программный код проверки и все классификаторы настолько близко к данным, насколько это возможно. Во-вторых, это гарантирует, что никакой сторонний код не обойдет уровень абстракции и не получит прямой доступ к данным. Понимаю, что некоторые не согласны с этой точкой зрения, и в этом нет ничего плохого — главное, чтобы уровень абстракции существовал.

В целом уровню абстракции данных нужен отдельный интерфейс для каждой из следующих задач:

- вставка данных;
- обновление данных;
- удаление данных;
- отбор отдельной строки и перечисление всех пользовательских объектов.


Проверено

При проектировании уровня абстракции данных избегайте матричного подхода функций CRUD (аббревиатура от операций создания, извлечения, обновления и удаления объектов) для каждой таблицы. В этом случае уровень абстракции данных будет жестко привязан к схеме. Планируйте этот уровень как набор логических соглашений, взаимодействующих с целевыми сущностями и задачами, даже если такое соглашение включает множество таблиц. Например, можно создать единый интерфейс, взаимодействующий с таблицами складских запасов, заказов и отгрузки.

В этой главе мы рассмотрим ряд хранимых процедур из уровня абстракции данных учебной базы OBXKites.

Хранимая процедура `AddNew`

Хранимая процедура `AddNew` отвечает за вставку в базу данных новых строк. Главной задачей этой процедуры является проверка допустимости данных, преобразование кодов во

внешние ключи и выполнение инструкции INSERT. Процедура AddNew также способна решать вопросы ожидания блокировки. Ниже приведен полный код хранимой процедуры AddNew из базы данных OBXKites.

```
CREATE PROCEDURE pProduct_AddNew(
  @ProductCategoryName NVARCHAR(50),
  @Code CHAR(10),
  @Name NVARCHAR(50),
  @ProductDescription NVARCHAR(100) = NULL
  )
AS
SET NOCOUNT ON
DECLARE
  @ProductCategoryID UNIQUEIDENTIFIER
SELECT @ProductCategoryID = ProductCategoryID
  FROM dbo.ProductCategory
  WHERE ProductCategoryName = @ProductCategoryName
IF @@Error <> 0 RETURN -100
IF @ProductCategoryID IS NULL
  BEGIN
    RAISERROR ('Категория товаров: ''%s'' не найдена',
               15,1,@ProductCategoryName)
    RETURN -100
  END
BEGIN TRY
  INSERT dbo.Product (ProductCategoryID, Code, ProductName,
          ProductDescription)
      VALUES (@ProductCategoryID, @Code, @Name,
          @ProductDescription )
END TRY
BEGIN CATCH
  RAISERROR ('Невозможно вставить новый товар', 15,1)
  RETURN -100
END CATCH
```

Для гестирования процедуры попытаемся вставить в таблицу Product товар с кодом 999:

```
EXEC pProduct_AddNew
  @ProductCategoryName = 'OBX',
  @Code = '999',
  @Name = 'Test Kit',
  @ProductDescription
      = 'official kite testing kit for contests.'
```

Убедимся, что вставка выполнена с помощью следующего запроса на отбор товара с кодом 999:

```
SELECT ProductName, ProductCategoryName
  FROM dbo.Product
    JOIN ProductCategory
      ON Product.ProductCategoryID
         = ProductCategory. ProductCategoryID
  WHERE Code = '999'
```

Получим ожидаемый результат:

```
Name              ProductCategoryName
----------------- --------------------
Test Kit          OBX
```

Хранимая процедура Fetch

Хранимая процедура Fetch извлекает данные. Если усложнить ее, то можно принимать множество параметров, возвращая, в зависимости от их состава, одну строку, их отфильтрованный набор или все множество строк. При этом совершенно не нужно писать для каждого конкретного набора данных отдельную процедуру. В предложении WHERE используется значение по умолчанию, null, так что если какой-либо параметр не предоставлен, соответствующий критерий отключается.

Данная хранимая процедура также решает вопросы ожидания блокировки и взаимоблокировок, используя методы, которые мы рассмотрим в главе 51. Следующая хранимая процедура Fetch возвращает информацию о товаре из базы данных OBXKites:

```
CREATE PROCEDURE pProduct_Fetch(
  @ProductCode CHAR(15) = NULL,
  @ProductCategory CHAR(15) = NULL )
AS
SET NoCount ON
SELECT Code, ProductName, ProductDescription, ActiveDate,
       DiscontinueDate, ProductCategoryName, [RowVersion] --,
-- Product.Created, Product.Modified
  FROM dbo.Product
    JOIN dbo.ProductCategory
      ON Product.ProductCategoryID
         = ProductCategory.ProductCategoryID
  WHERE ( Product.Code = @ProductCode
          OR @ProductCode IS NULL )
          AND ( ProductCategory.ProductCategoryName
             = @ProductCategory
          OR @ProductCategory IS NULL )
IF @@Error <> 0 RETURN -100
RETURN
```

Вызовем эту процедуру безо всяких параметров и получим данные обо всех товарах:

```
EXEC pProduct_Fetch
```

Представленные ниже результаты выполнения этой процедуры усечены по столбцам и строкам.

```
Code  Name                            Modified
----- ------------------------------- -----------------------
1001  Basic Box Kite 21 inch          2002-02-18 09:48:31.700
1002  Dragon Flight                   2002-02-18 15:19:34.350
1003  Sky Dancer                      2002-02-18 09:48:31.700

...
```

Если подставить параметр @ProductCode, то хранимая процедура Fetch вернет информацию только об одном товаре (столбцы результата усечены):

```
EXEC pProduct_Fetch
  @ProductCode = '1005'
```

Вот ее результат:

```
Code  Name                            Modified
----- ------------------------------- -----------------------
1005  Eagle Wings                     2002-02-18 09:48:31.700
```

Установка второго параметра позволяет получить список всех товаров указанной категории (в результате усечены строки и столбцы):

```
EXEC pProduct_Fetch
  @ProductCategory = 'Book'
```

Результат таков:

```
Code  Name                              Modified
----- ------------------------------- -----------------------
1036  Adventures in the OuterBanks    2002-02-25 17:13:15.430
1037  Wright Brothers Kite Designs    2002-02-25 17:13:15.430
1038  The Lighthouses of the OBX      2002-02-25 17:13:15.430
1039  Outer Banks Map                 2002-02-25 17:13:15.430
1040  Kiters Guide to the Outer Banks 2002-02-25 17:13:15.430
```

Хранимая процедура Update

Хранимая процедура Update использует основной метод идентификации строки и принимает в качестве параметров код товара и новые данные. На основе этих данных она выполняет инструкцию DML UPDATE.



Операции обновления подвержены потерям данных, и эту тему мы затронем в главе 51. Данную проблему можно обойти, устанавливая штампы или выполняя минимальные обновления. Каждая из этих методик будет продемонстрирована в этом разделе с сопутствующим примером хранимой процедуры.

В первом примере хранимая процедура обходит проблему потерянных обновлений с помощью проверки столбца rowversion с типом timestamp. При каждом обновлении строки значение этого поля автоматически обновляется. Если значение rowversion отличается от извлеченного, значит, строка обновляется другой транзакцией. При этом условие rowversion в предложении WHERE не удовлетворяется, что предотвращает обновление.

Обновление с условием RowVersion

Эта версия процедуры Update обновляет все столбцы строки. Ей должны передаваться все параметры, даже если некоторый столбец не обновляется. Эта процедура предполагает, что столбец rowversion был отобран при изначальном извлечении данных.

Если значение столбца rowversion отличается от извлеченного в момент отбора данных, то обновление не выполняется. Эта процедура определяет, что используется глобальная переменная @@rowcount, и передает ошибку вызывающему объекту.

В качестве примера можно привести текст хранимой процедуры pProduct_Update_RowVersion из базы данных OBXKites:

```
CREATE PROCEDURE pProduct_Update_RowVersion (
  @Code CHAR(15),
  @RowVersion Rowversion,
  @Name VARCHAR(50),
  @ProductDescription VARCHAR(50),
  @ActiveDate DateTime,
  @DiscontinueDate DateTime )
AS
SET NoCount ON
UPDATE dbo.Product
  SET
   ProductName = @Name,
   ProductDescription = @ProductDescription,
   ActiveDate = @ActiveDate,
```

```
   DiscontinueDate = @DiscontinueDate
  WHERE Code = @Code
        AND [RowVersion] = @RowVersion
IF @@ROWCOUNT = 0
BEGIN
  IF EXISTS ( SELECT * FROM Product WHERE Code = @Code)
    BEGIN
       RAISERROR ('Товар не был обновлен ввиду того,
          что другая транзакция обновила строку
          с момента последнего чтения.', 16,1)
       RETURN -100
    END
  ELSE
    BEGIN
      RAISERROR ('Товар не был обновлен,
              так как строка была удалена', 16,1)
      RETURN -100
    END
END
RETURN
```

Для тестирования процедуры обновления извлечем штамп для товара с кодом 1001 с помощью процедуры pProduct_Fetch:

```
EXEC pProduct_Fetch 1001
```

Получим следующий результат (столбцы усечены):

```
Code     Name                     RowVersion
-------- ------------------------ ------------------
1001     Basic Box Kite 21 inch   0x0000000000000077
```

Теперь вызовем процедуру pProduct_Update_Rowversion с полученным значением rowversion и попытаемся выполнить обновление:

```
EXEC pProduct_Update_Rowversion
  1001,
  0x0000000000000077,
  'updatetest',
  'new description',
  '1/1/2002',
  NULL
```

Процедура обновит все столбцы строки, при этом столбцу rowversion будет присвоено новое значение.

Минимальное обновление

Вторая версия хранимой процедуры Update демонстрирует метод минимальных обновлений. Обновляя только те столбцы, которые требуют изменения значений, мы существенно сокращаем вероятность замены обновлений, выполняемых другим пользователем. Этот вариант процедуры можно сравнить с хирургической операцией, когда врач вторгается только в нужную ему область, не нанося побочных повреждений.

Данная хранимая процедура не использует динамический SQL для создания инструкции UPDATE, хотя и это было бы сделать нетрудно. Следует отметить, что динамический SQL выполняется с профилем безопасности пользователя, а не хранимой процедуры, что приводит к необходимости перекомпиляции процедуры и соответствующему снижению производительности. Именно этот аспект делает нежелательным использование динамического SQL в хранимых процедурах производственных приложений.

Процедура минимального обновления выполняет инструкцию UPDATE отдельно для каждого передаваемого ей параметра:

```
CREATE PROCEDURE pProduct_Update_Minimal (
  @Code CHAR(15),
  @Name VARCHAR(50) = NULL,
  @ProductDescription VARCHAR(50) = NULL,
  @ActiveDate DateTime = NULL,
  @DiscontinueDate DateTime = NULL )
AS
SET NoCount ON
IF EXISTS (SELECT * FROM dbo.Product WHERE Code = @Code)
BEGIN
  BEGIN TRANSACTION
   IF @Name IS NOT NULL
    BEGIN
      UPDATE dbo.Product
        SET
          ProductName = @Name
        WHERE Code = @Code
      IF @@Error <> 0
        BEGIN
          ROLLBACK
          RETURN -100
        END
    END

   IF @ProductDescription IS NOT NULL
    BEGIN
      UPDATE dbo.Product
        SET
          ProductDescription = @ProductDescription
        WHERE Code = @Code
      IF @@Error <> 0
        BEGIN
          ROLLBACK
          RETURN -100
        END
    END

  IF @ActiveDate IS NOT NULL
   BEGIN
     UPDATE dbo.Product
       SET
         ActiveDate = @ActiveDate
       WHERE Code = @Code
     IF @@Error <> 0
       BEGIN
         ROLLBACK
         RETURN -100
       END
   END

  IF @DiscontinueDate IS NOT NULL
   BEGIN
     UPDATE dbo.Product
       SET
         DiscontinueDate = @DiscontinueDate
```

```
      WHERE Code = @Code
    IF @@Error <> 0
      BEGIN
        ROLLBACK
        RETURN -100
      END
  END
 COMMIT TRANSACTION
END
ELSE
 BEGIN
  RAISERROR
     ('Товар не был обновлен,
      так как строка была удалена', 16,1)
  RETURN -100
 END
RETURN
```

При вызове хранимой процедуры минимального обновления требуется передавать только те столбцы, которые требуют изменения значения. В первую очередь процедура определяет, существует ли такая строка, после чего переданные параметры обновляют столбцы таблицы. В следующем примере мы попробуем обновить описание товара с кодом `1001`:

```
EXEC pProduct_Update_Minimal
  @Code = '1001',
  @ProductDescription = 'a minimal update'
```

Протестируем результаты с помощью хранимой процедуры `pProduct_Fetch`:

```
EXEC pProduct_Fetch 1001
```

Будет получен следующий результат (столбцы усечены):

```
Code      Name              ProductDescription
--------- ----------------- ------------------
1001      updatetest        a minimal update
```

Хранимая процедура `Delete`

Хранимая процедура удаления выполняет инструкцию DML `UPDATE`. Она может оказаться наиболее сложной из всех, в зависимости от уровня архивирования данных и использования логических удалений. В примере, взятом из учетной базы данных OBXKites, переменная `@ProductCode` преобразуется в `@ProductID`, затем проверяется наличие данного товара, после чего он удаляется из таблицы.

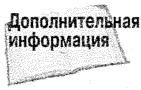
```
CREATE PROCEDURE pProduct_Delete(
  @ProductCode INT
  )
AS
SET NOCOUNT ON
DECLARE @ProductID UniqueIdentifier
SELECT @ProductID = ProductID
  FROM Product
  WHERE Code = @ProductCode
If @@RowCount = 0
  BEGIN
    RAISERROR
      ('Невозможно удалить товар с кодом %i
        - он не существует.', 16,1, @ProductCode)
```

```
    RETURN
  END
ELSE
  DELETE dbo.Product
    WHERE ProductID = @ProductID
RETURN
```

Для тестирования хранимой процедуры `pProduct_Delete` мы попытаемся удалить несуществующий товар с кодом `99` и в результате получим сообщение об ошибке:

```
EXEC pProduct_Delete 99

Невозможно удалить товар с кодом 99 - он не существует.
```



Несмотря на то что приведенная в качестве примера процедура удаления исключительно проста, на практике удаление данных является одной из самых сложных задач уровня абстракции данных. Все зависит от того, что нужно делать со старыми данными. Стратегии логического удаления и архивирования данных мы обсуждали в главе 24.

Резюме

Уровень абстракции данных является ключевым компонентом плана архитектуры базы данных. Он играет главную роль в возможности дальнейшего расширения и затратах на сопровождение базы данных. Даже если на первый взгляд кажется, что затраты на создание уровня абстракции данных и преобразование программного кода существующих приложений довольно велики (если существующая база пока использует прямое обращение к таблицам), опытные программисты и менеджеры поймут, что все вложенные средства и труд с лихвой окупятся в процессе дальнейшей эксплуатации базы данных.

В следующей главе мы продолжим разговор о разработке базы данных, обратив свое внимание на портативный мир SQL Server Mobile.

Программирование для SQL Server Everywhere

ГЛАВА

26

В этой главе...

Что нового в SQL Everywhere 2005

Создание базы данных SQL Everywhere

Обновление платформы SQL CE

Синхронизация мобильных данных

Упаковка и развертывание

Вопросы безопасности

Настройка, обслуживание и администрирование

Разработка приложений для платформы Windows Mobile достигла невероятных масштабов с момента введения в 2003 году среды .NET Compact Framework. Впервые программисты с базовым уровнем подготовки получили возможность использовать знакомую среду разработки Visual Studio для создания, тестирования и развертывания надежных мобильных приложений. На практике большинство промышленных мобильных приложений требуют надежного и защищенного хранилища данных на мобильном устройстве, а также способа синхронизации мобильных данных с сервером предприятия.

Рассмотрим типичный склад, где рабочие выполняют операции получения, хранения, отпуска и доставки материалов или товаров. Заведующий складом в начале каждого дня садится за терминал сервера базы данных и распечатывает расписание получения товаров, сводный отчет об их отпуске и остатках на складе. Успех работы склада зависит от того, своевременно ли заносят рабочие данные о своих операциях в течение дня, т.е. соответствует ли печатный отчет реальному положению дел. Естественно, этот процесс открыт для ошибок, сбоев в работе и даже мошенничества. Для автоматизации процесса занесения информации и предоставления возможности рабочим участвовать в процессе ее накопления склад планирует развернуть приложение Windows Mobile. Рабочие склада всегда находятся в движении, принимая и отпуская товары, перемещая их по складу, занося накладные, принесенные экспедиторами, и т.п. Чтобы обеспечить синхронизацию работы сотрудников склада друг с другом и с центральным сервером, мобильному приложению нужно часто подключаться к устройствам рабочих, чтобы отслеживать данные, изменяемые каждым из них. Естественно, заманчиво создать приложение в среде .NET Compact Framework, которое имеет непосредственный доступ к серверам на устройствах рабочих склада. Однако на практике выясняется, что вы-

сокие шлакоблочные стены складских помещений приводят к перебоям в работе мобильной сети, что препятствует взаимодействию с базой данных, и поэтому рабочим приходится часто перемещаться от одной базовой станции к другой. Решение заключается в создании локального кэша данных на мобильных устройствах, что позволит рабочим продолжать свою работу даже при отсутствии сетевого подключения и выполнять репликацию, когда это подключение станет доступным. SQL Server Mobile Edition предлагает эту возможность рабочим склада, продавцам многоэтажного супермаркета, врачам реанимации, а также миллиардам других мобильных сценариев.

Вооружившись Visual Studio 2005 и SQL Server 2005, компания Microsoft представила редакцию SQL Server 2005 Mobile Edition (также называемую SQL Everywhere). Это не просто модернизация предыдущей версии — SQL CE, — а полностью переделанная мобильная база данных. Она содержит перевооруженный процессор запросов, новое ядро хранилища данных, поддержку множества подключений к одной базе данных и способность обеспечения работы с SQL Everywhere, подобно тому, как работают с источниками данных Visual Studio 2005 и SQL Server 2005 Management Studio. В этой главе представлен обзор возможностей SQL Everywhere и рассмотрены типичные задачи, связанные с мобильными решениями масштаба предприятия.



Когда редактировалась настоящая книга, компания Microsoft анонсировала планируемые улучшения в SQL Server 2005 Mobile Edition и переименовала продукт в SQL Server 2005 Everywhere Edition. В дальнейшем эти названия можно считать взаимозаменяемыми.

Обзор SQL Server 2005 Everywhere Edition

Несмотря на то что SQL Everywhere представляет собой полностью переработанный продукт, обладающий бо́льшим количеством справочного материала и документации, чем его предшественники, краткий экскурс в историю развития этого продукта будет не лишним.

История

С пакетом Visual Studio 2003, средой .NET Compact Framework 1.0 и СУБД SQL Server 2000 компания Microsoft представила бесплатно распространяемую реляционную базу данных для карманных компьютеров и системы Windows CE, которая называется SQL Server 2000 Windows CE Edition (или SQL CE). Занимая всего порядка 1,5 Мбайт на устройстве, SQL CE предоставляла на удивление богатую реализацию основных функций SQL Server 2000. В эту СУБД была включена поддержка подмножества индексов и объединений ANSI SQL-92, оптимизированный процессор запросов и интерфейсы программирования ADOCE, OLEDB и ADO.NET. В SQL CE также были представлены две технологии синхронизации мобильных данных с корпоративными серверами: *удаленного доступа к данным (RDA)* и *репликации слияния*.

Хотя SQL CE представляла собой пригодную реляционную базу данных для решений Windows Mobile, она имела ряд ограничений. Наиболее существенным из них было то, что данная база могла существовать исключительно в контексте мобильных устройств. Приходилось создавать, настраивать и обслуживать базу данных SQL CE либо с помощью программного кода, либо с помощью миниатюрной версии анализатора запросов (Query Analyzer), запускаемой на устройстве. SQL CE также ограничивала доступ к базе данных всего одним подключением, не исключала возможности искажения данных, что приводило к частым операциям восстановления и обслуживания мобильной базы данных. Например, работая с SQL CE, мобильным пользователям приходилось делать перезагрузку во время операций записи, после

чего база данных становилась непригодной к работе и требовала восстановления. Очевидно, что это была совершенно не идеальная ситуация для администраторов баз данных, поддерживающих сотни мобильных устройств, разбросанных по широкому географическому региону.

Концепции

Для понимания того, где может найти место SQL Everywhere в архитектуре предприятия, в противоположность более знакомым СУБД SQL Server 2000 или 2005, важно осознать некоторые фундаментальные концепции. SQL Everywhere является компактной полноценной системой управления базами данных (СУБД), предназначенной для работы на устройствах с установленной системой Microsoft Windows Mobile. СУБД SQL Everywhere совместима с такими версиями, как Pocket PC 2003, Windows Mobile 5 и Windows Mobile 5 Smart Phone Edition. Особо отмечу, что в настоящее время платформа Windows CE.NET 4.2 не поддерживается в SQL CE. однако такую поддержку обещают с выходом в свет .NET Compact Framework 2.0 Service Pack 1.

Интересно заметить, что SQL Everywhere также лицензирована для системы Windows XP Tablet PC Edition, но ни для какой другой платформы x86, если, конечно, на том же компьютере не установлена лицензионная копия Visual Studio 2005 или SQL Server 2005. Компания Microsoft рассматривает Tablet PC как идеальную платформу для клиентских приложений, а SQL Everywhere — как подходящую и легко устанавливаемую СУБД для поддержки этих приложений.



Решение компании Microsoft лицензировать SQL Everywhere только для единственной x86-платформы Tablet PC может выглядеть несколько странным. В конце концов, разве редакция SQL Server 2005 Express не является облегченной версией СУБД, предназначенной для клиентских приложений? Сообщество MVP пыталось оказать давление на компанию Microsoft, чтобы та расширила лицензирование SQL Everywhere на другие платформы x86, чтобы избежать необходимости сложного конфигурирования репликации между портативными и стационарными устройствами, однако эти просьбы не встретили поддержки. К этому следует добавить сложность репликации баз данных SQL Express, развернутых "с помощью одного щелчка", вместе с полезными клиентскими приложениями, которая требует глубокого знания программирования RMO, что делает SQL Everywhere достойным решением в этом сценарии.

Единственным неприятным моментом является то, как именно компания Microsoft лицензирует SQL Everywhere. Эта СУБД может свободно распространяться на устройства, на которых установлены только что упомянутые операционные системы. В то же время необходимо приобрести лицензию SQL Server для использования доступа к удаленным данным (RDA), репликации слияния, а также для других подключений к SQL Server из SQL Everywhere. С особенностями лицензирования SQL Everywhere вы можете ознакомиться на сайте `www.microsoft.com/sql/editions/sqlmobile/howtobuy.mspx`.

Основной задачей SQL Everywhere является поддержка базы данных, которая является либо основным хранилищем для мобильных приложений, либо временным кэшем в сценариях, где данные извлекаются из сервера, создаются и модифицируются на устройстве, а затем снова возвращаются на сервер. SQL Everywhere можно интегрировать в проекты .NET Compact Framework Smart-Device с помощью Visual Studio 2005. Она предлагает поддержку OLEDBCE для приложений, написанных на языке Visual C++ for Devices в среде разработки Visual Studio 2005. Основные архитектурные компоненты SQL Everywhere показаны на рис. 26.1.

Как показано на рис. 26.1, двумя основополагающими компонентами SQL Everywhere являются *ядро хранилища данных* и *процессор запросов*. К ним можно получить доступ как из стандартного, так и из управляемого программного кода. Проще говоря, ядро хранилища данных SQL Everywhere является надежным долгосрочным хранилищем, а процессор запросов обеспечивает реляционный доступ к этому хранилищу. Ядро хранилища SQL Everywhere

было полностью переписано со времен SQL CE и сейчас является тем же ядром, которое использует Windows CE.NET Object Store (хотя к последнему невозможно непосредственно обратиться с помощью ADO.NET).

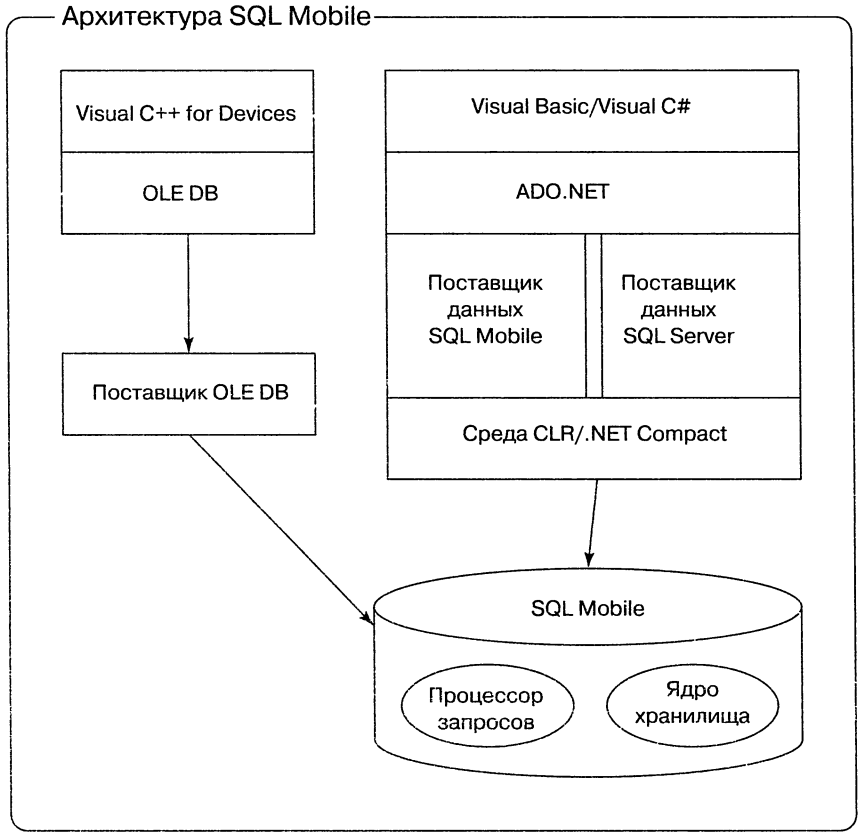


Рис. 26.1. Архитектура SQL Server 2005 Everywhere

Процессор запросов представляет собой основанный на стоимости эвристический исполнительный механизм, который отличается эффективностью, принимая во внимание малое место, занимаемое им на устройстве. Занимая всего лишь около 1.5 Мбайт на мобильном устройстве, SQL Everywhere реализует большую часть функций СУБД SQL Server 2005, которая, в свою очередь, потребляет минимум 38 Мбайт.



Термин *основанный на стоимости* означает, что процессор запросов рассматривает множество планов выполнения и выбирает из них самый эффективный. Термин *эвристический* означает, что процессор может заменить запрос его аналогом, который получает те же результаты, но рассматривается как более эффективный (например, выражение `IN` в запросе может быть переписано с использованием синтаксиса `JOIN`).

Возможности SQL Everywhere во многом аналогичны SQL Server 2005.

- Создание, модификация и удаление таблиц, индексов и ограничений.
- Выполнение инструкций языков определения данных (DDL) и манипуляций данными (DML) с подмножеством ANSI-92 SQL.

- Создание и гарантирование ACID-транзакций (т.е. обладающих свойствами атомарности, согласованности, изолированности и живучести).
- Выполнение запросов к представлениям схемы и метаданным.
- Влияние на выполнение запросов с помощью параметров и поддержание порядка объединений.

Как только вы погрузитесь в программирование для SQL Everywhere, то сразу заметите сходство с SQL Server 2000 и 2005. В то же время существует и ряд различий, о которых нужно знать, начиная работать с SQL Everywhere.

- База данных SQL Everywhere не различает регистры символов, и настроить ее по-другому невозможно.
- Поддерживаются только типы данных Unicode (т.е. тип `nvarchar` поддерживается, а `varchar` — нет).
- Во время доступа к удаленным данным и в процессе репликаций слияния могут произойти автоматические преобразования типов.
- Некоторые типы данных SQL Server не поддерживаются в SQL Everywhere (например, `smallmoney`).
- Некоторые типы данных нельзя включать в функции так, как это можно сделать в SQL Server. Например, столбцы `ntext` нельзя включать в строчные функции SQL Everywhere.
- Не вся спецификация ANSI-92 SQL, реализованная в языке T-SQL для SQL Server 2005, поддерживается в SQL Everywhere. Например, ограничены возможности подзапросов, многие встроенные функции T-SQL отсутствуют, а скобки не поддерживаются в качестве разделителей.
- Даже если в таблице определено множество индексов, при составлении плана выполнения процессор запросов рассмотрит их все, но выберет только один.
- SQL Everywhere не обрабатывает пакеты инструкций T-SQL — только по одной инструкции за раз.
- Отсутствует поддержка репликаций транзакций и мгновенных снимков.
- Web-репликации не требуют защищенной передачи посредством протокола SSL, так же как в SQL Server 2005 или SQL Express.
- Объем базы данных SQL Everywhere ограничен пределом 4 Гбайт, при этом автоматически устанавливается в 128 Мбайт.
- Ограничен уровень изменений, которые можно вносить в существующую базу данных посредством инструкций `ALTER TABLE`. Можно использовать только базовые операции, такие как добавление или удаление столбца или индекса.

Некоторые функции SQL Server 2005 вообще не включены в SQL Everywhere из соображений экономии памяти, в том числе:

- триггеры;
- представления;
- хранимые процедуры;
- пулы подключений;
- распределенные транзакции;
- шифрованные строки подключения.

Все эти отличия приспосабливают SQL Everywhere к особенностям мобильных решений. Перед тем как вы начнете возмущаться каким-либо из ограничений в приведенном выше списке ("Что?! Нет хранимых процедур?"), хочу заметить, что я разработал около двух десятков мобильных решений с использованием SQL CE и SQL Everywhere, а потому с уверенностью могу сказать, что никакие затраты на корректировку технического задания, которые пришлось выполнить, не могли сравниться с достоинствами этого на удивление маленького, но мощного ядра базы данных.

По мере ознакомления с возможностями SQL Everywhere вы начнете понимать, как увеличить мощь обычных или управляемых мобильных приложений за счет небольшой мобильной базы данных. Позже вы поймете, что самым выдающимся элементом SQL Everywhere являются технологии, заложенные в этом продукте, которые позволяют массе распределенных мобильных устройств поддерживать синхронизацию с серверами баз данных предприятия.

Забегая вперед, отмечу, что в SQL Everywhere существуют два метода синхронизации данных с SQL Server 2000/2005: доступ к удаленным данным (RDA) и репликация слияния. На рис. 26.2 показаны главные компоненты архитектуры синхронизации данных SQL Everywhere. Все эти компоненты мы детально рассмотрим в настоящей главе.

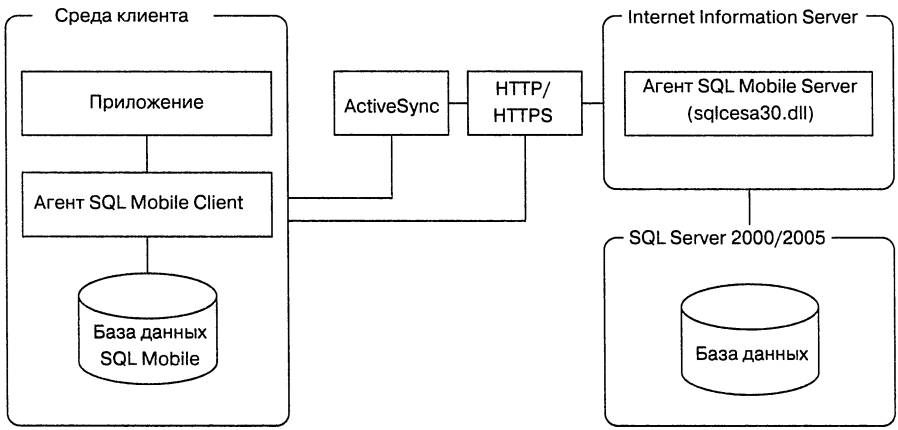


Рис. 26.2. Архитектура репликации SQL Everywhere

Следует заметить, что оба сценария синхронизации (RDA и репликация слияния) в качестве шлюза между SQL Everywhere и SQL Server используют сервер Internet Information Server (IIS), в котором запущен агент SQL Server Everywhere Server Agent (ISAPI DLL). Существуют ситуации, когда достойной альтернативой данного подхода к синхронизации данных является использование Web-служб; в то же время компания Microsoft считает, что RDA и репликация слияния способны покрыть львиную долю наиболее распространенных сценариев.

Что нового в SQL Server 2005 Everywhere

SQL Everywhere представляет собой нечто большее, нежели обновление SQL CE, — во многих отношениях это совершенно новый продукт. Он предлагает впечатляющие новые функции и обладает новыми характеристиками производительности. Все это результат переработки компанией Microsoft ядра хранилища данных и процессора запросов. Ниже приведен краткий список наиболее значительных новых функций, которые SQL Everywhere вносит в решения Windows Mobile.

- Возможность работы множества пользователей и с множеством подключением (SQL CE была однопользовательской базой данных с одним подключением).
- Переработанное ядро хранилища данных с повышенной производительностью и надежностью.
- Автоматическое повторное использование пустых страниц (СУБД SQL CE для этого требовала частых уплотнений для переназначения неиспользуемого пространства и пересчета статистики индексов).
- Улучшенная статистика процессора запросов и эвристический оптимизатор, основанный на стоимости.
- Новый процессор запросов различает оперативную память и долговременное хранилище мобильного устройства.
- Транзакции, поддерживающие принципы ACID, уменьшают риск потери или порчи данных в связи с переменным характером питания и сетевых подключений, специфичных для мобильной архитектуры.
- Блокировка на уровне строк и страниц, а также множество новых уровней изоляции.
- Интеграция с Visual Studio 2005.
- Интеграция с SQL Server Management Studio.
- Управляемый интерфейс программирования приложений делает возможным асинхронный контроль над репликацией слияния. При этом реализованы оперативная отчетность, сохранение и повторное использование настроек репликации, а также возможность отмены активных репликаций.
- Отслеживание репликаций слияния на уровне столбцов в значительной мере снижает объем данных, передаваемых между SQL Everywhere и SQL Server.
- Поддержка множества подписок слияния в одной базе данных SQL Everywhere.
- Поддержка именованных параметров в провайдере .NET Compact Framework 2.0 ADO.NET.
- Возможность экспорта плана выполнения из Query Analyzer 3.0 мобильного устройства в SQL Server 2005 Management Studio.
- Высокопроизводительный объект `SqlCeResultSet` теперь является частью поставщика данных .NET Compact Framework 2.0 ADO.NET. Он предлагает обновляемый прокручиваемый курсор, позволяющий работать с адаптером `SqlCeDataAdapter` для загрузки таблиц и наборов данных.
- Подключение и управление базами данных SQL Everywhere осуществляется на мобильном устройстве посредством ActiveSync.
- Встроенное уплотнение базы данных (SQL CE требовала указания баз данных источника и назначения и последующего переименования второй из них в исходную базу).

Как видите, компания Microsoft приложила много усилий, чтобы повысить эффективность SQL Everywhere как хранилища реляционных баз данных и механизма синхронизации. С точки зрения программиста, тесная интеграция с Visual Studio 2005 и SQL Server 2005 Management Studio существенно упростила работу по сравнению с программированием в SQL CE, которое необходимо было осуществлять непосредственно на мобильном устройстве. С точки зрения производительности и надежности СУБД SQL Everywhere сделала существенный рывок вперед. Теперь вы можете больше времени посвятить решению производственных задач с помощью мобильных решений, при этом потратив меньше времени на настройку и администрирование мобильных баз данных.

Основы SQL Everywhere

Если вы уже готовы опробовать SQL Everywhere, то в этом разделе вы узнаете о предварительных задачах, связанных с созданием первого приложения Windows Mobile, работающего с SQL Everywhere. Начнем с некоторых замечаний относительно получения и установки SQL Everywhere. Затем вы узнаете, как создать базу данных SQL Everywhere, используя разнообразные методики, и обновить существующую базу SQL CE 2.0 до SQL Everywhere. После этого мы рассмотрим в функции интеграции SQL Everywhere как с Visual Studio 2005, так и с SQL Server 2005 Management Studio.

Установка SQL Everywhere

СУБД SQL Everywhere включен в инсталляции всех редакций Visual Studio 2005, имеющих поддержку мобильных устройств (Standard, Professional и Team System), в качестве дополнительного компонента. Даже если вы уже установили одну из этих версий, посетите сайт `www.microsoft.com/sql/editions/sqlmobile/downloads.mspx` и проверьте наличие обновлений к Microsoft SQL Server 2005 Everywhere Edition Device SDK. На этом сайте для загрузки доступен и ряд других полезных ресурсов, в том числе SQL Everywhere Books Online, учебное приложение IBuySpy, а также другие компоненты сервера и репликации, необходимые для подготовки синхронизации данных между SQL Everywhere и SQL Server 2000 или 2005. Мне даже не хочется подчеркивать, насколько важно загрузить и ознакомиться с SQL Everywhere Books Online перед тем, как перейти к рассмотрению SQL Everywhere.

После того как вы последуете инструкциям Microsoft по загрузке и установке Device SDK, проверьте состав компонентов и местонахождение их на вашем компьютере.

Средства разработки

Если вы создаете приложение для смарт-устройств в Visual Studio 2005, используя среду .NET Compact Framework, добавьте в свой проект ссылку на пространство имен `System.Data.SqlServerCe`. Это пространство имен содержит все, что нужно для взаимодействия приложения с SQL Everywhere, — от простых запросов SQL до инициализации репликации слияния и доступа к удаленным данным. При установке Microsoft Server 2005 Everywhere Edition Device SDK соответствующая сборка `System.Data.SqlServerCe` помещается с расширением `.dll` в структуру каталога программы Visual Studio 2005 (*<устройство>*`:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0`).

Если для разработки мобильного приложения вы используете стандартный программный код Visual C++, то после установки Microsoft Server 2005 Everywhere Edition Device SDK вы найдете соответствующие файлы заголовков (`ssceoledb30.h`, `ca_merge30.h` и `ssceerr30.h`) в той же папке, которая была указана ранее.

Файлы инсталляции с расширением `.msi` в этой папке предназначены для конфигурирования SQL Everywhere Server Tools с целью синхронизации с SQL Server 2000 или 2005. Нет никакой необходимости устанавливать эти утилиты, пока вы не будете готовы к использованию репликации слияния или доступа к удаленным данным.

Средства времени выполнения

Чтобы приложение Windows Mobile могло связываться с SQL Everywhere во время выполнения на устройстве или его эмуляторе, на этом устройстве (как минимум) должно быть установлено ядро базы данных SQL Everywhere. Это ядро упаковано в дистрибутивный архивный файл CAB. В зависимости от архитектуры процессора устройства и платформы

Windows Mobile требуются различные дистрибутивы CAB. Ниже будет предложено руководство по выбору корректного дистрибутива, однако перед этим вам следует знать, что наряду с дистрибутивом ядра следует развернуть еще два файла CAB: агента клиента и утилит разработки SQL Everywhere. Агент клиента приступает к работе, когда происходит репликация слияния или удаленный доступ к базе данных. Утилиты разработки содержат анализатор запросов Query Analyzer 3.0, а также дополнительную информацию об отладке. В общем случае их не обязательно устанавливать вместе с рабочей версией мобильного приложения.

Файлы CAB можно скопировать на устройство при его подключении к компьютеру разработки средствами ActiveSync. При открытии эти файлы автоматически устанавливаются на устройстве (достаточно пером нажать на имя файла). Чтобы использовать на устройстве SQL Everywhere, требуется как минимум установить ядро базы данных. Следует отметить, что SQL Everywhere устанавливается автоматически при развертываниии проекта Visual Studio 2005 Smart Device, который содержит ссылку на пространство имен `System.Data.SqlServerCe`.



Если вы точно не знаете, какой из файлов CAB SQL Everywhere требуется вашему мобильному приложению, посмотрите на окно вывода Visual Studio 2005 во время развертывания приложения на устройстве. Visual Studio 2005 перечислит имена файлов CAB, которые сочтет подходящими для вашего проекта.



Новой функцией Visual Studio 2005 является то, что во время развертывания проекта на устройстве вы можете управлять тем, будет ли включена база данных SQL Everywhere в его содержимое всегда, только при обновлении или никогда.

Выбор корректных файлов CAB SQL Everywhere

В зависимости от типа операционной системы Windows Mobile, установленной на мобильном устройстве, и архитектуры его процессора вы можете найти все необходимые файлы CAB по одному из двух путей:

- `<`*`устройство`*`:>\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0\vce400\armv4`
- `<`*`устройство`*`:>\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0\vce500`

Общая схема имен и организация файлов CAB зависит в первую очередь от операционной системы и уж затем от архитектуры процессора. Таким образом, если из перечисленных выше путей вы выбираете `vce500`, то можете далее перейти в один из подкаталогов, соответствующих типу процессора (`armv4i`, `mips`, `sh4` и т.д.), и там найти соответствующие файлы для систем Windows Mobile 5.0, Windows CE.NET 5.0, Smartphone 5.0 и т.д. Новичкам в мобильном программировании будет несколько сложно в этом разобраться, поэтому разрешите дать несколько советов относительно выбора правильных файлов CAB для вашего устройства.

- Все устройства Pocket PC 2003, 2003 Second Edition и Phone Edition базируются на ядре Windows CE 4 и архитектуре процессора ARMV4. (Из этого правила есть отдельные исключения, но SQL Everywhere не поддерживает такие конфигурации.)
- Все устройства Windows Mobile 5.0, Windows CE.NET 5.0 и Windows Mobile 5.0 Smartphone Edition базируются на ядре Windows CE 5.
- Если вы не знаете точно тип операционной системы своего устройства или архитектуру процессора, выберите на устройстве пункт меню Start⇨Settings⇨System⇨About.
- Помните, что вы всегда можете развернуть на своем устройстве или эмуляторе проект из Visual Studio 2005 и сразу увидеть в окне Output состав необходимых файлов CAB.



В мобильном мире единицей развертывания на устройстве является файл CAB. Что бы вы ни устанавливали — Compact Framework, SQL Everywhere или проект Smart Device Setup, — вам следует скопировать на устройство соответствующий файл CAB, и он сам там установится без вашего участия.

Рассмотрим пример определения состава корректных файлов CAB. Предположим, что у вас есть устройство HP iPaq 4355 с установленной системой Pocket PC 2003. Эта система базируется на ядре Windows CE 4. Если вы выберете пункт меню Start⇨Settings⇨System⇨About, то увидите, что на этом устройстве установлен процессор Intel(r) PXA 255. Это ARMV4-совместимый процессор (этот факт вы можете проверить на сайте компании Intel). Таким образом, нужные файлы CAB вы можете найти в папке `<устройство:>\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0\vce400\armv4`. Приведенные ниже файлы из этой папки вам нужно будет скопировать на данное устройство.

- `Sqlce30.ppc.wce4.armv4.cab`. Ядро базы данных SQL Everywhere (его установка обязательна).
- `Sqlce30.dev.ENU.ppc.wce4.armv4.cab`. Средства разработки SQL Everywhere (установка не обязательна).
- `Sqlce30.repl.ppc.wce4.armv4.cab`. Агент клиента SQL Everywhere (установка не обязательна).

Также важно знать роль каждой из библиотек этих дистрибутивных файлов, которые будут установлены на мобильном устройстве. Если исключение `SqlCeException` инициируется в программе, в его сообщении или стеке, вы можете найти имя одной из этих библиотек. Вот краткий список динамических библиотек, которые могут быть установлены на вашем мобильном устройстве.

- `SQLCESE30.DLL` — ядро хранилища данных.
- `SQLCEQP30.DLL` — процессор запросов.
- `SQLCECA30.DLL` — агент клиента.
- `SQLCESA30.DLL` — агент сервера.
- `SQLCERP30.DLL` — поставщик данных репликации.
- `SQLCEME30.DLL` — управляемые исключения.
- `SQLCEOLEDB30.DLL` — поставщик данных OLEDB.
- `SQLCEER30`*xx*`.DLL` — локализованная строка исключения, где *xx* — это `EN`, `TW`, `CN`, `DE`, `ES`, `FR`, `IT`, `JA`, `KO`.

В следующем разделе будет представлена важная утилита Query Analyzer 3.0, включенная в состав средств разработки SQL Everywhere. Следуя приведенным выше указаниям, установите на своем мобильном устройстве необходимые ядро базы данных и средства разработки. Помните, что если вы планируете создавать управляемый программный код, то вам необходимо установить также и .NET Compact Framework 2.0.

Если при копировании вручную или установке файлов CAB у вас возникли проблемы, создайте новый проект Smart Device в Visual Studio 2005. Для этого в меню Visual Studio 2005 выберите пункт File⇨New⇨Project⇨Visual Basic или Visual C#⇨Smart Device⇨Pocket PC 2003⇨Device Application и присвойте проекту имя. После этого добавьте в проект ссылку на пространство имен `System.Data.SqlServerCe`, создайте проект в режиме отладки (DEBUG) и разверните его на мобильном устройстве или эмуляторе. При создании проекта в режиме отладки на устройстве или эмуляторе будут развернуты также и средства разработки SQL Everywhere. Во время развертывания в окне Output вы увидите состав устанавливаемых файлов CAB.

Query Analyzer 3.0

Мобильная версия анализатора запросов (Query Analyzer 3.0) представляет собой утилиту, предназначенную для работы с базами данных SQL Everywhere на мобильном устройстве или его эмуляторе. В следующем примере продемонстрировано использование этой утилиты для запуска запроса к базе данных SQL Everywhere под названием IbuySpyStore.



Копию базы данных IbuySpyStore вы можете найти в дереве каталога программы Visual Studio по пути `<устройство:>\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0\Northwind.sdf`. Чтобы выполнить пример, скопируйте этот файл на свое мобильное устройство. После этого запустите Query Analyzer 3.0 из меню Start или перейдите к файлу с помощью утилиты File Explorer на устройстве и нажмите пером на названии файла. Следует отметить, что Query Analyzer 3.0 работает исключительно с базами данных SQL Everywhere, — утилита даже не подключится к базе данных SQL CE 2.0. Аналогично, старая версия Query Analyzer для баз данных SQL CE (она называлась Pocket Query Analyzer) не подключится и не сможет работать с базами данных SQL Everywhere. На одном устройстве могут быть одновременно установлены обе версии анализатора запросов, однако если нажать пером на файле `.sdf`, автоматически будет запущена та версия, которая была установлена последней. Разумеется, это может привести к невозможности работы с конкретной базой данных.

Начнем рассмотрение примера с проверки того, скопирована ли база данных `Northwind` на устройство (рис. 26.3); не имеет значения, в каком именно месте устройства находится этот файл.

Теперь нажмем пером на имени файла, чтобы запустить утилиту Query Analyzer 3.0, в которой можно начать работу с базой данных (рис. 26.4).

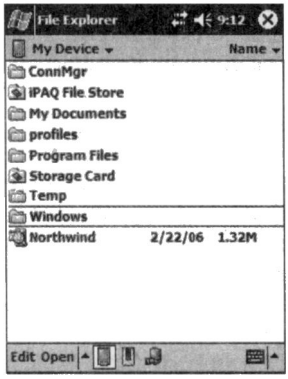


Рис. 26.3. База данных SQL Everywhere под названием Northwind на устройстве Pocket PC 2003

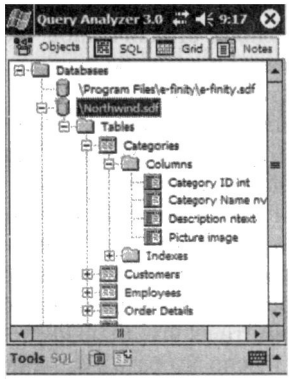


Рис. 26.4. Вкладка Objects утилиты Query Analyzer 3.0

Обратите внимание на то, что интерфейс пользователя утилиты Query Analyzer 3.0 содержит четыре вкладки. Вкладка Objects отображает все локальные базы данных SQL Everywhere мобильного устройства. В ней можно подключиться к любой из этих баз и просмотреть ее таблицы, столбцы, индексы и т.п. Вкладка SQL позволяет вводить и выполнять инструкции SQL (точнее, подмножество инструкций T-SQL, реализованное в SQL Everywhere). Во вкладке Grid отображаются результаты выполнения запросов, а во вкладке Notes — статистическая информация о выполнении запроса, такая как количество возвращенных строк, время выполнения и обнаруженные ошибки.

В нашем примере во вкладке SQL мы введем простую инструкцию SELECT, которая возвращает все строки таблицы Employee базы данных Northwind (рис. 26.5). После ввода запроса нажмите пером на зеленой стрелке, направленной вправо, которая расположена в нижней части окна утилиты, и запрос будет выполнен.

Если запрос выполнен успешно, то утилита автоматически переключается на вкладку Grid и отображает результаты (рис. 26.6). Если же запрос выполнить не удалось, то переключение происходит на вкладку Note, где можно просмотреть сообщения об ошибках.

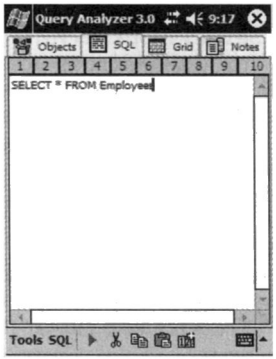


Рис. 26.5. Вкладка SQL утилиты Query Analyzer 3.0

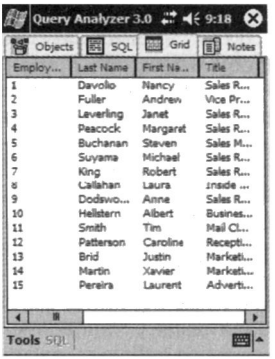


Рис. 26.6. Вкладка Grid утилиты Query Analyzer 3.0

Мы провели только краткий экскурс по утилите Query Analyzer 3.0, на самом деле в ней скрыто множество дополнительных полезных функций. Например, во вкладке SQL вы можете создать кнопки часто запускаемых запросов, чтобы не вводить их каждый раз заново. Вы можете выполнять ограниченное число манипуляций со схемой базы данных (например, добавлять индексы и внешние ключи), что бесценно на этапе проектирования и отладки приложения Windows Mobile.

Если приложение Windows Mobile поддерживает стандарты *Designed for Windows Mobile* компании Microsoft, то закрытие анализатора запросов щелчком на крестике в правом верхнем углу окна не останавливает выполнение приложения — оно просто переводится в фоновый режим. Если вы новичок в программировании, это может показаться вам неважным. В то же время это позволяет часто используемым программам оставаться загруженным в память, а значит, пользователи смогут запускать их быстрее, чем с диска. Следствием того, что Query Analyzer переводится в фоновый режим работы, является то, что база данных остается открытой. Чтобы выйти из утилиты Query Analyzer и закрыть все подключения к базам данных SQL Everywhere, нужно выбрать в ее меню, находящемся в нижней части окна, пункт Tools⇨Exit.

Создание базы данных SQL Everywhere

Пожалуй, первой трудностью, с которой приходится сталкиваться разработчику мобильных приложений, работающих с SQL Everywhere, является создание физической базы данных. Эту проблему можно решить множеством способов; вы вольны выбрать тот из них, который будет удобнее всего для вас и который будет соответствовать требованиям приложения. Базы данных SQL Everywhere могут быть созданы несколькими способами, среди которых следующие.

- С помощью обычного или управляемого программного кода.
- Визуально на устройстве с помощью утилиты Query Analyzer 3.0.
- Визуально на компьютере с использованием Visual Studio 2005.

- Визуально на компьютере с помощью сценариев и шаблонов в SQL Server Management Studio 2005.
- Автоматически с помощью репликации слияния.

В следующих разделах мы вкратце рассмотрим все эти технологии и приведем примеры.

Создание базы данных SQL Everywhere с помощью управляемого программного кода

Первым вариантом, который мы рассмотрим, является создание базы данных SQL Everywhere с помощью управляемого программного кода и использования Visual Studio 2005 и .NET Compact Framework 2.0. Этот вариант особенно полезен в ситуациях, когда выполняется создание и продажа мобильных приложений большой группе пользователей, сетевые подключения и частота синхронизации которых неконтролируемы. Например, несколько лет назад я создал приложение .NET Compact Framework, позволяющее врачам вводить названия всех медикаментов, которые должен принимать пациент, и просчитывать непосредственно на мобильных устройствах возможные их взаимодействия. Поскольку вычисление этих взаимодействий достаточно сложное, возникла необходимость использования на устройстве реляционной базы данных. Проблема состояла в том, что у меня не было никакого контроля над сетевыми подключениями мобильных устройств различных врачей к серверам в разных регионах и к Интернету для синхронизации. Данные, необходимые этому мобильному приложению на устройстве, использовались исключительно для чтения, а на серверах они обновлялись ежеквартально. Решение состояло в том, чтобы регулярно публиковать новую версию приложения, которое при первом запуске удаляло старую базу данных SQL CE и в процессе работы создавало новую. Схема этой базы и сами данные развертывались вместе с приложением как файлы CSV; после загрузки данных в базу SQL CE эти файла автоматически удалялись. Несмотря на то что при первом запуске новой версии программы медикам приходилось несколько минут ожидать, отсутствие необходимости подключения к сети и синхронизации данных с лихвой компенсировало этот недостаток.

Динамическое создание базы данных SQL Everywhere во время работы программы и даже наполнение таблиц информацией из управляемого программного кода непосредственно на устройстве является отличным решением для множества архитектур приложений Windows Mobile.

Теперь пришло время привести простой пример создания базы данных SQL Everywhere в приложении VB.NET Compact Framework.



Если вам никогда не приходилось создавать приложения Windows Mobile Smart Device в Visual Studio и .Net Compact Framework или если вы вообще не знакомы с языками программирования C# и VB.NET, то перед тем как выполнять пример, лучше ознакомиться со статьей "Getting Started" на сайте:

`http://msdn.microsoft.com/netframework/programming/netcf/gettingstarted/default.aspx`

Если вы готовы начать, запустите Visual Studio 2005 и выполните следующие действия.

1. В Visual Studio 2005 выберите пункт меню File⇨New⇨Project, чтобы открыть диалоговое окно создания нового проекта.
2. В списке типов проектов на левой панели разверните узел Visual Basic или C#, в зависимости от того, какой из этих языков платформы .NET вы предпочитаете, а затем разверните узел Smart Device.
3. Выберите платформу Windows Mobile для своего приложения (Pocket PC 2003, Smartphone 2003 и т.п.).

4. На панели Templates с правой стороны диалогового окна выберите пункт Device Application.
5. Присвойте имя своему проекту и, при желании, измените путь размещения (поле Location) и имя решения (поле Solution Name). После этого щелкните на кнопке OK.
6. Visual Studio 2005 создаст новое приложение .NET Compact Framework Smart Device и для начала сгенерирует простую форму.

Теперь, чтобы сделать приложение совместимым с SQL Everywhere, следует добавить в проект ссылку на поставщика данных ADO.NET и пространство имен `System.Data.SqlServerCe`.

1. В меню Visual Studio выберите пункт Project⇨Add Reference.
2. В открывшемся диалоговом окне (рис. 26.7) во вкладке .NET в списке выберите сборку `System.Data.SqlServer.Ce`. Если в списке эта сборка отсутствует, щелкните на вкладке Browse и выберите файл `<устройство:>\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL Server\Mobile\v3.0\System.Data.SqlServerCe.dll`. Если по данному пути файла `System.Data.SqlServerCe.dll` нет, значит, установка SQL Everywhere на компьютере была проведена некорректно. В этом случае вернитесь к разделу "Установка SQL Server Everywhere".
3. Закройте диалоговое окно добавления ссылки, щелкнув на кнопке OK.
4. Повторите описанное выше действия для пространства имен `System.Data`.

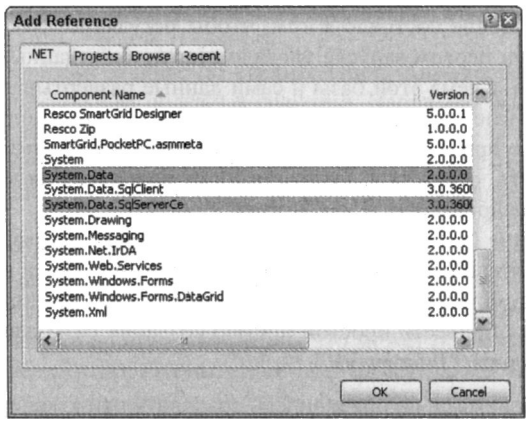


Рис. 26.7. Добавление ссылки на `System.Data.SqlServer.Ce`

Теперь проект Smart Device готов к работе с SQL Everywhere. На панели Solution Explorer программы Visual Studio 2005 щелкните правой кнопкой мыши на форме, автоматически созданной для проекта (например, `Form1.vb`), и выберите в контекстном меню пункт View Code. Измените программный код, чтобы он выглядел так, как в приведенном ниже примере.

Пример программного кода Visual Basic.NET

```
Imports System.IO
Imports System.Data
Imports System.Data.SqlServerCe
Imports System.Text
Imports System.Reflection
```

```
Public Class Form1
  Private _sqlMobileDB As String
  Private _connection As String

  Sub New()
      InitializeComponent()
      Dim strAppDir As String = Path.GetDirectoryName([Assembly].
GetExecutingAssembly().GetModules(0).FullyQualifiedName)
      Me._sqlMobileDB = strAppDir + Path.DirectorySeparatorChar.
ToString() + "ss2005bible.sdf"
      Me._connection = "Data Source=" & _sqlMobileDB
      CreateDatabase()
      CreateSchema()
      InsertRows()
  End Sub

  Private Sub CreateDatabase()
      If Not System.IO.File.Exists(_sqlMobileDB) Then
         Dim eng As SqlCeEngine = New SqlCeEngine(_connection)
         eng.CreateDatabase()
      End If
  End Sub

  Private Sub CreateSchema()
    Dim sql As String = "CREATE TABLE tb_clients (" & _
     "clientId int NOT NULL IDENTITY (1, 1) primary key," & _
     "clientCode nvarchar(50) NULL, " & _
     "clientName nvarchar(50) NOT NULL)"
    Dim conn As SqlCeConnection = New SqlCeConnection(_connection)
    conn.Open()
    Dim cmd As New SqlCeCommand(sql, conn)
    cmd.CommandType = CommandType.Text
    Try
       cmd.ExecuteNonQuery()
    Catch sqlex As SqlCeException
       DisplaySQLCEErrors(sqlex)
    Finally
       conn.Close()
       conn.Dispose()
    End Try
  End Sub

  Public Sub InsertRows()
    Dim sql As String = "INSERT INTO tb_clients (clientCode,
                         clientName) " &_
                        "VALUES (" & _
                        "'12345', " & _
                        "'Acme Corporation')"
    Dim conn As SqlCeConnection = New SqlCeConnection(_connection)
    conn.Open()
    Dim cmd As New SqlCeCommand(sql, conn)
    cmd.CommandType = CommandType.Text
    Try
       cmd.ExecuteNonQuery()
    Catch sqlex As SqlCeException
       DisplaySQLCEErrors(sqlex)
    Finally
       conn.Close()
```

```
      conn.Dispose()
    End Try
  End Sub

  Public Sub DisplaySQLCEErrors(ByVal ex As SqlCeException)
    Dim errorCollection As SqlCeErrorCollection = ex.Errors
    Dim bld As New StringBuilder()
    Dim inner As Exception = ex.InnerException
    Dim err As SqlCeError
    For Each err In errorCollection
       bld.Append(ControlChars.Lf + " Error Code: " +_
        err.HResult.ToString())
       bld.Append(ControlChars.Lf + " Message : " +_
        err.Message)
       bld.Append(ControlChars.Lf + " Minor Err.: " +_
        err.NativeError.ToString())
       bld.Append(ControlChars.Lf + " Source : " + err.Source)
       Dim numPar As Integer
       For Each numPar In err.NumericErrorParameters
         If (numPar <> 0) Then
            bld.Append(ControlChars.Lf + " Num. Par. : " +_
              numPar.ToString())
         End If
       Next numPar
       Dim errPar As String
       For Each errPar In err.ErrorParameters
         If (errPar <> String.Empty) Then
           bld.Append(ControlChars.Lf + " Err. Par. : " +_
              errPar)
         End If
       Next errPar
       MessageBox.Show(bld.ToString(), "SQL Everywhere Error")
     Next err
  End Sub
End Class
```

Скомпилируйте представленный программный код и разверните его либо на мобильном устройстве, подключенном к компьютеру через ActiveSync, либо на одном из эмуляторов, поставляемых в комплекте с Visual Studio 2005.

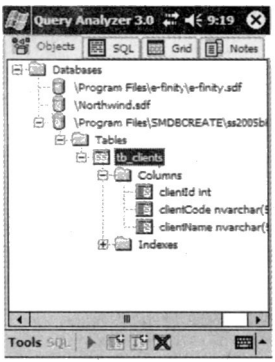


Рис. 26.8. База данных SQL Everywhere, созданная управляемым программным кодом

В нашем примере программный код создает базу данных SQL Everywhere под названием `ss2005bible.sdf` в папке, в которой запускается приложение, что мы и видим в окне Query Analyzer 3.0 на рис. 26.8.

Нажмите пером на зеленой стрелке в нижней части окна Query Analyzer 3.0, чтобы увидеть строку, вставленную в таблицу `tb_clients`.

Если вы знакомы с программированием в ADO.NET в полноценной среде .NET Framework, то будете приятно удивлены, насколько похож программный код в данном примере на тот, который пришлось бы написать в ADO.NET для доступа к SQL Server 2000 или 2005 (например, вместо класса `SqlConnections` мы использовали `SqlCeConnections`, а вместо `SqlCommands` — `SqlCeCommands`). В приведенном примере следует обратить внимание на некоторые моменты. Во-первых, мы не только закрываем подключения, но и удаляем сам объект, после того как работа

сделана. Несмотря на то что SQL Everywhere является многопользовательской базой данных, поддерживающей множество подключений, объект `SqlCeConnections` на мобильном устройстве занимает много памяти. Не следует забывать, что программный код будет запускаться на устройстве, память которого ограничена, а содержание коллекций ненужных элементов — относительно ресурсоемкая операция в среде Compact Framework. Не забывайте закрывать и удалять объект `SqlCeConnections`, а также объекты `SqlCeCommands`, `SqlCeDataReaders`, `RemoteDataAccess` и `Replication`, когда они больше не нужны. Обратите также внимание на использование отражения для определения текущего каталога, в котором запущена программа. Чтобы не приходилось жестко программировать путь к базе данных в коде, лучше размещать ее в том же каталоге, в котором содержится и сама мобильная программа.

Метод `DisplaySQLCEErrors`, приведенный в примере, рекомендуется добавлять во все проекты Smart Device, которые используют базы данных SQL Everywhere. Если вы столкнетесь с трудностями, этот метод обеспечит вас максимально возможной информацией о конкретной ошибке, вызванной исключением SQL Everywhere.

Когда вы приступите к программированию более сложных задач, работающих с SQL Everywhere, то обнаружите, что должен присутствовать класс, служащий интерфейсом ко всем основным операциям SQL Everywhere (`ExecuteQuery`, `ExecuteNonQuery`, `ExecuteResultSet` и т.д.). Существуют дистрибутивы Microsoft Enterprise Data Access Application Block, свободно доступные для SQL CE, которые без труда можно преобразовать для работы с SQL Everywhere. Отличный пример вы можете найти на сайте `http://www.businessanyplace.net/?p=daabcf`.

Создание базы данных SQL Everywhere в Query Analyzer 3.0

Еще одним вариантом создания базы данных SQL Enywhere является использование утилиты Query Analyzer 3.0 непосредственно на мобильном устройстве или в эмуляторе. В первую очередь проверьте, установлена ли эта утилита на устройстве (этот вопрос рассматривался выше в настоящей главе). Далее запустите Query Analyzer 3.0 либо из меню Start, либо из файла `Program Files\SQL Everywhere\locale\isqlw30.exe`. После того как программа будет запущена, перейдите на вкладку Objects и щелкните на желтом значке базы данных и зеленой стрелке на панели инструментов в нижней части окна (рис. 26.9).

Обратите внимание на то, что одно и то же диалоговое окно используется и для подключения к существующей базе данных, и для создания новой. Так как в данном случае мы создаем новую базу, введите ее имя и путь размещения, после чего нажмите пером на кнопке New Database. После этого раскладка элементов окна изменится, позволяя задать порядок сортировки, пароль доступа и/или шифрование содержимого базы. Эти вопросы мы рассмотрим в разделе "Вопросы безопасности". Нажмите пером на кнопке Create в нижней части диалогового окна, после чего база данных с заданным вами именем будет создана в указанном вами каталоге.



Рис. 26.9. Создание базы данных SQL Everywhere в утилите Query Analyzer 3.0



По соглашению базы данных SQL Everywhere всегда имеют расширение `.sdf`, которое ассоциировано с утилитой Query Analyzer 3.0 (если SQL Everywhere установлена на мобильном устройстве). Несмотря на то что у вас есть возможность не задавать это расширение, в этом случае нажатие на значке файла базы данных не приведет к ее открытию в Query Analyzer 3.0.

Теперь вы можете управлять созданной базой данных во вкладке Objects утилиты Query Analyzer 3.0, добавляя в нее таблицы, ограничения, ключи и индексы. В то же время во вкладке SQL вы можете создавать запросы, заполняющие таблицы данными.

Как видите, описанный ручной метод несколько неудобный, учитывая размеры экрана и механизм ввода данных устройства Windows Mobile. Эту методику можно использовать для быстрого создания начальной базы SQL Everywhere на устройстве при отсутствии подключения к компьютеру разработки или репликации слияния. Если база данных будет содержать больше одной-двух таблиц, то следующие описываемые методы покажутся вам удобнее.

Создание базы данных SQL Everywhere в Visual Studio 2005

Одной из основных целей создания пакета Visual Studio 2005 была реализация единого портала, в котором разработчик мог бы писать и тестировать программы, работать с базами данных, просматривать серверы, работать с управляющими библиотеками и даже просматривать Web-страницы без необходимости покидать привычный интерфейс Visual Studio. Одним из наиболее впечатляющих свойств версии Visual Studio 2005 является то, что теперь в ней можно работать с SQL Everywhere. В частности, в Visual Studio 2005 вы можете выполнять следующие операции.

- Создавать базы данных SQL Everywhere, подключаться к ним, создавать к ним запросы. При этом сами базы могут быть размещены как на компьютере разработки, так и на подключенном к нему посредством ActiveSync устройстве Windows Mobile.
- Создавать запросы к базам SQL Everywhere в графическом интерфейсе конструктора запросов.
- Использовать метод перетаскивания для автоматического связывания данных с элементами управления, генерируя при этом программный код для управления этими связями в процессе работы программы.
- Не волноваться о сворачивании и развертывании баз данных SQL Everywhere вместе с проектом Smart Device, так как эти задачи Visual Studio 2005 полностью берет на себя.

Объем настоящей книги позволяет привести лишь беглый обзор этих функций. В следующем примере мы создадим новую базу данных SQL Everywhere в программе Visual Studio 2005. Запустите эту программу и выполните следующие действия.

1. Выполните пп. 1–4 из примера раздела "Создание базы данных SQL Everywhere с помощью управляемого программного кода", дойдя до места, в котором будет существовать пустой проект Smart Device, готовый к работе.
2. В меню Visual Studio 2005 выберите пункт Data⇨Add New Data Source и на панели Where will the application get data from? диалогового окна Choose a Data Source Type щелкните на желтом значке Database. Щелкните на кнопке Next.
3. На следующей странице диалогового окна щелкните на кнопке New Connection. Щелкните на кнопке Change открывшегося диалогового окна Add Connection и выберите в качестве типа источника данных Microsoft SQL Server Mobile Edition. Щелкните на кнопке OK.
4. В диалоговом окне Add Connection вы можете либо создать новую, либо открыть уже существующую базу данных SQL Everywhere, которая находится или на сервере разработки, или на подключенном посредством ActiveSync мобильном устройстве (рис. 26.10). Так как в данном случае нас интересует создание новой базы данных SQL Everywhere, щелкните на кнопке Create.

5. В диалоговом окне создания новой базы данных SQL Server 2005 Everywhere Edition введите путь к новой базе данных либо воспользуйтесь кнопкой Browse. Присвойте файлу базы данных имя типа `SS2005BIBLE.SDF` (рис. 26.11). Отмечу, что в данном примере мы оставим пустым поле для ввода пароля, однако на практике лучше его задавать, и об этом Visual Studio 2005 вас предупредит, когда вы щелкнете на кнопке OK.

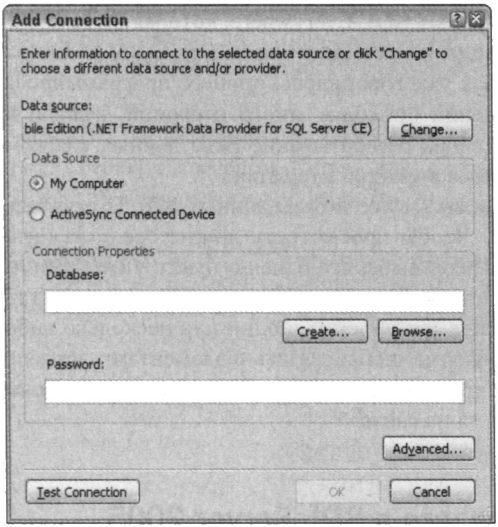


Рис. 26.10. Диалоговое окно Add Connection

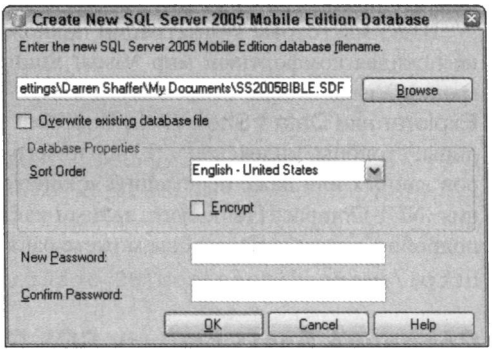


Рис. 26.11. Диалоговое окно создания новой базы данных SQL Everywhere

6. В окне предупреждения о целесообразности задания пароля к базе данных щелкните на OK, чтобы продолжить процесс. Вы вернетесь к диалоговому окну Add Connection, в котором нужно щелкнуть на кнопке OK, чтобы вернуться к мастеру конфигурирования источника данных.
7. Только что созданная база данных теперь будет отображаться в поле Which data connection should your application use to connect the database?. Щелкните на кнопке Next, чтобы продолжить работу с примером.
8. Visual Studio 2005 откроет окно с предупреждением о том, что если вы планируете создавать и развертывать проект вместе с базой данных SQL Everywhere, то последняя должна быть включена в проект. Это прекрасное нововведение, поскольку старые версии Visual Studio расценивали базы данных SQL CE как содержимое, в результате чего при каждом создании проекта эти базы заново разворачивались на мобильном устройстве или в эмуляторе. Когда вы получите это предупреждение, щелкните на кнопке Yes, чтобы скопировать созданную базу данных в текущий проект Smart Device.
9. Теперь мастер конфигурирования источника данных позволит вам создать объект набора данных `DataSet`, составленный из одной или нескольких таблиц SQL Everywhere. Так как в новой базе данных пока еще нет таблиц, на странице Choose Your Database Objects мастера пока еще нечего выбирать. Однако если вы будете подключаться к существующей базе, то можете создать набор данных ADO.NET, который приложение будет использовать для взаимодействия с базой данных. Стоит отметить еще одну деталь. Несмотря на то что Visual Studio уже знает, что источником данных является база данных SQL Everywhere, вы увидите список объектов, содержащий

представления, хранимые процедуры и функции, которые, как мы знаем, этой редакцией в настоящий момент не поддерживаются. Единственным объектом баз данных, который можно использовать для формирования набора данных, являются таблицы.

10. Щелкните на кнопке Finish, чтобы завершить работу мастера.

Обратите внимание на то, что в окне Solution Explorer программы Visual Studio теперь отображается копия только что созданной базы данных SQL Everywhere, которая была добавлена в проект с параметром Build Action of Content и значением параметра Copy to Output Directory, установленным в `Copy if Newer`. Как уже говорилось, процесс программирования/создания/развертывания мобильных приложений проходит много итераций, каждая из которых отнимает достаточно много времени. Теперь развертывание базы данных как компонента проекта уже не будет частью этого процесса в каждой итерации.

Итак, мы готовы к интеграции приложения Smart Device и базы данных SQL Everywhere, не покидая комфортный мир Visual Studio 2005. Чтобы просмотреть другие средства интеграции Visual Studio, относящиеся к SQL Everywhere, выберите в меню пункт View⇨Server Explorer или Data⇨Show Data Sources. В созданную базу данных вы можете добавить таблицы, столбцы, индексы и т.п., после чего включить эти объекты в один или несколько наборов данных или даже перетащить в конструктор форм, чтобы связать с элементами управления .NET Compact Framework, такими как сетки просмотра, комбинированные списки. Более подробно с этим вопросом вы можете ознакомиться на сайте:

`http://msreadiness.com/WS_abstract.asp?eid=15003229`

Создание базы данных SQL Everywhere в SQL Server 2005 Management Studio

Как и Visual Studio, SQL Server 2005 Management Studio позволяет создавать базы данных SQL Everywhere и управлять ими практически так же, как и при работе с базами данных SQL Server 2005. Компания Microsoft продвинула далеко вперед те средства интеграции с мобильными базами данных, которые существовали в SQL Server 2000 Enterprise Manager. Теперь средства интеграции включают следующие.

- Поддержка SQL Editor.
- Возможность создавать, наполнять и модифицировать базы данных SQL Everywhere.
- Управляемый API, способный реализовать бизнес-логику при репликациях слияния.
- Поддержка статей, предназначенных только для загрузки.
- Поддержка фильтрованных статей.
- Графический просмотр планов выполнения запросов.
- Улучшенные мастера конфигурирования репликаций.
- Поддержка SQL Everywhere как приемника службы интеграции SQL Server.

Пришло время рассмотреть пример создания базы данных SQL Everywhere в SQL Server Management Studio, чтобы проиллюстрировать некоторые из этих возможностей. Выполнив следующие действия, мы создадим новую базу данных SQL Everywhere прямо на рабочем столе, даже не подключая мобильное устройство.

1. Запустите SQL Server 2005 Management Studio.
2. В открывшемся диалоговом окне Connect To Server выберите в качестве типа сервера SQL Server Everywhere, а в списке баз данных — пункт New Database.

3. Как только вы выберете New Database из списка баз данных, откроется диалоговое окно, в котором будут отображены имя создаваемой базы SQL Everywhere и путь к ней, а также пароль и порядок сортировки. В поле имени базы данных введите `<устройство:>\<путь>\SS2005BIBLE.SDF`. Оставьте поле пароля пустым и щелкните на кнопке OK.
4. SQL Server 2005 Management Srudio предупредит вас о том, что вы оставили пустым поле пароля доступа к базе данных SQL Everywhere. В реальных приложениях из соображений безопасности следует всегда задавать этот пароль, однако в данном случае щелкните на кнопке Yes, чтобы продолжить работу с примером.
5. Вы снова вернетесь к диалоговому окну Connect to Server с уже заданным типом сервера и именем файла базы данных. Щелкните на кнопке Connect, чтобы закрыть это диалоговое окно (рис. 26.12).

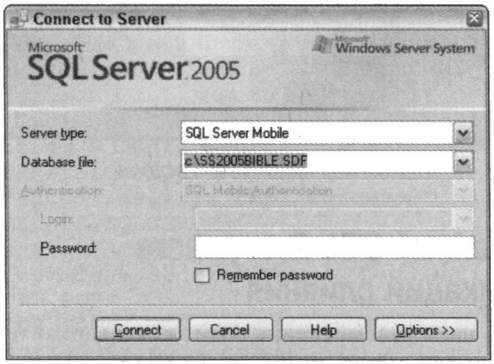


Рис. 26.12. Подключение к базе данных SQL Everywhere в SQL Server 2005 Management Studio

6. На панели Object Explorer утилиты Management Studio появится новый узел только что созданной базы данных SQL Everywhere. Обратите внимание на то, что графическое представление этой базы ничем не отличается от обычных баз данных SQL Server 2005.

Теперь можете щелкнуть в окне Object Explorer правой кнопкой мыши на базе данных SQL Everywhere, выбрать в контекстном меню пункт New Query и работать с этой базой данных, как с любой другой. Кроме выполнения инструкций DDL и DML, вы можете графически проектировать схему базы данных SQL Everywhere. Чтобы попробовать выполнить эту операцию, щелкните правой кнопкой мыши на узле Tables базы данных SQL Everywhere и выберите в контекстном меню пункт New Table. Откроется диалоговое окно (рис. 26.13), позволяющее добавлять в базу данных таблицы без написания кода SQL.

Мы только начали открывать для себя то множество операций, которые можно выполнить над базами данных SQL Everywhere в утилите Management Studio, но компания Microsoft позволит восполнить этот недостаток знаний. В утилите SQL Everywhere Books Online содержатся главы *"How To"*, описывающие выполнение распространенных задач управления базой данных с помощью управляемого программного кода, запросов SQL и интерфейса SQL Server 2005 Management Studio.

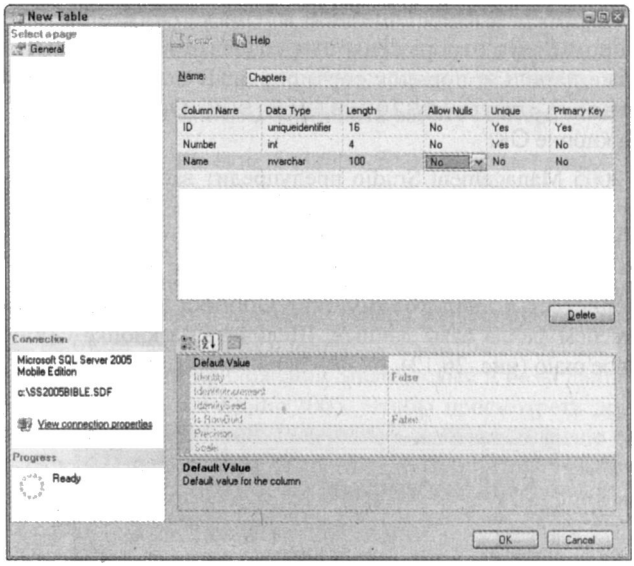


Рис. 26.13. Диалоговое окно New Table

Создание базы данных SQL Everywhere с помощью репликаций слияния

Этой темой мы завершаем рассмотрение возможных вариантов создания базы данных SQL Everywhere. Предлагаемая методика имеет ряд неожиданных преимуществ в развертывании крупных проектов на мобильных устройствах. Единственная сложность, с которой придется столкнуться в мобильных приложениях, использующих SQL Everywhere, — это изменения схемы базы данных после ее развертывания. Репликации слияния предлагают вариант создания новой базы данных подписчиков при первой репликации с SQL Server, и это имеет ряд достоинств.

- Вместо развертывания исходной базы данных SQL Everywhere каждой версией мобильного приложения, вы можете получить полную и обновленную базу данных, созданную на устройстве динамически, при первом выполнении мобильного приложения.
- Если необходимо внести существенные изменения в схеме или изменения, которые не могут быть автоматически распространены среди подписчиков в репликациях слияния с сервером, вы можете обязать мобильных пользователей загрузить все изменения, удалить базу данных SQL Everywhere на устройстве и перезапустить приложение. В результате будет создана новая база данных с самой свежей схемой.
- Если по какой-то причине какой-либо мобильный пользователь удалил или повредил без возможности восстановления базу данных SQL Everywhere на своем устройстве, новая может быть создана динамически, без необходимости установки на устройство новых файлов.
- Если существует серверная база данных и вы хотите создать базу SQL Everywhere, отражающую всю схему или ее подмножество, можете опубликовать серверные статьи, которые хотите включить в базу SQL Everywhere, и автоматически сгенерировать базу SQL Everywhere для подписчиков серверной публикации.

Репликация слияния будет детально рассмотрена далее в этой главе, сейчас же приведем пример, демонстрирующий использование параметра `AddOption.CreateFatabase` репликации слияния SQL Everywhere.

Пример программы VB.NET

```
Imports System.Data
Imports System.Data.SqlServerCe
'Предлагаемый программный код следует поместить в конструктор
'или в событие Load объекта или формы Startup
'проекта Smart Device, чтобы репликация всегда выполнялась
'при загрузке приложения. Если база данных SQL Everywhere
'не будет найдена локально, она будет создана.
Dim repl As SqlCeReplication = GetReplication()
repl.Synchronize()
'Все свойства репликации, приведенные ниже, должны быть
'изменены, чтобы соответствовать среде вашей репликации
'слияния. Значения, которые вы видите в тексте программы,
'использованы исключительно в качестве иллюстрации.
Public Function GetReplication() As SqlCeReplication
Dim databaseName As String = "\My Documents\SS2005BIBLE.SDF"
Dim connection As String = "Data Source=" & databaseName
Dim repl As New SqlCeReplication()
repl.InternetUrl = "http://dellxps/repl/sqlcesa30.dll"
repl.Publisher = "DELLXPS\SQLSERVER"
repl.PublisherDatabase = "Adventureworks"
repl.PublisherSecurityMode = SecurityType.NTAuthentication
repl.Publication = "AW_PUB"
repl.Subscriber = "DDS_PPC"
repl.SubscriberConnectionString = connection
'Параметр AddOption.CreateDatabase, показанный ниже,
'позволяет приложению запускаться без присутствия базы данных
'SQL Everywhere на устройстве, и затем создавать ее
'динамически при первом вызове функции Synchronize().
  If Not System.IO.File.Exists(databaseName) Then
    repl.AddSubscription(AddOption.CreateDatabase)
  End If
Return repl
End Function
```

Модернизация существующей базы данных SQL CE 2.0

Если есть мобильная база данных SQL CE 2.0 и вы хотите модернизировать ее до SQL Everywhere, то компания Microsoft предлагает для выполнения этой операции специальную утилиту командной строки. Замечу, что эта утилита запускается только на мобильном устройстве, ее нельзя выполнить на настольном компьютере.

В первую очередь вам нужно иметь на мобильном устройстве установленные ядра обеих баз данных: SQL Server CE 2.0 и SQL Everywhere. Эти два продукта успешно "уживаются" на одном устройстве. Установите последовательно дистрибутив SQL CE CAB, соответствующий операционной системе и архитектуре процессора, а затем SQL Everywhere CAB.

С помощью File Explorer или ActiveSync скопируйте файл `upgrade.exe`, расположенный в каталоге `<устройство:>\Program Files\Microsoft Visual Studio 8\`

`SmartDevices\SDK\SQL Server\Mobile\v3.0\[платформа]\[процессор]`, на мобильное устройство, поместив его в папку, содержащую базу данных SQL CE 2.0, которую вы собираетесь модернизировать. (Для устройств Windows Mobile 5/Windows CE 5.0 также существует соответствующая папка.)

Если вы щелкнете на файле `upgrade.exe` на своем устройстве, откроется уведомление о синтаксисе использования этой утилиты командной строки (рис. 26.14).

Откройте на устройстве меню Run, нажав и удерживая клавишу <Enter> или <Action>. Затем щелкните на значке часов в правом верхнем углу экрана и удерживайте кнопку мыши, отпустите клавишу <Enter>, выберите Run и введите строку **`upgrade /s [`*`имя_старой_базы_SQLCE`*`] /d [`*`имя_новой_базы_SQL_Everywhere`*`]`**.

В нашем примере мы модернизируем базу данных IbuySpyStore из SQL CE 2.0 в SQL Everywhere, для чего введем в меню Run устройства следующую строку:

```
Upgrade /s IbuySpyStore.sdf /d ibs30.sdf
```

На рис. 26.15 показаны файл получившейся базы данных SQL Everywhere `ibs30.sdf` и уведомление об успешном завершении операции.

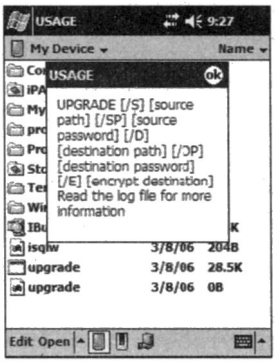


Рис. 26.14. Уведомление о синтаксисе использования утилиты модернизации базы данных SQL CE до SQL Everywhere

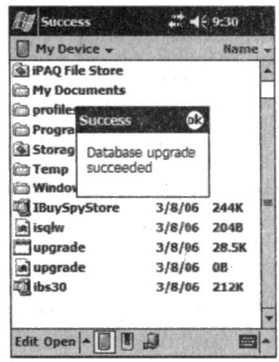


Рис. 26.15. Вывод утилиты модернизации базы данных в SQL Everywhere

Следует отметить, что исходная база данных осталась на месте. Если во время модернизации возникнут какие-либо проблемы, то будет создан файл протокола с их описанием. Этот файл можно открыть и прочитать с помощью программы Pocket Word. В общем случае база данных результата будет на 10–17% меньше, чем исходная (в нашем примере она уменьшилась с 276 до 236 Кбайт). Если исходная база содержала какие-либо столбцы с типом `ntext`, то результирующая база будет больше.

В заключение следует сказать, что утилита upgrade.exe позволяет задать пароль для исходной базы и флаг шифрования для результирующей базы SQL Everywhere (вопросы безопасности мы рассмотрим далее в отдельном разделе).

Синхронизация данных

До сих пор в этой главе мы рассматривали вопросы, связанные с подключением мобильных приложений к реляционным базам данных SQL Everywhere. В этой главе будет показано, как SQL Everywhere может служить автономным кэшем данных в отношениях синхронизации с SQL Server. Как уже говорилось, SQL Everywhere поддерживает две различные, но одновременно мощные, технологии синхронизации данных: доступ к удаленным данным (RDA) и

репликацию слияния. Концепция репликации слияния, скорее всего, хорошо знакома администраторам SQL Server. В то же время технология RDA специфична для синхронизации данных с мобильными устройствами. В этом разделе мы рассмотрим обе эти технологии и еще одну дополнительную — Web-службы .NET.

Перед тем как начать, важно осознать все сложности, связанные с синхронизацией данных с мобильным устройством. В простейшем сценарии рабочий устанавливает связь с сервером, получает с него операционные данные, затем отключается и занимается автономно своей работой. В конце дня он снова устанавливает связь с сервером и отправляет на него все новые или измененные данные. На первый взгляд, все звучит достаточно просто.

Однако что случится, если в течение дня данные на сервере изменятся? А теперь представим ситуацию, когда несколько рабочих получают копию одних и тех же операционных данных на разные мобильные устройства и параллельно вносят в них изменения, в то время как на сервере данные также изменяются. Синхронизация, выполняемая два раза в день, становится скорее исключением, чем правилом. Это связано с ростом популярности беспроводных сетей и мобильных устройств. В такой ситуации вряд ли можно найти промежуток времени, в который данные вообще не изменяются ни в одной точке сети.

Можно представить себе множество еще более сложных сценариев, однако вывод заключается в том, что планирование стратегии синхронизации данных оказывается не такой простой задачей, как выбор технологии. На выбор правильной стратегии мобильных приложений влияют множество дополнительных факторов, в том числе:

- уровень логического разделения данных среди больших популяций мобильных пользователей;
- частота подключений и синхронизации;
- срок жизни батареи устройства;
- объем данных;
- полоса пропускания сети;
- шлюзы системы безопасности;
- требования к разрешению конфликтов.

Принимая все это во внимание, рассмотрим, какая технология лучше подходит для каждого из сценариев.

Удаленный доступ к данным

Удаленный доступ к данным (далее RDA) можно представить себе как модель доставки–отправки при синхронизации данных. RDA позволяет мобильному приложению доставлять все данные или их часть из одной или нескольких таблиц из SQL Server в SQL Everywhere, а затем включать в последнем отслеживание изменений в этих данных. Следует заметить, что в момент инициализации доставки таблица не должна существовать — она и ее схема создаются в SQL Everywhere в процессе каждой доставки. Затем SQL Everywhere отправляет данные назад на сервер, и все отслеженные изменения применяются к базе данных SQL Server. Процесс RDA можно запускать с отключенным отслеживанием изменений; в этом случае отправка данной таблицы не выполняется. RDA также предлагает механизм повторной отправки на сервер любой инструкции SQL для выполнения до тех пор, пока она не вернет какие-либо строки. Это простейший способ выполнения быстрого обновления на сервере в обход SQL Everywhere.

Используя ту же архитектуру синхронизации, доставку и отправку можно организовать с помощью агента сервера SQL Everywhere в Internet Information Server. Службу удаленного доступа к данным проще запускать, чем репликацию слияния, в то же время он требует

большего объема программирования в мобильном приложении (хотя этот код повторяемый и хорошо документированный). Отмечу, что все аспекты RDA управляются программным путем из мобильного приложения.

RDA является отличным решением для загрузки классификаторов (например, почтовых индексов или таблиц заказчиков), а также больших объемов данных для заполнения таблиц SQL Everywhere. Данные при перемещении сжимаются, в результате чего скорость их передачи значительно выше, чем в репликации слияния. Метод отслеживания изменений также удобен для отправки данных, введенных на мобильном устройстве, назад на сервер, с такой же высокой скоростью.

Существует один важный аспект RDA, который следует учитывать в архитектуре синхронизации данных, — это *оптимистическая конкуренция*. RDA использует оптимистическую конкуренцию, однако с чуть более смещенным определением, чем SQL Server. В данном случае оптимистическая конкуренция предполагает, что когда изменения отправляются назад на сервер, они применяются к серверным таблицам, *даже если те же данные уже были на нем изменены* с момента их доставки на данное устройство. Разумеется, это делает непригодным метод RDA для мобильных приложений, в которых промежуток времени между доставкой и отправкой велик, а данные на сервере и других мобильных устройствах изменяются часто.

Еще одним проблемным вопросом, связанным с RDA, является приращение столбца идентичности, когда множество мобильных устройств доставляют данные из одних и тех же серверных таблиц. Например, если одному мобильному пользователю доставлены данные и в этот момент самое большое значение идентификатора было 32, то следующая запись, которую он вставит в таблицу, получит идентификатор 33. Естественно, если одновременно те же данные получал и второй пользователь, следующая запись в его таблице получит тот же идентификатор. На сервере при обновлении таблиц методом RDA не предусмотрено разрешение конфликтов, поэтому в данном примере произойдет ошибка синхронизации при выполнении отправки данных. Вы можете вручную управлять значениями идентичности при выполнении каждой доставки, но RDA все равно найдет место, где не существует пересечения данных, которые мобильный пользователь будет модифицировать или добавлять, в пространстве доставок на мобильные устройства.

Репликация слияния

Репликацию слияния можно рассматривать как модель "издатель–подписчик" при синхронизации данных. Это мощная и сложная технология, которая попыталась приспособиться к самым сложным сценариям синхронизации мобильных данных. В то время как RDA управляется программным путем из мобильных приложений, большая часть логики правил синхронизации репликации слияния реализуется на сервере. Репликация слияния требует меньшего объема программирования на мобильном устройстве, чем RDA, в то же время процесс ее конфигурирования и управления более сложный и занимает больше времени.

Репликация слияния предполагает создание на сервере SQL Server 2000 или 2005 *публикаций*, в каждой из которых определены одна или несколько *статей* (в репликациях SQL Everywhere они лимитированы таблицами, столбцами, индексами, ключами и ограничениями), содержащих строки данных, которые затем фильтруются по вертикали или по горизонтали. Сервер рассматривается как *издатель*, а база данных SQL Everywhere — как *подписчик*. *Подписка* на публикации добавляется в базу данных SQL Everywhere и синхронизируется с публикациями. Эта синхронизация организована в виде двустороннего диалога между подписчиком и издателем, предназначенного для обработки изменений строк и схемы данных на обоих концах диалога.

Новым в SQL Server 2005 и SQL Everywhere является введение отслеживания изменений на уровне столбцов для репликации слияния. В качестве примера рассмотрим таблицу

SurveyAnswers, являющуюся частью публикации. Эта таблица состоит из пятнадцати столбцов, один из которых имеет тип image и содержит подпись или другое цифровое изображение (этот столбец может быть очень широким). Если все изменения в строке с момента последней синхронизации происходили в столбце типа nvarchar, то только это значение при репликации будет передано на сервер. В репликациях SQL CE и SQL Server 2000 в данном случае на сервер была бы отправлена вся строка, включая неизмененное изображение.

Мобильные реализации на больших предприятиях получат выигрыш не только в виде расширенных возможностей синхронизации, но и от набора административных средств на стороне сервера. В корпоративном центре данных администратор может использовать *монитор репликаций*, входящий в состав утилиты Management Studio, для проверки состояния всех мобильных подписчиков, времени их последней синхронизации, достигнутой при этом пропускной способности, а также конфликтов, которые могли возникнуть в процессе синхронизации. Администратор также может повторно инициализировать любую мобильную подписку, чтобы заставить подписчика получить новый *снимок* публикации с сервера. Снимком называют представление схемы всех статей публикации, сценариев создания этих статей в SQL Everywhere, а также исходного набора данных, наполняющего подписку в SQL Everywhere. SQL Agent на сервере генерирует снимки, которые агент сервера SQL Everywhere знает, где найти, основываясь на конфигурации публикации для Web-репликации в SQL Everuwhere.

Полноценный пример создания приложения Compact Framework, подключенного к базе данных SQL Everywhere, связанной репликацией слияния с базой SQL Server, вы найдете на сайте:

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/
dnppcgen/html/desktop_device_tools_sql_mobile.asp
```

Web-службы

Среда .NET Compact Framework 2.0 имеет встроенную поддержку вызова Web-служб и синтаксического разбора XML. Фраза "Разработка стратегии синхронизации данных, основанной на Web-службах" может звучать устрашающе, однако на самом деле в хороших мобильных сценариях эта стратегия может быть реализована гораздо меньшими усилиями, чем конфигурирование репликации слияния. Вот несколько признаков того. что подход с применением Web-служб имеет смысл использовать в конкретной мобильной архитектуре.

- Данные, запрашиваемые и/или создаваемые конкретным мобильным пользователем, могут быть логически или принудительно отделены от других мобильных пользователей. Другими словами, либо возможность конфликтов синхронизации настолько мала, что ею можно пренебречь, либо существует возможность написать программный код, разрешающий эти конфликты.
- Факты, обрабатываемые мобильным приложением, используются конкретным пользователем. Другими словами, два или более мобильных пользователя, запустившие одно и то же приложение, не выполняют одновременно одну и ту же единицу работы. Сценарий, в котором три разных рабочих склада вводят данные об одном и том же только что прибывшем контейнере, имеет большую вероятность возникновения конфликтов данных.
- Объем данных, которыми мобильный пользователь обменивается с сервером, в среднем меньше 1 Мбайт.
- Существует типичный сценарий подключения устройства к серверу, в котором задействован Интернет и данные проходят через множество брандмауэров и/или прокси-серверов.
- Существуют требования к уровню управления над составом публикуемых статей, фильтрами и разделением, преобразованием данных или процессом администрирования синхронизации, которые невозможно реализовать с помощью репликации слияния.

- Существуют серверные базы данных, отличные от SQL Server, с которыми нужно выполнять синхронизацию.

Если ваша ситуация согласуется с одним из этих сценариев, то лучше остановиться на синхронизации с помощью Web-служб. На практике наиболее эффективные реализации синхронизации с помощью Web-служб имеют следующие общие черты.

- Гарантируется уникальность всех данных в доменах решений, обычно с помощью использования в качестве первичных ключей уникальных идентификаторов, таких как GUID, во всех синхронизируемых таблицах данных.
- Объем данных в одном вызове Web-службы с мобильного устройства не превышает 1 Мбайт.
- Для гарантии отсутствия отклоненных синхронизируемых записей используется стратегия полной замены в соединении с наличием выходного буфера на устройстве. Например, если существует таблица `Orders`, в которой мобильные пользователи обновляют или вставляют данные, первичный ключ идентификатора GUID каждой новой, изменяемой или удаляемой записи помещается в простую таблицу `Orders_Outbox` (выходной буфер). Таблица `Orders_Outbox` имеет два столбца: столбец идентификатора GUID записи таблицы `Orders`, с которой проводилась работа, и столбец даты-времени, чтобы применять множество изменений данной записи в правильном порядке.
- Для получения данных с SQL Server и помещения их туда используются простые Web-методы. В любом из направлений переправляется объект `DataSet` с одним или несколькими объектами `DataTable`, соответствующими конкретным таблицам в базе данных SQL Everywhere.
- Когда приходит время синхронизации SQL Everywhere с сервером, создается объект `DataSet` с одним объектом `DataTable` для каждого выходного буфера, описанного выше. Каждая из таблиц заполняется полными строками данных исходной таблицы, соответствующими идентификаторам GUID в выходном буфере. Этот объект `DataSet` отправляется на сервер, откуда возвращается объект `DataSet` с тем же набором объектов `DataTable`, однако на этот раз таблицы заполнены идентификаторами отправленных строк и кодами состояния, указывающими на успех или неудачу операции. Идентификаторы GUID записей, которые были успешно обработаны, удаляются из соответствующего выходного буфера. Те же записи, которые не были обработаны, остаются в буфере для повторной отправки при последующих синхронизациях.
- При доставке данных с сервера, если не существует видимых записей в выходном буфере таблицы, эта таблица может быть сжата и заменена результатами вызова Web-службы. Например, вы можете использовать Web-метод с заголовком `DataSet GetNamedTable (string TableName)`, который возвращает объект `DatSet` с одним объектом `DataTable`, заполненным строками из запрошенной таблицы сервера. На мобильном устройстве эти строки могут быть применены к соответствующей таблице SQL Everywhere с помощью метода прямой замены. Довольно часто в этом случае применяют критерии фильтров (предложения `WHERE`) и концепцию сравнения даты для доставки с сервера только новых и измененных записей, начиная с некоторого момента времени.

Несмотря на то что конкретные условия вашей задачи будут диктовать детали реализации синхронизации с использованием Web-служб, общий подход, описанный в настоящем разделе, можно относительно быстро реализовать, к тому же он очень эффективный.

Упаковка и развертывание

Существует масса вариантов развертывания мобильного приложения, включающего базу данных SQL Everywhere. Сложность состоит не только в доставке SQL Everywhere и поддерживающих библиотек репликации на устройство, но и в создании исходных баз данных SQL Everywhere, необходимых для первого запуска приложения.

Среди вариантов развертывания SQL Everywhere следующие.

- Ручное копирование необходимых файлов CAB на все устройства.
- Подключение устройств к персональному компьютеру или серверу с помощью ActiveSync и развертывание мобильных приложений с помощью Visual Studio 2005.
- Доставка файлов CAB SQL Everywhere с Web-страницы сервера IIS с помощью программы PocketIE подключенного устройства.
- Использование утилит управления автоматизированным конфигурированием устройства, например созданных компаниями Odyssey Software или Symbol Technologies.

В этом разделе мы рассмотрим все эти варианты развертывания базы данных SQL Everywhere в совокупности мобильных устройств.

Программное создание базы данных

Несмотря на то что этот вариант кажется наименее привлекательным с практической точки зрения, он является единственным, когда база данных SQL Everywhere автоматически создается и наполняется во время первого запуска приложения на устройстве. Этот метод может оказаться полезным в ситуациях, когда приложение имеет относительно небольшой объем статических данных. Этот подход обычно реализуется с помощью чтения файлов CSV, развертываемых вместе с мобильным приложением, загрузки из них данных в таблицы и последующего удаления этих файлов.

Развертывание исходной базы данных вместе с мобильным приложением

Пожалуй, этот подход является самым распространенным. Он предполагает предварительное создание исходной базы данных и ее включение в проект в качестве содержимого. Исходная база данных предварительно заполнена классификаторами или другими статичными данными, а при первой синхронизации она получает с сервера необходимые оперативные данные.

Динамическое создание базы данных с помощью репликации слияния

Еще один подход, рассмотренный ранее в этой главе, предполагал динамическое создание базы данных с помощью репликации слияния и установки в объекте репликации параметра `AddOption.CreateDatabase`. Кроме достоинств небольшого объема развертывания, такой подход позволяет новой базе данных развернуться на устройстве путем удаления старой базы данных и запуска процесса репликации. Это простейший способ распространения такого количества изменений в схеме базы данных, которое превосходит возможности параметра Propagate Schema Changes в репликации слияния. В сценариях с ограниченной полосой

пропускания подобный подход не рекомендуется использовать, учитывая время, которое может потребоваться на динамическое создание базы данных с помощью репликации слияния.

Прочие подходы

Следует выделить ряд дополнительных подходов к решению данной задачи, включая следующие.

- Доставка файлов CAB, содержащих базу данных SQL Everywhere, на мобильное устройство с помощью программы PocketIE.
- Развертывание баз данных с карт памяти, которые предоставляются мобильным пользователям и вставляются в соответствующие слоты устройств.
- Реализация произвольного механизма обновления приложения; пример, предложенный MSDN, можно найти на сайте:

  ```
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/
  dnnetcomp/html/AutoUpdater.asp
  ```

Вопросы безопасности

Обеспечение безопасности особенно важно в условиях мобильных корпоративных реализаций. Мобильные устройства могут быть легко утеряны, похищены, повреждены или использованы кем-то, кроме владельца. Синхронизация данных с SQL Everywhere может осуществляться посредством подключения к незащищенным сетям. По этим причинам компания Microsoft предлагает множество способов защиты корпоративных данных, хранимых в базе данных SQL Everywhere и передаваемых во время синхронизации.

Защита паролем

В первую очередь напомним, что база данных SQL Everywhere — это всего лишь файл, находящийся на мобильном устройстве клиента. Предположим, что недобросовестный пользователь получил доступ к утилите Query Analyzer 3.0 или пользовательскому приложению, подключенному к вашему файлу базы данных. Первой линией обороны в такой ситуации может стать защита базы данных паролем. Пароли могут быть определены во время создания базы данных, а также добавлены или изменены после создания во время работы с ядром базы данных. Пароль может содержать не больше 40 символов и включать в себя буквы, символы и цифры. После добавления пароля к базе данных невозможно будет получить доступ без его указания в строке подключения ADO.NET или OLE DB (или в запросе Query Analyzer 3.0).



Не забывайте и не теряйте назначенный базе данных пароль, в противном случае невозможно будет получить доступ к базе или изменить установки так, чтобы она не запрашивала пароль. Пароли SQL Everywhere не подлежат восстановлению.

Шифрование

Следующей линией обороны в системе безопасности базы данных SQL Server является шифрование ее содержимого. Подобно защите паролем, шифрование определяется либо на этапе создания базы, либо с помощью операций с ядром базы данных. В шифровании применяется собственный 128-битовый алгоритм компании Microsoft. Компания Microsoft держит

специфику этого алгоритма в строжайшей тайне, что имеет свои преимущества для пользователей. Хотелось бы особо отметить, что применение шифрования к базе данных SQL Everywhere практически никак не сказывается на ее производительности. В качестве примера скажу, что шифрование базы данных SQL Everywhere объемом в 5 Мбайт привело к падению производительности на 2% наиболее интенсивных инструкций `SELECT`, `INSERT` и `UPDATE`.

Шифрование в сочетании с надежным паролем — лучший способ защиты данных SQL Everywhere, особенно при использовании съемных карт памяти на мобильных устройствах.

Безопасная синхронизация данных

При использовании доступа к удаленным данным репликации слияния или собственной стратегии синхронизации с использованием Web-служб защита передаваемых во время синхронизации данных исключительно важна. Так как все эти стратегии предполагают использование протокола HTTP и сервера IIS в качестве шлюза, вы можете использовать параметры защиты IIS для создания безопасной модели синхронизации.

Начать стоит с использования протоколов SSL и HTTPS для безопасной транспортировки. Для этого создайте и сконфигурируйте для каждого мобильного приложения отдельный виртуальный каталог, с которым будет выполняться синхронизация, после чего сконфигурируйте их с принципами наименьших привилегий для каждого из конкретных приложений. Использование на сервере IIS файловой системы NTFS (а не FAT) позволит дополнительно ограничить объем и состав подписчиков, имеющих доступ к данному виртуальному каталогу. И еще, никогда не используйте анонимную аутентификацию для любой синхронизации данных с базами SQL Everywhere.

Между IIS, агентом SQL Everywhere Server и SQL Server 2005 можно установить дополнительные посты безопасности. Детальное описание этих возможностей вы найдете в статье "Planning Server Security" утилиты SQL Server 2005 Mobile Edition Books Online.

Настройка, обслуживание и администрирование

Подобно любой реляционной базе данных, находящейся в промышленной эксплуатации, для достижения оптимальной производительности и продолжительной безошибочной работы SQL Everywhere требует настройки и обслуживания. В этом разделе вам будут представлены основные концепции настройки производительности SQL Everywhere; также мы пройдемся по постоянно расширяющемуся множеству методов поддержания производительности и целостности мобильных баз данных. В то время как большинство приложений, в основном выполняющих операции чтения из баз данных SQL Everywhere, могут работать бесконечно долго, не требуя к себе повышенного внимания, те приложения, которые интенсивно извлекают, хранят и синхронизируют данные, требуют как обслуживания, так и периодической подстройки, чтобы в полной мере реализовать свой потенциал.

Измерение производительности выполнения запросов и ее повышение

Естественно, вопрос настройки производительности запросов не может вместиться в рамки рассмотрения настоящей главы. В то же время практический опыт эксплуатации SQL Everywhere указывает на небольшой набор действенных средств повышения производительности.

- Добавляйте в таблицы SQL Everywhere индексы, однако подходите к этому вопросу избирательно. В составных индексах те столбцы, по которым отбор выполняется чаще, помещайте левее.
- Замените подзапросы объединениями. Процессор запросов SQL Everywhere всегда переписывает подзапрос `IN` с помощью объединений `JOIN` и затем рассматривает совершенно отличный план выполнения.
- Избегайте объединения больше пяти таблиц.
- Используйте параметризованные запросы и подготавливайте `SqlCeCommands` для повторяющихся задач, таких как вставка в таблицу сотен строк.
- Возвращайте из запросов как можно меньше данных — берите только то, что нужно инструкции SQL и репликации.
- Для запроса или обновления больше одной строки используйте, когда это возможно, `SqlCeResultSet`.
- Денормализуйте базу данных, когда остальные методы не помогают добиться достаточной производительности.
- Минимизируйте и по возможности даже избегайте ограничений в таблицах.
- Если база данных SQL Everywhere находится на карте памяти, исследуйте скорость обмена данными карты и не вынимайте ее из устройства, пока существуют открытые подключения к базе данных.

Подробную информацию о вопросах настройки производительности вы можете получить на сайте SQL Server 2005 Performance по адресу `http://msdn.microsoft.com/sql/learning/perf/default.aspx`.

Благодаря интеграции SQL Everywhere с SQL Server 2005 Management Studio теперь стало возможным просматривать в графическом представлении план выполнения любого запроса DML, производительность которого вы хотите повысить в SQL Everywhere. Выберите в меню утилиты пункт Query⇨Display Estimated Execution Plan, и вы увидите графическое представление плана; к тому же при желании вы можете его сохранить в формате XML. Это может оказаться полезным, когда в одном регионе запрос выполняется крайне медленно, а в другом регионе была выполнена его настройка и скорость стала гораздо выше.

Обслуживание SQL Everywhere

Ядро хранилища SQL Everywhere было полностью переработано с целью уменьшения необходимости обслуживания и вероятности повреждения данных, чего так требовал опыт эксплуатации SQL CE. Файлы баз данных SQL Everywhere поделены на логические единицы (или "страницы") емкостью в 4 Кбайт. Во время хранения, индексации и удаления данных на страницах могут оставаться незаполненные участки, а некоторые страницы базы данных могут оказаться и полностью пустыми. Для оптимизации производительности и экономии пространства на устройстве клиента нужно перераспределять неиспользуемое пространство, периодически пересчитывать статистику индексов и очищать буфер транзакций. SQL Everywhere для выполнения этих задач реализует концепции автосжатия, уплотнения и AutoFlush.

Автосжатие

Автосжатие является новой функцией SQL Everywhere, которая автоматически реорганизует страницы файла базы данных так, чтобы пустые страницы переместить в конец файла, а затем обрезать их, уменьшая размер файла. Автосжатие можно настроить, определяя параметр `AutoShrink_Threshold` в строке подключения. В этом параметре указывается процент

свободного пространства в базе данных, по достижении которого выполняется автосжатие. Не устанавливайте для этого свойства слишком большое значение, поскольку сам процесс автосжатия имеет малое воздействие на ресурсы процессора и общую производительность.

Если вы хотите выполнить сжатие файла базы данных вручную, то можете воспользоваться специальным методом объекта `SqlCeEngine` под названием `AutoShrink`, выполняющим немедленное сжатие, независимо от текущего объема свободного пространства в базе данных.

Уплотнение

Уплотнение выполняется с помощью метода `Compact` объекта `SqlCeEngine`. Он позволяет перераспределить пространство, а также изменить такие настройки базы данных, как пароль, идентификатор региона и шифрование. В отличие от автосжатия уплотнение не только перераспределяет пустые страницы, но и создает полностью новую базу данных, которая имеет следующие преимущества.

1. Все индексы создаются заново, а статистика пересчитывается.
2. Страницы таблиц реорганизуются в последовательные.
3. Столбцы идентичности переустанавливаются так, чтобы следующее доступное значение было на один шаг больше максимального текущего.
4. К новой базе данных применяются все изменения пароля, шифрования и/или идентификатора региона.

Приведем пример использования метода `Compact` в базе данных SQL Everywhere.

Пример на языке C#

```
string originalDB = "AWMobile.sdf";
string compactedDB = "AWMobile.sdf.tmp";
SqlCeEngine engine = new SqlCeEngine("Data Source = " + originalDB);
try
{
   engine.Compact("Data Source = " + compactedDB);
   engine.Dispose();
   File.Delete(originalDB);
   File.Move(compactedDB, originalDB);
}
catch(SqlCeException e)
{
   DisplaySQLErrors(e);
}
finally
{
   engine.Dispose();
}
```

С помощью SQL Everywhere и нового объекта `SqlCeEngine` среды Compact Framework 2.0 уплотнение может быть выполнено непосредственно в исходной базе. Это значит, что в свойстве `Data Source` не указывается база данных назначения, вместо этого переписывается исходная база. Таким образом, приведенный выше пример можно преобразовать следующим образом:

```
string originalDB = "AWMobile.sdf";
SqlCeEngine engine = new SqlCeEngine("Data Source = " + originalDB);
try
{
```

```
    engine.Compact();
    engine.Dispose();
}
catch(SqlCeException e) {}
finally
{
    engine.Dispose();
}
```

AutoFlush

SQL Everywhere поддерживает транзакции посредством интерфейсов ADO.NET и OLEDBCE. Буфер транзакций содержит изменения, которые ожидают подтверждения транзакций и последующей записи в базу данных SQL Everywhere. Этот буфер гарантирует, что транзакции всегда будут записаны в базу данных в том порядке, в котором они подтверждены. Транзакции могут быть подтверждены, но еще не записаны в базу данных ввиду ненормального завершения программы или блокировок, установленных другими процессами, записывающими собственные транзакции. Интервал времени, в течение которого изменения, хранящиеся в буфере, должны быть записаны в базу данных, в SQL Everywhere можно настроить с помощью параметра `Flush Interval` строки подключения. В этом параметре определяется максимальное количество секунд, которые должны истечь, прежде чем подтвержденные транзакции будут записаны в базу данных.

Восстановление поврежденной базы данных SQL Everywhere

В ходе эксплуатации SQL CE компания Microsoft собирала от разработчиков мобильных приложений сведения об инцидентах повреждения баз данных. В результате она пришла к выводу, что ядро хранилища данных SQL CE имеет фундаментальную брешь. Оно не учитывало то, что мобильные устройства имеют кратковременный срок действия аккумуляторов и съемные карты памяти. Как уже говорилось, ядро хранилища данных SQL Everywhere было полностью переписано и теперь обеспечивает безопасность мобильных данных. Любая операция записи в базу данных остается в области теневой копии, равно как применяется и к самой базе данных. Таким образом, в результате потери напряжения или перезагрузки устройства SQL Everywhere гарантирует, что все незаконченные операции будут завершены в нужном порядке и не будут утеряны. В этом разделе мы рассмотрим новые возможности проверки и восстановления баз данных SQL Everywhere, которые разработчик может включить в план обслуживания базы данных.

Проверка

Проверка выполняется с помощью метода `Verify` объекта `SqlCeEngine`, который определяет необходимость выполнения восстановления данных. Метод `Verify` вычисляет контрольные суммы всех страниц базы данных SQL Everywhere и сравнивает их с ожидаемыми. Если эти значения не совпадают, то метод возвращает значение `false`, указывая, что необходимо выполнить восстановление данных.

Восстановление

Несмотря на то что ядро хранилища данных SQL Everywhere было полностью переписано, чтобы максимально уменьшить вероятность повреждения данных, тот фундаментальный факт, что SQL Everywhere все равно работает на мобильных устройствах, указывает на необходимость

обработки ситуации, когда повреждения все-таки произошли. Представьте себе ситуацию, в которой база данных SQL Server 2005 запущена на сервере, а пользователь имеет возможность в течение дня просто отключать питание или извлекать съемные диски, содержащие файлы данных. Базы данных SQL Everywhere часто помещают на съемные карты памяти мобильных устройств, которые потенциально можно извлечь в процессе выполнения транзакции записи. Исходя из этого, рекомендуется периодически проверять базы данных SQL Everywhere и при получении отрицательного результата вызывать метод `Repair` объекта `SqlCeEngine`.

К тому же теперь компания Microsoft предлагает два параметра метода `Repair`, определяющие, что делать, если будут найдены поврежденные данные.

1. `RepairOption.DeletedCorruptedRows`. Удалить поврежденные строки из базы данных SQL Everywhere.
2. `RepairOption.RecoverCorruptedRows`. Выполнить попытку прочитать данные с поврежденной страницы, однако успех операции не гарантирован. К тому же, даже если удалось восстановить одну или несколько строк, нет никакой гарантии, что в базе данных не осталось логических повреждений. Как бы там ни было, наличие дополнительного варианта действий все равно является шагом в правильном направлении.

Следующий фрагмент программного кода иллюстрирует, как два описанных выше метода работают совместно. В процессе обслуживания базы данных SQL Everywhere рекомендуется выполнять подобные программы еженедельно или после определенного количества запусков приложения.

Пример на языке C#

```
SqlCeEngine engine = new SqlCeEngine("Data Source = AWMobile.sdf");
if (!engine.Verify())
{
   Cursor.Current = Cursors.WaitCursor;
   engine.Repair(null, RepairOption.RecoverCorruptedRows);
   Cursor.Current = Cursors.Default;
   MessageBox.Show("База данных SQL Everywhere Database
                    была восстановлена.");
}
```

Поддержание производительности репликации слияния

Репликация слияния является хорошим вариантом синхронизации баз данных SQL Everywhere и SQL Server 2000/2005. Несмотря на то что производительность репликации слияния может быть исследована в мониторе репликаций утилиты Management Studio, существует естественная необходимость периодически выполнять ее обслуживание, чтобы поддерживать оптимальную производительность. На практике большое количество корпоративных мобильных реализаций обнаруживали быстрое и существенное падение производительности репликаций слияния уже после нескольких дней обслуживания 20–30 подписчиков.

Чтобы всегда поддерживать производительность репликаций слияния на должном уровне, нужно периодически выполнять три основные задачи настройки и обслуживания.

1. Как можно чаще дефрагментировать все индексы таблиц `Distribution.Msmerse_*` (желательно каждую ночь).

2. Уменьшить период сохранения снимков (по умолчанию он составляет 14 дней), если модель использования приложения это позволяет.
3. Увеличить число генераций в каждом пакете на сервере.

Очевидно, что можно намного больше сделать для точной настройки производительности репликации слияния, но эти три задачи часто пропускают, что имеет непосредственное негативное влияние на скорость репликаций.

Более подробную информацию о настройке и обслуживании репликации слияния вы можете получить на сайте:

`www.microsoft.com/technet/prodtechnol/sql/2000/maintain/mergperf.mspx`

Дополнительная информация

Ниже приведено несколько исключительно полезных ссылок на ресурсы, посвященные SQL Server 2005 Everywhere Edition. Несмотря на то что масса информации содержится на сайте MSDN, большая ее часть посвящена SQL Server 2000 Windows CE Edition 2.0. За небольшим исключением вся эта информация применима и к SQL Everywhere.

- SQL Server 2005 Everywhere Edition Developer Center

  `http://msdn.microsoft.com/SQL/2005/mobile/default.aspx`
- SQL Server 2005 Everywhere Edition Books Online

  `http://msdn2.microsoft.com/en-us/library/ms173053(SQL.90).aspx`
- MSDN Newsgroup (news.microsoft.com)

  `Microsoft.public.sqlserver.ce`
- MSDN webcasts

  `http://msdn.microsoft.com/sql/webcasts/default.aspx`
- Множество вопросов, посвященных на сайте MSDN серверу SQL Server 2000 Windows CE Edition, применимы и к SQL Everywhere

  `http://msdn.microsoft.com/SQL/SQLCE/default.aspx`

Резюме

SQL Server 2005 Everywhere Edition является мощной СУБД для корпоративных мобильных приложений. SQL Everywhere предлагает богатый ассортимент средств для работы с реляционной базой данных непосредственно на мобильном устройстве в автономном режиме. Также он имеет встроенные возможности синхронизации данных с SQL Server в сценариях с подключением к серверу. Базы данных SQL Everywhere можно защищать, устанавливать и обслуживать в корпоративных проектах. Для программистов и администраторов баз данных, знакомых с SQL Server 2000 или 2005, работа с SQL Everywhere потребует лишь небольшого обучения, учитывая тесную интеграцию SQL Everywhere с программами Visual Studio 2005 и SQL Server 2005 Management Studio.

Программирование сборок CLR в SQL Server

ГЛАВА

27

В этой главе...

Беглое знакомство со средой .NET Framework

Обзор типов CLR в SQL Server

Поддержка языков .NET для интеграции с CLR

Инструкции T-SQL CLR DDL и представления каталогов

Создание типов баз данных в Visual Studio

Когда использовать CLR вместо T-SQL

С выходом версии SQL Server 2005 общеязыковая среда выполнения CLR среды .NET Framework стала доступной серверу баз данных. В CLR можно реализовать более сложную логику и вычисления, чем на уровне базы данных, с помощью хранимых процедур, функций, триггеров и пользовательских консолидаций. С помощью определяемых пользователем типов CLR позволяет также более точно моделировать в базе данных реальный мир. В то время как T-SQL остается оптимальным инструментом доступа к данным, CLR можно рассматривать как новое измерение гибкости в многоуровневой архитектуре приложения. До интеграции CLR приходилось либо мириться с медленными вычислительными способностями T-SQL, либо перемещать большие объемы данных по сети, чтобы получить маленький по объему результат, необходимый приложению. С интеграцией CLR все изменилось к лучшему, при условии того, что к задаче подключено все оборудование компьютера, а за клавиатурой сидит грамотный разработчик.

SQL Server не только использует компоненты среды .NET Framework, предлагаемые операционной системой. Среда CLR, используемая сервером, на самом деле запускается в пространстве оперативной памяти, используемой самой СУБД. SQL Server управляет синхронизацией и целостностью своего идентификатора профиля службы (SPID) и приоритетных потоков CLR. Сервер способен обслуживать как потоки CLR, так и собственный SPID в контексте одной и той же локальной транзакции.

Существуют некоторые ограничения того, что может сделать встроенная в SQL Server среда CLR по сравнению со средой CLR, поддерживаемой операционной системой Windows. Некоторые ограничения связаны с тем, что CLR только недавно была встроена в ядро базы данных, другие явились следствием конструктивных особенностей. В следующих

версиях сервера число ограничений будет уменьшаться, а объем возможностей — увеличиваться. В настоящее время SQL Server не поддерживает статические и общие компоненты .NET, которые требуют сохранение состояния между разными обращениями к данному компоненту.

В качестве простого примера рассмотрим классическое значение с автоматическим приращением, используемое в базе данных. Оно не может быть реализовано как функция CLR, и это вызвано тем, что статическое постоянство состояний между вызовами функции CLR не поддерживается. Функция просто не способна запомнить значение, использованное при ее предыдущем вызове. Это сходно с режимом работы обычной функции или процедуры T-SQL.

В интегрированной на сервере среде выполнения CLR доступно только подмножество базовых библиотек классов .NET. Например, пространства имен `System.Windows.Forms` и `System.Drawing` нельзя использовать в базах данных, так как понятия пользовательского элемента управления в базе данных не существует. В диалоговом окне отображаются только доступные базовые классы, которые можно использовать для добавления ссылок на проект интеграции CLR в Visual Studio.

К тому же существует множество ограничений в общей модели объектно-ориентированного программирования (ООП), когда объекты базы данных определяются с использованием интеграции CLR. Все эти ограничения, наряду с некоторыми разрешающими функциями программирования сборок CLR, предназначенных для использования в SQL Server, будут подробно рассмотрены в настоящей главе.

Компания Microsoft осознала, что признание интеграции технологий .NET Framework и SQL Server сообществом администраторов баз данных и разработчиков T-SQL станет значительной победой во всеобщем признании этого нового семейства средств базы данных. Несомненно, компания также понимает, что у многих из этих людей недостает подготовки и опыта, необходимого для адекватной оценки методик создания компонентов базы данных, в полной мере использующих силу и гибкость интеграции CLR. Эта глава организована таким образом, чтобы помочь администраторам и разработчикам понять суть интеграции CLR.

Начинается глава с введения в среду .NET Framework. Затем описывается, как использовать программу Visual Studio для создания компонентов базы данных, запускаемых в CLR. За этим последует обзор компонентов архитектуры .NET Framework, предназначенных для поддержки интеграции CLR. В главе будет приведен ряд примеров компонентов, демонстрирующих каждый из новых типов, которые стали доступны в SQL Server 2005 с введением интеграции CLR. В завершение будет предложен ряд рекомендаций относительно того, когда стоит использовать компоненты интеграции CLR.

Беглое знакомство со средой .NET Framework

Среда *Microsoft .NET Framework* является компонентом служб приложений операционной системы Windows. Назначением этой среды является обеспечение надежных и безопасных средств разработки и выполнения приложений, в которых множество языков программирования совместно используют богатый набор общих библиотек.

Приложения, связанные со средой .NET Framework, создаются согласно стандартным ориентированным на службы протоколам (SOA). Так, XML позволяет установить соединение с компонентами других приложений, как ориентированных на .NET, так и других, или даже запускаемых в блоках Windows.

Процессы, создаваемые с использованием среды .NET Framework, по умолчанию предназначены для защиты целостности операционной среды, независимо от того, завершится процесс ошибкой или попытается выполнить недопустимую операцию.

Среда .NET Framework должна была стать реализацией ранее невыполненных обещаний компании Microsoft в отношении масштабируемости и увеличения производительности Windows, хотя в версии 2.0 производительность остается ограниченной интересами возможности взаимодействия сетей и интеграции между приложениями Microsoft. В данной версии было взято за цель достижение производительности таких существующих интерфейсов программирования, как COM и OLE DB. В целом этот план был выполнен и даже перевыполнен. В будущих версиях можно небезосновательно ожидать дополнительных значительных прорывов в производительности.

Среда .NET Framework доступна в операционных системах Windows 2000, Windows 2003, Windows XP, Windows CE и Windows Vista; также она будет доступна во всех будущих версиях этой операционной системы. Гибкая система поддержки версий позволяет различным сборкам базовых библиотек классов .NET Framework быть доступными разным приложениям, одновременно запущенным на одном компьютере. Основной задачей здесь было преодоление жесткости в предыдущих моделях регистрации библиотек `.dll`, до боли знакомой многим разработчикам.

Сборки

Компоненты .NET, называемые *сборками*, являются базовыми единицами при компиляции. Сборки используют строгие соглашения об именах, предназначенные для уникальной идентификации версий. Они могут размещаться в самом приложении, а также в иерархии среды системы Windows, известной как *глобальный кэш сборок* (GAC). По умолчанию общие библиотеки классов среды .NET Framework расположены именно в этом кэше. Независимо от используемого языка программирования все приложения .NET имеют доступ к одним и тем же общим библиотекам в GAC.

Процесс компиляции приложений .NET разделен на два этапа. Сборки хранятся на промежуточном языке *MSIL* (Microsoft Intermediate Language). На каком бы языке .NET программист ни написал программу, она будет скомпилирована в MSIL и сохранена в сборке. В целом генерируемый код MSIL идентичен, независимо от того, на каком языке изначально была написана программа.

Во время выполнения сборки MSIL преобразуются в машинные инструкции компилятором *JIT* (Just In Time) интерпретатора CLR и загружаются в память. Интерпретатор CLR знает, когда именно MSIL нужно преобразовать в машинный код, стремясь перемещать в память команды только по мере необходимости и удаляя их из памяти, когда потребность в них отпала. Последнюю операцию часто называют *уборкой мусора*.

Все типы проверяются компилятором JIT во время выполнения программы. Остальные вопросы защиты, такие как сертификаты, подписи, политики и т.п., также решаются во время выполнения программы. На этом основана магия переносимости среды .NET Framework. Если операционная система понимает MSIL и файлам разрешено запускаться на локальном компьютере, то приложение будет выполнено.

Основными компонентами среды .NET Framework являются типы. Типом может быть значение или ссылка. Типы значений обладают следующими характеристиками:

- включают примитивы, такие как `integer`, `char` или `boolean`;
- обслуживаются в стеке выполнения программы;
- не могут содержать поля, методы или свойства;
- однозначно назначены переменным.

Типы ссылок имеют следующие характеристики:

- содержат объекты, такие как классы, интерфейсы, делегирования, массивы и перечисления;
- обслуживаются в куче управляемых объектов, отделенной от стека выполнения;

- могут содержать поля, методы и свойства;
- совместно используются всеми переменными, назначенными данному типу.

Используя интеграцию CLR, разработчик может расширить круг примитивов и использовать поля, методы и свойства. При этом тип значения в примитиве преобразуется в тип ссылки.

Только типы, соответствующие спецификации CTS (Common Type System) среды .NET, могут загружаться и обрабатываться интерпретатором CLR. Эти типы принято называть *управляемыми*. В соответствии с архитектурой среды .NET Framework все языки программирования, согласующиеся с CTS, должны генерировать идентичные управляемые типы. На практике существуют некоторые отличия между разными языками .NET. Среда .NET Framework реализует общеязыковую спецификацию CLS (Common Language Specification), чтобы устранить различия между языками программирования. За удовлетворение требований спецификации CLS несут совместную ответственность и программист, и компилятор. Только если код является CLS-совместимым, сгенерированный код MSIL будет доступен всем остальным приложениям, которые также генерируют управляемые типы. Следует отметить, что для запуска в интерпретаторе CLR необходимыми компонентами являются CTS- и (не обязательно) CLS-совместимость. CLS-совместимость всего лишь гарантирует то, что данная сборка может быть корректно использована компонентами, написанными на других языках программирования. К тому же CLS-совместимыми должны быть только типы и члены компонентов, предлагаемые для общего использования.



Спецификацию CTS, включая CLS, можно найти в документации .NET Framework SDK, входящей в состав документации Visual Studio 2005, а также на Web-сайте компании Microsoft.

Основной единицей компиляции, распределения, повторного использования, отслеживания версий и защиты является *сборка*. В терминогогии объектно-ориентированного программирования (ООП) сборку можно рассматривать как границы инкапсуляции, а среда .NET Framework является объектно-ориентированной. Сборка содержит исполняемый программный код в виде MSIL и все метаданные ресурсов для одного или нескольких типов. Большая часть метаданных содержит информацию о типах. содержащихся в сборке. Каждая сборка также содержит *манифест метаданных*. Этот манифест содержит дружественное имя и атрибуты, которые обобщают строгое имя сборки, назначенное ей при создании. Манифест также содержит все ссылки сборки. т.е. список всех сборок, зависимых от нее. На основании манифеста сборки компилятор JIT может по мере необходимости эффективно преобразовать код MSIL в машинные команды.

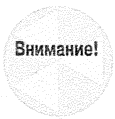
Домены приложений

Основной единицей пространства памяти запущенного приложения .NET является *домен приложения*. Теоретически приложение .NET может открывать сборки либо во время компиляции, либо во время работы приложения (примерно аналогично раннему и позднему связыванию ActiveX). На практике же без твердого понимания многоуровневых методов подстановок, используемых для переноса сборок в домен приложения, такой режим работы может показаться несколько странным.

В манифесте сборки определяется состав сборок, включаемых компилятором JIT в конкретный домен. Несмотря на то что один процесс может использовать несколько доменов приложений, последние являются областью определения типа. Тип не может ссылаться на код или ресурсы вне своего домена приложения, если, конечно, не соблюдаются определенные правила удаленного протокола, которые гарантируют целостность каждого процесса.

Удаленность — это кумулятивный механизм взаимодействия процессов. Она использует механизм прослушивания на сервере, чтобы позволить клиентам создавать ссылки на объекты, которые существуют где-то вне домена приложения клиента.

Следует заметить, что домен сборки приложения должен быть получен из метаданных манифеста, прежде чем сборка из домена другого приложения сможет к нему обратиться. Это справедливо независимо от того, запущены ли домены приложений в одном процессе, в разных процессах одной операционной системы или вообще на разных компьютерах, находящихся в отдаленных друг от друга регионах. В сущности, это требует, чтобы каждая из сборок, которая должна взаимодействовать с другими доменами приложений, компилировалась в код MSIL со ссылками на сборки, к которым она будет обращаться в других доменах. В некоторых случаях удаленный доступ между доменами сборок требуется из соображений производительности или безопасности, но наличие строгих правил является признаком того, что использование XML чаще всего является более гибким и универсальным методом взаимодействия доменов разных приложений, чем попытки совместного использования безопасных типов разными доменами.



Удаленность можно рассматривать как версию .NET технологии DCOM. Разработчики могут действительно захотеть воспользоваться этими возможностями, чтобы избежать необходимости включения явных ссылок в манифестах на пользовательские типы, определенные в сборках уровня приложения SQL Server. Только в очень редких случаях удаленность используется для создания ссылок на типы внутри процесса SQL Server, так как в этом случае решение становится слишком сложным и по определению раскрывает SQL Server. Такой подход вряд ли сможет оправдать связанные с ним риски и усилия программиста.

Приложение, содержащее только управляемые типы и использующее среду .NET Framework, принято называть *управляемым кодом*. Управляемый код отличается от неуправляемого тем, что среда выполнения CLR абстрагирована от аппаратного обеспечения и операционной системы. Программист может обойти этот защищенный режим выполнения программы, особенно учитывая накопившийся объем обычного кода, который было бы непрактично переписывать немедленно (или когда-нибудь) в управляемый код .NET. Например, на неуправляемые компоненты COM можно ссылаться, используя систему, предлагаемую сборкой периферийного интерфейса адаптера (PIA). Также можно выполнить функции любой отличной от COM библиотеки, используя механизм инициации платформы.



Программист на C++ (и, до определенного уровня, на C#) может выделять память и использовать указатели для ссылок на область памяти. К сожалению, эта возможность обхода требований безопасности типов приоткрывает лазейки для создания приложений, которые смогут повредить операционную среду. Хотя бы только по этой причине одним из наиболее важных действий в обеспечении защиты SQL Server является построение и развертывание на сервере баз данных лишь CLS-совместимых сборок.



Дополнительную информацию о сборках периферийных интерфейсов адаптеров вы найдете в главе 30.

Совершенно очевидно, что программный код .NET может существовать отдельно от приложения, которое его использует. Эти реалии открывают хорошую возможность осветить новый тип защиты, встроенный в CLR, — *безопасный доступ к коду (CAS)*. CAS гарантирует, что весь управляемый код для доступа к типам использует безопасные методы. Подобно CLS, проверка безопасности типов возлагается на компилятор. В то же время написание пригодного к проверке CAS-совместимого кода вменяется в обязанность программисту. CAS реализует синтаксис защиты; разработчик может декларативно устанавливать свойства разрешений или императивно создавать объекты разрешений.

В модели CAS, как и ожидалось, каждая сборка определяет необходимые разрешения в своем манифесте. Так как манифест сборки является точкой входа в сборку в процессе выполнения, перед выполнением любого программного кода сборки проверяется наличие соответствующих разрешений. Аналогично, библиотеки классов системы безопасности проверяют, имеет ли вызывающий процесс разрешение на их использование. Система CAS выходит за рамки традиционной модели аутентификации пользователя и его разрешений. В системе CAS код должен убедиться, что другие его фрагменты имеют право на взаимодействие.

Очень важно, чтобы программист при создании приложения учитывал требования CAS, в противном случае это может привести к сбою работы сервера. К счастью, интеграция CLR в SQL Server проверяет, является ли запускаемый в нем код CAS-совместимым. Среда выполнения CLR в SQL Server предоставляет разработчику частичные классы, которые за кулисами комбинируются с другим кодом для создания компонентов баз данных CLR. В части класса, которой пользователь не может манипулировать, содержатся некоторые важные CAS-совместимые инструкции защиты.

Когда JIT компилирует программный код в SQL Server, он проверяет соответствие требованиям CAS, безопасность типов, а также разрешения SQL Server. Когда программируется .NET-компонент базы данных, он может маркироваться как `SAFE`, `EXTERNAL ACCESS` или `UNSAFE`. Это определяет используемые ограничения модели программирования, а также объявления и императивы CAS, которые будут использованы этой сборкой.

Для построения CLS-совместимых сборок разработчик должен твердо придерживаться правил, изложенных в документе *Common Language Infrastructure, Partition I: Concepts and Architecture*, который можно найти в папке `Tool Development Guide`, устанавливаемой вместе с Microsoft .NET Framework SDK. Даже если разработчик строго следует спецификации, CLS-совместимому компилятору также необходимо гарантировать совместимость сборки с CLS. Все компиляторы пакета Visual Studio 2005 гарантируют такую совместимость. CLS-совместимость должна рассматриваться как обязательное требование для компонентов интеграции CLR, хотя бы потому, что нет иного способа задействовать языки, которым предстоит работать с конкретным объектом базы данных. В качестве одного из вариантов практически безболезненного обеспечения CLS-совместимости можно рассмотреть создание интегрированных в базу данных объектов CLR с помощью языка VB.NET в среде Visual Studio 2005.

Почему VB.NET является лучшим выбором для обеспечения CLS-совместимости? Ведь компиляторы C++ и C# в Visual Studio 2005 также способны создавать CLS-совместимый код, равно как и компилятор VB. Разработчикам, слабо знакомым с программированием в среде .NET Framework, VB.NET окажет наибольшую помощь в написании CLS-совместимого программного кода. VB.NET по праву считается самым простым в использовании языком семейства .NET, предназначенным для использования в среде *быстрой разработки приложений* (RAD). Образно говоря, VB.NET предлагает только CLS-совместимые инструкции. Естественно, выбор исключительно за вами; и если вы хорошо знакомы с каким-либо другим управляемым языком, естественно, лучше использовать его. Если же вы только начинаете программировать в среде .NET Framework или работаете преимущественно в среде быстрой разработки приложений, серьезно подумайте о разработке компонентов базы данных на языке VB.NET.

Что делает Visual Studio лучшим выбором для компиляции приложений? Ответ заключается в наличии множества дополнительных функций, которые предлагает эта программа. Такой же компилятор VB.NET входит в состав .NET Framework SDK, так что технически возможно создавать программный код даже в простом текстовом редакторе, таком как Notepad, а затем скомпилировать его в командной строке и получить полностью CLS-совместимые файлы MSIL. Некоторые даже могут аргументировать этот подход тем, что прежде чем переходить к такому элегантному инструменту, как Visual Studio, нужно понять, как все работает на самом нижнем уровне.

Может быть, в вышеописанном подходе и есть разумное зерно с точки зрения учебного процесса, но лично я считаю, что нет никакого смысла бороться с невооруженным специальными средствами текстовым редактором или компилятором командной строки. Максимальную пользу и качество программного кода вы получите, используя Visual Studio. Может, вы почувствуете больший контакт с языком программирования, используя напрямую SDK, но практически наверняка те, кто сразу связал свою судьбу с Visual Studio, развернут свой первый успешный .NET-компонент базы данных гораздо раньше тех, кто начал с SDK. Для большинства начинающих программистов знакомства с основными концепциями среды .NET более чем достаточно, и совершенно не нужно создавать себе лишние сложности, связываясь с SDK. Как вы вскоре увидите, средства Visual Studio 2005 сами докажут, что эта программа является наилучшим выбором. Перед тем как более внимательно взглянуть на инструменты, используемые для создания типов интеграции CLR, посмотрим, какими же могут быть эти типы.

Обзор типов CLR в SQL Server

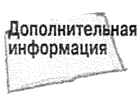
С помощью среды .NET Framework для SQL Server 2005 можно создать пять типов объектов:

- хранимые процедуры;
- функции;
- триггеры;
- пользовательские типы;
- пользовательские консолидации.

Первые три являются расширением функциональности, доступной в предыдущих версиях SQL Server с помощью языка T-SQL. Двойники этих объектов на языке T-SQL остались доступными и в версии SQL Server 2005 были даже улучшены. В конце главы будет показано, когда следует использовать типы CLR, а когда испытанные объекты T-SQL. Оставшиеся два типа объектов базы данных могут быть созданы исключительно с использованием интеграции CLR — это пользовательские типы и пользовательские консолидации. Эти типы появились только в версии SQL Server 2005. В этом разделе мы рассмотрим общие черты всех пяти типов объектов CLR SQL Server, а затем отдельно остановимся на характеристиках каждого из них.

Атрибуты типов .NET интеграции CLR

Ранее в этой главе была описана концепция домена приложения среды .NET Framework. В SQL Server домен приложения определяется владельцем объекта. Владельцем объекта является администратор базы данных. Это значит, что доменом приложения, или базовым уровнем изоляции программного кода для интеграции CLR, являются все объекты, принадлежащие одному и тому же пользователю или роли в текущей базе данных.



Дополнительная информация об администраторах базы данных приведена в главе 40.

Объекты CLR в SQL Server всегда содержат элементы метаданных, называемые *атрибутами*. Атрибуты в среде .NET используются для определения назначения конкретной сборки, класса, метода, свойства, события или типа. Например, атрибут может быть использован для определения метода как CLS-совместимого. Забегая вперед, в материал глав, посвященных Web-службам, можно сказать, что атрибут можно использовать для маркировки метода как Web-службы для объектов, находящихся за пределами домена приложения данного метода.



Web-службы во всех деталях будут рассмотрены в главе 32.

Именованные атрибуты типов объектов SQL Server более корректно называют пользовательскими атрибутами .NET Framework. Разработчики интеграции CLR в компании Microsoft для интерпретации этих атрибутов создали управляемые расширения класса из пространства имен `System.Attribute`. Пользовательские атрибуты вычисляются, когда тип импортируется в SQL Server и когда компилируется в среду выполнения. В объектах SQL Server пользовательские атрибуты определяют, какой из пяти типов объектов базы данных находится в источнике. Пользовательские атрибуты также содержат всю информацию, необходимую SQL Server для импорта объекта при развертывании. Все сопутствующие пользовательские атрибуты также вычисляются, когда для объекта выполняется инструкция T-SQL `CREATE OBJECT` после его импорта.



Помещайте блоки пользовательских атрибутов непосредственно перед программным кодом, к которому они относятся. Блок пользовательских атрибутов может быть также помещен в начало файла источника, если задан модификатор атрибута, указывающий, в каком месте он должен быть применен. Однако в этом случае связь между атрибутом и программным кодом, к которому он относится, будет не такой очевидной для того, кто будет сопровождать незнакомый код. Шаблоны объектов SQL Server в Visual Studio 2005 всегда используют первый метод. Каждый из атрибутов заключается в пару угловых скобок (<>) и содержит имя атрибута, за которым следуют обязательные позиционные параметры, а затем необязательные аргументы. Атрибуты объектов SQL Server хорошо документированы.

Ниже приводится несколько примеров пользовательских атрибутов объектов базы данных на языке VB.NET в том виде, в каком они сгенерированы шаблонами Visual Studio 2005.

```
<SqlProcedure()> _
<SqlTrigger(Name:="name",'Target:="Table", Event:="FOR UPDATE")> _
<SqlFunction()> _
<Serializable()> _
<SqlUserDefinedType(Format.Native)> _
    <SqlMethod> _
    <SqlFacet> _
<Serializable()> _
<SqlUserDefinedAggregate(Format.Native)> _
```

Как будет показано позже, установка списков аргументов может оказаться технически трудоемкой. Отметим, что атрибут `SqlProcedure` не имеет аргументов, и остальные атрибуты типов SQL Server, если явно не заданы, получают значения по умолчанию.

В шаблоне для пользовательских типов и консолидаций также присутствует атрибут `Serializable` (упорядочиваемость). Это не пользовательский, а системный атрибут; он используется во время выполнения, а не загрузки. Под упорядочиваемостью понимается то, что объект, которому присваивается этот атрибут, может записывать свое состояние в базу данных или некоторое другое постоянное хранилище, а затем считывать его оттуда. Упорядочиваемость является важным атрибутом объектов пользовательских типов и консолидаций, которые должны быть материализованы и индексированы в базе данных. Неупорядочиваемые типы могут быть не отсортированы, и их использование в некоторых случаях может оказаться невозможным. Например, неупорядочиваемые типы при использовании в качестве вычисляемых столбцов базы данных могут оказаться не проиндексированными, не отсортированными, не сгруппированными, не поддающимися сравнению и не живучими.

Пользовательские типы могут содержать два члена для атрибутов: `SqlMethod`, который унаследован из объекта `SqlFunction`, и `SqlFacet`, который используется для описания возвращаемого типа UDT.

Пользовательские атрибуты объектов SQL Server служат прототипом для конструктора классов этого типа. Естественно, это не единственный способ, которым атрибуты могут использоваться в среде .NET Framework. Вспомним, что экземпляры класса создаются с помощью метода new этого класса. Например, явный вызов метода создания нового экземпляра хранимой процедуры никогда нельзя увидеть в программном коде пользователя. Однако когда создается хранимая процедура CLR, на самом деле CLR вызывает конструктор класса каждый раз, когда эта процедура выполняется.

Общие характеристики типов CLR в SQL Server

На самом деле среда выполнения достойна рассмотрения как общая характеристика всех пяти типов объектов SQL Server. В кратком обзоре среды .NET Framework, с которого началась настоящая глава, была описана механика среды выполнения CLR: исходный программный код предварительно компилируется на язык Microsoft Intermediate Language, независимый от исходного языка программирования. Когда на данный код выполняется ссылка в среде выполнения, компилятор JIT загружает код MSIL в память в виде машинных инструкций. Самый обычный код .NET компилируется в инструкции, которые может выполнить операционная система Windows. Интеграция CLR подразумевает, что компилятор JIT читает код MSIL из таблиц базы данных, а не из файлов библиотек .dll. Этот компилятор создает инструкции, которые SQL Server хранит, распределяет и обрабатывает в своем пространстве памяти. Это позволяет использовать SQL Server не только для управления традиционным ядром, кэшем данных и процедур и постоянным источником проблем MemToLive, но и в качестве полноценной среды выполнения с интенсивной обработкой данных.

Изящество решения состоит в том, что мало что происходит при вызове компонента CLR, если сама среда выполнения не была активизирована на сервере. По умолчанию интеграция CLR отключена. Чтобы ее включить, нужно использовать утилиту SQL Server Surface Area Configuration Tool или хранимую процедуру T-SQL sp_configure:

```
exec sp_configure 'clr enabled', 1
reconfigure with override
```

После включения среды выполнения CLR при вызове компонента интеграции CLR при первом использовании он будет скомпилирован компилятором JIT в память. Полученные машинные инструкции будут доступны в памяти до тех пор, пока их срок жизни не истечет или не будет остановлена служба SQL Server. При создании первого экземпляра объекта производительность несколько падает, что связано с его компиляцией. После этого объект остается в качестве двоичного элемента в ядре, подобно любой встроенной функции или типу данных.

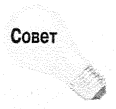
Было бы наивно ожидать, чтобы функции CLR могли сравниться по производительности с системными. В то же время вполне реалистично, чтобы функции CLR работы с датой-временем в запросах, обрабатывающих множество строк, выполняли гораздо больше, чем обычные функции T-SQL, используемые в предложении WHERE. Функции CLR не смогут по быстродействию сравниться со встроенными функциями, однако в контексте работы с множествами данных по производительности они все-таки намного ближе к встроенным, чем к функциям T-SQL.

Типы данных

Перед тем как исследовать аргументы атрибутов и прочие уникальные характеристики каждого из типов объектов SQL Server, некоторое внимание следует уделить одной общности между ними. Это типы данных. В манере, не менее неправдоподобной, чем обслуживание податливой средой выполнения SQL Server, основанной на приоритетах среды выполнения CLR, CLR также должна свести воедино "родные" типы данных SQL Server и .NET Frame-

work. Числовые типы данных .NET рассматриваются как типы значений. Типы значений обычно обслуживаются в области памяти стека вызова. Символьные типы данных .NET рассматриваются как объекты, на которые имеются ссылки. Они всегда используют память, отделенную от стека вызова, — она еще известна под названием *куча управляемого объекта*.

Типы данных, унаследованные от объекта (а все в куче управляемого объекта унаследовано от объекта общего базового класса), впечатляют. Такие типы содержат члены, в том числе свойства и методы, чтобы сделать манипуляции более гибкими, чем примитивные типы данных значений. Тип данных может быть перемещен или упакован из стека вызова в кучу объекта, если та определена, или использован, как если бы он имел свойства, методы или поля. Упаковка может быть явной и неявной.



Иногда полезно перемещать примитив в управляемую кучу, чтобы его поведение было похоже на поведение объекта. Этот процесс называется *упаковкой*. В качестве классического примера базы данных, в которой может использоваться упаковка, можно привести базу, в которой даты хранятся в виде символьных литералов или в которой интегральные значения должны сравниваться со столбцами, содержащими символьные данные. В то же время при сравнении десятков тысяч строк, основанных на столбце с типом ссылки, в коде CLR производительность будет значительно падать, если данные на самом деле имеют тип значения. При этом не всегда совершенно очевидно, что конкретная операция доступа к данным будет страдать от побочного эффекта упаковки. Обычно вопросы, касающиеся упаковки, всплывают после накопления некоторого опыта работы с интеграцией CLR. Тогда становится возможным идентифицировать типы CLR, которые по необъяснимым причинам работают медленнее других типов CLR. Чтобы определить, произошла ли упаковка, можно посмотреть на код MSIL. Читателям, заинтересовавшимся данным вопросом, будет небезынтересно узнать, что в состав Visual Studio 2005 входит утилита дизассемблирования кода MSIL, которая называется `ILDASM.EXE`.

Работа с типами данных CLR имеет свои сложности. Одной из самых больших проблем является то, что типы данных .NET Framework не распознают пустые значения. В SQL Server значение `null` указывает на то, что атрибут сущности не имеет определенного значения; таким образом, по умолчанию в стандарте ANSI SQL такое значение, среди прочего, не может участвовать в операциях сравнения, а также в консолидациях. Примитивные типы данных в модели программирования CLR лишены этой концепции. К счастью, существует коллекция типов данных, доступная в пространстве имен `System.Data.SqlTypes`, которая отображается непосредственно на типы данных SQL Server. Все эти типы реализуют интерфейс `INullable`. Это совершенно не значит, что сложности отображения типов данных между средой .NET и SQL Server на этом исчерпываются, но, тем не менее, это пространство имен может оказать существенную помощь. К примеру, если вам нужно сравнить десятичное значение базы данных с точностью, превышающей 28 знаков, с оперативным измерением, хранимым в десятичном значении .NET, необходимо будет преобразовать тип `decimal` среды .NET в тип `SqlDecimal`, чтобы избежать вероятных исключений при выполнении программы. В сходном, но полностью противоположном сценарии значение `SqlMoney` может быть переполнено при сравнении с десятичным значением .NET, которое пришло от поставщика, отличного от SQL Server.

С точки зрения программиста, при написании только "безопасного" CAS-совместимого кода CLR-компоненты базы данных, использующие типы из пространства имен `System.Data.SqlTypes`, позволят установить более прозрачные отношения между компонентами интеграции CLR и базой данных. К тому же использование `SqlTypes` для данных, которые происходят из базы данных, позволят программному коду выполняться существенно быстрее. поскольку типы данных .NET Framework будут неявно преобразовываться в `SqlTypes` средой выполнения CLR при отправке в базу данных. В противном случае для выполнения операций

сравнения и присвоения типы данных .NET должны преобразовываться явно. В табл. 27.1 приведены "родные" типы данных SQL Server в порядке, в котором они перечислены в пространстве имен `System.Data.SqlDBType`, соответствующем `System.Data.SqlTypes`, и эквивалентные им встроенные типы данных среды .NET Framework. Отдельные позиции будут сопровождаться комментариями автора.

Таблица 27.1. Типы данных SQL Server/CLR/.NET

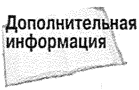
"Родной" тип SQL Server	Тип интеграции CLR в SQL Server	Тип CLR .NET Framework	Комментарии
`Bigint`	`SqlInt64`	`Int64`	
`Binary, Varbimary, Varbinary(max)`	`SqlBytes, SqlBinary`	`Byte[]`	
`Bit`	`SqlBoolean`	`Boolean`	
`Char, Nchar, Nvarchar, Nvarchar(max), sysname, Varchar, Varchar(max)`	`SqlChars, SqlString`	`String, Char[]`	В .NET Framework все символы принадлежат таблице Unicode
`DateTime, SmallDateTime`	`SqlDateTime`	`Datetime`	
`Decimal, Numeric`	`SqlDecimal`	`Decimal`	Разные диапазоны значений: в SQL Server и SqlDecimal +/-10^38+1, а в .NET несколько меньше: +/-7,9228162514264337593543950335E+28
`Float`	`SqlDouble`	`Double`	В SQL мантисса масштабируемая; по умолчанию `float(53)`
`Image`		`Byte[], Bitmap`	Все двоичные объекты поставляются в базу данных и из нее в потоках как `SQLBinary`
`Int`	`SqlInt32`	`Int32`	
`Money, Smallmoney`	`SqlMoney`	`Decimal`	Диапазон `SqlMoney` составляет 2^63 с точностью за одной десятитысячной
`Ntext, Text`			
`Real`	`SqlSingle`	`Single`	В SQL-92 эквивалентом типа `real` является `float(24)`
`Smallint`	`SqlInt16`	`Int16`	
`SQL_variant`		`Object`	
`Table`		`ISqlResultSet`	
`Timestamp`		`Byte[]`	
`Tinyint`	`SqlByte`	`Byte`	
Пользовательский тип			Сборка пользовательских типов должна присутствовать на стороне клиента, чтобы приложение знало о них

Окончание табл. 27.1

"Родной" тип SQL Server	Тип интеграции CLR в SQL Server	Тип CLR .NET Framework	Комментарии
`Uniqueidentifier`	`SqlGuid`	`Guid`	
`XML`	`SqlXML`		

Методы .NET, поддерживающие интеграцию CLR

Разработчику, создающему компоненты CLR базы данных с помощью пространства имен `SqlClient`, доступна богатая структура программирования среды .NET Framework. В этом пространстве имен существует набор классов, предназначенный для возвращения результатов и сообщений вызывающему приложению. Еще один набор классов используется для доступа к операциям базы данных из типов интеграции CLR.



Подробно о доступе к данным с помощью пространства имен `System.Data.SqlClient` вы узнаете в главе 30.

Некоторые операции (например, массовая загрузка, асинхронное выполнение и пакетные обновления) допустимы только в том случае, когда соединение выходит за пределы локального контекста процесса. Несмотря на то что большая часть серверных средств теперь интегрирована в `SqlClient`, существует масса классов для очень важных объектов, которые используются для серверной обработки, в пространстве имен `Microsoft.SqlServer.Server` сборки `System.Data`. Эти классы обеспечивают поддержку специальных видов деятельности, необходимых только в контексте локального процесса. Что очень важно — явное именование подключения как "the current context" позволяет доставлять результаты и сообщения клиенту и выполнять операции, используя контекст текущего процесса.



Интеграция CLR в SQL Server отделила серверный контекст процесса интеграции CLR от сетевого контекста выполнения `SqlClient` вне процесса. Одним из ожидаемых достоинств такой изоляции была возможность поддержки серверных курсоров на стороне сервера при одновременном устранении ограничений масштабируемости серверных курсоров на стороне клиента. Случайно эта технология оказалась таковой, что если серверный курсор должен поддерживаться программным кодом интеграции CLR, то он должен быть представлен в пространстве имен `SqlClient`. Следствием стало то, что серверный контекст выполнения процесса был ассимилирован в `SqlClient`. Разработчикам пришлось вносить существенные изменения в программу задним числом. Так что в Интернете и даже в группах новостей Microsoft циркулируют сотни примеров, которые были созданы до внесения этих изменений в архитектуру интеграции CLR. Если программный код найденного вами примера ссылается на пространство имен `System.Data.SqlServer`, то это явно указывает на то, что он был написан до выхода версии SQL Server 2005 RTM, т.е. до внесения вышеуказанных изменений. В цикле разработки среды .NET Framework 2.0 слишком поздно отразилось то, что компоненты интеграции CLR были вынесены в пространство имен `System.Data.SqlClient`. Серверная обработка теперь представляет собой один из режимов `SqlClient`, который задается и распознается в строке подключения:

```
Dim cnSQL As SQLConnection = _
   New SQLConnection("context connection=true")
```

Текущее определение контекста подключения можно найти в пространстве имен `Microsoft.SqlServer.Server` в Visual Studio 2005.

Разработчики часто используют преимущества работы с программным кодом примеров. Принимая во внимание то, что цикл бета-тестирования SQL Server 2005 был, пожалуй, самым массовым за всю историю, была создана громадная масса примеров, предшествующая официальному выходу в свет этой версии продукта. Весь этот программный код остается действенным и полезным в большинстве случаев, если разработчик скорректирует в случае необходимости ссылки на пространства имен. Только в этом случае код примера сможет функционировать.

Класс `Microsoft.SqlServer.Server.SqlContext` используется для установки специализированных серверных компонентов SQL Server. Открытие протокола взаимодействия с запрашиваемым приложением или клиентом выполняется с помощью свойства `SqlContext.Pipe()`. Это свойство используется для перемещения результирующих множеств и сообщений из хранимых процедур интеграции CLR и запрашиваемых приложений. `SqlContext.Pipe()` содержит метод `Send`, способный вернуть сообщение (об этом мы поговорим немного позже в этой главе):

```
pPipe.Send(System.Data.String)
```

ошибку:

```
pPipe.Send(System.Data.SqlClient.Error)
```

результирующий набор данных:

```
pPipe.Send(System.Data.Sql.ISqlReader)
```

или строку данных:

```
pPipe.Send(System.Data.Sql.ISqlRecord)
```

в зависимости от аргумента, передаваемого методу.

При возвращении результата метод `Pipe.Send` может доставлять данные из уже выполненного запроса и создавать объект `DataReader` на наборе данных `DataSet` или посылать уже созданный объект `DataReader` результирующего набора непосредственно из базы данных.



Коммуникационный канал предназначен в основном для процедур и функций, но в то же время может использоваться для триггеров. Как и в случае с триггерами T-SQL, отправка сообщений из подчиненных данным программ является потенциально рискованной операцией, поскольку не существует способа проверить, было ли получено сообщение.

Объект `SqlContext` позволяет получить в триггерах полезную информацию посредством свойства `SqlContext.TriggerContext()`. Это свойство идентифицирует перечисление `Microsoft.SqlServer.Server.TriggerAction`, которое вызвало триггер DDL или DML.

Одним из самых больших достижений внедрения недолго просуществовавшего пространства имен `System.Data.SqlServer` в пространство имен `System.Data.SqlClient` стало сокращение числа различных технологий, необходимых для эффективной реализации интеграции CLR. Теперь в пространстве имен `Microsoft.SqlServer.Server` стало меньше членов и уменьшилось количество ограниченных контекстом подключения функций. В то же время средства создания доступа к данным остались такими же, независимо от того, развертывается ли код на сервере или на другом уровне приложения.

Инструкции CLR DDL языка T-SQL и представления каталогов

Загрузка типов CLT в SQL Server требует создания двух объектов базы данных. В любом случае первой должна быть создана сборка, содержащая типы, и уже затем в ней должны быть созданы сами типы. Создание сборки — это, по существу, акт проверки и загрузки кода MSIL в представление каталога `sysassemblies`. При выполнении инструкции `CREATE ASSEMBLY` не может быть создано полезных артефактов. Иерархия сборки состоит из трех представлений.

- `sysassemblies`. Это представление содержит одну строку метаданных для каждой сборки, включающую имя сборки и режим безопасности CAS.
- `sysassembly_files`. Это представление содержит двоичные данные для каждого файла сборки.
- `sysassembly_references`. Это представление содержит по одной строке для каждой сборки, на которую в манифесте текущей сборки содержится ссылка.

Сборки

Инструкция DDL `CREATE ASSEMBLY` содержит всю информацию, необходимую для заполнения представления каталога. Одновременно на сервере не может существовать множество копий одной и той же версии сборки. Для замены простой именованной сборки или ее версии строго именованной необходимо в первую очередь удалить существующую сборку с помощью инструкции `DROP ASSEMBLY`. Также возможно модифицировать существующую сборку с помощью инструкции `ALTER ASSEMBLY`. При этом текущая учетная запись NT должна иметь права чтения в месте расположения файлов UNC, откуда должна быть загружена библиотека `.dll` данной сборки. Даже если проект содержит множество файлов источников, инструкция `CREATE ASSEMBLY` требует, чтобы только сборки, состоящие из одного файла MSIL, загружались в SQL Server. Делегирование не поддерживается инструкцией `CREATE ASSEMBLY`. Все сборки, на которые существуют ссылки, и их манифесты должны быть доступны в том же месте сети, что и данная сборка, в противном случае выполнение инструкции `CREATE` завершится ошибкой.



Всегда создавайте сборки для производственной среды SQL Server с "жесткими" именами.

Инструкции `CREATE ASSEMBLY`, `DROP ASSEMBLY` и `ALTER ASSEMBLY` по умолчанию разрешены для использования членам серверной роли `sysadmin` и членам роли базы данных `db_owner`. Если уровень защиты CAS для сборки установлен в `EXTERNAL_ACCESS`, то учетная запись пользователя, создающего сборку, также должна иметь разрешение `EXTERNAL_ACCESS`:

```
GRANT EXTERNAL_ACCESS to <пользователь_или_группа_Windows>
```

Если режим защиты сборки установлен в `unsafe`, сборку могут создать только члены серверной роли `sysadmin`.

Объекты базы данных

После того как сборка создана, в ней может быть создан любой тип с помощью инструкции CREATE для соответствующего типа. Инструкции DDL и заполняемые представления каталогов могут включают следующее.

- CREATE PROCEDURE. Представление sysobjects с параметром type, равным "PC".
- CREATE FUNCTION. Представление sysobjects с параметром type, равным "FS", для скалярных функций или "FT" — для табличных функций.
- CREATE TRIGGER. Представление systriggers с параметром parent_class, равным нулю, для триггеров DDL и равным единице для триггеров DML. Представления sysobjects также заполняются с параметром type, равным "TR", для триггеров DML, но не заполняются для триггеров DDL.
- CREATE TYPE. Представление systypes с параметром is_user_type, равным единице, и с параметром is_assembly_type, также равным единице.
- CREATE AGGREGATE. Представление sysobjects с параметром type, равным "AF".

Пользователь, создающий типы CLR в базе данных, должен быть либо владельцем сборки, либо иметь к ней разрешение REFERENCES:

```
GRANT REFERENCES ON <имя_сборки> TO <пользователь_или_группа_Windows>
```

Для просмотра сборок, скомпилированных на сервере, доступно динамическое представление управления sysdm_loaded_assemblies. Типичным запросом, предназначенным для определения состава сборок, используемых базой данных в настоящее время, является следующий:

```
SELECT a.clr_name
   , USER_NAME(a.principal_id) as [principal]
   , la.load_time
   , a.permission_set_desc
FROM sysdm_clr_loaded_assemblies la
JOIN sysassemblies a
ON la.assembly_id = a.assembly_id
```

Программа Visual Studio 2005 для развертывания объектов интеграции CLR в базе данных предлагает графический интерфейс. Во время развертывания "за кулисами" выполняются инструкции CREATE ACCEMBLY и CREATE языка DDL. Администраторы и разработчики должны понимать, как используются обе эти инструкции для загрузки компонентов интеграции CLR в базу данных. Вряд ли развертывание продукта в производственных условиях с помощью Visual Studio будет приемлемым вариантом.

Создание типов данных в Visual Studio 2005

Хранимые процедуры, функции и триггеры CLR и те же объекты T-SQL имеют больше сходств, чем отличий. В общем случае эти типы CLR будут состоять из одного метода. В противоположность этому пользовательские типы (далее UDT) и консолидации (далее UDA) вообще не похожи на хранимые процедуры, функции и триггеры CLR.

Объекты UDT и UDA могут быть структурами или классами. Следует заметить, что использование структур в общем случае предпочтительнее, поскольку по определению структуры хранятся в стеке, а типы ссылок — в куче объекта. UDT и UDA, оформленные как класс, ве-

роятнее всего, будут иметь тип ссылки, что сильнее повлияет на производительность, чем на использование структур. Как бы там ни было, UDT и UDA будут содержать множество членов и должны удовлетворять большому числу критериев проектирования. В оставшейся части настоящей главы мы сосредоточим внимание на типах хранимых процедур, функций и триггеров, основанных на методах.



Более подробная информация о пользовательских типах приведена в главе 29.

Создание проекта CLR

Первые шаги в создании компонента интеграции CLR в Visual Studio 2005 практически идентичны, независимо от того, какой тип компонента создается. Например, даже несмотря на то, что по умолчанию проект интеграции CLR не требует наличия подключения к базе данных, поскольку в будущем он будет существовать в самой базе, Visual Studio потребует определить его, поскольку он будет использоваться на этапах разработки и отладки компонента. Определите в проекте то местоположение базы данных, в которой компонент будет тестироваться, уже на этапе создания проекта.

После того как источник данных будет определен, в любой момент можно щелкнуть на нем правой кнопкой мыши в окне Server Explorer, чтобы открыть контекстное меню, содержащее команды отладки приложения, объектов T-SQL и типов интеграции CLR. Пользователь также должен иметь разрешения к хранимой процедуре `sp_enable_sql_debug` (находящейся в базе данных `master`) и разрешения изменения (`ALTER`) для всех объектов, отладка которых будет производиться в базе данных. Следует отметить, что только администраторы базы данных имеют разрешение на отладку типов CLR, содержащих предложение `EXECUTE AS` или использующих сертификаты; к тому же на сервере одновременно может отлаживаться только один тип. Во время сессии отладки замораживаются все остальные потоки CLR. Это, естественно, нельзя назвать идеальным вариантом для совместно используемого сервера, если именно на нем выполняется тестирование интеграции. Отладка компонентов CLR больше подходит для обособленной среды тестирования, в которой сервер базы данных является экземпляром, выделенным для монопольного использования одним разработчиком.



Развертывание и тестирование интеграции CLR требует установки SQL Server 2005 и Visual Studio 2005. Отладка и совместная работа во время отладки не особо хорошо поддерживаются, если используется общая среда SQL Server. Наилучшей конфигурацией для разработчика является наличие на локальной рабочей станции как Visual Studio 2005, так и полноценной версии SQL Server Developer Edition. Программисты, работающие над компонентами для распределенных или федеративных приложений баз данных, могут рассмотреть как вариант локальную рабочую станцию с ресурсами, адекватными для поддержки экземпляров виртуальных серверов, моделирующих реальную производственную среду. В такой виртуальной среде желательно использовать именованные экземпляры серверов, чтобы уже на начальной стадии проектирования учесть все соглашения системы безопасности.

После того как источник данных будет настроен, откроется диалоговое окно, показанное на рис. 27.1. Выберите тип проекта базы данных для языка, который будет использован для написания программного кода. На данном этапе особое внимание уделите именованию проекта. В некоторых ситуациях в одном проекте приходится держать множество типов интеграции, в других же каждый тип должен быть выделен в отдельный проект. При принятии решения следует учитывать количество разработчиков, сложность типов и объем программного кода, совместно используемого разными типами.

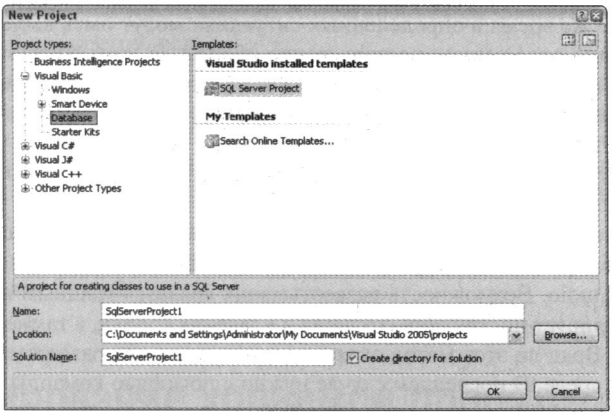


Рис. 27.1. Создание проекта интеграции CLR в Visual Studio 2005. После того как выберите тип базы данных для используемого языка программирования на левой панели, на правой выберите используемый шаблон проекта

После создания проекта шаблон для данного типа может быть добавлен в проект либо с помощью щелчка правой кнопки мыши на его имени в контекстном меню Solution Explorer и выбора из него пункта Add, либо с помощью соответствующей команды меню Project программы Visual Studio. В контекстном меню окна Solution Explorer (рис. 27.2) можно выбрать пункт Properties, чтобы открыть диалоговое окно Project Desigher. В этом окне можно определить строгое имя, задать уровень разрешений CAS для сборки, а также настроить параметры компиляции и развертывания проекта.

Если сборка будет использоваться несколькими приложениями, компания Microsoft рекомендует присваивать ей строгое имя. Строгое имя создает глобальный уникальный идентификатор для сборки и манифеста, состоящий из текстового имени, номера версии, необязательной региональной информации, общей части ключевой пары `public/private` и цифровой подписи. Так как вряд ли существуют практические основания запрета использования базы данных разными приложениями, все компоненты интеграции CLR лучше строго именовать.

Ключевая пара `public/private` создается с помощью утилиты командной строки `sn.exe`. Она помещается в текстовый файл с расширением `.snk`, как если бы это был контейнер *поставщика службы криптографии* (CSP). Файл ключевой пары или контейнер CSP может быть сгенерирован, а подпись сборки — завершена во время компиляции на основе информации, введенной во вкладках Project Designers Signing и Publish. В корпоративных условиях следует уделить особое внимание стратегии подписывания; практически во всех случаях будет предпочтительнее иметь единый ключ, совместно используемый всеми

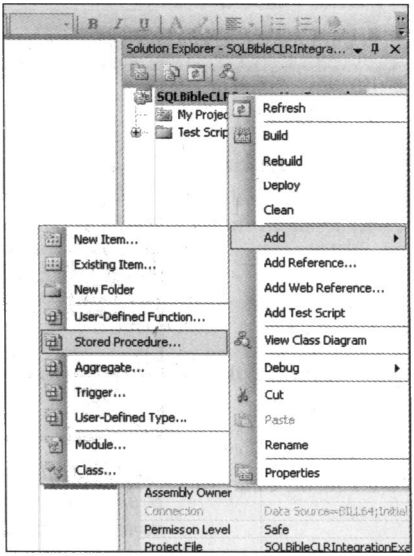


Рис. 27.2. Добавление хранимой процедуры CLR в проект SQL Server. Далее нужно присвоить имя сгенерированному шаблону, которое включает атрибуты для хранимой процедуры и "скелет" ее метода

приложениями. В то же время в определенных ситуациях могут быть выделены персональные ключи и использовано отложенное подписывание сборок. В этом сценарии сборка подписывается одним или несколькими доверенными лицами, чтобы гарантировать, что только нужные сборки будут загружены в производственный сервер.

В контекстном меню Solution Explorer доступны команды Build (Построить), Clean (Очистить), Rebuild (Перестроить) и Deploy (Развернуть). Те же команды доступны и в меню Build программы Visual Studio.

Запустите Profiler на источнике данных SQL Server, который будет использован во время развертывания, чтобы увидеть инструкции `CREATE ASSEMBLY` и `CREATE <объект>`, сгенерированные Visual Studio. Возможность развертывания является мощным инструментом разработки, который генерирует инструкции создания и удаления сборки, а также отдельных типов на сервере разработки. Вряд ли этот метод окажется пригодным для развертывания объектов CLR в производственной среде. В последнем случае нужно с помощью команды Create создать библиотеки `.dll` с кодом MSIL и поместить эти файлы в сетевой каталог. После этого данные сборки могут быть загружены с помощью инструкций `CREATE ASSEMBLY` сценариев.

Возвращаясь к рис. 27.2, отметим, что в проект можно добавить два типа ссылок: ссылку на существующий компонент, который будет использован в данном проекте, и ссылку на Web-службу.

Добавление ссылки на компонент типа интеграции CLR предложит вам только часть возможностей, доступных другим приложениям .NET. На рис. 27.3 показан этот сокращенный список доступных общих компонентов .NET, пригодных к использованию в CLR SQL Server. Ссылки на компоненты, созданные в локальных проектах, также могут быть добавлены во вкладке Projects этого диалогового окна. Чтобы локальные проекты стали доступными в диалоговом окне References, путь к ним должен быть предварительно прописан в окне Project Designer.

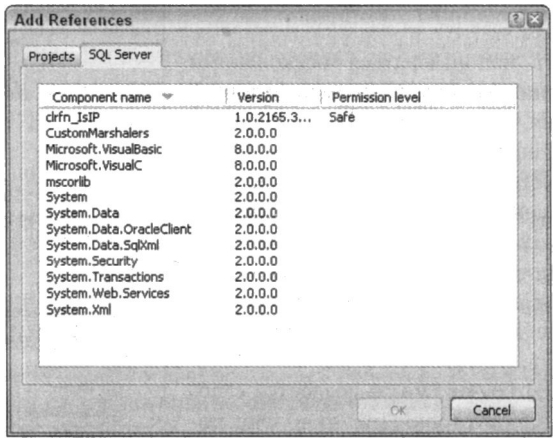


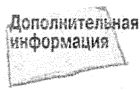
Рис. 27.3. Добавление ссылки в проект SQL Server. В этом окне отображаются все остальные компоненты сборки в пространстве проекта и доступные сборки системных компонентов

Ссылки на Web-службу XML создают в проекте прокси-сервер для файлов `.wsdl` и `.asmx`. Этот сервер позволит выполнять инструкции на стадии разработки проекта, а также стать компоненту интеграции CLR клиентом для указанной Web-службы. Файл определения Web-службы может существовать на локальном сервере, в локальной сети или в Интернете. Как только Web-служба будет добавлена в проект, члены, сгенерированные в прокси-сервере,

можно просмотреть в окне Object Explorer. Потенциал использования Web-служб хранимыми процедурами CLR несказанно велик.



Будьте осторожны при использовании Web-служб на стадии разработки. Сгенерированный прокси-сервер может выполнять программный код на локальном компьютере в контексте безопасности пользователя, зарегистрировавшегося в момент открытия IDE программы Visual Studio. В зависимости от того, что делает Web-служба, локальная рабочая станция может быть во время ее работы чрезмерно загружена.



Дополнительные детали использования Web-служб в SQL Server описываются в главах 31 и 32.

Программирование хранимой процедуры CLR

Хранимые процедуры являются наиболее гибкой формой типов интеграции CLR. Не нужно и говорить, что они — также самый простой из всех типов. Хранимые процедуры являются наиболее подходящими объектами для использования уровней защиты CAS EXTERNAL_ACCESS и UNSAFE. Несмотря на то что в некоторых сценариях предпочтительнее задействовать функции или, что менее вероятно, триггеры, хранимые процедуры CLR реализуют более естественный канал интеграции в базу данных гетерогенных хранилищ. Риск использования функций и триггеров заключается в том, что при привлечении их в потенциально "долгоиграющие" сценарии в среде выполнения с привилегиями это станет ограничивающим фактором в их использовании в качестве интерфейса между ядром базы данных и внешним миром. В противоположность этому хранимые процедуры .NET могут легко и безопасно взаимодействовать с файловой системой.



Весь программный код, описываемый в настоящей главе и касающийся проектов Visual Studio 2005, можно загрузить с Web-сайта книги по адресу www.SQLServerBible.com.

Рассмотрим хранимую процедуру CLR, которая служит утилитой экспорта любого набора данных или их коллекций в файл с разделителями CSV. В этом примере будет продемонстрировано множество только что описанных концепций. Сама процедура выполняет следующее.

- Читает данные, требующие доступа в контексте подключения.
- Выполняет запись в файловую систему, что влечет за собой использование уровня защиты CAS EXTERNAL ACCESS.
- Выводит запрашивающему информационные сообщения, реализуя возможность использования класса канала.

Процедура принимает два аргумента.

- Допустимый запрос T-SQL, функция или хранимая процедура, которая возвращает один или более результирующих наборов данных.
- Допустимое имя файла, включая путь к нему.

В приведенном ниже примере продемонстрированы следующие части этой хранимой процедуры. Вначале создается шаблон хранимой процедуры. После добавления шаблона хранимой процедуры в новый проект базы данных и добавления пространства имен System.IO в сборку методу хранимой процедуры может быть присвоено имя, также могут быть определены аргументы. Затем для доступа к локальным данным будет необходимо использовать

контекст внутреннего подключения процесса. Так как внешняя процедура должна иметь возможность получить доступ к данным с помощью вызова хранимой процедуры и/или инструкции SQL, следует реализовать объект, способный выполнить обе эти операции. Далее может оказаться полезным информировать о выполняемом действии вызывающую процедуру. Это легко реализовать с помощью объекта `SQLPipe` и форматирования сообщения. Например, значения входных аргументов могут быть переданы как информационное сообщение.

Обратите внимание на то, что аргументы, объявленные с "родным" типом `SqlString`, должны быть преобразованы перед использованием в операциях .NET, в то время как строка .NET (тип `string`) может использоваться и без преобразования. В последнем случае преобразование выполняется неявно при вызове процедуры. Таким образом, затрат на выполнение преобразования не избежать, хотя трудоемкость программирования можно несколько снизить.

Перед использованием объекта `SqlConnection` он должен быть открыт. К тому же перед записью результатов в файл должен быть открыт поток к этому файлу. Здесь мы впервые используем пространство имен `System.IO`, ранее добавленное в сборку. Более того, так как процедура использует ресурсы вне контекста SQL Server, в котором запущена, уровень защиты должен быть установлен в `EXTERNAL ACCESS`, при этом должны быть заданы необходимые права и разрешения для доступа к файловой системе.

Как только соединение будет открыто и файл готов к получению данных, инструкция может выполняться. Существует масса методов выполнения инструкций. Можно наполнять набор данных `DataSet`, находящийся в памяти, чтобы направить потоком результаты в формате XML, а можно просто получить скалярное значение. Каждая из этих операций требует наличия специального метода. В примере экспорта в файл CSV результат должен быть направлен в поток как односторонний результирующий набор данных или объект `DataReader` с использованием метода `ExecuteReader`. Метод `ExecuteReader` принимает аргумент `CommandBehavior`, который подстраивает результат к предъявляемым требованиям. В данном примере определен режим работы команды по умолчанию, так что этот аргумент может быть опущен. Например, если требования определяют, что только один результирующий набор данных может быть направлен в файл, а не несколько, следует указать аргумент `CommandBehavior.SingleResult`.

Когда выполняется `DataReader`, схема результирующего набора данных может быть легко идентифицирована на лету с помощью создания в памяти объекта `DataTable` и наполнения его с помощью вызова метода `GetSchemaTable` существующего объекта `DataReader`. Этот метод поможет включить в файл CSV имена столбцов.

В заключение результат направляется в файл, после чего буфер потока, а также объект `DataTable` и само подключение должны быть закрыты. Только после этого выполнение метода можно завершить.

```
Imports System.IO
<Microsoft.SqlServer.Server.SqlProcedure()> _
Public Shared Sub clrusp_ResultToCSVFile(ByVal sTSQL As SqlString, _
ByVal sFile As String)

Dim cnCLR As SqlConnection = New _
                          SqlConnection("context connection=true")
Dim qryGetResult As SqlCommand = cnCLR.CreateCommand
qryGetResult.CommandText = CStr(sTSQL)
qryGetResult.CommandType = CommandType.Text
Dim pInfo As SqlPipe = SqlContext.Pipe
pInfo.Send("T-SQL batch to execute: " & CStr(sTSQL))
pInfo.Send("Destination file: " & sFile)

cnCLR.Open()

Dim strmResult As StreamWriter = New StreamWriter(sFile)
```

```
Dim sqldrResult As SqlDataReader = _
qryGetResult.ExecuteReader(CommandBehavior.Default)

Dim sqldrResultSchema As DataTable
sqldrResultSchema = sqldrResult.GetSchemaTable
For Each drRow As DataRow In sqldrResultSchema.Rows
  strmResult.Write(drRow("ColumnName").ToString)
  If CInt(drRow("ColumnOrdinal")) < FieldCount - 1 Then
    strmResult.Write(",", 0, 1)
  Else
    strmResult.Write(vbCrLf, 0, 2)
  End If
Next

If sqldrResult.HasRows Then
  While sqldrResult.Read()
    For i As Int32 = 0 To sqldrResult.FieldCount - 1
     strmResult.Write(sqldrResult.GetSqlValue(i).ToString, 0, _
                      Len(sqldrResult.GetSqlValue(i).ToString))
     If i < sqldrResult.FieldCount - 1 Then
       strmResult.Write(",", 0, 1)
     Else
       strmResult.Write(vbCrLf, 0, 2)
     End If
    Next
  End While
End If

strmResult.Flush()
strmResult.Close()
sqldrResult.Close()
cnCLR.Close()
```

Несомненно, нужно было бы добавить еще кое-что, например обработку ошибок и множества результирующих наборов данных, если таковые существуют. При желании читатель может ознакомиться с полным вариантом проекта на Web-сайте книги.

Функции CLR

Функция CLR в большинстве случаев не требует такого объема программного кода, как хранимые процедуры. По той же причине, по которой рекомендуется быть в функциях T-SQL предельно кратким, желательно несколько ограничить области применения функций CLR. Это позволит сохранить транзакции максимально краткими и одновременно избежать издержек производительности, которые могут быть скрыты в программном коде функций. Хранимые процедуры CLR способны элегантно модифицировать структуру данных, в то время как функции никогда не смогут изменить состояние базы данных.



Компания Microsoft решила не препятствовать созданию клиентских подключений ADO.NET 2.0 в теле функций CLR. Это значит, что существует возможность изменения состояния базы данных и нельзя предотвратить такие действия, выполненные из функции. Такая возможность может свести на нет все хорошее, что есть в функциях CLR. Остерегайтесь этих средств!

Теперь посмотрим, как можно использовать скалярную функцию для проверки IP-адреса. Такую операцию легко выполнить с помощью функции CLR, использующей обычные выражения .NET. Эта функция является достаточно мощной, поскольку может быть использована в инструкциях DML, хранимых процедурах T-SQL, ограничениях проверки и вычисляемых столбцах.

Для создания функции добавьте импорт пространства имен `System.Text.RegularExpression` в шаблон Visual Studio 2005, установите атрибуты функции, определяя, что она детерминированная и точная, введите несколько строк программного кода, после чего разверните функцию. Эта функция может оказаться полезной в ограничениях проверки и индексируемых вычисляемых столбцах, так как она выполняет необходимые манипуляции строками гораздо быстрее, чем эквивалент на языке T-SQL.

```
Imports System.Text.RegularExpression

<Microsoft.SqlServer.Server.SqlFunction _
(DataAccess:=DataAccessKind.None, IsDeterministic:=True, _
 IsPrecise:=True)> _
Public Shared Function clrfn_IsIP(ByVal Value As String) As Boolean
Dim rx As New Regex( _
"((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)", _
     RegexOptions.IgnoreCase _
     Or RegexOptions.IgnorePatternWhitespace)
Return rx.Match(CType(Value, String)).Success
End Function
```

Если функция в качестве результата возвращает набор данных, а не скалярное значение, результатом станет *потоковая табличная функция*. Ключевым словом здесь является "потоковая". Табличные функции T-SQL должны вернуть всю таблицу до того, как она сможет быть использована. Это задействует большой объем памяти, равно как и пространства в базе `tempdb`, если результирующий набор данных велик, или если запрос для получения результата обрабатывает большой объем данных. Та же функция, описанная как потоковая функция CLR, может быть использована, как только создана, таким образом, снижая требования к памяти и базе `tempdb`. К счастью, табличные функции позволяют набору данных быть инициализированным, используя контекст подключения, а не создавая новое клиентское подключение ADO. Использование клиентского подключения ADO.NET сводит на нет роль функции — обеспечение эффективного выполнения не основанных на множествах операций. Следовательно, таких подключений следует избегать.

Для завершения обзора типов CLR, относящихся к базам данных, рассмотрим триггер. В SQL Server 2005 триггеры могут предложить много нового. Триггеры CLR реализуют всю функциональность, доступную в триггерах T-SQL. Триггер может быть ассоциирован с действием DDL или DML. В триггерах DML доступны таблицы `Inserted` и `Deleted`. Триггерами DDL могут быть события или группы событий.

Триггер, предложенный в качестве примера в Books Online, демонстрирует использование стандартных выражений для проверки правильности адреса электронной почты и последующего занесения записи в журнал аудита. Этот пример может показаться интересным, однако логичнее было бы реализовать данную задачу с помощью ограничения проверки и триггера аудита T-SQL. На самом деле в большинстве задач, для которых предлагается решение с помощью триггера CLR, триггер T-SQL, делающий запись в очереди брокера служб, был бы удобнее как для разработчика, так и для администратора. Время покажет, но триггер CLR претендует на роль наименее полезного из типов CLR. Давайте рассмотрим следующий триггер CLR DDL, который реализует свойства контекста триггера для операций DDL уровня базы данных. Это позволит продемонстрировать способ формирования триггера CLR.



Триггеры интеграции CLR могут показаться интересными тем, кто заинтересован в совместимости с законом Сарбанеса–Оксли, поскольку действия аудита компилируются и потому не так чувствительны к подстановке и непреднамеренной модификации, как триггеры T-SQL. Контекст XML события триггера также содержит стандартизированный и полный набор информации аудита.

```
<Microsoft.SqlServer.Server.SqlTrigger _
  (Name:="clrtr_ddlchanges", _
  Target:="DATABASE", _
  Event:="DDL_DATABASE_LEVEL_EVENTS")> _
Public Shared Sub clrtr_ddlchanges()
  Dim pInfo As SqlPipe = SqlContext.Pipe
  pInfo.Send(SqlContext.TriggerContext.TriggerAction.ToString)
  pInfo.Send(SqlContext.TriggerContext.EventData.Value.ToString)
  pInfo.Send(SqlContext.WindowsIdentity.Name.ToString)
End Sub
```

В данном примере были продемонстрированы информативные возможности триггера CLR. Однако всю информацию, доступную в данном триггере, можно с легкостью найти и в триггере T-SQL, при этом вероятность того, что данному триггеру понадобится записать данные в другую таблицу, достаточно высока. Выглядит так, будто затраты на перемещение информации между таблицей базы данных и CLR с учетом компиляции триггера будет сложнее оправдать, чем использование хранимой процедуры или функции CLR. При этом, естественно, предполагается, что использованы правильные основания для определения того, когда хранимую процедуру, функцию или триггер лучше писать на CLR и когда на T-SQL.



Интеграцию CLR можно также использовать для создания пользовательских функций консолидации (типа `min()`, `max()` или `sum()`). Подробное описание процесса разработки пользовательских функций консолидации данных выходит за рамки настоящей книги, однако вы можете найти примеры таких функций на сайте книги (`www.SQLServerBible.com`).

Что использовать: CLR или T-SQL

Несомненно, редко можно встретить реализацию, которая совершенно не использует CLR. Однако некоторые реализации используют в базе данных исключительно компоненты CLR. Ни один из этих подходов нельзя назвать приемлемым, если на первое место выходят такие фундаментальные понятия, как масштабируемость и производительность.

В большинстве реализаций первым шагом в направлении интеграции CLR было обеспечение высокой скорости посредством замещения существующих проблематичных пользовательских функций и хранимых процедур их более производительными эквивалентами — компонентами CLR. Никто не говорит, что плохо написанный компонент базы данных может быть преобразован в плохо написанный объект CLR, и все сразу станет хорошо. Наличие опыта оптимизации и мониторинг производительности остаются ключевыми предпосылками создания хорошей модели базы данных. В то же время ряд существующих очевидных проблем дает нам в руки материал для преобразования в объекты CLR.

- Существующие расширенные хранимые процедуры COM подвергают риску устойчивость всего сервера базы данных. Пользовательские расширенные хранимые процедуры следует рассматривать как первых кандидатов на преобразование в компоненты интеграции CLR.
- Хранимые процедуры автоматизации OLE также завоевали и постоянно подтверждают дурную репутацию расширенных хранимых процедур за плохую обработку ошибок, низкую производительность, утечки памяти и нечеткость отношений между уровнем T-SQL и средой COM. Безопасность управляемого программного кода бесспорно является привлекательной альтернативой хранимым процедурам `sp_OA`.
- Сложные правила бизнес-логики и логики, содержащей интенсивные вычисления, в существующих хранимых процедурах и функциях могут получить преимущества компи-

лируемого кода и потоковых табличных функций. Обработка массивов на сервере баз данных теперь стала возможной, равно как и компилированные групповые консолидации правил бизнес-логики.

- Сложные правила бизнес-логики и логики, содержащей интенсивные вычисления, в настоящее время реализуются путем перемещения больших массивов данных по сети для обработки на среднем уровне. Если эту обработку вынести на сервер, то можно высвободить ресурсы сети и применить более совершенную модель ввода-вывода.
- Существующие приложения, которые проигрывают от применения даже относительно простых пользовательских функций при их использовании в предложении `WHERE` запросов, обрабатывающих большие массивы строк, могут задействовать эти функции на основе CLR, что обеспечит их производительность на уровне системных функций.
- Параметризованная обработка списков и прочие операции манипулирования строками и их проверки, которые в настоящее время требуют итераций и большого объема программного кода, могут быть просто и эффективно реализованы в типах CLR базы данных с использованием членов типа данных `string` среды .NET и мощного класса `REGEX`.
- Безопасность и устойчивость сервера подвергаются риску, когда для доступа к файловой системе, сетевым ресурсам или прочим внешним данным с помощью интерфейса OLEDB или служб командной строки используется T-SQL. Модель безопасности CAS среды выполнения CLR предоставляет собой ценный уровень абстракции системы безопасности между SQL Server и внешним миром, поскольку рассматривает программный код, а не отдельных пользователей. Еще одним доказательством преимуществ использования компонентов CLR в качестве интерфейса между SQL Server и внешними ресурсами является то, что CLR реализует элегантные методы создания табличного вывода в ядре базы данных из нетабличных данных.

Вряд ли множество решений способно мгновенно реализовать значительное число пользовательских типов и пользовательских консолидаций. Как всегда, адаптация к новой технологии будет плестись в хвосте адаптации к новым формам CLR существующих технологий базы данных. Большинство организаций использует еще более консервативный подход к внедрению пользовательских типов и консолидаций, так как их полный потенциал еще не был продемонстрирован производителем. Не нужно и говорить, что организации, слишком рано попытавшиеся реализовать модели данных, включающие пользовательские типы и консолидации, вряд ли получат большую отдачу от новых технологий. В следующих двух главах будет представлен ряд частных случаев использования пользовательских типов и консолидаций, а также детали их реализации.

T-SQL еще рано сбрасывать со счетов

Пакетные запросы, с использованием интеграции CLR или без нее, остаются лучшим способом доступа к реляционной базе данных. Такие запросы можно создать только на языке T-SQL. Естественно, можно реализовать их и в компонентах CLR, но разве сопоставимы затраты? Будет ли у оптимизатора такой же шанс сгенерировать наилучший план выполнения, если все запросы будут реализованы в коде .NET, а не в хранимых процедурах T-SQL? Будет ли прозрачность такого стиля программирования адекватной удобству обеспечения защиты данных и сопровождения программного кода? Ответ на каждый из этих вопросов будет строго отрицательным. Именно поэтому SQL уверенно сохраняет свои позиции, и именно поэтому хранимым процедурам отдается предпочтение, а динамический SQL считается рискованным и трудно поддерживаемым решением.

С появлением CLR возникла новая неопределенность. Только опыт и знание приложения могут помочь точно установить, где лучше всего реализовывать заданные требования: на сервере базы данных или в потоке. Если выбирается база данных, то следует решить, чему лучше адресовать выполнение требований: языку T-SQL или интеграции CLR. Баланс между дополнительной нагрузкой на сервер и уменьшением объема сетевых потоков данных должен быть тщательно просчитан. Высокотехнологичные приложения при перемещении логики на сервер баз данных теряют больше, чем обычные офисные утилиты с ограниченной нагрузкой. Также понятно, что цена ошибочного решения и перемещения всей логики на сервер баз данных лишь с целью дальнейшего расширения приложения будет высокой, если станет очевидной необходимость переноса функций, основанных на T-SQL, в управляемый код на другом уровне приложения.

Компания Microsoft как бы подсказывает нам, что цена перемещения кода интеграции CLR из базы данных на уровень приложения будет значительно меньше. Время покажет. Могут настать времена, когда выполнение запроса доступа к данным из компонентов CLR станет лучшим вариантом решения задачи, а может случиться и так, что использование старого доброго курсора T-SQL обеспечит наилучшую структуру запроса. Большей частью CLR не заменяет пакетную обработку запросов, так же как T-SQL нельзя назвать лучшим решением итеративной обработки наборов данных. Предполагая, что логика приложения корректно воплощена на уровне базы, лучше использовать T-SQL для доступа к базе данных, а CLR — когда в одном запросе нужно выполнить нечто большее, нежели обычная операция доступа. Создание приложений на уровне базы данных с целью дальнейшего масштабирования на другие уровни является довольно сомнительным решением; оно будет требовать большего использования типов интеграции CLR, а не объектов T-SQL.

Резюме

В этой главе мы лишь кратко познакомились с обширной средой .NET Framework. Были представлены ключевые концепции общеязыковой среды выполнения CLR и объектно-ориентированного программирования в среде .NET Framework. Эти концепции являются фундаментом понимания технических требований, предъявляемых к созданию компонентов интеграции CLR в базе данных. Были также представлены все пять типов объектов SQL Server, которые можно создать с помощью интеграции CLR. Было показано, как для поддержки интеграции CLR используются новые классы .NET и какие новые инструкции DDL T-SQL необходимы для интеграции CLR в производственной базе данных. Было показано, как, сведя вместе все эти концепции, создавать и устанавливать объекты CLR в базе данных. Также мы поразмышляли о том, в каких случаях стоит подумать об интеграции CLR, а когда лучше применять старый добрый T-SQL.

Влияние интеграции CLR на SQL Server с выходом версии SQL Server 2005 все еще остается неизведанной территорией, однако ее потенциал неоспорим.

ГЛАВА

28

Создание запросов в брокере служб

В этой главе...

Определение рабочей очереди

Обслуживание сообщений в очереди

Управление диалогами

Брокер служб (Service Broker) является одним из самых любимых мною компонентов SQL Server 2005. Эта мощная и одновременно простая в эксплуатации система обслуживания очередей позволяет создавать асинхронные сообщения и очереди их обслуживания на уровне абстракции базы данных, что обеспечивает высокую масштабируемость. Это существенно в любой архитектуре данных SOA.

Даже если вы создаете таблицу с подлежащей для выполнения работой (например, заказы, которые должны быть обработаны в системе MRP), то фактически вы создаете очередь. Для приложений брокер служб выполняет именно это — поддержку высокопроизводительных рабочих очередей, интегрированных в SQL Server, с возможностью мониторинга и выполнения инструкций DDL.

Брокер служб можно также использовать для передачи сообщений с гарантированной доставкой между разными рабочими очередями. Это открывает массу новых возможностей.

Так как брокер служб по сути является всего лишь таблицей SQL Server, он наделен всеми встроенными в SQL Server возможностями транзакций и резервирования. Именно это отделяет брокер служб от других технологий обслуживания очередей, таких как MSMQ.

Очередь содержит один широкий столбец с телом сообщения, так как сообщение обычно содержит один файл XML, его фрагмент, или сообщение SOAP.



Технология брокера служб была внедрена только в версии SQL Server 2005, так что всю эту главу можно считать нововведением в SQL Server.

Брокер служб по умолчанию отключен, поэтому первым шагом является его включение с помощью инструкции ALTER DATABASE:

```
ALTER DATABASE AdventureWorks SET
ENABLE_BROKER;
```

Конфигурирование очереди сообщений

Брокер служб использует обмен сообщений (или метафору диалога), однако занимается он не только сообщениями. Объект брокера можно определить следующим образом.

1. *Тип сообщения* определяет реальные требования, предъявляемые к сообщению.
2. *Контракт* определяет соглашение между исходной и целевой службой, в которое входит тип сообщения, очередь и службы.
3. *Очередь* хранит список сообщений.
4. *Служба* взаимодействует с очередью, отправляя или получая сообщения на объектах источника и приемника.

Кроме определения типа сообщения как XML и именования объектов, в настройке брокера служб нет никаких сложностей. Причиной этого является то, что язык определения данных DDL выполняет всю работу, а брокер служб всего лишь обслуживает очередь, являющуюся инфраструктурой для сообщений. Гораздо больше работы заключено в помещении сообщения в очередь и извлечении его из нее.

Так как брокер служб интегрирован в SQL Server, его объекты создаются с помощью хорошо знакомой инструкции DDL `CREATE`.

При создании очереди брокера служб в первую очередь следует определить тип сообщения и контракт, используемый для этого типа:

```
CREATE MESSAGE TYPE HelloWorldMessage
  VALIDATION = WELL_FORMED_XML ;
CREATE CONTRACT HelloWorldContract
  ( HelloWorldMessage SENT BY INITIATOR);
```

Очереди источника и приемника также создаются с помощью DDL:

```
CREATE QUEUE [dbo].[TargetQueue] ;
CREATE QUEUE [dbo].[InitiatorQueue] ;
```

Аналогично, с помощью DDL определяются службы источника и приемника. Обе службы ассоциированы с очередью, а в службе приемника дополнительно определяется то, что она может получать сообщения из контракта:

```
CREATE SERVICE InitiatorService
ON QUEUE [dbo].[InitiatorQueue];
GO

CREATE SERVICE TargetService
ON QUEUE [dbo].[TargetQueue]
(HelloWorldContract);
```

Когда объект брокера служб создан, его можно увидеть в списке узла `Servece Broker` в окне Object Explorer.

Работа с диалогами

После того как объект брокера служб создан, сообщения могут помещаться в очередь и извлекаться из нее. Сообщения существуют как часть *общения*, которое может быть разбито на *группы взаимодействия*.

Отправка сообщения в очередь

В следующем программном коде создается объект общения, использующий идентификатор GUID conversationhandle. Команда OPEN CONVERSATION открывает общение, а команда SEND непосредственно помещает сообщение в очередь:

```
BEGIN TRANSACTION ;

DECLARE @message XML ;
SET @message = N'<message>Hello, World!</message>' ;

DECLARE @conversationHandle UNIQUEIDENTIFIER ;

BEGIN DIALOG CONVERSATION @conversationHandle
  FROM SERVICE [InitiatorService]
  TO SERVICE 'TargetService'
  ON CONTRACT [HelloWorldContract]
  WITH ENCRYPTION = OFF, LIFETIME = 1000 ;

SEND ON CONVERSATION @conversationHandle
  MESSAGE TYPE [HelloWorldMessage]
  (@message) ;

END CONVERSATION @conversationHandle ;

COMMIT TRANSACTION ;
```

Для просмотра сообщений в очереди их можно извлечь из таблицы очереди точно так же, как из обычной реляционной таблицы:

```
SELECT CAST(message_body as nvarchar(MAX)) from [dbo].[TargetQueue]
```

Получение сообщений

Команда RECEIVE извлекает и удаляет из очереди самые старые сообщения. Заключайте эту команду в транзакцию, чтобы, если что-то пойдет не так, операцию получения можно было откатить и сообщение осталось в очереди.

Брокер служб не является триггером, который запускается, когда сообщение помещается в очередь. Для извлечения сообщения нужно написать некоторый программный код. Компания Microsoft предложила для этих целей новый параметр команды WAITFOR, позволяющий ожидать поступления в очередь сообщения. Если этого параметра не было, программе пришлось бы для проверки наличия в очереди нового сообщения запускать цикл.

Следующий программный код в хранимой процедуре будет ожидать поступления сообщений в очередь, а затем получать первое из них:

```
USE AdventureWorks ;
GO
-- Обработка всех групп взаимодействия.
WHILE (1 = 1)
BEGIN
DECLARE @conversation_handle UNIQUEIDENTIFIER,
  @conversation_group_id UNIQUEIDENTIFIER,
  @message_body XML,
  @message_type_name NVARCHAR(128);

BEGIN TRANSACTION ;
-- Получение следующей группы взаимодействия.
WAITFOR(
```

```
  GET CONVERSATION GROUP @conversation_group_id
  FROM     [dbo].[TargetQueue]),
  TIMEOUT 500 ;

-- Если групп взаимодействия больше нет, откатываем
-- транзакцию и выходим из внешнего цикла WHILE.
IF @conversation_group_id IS NULL
BEGIN
  ROLLBACK TRANSACTION ;
  BREAK ;
END ;

-- Обрабатываем все сообщения в группе взаимодействия.
-- Обратите внимание, что вся обработка сосредоточена
-- в одной транзакции.
WHILE 1 = 1
BEGIN
   -- Получение следующего сообщения в группе взаимодействия.
   -- Обратите внимание, что инструкция получения
   -- включает предложение WHERE для гарантии того,
   -- что полученное сообщение относится именно к данной
   -- группе взаимодействия
   RECEIVE
     TOP(1)
     @conversation_handle = conversation_handle,
     @message_type_name = message_type_name,
     @message_body =
     CASE
        WHEN validation = 'X' THEN CAST(message_body AS XML)
        ELSE CAST(N'<none/>' AS XML)
     END
   FROM [dbo].[TargetQueue]
   WHERE conversation_group_id = @conversation_group_id ;

   -- Если больше нет сообщений или возникла ошибка,
   -- завершаем обработку данной группы взаимодействия.
   IF @@ROWCOUNT = 0 OR @@ERROR <> 0 BREAK;
   -- Отображаем полученную информацию.
      SELECT 'Conversation Group Id' = @conversation_group_id,
             'Conversation Handle' = @conversation_handle,
             'Message Type Name' = @message_type_name,
             'Message Body' = @message_body ;

   -- Если message_type_name указывает на то, что сообщение
   -- является ошибкой или сообщением конца диалога,
   -- завершаем общение.
   IF @message_type_name =
      'http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog'
       OR @message_type_name =
       'http://schemas.microsoft.com/SQL/ServiceBroker/Error'
   BEGIN
     END CONVERSATION @conversation_handle ;
   END ;
 END; -- Обработка всех сообщений в группе взаимодействия.

 -- Подтверждение инструкций получения и завершение общения.
 COMMIT TRANSACTION ;
END ; -- Обработка всех групп взаимодействия.
use tempdb;;
```

Мониторинг брокера служб

Несмотря на отсутствие средств визуализации очередей в Management Studio и сводного отчета для объекта очереди, можно создать выборку непосредственно из очереди. К тому же существуют представления каталогов базы данных, которые могут пролить свет на очереди.



Чтобы просмотреть деморолики работы с Web-службами, брокером служб и хранимыми процедурами, а также средств мониторинга брокера служб, обратитесь к разделу ScreenCast, доступному на Web-сайте книги (`www.SQLServerBible.com`).

Резюме

Брокер служб является одной из тех технологий, которые нельзя просто "распаковать" и сразу воспользоваться всеми ее преимуществами. Если не приложить усилий к проектированию базы данных с использованием брокера служб, то невозможно будет получить никакой отдачи. Однако если уделить этому вопросу особое внимание, то вы получите существенный прорыв в вопросах масштабируемости, достигаемый благодаря использованию очередей брокера служб.

В следующей главе мы продолжим разговор о расширенных технологиях SQL Server и рассмотрим библиотеку ADO.NET 2.0 с ее мощными методами взаимодействия с базой данных.

Поддержка пользовательских типов данных

ГЛАВА

29

В этой главе...

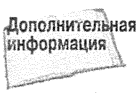
Компоненты и требования пользовательских типов данных среды .NET

Создание пользовательских типов интеграции CLR в Visual Studio 2005

Соглашения относительно отладки и производительности для пользовательских типов

Работа с пользовательскими типами интеграции CLR в базе данных и приложениях .NET

Пользовательские типы CLR являются нововведением в версии SQL Server 2005. Не следует путать их с пользовательскими типами данных. СУБД SQL Server уже поддерживала некоторое время пользовательские типы данных в ограниченном контексте. Это расширение языка T-SQL было предназначено в основном для удобства работы с базой данных. Ранее создание новых типов данных не поддерживалось — существовал механизм присвоения псевдонимов существующим типам, таким как `integer`, `char` и `binary`, для обеспечения связности в базе данных. С его помощью было удобнее идентифицировать взаимоотношения типов в столбцах множества таблиц и придавать столбцам, предназначенным для одной цели в нескольких таблицах, одинаковую длину. Например, все такие столбцы могли быть отображены на один пользовательский тип данных. В то же время приложения, ссылающиеся на такие столбцы, видели их с соответствующим "родным" типом данных SQL Server. Приложения фактически не догадывались о существовании созданного пользовательского типа — он не имел смысла вне базы данных, в которой был создан. На уровень приложения действие пользовательских типов данных не распространялось.



Пользовательские типы являются одним из пяти типов интеграции CLR в SQL Server 2005. Общее рассмотрение типов интеграции CLR в SQL Server 2005 см. в главе 27.

Все изменилось с интеграцией в SQL Server 2005 общеязыковой среды выполнения CLR. В ней были введены пользовательские типы данных. Ранее существовавший механизм создания пользовательских типов из существующих примитивов остался в силе — в T-SQL по-прежнему можно создавать псевдонимы для типов данных. На этом сходство прежних и нынешних пользовательских типов данных заканчивается. Новые пользовательские типы имеют силу вне локальной базы

данных. При реализации в среде .NET пользовательские типы могут включать в себя больше одного примитива. Эти типы могут иметь любых членов классов .NET, таких как свойства, методы и поля. Сам разработчик несет ответственность за то, чтобы пользовательский тип был известен на всех уровнях приложения, которые будут его использовать, не только с помощью определения, лежащего в основе примитива, но и с помощью определения имени и структуры типа. Ссылка на определение пользовательского типа должна присутствовать во всех программах, которые будут его использовать, — как на уровне базы данных, так и на уровне приложения.

Типы интеграции CLR отличаются от объектов базы данных SQL Server, существовавших до выхода версии SQL Server 2005. Хранимые процедуры, функции и методы, созданные в CLR, состоят из одного метода. Эти методы имеют своих двойников в языке T-SQL; того же нельзя сказать о пользовательских типах.

Пользовательские типы определяются как класс или структура. Когда они создаются как класс, в него внедряется вся функциональность типа. Компания Microsoft высказала несколько обескураживающее предупреждение о том, что пользовательские типы не рекомендуется использовать в качестве бизнес-объектов, в то время как на самом деле пользовательские типы позволяют хранить объекты в базе данных. Возможно, это предупреждение было вызвано тем, что компания Microsoft еще не считает интеграцию CLR полностью готовой для создания бизнес-объектов в базе данных, или тем, что пользовательские типы никогда не смогут обеспечить полноту поддержки таких бизнес-объектов, как клиенты или заказы, в базе данных. Чтобы получить ответ на этот вопрос, необходимо исследовать работу пользовательских типов вне SQL Server 2005.

Аналогично, пользовательские типы нельзя рассматривать как структуры массивов, несмотря на то, что в период бета-тестирования версии SQL Server 2005 существовало множество примеров, где эти типы были обычными массивами. Массив является коллекцией повторяющихся данных. которую можно сравнить со строкой таблицы базы данных. В общем случае массивы меньше таблиц, но имеют срок жизни, ограниченный текущим контекстом выполнения. Если базе данных нужны массивы, подумайте об использовании хранимых процедур или функций CLR, но не о пользовательских типах.

Пользовательские типы завладеют сознанием креативных программистов, однако посеют страх среди традиционных администраторов баз данных. Очень важно для каждого хорошо осознать правила и ответственность, связанные с внедрением пользовательских типов в SQL Server 2005. Несмотря на фундаментальное понимание процессов создания пользовательских типов и управления ими, существует доля неопределенности в том, как пользовательские типы нарушили равновесие тех баз данных, какими мы их знали ранее. Хотите вы того или нет, но дверь в объектно-реляционную модель базы данных уже открыта.

Планка для разработчиков T-SQL и Visual Basic, создающих другие типы интеграции CLR, поднята не так высоко, как для тех, кто стремится "приручить" пользовательские типы. Исключением, пожалуй, являются пользовательские консолидации — далее будет показано, что они являются частным случаем пользовательских типов.



Пользовательские консолидации являются еще одним типом интеграции CLR, введенной в SQL Server 2005.

Остальные типы интеграции CLR дублируют аналогичные объекты базы данных по форме и функциональности. В то же время для пользовательских типов параллель в SQL Server провести нельзя. Все разработчики должны понимать следующее:

- каковы фундаментальные различия между классом и структурой;
- какова связь между типами базы данных и типами .NET;

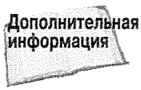
- как поддерживать пустые значения базы данных в среде .NET Framework;
- как отлаживать и поддерживать типы интеграции CLR в базе данных;
- как зависимости пользовательских типов базы данных отличаются от других типов интеграции CLR;
- почему необходимо развертывать тип на всех уровнях приложения, которые на него ссылаются.

Даже самым опытным программистам приложений необходимо провести адекватную экспертизу T-SQL для функционального тестирования нового типа и — даже если было принято решение реализовать пользовательский тип как класс, а не структуру, — "прогнать" на примерах структуру и членов типа. И наоборот, опытным разработчикам T-SQL необходимо понимать основы и концепции программирования в среде .NET Framework, от делегирования до публикации.

Создание пользовательских типов интеграции CLR

Программирование пользовательских типов требует создания полноценного класса или структуры. Как и в других типах интеграции CLR, бо́льшая часть программного кода скрыта от разработчика, который должен реализовать только частичный класс, что требует написания одного метода. Проще говоря, требования, выдвигаемые при создании пользовательского типа, не отличаются от тех, которые выдвигаются при создании любого другого класса. К тому же существует значительный список дополнительных требований к созданию, тестированию и использованию пользовательских типов. Очень важно выработать единые соглашения для пользовательского типа, поскольку он будет потенциально использоваться на всех уровнях приложения. Вносить изменения гораздо сложнее, чем тщательно продумать все с самого начала. Отладка и тестирование пользовательского типа добавляют уровень зависимостей данных и сборок .NET, выходящих за пределы типа CLR. Развертывание пользовательского типа требует полного понимания архитектуры среды .NET Framework. Во многих реальных ситуациях требуются манипуляции с глобальным кэшем сборки или, в качестве альтернативы, беспрецедентное объединение уровней приложения и базы данных.

Большинство разработчиков для создания пользовательских типов используют программы Visual Studio 2005 Professional или Team System. В редакциях Standard и Express этой среды разработки не реализованы все функции отладки компонентов интеграции CLR. Также можно создавать и развертывать пользовательские типы с помощью инструментария разработки программ .NET SDK. Этот SDK использует те же специальные компиляторы языков программирования, что и Visual Studio. После компиляции сборка пользовательского типа загружается в SQL Server с помощью инструкций T-SQL `CREATE ASSEMBLY` и `CREATE TYPE`, независимо от того, какие инструменты были использованы для написания и компиляции типа. Реальное значение использование интерфейса Visual Studio 2005 приобретает в процессе разработки и отладки. С помощью Visual Studio 2005 разработчик может создавать тестовые сценарии и управлять ими, что позволяет решить массу вопросов развертывания, пока тип не будет готов для тестирования интеграции. Хотя бы только по этой причине программистам, только начинающим работать со средой .NET Framework, рекомендуется использовать интерфейс Visual Studio, а не пытаться одновременно изучить .NET SDK и .NET Framework. Сэкономленное время и повышенное качество программного кода с лихвой окупят средства, потраченные на покупку лицензии Visual Studio 2005.



Все редакции SQL Server 2005 поддерживают интеграцию CLR. Для определения того, какая версия ядра базы данных лучше подходит для производственного использования или разработки, сравните характеристики SQL Server с требованиями, выдвигаемыми производственными условиями. Описание характеристик всех доступных редакций см. в главе 3.

Независимо от того, какие инструменты вы используете, .NET Software Developement Kit (SDK) или Visual Studio 2005, для успешного создания пользовательского типа нужно понять некоторые требования и принять конструктивные решения. Чем должен быть тип: классом или структурой? Как тип будет удовлетворять контракту программирования SQL Server 2005? Какой режим защиты доступа к коду (CAS) необходим для данного типа? Какой метод использовать для подписания сборки? Как управлять развертыванием на всех уровнях приложения?



Варианты подписания сборки .NET и режимы защиты CAS см. в главе 27.



Пренебрежение тщательным рассмотрением данных вопросов может создать клубок зависимостей, который будет тяжело распутать. Например, в базе данных тип нельзя изменить, если он используется в таблице, а на уровне приложения структура типа должна в точности соответствовать его структуре, используемой в базе данных. Внесение изменений в пользовательский тип может потребовать, чтобы данные из соответствующих столбцов были сохранены, а сам столбец удален (так как не существует инструкции `ALTER TYPE`). После этого новое определение типа должно быть загружено в базу данных, данные перемещены в новые столбцы, а ссылки в приложении на данный тип обновлены. Только после этого работа приложения может быть продолжена. Естественно, некоторые измененные части могут потребовать восстановления. Сбои приложения более вероятны, если одна из частей не была корректно модифицирована.

Удовлетворение требований

Создание и использование пользовательских типов интеграции CLR требуют наличия SQL Server 2005 и среды .NET Framework версии 2.0. Реализации пользовательских типов не имеют обратной совместимости. Пользовательские типы не могут быть использованы, и на них не могут содержаться ссылки в любых предыдущих версиях .NET Framework. Пользовательские типы могут существовать в приложениях .NET без наличия манифеста в базе данных, хотя требования при этом несколько изменяются. Несмотря на отдаленное сходство с типами VB6, которые могут объявляться в процедуре, со старыми псевдонимами типов SQL Server, пользовательские типы интеграции CLR должны объявляться как структура или класс VB.NET.

Причины того, почему для большинства пользовательских типов используют структуры, можно найти в самом фундаменте среды .NET Framework. *Структура* имеет тип значения, а *класс* — тип ссылки. Структура размещается в стеке вызова, а класс — в куче объекта. Эти, на первый взгляд, незначительные отличия предопределяют некоторые существенные различия в поведении классов и структур. Собранные вместе, эти отличия выливаются в относительно большой объем работы при программировании типов ссылок или классов. В табл. 29.1 перечислены некоторые отличия между структурой и классом в среде .NET Framework.

Среди причин реализации пользовательского типа в форме класса, а не структуры, можно упомянуть следующие.

- **Управление сроком жизни**. Можно рассматривать как достоинство то, что один процесс может создать экземпляр пользовательского типа, а другой — ссылаться на него даже после завершения процесса создателя. Например, заголовок управляющего пакета или структура идентификации родословной могут сохраняться в памяти на протяжении выполнения пакета или итерации.

Таблица 29.1. Сравнение структур и классов VB.NET

	Структура	Класс
Тип объявления	Значение	Ссылка
Размещение в памяти	Стек программы	Куча объекта
Минимальное количество членов	1	0
Обязательное количество защищенных членов	1	0
Конструктор	Конструктор по умолчанию недопустим. Члены инициализируются в значения по умолчанию	Конструктор по умолчанию обязателен (он используется для создания нового экземпляра)
Ограничения модели доступа	Защищенные члены недопустимы	Нет ограничений
События	Только общие (в языке C# статические) процедуры могут быть обработчиками событий	Обработчиком событий может быть любой метод
Откуда наследуется	Из `System.ValueType`	Из любого класса, кроме `System.ValueType`
Может ли иметь наследников	Нет	Да, если явно не объявлено `NotInheritable`
Полиморфизм	Для реализации интерфейса необходима упаковка (преобразование в тип ссылки)	Контракт программирования основанного на наследовании объекта поддерживает интерфейсы
Срок жизни	Не может существовать независимо	Существует, пока достижим
Уборка мусора	Нет. Экземпляры являются копиями, а не ссылками. Основывается на классе контейнера для уборки мусора	Да. Финализируется, когда не остается активных ссылок
Присвоение	При присвоении копируется (по значению)	Клонирование объекта (по ссылке)
При отсутствии присвоения	Члены доступны	Члены недоступны
Изоляция изменений	Экземпляр изолирован	Изменения влияют на другие ссылки
Равенство	Сравниваются отдельные члены	Для сравнения объектов используется метод `Equals`

- **Полиморфизм**. Даже несмотря на то, что в SQL Server не поддерживается наследование, если пользовательский тип управляется несколькими базовыми классами, то наследование может быть использовано с помощью интерфейсов управляемого класса пользовательского типа. В этом случае нагрузка на структуру в стеке будет выше нагрузки на класс в куче, поэтому тип значения будет *упаковываться*. Упаковка результата в копии значения обслуживается как в стеке программы, так и в куче объекта.
- **Наследование**. Если модель программирования имеет основания создать базовый пользовательский тип, из которого будут управляться другие типы, это может быть реализовано только с помощью класса. Структуры не поддерживают наследование.

- **Базовые типы объектов в пользовательском типе**. Если член пользовательского типа имеет тип ссылки, такой как `string`, то ссылка на него потребует создания объекта в куче объекта. Если тип создается как структура, то пользовательский тип будет создан в куче, но он будет содержать ссылку на этого члена объекта. Редко используемый член может быть инициализирован как `nothing`, что позволит лучше сохранить структуру типа значения. Часто используемый член менее вероятно сможет воспользоваться какими-либо преимуществами типа значения, которое существует в программном стеке.
- **Использование COM**. Это можно заявить только с небольшой натяжкой, но в случае, когда пользовательский тип внедряет некоторый старый программный код, важный для организации, с помощью interop-сборки, необходима тщательная проверка типа, созданного как класс или структура. Например, когда неспособная к корректному преобразованию переменная размещается interop-сборкой, в документации предупреждается о возможных проблемах. Среди таких типов — строки, массивы, объекты, классы и типы значений. Это не обходит стороной и множество других встроенных типов, поэтому, из соображений безопасности, если в типе планируется использование COM, то этот тип разумно создавать как класс.

Остальные важные требования будет лучше обсудить в контексте шаблона пользовательского типа программы Visual Studio 2005.

Программирование пользовательских типов CLR в Visual Studio

Некоторые из требований выдвигаются конкретным шаблоном, в то время как другие необходимы для всех типов. Последовательно просматривая сгенерированный шаблоном программный код для тестового типа, в этом разделе мы исследуем требования и различия в программировании, необходимые для достижения нужного поведения.

Создаваемый в качестве примера тип будет использоваться для хранения IP-адресов. Это самая распространенная схема адресации, используемой в настоящее время в Интернете. IP-адрес традиционно представляется четырьмя целыми числами, каждое из которых находится в диапазоне от нуля до 255, разделенными точками. В настоящее время большинство сетей назначают каждому узлу один или несколько IP-адресов для поддержки транспортного и сетевого уровней модели OSI. В Интернете используются символьные имена DNS, присвоенные IP-адресам. На практике довольно часто возникает потребность представлять IP-адреса в базе данных, однако в предыдущих версиях SQL Server было довольно проблематично обеспечивать доменную целостность этих значений. Обычно IP-адреса хранились как строки, а проверка выполнялась уже на уровне приложения, которому нужны были эти данные. С помощью пользовательских типов в SQL Server можно организовать проверку IP-адресов уже на уровне базы данных с помощью обычных выражений .NET. К тому же теперь стало возможным ссылаться на пространство имен `System.Net` из членов пользовательского типа.



Полное решение Visual Studio 2005 для пользовательского типа, обсуждаемого в настоящей главе, можно загрузить с Web-сайта книги. В этом решении содержатся рабочие примеры типа IP-адреса, реализованного тремя способами:

- как структуры в своей простейшей форме;
- как структуры с пользовательским форматом;
- как классы.

Давайте посмотрим, как можно создать пользовательский тип для IP-адреса. В простой структуре пользовательского типа можно увидеть его элементы. В первую очередь обратите внимание на системные сборки, которые должны быть включены во все пользовательские типы:

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Text.RegularExpressions

' Если столбец с пользовательским типом будет индексироваться,
' необходимо установить свойство IsByteOrdered в значение true
<Serializable()> _
<Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native,
IsByteOrdered:=True)> _
Public Structure IPType1
  Implements INullable

  Private Shared ReadOnly _parser As _New _
   Regex("\A((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|
          [01]?\d\d?)")
  Private Const _NULL As String = "NULL_IP"
  Private m_Null As Boolean
  Private m_OctetA As Byte
  Private m_OctetB As Byte
  Private m_OctetC As Byte
  Private m_OctetD As Byte

  Public Overrides Function ToString() As String
    If Me.IsNull Then
      Return _NULL
      ' можно также использовать
      ' Return Nothing
    Else
      Return Me.m_OctetA.ToString() & "." & _
       Me.m_OctetB.ToString() & "." & _
       Me.m_OctetC.ToString() & "." & _
       Me.m_OctetD.ToString()
    End If
End Function

Public ReadOnly Property IsNull() As Boolean Implements _
INullable.IsNull
  Get
    Return m_Null
  End Get
End Property

Public Shared ReadOnly Property Null() As IPType1
  Get
    Dim h As IPType1 '= New IPType1
    h.m_Null = True
    Return h
  End Get
End Property

Public Shared Function Parse(ByVal s As SqlString) As IPType1
  If s.IsNull Or s.Value = _NULL Then
  ' база данных определяет допустимость в столбце пустых значений
```

```
    Return Null
  End If
  ' конструктор не обязателен для структуры
  ' Допустимо, но не обязательно, чтобы
  ' каждый член имел значение по умолчанию, чтобы каждый октет
  ' IP-адреса был инициализирован нулем
  Dim u As IPType1 '= New IPType1

  'строка .NET должна быть разобрана на составляющие
  Dim str As String = Convert.ToString(s)
  Dim m As Match = _parser.Match(str)
  If m.Success Then
    Dim arr() As String = str.Split(CType(".", Char))
    u.OctetA = CType(arr(0), Byte)
    u.OctetB = CType(arr(1), Byte)
    u.OctetC = CType(arr(2), Byte)
    u.OctetD = CType(arr(3), Byte)
    Return u
  Else
    Throw New ArgumentException("Invalid IP v4 Address")
    ' Return Nothing
  End If
End Function

'results in 0.0.0.0 = null
Public Property OctetA() As Byte
  Get
    OctetA = m_OctetA
  End Get
  Set(ByVal value As Byte)
    m_OctetA = value
  End Set
End Property

Public Property OctetB() As Byte
  Get
    OctetB = m_OctetB
  End Get
  Set(ByVal value As Byte)
    m_OctetB = value
  End Set
End Property

Public Property OctetC() As Byte
  Get
    OctetC = m_OctetC
  End Get
  Set(ByVal value As Byte)
    m_OctetC = value
  End Set
End Property

Public Property OctetD() As Byte
  Get
    OctetD = m_OctetD
  End Get
  Set(ByVal value As Byte)
    m_OctetD = value
  End Set
```

```
End Property

<SqlMethod(IsDeterministic:=True, IsPrecise:=True)> _
Public Function GetCSubNet() As String
  GetCSubNet = m_OctetC.ToString + "." + m_OctetD.ToString
End Function
End Structure
```

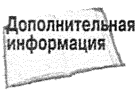
Если нужно заложить в тип другие функции, то в проект интеграции CLR нужно включить и другие пространства имен с помощью команды `Imports` и ссылки на соответствующую сборку. Например, к обычным выражениям, инструментам WMI (Windows Management Instrumentation) и множеству сетевых служб можно получить доступ из типов интеграции CLR с помощью предлагаемых системой пространств имен .NET. Для использования регулярных выражений добавьте следующий оператор:

```
Imports System.Text.RegularExpressions
```

Далее объявляется класс или структура. Некоторые важные аспекты объявления могут варьироваться в зависимости от того, на чем основан тип — на структуре или классе. В первую очередь следует определить обязательные атрибуты. Затем указывается область определения класса и его имя. Если класс является производным от другого, то далее указывается родительский класс. Заметим, что класс может наследовать только от одного класса, а структура не может наследовать вообще. Если класс реализует какой-либо интерфейс, реализации этого интерфейса перечисляются после родительского объекта. Закрывается класс или структура оператором `End Class` или `End Structure`.

Теперь рассмотрим объявления элементов структуры или класса более подробно.

Атрибут `Serializable` обеспечивает поддержку метаданных для представления состояния данных пользовательского типа в потоке байтов при их транспортировке и использовании, а также для упаковки потока с целью хранения в типе данных. Атрибут `Serializable` не обязателен в пользовательском типе, однако в большинстве ситуаций может оказаться полезным. Конечно, сериализация помогает определить, как поток байтов будет разделяться на данные членов пользовательского типа. В общем случае сериализация занимается объединением всех полей пользовательского типа в двоичный поток или поток XML. Атрибут `Sertializable` не имеет параметров — он просто должен быть определен, после чего в теле структуры или класса должны быть определены интерфейсы и методы сериализации.



Дискуссию об использовании пользовательских атрибутов типов интеграции CLR см. в главе 27.

Атрибут `SqlUserDefinedType` является специализированным и используется только для пользовательских типов. Как отмечалось в главе 27, компилятор использует этот атрибут при компиляции сборки в код MSIL. Он также оказывает влияние на манифест сборки и используется SQL Server во время загрузки сборки в сервер. Этот атрибут довольно емкий; он содержит четыре параметра. Во-первых, следует четко определить формат хранения типа; по умолчанию в шаблоне предусмотрен формат `Fonmat.Native`. Этот формат способен обеспечить наибольшую совместимость и производительность при минимальном объеме дополнительного программирования. Если столбцы, создаваемые с данным типом, будут участвовать в индексации, также необходимо установить параметр `IsByteOrdered`. Среди остальных значений атрибута очень полезный `Format.UserDefined` и относительно безынтересный `Format.Unknown`.

Если пользовательский тип определяется как класс, использующий "родной" формат, то для обеспечения совместимости с COM должен быть установлен атрибут `StructLayout`. Это выполняется с помощью добавления пространства имен `InteropServices` и присвоения

свойству LayoutKind значения StructLayout. Возможными значениями атрибута также являются Auto, Explicit и Sequential, и в большинстве пользовательских типов интеграции CLR обычно используют значение Sequential. В общем случае значение Auto указывает системе самой принять решение относительно компоновки полей пользовательского типа; значение Explicit позволяет программисту явным образом определить компоновку, а значение Sequential инструктирует программу располагать поля в порядке их отправки. По умолчанию компилятор Visual Studio использует раскладку Sequential.

Импортируется пространство имен:

```
Imports System.Runtime.InteropServices
```

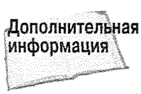
Параметр LayoutKind.Sequential включается с остальными пользовательскими атрибутами:

```
<System.Runtime.InteropServices.StructLayout(LayoutKind.Sequential)> _
```

Для определения способа доступа к типу используется область определения структуры. В VB.NET уровни доступа могут быть установлены с помощью следующих модификаторов: Public, Private, Friend, Protected и ProtectedFriend. В других языках программирования семейства .NET существуют аналогичные модификаторы. Чтобы пользовательский тип был доступен вне домена приложения (как вы помните из главы 27, домен приложения является аналогом схемы базы данных), уровень доступа должен быть установлен в Public:

```
<Serializable()> _
<Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native,
IsByteOrdered:=True)> _
Public Structure IPType1
  Implements INullable
```

В отличие от класса структура не может наследовать от другой структуры. В то же время, подобно классу, структура может оттенять (или скрывать, или полностью замещать) другой программный элемент, имеющий точно такое же имя, с помощью модификатора Shadows.



Дополнительные ресурсы, из которых можно почерпнуть информацию о работе со средой .NET Framework, см. в главе 27.

Класс может наследовать от другого класса. Это значит, что в дополнение к модификаторам уровня доступа и Shadows доступны и два дополнительных для базового класса типа, определяющего область наследования класса.

- Модификатор MustInherit требует, чтобы класс мог использоваться только для наследования, и запрещает создание экземпляров данного класса.
- Модификатор NotInheritable запрещает использование данного класса другими для наследования.

Если создается базовый тип, на котором будет основана масса других типов, и вы не хотите, чтобы базовый тип был реализован сам по себе в SQL Server, используйте модификатор MustInherit. Если из соображений безопасности или производительности вы не хотите, чтобы на базе данного типа создавались другие, используйте модификатор NotInheritable.

Если тип объявляется как класс, то он может наследовать только от одного базового класса. Для указания базового класса в языке VB.NET предусмотрено ключевое слово Inherits (сопровождаемое именем класса), которое вставляется в объявление непосредственно после имени типа. Необходимо также включить ссылку на сборку, где можно найти базовый класс (если он находится в другой сборке), а также пространство имен в разделе Imports в верхней части файла. Несмотря на то что тип может управляться из базового класса, SQL Server

не гарантирует наследование. Если должны использоваться члены управляющего класса, они должны быть явно включены в текущий класс.

Когда в работе не участвует SQL Server, члены базового класса неявно доступны в управляемом классе. Однако в интеграции CLR в SQL Server в настоящее время наследование не поддерживается. SQL Server требует, чтобы общедоступные члены не замещались, и неявное наследование не распознается средой выполнения CLR в SQL Server. Совместно все это выдвигает требование не использовать в интеграции CLR наследование в его классической форме. Это ограничение существенно влияет на рассмотрение интеграции CLR как объектно-ориентированной среды выполнения. В этом смысле более дружественным подходом к управлению функциональностью из других классов будет реализация интерфейса, а не наследование из базового класса.

```
<Serializable()> _
<Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native,
IsByteOrdered:=True)> _
<System.Runtime.InteropServices.StructLayout(LayoutKind.Sequential)> _
Public Class IPType3
  Implements INullable
```

Последнее, что нужно вставить в объявление класса, — это имена интерфейсов, которые будут использоваться типом. Класс может использовать несколько интерфейсов. В пользовательский тип всегда нужно внедрять интерфейс `INullable` из пространства имен `System.Data.SQLTypes`. Этот интерфейс содержит свойство `IsNull`, позволяющее среде .NET Framework функционировать в базе данных, допускающей пустые значения в какомлибо из типов данных.

Среди других интерфейсов, которые реализуются встроенными типами данных SQL Server, можно упомянуть `IComparable` и `IXMLSerializable` из пространства имен `System`. Интерфейс `IComparable` служит для поддержки сравнений в среде .NET. и позволяет разработчику определять, как экземпляр пользовательского типа может быть вычислен в выражении. `IXMLSerializable` является интерфейсом из пространства имен `System.XML`, обеспечивающим поддержку сериализации и десериализации между структурой хранилища и потоками XML путем подмены методов `ReadXML` и `WriteXML`. Еще один интерфейс, интересный с точки зрения пользовательских типов, используется в удаленных и внутренних организационных ситуациях. Это интерфейс `IBinarySerialize` из пространства имен `Microsoft.SQLServer.Server`. Его методы `Read` и `Write` должны быть реализованы при использовании формата сериализации `UserDefined`.

```
<Serializable()> _
<Microsoft.SqlServer.Server.SqlUserDefinedType(Format.UserDefined, _
  IsByteOrdered:=True, _
  IsFixedLength:=True, _
  MaxByteSize:=4, _
  Name:="IPv4")> _
Public Structure IPType2
  Implements INullable, IComparable, IBinarySerialize,_
    IXMLSerializable
```

Напомню, что формат сериализации в атрибуте пользовательского типа `SQLUserDefinedType` обычно определен как `Native`. Формат сериализации `UserDefined` может пригодиться в случаях, когда структура или класс пользовательского типа содержит свойства, не соответствующие встроенным в SQL Server числовым и временным типам. К тому же, когда формат определен как `UserDefined`, идентификатор свойства `IsByteOrdered` является обязательным, а свойство `MaxByteSize` пользовательского типа должно находиться в диапазоне от 1 до 8 000. Эти ограничения размера выдвигает SQL Server. Лимит в 8 Кбайт

проясняет причину, по которой бизнес-объекты пока нельзя считать хорошими кандидатами на реализацию с помощью пользовательских типов.

Методы `Read` и `Write`, необходимые для сериализации `UserDefined`, выглядят несколько туманно, поскольку в документации отсутствует их четкое описание. К сожалению, это может привести многих разработчиков (в частности, тех, кто пытается найти свой путь из T-SQL в .NET) к решению замкнуться в сериализации `Native`, избегая того, чего они не понимают. Может, это и стало бы хорошим решением для редко используемых пользовательских типов, предназначенных для реализации правил бизнес-логики, однако сериализация `Native` может стать сдерживающим фактором внедрения пользовательских типов, когда на первое место выходят вопросы производительности. Для примера представим, насколько полезно было бы сортировать географические координаты по таким атрибутам, как континент, страна или область. Аналогично, представьте, насколько удобнее сортировать IP-адреса по октетам, а не как единую строку. Сериализация `Native` совершенно не предназначена для потребностей подобных сортировок. В то же время сериализация `UserDefined` полностью развязывает программисту руки в вопросах перемещения и хранения данных.

В документации программы Visual Studio содержится гораздо лучшая дискуссия по вопросам сериализации, чем в документации по интеграции CLR в SQL Server. К сожалению, даже документация Visual Studio требует от читателя концептуального подхода к сериализации в среде .NET Framework. Когда она применяется в сложных бизнес-объектах, реализованных с помощью специализированных пользовательских типов SQL Server, которым необходимы только методы `Read` и `Write`, сложность таких объектов может охладить программиста. *Сериализация* является ничем иным, как кодированием значений всех полей членов экземпляра структуры или класса в позиционный поток битов в операциях транспортировки и копирования. Несколько странно узнать, что все поля, общедоступные и частные, должны быть загружены и направлены в поток, хотя для разработчика было бы достаточно явно сериализовать только поля членов, чтобы позволить состоянию экземпляра быть точно воссозданным на другом конце операции транспортировки или копирования.

В примерах с IP-адресом, описываемых в настоящей главе, один из типов использует сериализацию `UserDefined`. Четырехбайтовые частные поля должны всегда находиться в одной и той же последовательности, чтобы гарантировать представление одного и того же IP-адреса. Следствия нарушения порядка октетов IP или их неполноты при транспортировке очевидны для любого читателя, хотя бы немного знакомого с сетевым стеком TCP/IP. Сериализация гарантирует порядок и качество данных при передаче из базы данных клиенту. В методе `Read` необходимо только загрузить поля в объект `BinaryReader`. После этого в методе `Write` нужно извлечь поля из объекта `BinaryWriter` точно в том же порядке, который использовался в методе `Read`.

```
Public Sub Read(ByVal r As System.IO.BinaryReader) _
               Implements IBinarySerialize.Read
  m_OctetA = r.ReadByte()
  m_OctetB = r.ReadByte()
  m_OctetC = r.ReadByte()
  m_OctetD = r.ReadByte()
End Sub
Public Sub Write(ByVal w As System.IO.BinaryWriter) _
                Implements IBinarySerialize.Write
  w.Write(m_OctetA)
  w.Write(m_OctetB)
  w.Write(m_OctetC)
  w.Write(m_OctetD)
End Sub
```

В приведенном выше фрагменте программного кода октеты IP были определены в порядке от A до D. Это не так технически важно — главное, чтобы порядок полей в методах `Read` и `Write` в точности совпадал. Эта гибкость продемонстрирована в загружаемом коде, равно как и реализация интерфейса `IComparable`.

Тестирование и отладка пользовательского типа

Перед тем как пользовательский тип сможет быть адекватно использован, успешно скомпилированная сборка должна быть развернута в SQL Server. К тому же наиболее вероятные сценарии функционирования должны включать создание таблицы и выполнение солидного пакета инструкций DML, чтобы проверить правильность работы нового пользовательского типа в контексте базы данных.

Visual Studio в этом отношении предлагает для этого этапа разработки полезный компонент *Test Scripts* уровня проекта. Существуют три существенных преимущества использования этого проекта над развертыванием пользовательского типа на тестовом сервере (даже если Developer Edition SQL Server развернут в той же среде, что и Visual Studio) и тестированием с помощью Management Studio или утилиты командной строки `SQLCMD`. Первое из них заключается в том, что внимание разработчика может быть эксклюзивно сфокусировано на тестировании функциональности. Когда тестовый сценарий используется в проекте, пользовательский тип может быть развернут и переразвернут в ходе разработки с помощью пункта меню Debug⇨Visual Studio Deploy. Когда же используется Management Studio или какая-либо другая утилита, внешняя по отношению к Visual Studio, нужно постоянно вручную проверять наличие в ней ссылки на версию пользовательского типа, скомпилированную последней. Вторым преимуществом является то, что пошаговое прохождение из сценария T-SQL по коду CLR полностью поддерживается только в интерфейсе Visual Studio. Третье преимущество заключается в том, что версии тестовых сценариев можно поддерживать с помощью SourceSafe вместе с другими компонентами проекта.

Для создания и удаления любых таблиц и представлений, на которые содержатся ссылки в тестовом сценарии, полезно создать специальные инструкции DDL. Это сэкономит время, которое пришлось бы затратить на поиск таблиц и представлений для удаления в них объектов с изменяемым в процессе разработки типом.

Вопросы производительности

Оптимизация программного кода CLR требует не столько усилий, как настройка производительности кода T-SQL. В то время как на практике доводилось слышать множество историй о преобразовании 30-часовой хранимой процедуры в 10-минутный запрос, пользовательские типы интеграции CLR вряд ли окупят героические усилия, вложенные в их оптимизацию. Если пользовательские типы создавались более-менее корректно, то максимум, чего можно ожидать, — это устранения из кода нескольких циклов работы процессора. Если реализация типа была некорректной, то в производительности могут остаться некоторые узкие места, и аккуратный разработчик может их избежать. Например, если в пользовательском типе реализован метод, требующий уровня защиты `EXTERNAL_ACCESS`, и результат этого метода остался в наборе данных, когда значение типа было добавлено в таблицу, пользователь должен осознавать, что не существует способа управления тем, что может произойти в процессе извлечения внешнего значения. Очень вероятно, что низкая производительность операций ввода-вывода связана с задержками в сети или невосприимчивыми внешними источниками данных, к которым выполняется обращение из программного кода .NET класса пользовательского типа.

Рассмотрим следующий пример пользовательского типа IP-адреса, который использует пространство имен `System.Net` для преобразования IP-адреса в имя сервера с помощью класса DNS:

```
<SqlMethod(IsDeterministic:=False)> _
Public Function GetDNSName() As String
If Not (Me.IsNull) Then
  GetDNSName = _NULL
Else
  Try
    GetDNSName =_
      Dns.GetHostEntry(IPAddress.Parse(Me.ToString)).HostName
  Catch ex As Sockets.SocketException
    GetDNSName = "Socket Error: " & ex.SocketErrorCode.ToString
  End Try
End If
End Function
```



Как правило, следует остерегаться мощи интеграции CLR. Предположим, что описанный выше метод используется при каждом добавлении IP-адреса в таблицу. При этом имя компьютера будет сохраняться в таблице как вычисляемый столбец. Некоторые помехи способны затянуть продолжительность транзакции вставки и вывести ее за установленные пределы. А в SQL Server может случиться масса неожиданных и непредсказуемых событий, непосредственно не касающихся выполняемого кода CLR. Узкие места сети и конфликты DNS могут создать недопустимый уровень конкуренции в таблице. Все, что остается в данной ситуации операции вставки, — это смириться с задержками. К тому же в случаях, когда добавляемый IP-адрес не может быть преобразован в имя узла, увеличение нагрузки за счет обработки исключения `SocketException` для каждого вставляемого значения может также стать причиной замедления работы базы данных. Другими словами, потенциальные возможности интеграции CLR могут запросто привести к нежелательным последствиям, которые следует предвидеть при создании и использовании пользовательских типов.

Как и любой другой столбец данных, столбец, созданный с пользовательским типом, может существенно поднять производительность, если он будет проиндексирован. Столбец с пользовательским типом может индексироваться только в том случае, если он может быть отсортирован в двоичном порядке. *Гетерогенные пользовательские типы* (т.е. те, которые содержат смесь встроенных типов) могут помочь создать более полезные индексы, если был тщательно продуман порядок десериализации полей членов. Рассмотрим пользовательский тип с интегральными и символьными полями. Если символьное поле более интересно с точки зрения поиска и является наиболее вероятным претендентом на включение в выражение аргумента поиска предложения `WHERE`, при сериализации разумно вставлять значение этого поля в хранилище первым. Вычисляемые столбцы, управляемые членами класса пользовательского типа, также могут быть проиндексированы, если в атрибуты этих членов включено свойство `IsDeterministic`.

Еще один вопрос, который уже обсуждался в контексте структурных отличий семантики значений и ссылок, связан с тем, что пользовательские типы, созданные как структуры, имеют тип значений и хранятся в стеке программы, в то время как типы, созданные как класс, имеют тип ссылок и находятся в пространстве кучи объекта. В общем случае влияние этого отличия на производительность невелико. В примере кода для этой главы, содержащемся на Web-сайте книги, выполняется тест производительности, который заключается в следующем. Выполняется одно и то же количество вставок в пользовательские типы, созданные с помощью структур (семантика значений) и классов (семантика ссылок) и встроенной сериализации. Класс пользовательского типа использует сериализацию `UserDefined` и столбец со встроенным типом `varchar`. В результате выполнения этого теста не было замечено значительных отличий в про-

изводительности операций вставки. Тем не менее при проектировании пользовательского типа полезно оценить затраты на внедрение и знать о существующих различиях.

Когда технология пользовательских типов интеграции CLR пробьет себе дорогу в приложения, работающие с SQL Server, естественно, возникнут новые вопросы, связанные с производительностью. В этом процессе компания Microsoft, несомненно, будет постепенно совершенствовать модель пользовательских типов с целью повышения их производительности.

Развертывание пользовательских типов интеграции CLR

Существуют по крайней мере три способа работы с пользовательскими типами в среде разработки. Во время создания и тестирования функциональных единиц может использоваться интерфейс Visual Studio 2005, в котором будут выполняться инструкции T-SQL DDL и запросы, предназначенные для тестирования пользовательских типов. Как только новый тип будет готов для использования другими разработчиками в таблицах и серверном программном коде (например, в таблицах и представлениях T-SQL или в процедурах, функциях, триггерах и консолидациях CLR или T-SQL), его сборка может быть развернута в локальной папке каждой клиентской утилиты, в которой будет использована для разработки. Когда пользовательский тип готов к применению в среде разработки, он может быть развернут в глобальном кэше сборки или в рабочем пространстве всех клиентских приложений, которые будут ссылаться на новый тип.

Способность одновременно использовать разные версии одного типа может быть благом, а может привести к пустой трате усилий. Правильное планирование и управление является критичным моментом тестирования с помощью подходящих клиентских библиотек. В базе данных легче поддерживать порядок, чем в клиентском пространстве. Вполне вероятна ситуация, когда четыре разработчика используют для тестирования Management Studio с установленными ссылками на четыре разные версии типа, в то время как в базе данных существует только одна версия. Порой все, что нужно для развития такого сценария, — это скопировать файл `.dll` для данного типа в папку выполнения приложений .NET в разное время.

Например, по умолчанию утилита Management Studio будет искать файлы `.dll` в каталоге:

```
c:\Program Files\Microsoft SQL Server\90\Tools\Binn\VSShell\Common7\IDE
```

Помещение файла `.dll` в этот каталог обеспечит возможность установки ссылок на данный пользовательский тип в запросах, выполняемых в среде утилиты Management Studio. Неуправляемые подключения OLE DB, такие как утилита `SQLDMD`, а также подключения ODBC, такие как утилита `osdl`, распознают экземпляры пользовательского типа только как двоичные данные.

В общем случае для использования пользовательского типа данных необходимо наличие разрешения SQL Server `REFERENCES`. Если пользовательский тип используется в процедуре, функции или триггере T-SQL или интеграции CLR в качестве локальной переменной, также необходимо наличие разрешения `EXECUTE`.

Строго именованные сборки и глобальный кэш сборки

Приложения, которые ссылаются на конкретную сборку, будут стремиться использовать ее, точно идентифицируя по имени, версии и подписи. Если сборка не использует строгое имя, приложение может не волноваться о ее версии. Это усложняет процесс развертывания сборок, так как велика вероятность потери обновления в некоторых местах архитектуры, ори-

ентированной на службы. В результате этого сборка, существующая в базе данных, не будет приведена в соответствие со сборками, на которые ссылаются службы узлов. Пока ссылки из приложения .NET на общедоступную структуру и имя соответствуют имени и структуре типа в базе данных, приложение может оперировать типом так, будто оно имеет полные и точные знания о типе, содержащиеся в базе данных. Это так похоже на ад динамических библиотек `.dll`, который была призвана преодолеть среда .NET Framework 2.0.

В то время как реверсная инженерия и дизассемблирование всегда несли с собой потенциальный риск во всех языках программирования и на всех платформах, среда .NET Framework предлагает надежную утилиту дизассемблирования кода MSIL. К счастью, модель системы безопасности среды .NET Framework не предполагает, что сокрытие кода от внешних представлений будет необходимой ее составляющей. Подпись и аутентификация являются ключевыми компонентами защиты сборок .NET. Сборки могут быть подписаны сильно защищенным ключом, предотвращающим возможность введения в нее компонентов, которые могут подвергнуть риску все приложение. Такие сборки могут ссылаться только на те сборки (не считая, естественно, пространства имен `System`), которые подписаны той же парой ключей. Подписи необходимы для всех сборок, помещаемых в GAC.

Компания Microsoft рекомендует при совместном использовании некоторой сборки несколькими приложениями на одном компьютере помещать эти сборки в GAC. В этом случае гораздо легче дать гарантию того, что сборка закрыта в кэше от всех, кроме администратора. Из кэша GAC также можно параллельно запустить несколько версий одной и той же сборки. И наоборот, если хотя бы одна сборка в приложении помещена в GAC, то развертывание приложения усложняется необходимостью использования Windows Installer 2.0 или утилиты Global Assembly Cache Tool для перемещения сборки в кэш GAC всех узлов приложений, которые будут использовать данный пользовательский тип. Сборки, развернутые вместе с приложением в папке приложения, могут быть перемещены в GAC с использованием основанного на команде `xcopy` продвижения и тактики откачки. К тому же, если сборки не развернуты в глобальном кэше GAC, разработчику придется самому заботиться о том, какие сборки принадлежат его приложению. При принятии решения относительно того, где развертывать сборки: в глобальном кэше GAC или в папке приложения, следует основательно изучить требования организации к системе безопасности.

Создание строго именованных сборок .NET

При работе с пользовательскими типами, которые должны быть развернуты в GAC, исключительно важно четко понимать порядок создания строго именованных сборок. Существуют три аспекта строгого именования: идентификатор сборки, открытый и секретный ключи. Секретный ключ еще иногда называют *цифровой подписью*. Все аспекты строгого именования управляются с использованием Visual Studio 2005.

Идентификатор сборки можно увидеть и изменить во вкладке Application окна Project Designer (рис. 29.1), которое открывается с помощью команды меню Project⇨*<имя проекта>* properties или аналогичной команды контекстного меню (открываемого щелчком правой кнопкой мыши на имени проекта в Solution Explorer).

Пара ключей, необходимая для завершения формирования строгого имени, может быть сгенерирована с помощью утилиты Strong Name Tool (`sn.exe`). Доступ к этой утилите открыт в окне командной строки программы Visual Studio. Эта утилита имеет массу функций. Как и во всех утилитах командной строки, полную справку по ней можно получить, набрав в строке запроса **`sn /?`**.

На рис. 29.2 показаны команды создания пары открытого и секретного ключей и развертывания ключа в папке решения проекта.

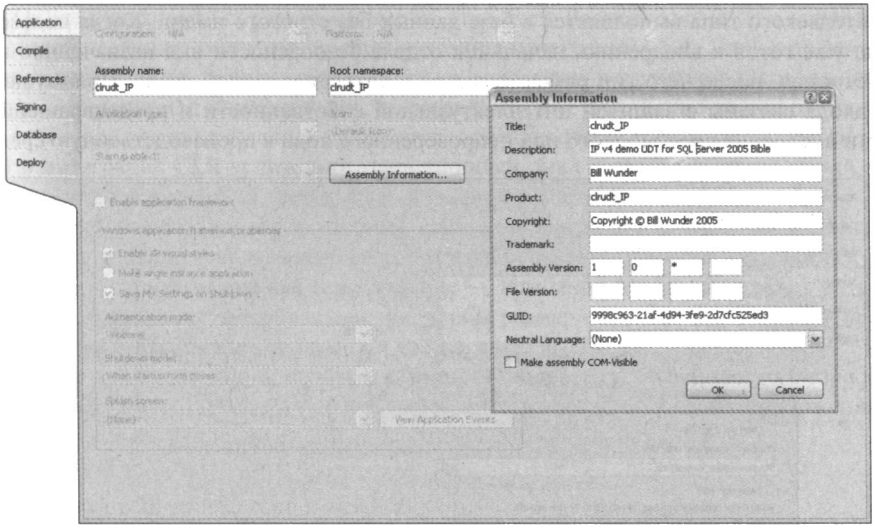


Рис. 29.1. Идентификатор сборки. В его формировании участвует информация о названии и версии, а также региональные настройки (такие, как язык)

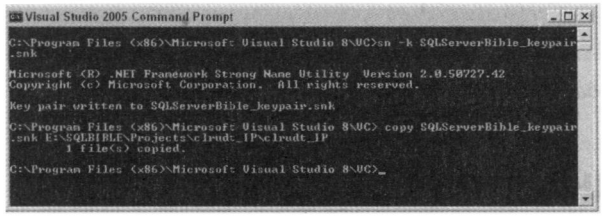


Рис. 29.2. Создание пары открытого и секретного ключей, используемых для создания строгого имени сборки. Данный пример создан в 64-разрядной операционной системе, однако та же процедура используется и в 32-разрядных системах

После развертывания ключа в папке проекта включить его в сборку можно в окне Project Designer (рис. 29.3).

Информация о строгом имени сборки будет включена в ее манифест. Для добавления ключа в сборку можно использовать специальную утилиту командной строки Assembly Linker (`al.exe`). Эта утилита выполняет ту же работу, что и вкладка Signing окна Project Designer, однако она требует передачи в качестве аргумента имени файла `.dll` сборки. Окно Project Designer, с точки зрения пользователя, является более дружественным. Работа с файлом `.dll` предполагает, что пользователь не может использовать функцию Deploy программы Visual Studio в процессе разработки.

Для определения файла ключа можно также использовать и внутренние атрибуты сборки, например:

```
<Assembly:AssemblyKeyFileAttribute("SQLServerBible_keypair.snk")>
```

Политика безопасности организации может потребовать использования и более сложной реализации строгого имени. Например, может быть принято решение не отдавать цифровую подпись (т.е. секретный ключ) разработчикам. В такой ситуации разработка и тестирование

пользовательского типа выполняется в базе данных без строгого имени. Когда пользовательский тип уже готов к внедрению, начальник отдела безопасности или назначенное им лицо добавляет ключ, после чего тип развертывается в интегрированной среде. Преимущества такого подхода связаны с защитой интеллектуальной собственности и предотвращением возможности внедрения вредоносного или непроверенного кода в производственную среду.

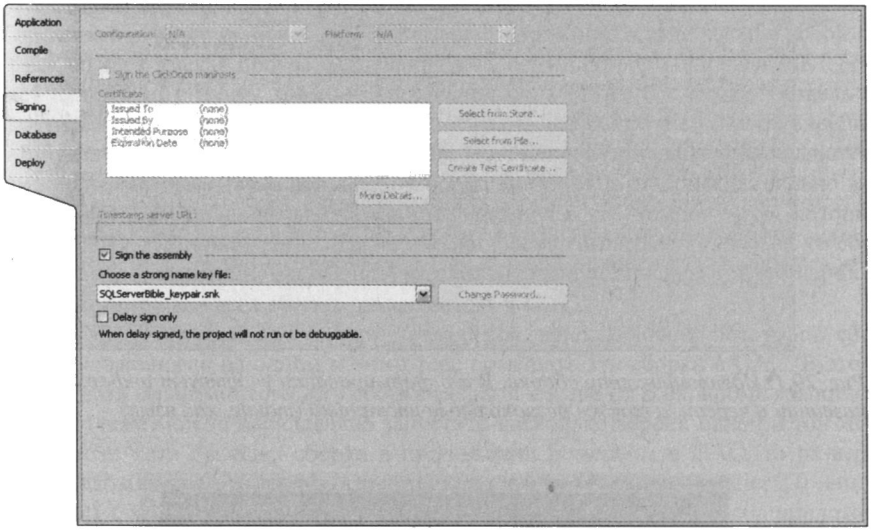


Рис. 29.3. Добавление ключа в сборку в окне ***Project Designer***

Обслуживание пользовательских типов

Будьте готовы к беспрецедентному уровню детализации при внесении изменений в пользовательские типы, которые уже проложили себе дорогу в производственные системы. Изменение определения структуры пользовательского типа требует его удаления и последующего воссоздания. Сохранение существующих данных и обеспечение их поддержки новой структурой также требуют особого внимания и тестирования. Идентификация фрагментов программного кода и узлов, которые следует обновить одновременно при внесении изменений в пользовательские типы, является примером кооперации и взаимодействия, в отличие от процессов, используемых в традиционной среде проектирования баз данных.

Резюме

В этой главе было показано, насколько разительны отличия между пользовательскими типами интеграции CLR и уже давно знакомыми псевдонимами типов данных. В утилите Management Studio они даже разнесены в отдельные папки в представлении базы данных. Старые пользовательские типы пока остаются в арсенале доступных средств, однако их дни сочтены. В этой главе было продемонстрировано, что пользовательские типы .NET предлагают намного более богатый, всеобъемлющий механизм инкапсуляции сложных данных в столбцы и переменные базы данных.

Несомненно, компания Microsoft встретит большое сопротивление пользователей, когда в следующих версиях удалит старые пользовательские типы из SQL Server, несмотря на то, что

в Books Online существует предупреждение на этот счет. Равновероятно, что сопротивление пользователей помешает быстрому внедрению типов интеграции CLR. Любая новая технология вначале должна быть ассимилирована. Также существует потребность в совершенно ином подходе к контролю над изменениями, по мере того, как мышление разработчиков будет смещаться в сторону изолированных архитектур, ориентированных на службы. Изолированные системы и интеграция CLR усложнили традиционный цикл разработки приложений, и в нем сообщество пользователей SQL Server еще не накопило достаточного опыта.

В некоторый момент появятся удобные стратегии миграции и внесения изменений. По мере эволюции архитекторы данных и разработчики осознают значение малых объектов в базах данных. И — примите это как предсказание — компания Microsoft ответит на давление пользователей и расширит возможности так, чтобы размер объектов базы данных преодолел роковой барьер в 8 Кбайт. Вам не кажется, что когда-то вы это уже слышали?

Возможно, практичность и функциональность подвигнут большинство разработчиков разморозить потенциал пользовательских типов интеграции CLR в своих приложениях и начать неотвратимое наступление на мир объектно-реляционных баз данных.

ГЛАВА

30

Программирование в ADO.NET 2.0

В этой главе...

Основы ADO и ADO.NET 2.0

Средства Visual Studio 2005 для ADO.NET 2.0

Создание приложений, использующих ADO.NET 2.0

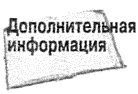
Если данные не перемещаются в базу данных и из нее, то нет никакой необходимости в самой базе данных, ее администраторе и конструкторе. В этой главе мы сместим свое внимание на уровень приложения для исследования одной из самых важных технологий доступа к данным SQL Server — семейству ActiveX Data Objects (ADO). Самая новая версия — ADO.NET 2.0 — это пакет управляемых средств взаимодействия с множеством различных систем управления базами данных (СУБД). Естественно, семейство технологий Microsoft ADO.NET 2.0 находится в "близких отношениях" с SQL Server 2005, и разумно ожидать, чтобы взаимосвязь между ними была наиболее полной. Как вы вскоре увидите, интерфейс между кодом ADO.NET в приложении и SQL Server оптимизирован.

В этой главе мы рассмотрим как ADO, так и ADO.NET, уделив особое внимание новым концепциям и функциям, введенным в ADO.NET версии 2.0. Первое, что следует усвоить, — это то, что технологии ADO и ADO.NET не являются взаимоисключающими — обе они доступны программисту в Visual Studio 2005. Обе эти технологии нашли свою нишу в арсенале инструментов программиста, поэтому полезно знать, чем они отличаются. В этой главе сравниваются и противопоставляются технологии ADO и ADO.NET как средства создания хороших решений в разработке приложений. В ней также уделено особое внимание различию версий 1.0 и 2.0 продукта ADO.NET, так как внесенные в продукт изменения позволяют получить максимум отдачи не только от SQL Server 2005, но и от более старых версий этого сервера баз данных.



Понимание среды .NET Framework и основ объектно-ориентированного программирования (ООП) является фундаментом использования ADO.NET. Краткое введение и дополнительные ссылки на ресурсы, предназначенные для более глубокого изучения .NET Framework и концепций ООП, можно найти в главе 27.

В этой главе вам будет представлена информация о средствах Visual Studio 2005, предназначенных для создания и отладки решений доступа к данным, использующих ADO и ADO.NET. К сожалению, объем книги не позволяет обсудить многочисленные сходства между Visual Studio и SQL Server Management Studio. В этой главе мы рассмотрим только средства Visual Studio, которые способны помочь программисту в разработке в приложении методов доступа к данным, использующих ADO и ADO.NET.



Читатели, не знакомые с Visual Studio, могут найти материал, посвященный этой программе, в главе 6, а также в части V. Там вы сможете ознакомиться с интерфейсом пользователя, общим для Visual Studio и SQL Server Management Studio.

Обзор ADO.NET

Сохраняя традиции доступа к данным компании Microsoft, пакет ADO.NET 2.0 создан на основе существующих технологий доступа к данным. С выходом в свет SQL Server версии 1.1 в этих традициях проявились некоторые общие направления развития. Каждое следующее поколение продуктов делало подключения приложения к данным проще, чем в предыдущем, при этом расширяя список доступных средств и обеспечивая бо́льшую гибкость. Все эти улучшения всегда вели в сторону основных целей современного программирования — использования распределенной структуры вычислений, многоуровневой и клиент-серверной архитектуры, а также архитектуры, основанной на службах (SOA), в том числе на Web-службах. С каждым новым шагом технология доступа к данным стремилась устранить проблемы, выявленные у своих предшественников, в таких областях, как поддержка ссылочной целостности, производительности и надежности приложений.

В каждой новой версии SQL Server все старые технологии сохраняли свою силу, одновременно дополняясь новыми. Только в очень редких случаях доступ данных претерпевал революционные изменения. SQL Server 2005 в этом отношении не стал исключением. С выходом этой версии интерфейс библиотеки DB отошел в прошлое.



Библиотека DB больше не поддерживается в версии SQL Server 2005, а библиотека MDAC (Microsoft Data Access Components) считается устаревшей. Компания Microsoft больше не предлагает средства разработки приложений, использующих библиотеку DB. Если такое возможно, DB-приложения следует обновить. В то же время существует возможность продолжить использование старых приложений в SQL Server 2005. Для этого необходимо скопировать динамическую библиотеку `ntwdblib.dll` с носителя установки предыдущей версии SQL Server на клиентский компьютер. Этот файл нужно скопировать в каталог приложения, использующего библиотеку DB, или в папку, указанную в строке переменной среды `%PATH%`.

Исходная, или "классическая", библиотека ADO уже устаревает и не поддерживается новым клиентом SQL Native Client. Для ссылок на этот новый интерфейс программирования в документации компания Microsoft использует аббревиатуру SQLNCLI. В то же время пользователи отдают предпочтение более краткой аббревиатуре SNAC.

Библиотека ADO поставляется со старыми библиотеками MDAC, к тому же у нее сохранилась возможность доступа к базам данных SQL Server 2005. Однако новые средства SQL Server 2005 больше не доступны для данной библиотеки, поэтому в новых приложениях лучше ее не использовать. Поэтому спланируйте, как лучше заменить или обновить приложения, использующие библиотеку ADO и лежащую в ее основе технологию COM (Component Object Model). В будущих версиях SQL Server не будет поддерживать ни ADO, ни COM.

В данной главе будут описаны действия, которые следует предпринять для подготовки к отказу SQL Server от поддержки этих технологий. Количество существующих приложений, использующих ADO, настолько велико, что вряд ли можно ожидать в ближайшее время, что эта технология сойдет с арены. Несмотря на то что в настоящий момент настоятельно рекомендуется создавать приложения, использующие модель с архитектурой, ориентированной на службы (SOA), и технологию ADO.NET 2.0, для поддержки и обслуживания существующих приложений необходимо также быть хорошо знакомым и со всеми предшествующими версиями ADO и ADO.NET.

Технология ADO принесла в платформу разработки приложений Windows невиданные ранее скорость и гибкость обращения к данным и манипулирования ими. Начиная с ADO и основанных на технологии COM интерфейсов OLE DB, разработчики получили возможность работы с гетерогенными источниками данных — от документов до баз данных — с помощью единой состоятельной методологии. Технология ADO абстрагировала мощные, но сложные компоненты COM и интерфейсы OLE DB в простую и дружественную объектную модель, позволяющую большому числу программистов и Web-разработчиков создавать качественные приложения.

ADO

Даже сегодня ADO остается технологией доступа к данным, основанной на компонентах COM. Очень важно осознавать, что компоненты Object Linking and Embedding (OLE DB) окружали нас с первых дней существования платформы Microsoft Windows. С тех пор многое изменилось. Самыми заметными событиями были публикация спецификации COM и совсем недавнее перемещение доступа к данным с уровня операционной системы на уровень общеязыковой среды выполнения CLR. В то же время многое осталось неизменным. Несмотря на стремительные изменения в мире баз данных, принятие сообществом разработчиков ранних версий OLE и COM продолжалось достаточно долго. Только после этого можно было достигнуть совершенства. Хотелось бы надеяться, что проникновение в массы технологии .NET будет не столь долгим, и на гребне волны она продержится гораздо дольше. Хотя компания Microsoft заверяет, что двоичный стандарт COM переживет следующие версии системы Windows, велика вероятность того, что он постепенно отойдет в небытие. Поступательное движение в сторону перемещения данных с помощью XML оставляет технологиям COM и ADO только шанс играть роли второго или третьего плана.

Характерным сигналом в этом отношении явилось то, что с выходом версии ADO.NET 2.0 компания Microsoft рекомендует реализовать высокоуровневый доступ к данным в ADO только с помощью первичной interop-сборки (PIA) (`ADODB.dll`). На рис. 30.1 показано, как эта сборка представляет неуправляемые компоненты COM в управляемой среде .NET. Программный код ADO, в сущности, остается неизменным, однако управляться общеязыковой средой выполнения CLR он будет через PIA, а не на уровне операционной системы. Это не диктует необходимость больших изменений в старых реализациях, использующих ADO, хотя некоторые изменения все же придется внести. Это не значит, что современные реализации ADO при их создании в среде .NET будут отличаться от предыдущих. Например, в строке подключения по-прежнему необходимо задавать конкретного поставщика OLE DB; также необходимо знать интерфейсы и требования каждого из них. С другой стороны, вступают в игру конструкторы и уборка мусора, система безопасности и проверка типа среды выполнения.

В этой модели проблемы приложений могут привести к неустойчивости операционной системы. В SQL Server 2005 технология ADO использует сборку `adodb.dll` для поддержки своего программного кода при обеспечении безопасности клиента управляемой среды .NET Framework. Клиент SQL Native Client (SNAC) оптимизирован для доступа к SQL Server, поскольку он непосредственно взаимодействует с сетевыми службами.

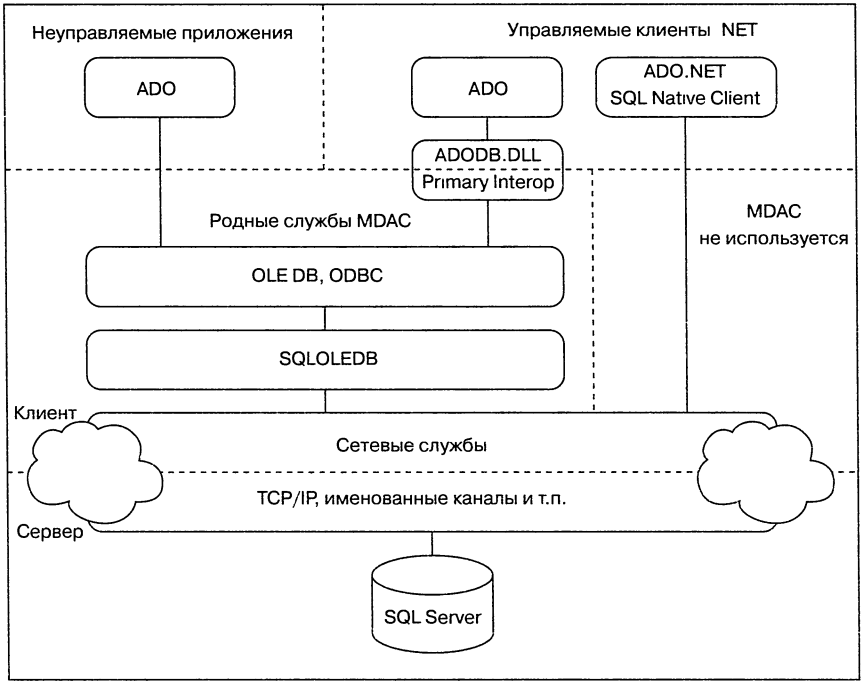


Рис. 30.1. Неуправляемые интерфейсы ADO используют "родные" службы MDAC для доступа к данным

Первоочередной задачей программистов является перевод программного кода на рельсы ADO.NET 2.0. Для облегчения этого перехода используйте предлагаемую первичную interop-сборку. Например, как вариант можно рассмотреть поэтапный подход к обновлению приложений до ADO.NET 2.0. Разумный сценарий предполагает использование в приложениях ADO сборки `adodb.dll`, что требует минимальных изменений в программном коде. При этом приложение получает возможность перемещать структуры из базы данных в наборы данных .NET и потоки XML. Как только структуры оказываются у клиента, открываются возможности для кэширования и управления данными на стороне клиента, равно как и для использования архитектур, ориентированных на службы. Как результат, мы получаем более надежное приложение.



При переводе приложений, использующих ADO, на рельсы .NET Framework необходимо обдуманное планирование. Компания Microsoft в этом отношении помогла разработчикам тем, что опубликовала в Интернете свои рекомендации относительно перехода от ADO к ADO.NET. В июле-августе 2004 года журнал для разработчиков сайта MSDN опубликовал статью Джона Папа "Migrating from ADO to ADO.NET". Эта и многие другие статьи из этого информативного журнала доступны бесплатно на сайте

`http://msdn.microsoft.com/msdnmag/default.aspx`

Первая часть вышеупомянутой статьи находится по адресу

`http://msdn.microsoft.com/msdnmag/issues/04/07/DataPoints/default.aspx`

Вторая — по адресу

`http://msdn.microsoft.com/msdnmag/issues/04/08/DataPoints/default.aspx`

Читателям может также оказаться полезной информация из интерактивной библиотеки MSDN:

```
http://msdn.microsoft.com/library/default.asp
```

В заключение, при планировании перехода от ADO к ADO.NET не забудьте заглянуть в утилиту SQL Server Books Online. В содержащейся в ней статье "Updating an Application to SQL Native Client from MDAC" перечислено множество потенциальных проблем, которых можно избежать, если идентифицировать их на ранней стадии миграции.

С момента выхода в свет ADO для поддержки, обслуживания и расширения существующих приложений, использующих ADO, стало не обязательно быть продвинутым программистом COM или OLE DB. В то же время для поддержки и отладки многих вопросов, связанных с ADO, полезно иметь четкое понимание механизма работы компонентов COM и интерфейса OLE DB. Знание и опыт работы с ADO может пригодиться и в работе с ADO.NET. В конце концов, ADO является базисом, на котором построена ADO.NET; аналогично, OLE DB является основой ADO. Понимание OLE DB поможет вам в работе как с ADO, так и с ADO.NET.

OLE DB

Ключом к пониманию OLE DB являются метафоры *потребителя* и *поставщика*, используемые для описания этой технологии. OLE DB — это метод подключения данных поставщика к приложению, потребляющему эти данные. Для создания некоторых распространенных подключений между разными поставщиками и потребителями должны быть выполнены четко очерченные действия. OLE DB представляет собой компонент COM CoType, или связанную группу интерфейсов COM, которые описаны в раскрываемой иерархии, определяемой свойствами и управляемой событиями. В общем случае OLE DB пытается доставить строки и столбцы данных потребителям. Некоторые поставщики не в полной мере соответствуют данному описанию. Иерархия CoType является своеобразным трапом, по которому различные типы данных могут подниматься из неизвестных источников данных к потребителю, начиная с разных ступенек трапа и заканчивая точкой доставки, известной как `CoCreateInstance`. Каждую ступеньку этого концептуального трапа можно рассматривать как потребителя нижней соседней ступеньки и поставщика верхней. Эти переходы в совокупности называют *интерфейсами*. Все интерфейсы являются наследниками фундаментального интерфейса `IUnknown`. Это демонстрирует полезность и возможность повторного использования компонентов, поскольку все компоненты CoType аналогичны по форме, даже если сильно отличаются функционально. Это легко можно подтвердить, если принять во внимание структурные сходства между разными объектами ADO. На простейшем уровне каждый объект имеет свойства и методы и участвует в иерархии членов.



Читатели, желающие поближе ознакомиться с иерархиями CoType и OLE DB, могут обратиться к статье "Introduction to OLE DB Programing" в библиотеке MSDN:

```
http://msdn.microsoft.com/library/default.asp?url=/library/
en-us/oledb/htm/oledbpart1_introduction_to_ole_db.asp
```

Количество CoTypes в OLE DB крайне велико. Самым мощным и полезным свойством OLE DB является способность выражать и передавать все поставляемые данные потребителю в простейших терминах значения, его длины и отличительных характеристик, таких как состояние. В идеальном случае состоянием будет `DBPROPSTATUS_OK` (т.е. признак хороших данных), в то же время состояние может указывать среди прочего на пустое значение, испорченные данные или по какому критерию данные считаются испорченными.

OLE DB (как, в принципе, и все другие компоненты COM) не зависит от используемых языков программирования. Это делает данный интерфейс идеальной низкоуровневой архи-

тектурой для языков высокого уровня, использующих ADO. Низкоуровневые спецификации способны остаться неизменными, в то время как объектная модель ADO может быть описана в VB, C#, на языках сценариев VBScript и JScript и на языках, способных войти в пространство низкоуровневой спецификации, таких как C++. OLE DB является могучим программным решением, предназначенным для обработки информации из баз данных и прочих источников в наборах записей и потоках. Объектная модель ADO делает интерфейс OLE DB доступным для разработчиков, образуя его оболочку. В то же время ее нельзя назвать окончательным решением доступа к данным.



Если программист неправильно определил область памяти, а компилятор не проверяет безопасность типов, могут возникнуть утечки памяти и ошибки доступа к данным, что нарушит устойчивость базы данных и приложения. *Утечки памяти* возникают, когда приложение выделяет память, но не возвращает ее в пул свободной памяти после того, как потребность в данной области отпала. *Ошибки доступа* возникают, когда приложение пытается прочитать или записать по адресу, который уже используется другим потоком выполнения. Компилятор C++ в Visual Studio2005 не может проверить корректность ссылок на области памяти, поэтому данный язык не защищает программиста от потенциально опасных дефектов в программном коде. Компиляторы VB.NET и, с некоторой натяжкой, компиляторы C#, равно как и Windows Scripting Host, предлагают более совершенную защиту от таких искажений памяти. Утилита `PEVerify`, входящая в состав .NET Framework 2.0 SDK, может быть использована для идентификации безопасного для типов кода MSIL. В то же время эта утилита совершенно бесполезна при поиске местонахождения и типов дефектов защиты типов в исходном программном коде.

Первичные interop-сборки ADODB

COM и .NET являются совместимыми технологиями. Сборки .NET могут использоваться в приложениях COM, а компоненты COM — в программном коде .NET. Сборки .NET могут управляться оболочками COM. Оболочка COM должна реализовать ядро interop-сборки COM. По своей сути interop-сборки являются метаданными (или определениями типов) для компонентов COM, выраженными в управляемом коде.



Дополнительную информацию по вопросу интероперабельности компонентов COM и .NET вы можете найти в статье "Microsoft .NET/COM Migration and Interoperability", размещенной по адресу:

```
http://msdn.microsoft.com/library/default.asp?url=/library/
en-us/dnbda/html/cominterop.asp
```

Первичная interop-сборка (далее PIA) подписывается владельцем объекта COM. PIA является interop-сборкой, рекомендуемой для использования в среде .NET Framework при раскрытии объекта COM в управляемом коде. Компания Microsoft предлагает такие подписанные PIA для ADO в среде разработки Visual Studio .NET — они содержатся в сборке `adodb.dll`. Microsoft рекомендует использовать только эту сборку при использовании ADO в управляемом коде .NET. Как показано на рис. 30.2, все ссылки для первичной interop-сборки выбираются из списка ссылок .NET при добавлении в проект Visual Studio, а не с помощью добавления ссылки на компонент ADO во вкладке COM диалогового окна Add Reference.

Следует сделать еще одно замечание, касающееся использования ADO в Visual Studio 2005. Дело в том, что клиент SNAC не поддерживает объектную модель ADO, а это означает, что для доступа к SQL Server приложения ADO должны полагаться исключительно на компоненты MDAC. ADO.NET поддерживается клиентом SNAC и не требует координации со сложной паутиной библиотек и компонентов MDAC.

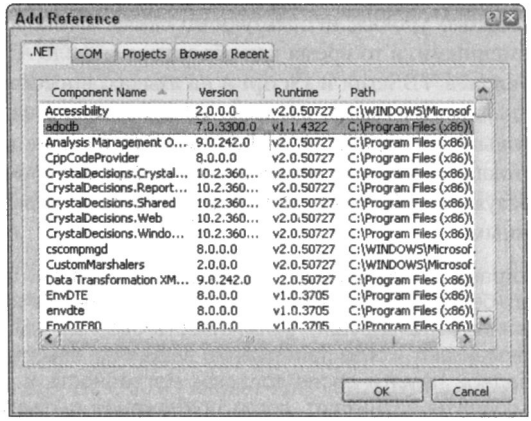


Рис. 30.2. Выбор `adodb.dll` в качестве ссылки .NET в проекте Visual Studio

MDAC становится составной частью различных операционных систем от Microsoft и в дальнейшем не будет выходить в свет отдельно от них. Специально установленная библиотека MDAC не используется ни в одном примере этой главы. Здесь используются драйверы MDAC, поставляемые с операционными системами Windows XP SP2, Windows XP x64, Windows Server 2003 R2 Standard и Windows Server 2003 R2 Enterprise x64. Вряд ли Microsoft будет выпускать в дальнейшем новые версии обособленных библиотек MDAC. Возможно, лучшим местом отслеживания изменений в MDAC станет документация к новым версиям операционных систем, пакетов ообновления и заплаток. Аналогично, совместимость версий MDAC будет проверяться методом тестирования изменений, примененных к операционной системе.

SNAC является программным интерфейсом, совершенно не связанным с MDAC. Этот интерфейс был разработан для упрощения синхронизации обновления клиентов с сервером. Когда клиенты SQL полагаются на компоненты MDAC, обновления этих компонентов, необходимые для совместимости клиента и базы данных, могут вызывать проблемы в других компонентах приложений, запущенных у клиента. Отсюда следует, что изменения MDAC, необходимые для других приложений, могут вызвать проблемы с подключениями к SQL Server. В SNAC клиент SQL содержится всего в одном файле `.dll`. Риск, связанный с изменением SNAC на сервере приложений, является мизерным по сравнению с риском, связанным с изменением MDAC.



С изменениями в MDAC и SNAC также можно ознакомиться в центре разработчиков *Data Access and Storage Developer Center* по адресу:

`http://msdn.microsoft.com/data/default.aspx`

Одним из самых информативных документов, которые можно там найти, является "Data Access Technologies Road Map":

```
http://msdn.microsoft.com/data/mdac/default.aspx?pull=/
library/en-us/dnmdac/html/data_mdacroadmap.asp
```

Объектная модель ADO

До сих пор в этой главе мы говорили о месте ADO в мире .NET и инфраструктуре OLE DB, а также о методах использования ADO в среде .NET Framework. Теперь посмотрим, как ADO вписывается (или более уместно сказать — не вписывается) в общую схему ADO.NET.

Одной из основных задач ADO.NET была реализация всех возможностей, заложенных в старой доброй объектной модели ADO. Как минимум, ADO является ролевой моделью ADO.NET. Версии ADO.NET 1.*x* предстали перед нами с несколько сокращенным набором возможностей ADO, в результате чего реализация средств .NET была ограничена пространством доступа к данным. С выходом версий ADO.NET 2.0 и SQL Server 2005 полный набор средств ADO был объединен с независимостью и безопасностью среды .NET Framework и перспективами XML. Нельзя сказать, чтобы не требовалось никакой работы и чтобы ADO была полностью реализована в своем потомке, особенно в вопросах производительности. Понять произошедшую трансформацию можно только с помощью сравнения этих двух объектных моделей. Для адекватного сравнения рассмотрим функции и компоненты объектной модели ADO. Это заложит фундамент для последующей оценки того, что появилось нового и что было улучшено в ADO.NET.

Объектная модель ADO является не просто оболочкой интерфейса OLE DB. Она представляет реальную ценность для разработчика и имеет некоторые преимущества перед предыдущими методами доступа к данным. Ниже вкратце перечислены эти преимущества.

- **Независимое создание объектов**. В ADO больше нет необходимости последовательно проходить по иерархии объектов. Разработчик создает только необходимый ему объект, таким образом, снижая нагрузку на память, повышая скорость выполнения приложения и уменьшая количество строк программного кода.
- **Пакетное обновление**. Вместо отправки на сервер одного сообщения они могут накапливаться в локальной памяти и отправляться одновременно. Использование этой функции повышает производительность приложения (потому что поставщик данных может выполнять обновления в фоновом режиме) и сокращает нагрузку на сеть.
- **Хранимые процедуры**. Эти процедуры хранятся и выполняются на серверной стороне СУБД. Они используются для выполнения конкретных задач с наборами данных. ADO использует хранимые процедуры с входными и выходными параметрами и возвращает значения.
- **Множество типов курсоров**. Курсоры указывают на строку, с которой выполняется работа в текущий момент. Курсоры могут располагаться как на стороне сервера, так и на стороне клиента.

Дополнительная информация

Очень важно понимать отличия клиентского курсора в приложении и курсора T-SQL. Первые оказывают довольно слабое влияние на производительность. С другой стороны, использование серверных курсоров, особенно обновляемых, может негативно сказаться на базе данных. Подробно о курсорах см. в главе 20.

- **Ограничение числа возвращаемых строк**. Эта функция позволяет ограничить количество возвращаемых запросом строк только теми, которые действительно нужны пользователю.
- **Множество объектов наборов записей**. Возможность работы с множеством наборов данных, возвращаемых хранимой процедурой или пакетом инструкций.
- **Свободные потоковые объекты**. Эта функция повышает производительность Web-сервера, позволяя ему одновременно выполнять несколько задач.

Подобно всем компонентам OLE DB, ADO использует COM. ADO предоставляет двойственный интерфейс: идентификатор программы `ADODB` для локальных операций и идентификатор `ADOR` — для удаленных. Сама библиотека ADO работает со свободными потоками, несмотря на то, что в реестре она показана как использующая модель с разделенными потоками. Безопасность потоков в ADO зависит от используемого поставщика OLE DB. Другими

словами, если используются такие поставщики от Microsoft, как ODBC или OLE DB, не следует ожидать возникновения каких-либо проблем. Если же используется сторонний поставщик OLE DB, может понадобиться ознакомиться с документацией производителя перед тем, как предполагать защищенность потоков в ADO (это требование к использованию ADO в Интернете и корпоративных сетях).



ODBC (аббревиатура от Open Database Connectivity) — одна из технологий подключения приложений к базам данных, которая гораздо старше OLE DB. В отличие от последней, ODBC создавалась для подключения только к источникам данных СУБД. В то же время драйвер ODBC включен в SNAC.

Для работы с ADO используется небольшое множество объектов. В табл. 30.1 перечислены эти объекты и кратко описаны методы их использования. Большая часть из этих типов объектов имеет своих двойников в предыдущих технологиях, представленных компанией Microsoft. Одновременно с этим уровень функциональности объектов ADO значительно выше, чем предлагался более ранними технологиями, и, как будет продемонстрировано ниже, потенциальные возможности, реализуемые более современными технологиями, такими как ADO.NET и XML, превосходят даже ADO.

Таблица 30.1. Обзор объектов ADO

Объект	Описание
`Connection`	Этот объект определяет подключение к поставщику OLE DB. Он используется для выполнения таких задач, как открытие, подтверждение и откат транзакций. Существуют также методы открытия и закрытия подключений и выполнения команд
`Error`	ADO создает объект ошибки как часть объекта подключения. Этот объект содержит дополнительную информацию об ошибке, сгенерированной поставщиком OLE DB. Один объект Error может содержать информацию о нескольких ошибках. Каждый объект ассоциирован с конкретным событием, таким как подтверждение транзакции
`Command`	Объект команды выполняет задачи в объектах набора записей или подключения. Несмотря на то что команды можно выполнить с помощью объектов подключения или набора записей, объект команды более гибкий и допускает выходные параметры определения
`Parameter`	Этот объект определяет один параметр команды. Параметры изменяют результат хранимой процедуры или запроса. С помощью объекта параметра можно организовать ввод, вывод или и то и другое
`RecordSet`	Объект набора записей содержит результат запроса и курсор для выбора отдельных элементов возвращенной таблицы
`Record`	Запись — это одна строка данных. Она может быть обособленной, а может управляться и набором данных
`Field`	Объект поля содержит один столбец данных из записи или набора данных. Другими словами, поле можно рассматривать как отдельный столбец таблицы. Он содержит один тип данных во всех записях, ассоциированных с набором данных
`Stream`	Когда поставщик данных не способен выразить значение и длину данных как набор записей с дискретными полями (как в случае с большими текстовыми полями, особо крупными объектами или данными документов), данные могут быть отправлены потребителю с помощью объекта потока
`Property`	Для некоторых поставщиков OLE DB требуется расширение стандартных объектов ADO. Одним из методов такого расширения является использование объекта свойств. Этот объект содержит информацию об атрибуте, имени, типе и значении

В объектной модели ADO также существуют четыре коллекции объектов: `Errors`, `Parameters`, `Fields` и `Properties`. Обратите внимание на то, что эти коллекции служат своеобразными контейнерами для соответствующих дочерних объектов. Не существует коллекций в корне объектной модели, и модель никогда не имеет больше двух уровней в глубину. Структура объектной модели — последовательная и простая и всегда имеет следующую структуру:

Родительский объект⇨Коллекция зависимых объектов⇨Дочерний объект

Поставщики данных OLE DB

Даже когда ADO используется посредством предоставленной первичной interop-сборки .NET, все операции доступа к данным выполняются с использованием одного из доступных поставщиков данных OLE DB. *Поставщик данных* управляет подключением клиента к СУБД с помощью множества объектов. Естественно, это значит, что поставщик данных требует указания источника информации для определения специфики создаваемого подключения. В общем случае поставщик специфичен для типа базы данных и предоставляет механизм конфигурирования конкретной базы данных. На рис. 30.3 показан типичный список поставщиков баз данных. Некоторые поставщики в этом списке являются узкоспециализированными.

Источник объекта OLE DB называют поставщиком. Таким образом, ADO расценивает поставщика как источник данных. Даже в современном мире господства среды .NET Framework количество поставщиков OLE DB, специализированных под конкретные источники данных, больше, чем когда бы то ни было. Следует особо отметить одну прекрасную особенность, касающуюся OLE DB, — один и тот же поставщик может работать с разными языками программирования Visual Studio.

В общем случае (особенно это касается конструкторов баз данных) всегда лучше использовать поставщика, специфичного для SQL Server. Несмотря на то что поставщики общего назначения также способны справиться с работой, компания Microsoft оптимизировала работу поставщика SQL Server для работы с этой конкретной СУБД, и отличия его производительности по сравнению с поставщиком общего назначения разительны.

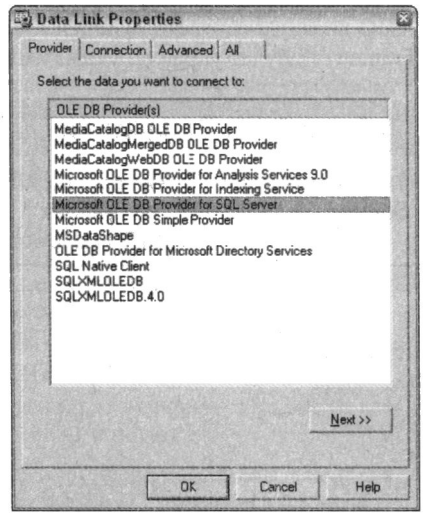


Рис. 30.3. Типичный список поставщиков баз данных



Чтобы узнать, какие поставщики данных доступны на конкретном компьютере, создайте текстовый файл и переименуйте его с использованием расширения `.udl` (например, в `temp.udl`; имя в данном случае не имеет значения, главное — расширение). Откройте файл, и вы увидите диалоговое окно, аналогичное показанному на рис. 30.3. В нем будут перечислены все поставщики OLE DB, локально установленные на данном компьютере.

Отображение типов данных

При работе исключительно в SQL Server основной задачей является выбор оптимального типа данных для конкретных требований к хранению данных. В то же время при перемещении информации из СУБД клиенту с помощью поставщика данных появляется несколько

промежуточных уровней. В некоторых СУБД это представляет существенную проблему, так как поставщики общего назначения, входящие в OLE DB, не поддерживают множество специализированных типов данных. В любом поставщике затраты на преобразование типов данных, необходимые для прохождения всех этих уровней, значительны. Эта проблема типов данных является еще одним поводом использовать в работе с ADO специфичные для SQL Server поставщики данных OLE DB.

При использовании данных из таблиц SQL Server в клиентском приложении поставщик должен отобразить тип данных, который понимает SQL Server, на тип данных, который понимает приложение. К счастью для разработчиков SQL Server, отображение ADO для поставщика SQLOLEDB относительно простое. В табл. 30.2 показано, как SQL Server отображает типы данных ADO. Одна из проблем возникает, когда ADO использует один и тот же свой тип для представления двух или трех типов данных SQL Server, в то время как в интерфейсе пользователя последние должны отображаться с разным форматированием. Полный набор типов данных SQL Server 2005 представлен в клиенте SQL Native Client. Объектная модель ADO знает только типы данных SQL Server 2000 — она не способна работать с новыми типами, такими как XML или `varchar(max)`, о которых мы поговорим немного позже в этой главе.

В табл. 30.2 перечислены типы данных SQL Server 2005 и эквивалентные им типы данных ADO наряду с соответствующей функцией преобразования, с помощью которой среда .NET Framework будет управлять каждым из типов ADO.

Таблица 30.2. Отображение данных SQL Server в ADO/ADO.NET

Тип данных SQL Server	Тип данных ADO (тип данных .NET Framework)	Примечания
`Bigint`	`adBigInt` (`int64`)	Тип `bigint` имеет диапазон значений от -2^63 (-9223372036854775807) до 2^63-1 (9223372036854775807). Эти значения доступны только в SQL Server 2000, однако поставщик OLE DB будет пытаться отправить его и в более старые версии сервера, в результате теряя данные. Используйте тип `adBigInt` только в случае крайней необходимости и с большой осторожностью
`Binary`	`adBinary` (`byte[]`)	ADO использует один и тот же эквивалент типа данных для исходных типов `binary` и `timestamp`
`Bit`	`adBoolean` (`int16`)	Несмотря на то что это преобразование работает всегда, между данными типами существуют определенные различия. Например, тип `bit` может иметь значения `0`, `1` и `NULL`, в то время как `adBoolean` — только `true` и `false`
`Char`	`adChar` (`string`)	ADO использует один и тот же эквивалент типа данных для исходных типов `char`, `varchar` и `text`. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode
`Datetime`	`adDBTimeStamp` (`DateTime`)	По умолчанию точность типа `Datetime` в SQL Server составляет 3,33 миллисекунды
`Decimal`	`adNumeric` (`Decimal`)	ADO использует один и тот же эквивалент типа данных для исходных типов `decimal` и `numeric`

Тип данных SQL Server	Тип данных ADO (тип данных .NET Framework)	Примечания
`Float`	`adDouble` (`Double`)	
`Image`	`adVarbinary` (`byte[]`)	Этот тип данных может быть настолько большим, что может не поместиться в памяти. Утечки памяти могут вызвать ошибки поставщика и, возможно, только частичный возврат данных. Когда такое случается, разработчик должен создавать специальные процедуры для извлечения данных по частям. ADO использует один и тот же эквивалент типа данных для исходных типов `image`, `tinyint` и `varbinary`
`Int`	`adInteger` (`Int32`)	
`Money`	`adCurrency` (`Decimal`)	ADO использует один и тот же эквивалент типа данных для исходных типов `money` и `smalmoney`
`Nchar`	`adWChar` (`string`)	ADO использует один и тот же эквивалент типа данных для исходных типов `nchar`, `ntext`, `nvarchar` и `sysname`. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode (UTF-16)
`Ntext`	`adWChar` (`string`)	Этот тип данных может быть настолько большим, что может не поместиться в памяти. Утечки памяти могут вызвать ошибки поставщика и, возможно, только частичный возврат данных. Когда такое случается, разработчик должен создавать специальные процедуры для извлечения данных по частям. ADO использует один и тот же эквивалент типа данных для исходных типов `nchar`, `ntext`, `nvarchar` и `sysname`. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode (UTF-16)
`Numeric`	`adNumeric` (`decimal`)	ADO использует один и тот же эквивалент типа данных для исходных типов `decimal` и `numeric`
`Nvarchar`	`adWChar` (`string`)	ADO использует один и тот же эквивалент типа данных для исходных типов `nchar`, `ntext`, `nvarchar` и `sysname`. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode (UTF-16)
`NvarChar(max)`	Отсутствует (`string`)	SQL Server 2005 использует один эквивалент типа данных для бывшего типа `Nvarchar`, если объем данных не превышал 8 Кбайт, или `text`, если превышал. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode (UTF-16)
`Real`	`adSingle` (`Single`)	
`Smalldatetime`	`adTimeStamp` (`DateTime`)	
`Smallint`	`adSmallInt` (`Int16`)	

Тип данных SQL Server	Тип данных ADO (тип данных .NET Framework)	Примечания
`Smallmoney`	`adCurrency` (`Decimal`)	ADO использует один и тот же эквивалент типа данных для исходных типов `money` и `smallmoney`
`sql_variant`	`adVariant` (`object`)	Этот тип может содержать данные любого из множества примитивных типов, таких как `smallint`, `float` или `char`. Он не способен хранить большие объемы данных, такие как типы `text`, `ntext` и `image`. Тип `adVariant` отображается на тип `DBTYPE_VARIANT` интерфейса OLE DB и пригоден к использованию только в SQL Server 2000. Будьте осторожны при использовании этого типа данных, поскольку это может привести к непредсказуемым результатам. Несмотря на то что OLE DB полностью поддерживает данный тип, того же нельзя сказать о ADO
`Sysname`	`adWChar` (`string`)	ADO использует один и тот же эквивалент типа данных для исходных типов `nchar`, `ntext`, `nvarchar` и `sysname`. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode (UTF-16)
`Text`	`adChar` (`string`)	Этот тип данных может быть настолько большим, что может не поместиться в памяти. Утечки памяти могут вызвать ошибки поставщика и, возможно, только частичный возврат данных. Когда такое случается, разработчик должен создавать специальные процедуры для извлечения данных по частям. ADO использует один и тот же эквивалент типа данных для исходных типов `char`, `text` и `varchar`. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode (UTF-16)
`Timestamp`	`adBinary` (`byte[]`)	ADO использует один и тот же эквивалент типа данных для исходных типов `binary` и `timestamp`
`Tinyint`	`adTinyInt` (`byte`)	
`Uniqueidentifier`	`adGUID` (`Guid`)	Этот тип поддерживает строки глобальных уникальных идентификаторов GUID, но не сами идентификаторы. Это значит, что когда нужен фактический идентификатор GUID, программный код должен явно преобразовать этот тип данных в структуру GUID
`Varbinary`	`adVarbinary` (`byte[]`)	ADO использует один и тот же эквивалент типа данных для исходных типов `image` и `varbinary`
`Varbinary(MAX)`	Отсутствует (`byte[]`)	SQL Server 2005 использует один эквивалент типа данных для бывшего типа `varbinary`, если объем данных не превышал 8 Кбайт, или `image`, если превышал
`Varchar`	`adChar` (`string`)	ADO использует один и тот же эквивалент типа данных для исходных типов `char`, `varchar` и `text`. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode

Тип данных SQL Server	Тип данных ADO (тип данных .NET Framework)	Примечания
`Varchar(MAX)`	Отсутствует (`string`)	SQL Server 2005 использует один эквивалент типа данных для бывшего типа `varchar`, если объем данных не превышал 8 Кбайт, или `text`, если превышал. Среда .NET Framework для представления всех символьных данных использует таблицу Unicode

Примечания в табл. 30.2 затрагивают только самые существенные проблемы, которые могут возникнуть при отображении данных между ADO и SQL Server. Очень важно также принимать во внимание ошибки преобразования данных. Все непрямые преобразования данных могут стать причиной потери данных. Например, ни поставщики, ни SQL Server не сгенерируют ошибку, если 8-байтовое число преобразуется в 4-байтовое; в то же время при этом совершенно очевидна вероятность потери данных. К тому же некоторые типы не могут быть преобразованы в другие. Например, невозможно преобразовать тип данных `adBinary` в `adSmallInt`. В этой ситуации среда разработки укажет на ошибку. Следует также отметить, что порядок сортировки символьных типов данных SQL, принятый по умолчанию, отличается от порядка сортировки типов данных .NET.

Среда .NET Framework добавляет еще один уровень потенциальных ошибок преобразования в реализацию ADO. Типы данных модели ADO определены в ее объектах. Среда .NET Framework преобразует эти типы данных в допустимые типы данных .NET неявно, если они используются в программе вне объектов ADO. Такое преобразование не требует явного определения, хотя на практике всегда рекомендуется иметь гарантию того, что данные всегда будут иметь ожидаемый тип. При программировании с использованием ADO, даже в среде .NET Framework, разработчик должен неустанно следить за преобразованием данных. Использование типов данных .NET, поддерживающих пустые значения, во многих приложениях открывают новые горизонты для проблем преобразования. Похоже, не существует идеального решения этой проблемы; остается только следовать стандартам среды, которые определяют связные методики программирования, и тщательно тестировать приложения.


Проверено

Чтобы избежать неожиданных преобразований в операциях присвоения, следует явно назначать нужные типы данных ADO.

ADO и сценарии

Объектная модель ADO часто используется в сценариях различных типов. Так как ADO основана на технологии COM, любой язык сценариев, способный создать объект, может использовать ADO для извлечения информации из базы данных. Использование сценариев для выполнения небольших задач имеет смысл, поскольку при необходимости они могут быть легко модифицированы, к тому же их легко создавать. Имейте в виду, что некоторые инструменты сценариев, такие как Windows Script Components, могут быть созданы с помощью VB.NET и C# и на них может быть установлена ссылка. К тому же Windows Scripting Host запускает ADO только как компонент COM. Интероперабельность в .NET не применяется.

Как вы помните из главы 27, исходный программный код .NET компилируется в код MSIL. Так как компиляция в MSIL может выполняться в любой момент перед запуском про-

граммы, концепция *позднего связывания* недоступна в программном коде .NET. В противоположность этому реализации ADO, основанные на сценариях, такие как Windows Scripting Host 5.6, SQL Server 2000 DTS ActiveX и ASP-страницы, предшествующие появлению платформы .NET, требуют позднее связывание. Позднее связывание предполагает, что объект COM, на который имеется ссылка, создается во время выполнения и инициализируется с помощью функции `CreateObject()`. Таким образом, для использования в сценариях ADO на всех компьютерах, на которых будет выполняться сценарий, должен быть установлена MDAC. В противоположность этому ADO.NET полностью поддерживается клиентом SQL Native Client и не требует установки MDAC. Это важная отличительная черта ADO.NET 2.0 более подробно будет рассмотрена немного позже в этой главе.

Естественно, языки сценариев не обеспечивают надежную интерактивную среду, как языки программирования типа C# или Visual Basic и даже ASP.NET. Лучше ограничить использование сценариев только малыми задачами, такими как вызов хранимой процедуры для автоматического выполнения некоторого задания или извлечение результатов запроса к данным для их отображения на экране.

Компании Microsoft следовало бы продемонстрировать гибкость объектной модели ADO при ее использовании с различными языками (такими как Java, JavaScript, VBScript и основанным на XML Windows Script Component), доступными в Visual Studio 2005.

ADO.NET

Несколько странно, что в первой части названия ADO.NET используется название модели ADO (аббревиатура от ActiveX Data Objects). Термин *ActiveX* служит явным сигналом, что используется прототип `IUnknown` и технология COM. `IUnknown` — это прототип всех классов COM. В своей оригинальной форме он указывает на то, что вызывающему модулю ничего не нужно знать о вызываемом объекте. При этом вызывающий модуль сохраняет способность активно взаимодействовать с вызываемым объектом. Несмотря на то что в недрах среды .NET продолжает существовать возможность программирования компонентов COM, в целом платформа .NET Framework и общеязыковая среда выполнения CLR отошли от ограничений среды выполнения COM в направлении согласованности объектно-ориентированной модели классов. Использование прототипа `IUnknown` подразумевает, что о классе не обязательно что-либо знать, чтобы создавать его экземпляры и использовать их. Среда .NET Framework предполагает, что системные сборки и базовые классы являются надежными и согласованными типами. Короче говоря, с технической точки зрения ADO и ADO.NET мало что связывает.

Только с функциональной точки зрения ADO.NET является продуктом естественной эволюции объектной модели ADO. Для преодоления ограничений производительности и масштабируемости модели ADO нужна была новая технология — та, которая обеспечит разработчика массой разнообразных вариантов выполнения операций. ADO.NET разрабатывалась одновременно как для преодоления ограничений ADO, так и для поднятия процесса разработки программ на новый уровень. В ADO.NET можно найти базовые объекты ADO. Команды, выполняемые в подключениях к поставщикам данных, только слегка отличаются в ADO.NET от тех же команд ADO. В то же время в ADO.NET разработчики имеют больший контроль над извлечением и манипуляциями данными. Выполнение может быть асинхронным и пакетным. Результаты могут храниться и обрабатываться в приложениях, отключенных от базы данных. В качестве альтернативы результаты могут быть преобразованы в поток двоичных данных или XML.

Многие разработчики страдают от того, что ошибочно считают ADO.NET всего лишь очередным обновлением объектной модели ADO. Модель ADO создавалась для поддержки приложений, использующих архитектуру "клиент/сервер" и предполагающих наличие постоянного соединения между пользователем и источником данных на протяжении всего цикла выполнения программы. Рискуя слишком упростить различия между этими моделями, отме-

чу, что ADO хранит состояние в источнике данных, а ADO.NET создавалась для возможности поддержки состояния при отсутствии подключения к базе данных. Одним из главных достоинств среды .NET является отсутствие необходимости наличия постоянного подключения приложения к базе данных на протяжении всего цикла выполнения. В некоторой степени эта конструктивная цель была достигнута с помощью технологии XML, которая легла в основу ADO.NET. В большей степени управление состоянием выполняется на уровне приложения ADO.NET в кэше локального приложения, известном как класс `DataSet`. В ADO.NET 2.0 управление состоянием было расширено с помощью асинхронного выполнения команд и использования *множественных активных результирующих наборов данных* (MARS).

В следующих разделах будут описаны объекты ADO.NET как результат естественной эволюции объектов ADO, а также отмечены новые функции, появившиеся в версии ADO.NET 2.0. Другими словами, при рассмотрении концепций ADO.NET дискуссия будет построена на основе информации об ADO, представленной ранее в этой главе. ADO.NET представляет собой управляемую объектную модель с функциональными возможностями, приближенными к классической модели ADO, но позволяющими создавать более масштабируемые приложения, в частности, в изолированной среде, которую можно встретить во многих приложениях, ориентированных на службы и Web-службы и использующих многоуровневую архитектуру и ASP.NET.

Объектная модель ADO.NET

Объектная модель ADO.NET отличается от той, которая использовалась в ADO. Эта модель более сложная, хотя, несомненно, и более мощная.

Наиболее разительным отличием явилось введение класса кэша данных в памяти, называемого `DataSet`. Этот объект ADO.NET можно разделить на два компонента: собственно набор данных и поставщик данных. `DataSet` является специальным объектом, содержащим одну или несколько таблиц. Данные в объект `DataSet` извлекаются из источника данных с помощью поставщика и сохраняются в рабочей области приложения. Этими данными можно манипулировать непосредственно в приложении. `DataSet` является подмножеством источника данных, определенного в свойствах поставщика. Среди свойств поставщика следующие: `Connection`, `Command`, `DataReader` и `DataAdapter`. Каждый из этих объектов также имеет возможности, не существовавшие в объекте поставщика модели ADO. Например, объект `DataAdapter` может обслуживать больше одного подключения и набора правил. Как и во многих других управляемых объектах, для доступа из кода программы к различным объектам, содержащимся в основных объектах, используются перечисления.

- **Поставщик данных**. Содержит классы, создающие подключение, выполняющие команды, обслуживающие объект чтения данных и обеспечивающие поддержку адаптера данных. Объект `Connection` поддерживает канал к базе данных. Объект `Command` позволяет клиенту запрашивать информацию из базы через адаптер данных. В дополнение к обеспечению доступа к данным объект адаптера позволяет клиентскому кэшу синхронизироваться с источником данных и обновлять его. Объект `DataReader` является односторонним (т.е. обеспечивающим только чтение данных) каналом связи с базой данных в модели ADO. В то же время объект `DataAdapter` обеспечивает поддержку соединения в реальном времени, т.е. двусторонний обмен данными.
- **Набор данных**. Предоставляет информацию, содержащуюся в базе данных. Этот объект состоит из двух коллекций: `DataTableCollection` и `DataRelationCollection`. `DataTableCollection` содержит столбцы и строки таблицы наряду с ограничениями, применяемыми к этой информации. `DataRelationCollection` содержит информацию об отношениях, использованных для создания набора данных.

В табл. 30.3 представлен обзор наиболее часто используемых классов данных ADO.NET.

Таблица 30.3. Обзор классов ADO.NET 2.0

Тип класса	Описание
`Connection`	Создает физическое соединение между СУБД и объектом `DataAdapter`, `DataReader` или `factory`
`ProviderFactory`	Новый класс ADO.NET 2.0. Каждый поставщик .NET реализует класс `ProviderFactory`, который управляется из общего базового класса `DBProviderFactory`. Класс `factory` содержит методы создания специфичных для поставщика компонентов ADO.NET. Идея, заложенная в этом классе, позволяет разработчику написать обобщенный программный код, который использует поставщика, определяемого в ходе выполнения программы. Возможные поставщики, которые могут использоваться, хранятся в файле `machine.config`
`Command`	Определяет действие, выполняемое в СУБД, например вставку, обновление или удаление строки. Класс `DataAdapter` содержит объекты команд, необходимых для выполнения запросов к базе данных, а также удаления, вставки и редактирования ее записей
`Parameter`	Параметр команды
`Error`	Информация об ошибке или предупреждении, возвращенном из базы данных. В SQL Server эта информация содержит номер и текст ошибки, а также ее частоту
`Exception`	Исключение приложения, генерируемое, когда ADO.NET 2.0 встречает ошибку. Класс `Error` создается классом `Exception`. Класс `Exception` используется в ADO.NET 2.0 для обработки ошибок в выражениях `try...catch`
`DataAdapter`	Преобразует данные из источника данных поставщика в находящийся в памяти объект `DataSet` или `DataReader`. Класс `DataAdapter` выполняет все запросы, преобразования данных из одного формата в другой, а также отображение таблиц. Один объект `DataAdapter` может поддерживать одно отношение базы данных. В результате коллекция может иметь любой уровень сложности. `DataAdapter` также отвечает за отправку запросов на новые подключения и закрытие подключений после получения данных
`DataReader`	Обслуживает соединение с базой данных в реальном времени. Однако этот объект обеспечивает работу только механизма чтения данных. К тому же курсор DataReader работает только в прямом направлении. Этот объект обычно используют для быстрого извлечения данных из локальной таблицы, когда не существует потребности в обновлении базы данных. Объект `DataReader` блокирует последующие объекты `DataAdapter` и ассоциированные с ними объекты `Connection`. Исходя из этого, не забывайте закрывать объект `DataReader`, как только потребность в нем отпала
`DataSet`	Содержит локальную копию данных, извлеченных одним или несколькими объектами `DataAdapter`. Объект `DataSet` использует локальную копию данных, так что подключение к базе данных не обслуживается в реальном времени. Пользователь может внести все изменения в локальную копию, а затем приложение затребует обновление. (Обновления могут выполняться в пакетном режиме или по одной записи за проход.) Объект `DataSet` поддерживает информацию как об исходном, так и о текущем состоянии каждой модифицированной строки. Если исходная строка данных соответствует находящейся в базе данных, то объект `DataAdapter` выполняет затребованное обновление; в противном случае `DataAdapter` возвращает ошибку, которую приложение должно обработать. Наборы данных `DataSet` в ADO.NET 2.0 могут быть типизированными и нетипизированными. Объекты `DataSet` определяются в пространстве имен `System.Data`, они не специфичны для конкретного поставщика. С поставщиком ассоциированы только классы `DataAdapter`.

Тип класса	Описание
`Transaction`	Это новый класс ADO.NET 2.0. Объект транзакции ADO.NET по умолчанию является обычным контейнером, работающим с одним источником данных. Если код ADO затребует в транзакции другой источник данных, транзакция будет незаметно преобразована в распределенную или многоэтапную, не требуя от разработчика дополнительного программирования



Visual Studio 2005 предлагает разработчику класс `TableAdapter` — набор данных одной таблицы, который можно использовать в приложении, подобно объекту `DataSet` ADO.NET 2.0. Если работа выполняется с одной таблицей, то класс `TableAdapter` существенно легче программировать, чем используемые им компоненты ADO.NET 2.0. Класс `TableAdapter` не управляется из класса `System.Data.DataSet`, как все остальные наборы данных. Более того, класс `TableAdapter` даже не является составным компонентом среды .NET Framework. Он находится на уровне абстракции, поддерживаемом программой Visual Studio 2005. Все объекты `TableAdapter` наследуют из класса `System.Component.ComponentModel`. Это значит, что данные объекты полностью интегрированы с инструментами Visual Studio, такими как Data Grid. К тому же базовым классом для `TableAdapter` может служить любой из поставщиков .NET, оснащенный классом `DataTable`. Базовый класс задается при создании объекта `TableAdapter`. Класс `TableAdapter` защищает типы и содержит все свойства и методы, необходимые для подключения к источнику данных, извлечения информации из таблицы и обновления источника данных. Более подробную информацию о классе `TableAdapter` вы сможете найти на сайте MSDN в статье "TableAdapters in Visual Studio 2005":

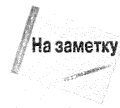
```
http://msdn.microsoft.com/library/default.asp?url=/library/
en-us/dnvs05/html/tableadapters.asp
```

Управляемые поставщики

В ADO.NET 2.0 содержится четыре управляемых поставщика.

- **OracleClient**. Поставщик от Microsoft для баз данных Oracle. Этот поставщик требует установки на компьютере клиента СУБД Oracle.
- **OleDb**. Связующий поставщик для использования поставщиков OLE DB в ADO.NET.
- **SqlClient**. Поставщик от Microsoft для SQL Servber 7.0 и более поздних версий. Точно так же как поставщик OLE DB напрямую связывает SQL Server и ADO, SQLClient использует закрытый протокол для прямого подключения к SQL Server из ADO.NET.
- **SqlServerCe**. Поставщик от Microsoft для СУБД SQL Server CE Mobile Edition.

Как уже говорилось ранее, поставщик `OracleClient` требует наличия установленного клиента СУБД Oracle. Поставщик `OleDB.NET` для обеспечения тех же функций полагается на компоненты MDAC. `SqlClient` и `SqlServerCe` содержатся в библиотеке SQL Native Client.



ADO.NET 1.1 содержит управляемый поставщик `Odbc`. Пространство имен для этого поставщика в ADO.NET 2.0 больше не поддерживается. В то же время SQL Native Client содержит драйвер ODBC, что обеспечивает поддержку существующих приложений, использующих ODBC. Однако удаление пространства имен `System.Data.ODBC` из ADO.NET 2.0 является сигналом завершения поддержки программного интерфейса ODBC API для доступа к данным.

Несмотря на то что объектные модели ADO.NET 1.*x* использовали общую архитектуру MDAC, в них не существовало единого объекта, обеспечивающего выполнение команд, чтение данных и поддерживающего адаптер данных. В них существовали разные объекты, выполняющие данные задачи, специфичные для классов поставщиков, которые находились в разных библиотеках. Исходя из этого, разработчику приходилось выбирать пространство имен, подходящее для конкретного приложения. Выбранное пространство имен должно было быть согласовано с конкретным поставщиком.

В контексте работы с SQL Server это подразумевало использование объектов, которые компания Microsoft оптимизировала для работы с SQL Server, в том числе `SqlConnection`, `SqlCommand`. `SqlDataReader` и `SqlDataAdapter`.

Эти специфичные для поставщика классы продолжают поддерживаться. Когда источником данных является SQL Server, естественно, предпочтительнее использовать классы, оптимизированные для данного поставщика. В этом случае можно ожидать лучшей производительности. К тому же в ADO.NET существует альтернативный общий базовый класс, который можно использовать как вариант в новых проектах. В среде, которая должна поддерживать множество источников данных, использование общего класса может потребовать меньшего объема программирования, чем дублирование одной и той же логики для каждого из поставщиков. В то же время высока вероятность того, что программирование с использованием общего базового класса потребует дополнительной настройки под конкретного поставщика.

В табл. 30.4 представлены перекрестные ссылки классов `System.Data.Common` и классов, специфичных для поставщиков классов, доступных для каждого из типов, перечисленных в табл. 30.2.



При работе в ADO.NET 1.1 с источниками данных других СУБД разработчику необходимо использовать отдельное, но параллельное множество классов поставщиков ADO.NET для каждого из поставщиков. При работе со средой .NET Framework 1.1 это относится к классам пространств имен `System.Data.OleDb`, `System.Data.Odbc` и `System.Data.OracleClient`. Разработчики также могут создавать и дополнительных поставщиков, а сторонние источники данных могут сделать доступными пространства имен своих поставщиков. Те же интерфейсы, которые были использованы компанией Microsoft для создания включаемых классов поставщиков, могут быть использованы и для создания дополнительных поставщиков.

Таблица 30.4. Ссылки на классы из пространств имен ADO.NET 2.0

Тип класса (из табл. 30.3)	System. Data.Common	System.Data. SQLClient	System.Data. OracleClient	System.Data. OleDb
`Connection`	`DbConnection`	`SqlConnection`	`OracleConnection`	`OleConnection`
`ProviderFactory`	`DbProviderFactory`	`SqlClientFactory`	`OracleClientFactory`	`OleDbFactory`
`Command`	`DbCommand`	`DqlCommand`	`OracleCommand`	`OleCommand`
`Parameter`	`DbParameter`	`SqlParameter`	`OracleParameter`	`OleParameter`
`Error`	Отсутствует	`SqlError`	Отсутствует	`OleError`
`Exception`	`DbException`	`SqlException`	`OracleException`	`OleException`
`DataAdapter`	`DbDataAdapter`	`SqlDataAdapter`	`OracleDataAdapter`	`OleDataAdapter`
`DataReader`	`DbDataReader`	`SqlDataReader`	`OracleDataReader`	`OleDataReader`
`Transaction`	`DbTransaction`	`SqlTransaction`	`OracleTransaction`	`OleTransaction`

В ADO.NET 2.0 пространство имен `System.Data.Common` представило базовый класс для создания независимого от поставщика программного кода (часто его называют *обобщенным*), использующего общий базовый класс, совместно используемый всеми поставщиками. В этой модели поставщик определяется, и к нему осуществляется доступ с помощью класса `DbProviderFactory`. Компонентами данного поставщика являются `DbConnection`, `DbCommand`, `DbDataReader` и `DbDataAdapter`. Независимая от поставщика модель позволяет определить в файле конфигурации приложения, реестре или другой структуре не только строку подключения, но и самого поставщика, а также принимать эти данные от самого пользователя в момент инициализации класса подключения. Данная модель получила название *фабричной* благодаря своей способности автоматически создавать экземпляры классов, специфичных для конкретных поставщиков. Создаваемые таким образом классы являются наследниками классов соответствующих пространств имен (`SqlClient`, `OracleClient` или `OleClient`). С точки зрения программиста, результатом такого подхода является упрощение кода и его инвариантность.

Во многих ситуациях можно ожидать определенных выгод от использования пространства имен `System.Data.Common`. Приложения, которые должны быть способны запускаться на многих платформах баз данных, являются первыми кандидатами на обобщение поставщиков в ADO.NET. Однако на практике каждый конкретный поставщик требует наличия расширений базовой модели. Базовый класс может оказаться не в полной мере пригодным для использования некоторыми поставщиками .NET без наличия в программном коде ссылок на специфичные для данного поставщика классы. Как результат, программирование с использованием обобщенных классов может оказаться более трудоемким, чем использование специфичных для поставщиков пространств имен. Несомненно, в следующих версиях ADO.NET практичность обобщенных базовых классов будет повышаться. Возможно даже, что специфичные для поставщиков классы, управляемые из обобщенного, в будущем даже попадут в немилость. В настоящее же время, пожалуй, всем программистам (разумеется, кроме самых безрассудных) имеет смысл подходить к обобщенной модели с изрядной долей осторожности.



Более подробную информацию об обобщенных базовых классах и `DBProvider Factory` вы можете получить из статьи Боба Бошемена (Bob Beauchemin) "Generic Coding with the ADO.NET 2.0 Base Classes and Factories" и статьи доктора Шахрама Хосрави (Shahram Khosravi) "Writing Generic Data Access Code in ASP.NET 2.0 and ADO.NET 2.0", опубликованных на сайте MSDN по адресам:

```
http://msdn.microsoft.com/library/default.asp?url=/library/
en-us/dnvs05/html/vsgenerics.asp
```

```
http://msdn.microsoft.com/library/default.asp?url=/library/
en-us/dnvs05/html/vsgenerics.asp
```

Управляемые поставщики в ADO.NET обладают некоторой долей интеллекта, которого не было в версии ADO тех же поставщиков. Например, поставщики .NET лучше используют подключения к базе данных. Они устанавливают и разрывают соединения по мере необходимости с целью оптимизации использования ресурсов сервера и клиента. Отличия между неуправляемым и управляемым поставщиком можно легко разбить на четыре категории.

- **Методика доступа к объектам**. Неуправляемый поставщик для доступа к необходимым объектам будет использовать идентификатор COM `progID`. При работе с управляемым поставщиком приложение полностью полагается на класс `Command`. Этот класс также имеет доступ к идентификатору COM `progID`, но он скрывает от разработчика детали этого доступа, что ускоряет процесс создания приложений и защищает его от ошибок. Этот класс также позволяет рационализировать доступ к данным клиента SQL и делает

возможным создание такого кода ADO.NET, который будет иметь один и тот же вид, независимо от того, использует ли доступ идентификатор COM `progID` или нет.

- **Обработка результирующих данных**. Неуправляемый поставщик для представления данных в приложении полагается на объекты `Parameter` объекта `Command`, а также на объекты `Recordset` и `Stream` модели ADO. Управляемый эквивалент использует классы `Parameter`, `DataSet`, `DataTable` и `DataReader`, а также методы `ExecuteReader`, `ExecutePageReader`, `ExecuteNonQuery` и `ExecuteScalar` объекта `Command` и потоки XML. Неуправляемый интерфейс COM всегда будет испытывать перегрузки от преобразования типов данных SQL в типы данных COM. Управляемые поставщики в этом отношении имеют существенное преимущество, так как используют транспортные потоки, основанные на XML.
- **Обновление данных**. Так как в неуправляемой среде используется постоянно поддерживаемое подключение, ресурсы также находятся в постоянном использовании. В результате этого пользователь всегда должен иметь подключение к базе данных, а разработчику приходится тратить массу времени на создание команд вручную. Управляемая среда использует подключения, только если требуется транспортировка данных. Вследствие этого использование ресурсов становится более эффективным, и пользователю нет необходимости поддерживать постоянное подключение. Как будет показано ниже в этой главе, управляемая среда предоставляет и другие средства автоматизации, включая метод `CommandBuilder`.
- **Формат передачи данных**. Неуправляемая среда использует передачу двоичных данных. Управляемые поставщики данных при передаче данных в ADO.NET 1.*x* полагаются исключительно на формат XML. Распределенные приложения в ADO.NET 2.0 также могут при передаче использовать двоичную сериализацию, значительно выигрывая в объеме передаваемых данных по сравнению с XML, в тех случаях, когда удаленность применима в задаче. Удаленность обеспечивает повышенную производительность и интероперабельность в процессах взаимодействия приложений .NET. Если либо источник, либо приемник не является приложением .NET, стандартизованный метод передачи данных обеспечивает формат XML. Это требует гораздо меньшего объема программного кода и меньших затрат на поддержку, чем неуправляемые методы передачи данных.

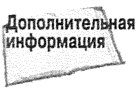
Отличия между методами передачи данных неуправляемых и управляемых поставщиков требуют от программиста основательного исследования. Формат передачи данных XML, используемый управляемым поставщиком, больше подходит для архитектуры SOA и Интернета, так как позволяет передавать данные через брандмауэры, которые обычно блокируют двоичные потоки. В то же время XML является более затратным методом передачи данных и не столь безопасен. В прошлом было привлекательно использовать ADO для потребностей локальных баз данных и ADO.NET для распределенных приложений, так как формат XML явно проигрывает в размерах и производительности. Также в этом имела значение обманчивая повышенная защищенность двоичных данных по сравнению с байтами ASCII при передаче по частной сети. Двоичная сериализация ADO.NET 2.0 обладает преимуществами производительности при направлении данных во внешние потоки, таким образом ослабляя, часто ущербное, стремление одновременно поддерживать модели ADO и ADO.NET.

SQL Native Client

В версиях SQL Server 2005 и ADO.NET 2.0 доступ к серверу баз данных уже не основывается на MDAC. Вместо этого в одном файле `.dll` содержится сборка, получившая название SQL Native Client. Ожидалось, что эта сборка решит все известные проблемы согласованности, связанные с распределенным обновлением массивного набора файлов MDAC, а также

повысит защищенность за счет сокращения количества доступных интерфейсов. Собственные протоколы компании Microsoft доступа из .NET в SQL Server, равно как и интерфейсы OLE DB, ODBC и собственные интерфейсы SQL Server, также были включены в эту сборку.

Было бы идеально, если бы для доступа к SQL Server требовалась только сборка SQL Native Client. К сожалению, идеал так и остался недостижимым. SQL Native Client поддерживает только SQL Server 7.0 и более поздние версии продукта. SQLXML не интегрирована в SQL Native Client. С большой вероятностью множеству приложений придется сохранить зависимость как от MDAC, так и от SQL Native Client. В конце концов, одним из основных преимуществ платформы .NET является доступ к гетерогенным источникам данных. Монолитный SQL Native Client способен упростить обслуживание и обеспечение безопасности содержащихся в нем интерфейсов, однако он так и не внес существенного сдвига в способе взаимодействия приложений с SQL Server. Аналогично, вероятнее всего, что доступ с помощью ODBC, OLE DB и, в некоторой мере, ADO так и останется проблематичным, поскольку расширение SQL Native Client предполагает зависимость от компонентов MDAC.



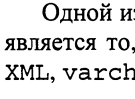
В SQL Server 2005 интерфейс SQLXML был обновлен до версии 4.0. Несмотря на то что он не интегрирован в поставщик SQL Native Client, SQLXML поддерживается средой .NET Framework посредством поставщика SQLXMLOLEDB, основанного на интерфейсе OLE DB. Дополнительную информацию об SQLXML и использовании его в программировании .NET вы найдете в главе 31.

Типы данных

ADO.NET использует XML для перемещения информации из базы данных в приложение. Не следует путать этот транспортный механизм с типом данных XML, поскольку он позволяет доставлять любые данные SQL, подобно тому, как TDS является основным механизмом транспортировки двоичных данных из SQL Server. XML может поддерживать все типы данных, поскольку является потоком битов. В противоположность этому приложения Visual Studio полагаются на управляемые типы данных для представления информации на экране. Другими словами, XML привнес в приложения дополнительный уровень преобразований. ADO.NET перемещает информацию между базой данных и приложениями в потоках XML. После этого поток XML нужно распаковать, чтобы представить реляционные данные приложению в нужном формате. Все ограничения данных и проблемы, рассмотренные в разделах, посвященных ADO, также применимы и к данным, поставляемым в приложение с помощью ADO.NET. Таким образом, на этапе разработки следует уделить особое внимание таким вопросам, как потеря данных и проблемы совместимости.

К счастью, управляемая среда обеспечивает надежное управление типами данных, используемых в СУБД. Использование ADO.NET может несколько понизить общую производительность транспортировки данных, поскольку использование потока XML требует операций упаковки и распаковки. В то же время маловероятно, чтобы в управляемой среде программист столкнулся с какими-либо проблемами преобразования данных. Ожидается, что в ближайшем будущем производительность операций ввода-вывода, использующих потоки XML, станет не хуже производительности передачи двоичных данных, используемой в ADO. Это сразу же поможет обуздать разрастание ошибок преобразования типов данных во время выполнения приложения.

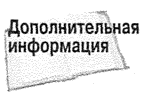
Одной из особо важных причин, почему стоит отдать предпочтение ADO.NET, а не ADO, является то, что ADO.NET лучше поддерживает новые типы данных SQL Server, такие как `XML`, `varchar(max)`, `varbinary(max)`, а также все пользовательские типы CLR.



Полную схему отображения типов данных SQL Server 2005 и среды .NET Framework см. в табл. 27.1 главы 27.

Тип данных XML в SQL Server 2005 не следует путать с потоками XML, используемыми ADO.NET для транспортировки данных. Тип данных XML поддерживается типом данных `System.Data.SqlTypes.SqlXml` в ADO.NET 2.0. Тип данных XML можно использовать для хранения документов XML и их фрагментов. ADO.NET 2.0 поддерживает чтение этого типа данных с помощью метода `XMLDataReader`. В отличие от других типов данных SQL Server 2005, тип XML проверяется на уровне SQL Server, а не на уровне ADO.NET. Это значит, что данные с типом XML несут с собой риск генерации ошибок при использовании метода обновления объекта `DataAdapter` (т.е. когда данные возвращаются в базу), которые невозможны при использовании других примитивных типов данных. SQLXML 4.0 предлагает мощную поддержку типа XML на стороне клиента, включая собственный набор компонентов доступа к данным. Напомню, что SQLXML 4.0 не реализован в составе ADO.NET 2.0.

Пользовательские типы данных CLR заслуживают особого внимания. Для использования этих типов в программном коде .NET структурно целостные версии соответствующих сборок должны существовать как на сервере баз данных, так и на серверах приложений. Это требование ограничивает гибкость решений. Совершенно не обязательно, чтобы строгие имена сборок на серверах баз данных и приложений совпадали, — главное, чтобы структура типа на всех этих серверах была определена одинаково. В этом требовании есть определенный смысл. Системные примитивные типы данных, такие как `int`, `char` или `bit`, должны существовать на всех серверах. Различие заключено в логистике, необходимой для поддержания синхронизации дополнительных свойств пользовательских типов на всех уровнях, в противоположность относительно статичным примитивным типам. Пользовательские типы, используемые только на сервере баз данных, на практике вряд ли будут иметь особую ценность, а пользовательские типы, развернутые в производственной среде, скорее всего, будут хрупкими благодаря своим неудобным требованиям. При использовании пользовательских типов в приложениях ADO.NET 2.0 необходимо тщательное планирование развертывания.



Развернутую дискуссию по вопросам создания и развертывания пользовательских типов вы найдете в главе 29.

Адаптеры данных и наборы данных

До настоящего момента мы рассматривали технологии ADO и ADO.NET как совершенно различные, несмотря на сходство имен и общее назначение. Ядром модели ADO является класс набора записей `Recordset`. Для модификации данных с помощью этого набора записей необходимо использовать либо программный интерфейс серверного курсора, который остается в базе данных, либо отправлять обновления в базу данных, как только они произошли в наборе данных. Оба метода доказали свою несостоятельность вследствие наличия вопросов конкуренции; к тому же оба метода требуют большого объема программного кода. Чтобы работать с множеством наборов записей `Recordset`, необходимо создавать множество подключений и жонглировать этими наборами в программном коде либо организовывать несколько наборов записей из базы данных в коллекцию и работать с ними по очереди. Каждый из методов доказал свою жесткость и требует больших цепочек часто повторяющегося программного кода.

Основной метод хранения данных в памяти в ADO.NET реализован с помощью класса набора данных `DataSet`. Объект `DataAdapter` используется для связывания источника данных и его образа в памяти. В ADO.NET 2.0 класс `DataSet` стал гораздо мощнее, чем когда бы то ни было ранее.

Объекты `DataSet` отключены от базы данных, что позволяет уменьшить нагрузку на базу и упростить масштабирование приложений. Программе требуется либо наполнять объект `DataSet` из базы данных, либо обновлять базу данных из этого объекта.

Объект DataAdapter наполняет объект DataSet из потока данных, идущего из базы. Также он обрабатывает вставки, обновления и удаления, которые должны быть распространены на базу данных.

Запросы DataSet могут ссылаться на множество объектов таблиц DataTable, содержащихся в объекте DataSet, при этом поддержка существующих отношений и типов данных происходит незаметно, когда данные модифицируются в объекте DataSet.

Оптимизация индексов в ADO.NET 2.0 существенно увеличила размеры кэша объекта DataSet, при этом существенно повысив быстродействие по сравнению с ADO.NET 1.*x*. Объект DataSet, наполняемый относительно небольшим объемом данных, не даст прочувствовать эффект от оптимизации индексов. По мере разрастания объекта DataSet эффект оптимизации станет более явным.



Пакет Visual Studio 2005 хорошо интегрирован с классами ADO.NET 2.0. Технология ADO.NET уже стала значительно проще в использовании. Множество средств интерфейса пользователя Visual Studio еще в большей мере упростили использование ADO.NET в приложениях. К примеру, DataSet может быть связан с элементом управления методом перетаскивания этого элемента с панели инструментов на объект элемента управления. К тому же на этапе разработки можно воспользоваться услугами мастеров объектов DataAdapter и DataSet, которые позволяют автоматически создавать наборы данных и программный код для их наполнения.

Метод CommandBuild позволяет автоматически создавать команды обновления, вставки и удаления, которые будет использовать объект DataAdapter для работы с базой данных, на основе инструкции SELECT, использованной для заполнения объекта DataSet. Одна строка программного кода поможет сгенерировать три операции над базой данных.

Функции двоичной сериализации в ADO.NET 2.0 позволяют ускорить перенос больших массивов между приложением и базой данных, а также между разными уровнями приложения. Эти функции особенно полезны при работе с удаленным источником данных, поскольку различные компоненты одного и того же приложения не страдают от дополнительных сложностей, связанных с расшифровкой двоичного потока данных.

По умолчанию класс DataReader использует односторонний последовательный курсор. Этот объект предлагает работу как в подключенном, так и в отключенном режиме. Пользователь может загрузить данные из базы компании, используя доступное подключение (в том числе и через Интернет). После этого в отключенном режиме данные будут доступны для просмотра (но не для непосредственной модификации, так как связь с базой данных была потеряна). В то время как пользователя может в большей мере заинтересовать использование обособленного объекта DataReader, ADO.NET 2.0 дополнительно предлагает методы преобразования объекта таблицы DataTable в DataReader с помощью метода GetTableReader, а также наполнения объекта DataTable из объекта DataReader или подключенного к базе данных объекта DataAdapter. Повышенная гибкость и сокращение объема программного кода, сопровождающие это усовершенствование класса DataTable, являются дополнительными причинами, по которым лучше использовать модель ADO.NET, чем ADO.

Одним из недостатков программного создания набора данных DataSet из объекта DataReader является то, что в таком случае DataSet не может быть типизирован. Преимуществом типизированного набора данных является возможность добиться большей производительности, так как не включаются дополнительные уровни явного преобразования данных или более сложные преобразования, кроме тех, которые необходимы в .NET. Еще одним достоинством является повышение читаемости программного кода. Например, для обнаружения и использования столбца объекта DataTable нетипизированного набора данных DataSet может использоваться следующий программный код:

```
City = dsOrder.Tables[n].Rows[n].ItemArray.GetValue(1).ToString()
```

Используя типизированный объект `DataSet` модели ADO.NET, программный код можно сделать более дружественным:

```
City = dsOrder.Invoice[0].Name;
```

Как мы видим, ссылки на типизированные объекты `DataSet` более краткие.

В то же время объекты `Recordset` модели ADO требуют меньшего объема программного кода для доступа к отдельным значениям. Кроме того, при использовании объекта `Recordset` к полям можно обращаться по имени, в то время как `DataSet` или обособленный объект `DataTable` требует указания индекса, основанного на порядковом номере значения в результирующем наборе данных. Использование ADO.NET предлагает и другие существенные преимущества, которые можно сразу не заметить, если позволить предубеждению, что "все равно ADO лучше", омрачить общую картину.

Модель ADO.NET 2.0 также предлагает использование множественного активного результирующего набора данных (MARS), доступного только для работы с SQL Server 2005. MARS можно себе представить как механизм создания пула сессий, подобно механизму создания пула подключений в технологии ADO. Так как существует масса причин создания множества подключений, в некоторых случаях (например, когда несколько запросов должны поддерживать целостность транзакции) MARS может обеспечить целостность транзакций и повысить производительность без необходимости разделения транзакции на два этапа, использующих разные подключения. Если в поставщике данных включена поддержка MARS, то инструкции отбора данных могут чередоваться согласно требованиям поставленной задачи. В операциях вставки, обновления и удаления, выполняемых со включенной поддержкой MARS, происходит сериализация инструкций DML и отбора данных. По умолчанию в SQL Server 2005 отключена поддержка MARS. Чтобы ее включить, достаточно задать в строке подключения параметр `MultipleActiveResultSets=true`.



Даже несмотря на то, что MARS по умолчанию отключен, для его поддержки все подключения подвергаются дополнительной нагрузке. Компания Microsoft заявляет, что установка параметра `MultipleActiveResultSets` в значение `false` не исключает эту нагрузку. Пожалуй, единственным основанием отключения поддержки MARS является необходимость генерирования состояния ошибки, когда в одном подключении отправляется больше одного запроса.

Модель ADO не реализует функции удаленного подключения. Но, подобно всем остальным технологиям, основанным на COM, она использует объектную модель распределенных компонентов DCOM как основу обмена данными в удаленных сетях. Это значит, что номер порта подключения часто изменяется и что сами данные имеют двоичную форму. Преимуществом этого подхода является то, что только немного взломщиков имеют знания, необходимые для просмотра данных (разумеется, если они после обнаружения данных смогут их дешифровать). Недостатками являются невозможность использования некоторых специфичных для ADO.NET функций, высокие технические затраты на передачу двоичных данных и невозможность прохождения через брандмауэры Web-сервера. Хорошо сконфигурированный брандмауэр закрывает порты и ограничивает прохождение двоичных данных.



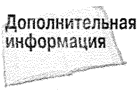
Проще говоря, DCOM — это технология COM, в которой компоненты могут взаимодействовать по сети. В DCOM компоненты для взаимодействия используют технологию удаленного вызова процедур (RPC). На практике защита и интеграция со стеком сетевых протоколов отдаляют технологию DCOM от COM, несмотря на то, что они имеют сходное назначение. Читателям, которым интересно детально ознакомиться с технологией DCOM, я бы посоветовал прочитать статью "DCOM Technical Overview", опубликованную в 1996 году на сайте библиотеки MSDN:

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/
dndcom/html/msdn_dcomtec.asp
```

ADO.NET обходит брандмауэры, используя для передачи данных формат XML и некоторый протокол передачи данных, например протокол передачи гипертекста HTTP. Данные передаются в формате ASCII, используя при этом один порт. В арсенале разработчика существует множество и других средств, которые позволяют дополнительно защитить прохождение потока XML по кабелю, таких как шифрование SSL, цифровая подпись, обмен сертификатами и самодостаточный протокол Web Services Security. Они могут оказаться полезными в условиях, когда передача текста по кабелю является недопустимой из соображений защиты данных.

В ADO.NET 2.0 содержится также масса и других улучшений. Разработчику будет целесообразно уделить внимание проверке того, действительно ли реальное поведение соответствует ожидаемому. Некоторые улучшения связаны с новыми функциями SQL Server 2005, и им будет уделено внимание в настоящей книге. Некоторые из наиболее интересных улучшений, связанных с платформой .NET, приведены ниже.

- Автоматическая эскалация транзакций из локальных в распределенные, когда в них задействовано множество источников данных.
- Улучшенная индексация объекта `DataTable`, что может привести к повышению производительности работы с большими массивами данных.
- Обновления из наборов данных `DataSet` могут быть отправлены в базу данных пакетами. Для этого свойство `UpdateBatchSize` адаптера данных `SQLDataAdapter` следует установить в значение больше нуля.
- Улучшенный поиск доступных источников данных. Например, объект `SqlClient` предназначен для поиска всех доступных экземпляров SQL Server в локальной сети.
- Улучшенное управление пулом подключений, в том числе возможность установки ограничений на количество подключений и очистки пула от всех существующих подключений.
- Программный интерфейс массовой вставки, основанный на ADO.NET 2.0, позволяет быстро переместить большие объемы данных из объектов `DataTable` и `DataReader`.
- ADO.NET 2.0 может служить источником данных для пакетов службы интеграции.
- Метод уведомлений, используемый для информирования процессов среднего уровня об изменении данных в базе. Это еще раз подчеркивает достоинства многоуровневой архитектуры, где наборы данных кэшируются, а исходные данные быстро меняются. Объект `SqlDependency` связан с объектом `SqlCommand`. После этого приложение с помощью метода `SQLNotificationRequest` опрашивает очередь брокера служб на предмет наличия уведомлений об обновлении данных.
- Возможность асинхронного выполнения. Способность создавать новые рабочие потоки, одновременно продолжая обработку в основном потоке, не является новой, за исключением возможности перечисления и управления асинхронными рабочими потоками. Связанные методы запросов теперь поддерживают типичную программную структуру .NET `begin...end` для асинхронных операций (например, `BeginExecuteNonQuery...EndExecuteNonQuery`). Следует обратить внимание, что WinForms не поддерживает асинхронные очереди ADO.NET 2.0. Асинхронное выполнение по умолчанию отключено. Для его включения в строке подключения нужно установить параметр `AsynchronousProcessing=true`.
- Метрика производительности SQL Server представлена в интерфейсе программирования как свойство подключения.



Четыре важных расширения ADO.NET могут быть использованы в серверном коде CLR: `SqlContext`, `SqlPipe`, `SqlTriggerContext` и `SqlDataRecord`. Все они детально описаны в главе 27.

ADO.NET в Visual Studio 2005

Администраторы баз данных испытали шок, когда компания Microsoft заявила о намерении перенести утилиту Enterprise Manager из SQL Server в Visual Studio. Смысл такого перемещения был совершенно непонятен. Утилита SQL Server Management Studio переняла привычный графический интерфейс Visual Studio, но эти утилиты остались отдельными продуктами, которые объединяет ненамного больше, чем сходный интерфейс пользователя.

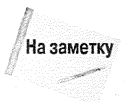
Полное описание интегрированной среды разработки Visual Studio выходит за рамки рассмотрения настоящей главы. Эта среда достаточно хорошо описана в документации Microsoft Visual Studio. В данной главе мы рассмотрим только те компоненты Visual Studio, которые непосредственно связаны с разработкой приложений, использующих ADO.NET 2.0.

Server Explorer

Создание любого проекта ADO.NET 2.0 начинается именно с окна Server Explorer. При создании проекта ADO.NET 2.0 программисту нужно либо добавить подключение к данным, либо выбрать существующее подключение. Новое подключение к данным можно определить с помощью мастера конфигурирования источника данных, выбрав в меню Data пункт DataSource Configuration Wizard. Можно также щелкнуть правой кнопкой мыши на значке Data Connections в окне Server Explorer и выбрать в открывшемся контекстном меню пункт Add Connection.

В дополнение к управляемым подключениям к базе данных Server Explorer предлагает некоторые другие полезные инструменты, предназначенные для работы с базой данных, в частности диаграммы метаданных базы, средства доступа и генерации программного кода. И хотя компоненты базы данных, предоставляемые в Server Explorer, не имеют поддержки, аналогичной реализованной в Management Studio, они в достаточной мере сходны. В некоторых отношениях Server Explorer даже лучше, чем Management Studio. В частности, в нем можно не только создавать, модифицировать и просматривать объекты базы данных, но и создавать типизированные объекты `DataSet` и `DataTable` путем перетаскивания таблиц из Server Explorer на `DataSet` в процессе проектирования.

Сторонники чистого программирования могут неприязненно относиться к этому простому подходу, однако не следует сбрасывать со счетов выигрыш в производительности и согласованности, связанный с типизированнными наборами данных, что при обычных условиях требует утомительного и точного программирования объектов `DataSet`. Разработчики и администраторы баз данных, зажатые временными рамками и большой нагрузкой, по достоинству оценят эту помощь. Если воспользоваться помощью Visual Studio, то у любого программиста появляется реальный шанс досрочно справиться с проектом.



Функциональность Server Explorer доступна из программ с помощью пространства имен `ServiceController`. Это пространство имен реализует изящный доступ высокого уровня к действиям автоматизации, таким как запуск и остановка служб.

Отладка ADO.NET

Интерактивную отладку кода приложения в среде Visual Studio можно провести просто и изящно. Код ADO.NET специализируется на источнике данных. Способность включать в сессию отладки не только код приложения, но и код T-SQL, выполняемый в источнике данных, дает в руки разработчика несомненные преимущества. В то же время проникновение отладчика в код T-SQL может оказаться несколько болезненным. Чтобы получить возможность проникновения в программный код базы данных в процессе отладки приложения, в Visual Studio должны быть удовлетворены все перечисленные ниже условия.

- В подключении к данным в окне Server Explorer должен быть включен режим Ailow SQL/CLR Debugging.
- Во вкладке Debug диалогового окна свойств проекта в Solution Explorer должен быть установлен флажок Enable SQL Server Debugging.
- Если SQL Server находится не в одном экземпляре операционной системы с Visual Studio, то на компьютере SQL Server должна быть выполнена установка компонентов удаленного доступа Remote Components Setup пакета Visual Studio.
- Если SQL Server и Visual Studio запущены на разных компьютерах и любая из этих систем работает в Windows XP с включенным брандмауэром, то в настройках брандмауэра должна быть разрешена удаленная отладка. На компьютере с Visual Studio это предполагает добавление приложения `devenv.exe` в список разрешенных и открытие порта 135. На компьютере с SQL Server это предполагает добавление разрешения для приложения `sqlservr.exe` и открытие порта 135.
- Если должны отлаживаться серверные компоненты CLR, необходимо установить и сконфигурировать на компьютере SQL Server 2005 утилиту Visual Studio Remote Debug Monitor.
- Если удаленный сервер является экземпляром версий SQL Server 7 или SQL Server 2000, то необходимо сконфигурировать DCOM, чтобы предоставить возможность удаленной отладки с помощью утилиты `dcomcnfg.exe`. Вид этой процедуры зависит от операционной системы. Подробности можно узнать в документации по DCOM или Visual Studio.
- В отношении процесса отладки документацию Visual Studio можно считать совершенной. В ней описано, как использовать инструменты отладки и что нужно устанавливать на SQL Server при включенном параметре удаленной отладки.

Отладка кода T-SQL может приостановить все управляемые подключения к экземпляру сервера баз данных на период, пока разработчик пошагово выполняет хранимую процедуру или функцию. При этом будут сохранены все блокировки ресурсов в соответствии с обычной настройкой конкуренции в SQL Server. По существу, это означает, что отладки на загруженном работой сервере баз данных следует избегать. Развивая эту мысль, можно сказать, что использование отладчика Visual Studio на производственном сервере может вызвать больше проблем, чем способно решить.

Трассировка приложения

Когда приложение уже было передано в эксплуатацию, такие инструменты, как Visual Studio или Server Explorer, могут мало что дать. Для подобной ситуации предлагаются другие инструменты, в частности отладчик Windows и SQL Profiler. Они способны помочь идентифицировать проблему, которая всплыла на поверхность только в условиях промышленной эксплуатации приложения. Утилита SQL Profiler является отличным инструментом настройки производительности и разрешения проблем в SQL Server. Многие читатели, должно быть, зна-

комы с возможностью трассировки в режиме выполнения приложения в ODBC. Несмотря на то что трассировка ODBC является исключительно словесной и поверхностной, она идентифицирует состояния ошибок и проблемы подключения на уровне инструкций приложения. Также возможно использовать трассировку ODBC даже в производственной среде, не рискуя заблокировать таблицы. Языки Visual Studio в свое время предлагали различные типы встроенных средств отладки. Среда .NET Framework в пространстве имен `System.Diagnostics.Trace` предлагает диагностическую альтернативу, которая может оказаться полезной как на стадии отладки, так и в процессе эксплуатации приложения.

Именно разработчик на стадии программирования должен установить в свое приложение *точки трассировки*. По умолчанию любой домен приложения .NET (т.е. основная единица изолированности в среде CLR) содержит объект прослушивания `DefaultTraceListener`; также он может содержать в своей коллекции и другие объекты прослушивания. Обычно такой объект направляет вывод результатов трассировки в файл или журнал событий. Разработчик приложения должен определить в программном коде точки трассировки при вызовах методов или даже метрик, таких как счетчики производительности.

Разработчик также может определить переключатели трассировки, устанавливающие различные уровни вывода трассировки на этапе производственной эксплуатации приложения. Переключатели трассировки могут быть установлены во время выполнения программы в файле `app.config`. При определенных условиях трассировку вообще можно отключить. Если возникают проблемы, то трассировку можно включить на уровне, соответствующем данной проблеме.



Среди прочей массы полезной информации, доступной в Интернете и посвященной инструментарию трассировки в среде .NET Framework, стоит отметить отличную статью Джона Фансета (Jon Fancet) "Powerful Instrumentation Options in .NET Let You Build Manageable Apps with Confidence":

`http://msdn.microsoft.com/msdnmag/issues/04/04/InstrumentationinNET/`

и статью Боба Бошемена (Bob Beauchemin) "Tracing Data Access":

`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/vsgenerics.asp`

В версии ADO.NET 2.0 инструментарий трассировки был расширен компанией Microsoft и теперь позволяет включать встроенные точки трассировки в сборки ADO.NET, сборки `.dll` SQL Native Client и прочих поставщиков .NET. Основной задачей при этом была интеграция этих функций трассировки с возможностями трассировки класса `System.Diagnostics.Trace`. К сожалению, на момент написания книги данная интеграция еще не была реализована, а метод сбора информации об этой трассировке не был документирован.

Основы создания приложений

Разговор об ADO.NET 2.0 нельзя считать законченным, если не упомянуть в нем о разных методиках программирования, о которых уже говорилось в этой главе. В этом разделе в качестве примеров вам будут предложены:

- программный код, использующий для взаимодействия с SQL Server первичную interop-сборку .NET `adodb.dll`;
- программный код, использующий `SqlClient` для выполнения эквивалентной операции;
- код, демонстрирующий доступ с помощью SQL Native Client и обобщенных базовых классов.

Как уже отмечалось выше, технологии ADO и ADO.NET фундаментально отличаются друг от друга, несмотря на сходное назначение. В предлагаемых примерах вы увидите, как различные операции взаимодействия с базой данных реализованы в каждой из этих технологий.

Здесь мы уделим мало внимания проектированию графического интерфейса, XML и удаленной работе в среде .NET Framework. Конструктор Windows Forms, ASP.NET, SOA и удаленный доступ — очень обширные вопросы, требующие больше места, чем им может быть отведено в настоящей книге. В этом разделе будут продемонстрированы методы перемещения информации в источник данных и из него, а также методы работы с компонентами ADO.NET. Основной нашей целью было показать, насколько просто и изящно можно перемещать информацию с уровня приложения на уровень данных и в обратном направлении. Что будет впоследствии происходить с данными, к которым получен доступ, пусть решит сам разработчик.

В любом проекте ADO.NET в первую очередь нужно подключиться к источнику (или источникам) данных, который будет использоваться на этапе разработки проекта в Server Explorer.

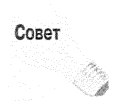


Примеры программ, приведенные в настоящей главе, можно загрузить с Web-сайта книги. В этом коде использовано консольное приложение для демонстрации нижеописанных методик.

Подключение к SQL Server

Простейшим способом создания подключения является использование мастера конфигурирования источника данных. Выберите в меню Visual Studio команду Data⇨Add New Data Source или просмотрите источники данных, ассоциированные с существующим проектом приложения, выбрав в меню пункт Data⇨Show Data Sources.

Не сложнее определить строку подключения и программным путем. Строка подключения представляет собой набор пар "имя–значение". Единственное, что несколько усложняет ситуацию, — это то, что строки подключения у разных поставщиков .NET несколько отличаются, а подключение ADO вообще стоит в стороне.



В среде .NET Framework 2.0 строки подключения более гибкие, чем в предыдущих версиях. Например, в строках подключения поставщика `ADODB.dll` или SNAC разрешено называть SQL Server и как "Data Source", и как "Server".

Можно определить строку подключения, основываясь на значениях, хранимых в файловой системе, реестре или определяемых самим пользователем. Если используется обобщенный базовый класс, можно определить используемого поставщика и строку подключения в файле `app.config`.

Адаптеры данных

Мастер конфигурирования источника данных позволяет задать таблицы, представления, хранимые процедуры и функции, которые будут использованы для создания набора данных `DataSet`, как типизированного, так и нетипизированного. В качестве альтернативы набор данных `DataSet` может быть добавлен в проект и определен в Solution Explorer. В последнем случае в проекте откроется окно Dataset Designer.

Для определения источника данных и набора данных приложения окно Dataset Designer позволяет использовать Server Explorer и панель инструментов. Инструмент Dataset Designer реализует еще один отличный способ создания наборов данных, гарантирующий создание только типизированных наборов. Сторонники "чистого" программирования могут при желании вводить вручную определения типизированных наборов данных, хотя вряд ли для использования такого подхода найдутся достаточные объяснения.

Вспомним, что типизированный набор данных DataSet позволяет избежать проникновения в код ошибок преобразования данных в процессе выполнения приложения. Если вы собираетесь использовать нетипизированный набор DataSet, то нет причин использовать и Dataset Designer. Все, что в данном случае нужно, — это объявить новый, пустой объект DataSet перед тем, как он будет заполняться с помощью заданного объекта адаптера DataAdapter:

```
'заполнение нетипизированного набора данных из адаптера
Dim daScrapReasons As SqlDataAdapter = _
     New SqlDataAdapter(sSQLScrapReasons, cnADONET2)

'создание нетипизированного набора данных
Dim dsWOWithScrap As New DataSet

'заполнение набора данных из адаптера
daScrapReasons.Fill(dsWOWithScrap, "ScrapReason")
```

Если разработчик выбирает типизированный набор данных, то можно использовать мастер конфигурирования адаптера таблицы (Table Adapter Configuration Wizard) для создания адаптера к каждой из таблиц, определенных в объекте DataSet. Этот мастер запускается из контекстного меню области конструктора Dataset Designer. В этом мастере разработчик может выбрать таблицы базы данных или определить запросы и хранимые процедуры, откуда будут наполняться адаптеры таблиц. Как только способ наполнения таблиц будет определен, мастер автоматически создаст инструкции INSERT, UPDATE и DELETE, которые будут выполняться при вызове в наборе данных метода DataAdapter.Update. Сформированные инструкции можно просмотреть и отредактировать на панели свойств адаптера. Также можно просмотреть и программный код, сформированный на панели конструктора. Редактирование этого кода не рекомендуется, так как любые изменения могут вызвать некорректную работу объекта. В то же время любые изменения, выполненные вручную, будут утеряны, если набор данных или адаптер таблицы будет заново сгенерирован мастером.

Объект чтения данных и наборы записей

Объект Recordset модели ADO может быть обработан как программный интерфейс серверного курсора или как клиентский курсор. На стороне клиента этот объект может использоваться после закрытия соединения с базой данных, но только от разработчика зависит то, как и когда данные могут быть изменены и возвращены на сервер.

```
'Набор записей из серверного курсора ADO
Dim rsADOWOWithScrap As New ADODB.Recordset
rsADOWOWithScrap.CursorLocation = CursorLocationEnum.adUseServer

rsADOWOWithScrap.Open(sSQLWOWithScrap, cnADO,_
   CursorTypeEnum.adOpenForwardOnly, _
   LockTypeEnum.adLockReadOnly)
' Изолированный набор данных из клиентского курсора ADO
Dim cmdADOWOWithScrap1 As New ADODB.Command
Dim rsADOWOWithScrap1 As New ADODB.Recordset
cmdADOWOWithScrap1.CommandText = sSQLWOWithScrap
   cmdADOWOWithScrap1.CommandType = _
CommandTypeEnum.adCmdText
rsADOWOWithScrap1.CursorType = _
   CursorTypeEnum.adOpenStatic
rsADOWOWithScrap1.LockType = _
   LockTypeEnum.adLockBatchOptimistic
rsADOWOWithScrap1.CursorLocation = _
   CursorLocationEnum.adUseClient
```

```
cmdADOWOWithScrap1.ActiveConnection = cnADO
rsADOWOWithScrap1 = cmdADOWOWithScrap1.Execute
```

Объект `DataReader` модели ADO представляет собой однонаправленный клиентский курсор. Его можно загрузить при необходимости в объект `DataTable`. Объект `DataAdapter` содержит методы вставки, обновления и удаления данных объекта `DataTable` набора данных `DataSet` и возвращения изменений в базу данных.

Потоки

Данные могут быть направлены в поток и извлечены из него либо в формате XML, либо в двоичном виде. Из-за переходной природы с потоками несколько тяжело работать. В общем случае поток может быть поглощен только единожды, после чего он исчезает:

```
Dim bfBINWOWithScrap As New Binary.BinaryFormatter
Dim msXMLWOWithScrap As New MemoryStream()
Dim msBINWOWithScrap As New MemoryStream()
'Получение потока XML из набора данных
bfBINWOWithScrap.Serialize(msXMLWOWithScrap, _
   dsBINWOWithScrap)
```

Обычно потоки XML используются в домене приложения .NET Framework и для перемещения данных в среде, ориентированной на службы. Двоичные потоки чаще всего используют в сценариях, где используется удаленный доступ между доменами приложений. Для сериализации двоичного потока (например, при записи на диск) необходимо глубокое знание данных. В то же время потоки XML содержат метаданные и могут быть сериализованы любым получателем, который способен манипулировать форматом XML.

Асинхронное выполнение

В отдельных случаях для работы с базой данных запросу или процедуре требуется продолжительное время, в течение которого может быть выполнена другая полезная работа. Несмотря на то что существует ряд правил, ограничивающих ситуации, когда может быть использовано асинхронное выполнение, эта возможность существенно повышает гибкость приложений, что и продемонстрировано в следующем примере:

```
Dim rdrAsyncScrapCountInit As IAsyncResult = _
   cmdScrapCount.BeginExecuteReader

'выполняем другую работу
Dim rdrAsyncScrapCount As SqlDataReader = _
   cmdScrapCount.EndExecuteReader(rdrAsyncScrapCountInit)
```

Подключение к базе данных может одновременно обслуживать только один асинхронный запрос. Это значит, что приложению или компоненту среднего уровня может потребоваться несколько подключений. В ADO.NET 2.0 не поддерживаются асинхронные запросы из серверных процедур CLR.

Использование одного значения из базы данных

Приложения часто интересуют значения только одного или нескольких столбцов строки таблицы. В ADO существовала возможность выполнить запрос и вернуть результат в объект `Recordset`, имеющий одну строку и один столбец. Также было возможно выполнить инструкцию с выходными параметрами, определенными в ее коллекции параметров. Использование в объекте `Recordset` одного значения или одной строки считается наименее масштаби-

руемым решением благодаря дополнительной нагрузке, необходимой для многократного создания и раздробления этого объекта. Исходя из этого, в ADO всегда отдавалось предпочтение использованию коллекции параметров:

```
paramscrapWOCount = _
   cmdScrapWOByProduct.CreateParameter("ScrapWOCount", _
DataTypeEnum.adInteger, _
   ParameterDirectionEnum.adParamOutput)
cmdScrapWOByProduct.Parameters.Append(paramscrapWOCount)
cmdScrapWOByProduct.Parameters.Item("ProductName").Value = sProduct-
Name
```

В ADO.NET выбор вариантов гораздо шире, и выбор самого подходящего из них не столь очевиден. Значение может быть возвращено в объект `DataTable`, имеющий одну строку в длину. Этот метод имеет определенные преимущества в некоторых сценариях, где точка данных используется совместно с другими таблицами `DataTable` в объекте `DataSet`. К тому же может быть использован метод `ExecuteNonQuery` для заполнения переменных с помощью коллекции параметров. когда интерес вызывают несколько столбцов одной строки. Также может быть использован метод `ExecuteScalar`, когда необходимо получение только одного значения:

```
Dim iScrapWOCount As Integer = _
   cmdScrapCount.ExecuteScalar()
```

или

```
cmdScrapCountByProduct.Parameters.Add("@ScrapWOCount", _
   SqlDbType.Int).Direction = _
   ParameterDirection.Output
cmdScrapCountByProduct.ExecuteNonQuery()
```

Модификация данных

Когда приложение .NET вносит в данные какие-либо изменения, они должны быть отражены в базе данных. Это можно осуществить как в пессимистической модели конкуренции, так и в оптимистической. Модель конкуренции по своей сути является механизмом, способным помочь разработчику решить, использовать ли для обмена данными с базой изолированный объект ADO.NET `DataClass` (в модели оптимистической конкуренции) или метод, требующий наличия подключения (в пессимистической модели).



Пессимистическая модель конкуренции может оказаться полезной при массовой загрузке данных или в случае использования в инструкции `ExecuteNonQuery` запросов DML (`INSERT`, `UPDATE` или `DELETE`). Использование оптимистической модели конкуренции выгодно в тех случаях, когда приложению нужно при сбоях операций модификации данных предпринимать повторную попытку выполнения операции или записывать сообщение в журнал ошибок.

Обновление набора данных в приложении .NET требует не более чем обычной операции присвоения. Все или несколько строк в объекте `DataSet` могут быть вставлены, удалены или обновлены. Когда некоторая строка модифицируется в объекте `DataSet`, она помечается как измененная. Запуск метода `Update` объекта `DataAdapter` вызывает выполнение инструкции `INSERT`, `UPDATE` или `DELETE`, определенной в объекте, в базе данных с использованием адаптера данных. Существует вероятность, что между операциями чтения в адаптер данных и записью обновлений из него в базу некоторый другой пользователь уже изменил исходные данные в базе. В таком случае, если операция вставки срывается, в программном коде следует определить, имела ли место ошибка конкуренции. Это обычно выполняется с использованием

столбца timestamp или с помощью сравнения исходных значений набора данных с текущими значениями базы данных в ходе выполнения метода Update. Разрешение возникающих коллизий конкуренции целиком возлагается на разработчика.

В ADO.NET 1.1 инструкции INSERT, UPDATE и DELETE, выполняемые методом Update, могут быть автоматически сгенерированы мастером конфигурирования адаптера таблицы. Сгенерированные инструкции сравнивают все столбцы таблицы в наборе данных с соответствующими столбцами базы данных, используя параметр конкуренции .NET CompareAllSearchableValues. Сгенерированные инструкции были неуправляемыми, малопроизводительными и не позволяющими удобно разрешать коллизии. В данной ситуации лучше использовать хранимые процедуры, даже для инструкций объекта DataAdapter. В ADO.NET 2.0 хранимые процедуры могут создаваться с помощью соответствующего мастера, хотя их сложно назвать оптимальными. Могут быть также определены различные параметры конкуренции, отличные от CompareAllSearchableValues. В частности, для идентификации возникновения коллизии можно использовать параметр CompareRowVersion для проверки изменения первичного ключа и версии строки, а также OverwriteChanges для проверки только первичного ключа. Эти параметры несколько улучшают работу метода обновления, однако все же лучше создавать собственные инструкции для адаптера данных, используя при этом хранимые процедуры, а не внедренные в объект инструкции SQL.

Связывание с элементами управления

Чтобы сократить объем создаваемого вручную программного кода, в Visual Studio для связывания данных с элементом управления требуется всего лишь перетащить компонент данных на нужный элемент. В результате будет создан связанный элемент управления данными. Ясно, что потребуются и дополнительные действия, однако это уже относится не к программированию в ADO.NET, а к вопросам проектирования форм.

Резюме

В этой главе мы сравнили объектные модели ADO и ADO.NET и исследовали некоторые их преимущества. По всеобщему признанию, предпочтение отдается ADO.NET. Несмотря на то что в некоторых сценариях использование ADO.NET 1.*x* вряд ли имеет превосходство над использованием старой доброй ADO, в этой главе мы попытались показать, что версия ADO.NET 2.0 предложила на настоящий момент наиболее полноценную технологию высокоуровневого доступа к данным, разработанную компанией Microsoft.

ГЛАВА

31

Использование XML, XPath и XQuery

В этой главе...

Использование типа данных XML

Создание запросов FLWOR

Слияние SQL и XQuery

Декомпозиция XML

Интернет постепенно переходит от роли обычного транспортного средства доставки электронной почты и представлений данных (т.е. Web-страниц) к роли высокопроизводительного средства обеспечения взаимодействия данных с помощью ориентированной на службы архитектуры и XML. Несмотря на повсеместное распространение технологии XML, архитектору данных следует решить, какую роль отвести XML в создаваемой им модели базы данных. SQL Server 2005 обеспечивает техническое сопровождение практически любого архитектурного решения.

Во время презентаций SQL Server 2005 в начале 2005 года, когда я описывал новые возможности, связанные с XML, чаще всего мне задавали вопрос о том, зачем нужно внедрять XML в базу данных. Ни один человек из сообщества пользователей SQL Server не согласится с мнением, что однажды XML вытеснит реляционные базы данных. В то же время бывают ситуации, когда имеет смысл хранить некоторые данные XML, например получать и обрабатывать документы XML. Когда данные псевдоструктурированы (т.е. часто изменяют свой формат), подвижный документ XML может стать более эффективным, чем стационарная реляционная база данных.



Введение в XML, XML Schema и XST, содержащееся в файле `XMLPrimer.pdf`, можно загрузить с Web-сайта книги. Примеры программного кода, приводимые в настоящей главе, содержатся в файле `ch31.sql`, находящемся там же.



В версии SQL Server 2000 были впервые продемонстрированы возможности XML первого поколения для публикации материалов в Web и работы с данными XML. По сравнению с этим набором функций возможности SQL Server 2005 выглядят просто революционными. Теперь само ядро базы данных поддерживает данные XML и выполнение запросов XQuery.



С выходом пакета обновлений SP1 была существенно повышена производительность компиляции запросов XQuery, содержащих сложные схемы XML, а также производительность использования языка XML DML, предназначенного для обновления XML.

Когда SQL Server получает данные XML, существуют три возможных варианта их хранения.

- Хранить их с использованием типа данных XML. Если структура XML подвижна, то данный вариант станет наилучшим выбором.
- Разобрать XML на составные части с помощью функции OpenXML, которые впоследствии хранить как реляционные данные. Если данные XML уже являются частью реляционной схемы или данные должны быть доступны для быстрой отчетности, их декомпозиция в реляционные таблицы — наилучший выбор.
- Последним вариантом является хранение XML как большого текста. Но поскольку в SQL Server 2005 имеется множество средств работы с данными XML, данный вариант нельзя назвать подобающим.

Тип данных XML

Будучи встроенным типом данных SQL Server, тип XML может использоваться любым из перечисленных ниже способов:

- объявляться как локальная переменная T-SQL;
- использоваться как параметр хранимой процедуры или пользовательской функции;
- возвращаться из пользовательской функции;
- создаваться как тип данных столбца таблицы.

Тип данных XML позволяет хранить как целый документ XML, так и его фрагмент, при максимальном объеме в 2 Гбайт. Фрагментом могут быть строки одного заказа, в то время как полный документ может содержать данные обо всех заказчиках вместе с их заказами и строками заказов.

Данные XML должны быть хорошо отформированными (т.е. синтаксически корректными) и при необходимости могут проверяться относительно коллекции схемы XML.

Использование типа данных XML имеет ряд существенных преимуществ. Ядро базы данных фактически разделяет отдельные элементы и атрибуты XML, а затем хранит их как обычные двоичные данные, подобно типу `VarBinary(max)`. Дескрипторы XML хранятся как метки, указывающие на фактические данные. Это существенно сокращает объем хранимых данных, повышает производительность и позволяет ядру базы данных использовать индексы XML и передвигаться по данным с помощью XQuery.

Распределение и преобразование

Поскольку тип данных XML является новым, существуют определенные правила преобразования данных из типа XML и в него. В документ XML и его узлы могут быть вставлены любые символьные данные.

Тип данных XML может быть преобразован в следующие связанные символьные типы и получен из них:

- `Char`, `VarChar`
- `NChar`, `NVarChar`
- `VarBinary(max)`;
- `SQL_Variant` (посредством типа `VarChar`).

Тип данных XML может быть получен из типов `text` и `ntext`, но не может быть преобразован в них.

Ограничения типа XML

В связи с тем, что тип XML хранится не так, как обычные реляционные данные, и имеет вложенный формат, при работе с ним существует ряд дополнительных ограничений.

- Его нельзя трактовать как обычные символьные типы.
- Он не может участвовать в операциях сравнения (таких как =, > или <), за исключением `IS NULL` и `IS NOT NULL`.
- Он не может участвовать в сортировке и группировке (т.е. в предложениях `SORT BY` и `GROUP BY`).
- Он не может использовать встроенные скалярные функции, за исключением `isnull()` и `coalesce()`.
- Он не может быть использован в качестве ключевого столбца или ограничения на уникальность.
- Он не может быть объявлен с сопоставлением.

Коллекции схем XML

Риск использования XML заключается в том, что он настолько гибок и подвижен, что без наличия каких-либо ограничений его данные могут содержать все, что угодно. С той же проблемой обычно сталкиваются в столбцах комментариев, которые могут содержать все, что угодно, от изменений адреса до дат рождения.

Решение заключается в создании дополнительного (но необязательного) пространства имен определения схемы XML (XSD) и подключения к нему переменной или столбца XML. Схема XML может быть определена для применения к документу в целом или только к его некоторому фрагменту.

Первым шагом является создание XSD как составной части коллекции схемы XML в базе данных. Как только схема будет создана, могут быть также созданы столбцы или переменные, ссылающиеся на нее. В следующем примере в схему учебной базы данных XMLearn добавляется определение XSD с названием `ItemSchema`:

```
CREATE DATABASE XMLearn
USE XMLearn

CREATE XML SCHEMA COLLECTION ItemSchema AS N'
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="resume-schema"
    targetNamespace="resume-schema"
    elementFormDefault="qualified">
  <xsd:element name="Item" type="ItemType"/>
  <xsd:complexType name="ItemType" mixed="true">
  <xsd:sequence>
    <xsd:element name="SKU" type="xsd:string" minOccurs="1"/>
    <xsd:element name="Size" type="xsd:string" minOccurs="1"/>
    <xsd:element name="Color" type="xsd:string" minOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>'
```



Чтобы просмотреть установленные схемы, выполните запрос к системному представлению `sysxml_schema_collections`.

Когда определение схемы XSD установлено в коллекции схем XML, переменная или столбец с типом XML при создании могут быть подключены к этому определению:

```
CREATE TABLE XMLGeneric (
  ID INT IDENTITY NOT NULL PRIMARY KEY,
  Data XML (имя_схемы)
  )
```

Схема XSD также может быть определена и в Management Studio. Свойства столбца содержат пространство имен схемы XSD, которое будет использоваться для выбора XSD из коллекции схем.



SQL Server 2005 не поддерживает старые определения типов документа (DTD), используя исключительно XSD.

Индексы XML

Существенным преимуществом использования типа XML является то, что узлы XML сами по себе могут быть проиндексированы. Индексы XML повышают производительность запросов XQuery, однако не используются при извлечении данных XML с помощью реляционных запросов. В моей практике эти индексы зарекомендовали себя очень хорошо. Создание индексов XML является многошаговой операцией, включающей в себя создание первичного и вторичного индексов.

Первичный индекс XML должен создаваться первым. Он связывает узлы XML с реляционным кластеризованным индексом и позволяет создавать индексы, специфичные для XML.

```
CREATE [ PRIMARY ] XML INDEX имя_индекса
  ON <объект> ( имя_столбца_xml )
  [ USING XML INDEX имя_индекса_xml
    [ FOR { VALUE | PATH | PROPERTY } ][ ; ]
```

Таблица должна иметь кластеризованный индекс, который не может быть изменен, пока существуют индексы XML.

Как только первичный индекс создан, для повышения производительности запросов XQuery могут быть созданы вторичные индексы.

- **Value**. Индексирование данных для поиска по значению.
- **Path**. Индексирование имен узлов для поиска по структуре XML.
- **Property**. Оптимизация предикатов пар "имя–значение".

Выполнение запросов к данным XML

Когда данные хранятся непосредственно в столбцах с типом XML, для извлечения этих данных, их вставки и обновления можно использовать XPath и XQuery. Язык XQuery является относительно молодым, и работа над ним продолжается в настоящее время.

Если вы когда-либо использовали XPath, считайте, что прошли половину пути в освоении XQuery. Практически все допустимые инструкции XPath являются также и инструкциями XQuery. К тому же XPath используется для отбора значений в выражениях XQuery FLWOR.

Для полного описания языка XQuery и методов слияния реляционных запросов с запросами XQuery могла бы понадобиться толстая книга объемом в тысячу страниц. Однако можно определенно заявить следующее: по мере эволюции технологий работы с данными, XML и XQuery станут неотъемлемой частью любой связанной с данными деятельности.

Xpath

XPath является функцией XML. Она привносит в XML ограничения результатов и средства обобщения при навигации по иерархии XML для доступа к любому узлу (элементу, атрибуту или фрагменту текста) в документе XML. Программный код XPath может быть добавлен для фильтрации данных в стилевую таблицу XSL или в другие запросы XML. Выгоднее всего использовать XPath в шаблонах SQL Server.

Вначале XPath должен идентифицировать узел. Идентификация узла функционально сходна с именем столбца в предложении `WHERE` инструкции `SELECT`. Например, для перехода к элементу `event`, содержащемуся в элементе `tour`, используется следующая структура:

```
/Tours/Tour/Event
```

В приведенном примере использована абсолютная адресация. Местонахождение узла может быть указано и относительно текущего положения, чтобы фрагмент XSL, работающий с дочерними данными, мог выбрать узел родителя для процесса фильтрации.

Чтобы спуститься по дереву иерархии XML до атрибута, используется символ `@`, как в следующем примере:

```
/Tours/Tour/@Name
```

После того как идентифицирован корректный узел, XPath позволяет использовать несколько средств фильтрации для отбора корректных данных. Для применения фильтра `WHERE =` критерий определяется сразу после названия узла и заключается в квадратные скобки:

```
/Tours/Tour/@Name ['Gauley River Rafting']
```

В следующем примере используется функция запроса для извлечения данных XPath в реляционном запросе:

```
SELECT Customer, OrderDetail.query(
  '/Items[1]/Item[1]/SKU')
  FROM XMLOrders
```

Запросы FLWOR

Запросы FLWOR являются сердцем языка XQuery. Аббревиатура FLWOR составлена из первых букв последовательности предложений For-Let-Where-Order By-Return. В самом названии заложен синтаксис данных запросов. Это аналогично последовательности операций в обычном реляционном запросе `SELECT`, однако запросы FLWOR последовательно проходят по иерархическим данным XML.

Назначение предложения `For` соответствует одноименному предложению в языке SQL. Так как язык XQuery проектировался разработчиками приложений, а не баз данных, под конструкцией `For` на самом деле подразумевается конструкция `For Each`, которая предполагает последовательное прохождение по всем элементам. Это раскрывает не пакетный характер мышления в XML.

Предложение Let используется для присвоения значения переменным, которые в дальнейшем будут участвовать в запросе XQuery. В SQL Server 2005 ключевое слово Let не поддерживается.

Назначение предложения Where идентично его назначению в SQL — оно ограничивает элементы и атрибуты XML, отбираемые запросом.

Предложение Order By также будет интуитивно понятно программистам SQL.

Предложение Return специфично для языка XQuery. Этот креативный синтаксис используется для конфигурирования формата данных XML, возвращаемых запросом XQuery.

```
SELECT Customer, OrderDetail.query(
  'for $i in /Items/Item
   where $i/Quantity = 2
   return $i/SKU')
  FROM XMLOrders
```

Как будет показано в следующем разделе, запросы XQuery используют в функциях реляционных запросов.

Слияние XQuery с инструкцией SELECT

Запросы XQuery в SQL Server слегка отличаются от стандарта. Причиной этому стало то, что стандарт XQuery создавался для извлечения данных из файлов XML, в то время как методы XQuery в SQL Server предназначены для интеграции в инструкции SELECT.

Метод exist проверяет наличие заданного узла, аналогично синтаксису IF EXISTS языка SQL:

```
SELECT Customer
  FROM XMLOrders
  WHERE OrderDetail.exist(
  '/Items/Item[SKU = 234]')=1
```

Метод exist возвращает значение 1, если узел существует, или 0 — в противном случае:

```
Name
------
Sam
```

Метод value возвращает фактическое значение узла; он может вернуть только скалярное значение:

```
SELECT Customer,OrderDetail.value(
'/Items[1]/Item[1]/SKU[1]', 'VARCHAR(25)')
FROM XMLOrders
```

Метод query позволяет запросу XQuery вернуть в реляционный запрос SELECT данные в полном объеме:

```
SELECT Customer, OrderDetail.query(
  'for $i in /Items/Item
   return $i/SKU')
  FROM XMLOrders
```

Метод nodes является, пожалуй, самым полезным при работе с XQuery. Он возвращает набор строк, который может быть объединен с реляционными данными в запросе SELECT.

Последний метод, modify, используется для изменения значений в переменных или столбцах с типом XML. Этот метод предназначен для использования в инструкции UPDATE и может изменять данные с помощью команд insert, delete и replace:

```
UPDATE XMLOrders
  SET OrderDetail.modify(
```

```
    'insert
    <Items SKU = "678" Quantity = "2"/>
     into /Items[1]'
    )
  WHERE OrderID = 1

UPDATE XMLOrders
  SET OrderDetail.modify(
    'delete /Items/Item[1]'
    )
  WHERE OrderID = 1
```

Декомпозиция данных XML в SQL Server

Одним из вариантов работы с данными XML является их декомпозиция с последующим сохранением элементов в реляционных таблицах. В настоящий момент SQL Server 2005 позволяет создавать и читать данные XML с помощью инструкции SELECT.

Чтение данных XML в SQL Server

Приложения читают данные XML с помощью процедуры синтаксического разбора, которая, в свою очередь, представляет данные XML объектной модели документа (DOM). Этот установленный консорциумом W3C стандарт является объектно-ориентированным представлением документа XML. Сам документ XML и каждый элемент, атрибут или текст в нем становится объектом DOM. Объектная модель документа является очень мощной и может использоваться в объектно-ориентированном программном коде для создания, чтения и модификации документа XML.

SQL Server использует процедуру разборки XML и объектную модель документа от Microsoft для чтения документа XML в два этапа.

1. Хранимая процедура `sp_xml_preparedocument` читает документ XML с помощью алгоритма синтаксического разбора MSXML, после чего создает внутренние объекты DOM в SQL Server. Объекты DOM идентифицируются целыми числами, возвращаемыми этой хранимой процедурой.
2. В инструкции SQL DML в качестве источника данных используется функция `OpenXML`, которая идентифицирует объект DOM с помощью целого числа, возвращенного хранимой процедурой `sp_xml_preparedocument`.

В приведенном ниже примере мы вначале присваиваем данные XML локальной переменной `@XML`. После этого SQL Server читает данные в SQL, используя результаты предыдущих двух этапов.

1. В результате выполнения хранимой процедуры `sp_xml_preparedocument` создаются объекты DOM.
2. Инструкция `SELECT` обращается к системной функции `OpenXML` как к источнику данных. Эта функция принимает три параметра.
   - Целочисленный идентификатор внутреннего объекта DOM, который хранится в переменной `@iDOM`.
   - Шаблон документа XML `rowpattern`, который функция `OpenXML` использует для идентификации структуры элементов данных XML. В данном случае значением параметра `rowpattern` является `'/Tours/Tour/Event'`.

- Флаг конфигурации XML, определяющий, как элементы и атрибуты будут интерпретироваться функцией XML (табл. 31.1).

3. Параметр `WITH` приводит в действие механизм проверки соответствия столбцов с возвращаемом функцией `OpenXML` в результирующем наборе данных. Столбец определяется именем XML, типом данных и необязательным параметром местонахождением элемента.

Таблица 31.1. Флаги конфигурации функции `OpenXML`

Значение флага	Режим работы	Описание
0	По умолчанию	По умолчанию ищет атрибуты
1	Основанный на атрибутах	Функция ищет атрибуты
2	Основанный на элементах	Функция ищет элементы
8	Комбинированный	Функция ищет атрибуты, а затем элементы

Пакет завершается вызовом хранимой процедуры `sp_removedocument`, которая удаляет объектную модель документа из памяти:

```
DECLARE
  @iDOM int,
  @XML VarChar(8000)

Set @XML = '
<?xml version="1.0" encoding="UTF-8"?>
<Tours>
<Tour Name="Amazon Trek">
  <Event Code="01-003" DateBegin="2001-03-16T00:00:00"/>
  <Event Code="01-015" DateBegin="2001-11-05T00:00:00"/>
</Tour>
<Tour Name="Appalachian Trail">
  <Event Code="01-005" DateBegin="2001-06-25T00:00:00"/>
  <Event Code="01-008" DateBegin="2001-07-14T00:00:00"/>
  <Event Code="01-010" DateBegin="2001-08-14T00:00:00"/>
</Tour>
<Tour Name="Bahamas Dive">
  <Event Code="01-002" DateBegin="2001-05-09T00:00:00"/>
  <Event Code="01-006" DateBegin="2001-07-03T00:00:00"/>
  <Event Code="01-009" DateBegin="2001-08-12T00:00:00"/>
</Tour>
<Tour Name="Gauley River Rafting">
  <Event Code="01-012" DateBegin="2001-09-14T00:00:00"/>
  <Event Code="01-013" DateBegin="2001-09-15T00:00:00"/>
</Tour>
<Tour Name="Outer Banks Lighthouses">
  <Event Code="01-001" DateBegin="2001-02-02T00:00:00"/>
  <Event Code="01-004" DateBegin="2001-06-06T00:00:00"/>
  <Event Code="01-007" DateBegin="2001-07-03T00:00:00"/>
  <Event Code="01-011" DateBegin="2001-08-17T00:00:00"/>
  <Event Code="01-014" DateBegin="2001-10-03T00:00:00"/>
  <Event Code="01-016" DateBegin="2001-11-16T00:00:00"/>
</Tour>
</Tours>'

-- Генерирование внутренней объектной модели документа
EXEC sp_xml_preparedocument @iDOM OUTPUT, @XML
```

```
-- Поставщик OPENXML
SELECT *
  FROM OPENXML (@iDOM, '/Tours/Tour/Event',8)
  WITH ([Name] VARCHAR(25) '../@Name',
    Code VARCHAR(10),
    DateBegin DATETIME
    )
EXEC sp_xml_removedocument @iDOM
```

Будут получены следующие результаты (сокращенно):

```
Name                 Code     DateBegin
-------------------- -------- -------------------------
Amazon Trek          01-003   2001-03-16 00:00:00.000
Amazon Trek          01-015   2001-11-05 00:00:00.000
Appalachian Trail    01-005   2001-06-25 00:00:00.000
Appalachian Trail    01-008   2001-07-14 00:00:00.000
...
```

Создание документов XML в SQL Server 2005

В SQL Server 2005 можно создавать документы XML непосредственно в запросах. Необязательный суффикс `FOR XML` инструкции `SELECT` дает указание серверу форматировать результат запроса как документ XML, а не стандартный набор данных SQL.



Вывод XML направляется в один столбец. По умолчанию параметр `maximum size per column` анализатора запросов установлен в слишком малое значение, чтобы можно было просмотреть результирующий набор данных в формате XML. Во вкладке Results диалогового окна Options этот параметр можно установить в значение `8192`. Открывается диалоговое окно параметров с помощью пункта меню Tools⇨Options.

Суффикс `FOR XML` позволяет сформировать данные XML, но не сам раздел объявлений корневого элемента `root`. Чтобы получить хорошо сформированный документ XML, приложению нужно "облачить" данные, полученные от SQL Server, в корректную оболочку.

Режим FOR XML RAW

Суффикс `FOR XML` имеет три режима: `RAW`, `AUTO` и `ELEMENTS`. Режим `FOR XML RAW` сбрасывает строки результирующего набора данных в документ XML, не генерируя при этом какую-либо иерархическую структуру. Каждая строка SQL становится элементом `raw` документа XML:

```
SELECT Tour.Name, Event.Code, Event.DateBegin
  FROM Tour
  JOIN Event
    ON Tour.TourID = Event.TourID
  FOR XML RAW
```

Будет получен следующий результат (сокращенно):

```
<row Name="Amazon Trek" Code="01-003"
  DateBegin="2001-03-16T00:00:00"/>
<row Name="Amazon Trek" Code="01-015"
  DateBegin="2001-11-05T00:00:00"/>
<row Name="Appalachian Trail" Code="01-005"
  DateBegin="2001-06-25T00:00:00"/>
<row Name="Appalachian Trail" Code="01-008"
  DateBegin="2001-07-14T00:00:00"/>
```

```
<row Name="Appalachian Trail" Code="01-010"
  DateBegin="2001-08-14T00:00:00"/>
...
```

Режим FOR XML AUTO

Режим AUTO определяет все иерархии в структуре данных и генерирует более адекватный документ XML. Пример документа XML, приведенный в начале настоящей главы, создан с помощью следующего запроса:

```
SELECT Tour.Name, Event.Code, Event.DateBegin
  FROM Tour
  JOIN Event
    ON Tour.TourID = Event.TourID
  FOR XML AUTO
```

Параметр ELEMENTS указывает режиму FOR XML AUTO генерировать элементы вместо атрибутов. Следующий вариант документа XML использует параметр ELEMENTS для генерирования исключительно элементов:

```
SELECT Tour.Name, Event.Code, Event.DateBegin
  FROM Tour
  JOIN Event
    ON Tour.TourID = Event.TourID
  FOR XML AUTO, ELEMENTS
```

Будет получен следующий результат (сокращенно):

```
<Tour>
  <Name>Amazon Trek</Name>
  <Event>
    <Code>01-003</Code>
    <DateBegin>2001-03-16T00:00:00</DateBegin>
  </Event>
  <Event>
    <Code>01-015</Code>
    <DateBegin>2001-11-05T00:00:00</DateBegin>
  </Event>
</Tour>
<Tour>
  <Name>Appalachian Trail</Name>
  <Event>
    <Code>01-005</Code>
    <DateBegin>2001-06-25T00:00:00</DateBegin>
  </Event>
  <Event>
    <Code>01-008</Code>
    <DateBegin>2001-07-14T00:00:00</DateBegin>
  </Event>
  <Event>
    <Code>01-010</Code>
    <DateBegin>2001-08-14T00:00:00</DateBegin>
  </Event>
</Tour>
...
```



Служба интеграции SQL Server 2005 также поддерживает импорт и экспорт данных XML. Более подробно эту службу мы рассмотрим в главе 42.

Резюме

Согласитесь вы с идеей хранения XML в базе данных или нет, XML прочно вошел в нашу жизнь, и знание этой технологии критично для любого профессионала баз данных. Архитектура, ориентированная на службы, частично базируется на концепции самоописательных документов XML. XQuery рассматривается как ключевая технология взаимодействия с данными XML. XML перестал быть словесным заменителем текстовых данных с разделителями — он стал *обобщенным языком* обмена данными.

В следующей главе мы продолжим дискуссию об использовании XML в SQL Server и рассмотрим Web-службы и оболочки SOAP как метод транспортировки данных XML.

Создание хранилищ данных SOA с помощью Web-служб

ГЛАВА

32

В этой главе...

Web-службы

Оболочки SOAP

Архитектура, ориентированная на службы (SOA), предлагает целый спектр возможностей. Чтобы это понять, достаточно оценить количество конференций и преобладание сессий, посвященных разным аспектам SOA. СУБД SQL Server 2005 содержит массу революционных нововведений, однако ее способность работать с SOA можно по праву считать главным из них.

По своей сути архитектура, ориентированная на службы, отражает философию разработки программного обеспечения, которая делает основную ставку на инкапсуляцию бизнес-логики на уровень приложения, при этом приложения свободно объединяются, используя общий метод взаимодействия.

- Термин *инкапсуляция на уровне приложений* подразумевает, что интерфейс, или контракт, между двумя приложениями монолитен. Ни одному из взаимодействующих приложений нет никакой необходимости знать что-либо о внутренней работе своего партнера. При этом какие-либо изменения в реализации функций одного приложения никак не отражаются на работе другого.
- Термин *общий интерфейс* подразумевает, что одно приложение может запросить данные у любого другого приложения, используя один и тот же протокол. Обычно это значит, что Web-службы и встроенный протокол SOAP (Simple Object Acess Protocol)/XML являются предпочтительным методом взаимодействия приложений, ориентированных на архитектуру SOA.

SQL Server 2005 располагает рядом технологий, разработанных специально для архитектуры SOA.

- Концевые точки HTTP и SOAP.
- Брокер служб. По своей природе приложения SOA должны быть хорошо масштабируемыми, и это — основное требование. Для достижения нужной масштабируемости в состав SQL Server был введен брокер служб — внутренняя очередь асинхронных сообщений, предназначенная для обработки больших информационных потоков.
- Тип данных XML и обработка запросов XQuery.
- Поддержка технологии .NET в SQL Server.

Создание витрины данных SOA начинается с определения *контракта*, или интерфейса, который должно поддерживать приложение. В практических терминах это подразумевает определение сообщений и параметров, которые будут поставляться Web-службами. После этого создается хранимая процедура, которая будет получать эти сообщения, и Web-служба, которая будет прослушивать наличие сообщений.



Так как существует множество методов подключения к базе данных, возникает вопрос, какой же из них использовать: ADO или Web-службы. В этом отношении существует одно простое и четкое правило. Если подключение клиентов .NET к базе данных осуществляется по локальной сети, следует использовать ADO. В то же время, если подключение выполняется через Интернет с прохождением через ряд брандмауэров, лучшим решением будет использование Web-служб. Если реализация задачи предполагает использование обоих типов подключения, подумайте об использовании в обоих случаях Web-служб, однако решение принимайте с учетом требований к нагрузке и производительности. Web-службы, даже при работе с ними по локальной сети, работают медленнее, так как в работе задействовано множество уровней и преобразований; в то же время программирование и обслуживание уровня доступа к данным будет более простым.

В этой главе будут описаны концевые точки SQL Server 2005 и методы их конфигурирования. Однако сами Web-службы являются всего лишь одной стороной системы взаимодействия. Чтобы полностью их понять, нужно рассмотреть, как само приложение использует Web-службы.



Чтобы ознакомиться с демонстрационным пакетом, иллюстрирующим взаимодействие с Web-службами, загрузите его с сайта книги.

Прослушивание HTTP

Как вы уже, наверное, знаете, протокол передачи гипертекста (HTTP) используется в Интернете для передачи запросов и доставки страниц HTML в браузеры. Технология Web-служб вывела этот протокол на новую орбиту, задействовав его для запроса и доставки информационных сообщений между разными системами.



С выходом пакета обновлений SP1 подключения по протоколу HTTP поддерживают сжатие.

Концевой точкой является прослушивающий узел, или адрес, используемый для получения запросов к Web-службам по протоколу HTTP. Сервер Web-служб может обслуживать множество концевых точек, каждая из которых может содержать несколько методов, или типов сообщений, которые может обработать.

Web-служба является концевой точкой вместе с ассоциированным кодом приложения и определением способа обмена данными. Вообще-то, концевой точкой является только сама Web-служба, но она обладает логикой поведения.



Чтобы иметь возможность получать данные Web-службы, нужно написать определенный программный код. В Visual Studio включен ряд примеров программ, предназначенных для работы с Web-службами. Используя только браузер, вы можете увидеть код WSDL, однако не сможете выполнить метод или получить данные от Web-службы SQL Server.

Процесс HTTP.sys

Раньше, для того чтобы создать Web-службу для данных SQL Server, необходимо было построить несколько уровней: IIS, программный код .NET, систему обслуживания очереди сообщений и объекты ADO для подключения к SQL Server на стороне клиента. Каждый из этих слоев вносил свой вклад в общую производительность системы.

Гораздо более эффективным способом реализации Web-службы является использование высокопроизводительного прослушивания запросов HTTP, встроенного в новое поколение операционных систем. На уровне ядра систем Windows Server 2003, Windows XP и Windows Vista запущен процесс `HTTP.sys`, который прослушивает запросы HTTP. SQL Server 2005 использует концевые точки процесса `HTTP.sys`.



Windows Server 2003 использует сервер IIS версии 6.0, который, в свою очередь, использует `HTTP.sys` в качестве службы прослушивания. Таким образом, в этой конфигурации не возникает конфликтов между одновременным прослушиванием серверами IIS и SQL Server. В то же время Windows XP SP2 продолжает использовать IIS 5.1, который осуществляет прослушивание самостоятельно, без использования службы `HTTP.sys`. Это значит, что если вам нужно создать Web-службу в Windows XP, то следует либо отключить сервер IIS, либо сконфигурировать порты так, чтобы избежать конфликта между IIS и `HTTP.sys`, которые по умолчанию прослушивают один и тот же порт с номером 80.

Неявные концевые точки

В SQL Server 2005 существуют два метода создания концевых точек: явный и неявный. Неявный метод использует инструкцию DDL `CREATE ENDPOINT` и позволяет управлять созданием самому серверу баз данных. Этот метод как раз и рекомендуется использовать:

```
CREATE ENDPOINT sql_endpointtest
STATE = STARTED
AS HTTP(
  PATH = '/sql',
  AUTHENTICATION = (INTEGRATED),
  PORTS = (CLEAR),
  SITE = '*' , CLEAR_PORT = 20000
  )
FOR SOAP (
  WEBMETHOD 'http://tempUri.org/'.'GetSqlInfo'
         (name='master.dbo.xp_msver',
          SCHEMA=STANDARD),
  WSDL = DEFAULT,
  SCHEMA = STANDARD ,
  DATABASE = 'master'
  );
```

Инструкция CREATE ENDPOINT имеет два раздела: транспортный, определяющий, как будет прослушиваться концевая точка, и раздел наполнения, содержащий детали методов и сообщений.

Транспортный раздел инструкции CREATE ENDPOINT, начинающийся с ключевого слова AS, конфигурирует путь к концевой точке, метод аутентификации и порты. В приведенном выше примере методом транспортировки был выбран HTTP (а не TCP/IP).

Порт может быть сконфигурирован как свободный (clear) или защищенный (SSL). Свободные порты требуют указания номера порта в параметре Clear_Port.

Раздел наполнения определяет содержимое концевой точки, обычно это FOR SOAP. В этом разделе также определяются Web-методы, тип WSDL, схема и база данных.

Раздел методов является критичным — именно в нем концевая точка подключается к хранимым процедурам или очередям сообщений брокера служб. В приведенном выше примере в разделе методов был определен вызов хранимой процедуры. Одна концевая точка может быть связана с несколькими различными методами.

Явные концевые точки

Альтернативой создания подключения к HTTP.sys самим сервером является жесткое программирование (конфигурирование) этого подключения с использованием явной концевой точки. Лично я рекомендую всегда использовать неявные концевые точки и отдавать управление подключениями к HTTP.sys на откуп серверу баз данных.

Конфигурирование HTPP.sys

Файл конфигурации HTTPcfg отлично подходит для опроса процессов, которые прослушивают HTTP.sys, и для управления прослушиванием подключений. Когда HTTPcfg обращается к ACL, он использует список управления доступом (Access Control List). Это утилита командной строки, входящая в состав операционной системы Windows Server 2003, однако вы можете загрузить ее копию и для использования в Windows XP. Эта утилита также включена в инструментарий MSFT Service Toolkit — windowsxp-kb838079-supporttools-enu.exe, — который можно загрузить с сайта корпорации Microsoft.

WSDL

Язык описаний Web-служб WSDL является стандартным средством, встроенным практически в любую Web-службу. Он используется клиентом для обнаружения методов и сообщений, доступных в конкретной Web-службе. SQL Server и HTTP.sys обеспечивают поддержку WSDL в трех возможных вариантах: стандартном, упрощенном и выборочном. Пожалуй, стандартный вариант можно назвать лучшим, несмотря на то, что он словесный.

После создания концевой точки протестируйте ее наличие и активность. Проще всего для этого в Web-браузере опросить WSDL. В Internet Explorer используйте адрес URL http://*сервер*:*порт*/path?wsdl (рис. 32.1).

Упрощенный вариант WSDL является более компактной версией WSDL.

Для создания выборочного WSDL существует системная хранимая процедура, которая возвращает стандартный WSDL, который может быть вызван из хранимой процедуры выборочного WSDL для генерирования базового WSDL. После этого стандартный WSDL может быть освобожден или дополнен нестандартной информацией, которую требует ваша архитектура SOAP.

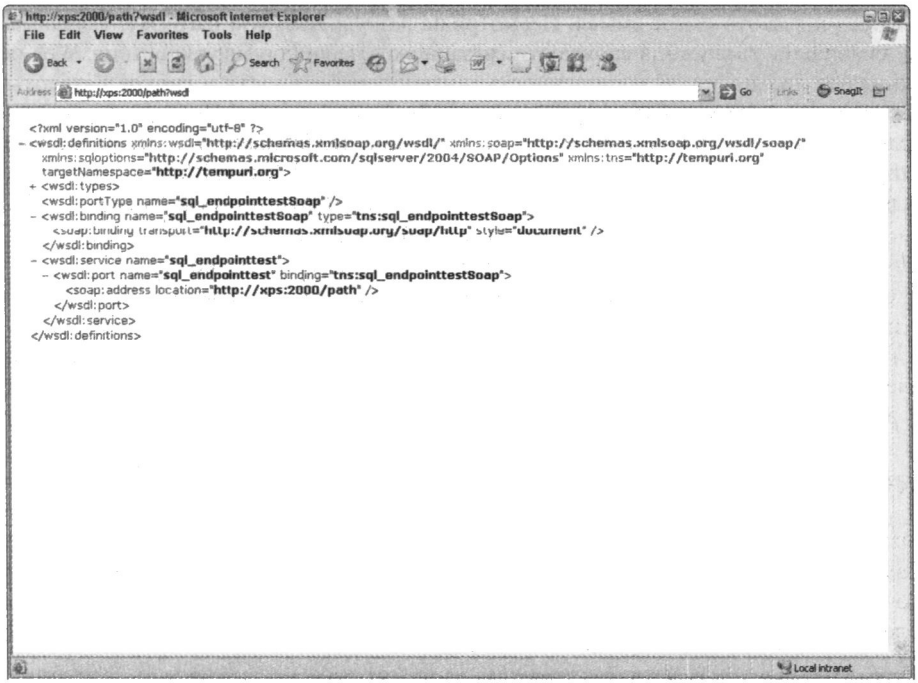


Рис. 32.1. Просмотр WSDL в Internet Explorer путем перехода к серверу по пути, предложенному Web-службой

Защита концевых точек

При передаче данных через Интернет основное внимание следует уделять вопросам безопасности. Web-службы SQL Server 2005 имеют двухуровневую защиту. Вначале права доступа проверяются на транспортном уровне библиотекой `HTTP.sys` путем аутентификации, а затем повторная проверка выполняется уже самим сервером. Ни один из этих уровней не обеспечивает анонимный доступ.

Аутентификация на транспортном уровне HTTP может быть сконфигурирована в режимах Basic, Digest или Integrated (NTLM, Kerberos).

Регистрационные данные SQL Server передаются в пакете SOAP, создаваемом взаимодействующим со службой приложением. Так как пакет может быть зашифрован, нет необходимости отправлять регистрационные данные в открытом виде. Если вы используете смешанную регистрацию SQL Server и пакет SOAP аутентифицируется в SQL Server с использование регистрационной записи SQL Server, для шифрования данных важно использовать защищенные подключения.

Резюме

Несмотря на то что большинство операций доступа к данным до сих пор осуществляется с помощью ADO.NET, мне кажется, что Web-службы являются альтернативным вариантом архитектурного решения, который требует глубокого понимания и должен использоваться

в ситуациях, когда это имеет смысл. Несмотря на дополнительную нагрузку, накладываемую использованием Web-служб, и сложности связанного с ними конфигурирования, Web-службы обеспечивают повышенную защищенность, интероперабельность и гибкое развертывание в Интернете.

В следующей главе мы расширим список вариантов подключения к данным и рассмотрим InfoPath — технологию Microsoft для доставки данных в офис. Сами решайте, как относиться к этой технологии, однако в любом случае с ней нужно познакомиться поближе.

InfoPath и SQL Server 2005

ГЛАВА

33

В этой главе...

Обзор InfoPath 2003

Создание шаблона формы

Прочие функции InfoPath

Дополнительные источники информации

Основными задачами приложений, осуществляющих доступ к SQL Server 2005, являются извлечение информации из баз данных, ее эффективное отображение пользователям и поддержка хранения производственной информации.

Доступ к этой информации обычно осуществляется с помощью форм. Многие пользователи, работающие с информацией, проводят большую часть рабочего дня за заполнением форм. Некоторые из данных, отображаемых и вводимых в этих формах, берут свое начало в базах данных SQL Server. В контексте производственных условий, в которых быстро изменяются требования к информационному обеспечению, очень важной является возможность быстро создавать и гибко адаптировать пользовательские формы.

Существует несколько подходов к созданию форм, используемых для ввода и отображения данных SQL Server, в том числе формы Windows и Web-формы. Программа Microsoft InfoPath 2003 предлагает альтернативный способ создания таких форм.

Эту главу можно рассматривать как введение в программу InfoPath 2003. В ней будут описаны некоторые средства этой программы и порядок их использования в работе с SQL Server.

Обзор InfoPath 2003

InfoPath 2003 представляет собой графический инструмент разработки форм, основанных на XML. В этой программе можно достаточно легко создавать множество типов форм, в то же время она оснащена рядом более сложных функций. Программа InfoPath 2003 доступна в составе пакета Microsoft Office 2003 Professional Enterprise Edition, а также как обособленный продукт. Она предоставляет возможность пользователям, не знакомым с языком XML, создавать XML-формы, используя стандартные приемы перетаскивания объектов. Одним из источников данных, с которыми может работать InfoPath, является SQL Server.



Все темы, рассматриваемые в этой главе, относятся к функциям программы InfoPath 2003 с установленным пакетом обновлений SP1. Этот пакет внес в программу множество функций, которые не были доступны в ее исходной версии.

В InfoPath конструктор или разработчик создает *шаблон формы*. Пользователь открывает шаблон как форму, в которую вводятся данные. Форма может сохраняться на сервере (для последующей передачи) или передаваться напрямую на сервер. В отличие от традиционных Web-форм, которые передают на сервер пары "имя–значение", InfoPath передает хорошо сформированный документ XML.



Шаблон формы имеет расширение `.xsn`. Локально сохраненный документ XML, созданный из шаблона, имеет расширение `.xml`.



На момент время написания книги пробная версия InfoPath 2003 была доступна на сайте

`www.microsoft.com/office/infopath/prodinfo/trial.mspx`

Автоматическая проверка данных

Одним из основных достоинств InfoPath 2003 является возможность проверки данных на стороне клиента до отправки документа XML (созданного из шаблона) на сервер. Проверка выполняется с помощью технологии XML Schema, разработанной консорциумом W3C. Нет никакой необходимости писать для проверки собственный код JScript или JavaScript.

Проверка выполняется сразу же после того, как пользователь вышел из некоторого элемента управления формы. Для указания на отсутствующие, но необходимые данные, и на данные, которые введены, но не корректны, существуют различные визуальные оповещения.

Если форма содержит некорректные данные, пользователь не может ее отправить на сервер. Исходя из этого, все ошибки должны быть устранены до отправки формы. В то же время не полностью заполненная форма и форма, содержащая некорректные данные, может быть сохранена на локальном компьютере для последующего дополнения или коррекции. После того как будут введены все необходимые данные, а все ошибки устранены, форма может быть отправлена на сервер.

Заполнение форм в автономном режиме

В мире, где все большее число рабочих проводят существенную часть времени вне офиса, будучи отключенными от Интернета, использование технологий форм, требующих наличия постоянного подключения, становится мало реалистичным. Программа InfoPath предлагает практическое решение для сценариев с непостоянным подключением.

Когда пользователь подключен к серверу (например, перед командировкой), на его компьютер может быть загружен *шаблон формы*. Из этого шаблона на локальном компьютере может быть создана одна или несколько пустых форм. В сценарии работы торгового представителя последний должен сохранить хотя бы по одной форме для каждого клиента, которого собирается посетить до следующего подключения к Интернету. Впоследствии, после того, как пользователю снова станет доступен Интернет или другое сетевое соединение, данные всех заполненных форм смогут быть отправлены на сервер.

Условное форматирование

Программа InfoPath 2003 поддерживает условное форматирование отдельных элементов управления формы или их группы в *разделе*. Условным называют форматирование, изменяющееся в зависимости от данных, введенных в форму. Условное форматирование управляет отображением отдельных частей формы. Оно не зависит от того, корректны введенные данные или нет.

Одним из применений условного форматирования является сокрытие элементов управления, которые не соответствуют конкретной ситуации. Бумажные формы обычно решают эту проблему путем объединения сходных элементов в блоки. Рано или поздно пользователь, заполняющий форму, должен каким-то образом определить, что часть формы не применима в данной ситуации. В формах InfoPath, если отображать только подходящие элементы формы, это только повысит опыт пользователя. Например, в медицинской анкете не имеет смысла отображать вопросы о беременности, если опрашивается мужчина. Этот вопрос уместен только в том случае, если в поле пола был выбран пункт "женский". Аналогично, в форме прайс-листа уместно отображать товары, запас которых вышел за пределы минимально допустимого, красным цветом.

Условное форматирование может быть также использовано для выделения информации, на которую пользователь должен обратить внимание или которая требует немедленных действий. Если, скажем, форма InfoPath используется для записи результатов обследования техники безопасности на объекте, те значения, которые вписываются в существующие нормы, должны отображаться одним цветом, а выходящие за пределы норм — другим.

Вопросы защиты в InfoPath 2003

Приложение InfoPath 2003 использует модель системы безопасности программы Internet Explorer. И это имеет определенный смысл, поскольку шаблон формы InfoPath может загружаться с Web-сервера или отправляться пользователю по электронной почте. Так как эти методы доставки открывают возможность проникновения в шаблон формы вредоносного кода, формы InfoPath обычно запускаются в изолированном кэше, не имеющем полного доступа к ресурсам системы компьютера конечного пользователя.



Если вы не знакомы с моделью системы безопасности Internet Explorer, то можете ознакомиться со статьей на сайте

```
http://msdn.microsoft.com/library/default.asp?url=/library/
en-us/ipsdk/html/ipsdkFormSecurityModel_HV01083562.asp
```

В то же время InfoPath также имеет концепцию *полностью доверительной формы*, иногда называемую просто *доверительной формой*. Доверительная форма специально устанавливается на компьютере пользователя. Обычно она устанавливается явно, используя Microsoft Windows Installer, перед тем, как пользователь получит дополнительные разрешения. Доверительные формы отображаются в разделе Custom Installed Forms интерфейса пользователя InfoPath.

Кэшированные формы InfoPath уникально идентифицируются по URL или URN. Доверительные формы идентифицируются исключительно по URN. Формы, в которых адрес URL, идентифицирующий ее, находится в том же домене, что и компьютер пользователя, принадлежат зоне корпоративной интрасети. Если URL формы находится в домене, отличном от компьютера пользователя, некоторые аспекты ее работы могут оказаться недоступными в интересах защиты от вредоносного кода. Кэшированные формы, идентифицируемые по URN, устанавливаются, например, с помощью Microsoft Windows Installer. Такие формы трактуются как принадлежащие зоне локального компьютера и, таким образом, получают полный доступ к компьютеру пользователя.

Возможность установки полностью доверительных форм может показаться заманчивой, однако, ввиду того, что такие формы имеют полный доступ к компьютеру пользователя, стоит дважды подумать перед их инсталляцией.

Ниже перечислены некоторые специфические функции системы безопасности программы InfoPath.

- **Защита шаблона формы**. Пользователь не может изменить шаблон формы при ее заполнении, однако он может модифицировать шаблон формы, открыв его в режиме конструктора. Новосозданный шаблон будет иметь иной адрес URL.
- **Цифровая подпись**. Информация, введенная в форму, может быть защищена цифровой подписью. Это обеспечивает уверенность в том, что форма была подписана и указывает на подписавшего ее.
- **Возможность отключения доверительных форм**. Эта функция пресекает запуск пользователем форм, имеющих полный доступ к ресурсам его компьютера.

Объектная модель InfoPath

Программа InfoPath имеет объектную модель, которой может воспользоваться программист. Эта объектная модель отражает иерархию узлов документа XML, с которым работает InfoPath, а также шаблон текущей формы. Так как форма InfoPath может быть использована в различных сценариях, очень важно то, что некоторые методы и свойства доступны только для полностью доверительных форм.

Сценарии и программный код .NET

В исходной версии InfoPath 2003 поддерживал сценарии в Microsoft Script Editor, использующий код JScript и JavaScript. С выходом пакета обновлений SP1 он стал дополнительно поддерживать программирование, основанное на платформе .NET.

Примечание

Чтобы обеспечить совместимость с законом Сарбанеса-Оксли, компания Microsoft выпустила Office Solution Accelerator. Если вас заинтересовало это приложение, то можете посетить сайт `http://msdn.microsoft.com/office/understanding/SOX/default.aspx`

Создание шаблона формы

Создание шаблона формы является первым этапом проектирования. Здесь возможно несколько вариантов действий. Приведенное ниже описание относится к созданию шаблона формы, которая подключается к серверу баз данных SQL Server.

Для создания шаблона формы выполните следующие действия.

1. Запустите InfoPath 2003.
2. Выберите в меню пункт File⇨Design a Form, чтобы открыть панель задач конструктора форм. По умолчанию она размещается в правой части экрана.
3. На открывшейся панели конструктора форм выберите пункт Data Connection.
4. На первой странице открывшегося мастера подключений к данным выберите принятый по умолчанию вариант — Database (Microsoft SQL Server and Microsoft Office Access only).

5. В следующем окне мастера щелкните на кнопке Select Database.
6. В открывшемся окне мастера источников данных выберите пункт `New SQL Server Connection.odc` и щелкните на кнопке Open.
7. В окне подключения к серверу баз данных введите имя компьютера сервера, к которому хотите подключиться.
8. Выберите метод аутентификации, который хотите использовать. В данном случае доступны два варианта: интегрированная защита Windows и аутентификация SQL. Если выбрать последний вариант, то для подключения к базе данных нужно дополнительно ввести регистрационное имя и пароль.
9. Щелкните на кнопке Next. Откроется диалоговое окно выбора базы данных и таблиц.
10. В раскрывающемся списке выберите базу данных, к которой хотите подключиться. В примере, показанном на рис. 33.1, выбрано подключение к базе данных `pubs`.



Учебную базу данных `pubs`, входившую в комплект установки SQL Server 2000, можно загрузить по адресу:

`www.microsoft.com/downloads/details.aspx?FamilyId=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en`

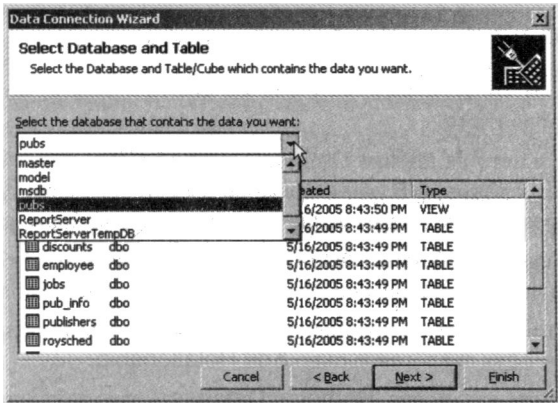


Рис. 33.1. Выберите базу данных, содержащую нужную информацию

11. После выбора нужной базы данных в нижней части диалогового окна Select Database and Table отобразится список таблиц и представлений, содержащихся в ней. В нем вы можете выбрать объект, к которому будет выполняться подключение. В примере, показанном на рис. 33.2, была выбрана таблица `authors` схемы `dbo`.



В мастере подключений к данным имеется ссылка на владельца, хотя это понятие было в версии SQL Server 2005 заменено понятием схемы. Это связано в тем, что на момент выхода в свет программы InfoPath 2003 самой свежей версией SQL Server была 2000. На тот момент у разработчиков программы еще не было специальных знаний о схемах SQL Server 2005.

12. Щелкните на кнопке Next. Откроется диалоговое окно сохранения подключения к данным. При желании вы можете заменить имя файла подключения, предложенное по умолчанию. Введите описание созданного подключения и щелкните на кнопке Finish. Подключение будет сохранено в файле с расширением `.odc`. Откроется следующее

окно мастера подключения к данным, показанное на рис. 33.2. Обратите внимание на то, что таблица `authors` отображается как `authors.dbo`, а не в форме, принятой в SQL Server 2005, — `dbo.authors`.

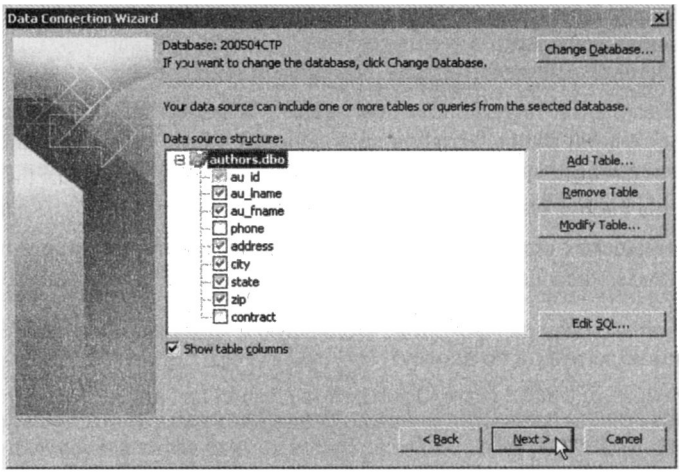


Рис. 33.2. Выберите столбцы таблицы

13. В зависимости от типа запроса, который вы хотите создать, можете воспользоваться следующими вариантами.
    - Если вы щелкнете на кнопке Change Database, вам будет предложено изменить подключение к данным. В данном случае вы сможете либо создать новое подключение, повторив приведенные выше действия, либо выбрать уже существующее.
    - Если вы щелкнете на кнопке Add Database, то сможете выбрать еще одну таблицу, находящуюся в дочернем подчинении уже выбранной таблицы.
14. Далее вы можете определить отношение между выбранными родительской и дочерней таблицами, используя диалоговое окно Edit Relationship.
    - Если щелкнуть на кнопке Remove Database, то выбранная родительская база данных будет удалена. Чтобы заменить ее другой, щелкните на кнопке Add Database и выберите альтернативную базу данных.
    - Если щелкнуть на кнопке Modify Database, появится возможность отсортировать данные, извлекаемые из базы, используя значения вплоть до трех столбцов. Флажок в нижней части диалогового окна поможет определить, следует ли отображать в форме несколько записей, извлеченных из базы.
    - Если щелкнуть на кнопке Edit SQL, отобразится созданная программой InfoPath инструкция извлечения данных; при желании вы можете ее отредактировать. Созданную инструкцию можно протестировать с помощью кнопки Test SQL Statement.



Программа InfoPath поддерживает ограниченный диапазон относительно простых запросов T-SQL. При редактировании инструкции, созданной мастером, будьте внимательны, чтобы случайно не создать запрос, с которым программа InfoPath не сможет справиться.

Редактируя текст инструкции T-SQL или используя графический интерфейс, вы можете создать запросы, с которыми программа InfoPath не сможет справиться; в результа-

те запрос не сможет быть отправлен на сервер. Обратите внимание на то, что диалоговое окно некорректно использует термин "база данных" для обращения к серверу, на котором размещена база данных SQL Server.

Кроме ситуации отношения "один ко многим", функциональность отправки запроса может быть отключена в следующих ситуациях.

- Когда пользователь не имеет достаточных разрешений на изменение данных в соответствующих таблицах.
- Когда для извлечения данных используется хранимая процедура.
- Когда в запросе участвуют типы данных, которые программа InfoPath не поддерживает (например, `image`, `text` и `ntext`).

В данном случае мы предполагаем, что вам не нужен в работе ни один из вариантов, перечисленных выше.

15. Щелкните на кнопке Next. В следующем окне важно проверить, включена ли функциональность отправки запроса.
16. Щелкните на кнопке Finish. Откроется конструктор программы InfoPath, в котором будет отображен скелет формы. Обратите внимание на панель Data Source, находящуюся в правой части окна. В разделе Data содержатся папки полей запроса и полей формы. Поля запроса используются при извлечении данных из базы. Поля данных используются для отправки данных в базу. Простейшим способом проектирования формы, предназначенной одновременно и для просмотра, и для изменения данных, является перетаскивание папки из области Data Source в соответствующее место рабочей области конструктора. При перетаскивании вам будет предложено два варианта: Section with Controls и Section. Выберите первый из них.
17. Замените текст заголовка `Authors Form`. Щелкните на подзаголовке, чтобы удалить его. Отредактируйте подписи полей, чтобы они выглядели так, как показано на рис. 33.3.
18. Сохраните созданный шаблон формы под именем `AuthorsForm.xsn`.
19. Щелкните на кнопке Preview Form панели инструментов. В режиме предварительного просмотра в текстовое поле Author Lastname введите **`Gringlesby`**. Щелкните на кнопке Run Query. В элементах управления формы в нижней части экрана отобразятся данные Берта Гринглзби (Burt Gringlesby).

    Если для выбранного имени существует несколько записей, то все они будут отображены в нижней части представления формы.

Прочие функции InfoPath

Программа InfoPath содержит таблицы компоновки, позволяющие получить более точную раскладку страницы, чем в приведенном выше примере. Чтобы получить доступ к таблицам компоновки, щелкните на кнопке Layout панели Data Source.

Чтобы иметь под рукой полный ассортимент элементов управления формы, поддерживаемых программой InfoPath, щелкните на кнопке Controls панели Data Source. Среди доступных элементов вы найдете текстовые поля, флажки, кнопки, разделы и т.п.

Каждый элемент формы имеет связанные с ним свойства. Чтобы открыть свойства текстового поля, щелкните на соответствующем элементе правой кнопкой мыши и выберите в контекстном меню пункт Textbox Properties. Откроется диалоговое окно свойств текстового поля.

Диалоговое окно Textbox Properties позволяет прикрепить к элементу правила и проверку.

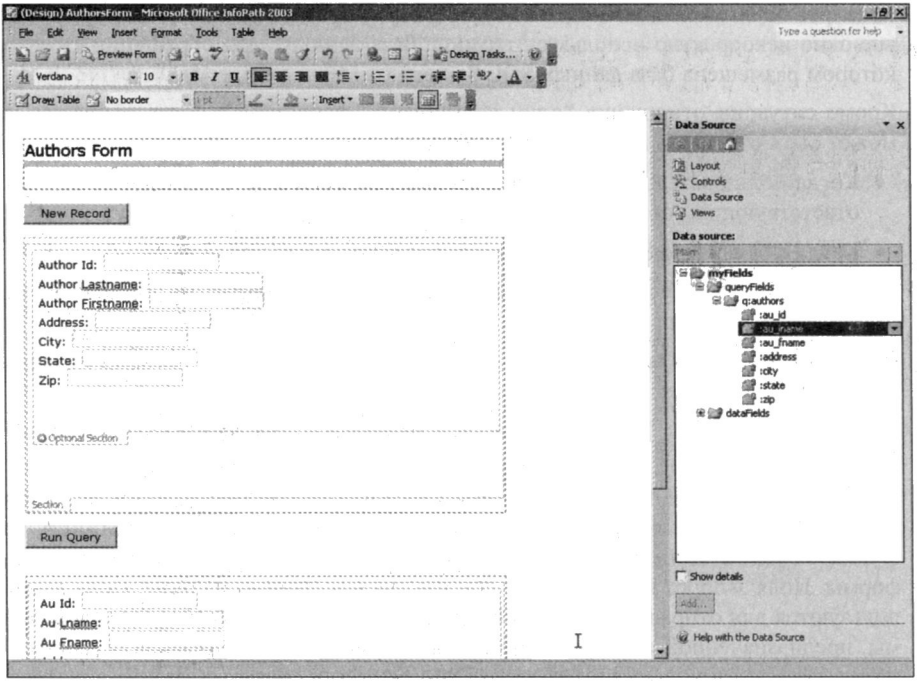


Рис. 33.3. Шаблон формы с добавленной функциональностью выполнения запроса

Пример шаблона формы, созданный в настоящей главе, имеет единственное представление, однако на практике формы обычно имеют несколько представлений. Например, отдельные преставления могут быть созданы для запроса к базе данных и для занесения в базу новой или исправленной информации.



Для получения более подробной информации о программе InfoPath 2003 можно использовать справочную панель, которая открывается с помощью комбинации клавиш <Ctrl+F1>.



Наиболее удобно работать с отдельными файлами справки. Файл общей справки по программе содержится в файле

```
C:\Program Files\Microsoft Office\Office 11\1033\InfMain.chm
```

Руководство программиста можно найти в файле

```
C:\Program Files\Microsoft Office\Office11\1033\InfRef.chm
```

Резюме

Эта глава является вводным экскурсом в программу InfoPath применительно к ее взаимодействию с SQL Server. В ней были описаны основные инструменты продукта, а также процедура создания шаблона формы. Настоящая глава завершила рассмотрение вопросов, связанных с программированием в T-SQL. В следующей части мы рассмотрим вопросы управления данными предприятия.

Управление данными на уровне предприятия

ЧАСТЬ

IV

В этой части...

Глава 34
Конфигурирование SQL Server

Глава 35
Перенос баз данных

Глава 36
Планирование восстановления

Глава 37
Обслуживание базы данных

Глава 38
Автоматизация обслуживания баз данных с помощью SQL Server Agent

Глава 39
Репликация данных

Глава 40
Защита баз данных

Глава 41
Администрирование SQL Server Express

В части IV мы поговорим о том, какую роль играет администратор базы данных масштаба предприятия.

Проект нельзя считать завершенным, когда производственная база данных передается в эксплуатацию. Успешная база данных требует как профилактического обслуживания (настройки), так и сопровождения (тщательного планирования процедур восстановления).

Принцип информационной архитектуры, сформулированный в главе 1, гласил, что "*информация... должна быть... защищена и сделана доступной в удобном формате для ежедневных операций и анализа отдельными сотрудниками, их группами и процессами как сегодня, так и в будущем*".

Несмотря на более высокую, чем когда-либо ранее степень автоматизации SQL Server и обманчивые заявления Microsoft, будто бы SQL Server — СУБД, не требующая администрирования, истина заключается в том, что необходимо тратить огромные усилия на поддержание производственной базы данных в работоспособном состоянии 24 часа в день, 7 дней в неделю, 365 дней в году.

ГЛАВА

34

Конфигурирование SQL Server

В этой главе...

Основы конфигурирования SQL Server

Установка параметров конфигурации уровня сервера

Установка параметров конфигурации уровня базы данных

Установка параметров конфигурации уровня подключения

СУБД SQL Server 2005 изобилует различными параметрами конфигурации. Сложность управления ими заключается в том, что они рассредоточены по трем уровням.

- Параметры уровня сервера определяют общую конфигурацию взаимодействия сервера с оборудованием, а также значения по умолчанию, принятые для всех баз данных.
- Параметры уровня базы данных определяют режим работы базы, а также значения по умолчанию, принятые для всех подключений.
- Параметры уровня подключения определяют режим работы в контексте конкретного подключения или хранимой процедуры.

Некоторые параметры замещают своих функциональных "собратьев", находящихся на уровень ниже, или определяют для них значения по умолчанию. В этой главе мы объединим все три уровня конфигурирования в единую понятную модель их взаимодействия друг с другом.

Установка параметров

Решите ли вы устанавливать параметры в графическом интерфейсе утилиты Management Studio или с помощью программного кода — зависит исключительно от ваших предпочтений. В то же время следует отметить, что не все параметры доступны в графическом интерфейсе Management Stuidio и не все можно установить с помощью запросов. В то время как графический интерфейс прост в использовании и имеет понятные диалоговые окна, в которых собраны взаимосвязанные параметры, ему недостает повторяемости сценариев T-SQL.



Чтобы просмотреть дополнительную информацию о компьютерной системе для ее использования в процессе конфигурирования сервера, выберите `sysdm_os_sys_info` в динамическим представлении управления.

Конфигурирование сервера

Параметры конфигурации уровня сервера управляют свойствами, распространяющимися на весь сервер баз данных. Эти свойства определяют характер взаимодействия SQL Server с оборудованием, организацию потоков в операционной системе Windows, а также возможность триггеров вызывать другие триггеры. При конфигурировании сервера помните о главной цели настройки — согласованности и производительности.

В графическом интерфейсе большинство параметров сервера можно сконфигурировать с помощью диалогового окна Server Properties. Для того чтобы открыть это окно, щелкните правой кнопкой мыши на имени сервера в дереве консоли Management Studio и выберите в контекстном меню пункт Properties. Во вкладке General этого окна (рис. 34.1) будут отображены версия сервера и общие параметры его среды.

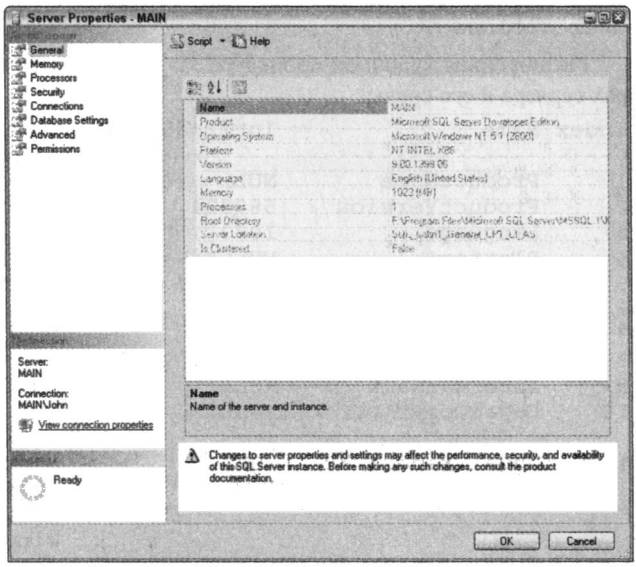
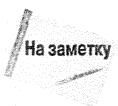


Рис. 34.1. Вкладка General диалогового окна Server Properties утилиты Management Studio

Ту же информацию можно получить и программным путем. Например, версия сервера может быть идентифицирована с помощью глобальной переменой `@@Version`:

```
Select @@Version
----------------------------------------------------
Microsoft SQL Server 2005 - 9.00.1399.06 (Intel X86)
  Oct 14 2005 00:33:37
  Copyright (c) 1988-2005 Microsoft Corporation
  Developer Edition on Windows NT 5.1 (Build 2600: Service Pack 2)
```



Многие параметры конфигурации не вступают в силу до перезагрузки SQL Server. По этой причине даже после выполнения изменений во вкладке General диалогового окна Server Properties будут отображаться текущие установки.

Многие параметры сервера можно установить с помощью системной хранимой процедуры `sp_configure`. Если при вызове этой процедуры не устанавливать параметры, она выдает отчет о текущих настройках сервера, как в следующем примере:

```
EXEC sp_configure
```

Будет получен следующий результат (используются стандартные параметры):

```
name                        minimum maximum     config_value run_value
--------------------------  ------- -------     ------------ ---------
allow updates               0       1           0            0
clr enabled                 0       1           0            0
cross db ownership chaining 0       1           0            0
default language            0       9999        0            0
max text repl size (B)      0       2147483647  65536        65536
nested triggers             0       1           1            1
remote access               0       1           1            1
remote admin connections    0       1           0            0
remote login timeout (s)    0       2147483647  20           20
remote proc trans           0       1           0            0
remote query timeout (s)    0       2147483647  600          600
server trigger recursion    0       1           1            1
show advanced options       0       1           0            0
user options                0       32767       0            0
```

Расширенная хранимая процедура `xp_msver` выводит отчет о дополнительных параметрах сервера и его среды:

```
Index Name                Internal_Value Character_Value
----- ------------------- -------------- ------------------------
1     ProductName         NULL           Microsoft SQL Server
2     ProductVersion      589824         9.00.1399.06
3     Language            1033           English (United States)
4     Platform            NULL           NT INTEL X86
5     Comments            NULL           NT INTEL X86
6     CompanyName         NULL           Microsoft Corporation
7     FileDescription     NULL           SQL Server Windows NT
8     FileVersion         NULL           2005.090.1399.00
9     InternalName        NULL           SQLSERVR
10    LegalCopyright      NULL           (c) Microsoft Corp. All
                                         rights reserved.
11    LegalTrademarks     NULL           Microsoft(r) is a
                                         registered trademark of
                                         Microsoft Corporation.
                                         Windows(TM) is a
                                         trademark of Microsoft
                                         Corporation
12    OriginalFilename    NULL           SQLSERVR.EXE
13    PrivateBuild        NULL           NULL
14    SpecialBuild        91684864       NULL
15    WindowsVersion      170393861      5.1 (2600)
16    ProcessorCount      1              1
17  ProcessorActiveMask   1              00000001
18    ProcessorType       586            PROCESSOR_INTEL_PENTIUM
19    PhysicalMemory      1023           1023 (1073201152)
20    Product ID          NULL           NULL
```

Системная функция `ServerProperty` предлагает еще один механизм получения информации о сервере. Преимущество этого метода заключается в том, что функция может использоваться в составе выражения в инструкции `SELECT`. В следующем примере эта функция используется для получения информации о редакции данного экземпляра SQL Server:

```
SELECT ServerProperty('Edition')
```

Будет получен следующий результат:

```
Developer Edition
```

Конфигурирование базы данных

Параметры конфигурации уровня базы данных определяют режим работы базы данных в части ее совместимости со стандартом ANSI и модели восстановления.

Большую часть этих параметров можно установить в диалоговом окне Database Properties утилиты Management Studio. Чтобы открыть это окно, щелкните правой кнопкой мыши на имени базы данных в дереве консоли Management Studio и выберите в контекстном меню пункт Properties. Страница Options этого диалогового окна показана на рис. 34.2.

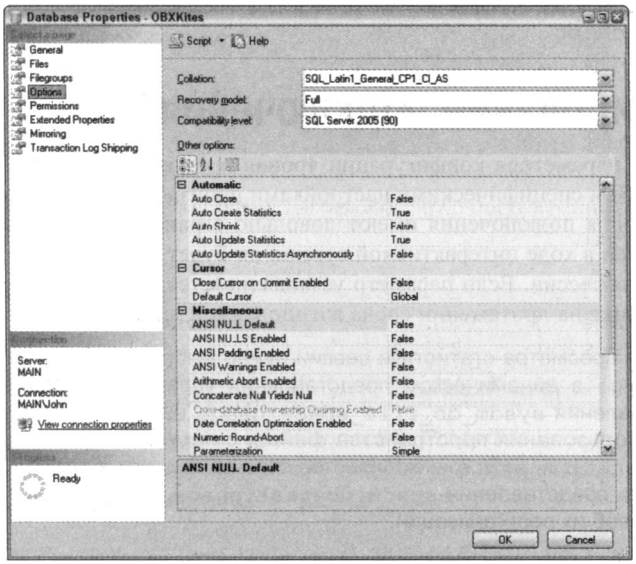


Рис. 34.2. Страница Options диалогового окна Dapabase Properties может быть использована для конфигурирования основных параметров базы данных

Параметры конфигурации базы данных можно установить и программным путем с помощью системной хранимой процедуры `sp_dboption`. При ее запуске без аргументов отобразится список всех доступных настроек базы данных:

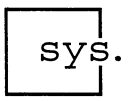
```
EXEC sp_dboption
Settable database options:
----------------------------------
ANSI null default
ANSI nulls
ANSI padding
ANSI warnings
arithabort
auto create statistics
auto update statistics
autoclose
autoshrink
concat null yields null
cursor close on commit
db chaining
dbo use only
default to local cursor
```

```
merge publish
numeric roundabort
offline
published
quoted identifier
read only
recursive triggers
select into/bulkcopy
single user
subscribed
torn page detection
trunc. log on chkpt
```

Конфигурирование подключения

Бо́льшая часть параметров конфигурации уровня приложения связана с совместимостью со стандартом ANSI и специфическими настройками производительности.

Параметры уровня подключения имеют довольно ограниченную область действия. Если параметр установлен в ходе интерактивной сессии, он будет активизирован до своего изменения или завершения сессии. Если параметр устанавливается в хранимой процедуре, то он сохраняет свое значение на протяжении срока жизни данной процедуры.



Для просмотра статистики сессии нужно выбрать `sysdm_db_session_space_usage` в динамическом представлении управления. Также с помощью представления `sysdm_db_file_space_usage` вы можете просмотреть статистику использования пространства файла, с помощью представления `sysdm_db_partition_stats` — количество страниц и строк в каждом разделе, а с помощью представления `sysdm_db_task_space_usage` — размещение страниц и способ их реорганизации.

Параметры уровня подключения обычно конфигурируют с помощью команды `SET`. В следующем примере конфигурируется режим работы SQL Server с пустыми значениями в текущей сессии:

```
SET Ansi_nulls Off
```

Будет получен следующий результат:

```
The command(s) completed successfully.
```

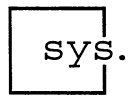
Свойства подключения можно проверить с помощью функции `SessionProperty()`:

```
Select SessionProperty ('ANSI_NULLS')
```

Результат будет следующим:

```
0
```

Некоторые параметры подключения устанавливаются в Management Studio в момент соединения с сервером баз данных. Доступ к этим параметрам открывается щелчком на ссылке View Connection Properties в диалоговых окнах параметров таких объектов, как сервер и база данных. На рис. 34.3 показано окно свойств подключения, открываемое из окна параметров сервера.



Чтобы просмотреть статистику текущей сессии, выберите `sysdm_exec_sessions` в динамическом представлении управления.

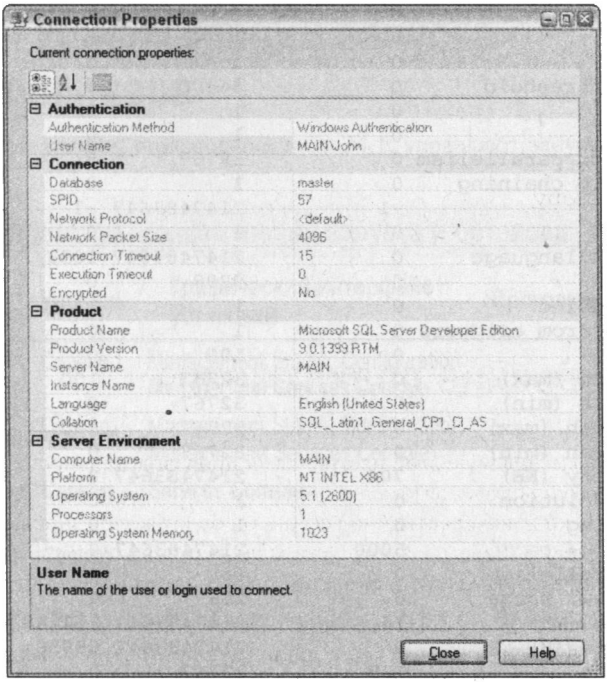


Рис. 34.3. Диалоговое окно Connection Properties *можно использовать для просмотра параметров уровня подключения, действующих в текущей сессии*

Параметры конфигурации

Так как множество сходных параметров конфигурации управляется на разных уровнях (сервера, базы данных и подключения) разными командами, в этом разделе мы организуем эти параметры по назначению, а не по алфавиту или уровню.

Отображение расширенных свойств

Перед тем как начать работу с множеством доступных параметров SQL Server, сервер нужно сконфигурировать для их отображения. Для выполнения этой задачи следует включить отображение дополнительных параметров с помощью следующего кода:

```
EXEC sp_configure 'show advanced options', 1
RECONFIGURE
```

После этого выполнение процедуры `sp_configure` приведет к отображению списка всех дополнительных параметров.

```
EXEC sp_configure
```

```
name                            minimum     maximum     config_value run_value
------------------------------- ----------- ----------- ------------ -----------
Ad Hoc Distributed Queries      0           1           0            0
affinity I/O mask               -2147483648 2147483647  0            0
affinity mask                   -2147483648 2147483647  0            0
```

```
Agent XPs                       0           1           1           1
allow updates                   0           1           0           0
awe enabled                     0           1           0           0
blocked process threshold       0           86400       0           0
c2 audit mode                   0           1           0           0
clr enabled                     0           1           0           0
cost threshold for parallelism  0           32767       5           5
cross db ownership chaining     0           1           0           0
cursor threshold                -1          2147483647  -1          -1
Database Mail XPs               0           1           0           0
default full-text language      0           2147483647  1033        1033
default language                0           9999        0           0
default trace enabled           0           1           1           1
disallow results from triggers  0           1           0           0
fill factor (%)                 0           100         0           0
ft crawl bandwidth (max)        0           32767       100         100
ft crawl bandwidth (min)        0           32767       0           0
ft notify bandwidth (max)       0           32767       100         100
ft notify bandwidth (min)       0           32767       0           0
index create memory (KB)        704         2147483647  0           0
in-doubt xact resolution        0           2           0           0
lightweight pooling             0           1           0           0
locks                           5000        2147483647  0           0
max degree of parallelism       0           64          0           0
max full-text crawl range       0           256         4           4
max server memory (MB)          16          2147483647  2147483647  2147483647
max text repl size (B)          0           2147483647  65536       65536
max worker threads              128         32767       0           0
media retention                 0           365         0           0
min memory per query (KB)       512         2147483647  1024        1024
min server memory (MB)          0           2147483647  0           0
nested triggers                 0           1           1           1
network packet size (B)         512         32767       4096        4096
Ole Automation Procedures       0           1           0           0
open objects                    0           2147483647  0           0
PH timeout (s)                  1           3600        60          60
precompute rank                 0           1           0           0
priority boost                  0           1           0           0
query governor cost limit       0           2147483647  0           0
query wait (s)                  -1          2147483647  -1          -1
recovery interval (min)         0           32767       0           0
remote access                   0           1           1           1
remote admin connections        0           1           0           0
remote login timeout (s)        0           2147483647  20          20
remote proc trans               0           1           0           0
remote query timeout (s)        0           2147483647  600         600
Replication XPs                 0           1           0           0
scan for startup procs          0           1           1           1
server trigger recursion        0           1           1           1
set working set size            0           1           0           0
show advanced options           0           1           1           1
SMO and DMO XPs                 0           1           1           1
SQL Mail XPs                    0           1           0           0
transform noise words           0           1           0           0
two digit year cutoff           1753        9999        2049        2049
user connections                0           32767       0           0
user options                    0           32767       0           0
Web Assistant Procedures        0           1           0           0
xp_cmdshell                     0           1           0           0
```

Параметры конфигурации запуска и останова сервера

Параметры конфигурации, описанные в табл. 34.1, управляют запуском процессов в SQL Server.

Таблица 34.1. Параметры конфигурации запуска и останова

Параметр	Уровень	Графический интерфейс установки	Программная установка
Автозапуск SQL Server при загрузке	Сервер	Management Studio, Configuration Manager или Services Console	—
Автозапуск MS DTC при загрузке	Сервер	Management Studio, Configuration Manager или Services Console	—
Автозапуск SQL Server Agent при загрузке	Сервер	Services Console	—
Отображение дополнительных параметров	Сервер	—	`EXEC sp_configure 'show advanced options'`
Сканирование процедур автозапуска	Сервер	—	`EXEC sp_configure 'scan for startup procs'`
Автозапуск MS DTC при загрузке	Сервер	Management Studio, Configuration Manager или Services Console	—

Параметры автозапуска

Параметры автозапуска используются службой SQL Server. Эти параметры аналогичны тем, которые передаются в программу при ее запуске в командной строке DOS. Для использования этих параметров следует остановить службу SQL Server; после чего параметры нужно ввести в поле Start Parameters вкладки General (рис. 34.4).

После ввода параметров нужно снова запустить службу. Кроме стандартных параметров места размещения базы данных `master`, еще два параметра могут оказаться особенно полезными.

- `-m`. Запуск SQL Server в монопольном режиме. Этот параметр необходим для воссоздания и перестройки поврежденной базы данных `master`. Во время работы базы данных в монопольном режиме избегайте использования утилиты Management Studio.
- `-x`. Отключение регистрации статистики процессора и кэша для достижения максимальной производительности.

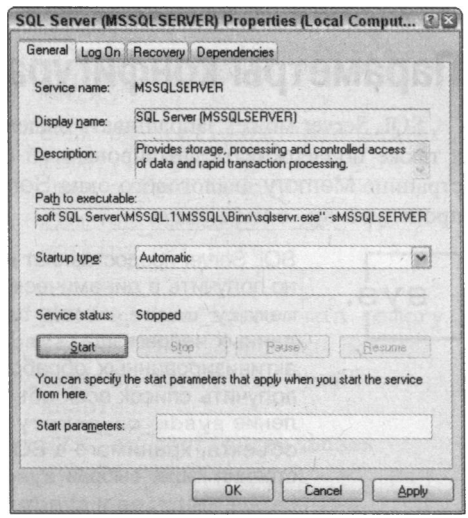
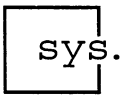


Рис. 34.4. Для изменения режима работы службы SQL Server введите параметры автозапуска в поле Start Parameters



Для регистрации счетчиков производительности в системе выберите `sysdm_os_performance_counters` в динамическом представлении управления. Также вы можете просмотреть список всех модулей, загруженных в систему, выбрав в динамическом представлении управления `sysdm_os_loaded_modules`.

При желании вы можете использовать следующие дополнительные параметры.

- `-d`. Используется для включения полного пути к файлу Master.
- `-e`. Используется для включения полного пути к файлу Errors.
- `-c`. Запуск SQL Server не в режиме службы Windows.
- `-f`. Запуск сервера в минимальной конфигурации.
- `-g`. Определение объема виртуальной памяти, доступной хранимым процедурам в SQL Server (в мегабайтах).
- `-n`. Отключение протоколирования в журнале событий Windows.
- `/Trace#`. Включение флагов трассировки по их номеру.

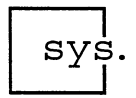
Автозапуск хранимых процедур

Два дополнительных параметра сервера не представлены в интерфейсе Management Studio, в то же время они доступны с помощью программного кода. SQL Server можно настроить для сканирования при каждом запуске хранимой процедуры автозапуска, подобно тому, как операционная система Windows использует файл `autoexec.bat`. У хранимой процедуры автозапуска нет какого-либо фиксированного имени, к тому же таких процедур может быть несколько. Для создания хранимой процедуры автозапуска запустите системную хранимую процедуру `sp_procoption` и пометьте нужную процедуру. В дальнейшем вы можете управлять процедурой автозапуска, включая и отключая параметр сервера `scan for startup procs`:

```
EXEC sp_configure 'scan for startup procs', 1
RECONFIGURE
```

Параметры конфигурации памяти

SQL Server может запрашивать выделение памяти у операционной системы динамически, а также поддерживать фиксированный объем памяти. Эти параметры конфигурируются на странице Memory диалогового окна Server Properties (рис. 34.5) или с помощью хранимой процедуры `sp_configure`.



SQL Server предоставляет массу информации, относящейся к памяти, которую можно получить в динамических представлениях управления. Выберите `sysdm_os_memory_cache_clock_hands`, чтобы получить состояние каждого таймера заданных часов кэша; `sysdm_os_memory_clercs`, чтобы получить набор всех активизированных обработчиков памяти; `sysdm_os_memory_objects`, чтобы получить список всех объектов, размещенных в памяти SQL Server. Представление `sysdm_os_memory_pools` возвращает отдельную строку для каждого объекта, хранимого в SQL Server. Также вы можете получить статистику состояния кэша, выбрав `sysdm_os_memory_cache_counters`, `sysdm_os_memory_cache_entries` и `sysdm_os_memory_cache_hash_tables`.

Параметры конфигурации памяти, приведенные в табл. 34.2, управляют использованием и выделением памяти в SQL Server.

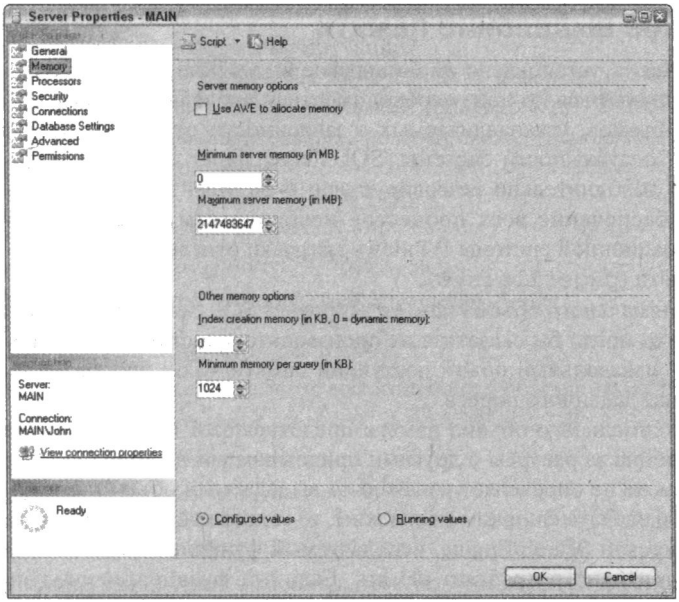


Рис. 34.5. Страница Memory диалогового окна Server Properties

Таблица 34.2. Параметры конфигурации памяти

Параметр	Уровень	Графический интерфейс установки	Программная установка
Минимальный объем динамической памяти	Сервер	Management Studio	`EXEC sp_configure 'min server memory'`
Максимальный объем динамической памяти	Сервер	Management Studio	`EXEC sp_configure 'max server memory'`
Фиксированный объем памяти	Сервер	Management Studio	`EXEC sp_configure 'min server memory'` и `EXEC sp_configure 'max server memory'`
Резервирование физической памяти для SQL Server	Сервер	Management Studio	`EXEC sp_configure 'set working set size'`
Минимальная память, выделяемая запросу	Сервер	Management Studio	`EXEC sp_configure 'min memory per query'`
Включение AWE	Сервер	—	`EXEC sp_configure 'AWE Enabled'`
Память, выделяемая для создания индекса	Сервер	—	`EXEC sp_configure 'index create memory'`
Блокировки	Сервер	—	`EXEC sp_configure 'locks'`
Максимальный размер реплицируемого текста	Сервер	—	`EXEC sp_configure 'max text repl size'`
Открытие объектов	Сервер	—	`EXEC sp_configure 'open objects'`

Динамическое выделение памяти

Если в SQL Server установлено динамическое выделение памяти, то ее объем может сокращаться и увеличиваться по мере необходимости, в пределах ограничений минимального и максимального объемов, устанавливаемых в зависимости от объема доступной физической памяти и общей загруженности системы. SQL Server будет стараться обеспечить необходимые требования, дополнительно резервируя еще 4–10 Мбайт. Главной задачей сервера при этом является обеспечение всех процессов необходимым объемом физической памяти, не требуя от операционной системы Windows выгрузки отдельных страниц в файл поддержки виртуальной памяти (`pagefile.sys`).

Параметр минимального объема памяти защищает SQL Server от чрезмерного сокращения объема памяти, что могло бы сказаться на производительности. В то же время этот параметр не устанавливает изначальный объем доступной памяти — он просто не допустит уменьшения ее объема ниже заданного порога.

Параметр максимального объема памяти предотвращает ее выход за предел, за которым может начаться война за ресурсы с другими приложениями и самой операционной системой. Этот параметр также не определяет изначально выделяемый объем памяти. Если максимальный объем памяти сделать слишком маленьким, то может пострадать производительность.

Механизм Microsoft Search Engine, используемый функцией полнотекстового поиска SQL Server, также интенсивно потребляет память. Если эта функция используется относительно часто, обеспечьте ее достаточным объемом памяти. Компания Microsoft при этом рекомендует воспользоваться следующей формулой:

Общая_виртуальная_память – (*Максимальная_виртуальная_память_SQL_Server* + *Виртуальная_память_для_других_служб*)≥ 1,5 * *Физическая_память*

Например, если в сервере, имеющем объем памяти 196 Мбайт, 96 Мбайт отведено для SQL Server и 64 Мбайт — для механизма Search Engine, общая виртуальная память должна быть больше 288 Мбайт (физическая память, умноженная на 1,5) плюс 160 Мбайт (плановая память SQL Server и MS Search) — т.е. 448 Мбайт. Так как сервер располагает 192 Мбайт физической памяти, файл поддержки виртуальной памяти должен быть по объему не меньше 252 Мбайт. Другими словами, чем бо́льшим объемом физической памяти располагает сервер, тем больший по объему требуется файл подкачки.



Чтобы просмотреть список активизированных каталогов полнотекстового поиска, в динамическом представлении управления выберите `sysdm_fts_active_catalogs`. Если вы нуждаетесь в информации о пулах памяти, используемых как часть полнотекстового поиска или диапазона поиска, выберите в динамическом представлении управления `sysdm_fts_memory_pools`. Если вам нужна информация о буферах памяти, принадлежащих заданному пулу памяти, выберите `sysdm_fts_memory_buffers`.

Лично я конфигурирую динамическую память, устанавливая минимальный объем в 16 Мбайт и приравнивая максимальный объем к общему объему памяти компьютера, за вычетом 128 Мбайт. Это резервирует минимальный объем памяти для SQL Server, позволяя ему увеличиваться по мере необходимости. Одновременно это гарантирует наличие резервных 128 Мбайт для нужд операционной системы и предотвращает возможность войны за ресурсы с Windows при выполнении особо больших запросов. Если ваш компьютер имеет в наличии больше ресурсов, то вы можете оставить для операционной системы и больший объем памяти.

Различные экземпляры SQL Server конкурируют друг с другом, когда им нужна память. В серверах с несколькими развернутыми экземплярами SQL Server высока вероятность того, что два особо загруженных экземпляра начнут войну за ресурсы, в результате которой один из

них останется обделенным памятью. Уменьшение максимального объема памяти в каждом из экземпляров поможет избежать подобных ситуаций.

В программном коде T-SQL параметры минимального и максимального объемов памяти устанавливаются с помощью системной хранимой процедуры sp_configure. Этот параметр принадлежит к группе дополнительных, поэтому может использоваться только в том случае, когда включен параметр сервера show advanced options:

```
EXEC sp_configure 'show advanced options', 1
EXEC sp_configure 'min server memory', 16
```

Будет получен следующий результат:

```
Configuration option 'min server memory (MB)'
  changed from 0 to 16.
Run the RECONFIGURE statement to install.
```

Аналогично устанавливается и параметр максимального объема памяти:

```
EXEC sp_configure 'max server memory', 128
```

Будет получен следующий результат:

```
Configuration option 'max server memory (MB)'
  changed from 128 to 128.
Run the RECONFIGURE statement to install.
```

Для автоматизации вычисления максимального объема памяти на основе объема физической памяти создана следующая хранимая процедура. В ней анализируется набор данных, возвращаемый хранимой процедурой xp_msver, и на основе найденного в нем объема физической памяти вычисляется значение, которое затем передается процедуре sp_configure для установки параметра:

```
CREATE PROC pSetMaxMemory (
  @Safe INT = 64 )
AS
  CREATE TABLE #PhysicalMemory (
    [Index] INT,
    [Name] VARCHAR(50),
    [Internal_Value] INT,
    [Character_Value] VARCHAR(50) )
  DECLARE @Memory INT
  INSERT #PhysicalMemory
    EXEC xp_msver 'PhysicalMemory'
  SELECT @Memory =
    (Select Internal_Value FROM #PhysicalMemory) - @safe
  EXEC sp_configure 'max server memory', @Memory
  RECONFIGURE

go

EXEC pSetMaxMemory -- установка макс. объема памяти - 64Mb
EXEC pSetMaxMemory 32 -- установка макс. объема памяти - 32Mb
```

Команда RECONFIGURE

После того как параметры конфигурации были изменены с помощью хранимой процедуры sp_configure, для того, чтобы они вступили в силу, нужно выполнить команду RECONFIGURE. Если этого не сделать, в поле config_value будет отображаться измененный вариант значения, однако изменения не затронут поле run_value, даже если перезагрузить службу. В то же время некоторые параметры для своего вступления в силу требуют перезапуска SQL Server:

```
RECONFIGURE
The command(s) completed successfully.
```

Вместо динамического потребления памяти SQL Server можно сконфигурировать так, чтобы у операционной системы немедленно запрашивалось выделение фиксированного объема памяти. Чтобы установить фиксированный объем памяти из программного кода, нужно задать для параметров минимального и максимального объемов памяти одно и то же значение.

Несмотря на кажущуюся дополнительную нагрузку, вызываемую вычислением необходимого объема памяти, опросом среды и запросом к операционной системе, на практике вы не почувствуете никаких различий в производительности системы при переключении между динамическим и фиксированным выделением памяти. Основной целью выделения фиксированного объема памяти является конфигурирование выделенного компьютера SQL Server, не допускающее подкачку страниц. При этом настройку фиксированной памяти нужно объединить с рядом других параметров, о которых мы поговорим немного позже.

Независимо от объема памяти, выделяемой операционной системой серверу баз данных, диспетчер памяти Windows может выгрузить некоторые страницы SQL Server в файл подкачки, если SQL Server простаивает. Если в настройках SQL Server установлен фиксированный объем памяти, то такой выгрузки можно избежать, если установить в параметре `Reserve Physical Memory` значение `true`.

Параметр SQL Server `Reserve Physical Memory` можно установить программным путем с помощью передачи системной хранимой процедуре `sp_configure` аргумента `set working set size`:

```
EXEC sp_configure 'set working set size', 1
RECONFIGURE
```

Чтобы данная настройка вступила в силу, сервер баз данных нужно перегрузить.

Временами команда разработчиков SQL Server поражает меня детализацией элементов управления базой данных, которые передаются в руки администратора. При необходимости SQL Server может выделять каждому выполняемому запросу заданный объем памяти. Параметр `min memory per query` позволяет установить минимальный объем памяти, используемый каждым запросом. Несмотря на то что увеличение значения этого параметра относительно выделяемых по умолчанию 1 Мбайт может обеспечить определенное повышение производительности некоторых запросов, я не вижу оснований заменять автоматическое управление памятью сервером и тем самым подвергаться риску нехватки памяти. В следующем примере минимальный объем памяти, выделяемый для запросов, повышается до 2 Мбайт:

```
EXEC sp_configure 'min memory per query', 2048
RECONFIGURE
```

Шесть дополнительных параметров конфигурирования памяти не доступны в интерфейсе Management Studio, но могут быть установлены с помощью программного кода.

Время ожидания запроса

Если доступной памяти недостаточно для выполнения большого запроса, SQL Server может подождать, пока не освободится нужный объем, в течение 25-кратного оценочного времени выполнения запроса. Если на протяжении этого промежутка времени нужный объем памяти так и не освободится, запрос завершится ошибкой превышения времени ожидания. В ходе ожидания все блокировки запроса сохраняются, что увеличивает вероятность возникновения взаимоблокировок. Если вы столкнулись именно с такой ситуацией, следует заложить в программный код предельное время ожидания выделения ресурсов памяти в секундах, как в следующем примере:

```
EXEC sp_configure 'query wait', 20
RECONFIGURE
```

В приведенном примере определено, что выполнение запроса либо должно начаться в течение 20 секунд, либо завершиться ошибкой превышения времени ожидания.

Расширение окна адресации памяти (AWE)

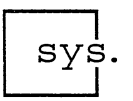
При обычных условиях СУБД SQL Server ограничена стандартным предельным объемом физической памяти в 3 Гбайт. В то же время редакция SQL Server 2005 Enterprise Edition, запущенная в среде Windows 2005 Datacenter, может оперировать 64 Гбайт физической памяти посредством API-интерфейса расширения окна адресации AWE. Параметр `AWE Enabled` позволяет включить в SQL Server AWE-адресацию памяти:

```
EXEC sp_configure 'AWE Enabled', 20
RECONFIGURE
```

Выделение памяти для формирования индексов

Объем памяти, который использует SQL Server для выполнения сортировки при создании индексов, обычно конфигурируется автоматически. В то же время этим объемом можно управлять и вручную, используя хранимую процедуру `sp_configure` и устанавливая нужное значение в килобайтах. В следующем примере память, выделяемая для создания индекса, определяется в размере 8 Мбайт:

```
EXEC sp_configure 'index create memory', 8096
RECONFIGURE
```



Для просмотра текущих операций ввода-вывода и блокировок в каждом разделе таблицы или индекса базы данных выберите в динамическом представлении управления `sysdm_db_index_operational_stats`. Статистику использования индексов можно увидеть с помощью `sysdm_db_index_usage_stats`, а физические детали, такие как статистика размеров и фрагментации, — с помощью `sysdm_db_index_physical_stats`.

Резервирование памяти для блокировок

Излишние блокировки могут в буквальном смысле поставить SQL Server на колени, как в отношении ожидания блокировок, так и в отношении памяти, потребляемой блокировками (одна блокировка использует 96 байт памяти). По умолчанию SQL Server резервирует для блокировок 2% отведенной для него памяти. В процессе работы это значение может быть динамически увеличено вплоть до 40% определенного в параметрах сервера максимального объема памяти. Чаще всего этого более чем достаточно. Если вы получили сообщение об ошибке, связанной с недостатком памяти для блокировок, не спешите увеличивать выделение для блокировок памяти — в данном случае проблема, скорее всего, связана с ошибками в программном коде. В следующем примере отключается динамическое выделение памяти для блокировок, при этом им отводится фиксированный объем памяти для 16767 блокировок (приблизительно 1,5 Мбайт):

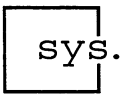
```
EXEC sp_configure 'locks', 16767
RECONFIGURE
```

Максимальное количество открытых объектов

SQL Server отдает предпочтение динамическому управлению своей памятью, включая пул, используемый для отслеживания объектов, открытых в текущий момент (таблиц, представлений, правил, хранимых процедур, значений по умолчанию и триггеров). Каждый объект представляет собой одну единицу выделения памяти, даже если ссылка на него выполняется многократно. SQL Server повторно использует пространство памяти в пуле объекта, однако однажды сервер может пожаловаться, что количество открытых объектов превысило установленный пре-

дел. В этом случае параметр максимального количества открытых объектов следует увеличить вручную. В следующем примере это количество увеличено до 16767 объектов:

```
EXEC sp_configure 'open objects', 16767
RECONFIGURE
```



Вы всегда можете узнать минимальное и максимальное значения конкретного параметра, если запустите на выполнение хранимую процедуру `sp_configure` с указание имени параметра, но без указания его значения. Например, если вы выполните команду `EXEC sp_configure 'open objects'`, то узнаете, что значение параметра `open objects` должно находиться в диапазоне от `0` до `2 147 483 647`.

Параметры конфигурации процессора

Параметры конфигурации процессора (табл. 34.3) управляют использованием симметричных многопроцессорных систем в SQL Server.

Таблица 34.3. Свойства конфигурации процессора

Параметр	Уровень	Графический интерфейс установки	Программная установка
Назначение потоков процессорам	Сервер	Management Studio	`EXEC sp_configure 'affinity mask'`
Максимальное количество рабочих потоков	Сервер	Management Studio	`EXEC sp_configure 'max worker threads'`
Повышение приоритета SQL Server в Windows	Сервер	Management Studio	`EXEC sp_configure 'priority boost'`
Использование волокон Windows NT	Сервер	Management Studio	`EXEC sp_configure 'lightweight pooling'`
Количество процессоров, используемых для параллельного выполнения запросов	Сервер	Management Studio	`EXEC sp_configure 'max degree of parallelsm'`
Минимальная стоимость запроса для его разделения на параллельные процессы	Сервер	Management Studio	`EXEC sp_configure 'cost threshold for parallelism'`
Время ожидания запроса	Сервер	Management Studio	`EXEC sp_configure 'query wait'`

Во вкладке Processors диалогового окна Server Proeprties (рис. 34.6) определяется режим работы сервера баз данных в симметричных многопроцессорных системах. Большая часть этих параметров не влияет на работу сервера в однопроцессорных компьютерах.

Назначение потоков процессорам

В многопроцессорных компьютерах операционная система может перемещать процессы на определенные процессоры, если того требует общая загрузка системы. Взаимосвязь процессов SQL Server и процессоров можно сконфигурировать. Формируя связь между SQL Server и процессором, можно определить доступность этого процессора для сервера баз данных; в то же время процессор не передается в монопольное использование SQL Server. Таким образом, пока процессор не назначен для запуска заданий SQL Server, он отделяется от сервера баз данных.

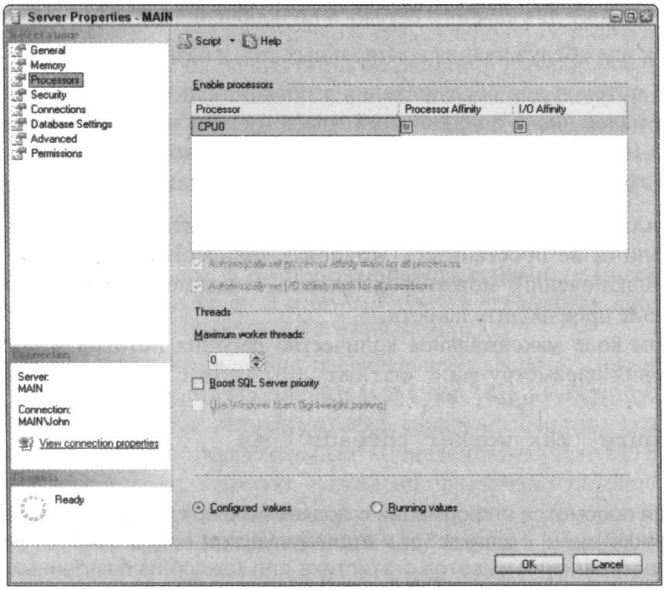


Рис. 34.6. На странице **Processors** *отображаются доступные в системе процессоры и определяется порядок их использования сервером баз данных*



Так как процесс переключения между процессами создает дополнительную нагрузку на систему, Windows получит определенные преимущества, если в системе существует один процессор, недоступный серверу баз данных. Если на сервере установлено восемь или более процессоров, рекомендуется вывести один из них из-под контроля SQL Server, чтобы системные процессы Windows не вступали в конкурентную борьбу с сервером баз данных за ресурсы компьютера.

В Management Studio взаимосвязь с процессорами конфигурируется посредством установки флажков во вкладке **Processors** (см. рис. 36.4).

В программном коде доступность отдельных процессоров для сервера баз данных конфигурируется установкой соответствующих битов в параметре маски родства `affinity mask` с помощью хранимой процедуры `sp_configure`. Поскольку в битовом представлении число 3 выглядит как `0000011`, в следующем примере серверу баз данных становятся доступными процессоры с номерами `0` и `1`:

```
EXEC sp_configure 'affinity mask', 3
RECONFIGURE
```



Нулевое значение параметра `affinity mask` определяет доступность серверу баз данных всех процессоров компьютера.

Максимальное число рабочих потоков

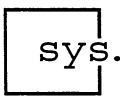
SQL Server является многопотоковым приложением. Это значит, что SQL Server для повышения производительности может распараллеливать процессы, разнося их выполнение на разные процессоры. Потоки проектируются следующим образом.

- По одному потоку для каждого сетевого подключения.
- Один поток для обслуживания контрольных точек базы данных.
- Несколько потоков для обслуживания запросов пользователей. Когда SQL Server обслуживает малое число подключений, каждому подключению выделяется один поток. Однако по мере увеличения количества подключений образуется пул потоков, который более эффективно обслуживает существующие подключения.

В зависимости от количества подключений и доли времени, в течение которого соединение активизировано (а не простаивает), установка количества рабочих потоков в значение, меньшее числа подключений, может форсировать создание пула подключений, экономию памяти и увеличение производительности.

В программном коде максимальное количество рабочих потоков устанавливается путем присвоения значения параметру `max worker threads` с помощью хранимой процедуры `sp_configure`:

```
EXEC sp_configure 'max worker threads', 64
RECONFIGURE
```



Для просмотра информации о подключениях к экземпляру SQL Server выберите `sysdm_exec_connections` в динамическом представлении управления. В этом представлении имеется статистика для каждой из баз данных, а также для каждого удаленного или локального подключения.

Повышение приоритета

Различные процессы в операционной среде Windows выполняются с разными уровнями приоритета, варьирующимися от 0 до 31. Процессы с бо́льшим уровнем приоритета выполняются первыми, поэтому эти уровни резервируются для процессов операционной системы. Обычно система Windows выделяет для приложений следующие уровни приоритетов: `4` (низкий), `7` (обычный), `13` (высокий) и `24` (реального времени). По умолчанию SQL Server устанавливается с обычным уровнем приоритета — `7`.

В однопроцессорных компьютерных системах, а также когда наряду с SQL Server в системе выполняются другие фоновые приложения, желательно установить обычный уровень приоритета. Искусственное завышение приоритета в этой ситуации может нарушить гладкость операций. В то же время в условиях многопроцессорного выделенного сервера баз данных рекомендуется установить для приоритета значение `13`. Для этого параметру повышения приоритета `priority boost` следует присвоить значение `1`:

```
EXEC sp_configure 'priority boost', 1
RECONFIGURE
```

Параметр `Lightweight Pooling`

Еще один параметр, предназначенный для симметричных многопроцессорных систем, позволяет уменьшить нагрузку, вызываемую частым переключением процессов с одного процессора на другой. Включение поддержки волокон Windows NT приводит к уменьшению числа рабочих потоков, так как некоторые потоки ассоциируются с дополнительными волокнами. Уменьшение количества рабочих потоков уменьшает дополнительную нагрузку, вызываемую частым переключением потоков, и, следовательно, повышает общую производительность. В Management Studio этот параметр носит название `Use Windows NT Fibres`; в программном коде он соответствует параметру `lightweight pooling`.

```
EXEC sp_configure 'lightweight pooling', 1
RECONFIGURE
```

Параллелизм

Редакция SQL Server Enterprise Edition (а также редакции Developer и Evaluation, поскольку они отличаются от Enterprise только условиями лицензирования) позволяет выполнять сложные запросы, параллельно используя несколько процессоров. Это может оказаться полезным, особенно в ситуациях, когда запрос консолидирует все строки базы данных объемом в 20 Гбайт. Но что можно сказать об остальных пользователях, которые вынуждены ожидать, пока один пользователь не освободит занятые им процессоры? Решение проблемы лежит в ограничении числа процессоров, которые могут быть задействованы для распараллеливания выполнения одного запроса, и в установке порога стоимости (в секундах оценочного времени выполнения), после выхода за который запрос становится кандидатом на распараллеливание вычислений.

При генерировании плана выполнения параллельного запроса, синхронизации вычислений и завершении запроса создается дополнительная нагрузка на систему. Однако, несмотря на это, распараллеленные запросы выполняются на удивление быстро. Чтобы определить, использует ли конкретный запрос параллельные вычисления, достаточно посмотреть на его план выполнения в анализаторе запросов. Разделение различных потоков вычислений в параллельных запросах отображается графически.

Я рекомендую для распараллеливания вычисления запросов сделать доступными половину доступных процессоров за исключением одного (см. раздел "Назначение потоков процессорам"). Например, если сервер имеет восемь процессоров и семь из них доступны SQL Server, для параллельных вычислений лучше использовать три процессора. Так как распараллеливание вычислений может существенно повысить производительность, вы можете немного занизить порог времени выполнения, чтобы как можно большее количество запросов могло выиграть от параллелизма.

В программном коде параллелизм запросов настраивается с помощью параметра `max degree of parallelizm` и `cost threshold for parallelizm`. Установка первого из них в значение `0` позволяет сделать доступными для распараллеливания выполнения запросов все процессоры:

```
EXEC sp_configure 'max degree of parallelism', 1
EXEC sp_configure 'cost threshold for parallelism', 1
RECONFIGURE
```


Проверено

Несмотря на то что описанные выше параметры сервера могут сказаться на производительности, львиная доля быстродействия зависит от схемы базы данных, структуры запросов и индексов. Никакая настройка параметров сервера не сможет преодолеть недостатки плохого проектирования и программирования.

Параметры конфигурирования системы безопасности

Параметры конфигурирования системы безопасности, показанные в табл. 34.4, используются для управления функциями защиты SQL Server.

Параметры системы безопасности, установленные в процессе инсталляции сервера, представлены во вкладке Security диалогового окна Server Properties (рис. 34.7), поэтому их конфигурацию можно скорректировать уже после установки сервера. Параметры модели аутентификации и учетных записей Windows в SQL Server — такие же, как и при установке.

Таблица 34.4. Параметры конфигурации системы безопасности

Параметр	Уровень	Графический интерфейс установки	Программная установка
Режим аутентификации	Сервер	Management Studio	—
Уровень аудита системы безопасности	Сервер	Management Studio	—
Учетная запись автозапуска	Сервер	Management Studio	—
Режим аудита C2	Сервер	—	`EXEC sp_configure 'C2 audit mode'`
Цепочки собственности между базами данных	Сервер	Management Studio	—

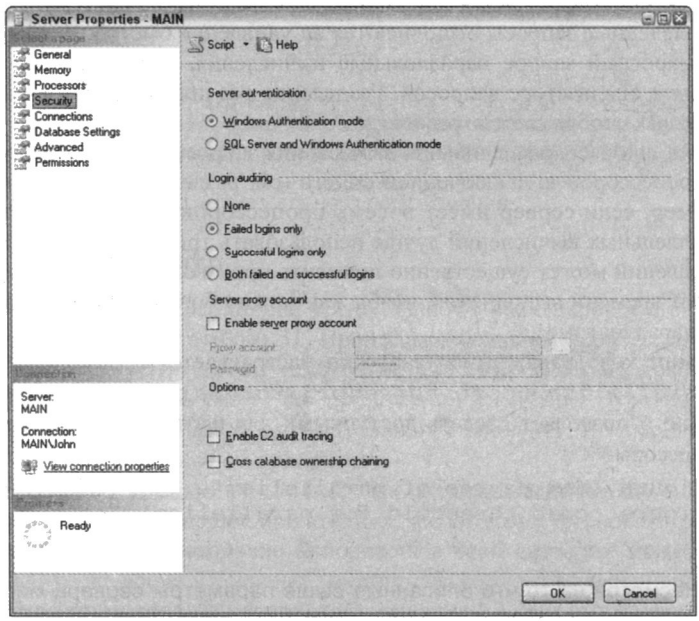


Рис. 34.7. Страница Security диалогового окна Management Studio

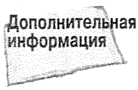
Уровень аудита системы безопасности

Этот дополнительный параметр конфигурирует уровень аудита регистрации пользователей, выполняемого сервером баз данных. На основании значения этого параметра SQL Server регистрирует все успешные и неудачные попытки входа на сервер либо в журнале событий Windows, либо в журнале SQL Server.

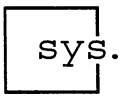
Защита C2

При конфигурировании SQL Server для уровня защиты C2 включение свойства `C2 Audit Mode` приведет к отказу в работе сервера, если он не будет способен записать информацию в журнал аудита системы безопасности. Этот параметр можно установить только программным путем — он недоступен в интерфейсе Management Studio:

```
EXEC sp_configure 'C2 audit mode', 1
RECONFIGURE
```



Более подробно о настройке системы безопасности SQL Sever вы узнаете в главе 40.



Вы имеете возможность получить текст любого запроса SQL, выбрав в динамическом представлении управления `sysdm_exec_sql_text`. Для того чтобы получить текст запроса, вы должны указать его маркер. Представление `sysdm_exec_requests` предлагает полный список всех запросов, выполняемых в текущий момент в данном экземпляре SQL Server. В выходных данных этого представления содержится также и маркер запроса, необходимый представлению `sysdm_exec_sql_text`.

Параметры конфигурации подключения

В табл. 34.5 представлены все параметры, используемые для конфигурирования подключения к SQL Server.

Таблица 34.5. Параметры конфигурации подключения

Параметр	Уровень	Графический интерфейс установки	Программная установка
Максимальное число одновременных подключений	Сервер	Management Studio	`EXEC sp_configure 'user connections'`
Параметры подключений по умолчанию	Сервер	Management Studio	—
Разрешить удаленные подключения к серверу	Сервер	Management Studio	`EXEC sp_configure 'remote access'`
Время ожидания для удаленных запросов	Сервер	Management Studio	`EXEC sp_confugure 'remote query timeout(s)'`
Включение DTC	Сервер	Management Studio	`EXEC sp_configure 'remote proc trans'`
Размер сетевого пакета	Сервер	—	`EXEC sp_configure 'network packet size'`
Время ожидания удаленного подключения	Сервер	—	`EXEC sp_configure 'remote login timeout'`

Вкладка Connections (рис. 34.8) позволяет установить параметры уровня подключения, в том числе значения по умолчанию, допустимое количество подключений и время ожидания подключения.

Максимальное число одновременных подключений

Параметр `Maximum concurrent user connections` лучше явно не устанавливать, поскольку разные приложения устанавливают подключения по-своему. Например, приложения, использующие интерфейсы ODBC и ADO, открывают по одному подключению для каждого объекта подключения (возможно, по одному для каждой формы, простого и комбинированного списков). Программа Access имеет свойство открывать как минимум два подключения.

Задачей этого параметра является ограничение количества одновременных подключений в условиях сервера с ограниченной памятью. Не стоит забывать, что каждое подключение потребляет около 40 Кбайт памяти. В большинстве конфигураций значение этого параметра устанавливают в нуль, тем самым не ограничивая число одновременных подключений.

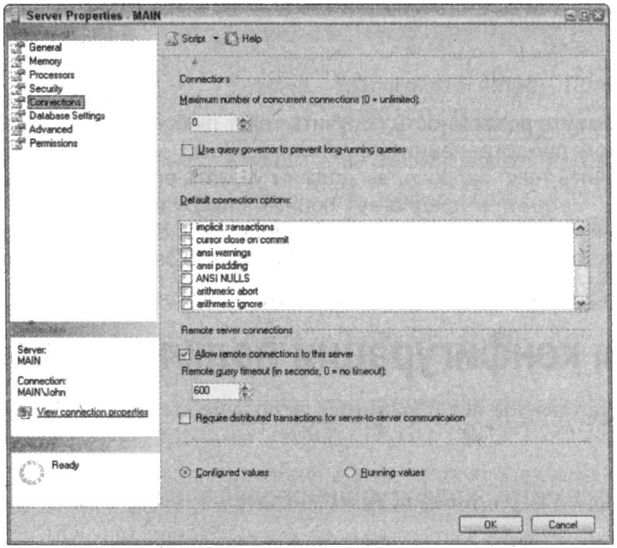


Рис. 34.8. Страница Connections *диалогового окна* Server Properties *утилиты Management Studio*

Для установки максимального числа одновременных подключений в программном коде используется параметр `user connections`:

```
EXEC sp_configure 'user connections', 0
RECONFIGURE
```

Для того чтобы программным путем узнать текущее значение этого параметра, проверьте значение глобальной переменной `@@max_connections`:

```
SELECT @@MAX_CONNECTIONS
```

Будет получен следующий результат:

```
-----------
32767
```

Удаленный доступ

Параметр `remote access` управляет доступностью сервера для удаленных и распределенных запросов — тех, в которых к данным одного сервера производится обращение из другого. По умолчанию удаленный доступ разрешен. Чтобы отключить возможность вызова сервера распределенными запросами, нужно либо снять соответствующий флажок во вкладке Connections, либо установить для параметра `remote connections` значение `0`:

```
EXEC sp_configure 'remote access', 0
RECONFIGURE
```

Время ожидания удаленного запроса

Параметр `remote query timeout` позволяет задать количество секунд, в течение которых должен быть выполнен удаленный запрос. Если этого не происходит, то выполнение запроса прерывается, а пользователю возвращается сообщение об ошибке превышения времени ожидания. Как правило, для выполнения удаленных запросов вполне достаточно значения, установленного в сервере по умолчанию, — 10 минут.

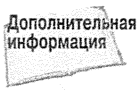
```
EXEC sp_configure 'remote query timeout', 600
RECONFIGURE
```

Включение DTC

Если в одной транзакции (единице работы) выполняется обновление нескольких серверов, то SQL Server может задействовать двухэтапное подтверждение, используя координатор распределенных запросов (DTC).

В программном коде включить DTC можно с помощью установки для параметра `remote proc trans` значения 1:

```
EXEC sp_configure 'remote proc trans', 1
RECONFIGURE
```



Транзакции будут подробно списаны в главе 51.

Размер сетевого пакета и время ожидания

Два параметра подключения доступны исключительно с помощью программного кода. Размер сетевого пакета можно изменить из принятого по умолчанию значения 4 Кбайт с помощью параметра `network packet size`. В следующем примере для сетевого пакета устанавливается размер в 2 Кбайт:

```
exec sp_configure 'network packet size', 2048
RECONFIGURE
```

Изменение размера сетевого пакета требуется в исключительно редких ситуациях. Этот параметр можно рассматривать как элемент точной настройки. Изменять его целесообразно только в том случае, когда передаваемые данные значительно превосходят по объему размер пакета, установленный по умолчанию (например, при передаче больших текстовых фрагментов или изображений).

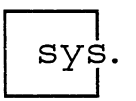
Параметр `remote login timeout` также недоступен в интерфейсе Management Studio; он устанавливает максимальное время ожидания подключения к удаленному источнику данных. Принятое по умолчанию значение 20 секунд можно изменить до 30 секунд с помощью следующего кода:

```
EXEC sp_configure 'remote login timeout', 30
RECONFIGURE
```

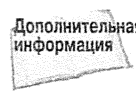
Максимальный размер реплицируемых данных

Несмотря на то что этот параметр нельзя сконфигурировать в интерфейсе Management Studio, в программном коде можно установить максимальный размер текстовых и двоичных данных, которые могут быть отправлены посредством репликации:

```
EXEC sp_configure 'max text repl size', 16767
RECONFIGURE
```



При работе с репликацией можно воспользоваться четырьмя исключительно полезными динамическими представлениями управления: `sysdm_repl_articles`, `sysdm_repl_tranhash`, `sysdm_repl_schemas` и `sysdm_repl_traninfo`.



Подробнее о репликации речь пойдет в главе 39.

Параметры конфигурации сервера

Параметры конфигурации сервера (табл. 34.6) позволяют установить настройки производительности и отображения информации SQL Server 2005.

Таблица 34.6. Параметры конфигурации сервера

Параметр	Уровень	Графический интерфейс установки	Программная установка
Язык сообщений сервера, принятый по умолчанию	Сервер	Management Studio	`EXEC sp_configure 'default language'`
Разрешение изменений в системных таблицах	Сервер	Management Studio	`EXEC sp_configure 'allow updates'`
Регулятор стоимости запроса	Сервер	Management Studio	`EXEC sp_configure 'query governor cost limit'`
Интерпретатор двузначного определения года	Сервер	Management Studio	`EXEC sp_configure 'two digit year cutoff'`
Язык, используемый по умолчанию в полнотекстовом поиске	Сервер	—	`EXEC sp_configure 'default full-text language'`

Во вкладке Advanced диалогового окна Server Properties (рис. 34.9) лучше оставить значения, принятые по умолчанию.

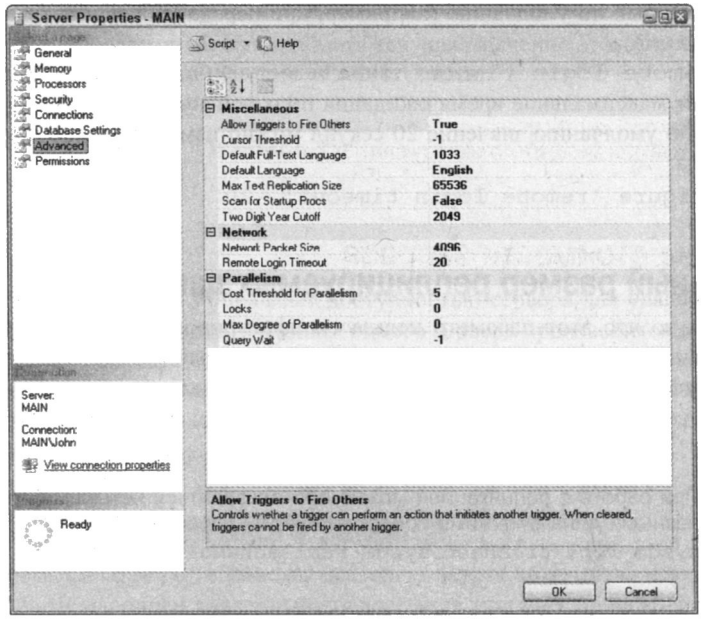


Рис. 34.9. Вкладка Advanced диалогового окна Server Properties утилиты Management Studio

Язык сообщений, принятый по умолчанию

Язык сообщений, используемый SQL Server по умолчанию, можно установить как в интерфейсе Management Studio, так и в программном коде:

```
EXEC sp_configure 'default language', 0
RECONFIGURE
```

Язык, используемый по умолчанию в полнотекстовом поиске

Язык, используемый по умолчанию в полнотекстовом поиске, может быть установлен только с помощью программного кода:

```
EXEC sp_configure 'default full-text language', 1033
RECONFIGURE
```

Значение, использованное в данном примере, соответствует английскому языку. Обычно сервер устанавливает корректное значение в процессе установки сервера баз данных. Это значение имеет смысл изменять только при необходимости поддержки языка, отличного от установленного в системе по умолчанию.



Большинство языковых настроек SQL Server основывается на идентификаторе региона (LCID). Список доступных значений LCID вы можете найти на сайте `http://krafft.com/scripts/deluxe-calendar/lcid_chart.htm`.

Разрешение внесения изменений в системные таблицы

Установка параметра сервера Allow modifications to be made to the system catalogs позволяет напрямую вносить изменения в системные таблицы, однако такой режим работы можно предотвратить. В программном коде этому параметру соответствует `allow updates`.



Я считаю, что не следует напрямую изменять значения в системных таблицах. В то же время самым эффективным механизмом внесения таких изменений являются системные хранимые процедуры, предлагаемые компанией Microsoft, а также стандартные инструкции SQL `ALTER`.

Ограничение стоимости запроса

Так же как педаль акселератора в автомобиле не позволит превысить количество оборотов двигателя, так и в SQL Server можно установить предел оценочной стоимости выполнения запросов. Если пользователь попытается выполнить запрос, выходящий за установленный предел, он не будет обработан сервером баз данных.

В следующем примере максимальное время выполнения запроса устанавливается в 10 секунд:

```
EXEC sp_configure 'query governor cost limit', 10
RECONFIGURE
```

В то же время предельную стоимость запроса можно установить программным путем и только для текущего подключения. В следующем примере снимаются какие бы то ни было ограничения на стоимость выполнения запроса в пределах текущего подключения к серверу:

```
SET QUERY_GOVERNOR_COST_LIMIT 0
```

Лично я не устанавливаю никаких ограничений на время выполнения запроса. Однако если в приложении среднее время выполнения запроса не превышает одну секунду, а пользователи часто создают собственные запросы, не задумываясь о потреблении ресурсов сервера, установка предельного значения времени выполнения запроса поможет немного разгрузить сервер.

Сокращенное определение года

Поддержка сокращенного определения года с помощью двух цифр позволяет решить так называемую "проблему 2000 года". SQL Server автоматически преобразует две цифры в четыре, руководствуясь установленными порогами, разделяющими XX и XXI век. Если две введенные цифры определяют число, превышающее заданное (по умолчанию первый порог установлен на 1959 год), то сервер интерпретирует год как принадлежащий к XX веку. Соответственно, если введенные две цифры формируют число, которое меньше второго порога (по умолчанию 2049), год будет отнесен к XXI веку. Например, дата `01/01/69` будет интерпретирована как `01/01/1969`, а `01/01/14` — как `01/01/2014`. В следующем примере второй порог, разделяющий столетия, устанавливается в сорок первый год:

```
EXEC sp_configure 'two digit year cutoff', 2041
RECONFIGURE
```

Параметры конфигурации индекса

Параметры статистики и заполнения индекса (табл. 34.7) устанавливают значения по умолчанию для новых индексов, создаваемых в сервере баз данных.

Таблица 34.7. Параметры конфигурации индексов

Параметр	Уровень	Графический интерфейс установки	Программная установка
Автоматическое создание статистики	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET auto_create_statistics {ON | OFF}`
Автоматическое обновление статистики	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET auto_update_statistics {ON | OFF}`
Множитель заполнения индекса	Сервер	Management Studio	`EXEC sp_configure 'fill factor'`

Все эти параметры не оказывают никакого влияния на уже существующие индексы. При желании вы можете просмотреть установки конфигурации индексов с помощью системной хранимой процедуры `sp_autostats`. В качестве аргумента эта хранимая процедура принимает имя базы данных, например:

```
sp_autostats 'Categories'
```

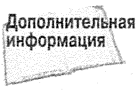
Будет получен следующий результат:

```
Global statistics settings for [Northwind]:
Automatic update statistics: ON
Automatic create statistics: ON

settings for table [Categories]

Index Name      AUTOSTATS Last Updated
--------------- --------- -----------------------
[PK_Categories] ON        2006-02-27 19:29:11.873
[CategoryName]  ON        NULL
```

Изменить параметры можно с помощью добавления флага `ON` или `OFF` после имени таблицы. К тому же существует возможность изменения параметров и конкретного индекса — для этого следует дополнительно передать хранимой процедуре его имя.



Подробнее о процедуре создания индексов см. в главе 17. Особенности управления индексами и их настройки мы обсудим в главе 50.

Конфигурирование автоматических настроек баз данных

Режим работы базы данных, принятый по умолчанию, конфигурируется с помощью четырех параметров (табл. 34.8). Все эти параметры можно установить в графическом интерфейсе Management Studio во вкладке Options диалогового окна Database Properties.

Таблица 34.8. Параметры конфигурации индексов

Параметр	Уровень	Графический интерфейс установки	Программная установка
Автоматическое закрытие	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET auto_close {ON | OFF}`
Автоматическое сжатие	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET auto_shrink {ON | OFF}`
Автоматическое создание статистики	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET auto_create_statistics {ON | OFF}`
Автоматическое обновление статистики	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET auto_update_statistics {ON | OFF}`

Автоматическое закрытие

Параметр автоматического закрытия дает указание серверу баз данных освобождать все ресурсы, используемые базой данных (страницы кэша данных, компилированные хранимые процедуры, планы выполнения запросов и т.п.), после отключения последнего пользователя или завершения всех процессов. При этом память освобождается для использования другими базами данных. Несмотря на то что использование данного параметра несколько повышает производительность других баз данных, при следующем подключении к закрытой базе данных время загрузки существенно увеличится. Это связано с тем, что придется заново компилировать хранимые процедуры и создавать планы выполнения запросов.

Если база данных используется регулярно, то не стоит активизировать параметр автоматического закрытия. Если же база данных используется от случая к случаю, то автоматическое закрытие поможет более экономно использовать физическую память.



Многие клиентские приложения постоянно открывают и закрывают подключения к серверу баз данных. В такой ситуации включение параметра автоматического закрытия способно свести к нулю производительность SQL Server.

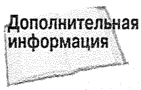
Чтобы управлять параметром автоматического закрытия программным путем, используйте следующую команду:

```
ALTER DATABASE база_данных SET AUTO_CLOSE ON | OFF
```

Автоматическое сжатие

Если база данных имеет больше 25% свободного пространства, то установка этого параметра приведет к ее автоматическому сжатию. Этот параметр также вызывает сжатие журнала транзакций при его резервировании.

Операция сжатия довольно затратная, так как в файле приходится перемещать большие массивы данных. Установка этого параметра также приводит к периодическим проверкам страниц данных с целью определения необходимости сжатия.



Процедура сжатия файлов данных и журнала транзакций подробно описана в главе 36.

Чтобы установить параметр автоматического сжатия, следует выполнить в программе следующую инструкцию:

```
ALTER DATABASE база_данных SET AUTO_SHRINK ON | OFF
```

Автоматическое создание статистики

Статистика распределения данных является ключевым моментом, на основании которого создается план выполнения запроса. Установка этого параметра дает указание серверу баз данных автоматически создавать статистику для всех столбцов, участвующих в запросах.

Чтобы установить параметр автоматического формирования статистики, следует выполнить в программе следующую инструкцию:

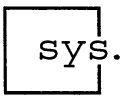
```
ALTER DATABASE база_данных SET AUTO_CREATE_STATISTICS ON | OFF
```

Автоматическое обновление статистики

Устаревшая статистика распределения данных вряд ли принесет какую-либо пользу. Установка данного параметра гарантирует, что статистика будет обновляться автоматически. По умолчанию автоматическое обновление статистики включено во всех редакциях SQL Server.

Для управления этим параметром в программном коде следует использовать следующую инструкцию:

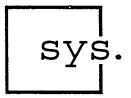
```
ALTER DATABASE база_данных SET AUTO_UPDATE_STATISTICS ON | OFF
```



Настройки запросов и индексов в большой мере зависят от статистики распределения данных. Стратегии, учитывающие статистику, подробно рассмотрены в главе 49.

Параметры конфигурации курсора

Для управления режимом работы курсора в SQL Server используются параметры, перечисленные в табл. 34.9.



Для просмотра статистики открытых курсоров в различных базах данных можно выбрать `sysdm_exec_cursors` в динамическом представлении управления.

Таблица 34.9. Параметры конфигурации индексов

Параметр	Уровень	Графический интерфейс установки	Программная установка
Порог курсора	Сервер	—	`EXEC sp_configure 'cursor threshold'`
Закрытие курсора при подтверждении	Сервер, база данных, подключение	—	`ALTER DATABASE <база_данных> SET cursor_close_on_commit {ON | OFF`
Значения курсора по умолчанию	База данных	—	`ALTER DATABASE <база_данных> SET cursor_default {LOCAL | GLOBAL}`

Порог курсора

Свойство порога регулирует количество строк, составляющих набор ключевых данных курсора, генерируемый асинхронно. Синхронные ключевые наборы данных работают быстрее, чем остальные типы, однако они потребляют больше ресурсов. Если параметр `cursor threshold` установить в значение `0`, то все ключевые наборы данных будут генерироваться асинхронно. Установка этого параметра в значение `-1` приведет к синхронной генерации ключевых наборов. Этот вариант может подойти для небольших ключевых наборов; в больших же наборах это может стать источником проблем. В следующем примере устанавливается максимальный размер синхронного ключевого набора данных курсора в 10000 строк:

```
EXEC sp_configure 'cursor threshold', 10000
RECONFIGURE WITH OVERRIDE
```

Закрытие курсора при подтверждении

Если этот параметр включен (т.е. имеет значение `on`), то после подтверждения транзакции курсор будет автоматически закрываться. Если же этот параметр отключен (т.е. имеет значение `off`), то курсор остается открытым до тех пор, пока не будет явно закрыт с помощью команды `close cursor`.

Для включения режима автоматического закрытия курсора используйте следующую команду:

```
SET CURSOR_CLOSE_ON_COMMIT ON
```

Также вы можете воспользоваться и альтернативной командой:

```
ALTER DATABASE база_данных SET CURSOR_CLOSE_ON_COMMIT ON | OFF
```

Значения курсора, принятые по умолчанию

Установка этого параметра в значение `local` определяет курсор локальным по отношению к объекту, объявившему его. Его установка в значение `global` задает область определения курсора, выходящую за пределы объекта, создавшего его.

Для присвоения значения этому параметру в программе следует использовать следующую инструкцию:

```
ALTER DATABASE база_данных SET CURSOR_DEFAULT LOCAL | GLOBAL
```

Параметры конфигурации SQL ANSI

Параметры конфигурации SQL ANSI (табл. 34.10) определяют меру поддержки сервером стандарта ANSI.

Таблица 34.10. Параметры поддержки стандарта ANSI

Параметр	Уровень	Графический интерфейс установки	Программная установка
Значения ANSI по умолчанию	Подключение	—	`SET ANSI_DEFAULTS {ON | OFF}`
Режим работы с пустыми значениями по умолчанию	Сервер, база данных, подключение	Management Studio	`ALTER DATABASE <база_данных> SET ansi_null_Default {ON | OFF}`
Порядок участия пустых значений в операциях сравнения	Сервер, база данных, подключение	Management Studio	`ALTER DATABASE <база_данных> SET ansi_nulls {ON | OFF}`
Дополнение нулями и пробелами	Сервер, подключение	—	`ALTER DATABASE <база_данных> SET ansi_padding {ON | OFF}`
Предупреждения ANSI	Сервер, база данных, подключение	—	`ALTER DATABASE <база_данных> SET ansi_warnings {ON | OFF}`
Реакция на арифметические ошибки	Сервер, подключение	—	`ALTER DATABASE <база_данных> SET arithabort {ON | OFF}`
Игнорирование арифметических операций	Сервер, подключение	—	`SET arithignore {ON | OFF}`
Реакция на ошибки округления	База данных	—	`ALTER DATABASE <база_данных> SET numeric_roundabort {ON | OFF}`
Конкатенация с пустым значением	База данных, подключение	—	`ALTER DATABASE <база_данных> SET concat_null_yields_null {ON | OFF}`
Использование идентификаторов в кавычках	База данных	—	`ALTER DATABASE <база_данных> SET quoted_identifier {ON | OFF}`
Совместимость со стандартом ANSI SQL-92	Подключение	—	`SET fips_flagger {ENTRY | FULL | INTERMEDIATE | OFF}`

Параметры, установленные по умолчанию для подключения, оказывают влияние на пакеты, выполняемые в текущей сессии соединения с сервером. Большая часть перечисленных выше параметров обеспечивает совместимость SQL Server со стандартом ANSI. Так как данные параметры редко устанавливаются в серверах баз данных, рекомендуется присваивать им значения явно, в начале программного кода. Это позволит обеспечить работоспособность программ в среде, отличной от разработок компании Microsoft.

Например, язык T-SQL требует наличия инструкции явного открытия транзакции `begin transaction`. В то же время в сервере баз данных Oracle эта инструкция подразумевается в начале каждого пакета. Если вы предпочитаете работать с неявными транзакциями, то проще установить параметр обязательного объявления начала транзакции в начале каждого пакета, чем полагаться на значения сервера, принятые по умолчанию. Параметры уровня сервера будут оказывать влияние на все выполняемые пакеты и могут привести к ошибкам выполнения некоторых расширений стандарта ANSI SQL, введенных в продуктах компании Microsoft. По этой причине рекомендуется оставлять параметры подключения без изменений, выполняя настройку уже в программном коде.

Параметры конфигурации SQL ANSI устанавливаются с помощью инструкции ALTER DATABASE. Из соображений совместимости с предыдущими версиями SQL Server также доступна хранимая процедура sp_dboption.

Режим работы с пустыми значениями, заданный по умолчанию

Параметр ansi_null_default управляет порядком работы с пустыми значениями, используемым по умолчанию. Значения этого параметра применяются, если при создании таблицы не был указан явно параметр NULL или NOT_NULL.

В программном коде режим работы с пустыми значениями, используемый по умолчанию, устанавливается следующим образом:

```
ALTER DATABASE база_данных SET ANSI_NULL_DEFAULT ON | OFF
```

Порядок участия пустых значений в операциях сравнения

Параметр базы данных ansi_nulls используется для определения порядка работы с пустыми значениями в операциях сравнения. Если этот параметр установлен в значение on, то результатом операции сравнения с пустым значением всегда будет пустое значение. Если же этот параметр установить в значение off, то операция сравнения двух пустых значений даст в результате значение true.

Установить этот параметр в программном коде можно следующим образом:

```
ALTER DATABASE база_данных SET ANSI_NULLS ON | OFF
```

Дополнение нулями и пробелами

Параметр базы данных ansi_padding оказывает влияние только на новые создаваемые столбцы. Если он установлен в значение on, то в числовых значениях остаются лидирующие нули, а в символьных значениях — лидирующие и замыкающие пробелы. Если значением этого параметра является off, то лидирующие и замыкающие нули и пробелы обрезаются.

Установить этот параметр в программном коде можно таким образом:

```
ALTER DATABASE база_данных SET ANSI_PADDING ON | OFF
```

Предупреждения ANSI

Параметр ansi_warnings управляет отображением предупреждений и сообщений об ошибках, связанных с соблюдением стандарта ANSI. Если этот параметр установлен в значение off, то все ошибки, такие как деление на нуль или участие пустых значений в итоговых функциях, пропускаются; в противном случае отображаются все ошибки.

Установить этот параметр в программном коде можно следующим образом:

```
ALTER DATABASE база_данных SET ANSI_WARNINGS ON | OFF
```

Реакция на арифметические ошибки

Если параметр arithabort установлен в значение on, то выполнение пакета прекращается в случае возникновения таких арифметических ошибок, как переполнение или деление на нуль. Если этот параметр установлен в значение off, то при возникновении арифметической ошибки выводится предупреждение, после чего выполнение программы продолжается.

Установить данный параметр можно следующим образом:

```
ALTER DATABASE база_данных SET ARITHABORT ON | OFF
```

Реакция на ошибки округления

Параметр numeric_roundabort используется для управления реакцией на ошибки округления десятичных чисел. Если он имеет значение on, то выполнение программы прекращается при потере точности в выражении. Если же он имеет значение off, то выполнение программы продолжается, а результат выражения округляется до точности объекта, в котором сохраняется число.

В программном коде данный параметр можно установить следующим образом:

```
ALTER DATABASE база_данных SET NUMERIC_ROUNDABORT ON | OFF
```

Конкатенация с пустым значением

Параметр базы данных concat_null_yields_null используется для управления режимом выполнения операции конкатенации с пустым значением. Если этот параметр имеет значение on, то результатом операции конкатенации с пустым значением будет также пустое значение. Если этот параметр имеет значение off, то результатом будет непустая строка, участвующая в операции (т.е. пустое значение попросту игнорируется).

Установить этот параметр в программном коде можно следующим образом:

```
ALTER DATABASE база_данных SET CONCAT_NULL_YIELDS_NULL ON | OFF
```

Использование идентификаторов в кавычках

Параметр quoted_identifier, будучи установленным в значение on, позволяет ссылаться на идентификаторы, заключенные в двойные кавычки. Если же этот параметр имеет значение off, то заключение идентификаторов в кавычки не допускается; к тому же идентификатором не может быть ключевое слово:

```
ALTER DATABASE база_данных SET QUOTED_IDENTIFIER ON | OFF
```

По умолчанию этот параметр имеет значение off. Для создания или изменения индексированных представлений или индексированных вычисляемых столбцов этот параметр должен быть установлен в значение on.

Параметры конфигурации триггеров

Параметры конфигурации триггеров перечислены в табл. 34.11. Они управляют порядком работы сервера баз данных с триггерами.

Таблица 34.11. Параметры поддержки стандарта ANSI

Параметр	Уровень	Графический интерфейс установки	Программная установка
Разрешение вложения триггеров	Сервер	Management Studio	EXEC sp_configure 'nested triggers'
Разрешение рекурсии триггеров	База данных	Management Studio	ALTER DATABASE <*база_данных*> SET recursive_triggers {ON | OFF}

Режим работы триггеров может быть установлен как на уровне сервера, так и на уровне базы данных.

Вложение триггеров

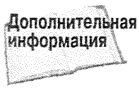
Триггеры могут быть вложены друг в друга, при этом число уровней может достигать тридцати двух. Этот параметр конфигурируется на уровне сервера.

В программном коде разрешить вложение триггеров можно следующим образом:

```
EXEC sp_configure 'nested triggers', 1
RECONFIGURE
```

Рекурсия триггеров

Триггером называется небольшая хранимая процедура, автоматически запускаемая при выполнении операции вставки, изменения или удаления записи в таблице. Если в ходе выполнения триггера снова выполняется операция, породившая триггер, то он может быть запущен повторно. Поддержку рекурсии триггеров можно включать и отключать на уровне баз данных.



На практике довольно часто путают понятия вложенных и рекурсивных триггеров. Подробнее о триггерах, а также о порядке вызова одних триггеров из других см. в главе 23.

По умолчанию этот параметр имеет значение `off`. Связанный параметр допустимости вложения триггеров устанавливается на уровне сервера. В программном коде T-SQL параметр допустимости рекурсии триггеров можно установить следующим образом:

```
ALTER DATABASE база_данных SET RECURSIVE_TRIGGERS ON | OFF
```

Параметры конфигурации состояния базы данных

Параметры конфигурации состояния базы данных перечислены в табл. 34.12. Как правило, изменения в эти параметры вносятся администратором при выполнении операций обслуживания базы данных.

Таблица 34.12. Параметры конфигурации состояния базы данных

Параметр	Уровень	Графический интерфейс установки	Программная установка
Автономный режим работы базы данных	База данных	—	`ALTER DATABASE <имя_базы> SET offline`
Доступ только для чтения	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET read_only`
Ограничение доступа только членами ролей `db_owner`, `dbcreator` и `sysadmin`	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET restricted_user`
Монопольный режим доступа	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET single_user`
Отключение монопольного режима	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET multi_user`
Уровень совместимости	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET read_only`

Состояние базы данных также может быть установлено и с помощью команды `ALTER DATABASE`. Из соображений совместимости с предыдущими версиями SQL Server также была оставлена хранимая процедура `sp_dboption`.

Уровень доступа к базе данных

Параметры конфигурации состояния базы данных позволяют установить различные уровни доступа. Когда база данных находится в автономном режиме, доступ к ней запрещен.

Чтобы перевести базу данных в автономный режим, нужно выполнить следующую инструкцию:

```
ALTER DATABASE база_данных SET OFFLINE
```

Параметр `read_only` используется для ограничения состава операций, выполняемых в базе данных, только чтением. Если в момент установки этого параметра кто-то из пользователей был подключен к базе, на него действие данного параметра не распространяется. Для восстановления стандартного режима доступа к базе данных, когда разрешены операции чтения и записи, используется параметр `read_write`.

Чтобы ограничить доступ к базе данных только операциями чтения, нужно выполнить следующую инструкцию:

```
ALTER DATABASE база_данных SET READ_ONLY
```

Существуют еще три режима доступа к базе данных: `single_user`, `restricted_user` и `multi_user`. Эти параметры определяют, кто из пользователей имеет право доступа к базе данных. Однопользовательский режим доступа уместен при выполнении операций обслуживания базы данных. Параметр `restricted_user` ограничивает круг лиц, имеющих доступ к базе данных, только членами ролей `db_owner`, `dbcreator` и `sysadmin`. Параметр `multi_user` позволяет восстановить обычный режим функционирования базы данных.

Для установки ограниченного режима доступа к базе данных используется следующая команда:

```
ALTER DATABASE база_данных SET SINGLE_USER
```

Уровень совместимости

В SQL Server уровень совместимости может быть установлен в значения от 60 (соответствующее версии SQL Server 6.0) до 80 (соответствующее SQL Server 2005). Установка уровня совместимости ниже восьмидесятого может понадобиться, если было выполнено обновление ядра базы данных, но необходимо продолжать поддерживать режим работы старых версий сервера.

Программным путем уровень совместимости можно установить следующим образом:

```
EXEC sp_dbcmptlevel база_данных, 80
```

Параметры конфигурации восстановления

Параметры конфигурации восстановления перечислены в табл. 34.13.

Таблица 34.13. Параметры конфигурации состояния базы данных

Параметр	Уровень	Графический интерфейс установки	Программная установка
Модель восстановления	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET RECOVERY {FULL | BULK_LOGGED | SIMPLE}`
Определение разрывов страниц	База данных	Management Studio	`ALTER DATABASE <имя_базы> SET TORN_PAGE_DETECTION {ON | OFF}`

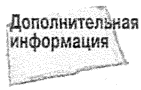
Параметр	Уровень	Графический интерфейс установки	Программная установка
Время ожидания резервного копирования	Сервер	—	—
Задержка носителя	Сервер	—	`EXEC sp_configure 'media retention'`
Интервал восстановления	Сервер	—	`EXEC sp_configure 'recovery interval'`

Параметры восстановления определяют порядок регистрации транзакций в журнале, а также его резервирования.

Модель восстановления

SQL Server 2005 использует модели восстановления, в которые объединены сходные параметры, регулирующие порядок ведения и резервирования журнала транзакций. Разные модели восстановления формируют архивы разного размера, что сказывается и на возможностях восстановления информации. В SQL Server представлено несколько моделей восстановления.

- **Простая модель**. В журнале регистрируются только те транзакции, которые еще не были записаны в файл данных. Эта модель не позволяет восстановить информацию в состояние на конкретный момент времени.
- **Модель с неполным протоколированием**. В журнале протоколируются все инструкции DML. Массовые операции вставки не протоколируются, а только маркируются.
- **Полная модель**. В журнале регистрируются все изменения, выполняемые в файле данных. Этот вариант обладает самым большим потенциалом восстановления.



Более подробно различные модели восстановления будут описаны в главе 36.

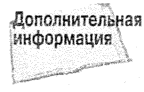
Для установки модели восстановления программным путем используется инструкция `SET RECOVERY`.

Определение разрывов страниц

Несмотря на то что SQL Server работает со страницами данных объемом в 8 Кбайт, операционная система выполняет физическую запись на диск секторами объемом в 512 байт. Если в ходе операции записи страницы данных произошла ошибка, то на диск могут быть записаны не все ее секторы.

Для поддержки свойств ACID базы данных в каждый сектор записывается контрольный бит. Если все секторы были успешно обновлены, то контрольные биты должны в них совпадать. Если в процессе восстановления какой-либо из контрольных битов будет отличаться, то SQL Server может определить условие разрыва страницы и пометить базу данных как сомнительную.

Некоторые считают, что если компьютер подключен к блоку бесперебойного питания, то этот параметр устанавливать не обязательно, однако я настоятельно рекомендую его использовать.



Все дополнительные параметры восстановления (такие как время ожидания резервирования, удержание носителей и интервал восстановления) подробно описаны в главе 36.

Резюме

Параметры конфигурации важны для совместимости с разными стандартами, настройки производительности и управления подключениями. Параметры конфигурации могут устанавливаться на уровне сервера, базы данных или подключения. Бо́льшую часть параметров можно установить в графическом интерфейсе утилиты Management Studio; практически все можно сконфигурировать с помощью программного кода.

В следующей главе мы продолжим рассмотрение темы администрирования базы данных и остановимся на вопросах переноса базы данных с гарантией сохранения ее согласованности.

Перенос баз данных

ГЛАВА

35

В этой главе...

Использование мастера копирования баз данных

Генерация сценариев SQL

Отключение и подключение баз данных

Перенос данных может показаться надуманной задачей, однако на практике чаще всего базы данных проектируются на одном сервере, а развертываются — на других. Без надежного и эффективного способа переноса схем и информации баз данных ни один проект далеко не достигнет успеха.

В SQL Server для перемещения баз данных предусмотрено несколько механизмов. Как разработчик или администратор баз данных вы должны быть хорошо подкованы в следующих вопросах, три из которых мы рассмотрим в настоящей главе.

- Мастер копирования баз данных.
- Сценарии SQL.
- Подключение и отключение баз данных.
- Резервирование и восстановление баз данных (этому вопросу посвящена следующая глава).

Ключевым моментом в определении лучшего способа организации переноса данных является знание того, какие именно данные следует переместить, и имеет ли сервер непосредственное подключение к скоростной сети. В табл. 35.1 перечислены требования к копированию и различные методы перемещения баз данных.

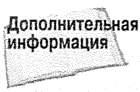
Мастер копирования баз данных

Мастер копирования баз данных (Copy Database Wisard) генерирует пакет службы преобразования данных (DTS), который может скопировать или переместить одну или несколько баз данных с одного сервера на другой. Если база данных перемещается между серверами одной локальной сети, то этот метод является наиболее предпочтительным. Этот метод не пригоден для копирования баз данных SQL Server 2005 в более старые версии SQL Server. К тому же на исходном и целевом серверах должны быть запущены агенты SQL Server Agent

(в SQL Server 2005 этот агент по умолчанию остановлен). Мастер копирования баз данных наделен большими способностями и обеспечивает достаточную гибкость в работе. Единственное ограничение заключается в том, что к базе данных должен быть установлен монопольный доступ. Запускается мастер копирования баз данных с помощью щелчка правой кнопкой мыши на копируемой базе и выбора в контекстном меню пункта Tasks⇨Copy Database. Пропустите страницу приветствия, щелкнув на кнопке Next.

Таблица 35.1. Методы перемещения баз данных

Требование	Мастер копирования баз данных	Сценарий SQL	Отключение и подключение	Резервирование и восстановление
Монопольный доступ к базе данных	Да	Нет	Да	Нет
Копирование между не связанными друг с другом серверами	Нет	Да	Да	Да
Копирование схемы базы данных	Да	Да	Да	Да
Копирование данных	Да	Нет	Да	Да
Защищенность копирования	Регистрационные записи сервера, пользователи базы данных, роли и разрешения	Зависит от конкретного сценария	Пользователи базы данных, роли и разрешения	Пользователи базы данных, роли и разрешения
Копирование заданий и пользовательских сообщений об ошибках	Да	Зависит от конкретного сценария	Да	Да



Более подробно о запуске и остановке агента SQL Server Agent см. в главе 4.

В первых двух окнах мастера копирования баз данных определяются серверы источника и назначения, а также регистрационная информация, необходимая для подключения.

В третьем окне будет предложено выбрать метод переноса базы данных. Метод отключения/подключения работает быстрее, однако требует дополнительных привилегий в базах данных источника и назначения; к тому же вы должны иметь монопольный доступ к обеим базам данных. Лучше всего метод отключения/подключения проявляет себя при работе с крупными базами данных. Метод, использующий модель SMO (SQL Management Objects), не требует какого-либо особого доступа к базам данных, и пользователи могут продолжать работу с базой источника. Однако этот метод работает значительно медленнее, и компания Microsoft не рекомендует его использовать для переноса крупных баз данных.

В четвертом окне мастер отобразит принятые по умолчанию папки размещения файлов базы данных на сервере назначения, которые вы можете изменить по мере необходимости. В заданную папку будут перемещены все объекты и данные.

В пятом окне вам будет предложено сконфигурировать базу данных назначения. Базе данных следует присвоить имя и определить, как мастер должен поступить, если на целевом сервере уже существует база с таким именем.

В четвертом окне (рис. 35.1) вы можете указать мастеру состав перемещаемых объектов.

- Все учетные записи или только те, которые имеют доступ к перемещаемой базе данных.
- Все или только избранные несистемные хранимые процедуры базы данных master, которые используются перемещаемой базой данных.
- Все или только избранные задания агента SQL Server Agent (автоматизированные и запускаемые по расписанию задания).
- Все или только избранные пользовательские сообщения об ошибках (используемые командой T-SQL `RAISEEROR`).

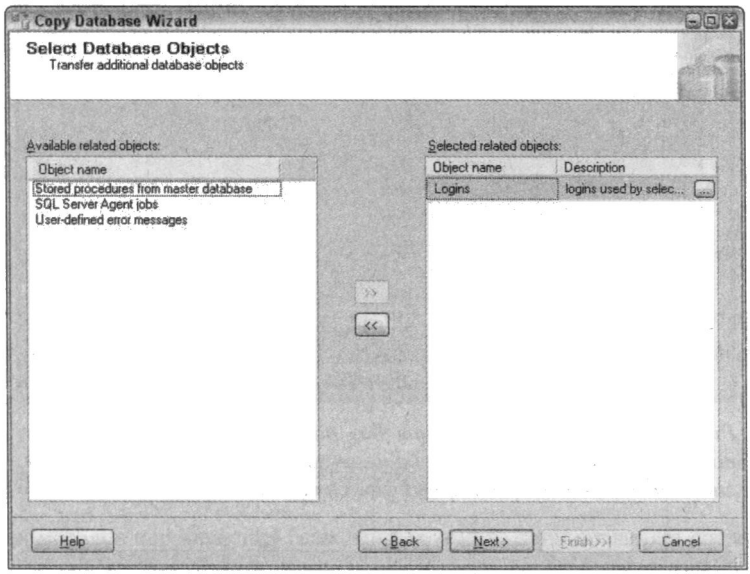


Рис. 35.1. Мастер копирования баз данных позволяет заодно переместить и информацию, связанную с сервером

В зависимости от выбранных параметров мастер может открыть дополнительные страницы для отбора несистемных хранимых процедур базы данных `master`, заданий SQL Server Agent и пользовательских сообщений об ошибках.

На странице конфигурирования пакета отображается место размещения пакета; также вы можете задать его имя и выбрать метод протоколирования ошибок. По умолчанию все ошибки заносятся в журнал событий Windows, однако можно направить вывод и в текстовый файл (в последнем случае мастер потребует ввести имя файла и место его размещения).

На странице определения времени запуска пакета (рис. 35.2) можно либо выбрать момент завершения работы с мастером, либо отложить запуск на более позднее время, либо запланировать регулярный запуск по расписанию.

После завершения работы мастер сгенерирует и запустит пакет службы интеграции (рис. 35.3), а также сохранит его на сервере назначения.



С выходом пакета обновлений SP2 стало возможно создавать файл словесного протокола, который можно просмотреть для выявления проблем, связанных с операцией переноса базы данных.

Вы можете открыть сгенерированный пакет (рис. 35.4), выбрав в дереве консоли SQL Server Agent узел `Jobs` и дважды щелкнув на нужном пакете. Если имя пакета не редактиро-

валось в мастере на странице создания расписания, то оно будет состоять из символов CDW, имен двух серверов и целого числа. В списке консоли вы можете также увидеть и дату создания пакета.

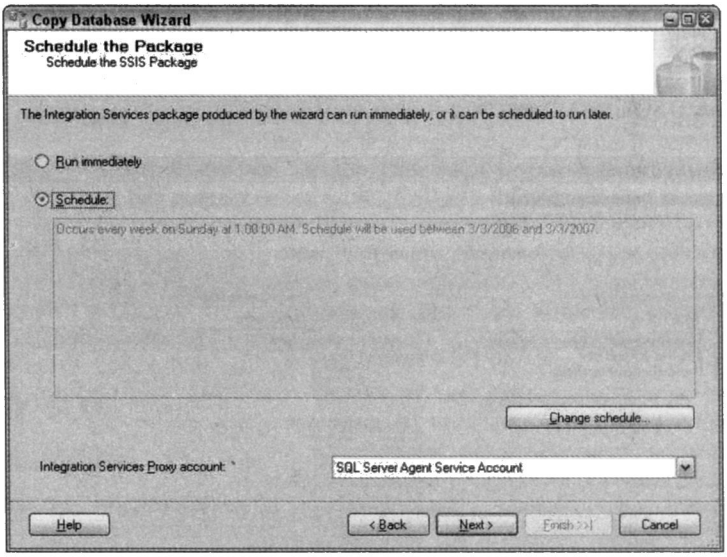


Рис. 35.2. Мастер копирования баз данных позволяет запустить пакет службы интеграции либо немедленно, либо в заданное время, либо запланировать регулярный запуск по расписанию

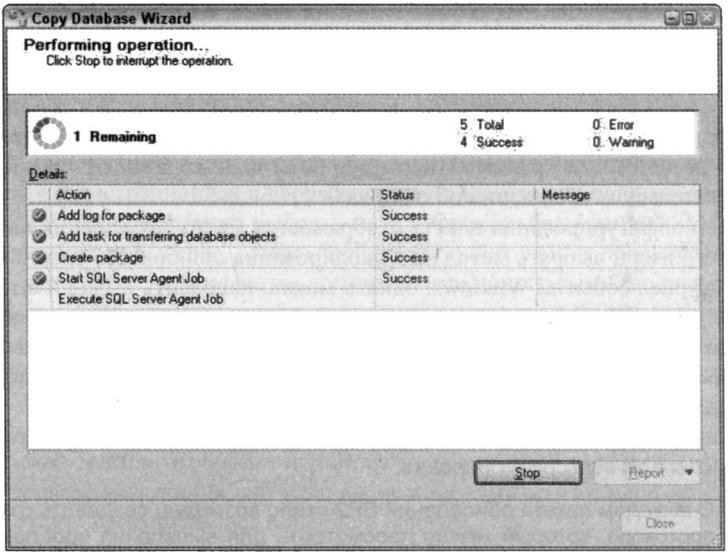


Рис. 35.3. При выполнении в мастере копирования баз данных пакета службы интеграции в окне отображается последовательность выполняемых действий

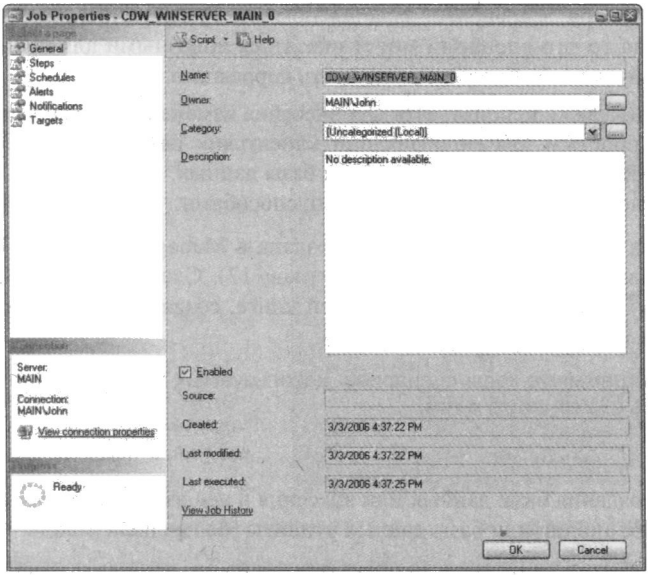


Рис. 35.4. Мастер копирования баз данных создает задание агента SQL Server Agent; в данном случае оно получило название `CDW_WINSERV_MAIN_0`

Использование сценария SQL

Из четырех методов перемещения базы данных только один позволяет создать новую базу данных — это запуск сценария (или пакета) SQL. С логической точки зрения это можно считать ошибочным, однако начинать работу с чистой установки на стороне клиента, без всяких остатков тестовых данных, несомненно, лучше.

Сценарии меньше по размерам, чем базы данных, поэтому их можно вмесить даже на обычную дискету. К тому же сценарии можно отредактировать с помощью обычного текстового редактора, такого как Блокнот (Notepad). Кстати, все учебные базы данных, используемые в настоящей книге, распространяются с помощью сценариев.

Сценарии могут оказаться полезными для распространения объектов, перечисленных ниже.

- Схемы баз данных (базы данных, таблицы, представления, хранимые процедуры, функции и т.п.).
- Роли системы безопасности.
- Задания базы данных.
- Ограниченные наборы учебных или начальных данных.

Несмотря на то что такое возможно, я бы не рекомендовал использовать сценарии для распространения следующих объектов.

- **Данные**. Сценарии можно использовать для вставки отдельных строк, однако перемещать с их помощью большие наборы данных чрезвычайно сложно.
- **Регистрационные записи сервера**. С помощью сценариев можно легко создавать регистрационные записи, однако те обычно привязаны к конкретным доменам, вследствие чего использовать данный метод имеет смысл только в пределах одного домена.

- **Задания сервера**. Серверные задания обычно требуют индивидуальной настройки. Несмотря на то что сценарии могут оказаться полезными для копирования заданий, перед их запуском необходимо выполнить коррекцию.

Сценарии можно также использовать для внесения изменений в базу данных, и этот метод считается самым простым для модификации клиентских баз. Работоспособность сценария при этом можно проверить на резервной копии базы данных.

Сценарии можно сгенерировать несколькими способами.

- База данных может изначально быть создана в Management Studio с использованием заранее созданных сценариев DDL (см. главу 17). Следует отметить, что все учебные базы данных, используемые в настоящей книге, создавались именно с помощью сценариев DDL. Это мой любимый метод.



Программные коды сценариев, используемых в настоящей главе, можно загрузить с Web-сайта книги.

- Сценарии создания базы данных или внесения в нее изменений можно сгенерировать в конструкторе таблиц или базы данных утилиты Management Studio.
- Большинство сторонних средств проектирования баз данных позволяют сгенерировать сценарии создания баз данных и внести в них изменения.

Теперь рассмотрим генерацию сценариев в утилите Management Studio. Откройте генератор сценариев, щелкнув правой кнопкой мыши на базе данных в дереве консоли и выбрав в контекстном меню пункт Tasks⇨Generate Scripts.

Пропустите страницу приветствия, щелкнув на кнопке Next. На странице Select Database мастера сценариев утилиты Management Studio (рис. 35.5) выберите объекты, которые должны быть включены в сценарий, и щелкните на кнопке Next.

На странице Choose Script Options (рис. 36.6) содержатся два набора параметров сценария. В разделе General доступны общие параметры, определяющие режим работы сценария (например, следует ли добавлять новый сценарий в уже существующий файл сценария). В разделе Table/View доступны параметры, определяющие функции сценария (например, следует ли включать в сценарий команды создания внешних ключей). Щелкните на кнопке Next.

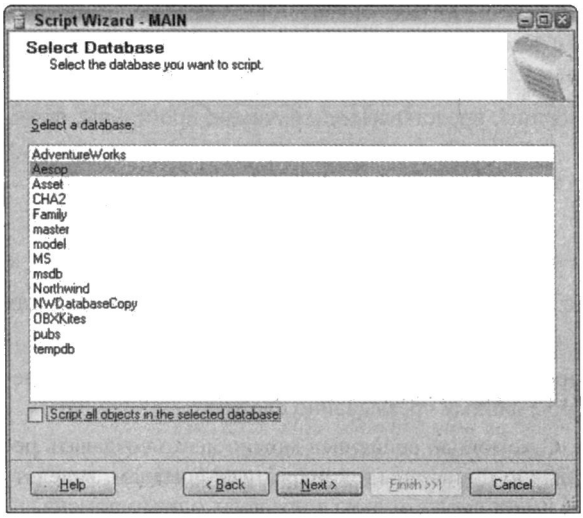


Рис. 35.5. Management Studio может сгенерировать сценарии для любой базы данных сервера

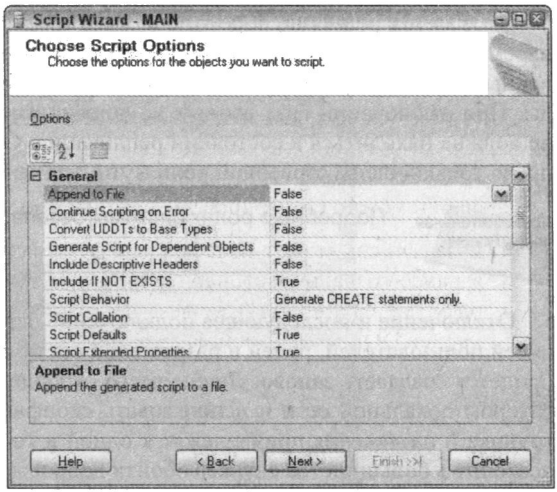


Рис. 36.6. Установите параметры сценария, соответствующие функциям базы данных, и определите действия, которые вы ожидаете от Management Studio

На странице Choose Object Types содержится список объектов, существующих в выбранных базах данных. Как минимум, вы увидите вариант занесения в сценарий инструкций создания таблиц.

Последовательность следующих страниц мастера зависит от состава выбранных объектов. Например, если вы выбрали тип объектов `Tables`, откроется страница выбора таблиц, которые нужно заложить в сценарий.

В конце концов вы окажетесь на странице Output Options (рис. 35.7). Параметры этой страницы позволяют выбрать метод вывода сценария. Если вы установите переключатель в положение Script to File, то вам нужно будет дополнительно ввести имя файла и выбрать формат вывода.

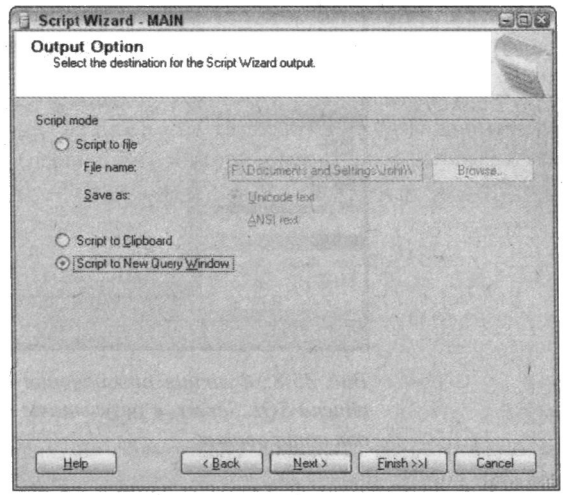


Рис. 35.7. Выбор метода вывода, соответствующий типу преобразования базы данных, которое вы хотите выполнить

Отключение и подключение

Этот метод часто обходят вниманием, несмотря на то, что он является простейшим. Он предполагает отключение базы данных, копирование файлов на сервер назначения и подключение на этом сервере скопированных файлов базы данных.

Этот метод нельзя не порекомендовать владельцам ноутбуков, которым приходится часто перемещать базы данных между сервером и портативным компьютером. Отключение базы данных отменяет ее контроль со стороны SQL Server, в то же время не затрагивая сами файлы. При отключении база данных не должна иметь активных подключений пользователей и не должна находиться в состоянии репликации. Отключать и подключать базы данных имеют право только члены серверной роли `SysAdmins`.

Дополнительная информация Подробно о ролях системы безопасности мы поговорим в главе 40.

Отключение и последующее подключение баз данных приводит к потере всех учетных записей пользователей, ролей и разрешений системы безопасности — на сервере назначения их придется создавать заново. Лучше всего скоординировать систему безопасности с администратором локальной сети и использовать сконфигурированные им группы. Если серверы источника и назначения принадлежат к одной и той же группе системы безопасности сети, это позволит в большинстве случаев обойти вопросы, связанные с регистрацией.

В Management Studio щелкните правой кнопкой мыши на копируемой базе данных и выберите в контекстном меню пункт Tasks⇨Detach. Откроется диалоговое окно, показанное на рис. 35.8.

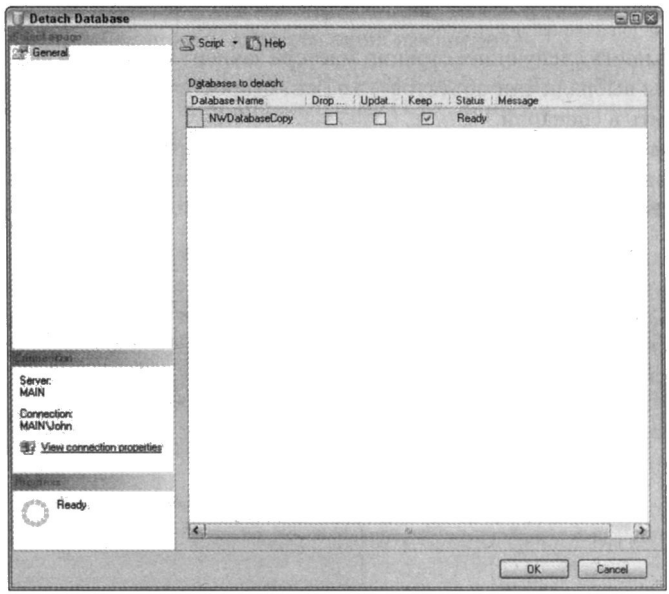


Рис. 35.8. Функция отключения базы данных удаляет ее из списка SQL Server, в результате чего файлы освобождаются для копирования

После отключения файла базы данных он исчезает из списка баз данных Management Studio. Теперь файлы базы данных можно копировать и перемещать так же, как обычные файлы.

Чтобы снова подключить базу данных, выделите в дереве консоли Management Studio узел `Databases` и выберите пункт Tasks⇨Attach из контекстного меню или из меню Action программы. Диалоговое окно подключения базы данных (рис. 35.9) предлагает выбрать файл данных и проверить его место размещения и имя.

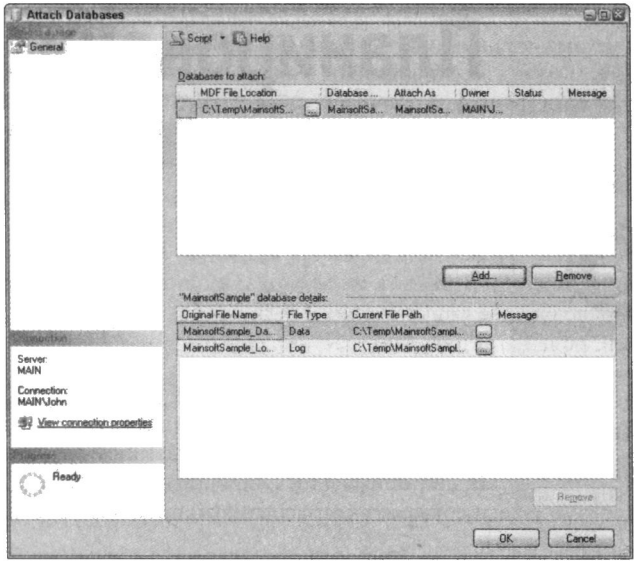


Рис. 35.9. База данных может быть снова подключена с помощью диалогового окна **Attach Database** *утилиты Management Studio*

Эти же операции можно выполнить и с помощью программного кода. Для отключения базы предназначена системная хранимая процедура `sp_detach_db`. Первым параметром является имя отключаемой базы данных; второй параметр позволяет отключить автоматическое обновление статистики индексов. С помощью следующей команды можно отключить учебную базу данных OBXKites:

```
sp_detach_db 'OBXKites'
```

Если вы хотите снова подключить эту базу данных и сделать это программным путем, используйте системную хранимую процедуру `sp_attach_db`. Подключение базы данных требует указания путей к файлам базы данных (первичному и вторичным файлам данных, а также к файлу журнала транзакций), например:

```
EXEC sp_attach_db @dbname = 'OBXKites',
@filename1 = 'e:\SQLData\OBXKites.mdf',
@filename2 = 'f:\SQLData\OBXKitesStatic.ndf',
@filename3 = 'g:\SQLLOG\OBXKites.ldf'
```

Резюме

Если вам нужно переместить базу данных, не обязательно создавать ее резервную копию — существуют и более простые способы ее переноса на другой сервер. Выбор правильного метода транспортировки основывается на пропускной способности сети, к которой подключены серверы источника и назначения, а также на составе перемещаемых объектов.

В следующей главе мы займемся вопросами формирования и реализации плана восстановления базы данных. Вы узнаете не только о методах выполнения резервного копирования, но и о внутренних механизмах журнала транзакций, а также об операциях восстановления различных объектов данных, вплоть до всего сервера.

ГЛАВА

36

Планирование восстановления

В этой главе...

Концепция модели восстановления

Резервирование пользовательских баз данных

Работа с журналом транзакций

Восстановление пользовательской или системной базы данных, а также сервера в целом

Настоящая книга начиналась с введения в основные принципы информационной архитектуры (см. главу 1). В ней объяснялись причины того, почему всегда должен существовать согласованный план восстановления.

> *Информация является организационным активом и, согласно своему содержанию и значению, должна быть соответствующим образом организована, учтена, защищена и сделана доступной в удобном формате для ежедневных операций и анализа отдельными сотрудниками, их группами и процессами как сегодня, так и в будущем.*

Выполнение принципа постоянной доступности в настоящее время и в будущем должно обеспечиваться независимо от технических сбоев, природных катаклизмов и случайных удалений.

Мир несовершенен, и неприятности случаются даже с хорошими людьми. Если вы уже начали читать эту главу, думаю, согласитесь, что регулярное резервирование данных не выглядит особо привлекательно. В некоторых сферах деятельности потеря информации равносильна катастрофе. Хороший администратор баз данных всегда должен быть готов к худшему, а это значит, что он должен иметь полноценный и неоднократно проверенный план восстановления.

Обеспечивая согласованность и гибкость, которую можно найти в других аспектах работы SQL Server, в этом сервере баз данных существует несколько возможных вариантов выполнения резервирования информации, каждый из которых лучше всего подходит к определенным ситуациям. SQL Server предлагает три модели восстановления, призванные упростить организацию резервирования и администрирование базы данных.

В этой главе мы обсудим концепции, связанные как с резервированием данных, так и с их восстановлением. Было бы глупо изучать резервирование, не зная, как из созданной резервной копии восстановить базу данных.

Дополнительная информация

Планирование восстановления данных нельзя рассматривать обособленно. В теорию, лежащую в основе полноценного плана восстановления, глубоко проникло понятие поддержки целостности транзакций (см. главы 1 и 51). Когда стратегия восстановления определена, ее, как правило, нужно внедрить в план обслуживания базы данных (глава 37). Так как восстановление является одним из факторов концепции доступности, при создании плана восстановления следует учитывать высокую доступность резервных серверов и доставки журналов (глава 52).

В то время как резервное копирование всегда кажется утомительным процессом, восстановление обычно выполняется в условиях крайнего возбуждения пользователей. По этой причине имеет смысл быть хорошо знакомым не только с процессом архивирования, но и с процессом восстановления.

Концепции восстановления

Концепция восстановления базы данных основана на поддержании свойства *живучести* принципов ACID целостности транзакций. Живучесть подразумевает, что если транзакция подтверждена, то она должна реализоваться базой данных, независимо от каких-либо технических катаклизмов.

SQL Server реализует поддержку целостности транзакций с помощью последовательно организованного журнала транзакций. Каждая транзакция перед записью в базу данных предварительно записывается в журнал транзакций. Это обеспечивает плану восстановления некоторые преимущества.

- Наличие журнала транзакций гарантирует, что любая транзакция может быть восстановлена на момент, предшествовавший сбою сервера.
- Журнал транзакций разрешает выполнение резервирования в процессе обработки транзакций.
- Журнал транзакций минимизирует влияние сбоев оборудования, так как может быть размещен в дисковой подсистеме, отдельной от той, которая используется для хранения файлов данных.

Стратегия плана восстановления должна базироваться на уровне толерантности (или *уровне болезненности*) утерянных транзакций. Тактика плана восстановления включает в себя выбор одного из возможных вариантов резервного копирования, создание графика резервирования данных, а также хранение данных вне офиса.

Резервирование и восстановление в SQL Server — достаточно гибкие операции. SQL Server предлагает на выбор три модели восстановления. Журнал транзакций может быть сконфигурирован на основе ваших потребностей восстановления, согласно одной из следующих моделей восстановления.

- **Простая модель**. Журнал транзакций не резервируется.
- **Модель с неполным протоколированием**. Массовые операции не заносятся в журнал транзакций.
- **Полная модель**. Все транзакции заносятся в журнал.

В дополнение SQL Server предлагает пять вариантов резервного копирования.

- **Полное**. Резервируются все данные.
- **Дифференцированное**. Резервирование всех страниц данных, измененных с момента последнего полного резервного копирования.
- **Журнал транзакций**. Резервирование всех транзакций в журнале.

- **Файл или файловая группа**. Резервирование всех данных, содержащихся в файле или файловой группе.
- **Файловое дифференцированное**. Резервирование всех страниц данных, модифицированных с момента последнего резервного копирования файла или файловой группы.



Резервирование базы данных является не единственным критичным резервным копированием, которое следует учесть в плане восстановления. Если схема обеспечения безопасности базы данных основана на аутентификации Windows, следует также резервировать учетные записи Windows. Таким образом, план восстановления SQL Server должен вписываться в глобальный план восстановления информационной структуры организации.

Резервное копирование в SQL Server довольно гибкое; оно поддерживает создание резервной копии в любом количестве файлов. Один архив может быть распределен в нескольких файлах, формируя *резервный набор*. И наоборот, один резервный набор может содержать множество экземпляров архивов.

Восстановление всегда начинается с использования полной резервной копии. После этого из архивов дифференцированного и транзакционного резервирования восстанавливаются все транзакции, выполненные с момента создания полной резервной копии.



Если вы знакомы с процессом резервирования в SQL Server 2000, то заметите ряд изменений и улучшений. Редко используемые функции, такие как мастер резервирования и отложенное резервирование, отошли в небытие. В то же время появился ряд новых, долгожданных функций, таких как восстановление частей поврежденного файла и восстановление к заданному моменту времени (эти функции доступны в редакции Enterprise Edition).

Модели восстановления

Модель восстановления конфигурирует настройки базы данных SQL Server так, чтобы обеспечить тот тип восстановления, который необходим базе данных (табл. 36.1). Ключевые отличия между разными моделями восстановления связаны с тем, в какой мере в них задействован журнал транзакций и какие данные в нем регистрируются.

Таблица 36.1. Модели восстановления в SQL Server

Модель восстановления	Описание	Атомарность транзакций	Живучесть транзакций	Массовые операции (`SELECT INTO` и `BULK INSERT`)
Простая	Журнал транзакций постепенно обрезается в контрольных точках	Да	Нет. Можно восстановить только последнюю полную или дифференцированную резервную копию	Не протоколируются с целью повышения производительности
С неполным протоколированием	Инструкции `SELECT INTO` и `BULK INSERT` не протоколируются как транзакции	Да	Не всегда. Всегда можно восстановить последнюю полную или дифференцированную копию. Также возможно восстановить последний транзакционный архив, если не выполнялись массовые операции	Только отмеченные (с целью повышения производительности)

Окончание табл. 36.1

Модель восстановления	Описание	Атомарность транзакций	Живучесть транзакций	Массовые операции (SELECT INTO и BULK INSERT)
Полная	Все транзакции заносятся в журнал и сохраняются в нем до выполнения резервирования журнала транзакций	Да	Да. Можно к любой точке	Выполняются медленнее, чем при использовании простой модели или модели с неполным протоколированием

Несмотря на то что живучесть транзакций поддается конфигурированию, журнал транзакций продолжает использоваться как последовательный список, обеспечивая тем самым атомарность каждой транзакции. В случае системного сбоя журнал транзакций используется сервером для отката всех незавершенных транзакций, равно как и для завершения всех подтвержденных.

Простая модель восстановления

Простая модель восстановления идеально подходит тем базам данных, которым требуется обеспечение атомарности транзакций, но не обязательна поддержка их живучести. Простая модель форсирует обрезание (или очищение) сервером баз данных журнала в контрольных точках. При этом журнал будет хранить транзакции, запись которых в базу данных еще не подтверждена; пространство же, отведенное для хранения всех остальных транзакций, освобождается для повторного использования.

Так как журнал транзакций в этой модели является только временным местом хранения, отпадает потребность в его резервировании. Эта модель восстановления имеет ряд преимуществ. Во-первых, журнал транзакций имеет маленькие размеры, однако расплачиваться за это придется потерей всех транзакций, выполненных с момента последнего полного или дифференцированного резервирования. Выбор простоя модели восстановления функционально эквивалентен установке в версиях SQL Server 7.0 и более поздних параметра базы данных `truncate log on checkpoints` в значение `true`.

План восстановления, основанный на простой модели, позволяет выполнять полное резервирование раз в неделю, а дифференцированное — в конце каждого дня недели (рис. 36.1). Полная и дифференцированные резервные копии заменяются, когда будет выполнено следующее полное резервное копирование.

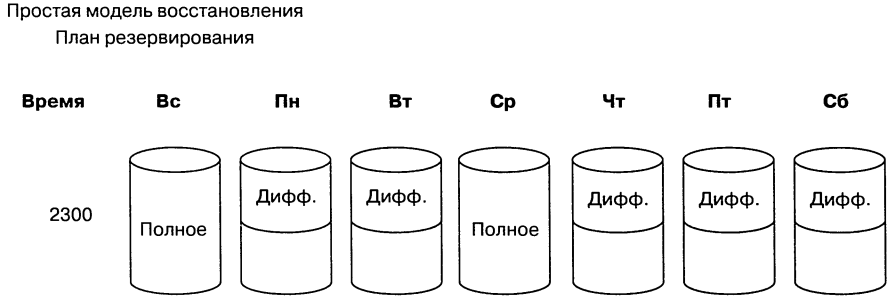


Рис. 36.1. Типичный план восстановления, использующий простую модель, содержит только полные и дифференцированные копии резервирования

Восстановление в простой модели предполагает две операции.

1. Восстановление из последней полной резервной копии.
2. Восстановление из последней (не обязательно) одной дифференцированной резервной копии.

Полная модель восстановления

Полная модель восстановления предлагает наиболее грубый и надежный план восстановления. В этой модели все транзакции, в том числе и массовые операции, протоколируются в журнале. Любая системная функция, такая как создание индекса, также протоколируется. Основным преимуществом этой модели является то, что все транзакции, выполненные в базе данных, могут быть восстановлены, вплоть до момента, предшествовавшего системному сбою.



В промышленных базах данных рекомендуется использовать полную модель восстановления. В то время как файлы базы данных находятся в одной дисковой подсистеме, журнал транзакций лучше разместить в другой, как можно более надежной. Это обеспечит высокую живучесть транзакций.

Баланс между высоким уровнем целостности транзакций и производительностью можно поддерживать с помощью рассмотрения следующих вопросов.

- Массовые операции будут выполняться значительно медленнее. Если база данных не импортирует информацию с использованием массовых методов, то данный вопрос следует обсудить отдельно.
- Журнал транзакций может достигнуть гигантских размеров. Если вы не стеснены нехваткой дискового пространства. то данный вопрос также следует обсудить отдельно.
- Резервирование и восстановление журнала транзакций будет отнимать больше времени по сравнению с другими моделями восстановления. В то же время в кризисных ситуациях восстановление всех данных может оказаться более важным, чем экономия времени за счет частичного восстановления информации.

Полная модель восстановления может использовать все пять типов резервирования базы данных. Типичный график создания резервных копий приведен на рис. 36.2.

Чаще всего полная модель восстановления предполагает выполнение полного резервирования дважды в неделю, а дифференцированного — в конце каждого дня. Резервирование журнала транзакций выполняется регулярно на протяжении всего дня; при этом промежутки времени между отдельными сессиями могут колебаться от нескольких часов до нескольких минут.

Восстановление базы данных в этой модели предполагает выполнение следующих операций.

1. Резервирование текущего журнала транзакций.



Если повреждена дисковая подсистема, содержащая журнал транзакций, то база данных помечается сервером как отключенная. Таким образом, невозможно создать резервную копию текущего журнала транзакций. В данном случае лучшим вариантом будет восстановление из последнего доступного архива журнала транзакций. Другими причинами отключения базы данных могут стать удаление или переименование файла журнала, а также перевод базы данных в автономный режим работы.

2. Выполните восстановление из последней полной резервной копии.
3. Выполните восстановление из последней дифференцированной резервной копии, созданной после выполнения последнего полного резервирования (если таковая имеется).

4. Последовательно восстановите все резервные копии журнала транзакций, созданные с момента последнего полного или дифференцированного резервного копирования. Если последней созданной резервной копией была полная, то ее восстановления будет достаточно. Если последней созданной резервной копией была дифференцированная, то перед ее восстановлением следует выполнить восстановление из последней полной резервной копии.

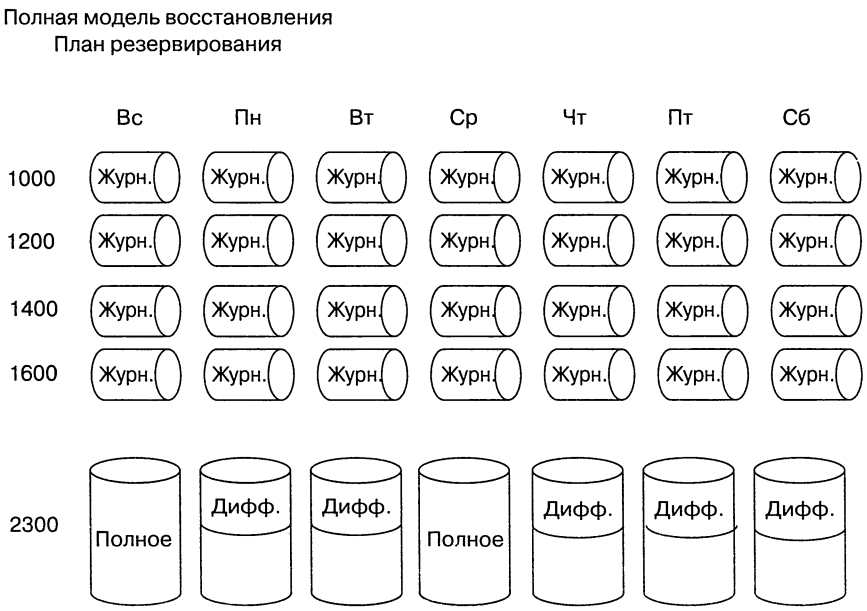


Рис. 36.2. Типичный план восстановления, использующий полную модель, содержит полные, дифференцированные и транзакционные резервные копии

Форма восстановления утилиты Management Studio (на ней мы остановимся немного позже) автоматически поможет вам выбрать корректный набор резервных копий, так что ваша задача на практике окажется не такой сложной, какой кажется на первый взгляд.

Модель с неполным протоколированием

Модель восстановления с неполным протоколированием аналогична полной модели, за исключением того, что в журнал транзакций не заносятся следующие операции:

- массовые вставки (с использованием утилиты BCP);
- инструкции DML `SELECT * INTO`;
- операции над особо крупными объектами `WRITETEXT` и `UPDATETEXT`;
- инструкции `CREATE INDEX` (включая индексированные представления).

Так как эта модель восстановления не предполагает протоколирования перечисленных выше операций, они выполняются достаточно быстро. В журнал транзакций заносятся только метки о выполнении данных операций и отслеживаются только экстенты (группы из восьми страниц данных), на которые эти операции повлияли. Когда журнал транзакций резервируется, в него копируются все эти страницы на место зарегистрированных маркеров.

Компромисс в производительности операций в этой модели восстановления достигается за счет того, что массовые операции не рассматриваются как транзакции. В то время как журнал транзакций остается относительно маленьким по размерам, копирование всех экстентов, на которые повлияли массовые операции, в резервную копию журнала может привести к тому, что файл резервной копии увеличится до гигантских размеров.

Поскольку массовые операции в данной модели восстановления не регистрируются, если сбой произошел после массовой операции, но до создания архива транзакций, массовая операция будет утеряна и в восстановлении сможет участвовать только предыдущая резервная копия журнала транзакций. Исходя из этого, если используется модель восстановления с неполным протоколированием, за каждой массовой операцией должно немедленно следовать создание резервной копии журнала транзакций.

Эта модель восстановления может оказаться полезной только в том случае, если в базе данных выполняется большое количество массовых операций и очень важно повысить их производительность. Если производительность массовых операций в базе данных вас удовлетворяет, то лучше остановить свой выбор на полной модели восстановления.

Следует отметить, что простая модель восстановления вообще не регистрирует массовые операции.

Использование этой модели аналогично установке параметра базы данных `Select Into/Bulkcopy` в значение `true`.

Установка модели восстановления

Модель восстановления, установленная в системной базе `model`, применяется ко всем создаваемым базам данных. Установка модели восстановления выполняется во вкладке Options диалогового окна Database Properties утилиты Management Studio. Чтобы открыть это окно, щелкните правой кнопкой мыши на базе данных и выберите в контекстном меню пункт Properties.

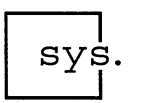
Модель восстановления можно изменить и программным путем; для этого используется инструкция DDL `ALTER DATABASE`:

```
ALTER DATABASE имя_базы_данных SET Recovery параметр;
```

Допустимыми параметрами в этой инструкции являются следующие: `Full`, `Bulk-Logged` и `Simple`. В следующем примере модель восстановления учебной базы данных `CHA2` изменяется на полную:

```
USE CHA2;
ALTER DATABASE CHA2 SET Recovery Full;
```

Настоятельно рекомендуется в программном коде, создающем базу данных, явно указывать модель восстановления.



Текущую модель восстановления, установленную в базе данных, можно определить с помощью представления каталога `sys.Database`:

```
SELECT [name], recovery_model_desc
  FROM sysdatabases;
```

Изменение модели восстановления

Несмотря на то что в базе данных обычно устанавливается только одна модель восстановления, никто не запрещает вам переключаться между разными моделями в зависимости от состава выполняемых операций с целью оптимизации производительности и удовлетворения текущих потребностей.

Будет совершенно резонно в течение рабочего дня для регистрации текущих операций переключаться на полную модель восстановления, а по вечерам заменять ее моделью с неполным протоколированием и выполнять необходимые операции импорта данных.

Во время восстановления берутся в расчет существующие полные, дифференцированные и транзакционные резервные копии — систему совершенно не волнует, как именно они создавались.

Так как простая модель восстановления не хранит постоянно весь набор транзакций, переключение в нее или из нее требует особого подхода.

- Если вы переключаетесь в простую модель восстановления, перед этим не забудьте создать резервную копию журнала транзакций.
- Если вы переключаетесь из простой модели восстановления, непосредственно после переключения следует выполнить полное резервирование базы данных.

Резервирование базы данных

Реальный процесс выполнения резервирования предлагает вам массу вариантов, каждый из которых будет рассмотрен в следующих подразделах.

Устройства резервирования

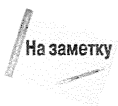
Резервная копия может создаваться на одном из двух возможных носителей.

- **Дисковая подсистема**. Резервирование может выполняться как на локальный диск (желательно в дисковой подсистеме, отличной от той, где хранятся файлы базы данных), так и на жесткий диск другого сервера с использованием *универсального соглашения об именовании*. Для сохранения файла архива на удаленном сервере учетная запись SQL Server должна иметь соответствующие привилегии.



Настоятельно рекомендуется вначале создавать резервную копию в файловой системе удаленного сервера, а затем копировать созданные файлы на магнитную ленту или DVD (последнее относится к небольшим базам данных), используя общепринятый в организации метод резервирования информации. С точки зрения SQL Server этот метод самый быстрый, к тому же он позволяет персоналу информационного отдела использовать единую стратегию резервирования информации компании. Если такой подход приводит к возникновению узких мест в сети, используйте выделенное подключение SQL Server к файловому серверу.

- **Магнитная лента**. SQL Server позволяет резервировать данные напрямую на большинство лентопротяжных устройств.



Некоторые компании предлагают сторонние продукты резервирования данных SQL Server, использующие именованные каналы. Несмотря на то что эти программы могут показаться вам привлекательными, советую все же ознакомиться и со встроенными методами восстановления SQL Server, прежде чем принимать решение относительно выбора стратегии резервирования.

Архивные файлы на диске и на ленте не ограничены только одним событием резервирования данных. Один файл может содержать несколько резервных копий, причем даже разных типов.

Хранение и ротация резервных копий

Если файл резервной копии создается на магнитной ленте, становится немаловажным вопрос об определении места хранения ленты и ротации лент.

Чаще всего используют следующую стратегию. Используется набор из пяти магнитных лент для еженедельной полной резервной копии и еще один набор из шести магнитных лент для ежедневных дифференцированных копий. Обычно эти ленты маркируются следующим образом: `Неделя1`, `Неделя2` и т.д. и соответственно `Понедельник`, `Вторник`, ..., `Суббота`.

Еще одним отличным методом ротации магнитных лент является так называемый *палиндром*. Палиндромом называют слово, фразу или число, которое в прямом и обратном порядке читается одинаково, например *около Мити молоко* или *123454321*. Некоторые числа, будучи перевернутыми и сложенными сами с собой, дают в результате палиндром, например `236+632=868`. У палиндромов богатая история. Еще в Древней Греции писали на фонтанах "Nispon anomemata me monan opsin", что в переводе означает "смывай грехи, как моешь свое лицо".

Используя четыре ленты, маркированные буквами от `А` до `Г`, ротацию лент можно организовать следующим способом: `АБВГВБА АБВГВБА...`

В качестве альтернативы метод палиндрома можно организовать так, чтобы каждая следующая буква представляла относительно больший интервал. Например, букву `А` можно использовать для ежедневных архивов, `Б` — для еженедельных, `В` — для ежемесячных, `Г` — для ежеквартальных и т.д.

Ротация магнитных лент вне рабочего места является важным аспектом плана восстановления. В идеальном случае можно поддерживать альтернативную площадку восстановления, укомплектованную сервером и рабочими станциями.

Выполнение резервного копирования в Management Studio

Первая резервная копия должна быть полной; с нее начинается цикл резервирования базы данных.

Резервирование базы данных можно выполнить в утилите Management Studio. Для этого нужно выделить базу данных, щелкнуть правой кнопкой мыши и выбрать в контекстном меню пункт Tasks⇨Backup. Откроется окно Back Up Database, показанное на рис. 36.3.

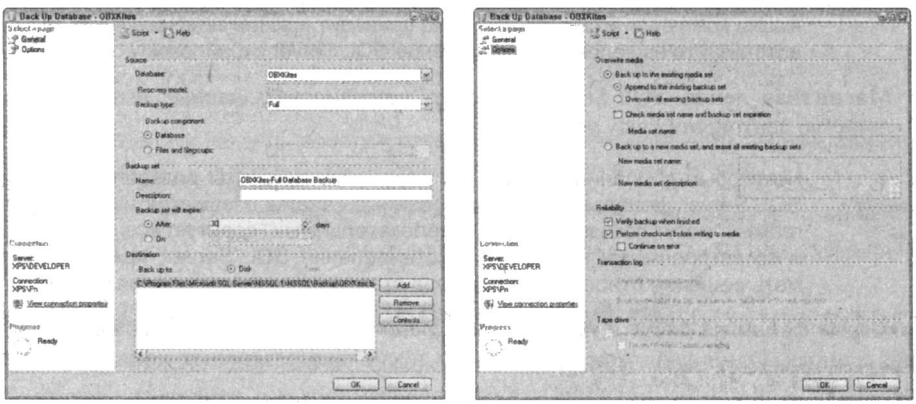


Рис. 36.3. Окно Back Up Database имеет две страницы: одну с общими настройками, другую — с параметрами

Источник данных резервного копирования конфигурируется на странице General.

- Database. Резервируемая база данных. По умолчанию это текущая база данных в Management Studio.
- Backup. Тип резервного копирования: full, differential или transaction-log. Если в базе данных установлена простая модель восстановления, то вариант transaction-log будет недоступным. Если выбирается полное или дифференцированное резервирование, то резервная копия может создаваться для всей базы данных или для выделенных файлов или файловых групп.

Все остальные элементы страницы предназначены для определения приемника резервной копии.

- Name. Обязательное имя резервной копии.
- Description. Необязательная дополнительная информация о резервной копии.
- Expiration Date. SQL Server не допустит замещение данного архива другим до этой даты истечения срока годности.
- Destination. Файл назначения на ленте или диске. Если текущий файл назначения не корректен, удалите его и добавьте корректный.
- Contents. Список резервных копий, уже существующих в файле назначения.

На странице Options представлены следующие параметры.

- Append to existing backup set или Overwrite all existing backup sets. Этот параметр определяет, будет ли новая резервная копия добавлена в существующий файл архива, или будет инициализирован носитель и в него будет помещен новый набор резервных копий. Дополнительно SQL Server может проверить даты истечения срока годности переписываемых наборов.
- Verify backup upon complution. Несмотря на свое название, этот параметр не сравнивает данные резервной копии с базой данных; также он не проверяет целостность архива. Он позволяет просто получить подтверждение о завершении формирования набора резервных копий и о том, что файл читаемый. Как бы там ни было, я всегда устанавливаю этот параметр.
- Eject tape after backup. Извлекает ленту из привода, что позволяет случайно не переписать существующий файл архива.
- Remove inactive entries from the log. Это эквивалент Management Studio операции сжатия журнала транзакций. Как только журнал транзакций был успешно заархивирован, из него удаляются все транзакции, чтобы файл журнала не разрастался бесконечно.
- Check media set name and backup set expiration. Проверяет носитель резервной копии на корректность.
- Backup set will expire. Установка срока годности резервной копии. До истечения указанного срока данные архива не могут быть замещены другими.
- Initialize and labels. Указывает серверу инициализировать ленту и присвоить ей метку.

Резервирование базы данных в программном коде

Команда backup предлагает больше вариантов резервного копирования, чем утилита Management Studio. Использование этой команды может оказаться полезным для ручной

сборки заданий агента SQL Server Agent, что считается более правильным, нежели использование плана обслуживания Maintenance Plan Back Up Task.

Если не принимать в расчет все дополнительные параметры, то общий вид команды резервного копирования будет следующим:

```
BACKUP DATABASE имя_базы_данных
  TO DISK = 'путь_к_файлу'
  WITH
    NAME = 'имя_архива'
```

Следующая инструкция резервирует базу данных CHA2 в файл на диске и присваивает архиву имя CHA2Backup:

```
BACKUP DATABASE CHA2
  TO DISK = 'e:\Cha2Backup.bak'
  WITH
    NAME = 'CHA2Backup'
```

Будет получен следующий результат:

```
Processed 200 pages for database 'CHA2',
  file 'CHA2' on file 1.
Processed 1 pages for database 'CHA2',
  file 'CHA2_log' on file 1.
BACKUP DATABASE successfully processed 201 pages
  in 0.316 seconds (5.191 MB/sec).
```

Команда резервного копирования имеет ряд важных параметров, которые хотелось бы отметить первыми.

- `Tape`. Чтобы создавать резервную копию на магнитной ленте, а не на диске, используйте параметр `TO TAPE` и укажите место размещения архива.

  ```
  TAPE = '\\.\TAPE0'
  ```

- `Differential`. Указывает команде выполнить дифференцированное резервное копирование, а не полное, принятое по умолчанию. Следующая команда выполнит дифференцированное резервирование базы данных CHA2:

  ```
  BACKUP DATABASE CHA2
    TO DISK = 'e:\Cha2Backup.bak'
    WITH
      DIFFERENTIAL,

  NAME = 'CHA2Backup'
  ```

- Для выполнения резервного копирования отдельных файлов данных или файловой группы перечислите их после имени базы данных. Этот прием поможет правильно организовать резервные копии. Например, из соображений удобства резервирования учебная база данных `OBXKites` хранит статические таблицы в одной файловой группе, а активные таблицы в другой — первичной.
- `Password`. Если резервная копия создается на незащищенной ленте, настоятельно рекомендуется защитить архив паролем. Пароль применяется к конкретному экземпляру архива.

Команда резервного копирования имеет ряд дополнительных, менее важных параметров.

- `Description`. Этот параметр идентичен одноименному полю Management Studio.
- `ExpireDate`. Идентичен Management Studio. До истечения указанного срока данные архива не могут быть замещены другими.

- `RetainDays`. Количество дней (целое число) до того, как SQL Server может заместить резервную копию.
- `Stats = %`. Указывает серверу информировать о ходе выполнения резервного копирования с указанным шагом (в процентах). По умолчанию принят шаг в 10%.
- `BlockSize`. Установка размера блока в архиве. Для резервных копий, сохраняемых на диске, данный параметр не нужен. Для копий, создаваемых на ленте, он тоже может не понадобиться, однако он способен решить некоторые проблемы совместимости. При создании резервных копий на диске автоматически применяется размер блока, установленный в операционной системе Windows. Если объем диска больше 2 Гбайт, то размер блока обычно устанавливается в 4096 байт. Если резервную копию, создаваемую на диске, впоследствии планируется записать на компакт-диск, то лучше установить размер блока в 2048 байт.
- `MediaName`. Определяет имя тома носителя. Этот параметр используется из соображений безопасности. Если архив добавляется в существующий носитель, то имена носителей в строке и приводе должны совпадать.
- `MediaDescription`. Записывает необязательное описание носителя.
- `MediaPassword`. Создает необязательный пароль и применяет его ко всему носителю (дисковому файлу или магнитной ленте). Этот пароль также может быть создан при первом создании носителя. Если пароль на носителе уже установлен, то он должен указываться при каждом последующем к нему обращении (при добавлении дополнительных резервных копий и при восстановлении данных).
- `Init/NoInit`. Инициализирует ленту или файл на диске, переписывая все существующие наборы резервных копий на носителе. SQL Server не допустит инициализации, если у любого из архивов на носителе еще не истек срок годности или если не истекло количество дней его хранения. По умолчанию используется параметр `NoInit`.
- `NoSkip/Skip`. Этот параметр позволяет пропустить проверку имени и срока годности, что при обычных условиях может запретить замещение существующей резервной копии. По умолчанию используется параметр `NoSkip`.

Следующие параметры применяются только при создании резервных копий на магнитной ленте.

- `NoFormat/Format`. Перед созданием резервной копии магнитная лента (но не диск!) форматируется. При форматировании автоматически устанавливаются параметры `Skip` и `Init`.
- `Rewind/NoRewind`. Указывает серверу автоматически перемотать ленту в начало (используется по умолчанию).
- `UnLoad/Load`. Автоматически перематывает ленту в начало и выгружает ее. Этот параметр используется по умолчанию, пока в другой сессии не будет явно определен параметр `Load`.
- `Restart`. Если резервное копирование с участием нескольких магнитных лент сорвалось до завершения операции, то параметр `Restart` поможет продолжить последовательность, начиная с точки сбоя, без необходимости возвращения к первой магнитной ленте. Использование этого параметра позволяет сэкономить время, однако после завершения операции резервирования не забудьте запустить проверку архива с помощью команды `restore verifyonly` (подробнее об этом — в следующем разделе).

Программная проверка резервной копии

При создании резервной копии в Managenet Studio у пользователя есть возможность проверить архив, чего не позволяет сделать команда `backup`. После завершения архивирования Management Studio вызывает команду `restore verifyonly` для выполнения верификации:

```
RESTORE VERIFYONLY
  FROM DISK = 'e:\Cha2Backup.bak'
```

В случае успешной проверки будет получен следующий результат:

```
The backup set is valid.
```

Проверка архива имеет несколько параметров, таких как `Eject tape after backup`. Большинство этих параметров предназначено только для магнитных лент (к тому же их название говорит само за себя).

Работа с журналом транзакций

Иногда возникает впечатление, что журнал транзакций живет собственной жизнью. Пространство внутри файла журнала может разрастаться и сжиматься, без каких бы то ни было видимых причин. Если вы почувствовали себя потерянным при работе с журналом, то вы не одиноки. Настоящий раздел прольет некоторый свет на внутреннюю жизнь журнала транзакций.

Внутренний мир журнала транзакций

Журнал транзакций содержит все транзакции базы данных. Если сервер уничтожает журнал транзакций, все записанные в него транзакции используются для восстановления. При этом откатываются все частично выполненные транзакции и завершаются те, которые были подтверждены в журнале транзакций, но еще не записаны в файл данных.

Виртуально этот журнал можно представить себе как последовательный список транзакций, отсортированный по дате и времени. В то же время на физическом уровне SQL Server записывает различные части физического журнала в виртуальные блоки без какого-либо определенного порядка. Некоторые его части могут находиться в эксплуатации, в то время как другие части оказываются доступными для перезаписи, подобно перевертышу.

Деление на активные и неактивные части

Все транзакции в журнале можно разделить на две группы (рис. 36.4).

- **Активные транзакции** — это те, которые еще не подтверждены и не записаны в файл данных.
- **Неактивные транзакции** — которые предшествуют самой ранней активной транзакции.

Так как все транзакции имеют разную длительность и подтверждаются в разное время, вероятность того, что в активной части журнала будут находиться подтвержденные транзакции, достаточно высока. Активная часть журнала не обязательно содержит только неподтвержденные транзакции — она содержит все транзакции с момента начала самой старой неподтвержденной транзакции. Всего одна очень старая неподтвержденная транзакция может сделать активной очень большую часть журнала.

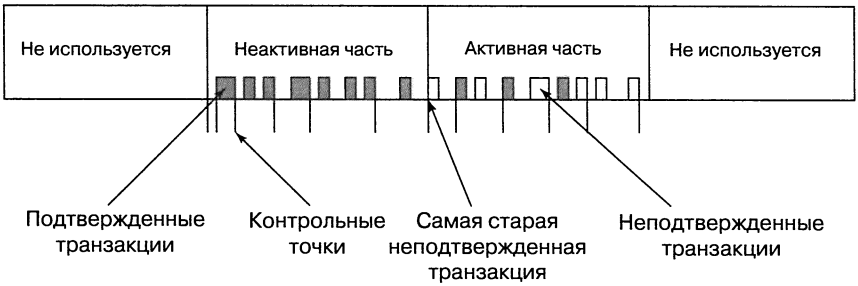


Рис. 36.4. Неактивные транзакции предшествуют самой старой активной

Контрольные точки транзакции

Понимание того, как SQL Server использует контрольные точки в журнале транзакций, исключительно важно для понимания процесса архивирования и очистки журнала. Контрольные точки вычисляют объем работы, необходимый для восстановления базы данных.

Контрольная точка автоматически устанавливается при выполнении одного из приведенных ниже условий.

- Когда инструкция `ALTER DATABASE` изменяет некоторый параметр базы данных.
- Когда сервер завершает работу.
- Когда количество записей в журнале превосходит ожидаемый объем работ, установленный в параметре конфигурации сервера `recovery interval`.
- Если в базе данных установлена простая модель восстановления или режим сжатия, а журнал транзакций заполнен на 70%.

Контрольные точки можно инициировать и вручную, с помощью команды `checkpoint`. При установке контрольной точки выполняются следующие действия.

- В журнале транзакций маркируется зона контрольной точки.
- В журнал заносится запись об установке контрольной точки, содержащая следующее:
  - самую старую активную транзакцию;
  - самую старую транзакцию, которая еще не была реплицирована в репликации транзакций;
  - список всех активных транзакций;
  - информацию об объеме работ, необходимых для отката базы данных.
- На диск записываются все заполненные страницы данных и журнала

В сущности, контрольная точка подводит черту под текущим состоянием базы данных, а затем записывает состояние разделительной линии между активной и неактивной частью журнала

Резервирование журнала транзакций

Выполнение резервирования журнала транзакций практически не отличается от полного или дифференцированного резервного копирования. Существует только несколько заметных отличий.

Инструкция T-SQL имеет следующий вид:

```
BACKUP LOG CHA2
  TO DISK = 'e:\Cha2Backup.bak'
  WITH
    NAME = 'CHA2Backup'
```

Будет получен следующий результат:

```
Processed 1 pages for database 'CHA2',
  file 'CHA2_log' on file 9.
BACKUP LOG successfully processed 1 pages
  in 0.060 seconds (0.042 MB/sec).
```

При резервировании журнала транзакций используются те же параметры, которые используются при резервном копировании базы данных. Но доступны и два дополнительных параметра, имеющие смысл только для резервирования журнала транзакций. Параметр `no_truncate` предназначен для резервирования журнала транзакций во время выполнения операции восстановления; параметр `norecovery/standby` предназначен для запуска резервирования на приостановленном сервере. Оба этих параметра мы рассмотрим более детально в разделе "Восстановление с помощью программного кода T-SQL".

Журнал транзакций не может быть зарезервирован, если выполняется любое из следующих условий.

- В базе данных используется простая модель восстановления.
- В базе данных используется модель восстановления с неполным протоколированием, была выполнена массовая операция, и файлы базы данных были повреждены.
- Файлы базы данных были добавлены или удалены.

В любом из этих случаев следует выполнить полное резервирование базы данных.

Сжатие журнала транзакций

Операции обновления и удаления могут не увеличить размер файла данных. В то же время в журнал транзакций данные добавляет любая операция. Журнал транзакций продолжает увеличиваться с каждой операцией модификации данных.

Решение данной проблемы заключается в возвращении к неактивной части журнала и последующем ее удалении. По умолчанию резервирование журнала транзакций приводит к автоматическому его усечению (т.е. сжатию) (см. рис. 36.3).

Если, к примеру, диск переполнен, то журнал транзакций может потребовать усечения без резервирования базы данных. Однако не существует способа сжатия журнала без резервирования. В то же время в языке T-SQL предусмотрен вариант усечения журнала транзакций с помощью команды `Backup...NoLog` или `Backup...TruncateOnly` (эти команды взаимозаменяемы):

```
BACKUP LOG CHA2
  WITH TRUNCATE_ONLY
```



Если журнал транзакций усечен вручную, а затем была создана его резервная копия, то в последовательности транзакций образуется брешь. Любые последовательности транзакций, находящиеся после этой бреши, не могут быть восстановлены. Чтобы заново начать создавать последовательность резервных копий журнала транзакций, рекомендуется выполнить полное резервирование базы данных.

Журнал транзакций и простая модель восстановления

Когда в базе данных используется простая модель восстановления, журнал транзакции гарантирует то, что все подтвержденные транзакции будут записаны в файл данных. Когда журнал транзакций заполняется на 70%, SQL Server устанавливает контрольную точку, а затем усекает журнал. При этом размеры свободного пространства журнала будут колебаться, однако минимальным будет размер его активной части.

Операции восстановления

Существует множество причин восстановления базы данных, среди которых следующие.

- Произошло повреждение дисковой подсистемы.
- Невнимательный программист забыл вставить предложение `WHERE` в инструкцию `UPDATE` и заменил зарплату всех сотрудников минимальной.
- Сервер упал в резервуар с силиконом, и винчестер превратился в булыжник.
- Была проделана большая работа по импорту данных, но не тем числом.

Лучше всего вначале опробовать цикл резервирования/восстановления и убедиться, что план восстановления работает. Если нет гарантий работоспособности плана, то нет никакого смысла выполнять резервирование данных.

Идентификация проблемы

Если существует какая-либо проблема, связанная с базой данных, Management Studio отображает ее как недоступную В этом случае, чтобы идентифицировать проблему, нужно просмотреть файл журнала (рис. 36.5).

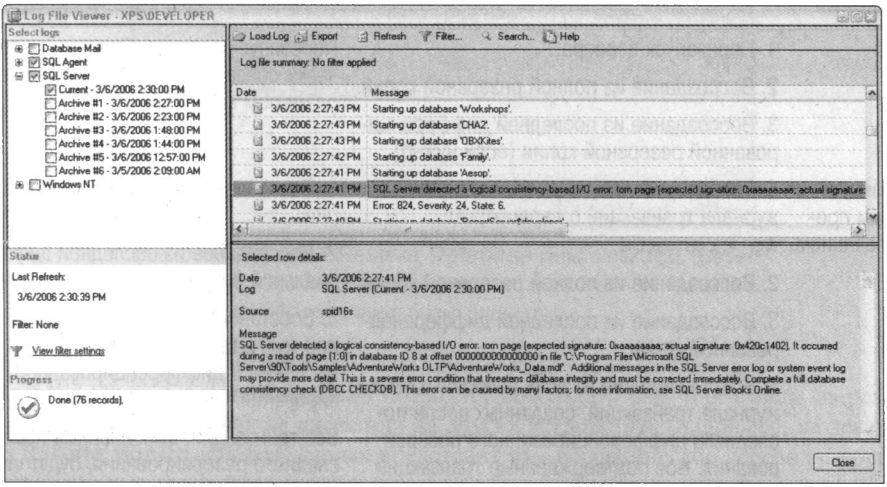


Рис. 36.5. База данных AdventureWorks повреждена (для этого я остановил SQL Server и подправил вручную в шестнадцатеричном редакторе заголовок файла). Management Studio отображает базу без дерева объектов. Ниже в дереве консоли находятся узлы журналов SQL Server

Чтобы продолжить исследование проблемы, просмотрите журнал SQL Server. Для этого в Management Studio выберите пункт меню Management⇨SQL Server Logs. SQL Server записывает ошибки и события в специальный файл журнала, который находится в подкаталоге `/error` папки установки сервера. Этот файл создается заново при каждом запуске SQL Server, при этом шесть предыдущих версий этого файла сохраняются там же. Некоторые ошибки параллельно могут быть записаны в журнал событий приложений операционной системы Windows.

Последовательности восстановления

С восстановлением данных связаны две очень важные концепции.

- Операция восстановления всегда начинается с воссоздания полной резервной копии, после чего воссоздаются дифференцированные и транзакционные архивы. В процессе воссоздания невозможно скопировать только вчерашнюю работу — реставрируется вся база данных в состоянии на определенный момент времени.
- Реставрация и восстановление — две различные операции. При реставрации данные копируются назад в базу, при этом транзакции остаются открытыми. Восстановление представляет собой процесс обработки транзакций, которые остались открытыми в журнале. Например, если в операции участвуют четыре файла архива журнала транзакций, только последний из них реставрируется с восстановлением.

Только члены фиксированной серверной роли `SysAdmins` могут воссоздавать базу данных, которая еще не существует. Те же базы данных, которые уже существуют, могут воссоздавать члены уже двух ролей: `SysAdnins` и `db_owners`.

Реальный состав работ при восстановлении зависит от типа повреждения и существующего плана восстановления. В табл. 36.2 приведены сравнительные списки операций восстановления в различных ситуациях.

Таблица 36.2. Последовательности восстановления

Модель восстановления	Поврежден файл данных	Поврежден журнал транзакций
Простая	1. Перезапуск сервера. 2. Воссоздание из полной резервной копии. 3. Воссоздание из последней дифференцированной резервной копии (если нужно)	Перезапуск сервера. При этом автоматически создается новый журнал транзакций размером в 1 Мбайт
Полная или с неполным протоколированием	1. Резервирование текущего состояния журнала транзакций с параметром `no_truncate`. 2. Воссоздание из полной резервной копии. 3. Воссоздание из последней дифференцированной копии. 4. Восстановление из всех резервных копий журнала транзакций, созданных после последнего дифференцированного резервирования. Все подтвержденные транзакции будут восстановлены	1. Воссоздание из полной резервной копии. 2. Воссоздание из последней дифференциальной резервной копии (если нужно). 3. Восстановление из всех резервных копий журнала транзакций, созданных после последнего дифференцированного резервирования. Все транзакции, выполненные после последнего резервирования, будут утеряны

Если в базе данных используется модель восстановления с неполным протоколированием и с момента последнего резервирования журнала транзакций была выполнена какая-либо массовая операция, то восстановление не произойдет. Все транзакции, выполненные после последнего резервирования журнала транзакций, не восстанавливаются.

Воссоздание базы данных в Management Studio

Как и в случае резервирования, существует несколько способов запуска восстановления в утилите Management Studio.

- Выделите базу данных, которую будете воссоздавать. В контекстном или главном меню Action выберите пункт All Tasks⇨Backup Restore.
- Выберите в меню пункт Tasks⇨Restore⇨Database.
- Щелкните правой кнопкой мыши на узле `Databases` дерева консоли и выберите в контекстном меню пункт Restore Database.

В любом случае откроется форма Restore Database (рис. 36.6), которая проведет вас сквозь потенциальный хаос последовательностей восстановления; она всегда предлагает только допустимые варианты воссоздания.

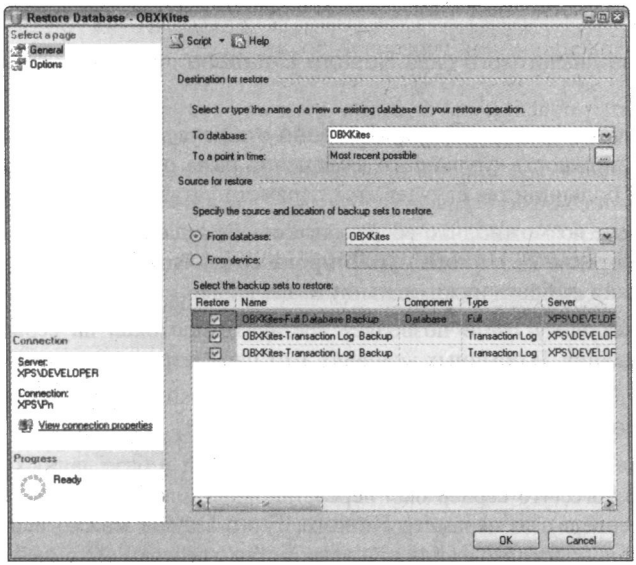


Рис. 36.6. В форме Restore Database утилиты SQL Server Management Studio доступны только корректные последовательности воссоздания базы данных

В верхней части окна выберите имя базы данных после ее воссоздания.

Диалоговое окно Restore Database позволяет воссоздать базу или файлы из резервных копий, находящихся в файлах на диске или на других устройствах (например, на ленточных). Мастер Restore Wizard для баз данных представит вам иерархическое дерево резервных копий, в то время как для файлов и файловых групп будут перечислены файлы, которые должны быть восстановлены вручную в корректном порядке.

Параметр Show backups of database используется для выбора первой резервной копии из последовательности, которая подлежит воссозданию. Основываясь на выбранной

последовательности, в таблице отобразится иерархическое дерево возможных последовательностей архивов.

- Полные резервные копии базы данных отображаются с золотым значком жесткого диска на верхнем уровне дерева.
- Дифференцированные резервные копии отображаются с синим значком жесткого диска на втором уровне дерева.
- Резервные копии журнала транзакций отображаются со значком ноутбука на нижнем уровне дерева.

В зависимости от выбранных полных и дифференцированных резервных копий далее могут быть выбраны только определенные дифференцированные и транзакционные архивы.

Для обработки одной полной резервной копии, следующей за ней одной резервной копии и далее пятнадцати резервных копий журнала транзакций в правильной последовательности достаточно выбрать последний журнал транзакций, который должен быть восстановлен. Таким образом, воссоздание базы данных из всех семнадцати файлов резервных копий может быть выполнено с помощью всего одного щелчка на кнопке OK.

Если одним из воссоздаваемых файлов является резервная копия журнала транзакций, то становится доступным параметр Point in Time Restore. Этот параметр позволяет восстановить из копии журнала только часть транзакций, зарегистрированных до заданной точки во времени.

Вкладка Options диалогового окна Restore Database позволяет установить ряд важных параметров.

- Параметр Overwrite the existing database отключает проверку безопасности, исключающую возможность случайного замещения базы данных A резервной копией базы данных Б. В большинстве случаев этот параметр устанавливать не нужно.
- Так как вполне возможно, что воссоздаваемая база данных должна находиться на диске в месте, отличном от того, из которого создавалась резервная копия, во вкладке Options можно выбрать нужную папку назначения.
- Параметр Recovery State позволяет доставить протокол на сервер, находящийся в режиме ожидания. В обычных условиях этот параметр следует оставить рабочим.
- Если воссоздавать следует только определенные файлы и файловые группы, то параметр Restore: Files or File Groups позволит выбрать эти объекты.
- Если журнал создания резервных копий, хранимый в базе данных `msdb`, недоступен (по причине того, что сервер был переустановлен или база данных восстанавливается на другом сервере), то параметр Restore: From Device можно использовать для ручного выбора конкретного файла или устройства с резервной копией, а также экземпляра резервной копии в файле.



SQL Server способен частично воссоздавать страницы данных. Эта возможность может оказаться особенно полезной, когда в базе данных объемом в несколько терабайт повреждена всего одна страница данных. Несмотря на то что я очень благодарен компании Microsoft за предоставление такой возможности, мне кажется, что на практике она будет использоваться крайне редко, к тому же она может создать больше проблем, чем решить. Если в файле базы данных возникла ошибка, гораздо лучше потратить дополнительное время, выполнив полноценное воссоздание базы данных, чем пытаться по крупицам воссоздавать потерянные страницы. Исходя из вышесказанного, работа с данной функцией в книге описана не будет.

Воссоздание базы данных программным путем

Обычно резервное копирование базы данных выполняется по строго определенному графику, так что если вам не нравится работать с мастером плана обслуживания SQL Server, вы можете сами написать программный код выполнения резервирования, после чего настроить соответствующее задание агента SQL Server Agent.

Однако (если, конечно, план восстановления не является полным) невозможно заранее узнать, какое количество дифференцированных и транзакционных резервных копий должно быть восстановлено. Так как каждый файл резервной копии требует отдельной инструкции RESTORE, невозможно создать корректный код, не включив в него обследование базы данных msdb и создание правильной последовательности восстановления.

Инструкция RESTORE может выполнять воссоздание базы из полной, дифференцированной и транзакционной резервной копии. Ее общий синтаксис приведен ниже.

```
RESTORE DATABASE|LOG имя_базы_данных
  [файл|файловая_группа PARTIAL]
  FROM устройство_резервирования
  WITH
    FILE = номер_файла,
    PASSWORD = пароль,
    NORECOVERY|RECOVERY|STANDBY = имя_файла_отмены,
    REPLACE,
    STOPAT дата_и_время,
    STOPATMARK = 'имя_метки'
    STOPBEFOREMARK = 'имя_метки'
```

Для воссоздания полной или дифференциальной резервной копии используется инструкция RESTORE DATABASE; если воссоздается журнал транзакций, используется инструкция RESTORE LOG. Для воссоздания определенного файла или файловой группы добавьте ее имя сразу после имени базы данных. Если файл или файловая группа являются единственными воссоздаваемыми данными, добавьте в инструкцию параметр PARTIAL.

Набор резервных копий часто состоит из нескольких архивов. Например, состав файла резервной копии может быть таким.

1. Полная резервная копия.
2. Дифференцированная резервная копия.
3–6. Резервные копии журнала транзакций.
7. Дифференцированная резервная копия.
8, 9. Резервные копии журнала транзакций.

Параметр WITH FILE позволяет задать номер восстанавливаемой резервной копии в файле или на устройстве резервирования.

Параметры RECOVERY/NORECOVERY являются жизненно важными в инструкции воссоздания. При каждом запуске SQL Server автоматически проверяется журнал транзакций. При этом откатываются все неподтвержденные транзакции и доводятся до конца все подтвержденные. Этот процесс получил название *воссоздания* (recovery); он является составной частью свойств ACID базы данных.

Таким образом, если в инструкции воссоздания указан параметр NORECOVERY, SQL Server воссоздаст журнал, не обработав при этом ни одной транзакции; если же в инструкции будет указан параметр RECOVERY, транзакции будут обработаны. В последовательности операций восстановления все инструкции, кроме последней, должны иметь параметр NORECOVERY; в последней инструкции должен быть задан параметр RECOVERY.

Принятие решения относительно использования параметров RECOVERY и NORECOVERY является одним из самых сложных моментов написания сценария, который должен подойти ко всем возможным операциям восстановления базы данных в будущем.

Если в операции воссоздания базы данных участвует резервная копия журнала транзакций, то процесс может остановиться, не достигнув конца этого журнала. Параметры STOPAT и STOPATMARK позволяют не воссоздавать транзакции, следующие *за* определенным моментом времени или меткой. При использовании параметра STOPBEFOREMARK воссоздание транзакций останавливается перед началом транзакции, на которой установлена заданная метка.



Об особенностях транзакций в SQL Server и способах создания маркированных транзакций см. в главе 23.

В следующем сценарии продемонстрирован пример последовательности восстановления базы данных, в которой участвуют полная резервная копия и два архива журнала транзакций:

```
-- Пример резервирования и восстановления базы данных
CREATE DATABASE Plan2Recover;
```

Получим следующий результат:

```
The CREATE DATABASE process
  is allocating 0.63 MB on disk 'Plan2Recover'.
The CREATE DATABASE process
  is allocating 0.49 MB on disk 'Plan2Recover_log'.
```

Продолжаем операции:

```
USE Plan2Recover;

CREATE TABLE T1 (
  PK INT Identity PRIMARY KEY,
  Name VARCHAR(15)
  );
Go
INSERT T1 VALUES ('Full');
go
BACKUP DATABASE Plan2Recover
  TO DISK = 'e:\P2R.bak'
  WITH
    NAME = 'P2R_Full',
    INIT;
```

Получаем следующий результат:

```
Processed 80 pages for database 'Plan2Recover',
  file 'Plan2Recover' on file 1.
Processed 1 pages for database 'Plan2Recover',
  file 'Plan2Recover_log' on file 1.
BACKUP DATABASE successfully processed 81 pages
  in 0.254 seconds (2.590 MB/sec).
```

Продолжаем операции:

```
INSERT T1 VALUES ('Log 1');
go
BACKUP Log Plan2Recover
  TO DISK = 'e:\P2R.bak'
  WITH
    NAME = 'P2R_Log';
```

Получаем следующий результат:

```
Processed 1 pages for database 'Plan2Recover',
  file 'Plan2Recover_log' on file 2.
BACKUP LOG successfully processed 1 pages
  in 0.083 seconds (0.055 MB/sec).
```

Продолжаем операции:

```
INSERT T1 VALUES ('Log 2');
go
BACKUP Log Plan2Recover
  TO DISK = 'e:\P2R.bak'
  WITH
    NAME = 'P2R_Log';
```

Получаем следующий результат:

```
Processed 1 pages for database 'Plan2Recover',
  file 'Plan2Recover_log' on file 3.
BACKUP LOG successfully processed 1 pages
  in 0.083 seconds (0.065 MB/sec).
```

Продолжаем операции:

```
SELECT * FROM T1;
```

Текущее состояние таблицы T1:

```
PK       Name
-------- ---------
1        Full
2        Log 1
3        Log 2
```

Предположим, что в этот момент вдруг отключилось электричество и все устройства сгорели, за исключением устройства резервного копирования. Следующая операция воссоздания базы данных использует полную резервную копию и два архива журнала транзакций, которые мы только что создали. При просмотре программного кода особое внимание обратите на использование параметров RECOVERY и NORECOVERY:

```
-- Теперь выполняем воссоздание базы данных
Use Master;
RESTORE DATABASE Plan2Recover
  FROM DISK = 'e:\P2R.bak'
  WITH FILE = 1, NORECOVERY;
```

Получаем следующий результат:

```
Processed 80 pages for database 'Plan2Recover',
  file 'Plan2Recover' on file 1.
Processed 1 pages for database 'Plan2Recover',
  file 'Plan2Recover_log' on file 1.
RESTORE DATABASE successfully processed 81 pages
  in 0.089 seconds (7.392 MB/sec).
```

Продолжаем восстановление:

```
RESTORE LOG Plan2Recover
  FROM DISK = 'e:\P2R.bak'
  WITH FILE = 2, NORECOVERY;
```

Получаем следующий результат:

```
Processed 1 pages for database 'Plan2Recover',
  file 'Plan2Recover_log' on file 2.
RESTORE LOG successfully processed 1 pages
  in 0.009 seconds (0.512 MB/sec).
```

Продолжаем восстановление:

```
RESTORE LOG Plan2Recover
  FROM DISK = 'e:\P2R.bak'
  WITH FILE = 3, RECOVERY;
```

Получаем следующий результат:

```
Processed 1 pages for database 'Plan2Recover',
  file 'Plan2Recover_log' on file 3.
RESTORE LOG successfully processed 1 pages
  in 0.044 seconds (0.011 MB/sec).
```

Теперь мы можем проверить результат операции воссоздания базы данных:

```
USE Plan2Recover;
SELECT * from T1;

PK       Name
-------- ------
1        Full
2        Log 1
3        Log 2
```

Как было показано в приведенном выше примере, базы данных можно воссоздать и с помощью программного кода T-SQL, однако в данном случае использование утилиты Management Studio обеспечит более эффективную работу.

Восстановление системных баз данных

До сих пор в этой главе мы имели дело только с пользовательскими базами данных, но системные базы данных также являются важными объектами восстановления. База данных `master` содержит ключи базы данных, а также настройки системы безопасности; база данных `msdb` — графики и задания SQL Server, а также историю создания резервных копий. Полноценный план восстановления должен включать в себя воссоздание системных баз данных.

Резервное копирование базы данных `master`

Резервное копирование базы данных `master` ничем не отличается от резервирования пользовательских баз.

Обязательно резервируйте эту базу данных при выполнении операций, перечисленных ниже.

- Создание и удаление баз данных.
- Модификация системы безопасности за счет дополнения регистрационных записей и изменения ролей.
- Изменение каких-либо параметров конфигурации базы данных.

Поскольку база данных `msdb` хранит записи всех операций резервного копирования, после резервирования базы данных `master` заархивируйте и `MSDB`.

Восстановление базы данных `master`

Если база данных `master` была повреждена, то сервер не может быть запущен. Попытка запустить SQL Server в диспетчере служб операционной системы ни к чему не приведет. Попытка подключиться к экземпляру сервера в Management Studio приведет к открытию окна предупреждения о том, что сервер не существует или доступ к нему запрещен.

Единственным решением является запуск SQL Server в монопольном режиме (т.е. с параметром `-m`) и восстановление базы `master` так, будто она является обычной пользовательской базой данных.

Системная база данных `msdb`

Подобно базе данных `master`, база `MSDB` по умолчанию использует простую модель восстановления.

Так как эта база данных содержит информацию, относящуюся к заданиям и графикам агента SQL Server Agent, а также к истории создания резервных копий, она должна резервироваться в следующих ситуациях.

- Выполнение любого типа резервирования.
- Сохранение пакетов DTS.
- Создание новых заданий агента SQL Server Agent.
- Конфигурирование почты и операторов SQL Server Agent.
- Конфигурирование репликаций.
- Создание и изменение графиков заданий.

Резервирование базы данных `msdb` ничем не отличается от архивирования обычных пользовательских баз данных.

Для восстановления базы данных `msdb` следует перевести сервер баз данных в монопольный режим, как и в случае с базой данных `master`. В то же время данное восстановление отличается от обычного, поскольку текущая база данных `msdb`, в которой хранится история создания резервных копий, повреждена. Таким образом, архив `msdb` не может быть выбран как резервная копия базы данных — он должен быть выбран как устройство резервирования.

Для проверки дискового устройства на предмет наличия конкретных резервных копий используется кнопка Contents. Если на устройстве содержится несколько экземпляров архивов, то для выбора нужного используется диалоговое окно Contents. После выбора нужной резервной копии ее номер подставляется в форму воссоздания базы данных.

Полное воссоздание сервера баз данных

Если сервер баз данных вышел из строя, резервные копии всех баз данных должны быть воссозданы на новом сервере. Порядок действий при этом следующий.

1. В операционной системе Windows восстановите все учетные записи домена для поддержки аутентификации Windows.
2. Установите SQL Server и все доступные пакеты обновлений.
3. Переведите SQL Server в монопольный режим и восстановите базу данных `master`.
4. Воссоздайте базу данных `msdb`.
5. Если база данных `model` изменялась, восстановите и ее.
6. Восстановите все пользовательские базы данных.


Проверено

Гладкое выполнение операций восстановления является одним из самых главных навыков администратора баз данных. Лично я рекомендую вам потратить время и пройти процесс полного восстановления сервера баз данных на запасном сервере, взяв за основу резервные копии производственной базы данных. Это придаст вам уверенность в повседневной работе с базой данных.

Резюме

Цикл восстановления начинается с резервного копирования базы данных. Способность выживать после технических сбоев и фатальных ошибок пользователей является одним из самых критичных свойств ACID базы данных. Если база данных не обеспечивает живучесть транзакций, то ей нельзя доверять в полной мере. По этой причине планирование восстановления и журнал транзакций призваны обеспечить живучесть подтвержденных транзакций.

В следующей главе мы перейдем к вопросам защиты базы данных.

Обслуживание базы данных

ГЛАВА

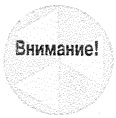
37

В этой главе...

Использование утилиты DBCC для проверки целостности базы данных

Создание плана обслуживания базы данных

Утилита DBCC всегда являлась важным инструментом обслуживания базы данных. СУБД SQL Server 2005 упростила задачу обслуживания базы, поэтому во многих эксплуатационных задачах, привычных по предыдущим версиям сервера, отпала необходимость. Мастер плана обслуживания поможет сконфигурировать задания агента SQL Server Agent в целостную систему поддержки работоспособности базы данных.



Утилита DBCC теперь интегрирована в системное приложение Dr. Watson. При обнаружении этой утилитой каких-либо повреждений базы данных приложение Dr. Watson отправляет отчет об ошибке в компанию Microsoft.

Команды утилиты DBCC

Для обслуживания SQL Server компания Microsoft предлагает утилиту DBCC, которая насчитывает тридцать четыре команды.

В первую очередь стоит ознакомиться с командой `help`, которая выводит на экран синтаксис всех доступных в утилите DBCC команд.

```
DBCC Help ('CheckDB');
```

Будет получен следующий результат:

```
CheckDB [('имя_базы_данных'
[, NOINDEX | REPAIR])]
[WITH NO_INFOMSGS[, ALL_ERRORMSGS]
[, PHYSICAL_ONLY]
[, ESTIMATEONLY][, TABLOCK]]

DBCC execution completed. If DBCC printed
error messages,contact your system
administrator.
```

Все команды утилиты DBCC выводят информацию о выполняемых действиях и обнаруженных ошибках, после чего предлагают пользователю обратиться со списком ошибок к системному администратору. Если вы считаете себя профессионалом, то лучше вас с данной утилитой никто не справится.



Для поддержания обратной совместимости с предыдущими версиями сервера в утилиту DBCC было включено несколько устаревших команд, которые в настоящей главе мы рассматривать не будем. Например, команда DBCC ROWLOCK больше не используется, поскольку SQL Server 2005 выполняет блокировку строк автоматически.



Полный список устаревших команд утилиты DBCC, в том числе поддерживаемых в новой версии сервера баз данных, вы найдете по адресу:
http://msdn2.microsoft.com/en-US/library/ms144262.aspx

Проверка целостности базы данных

Команда DBCC CheckDB выполняет серию проверок целостности внутренней структуры базы данных. Корректность физической структуры исключительно важна для правильного функционирования базы данных. Эта команда выполняет проверку таких объектов, как указатели индексов, смещения страниц данных, связи между страницами индексов и данных, а также структуры страниц индексов и данных. Если в результате сбоя питания какая-либо страница базы данных была записана только наполовину, то лучшим средством обнаружения и устранения ошибки будет команда DBCC CheckDB. В качестве примера проверим целостность учебной базы данных OBXKites:

```
DBCC CheckDB ('OBXKites');
```

Будет получен следующий результат (приводится с сокращениями):

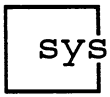
```
DBCC results for 'OBXKites'.
DBCC results for 'sysobjects'.
There are 114 rows in 2 pages for object 'sysobjects'.
DBCC results for 'sysindexes'.
There are 77 rows in 3 pages for object 'sysindexes'.
...
DBCC results for 'ProductCategory'.
There are 8 rows in 1 pages for object 'ProductCategory'.
DBCC results for 'Product'.
There are 55 rows in 1 pages for object 'Product'.
CHECKDB found 0 allocation errors
and 0 consistency errors in database 'OBXKites'.
DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```



Полученный вами результат может отличаться от приведенного в книге. Все зависит от конфигурации сервера и тех изменений, которые были внесены в базу данных.

С помощью двух параметров — all_errormsgs и no_infomsgs — можно задать необходимый уровень детализации информационных сообщений. Параметр estimate_only позволяет получить примерный размер базы данных tempdb, необходимый для выполнения команды CheckDB.

Если база данных слишком большая, можно воспользоваться параметром noindex и пропустить проверку целостности всех пользовательских некластеризованных индексов. Еще больше времени поможет сэкономить параметр Physical_Only — он выполнит только самую критичную проверку физической структуры страниц. Вы можете воспользоваться этим параметром, если нет времени для полноценной проверки базы командой CheckDB или если индексы были недавно созданы.



Чтобы получить список системных таблиц текущей базы данных, можете выполнить запрос к представлению каталога `sysobjects`, указав в качестве фильтра выражение `type_desc='SYSTEM TABLE'`. То же можно сделать и для других типов объектов, в частности, для пользовательских таблиц (`USER_TABLE`), различных типов ограничений (`PRIMARY_KEY_CONSTRAINT`, `DEFAULT_CONSTRAINT`, `CHECK_CONSTRAINT`, `UNIQUE_CONSTRAINT` и `FOREIGN_KEY_CONSTRAINT`), хранимых процедур (`SQL_STORED_PROCEDURE`), триггеров (`SQL_TRIGGER`) и представлений (`VIEW`). Вы также откроете для себя специальные объекты SQL Server, такие как внутренние таблицы (`INTERNAL_TABLE`) и служебные запросы (`SERVICE_QUEUE`).

Воссоздание базы данных

Если при проверке базы данных были выявлены ошибки, утилита DBCC попытается их устранить. Эта операция отличается от обычной проверки целостности, так как база данных должна быть предварительно переведена в монопольный (т.е. однопользовательский) режим с помощью хранимой процедуры `sp_dboption`. Только после этого можно будет выполнить команду DBCC `CheckDB` с параметром `Repair_Rebuild`. Не забудьте после проверки восстановить исходный режим работы базы данных.

```
EXEC sp_dboption OBXKites, 'single_user', 'True';
DBCC CheckDB ('OBXKites', Repair_Rebuild);
GO
EXEC sp_dboption OBXKites, 'Single_user', 'False';
GO
```

Будет получен следующий результат (приводится с сокращениями):

```
DBCC results for 'OBXKites'.
Service Broker Msg 9675, State 1: Message Types analyzed: 14.
Service Broker Msg 9676, State 1: Service Contracts analyzed: 6.
Service Broker Msg 9667, State 1: Services analyzed: 3.
...
DBCC results for 'OrderPriority'.
There are 1 rows in 1 pages for object "OrderPriority".
DBCC results for 'ProductCategory'.
There are 8 rows in 1 pages for object "ProductCategory".
CHECKDB found 0 allocation errors and 0 consistency errors in
database 'OBXKites'.
DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```

Утилита DBCC предлагает три режима восстановления, каждый из которых выполняет более тщательную проверку, чем предыдущий.

- `Repair_Fast`. Самый простой режим проверки. Восстанавливает ключи некластеризованных индексов, но оставляет нетронутыми страницы данных.
- `Repair_Rebuild`. Средний уровень проверки. Выполняет полную проверку некластеризованных индексов и индексных указателей, но не затрагивает страницы данных.
- `Repair_Allow_Data_Loss`. Наиболее сложная проверка. Воссоздает и перестраивает все индексы вместе с указателями на страницы данных, удаляет все повреждения, обнаруженные на страницах данных. Во время обновления структуры страниц данных возможна частичная потеря информации.



Выполняйте команду DBCC CheckDB ежедневно и после каждого сбоя оборудования. Если будет обнаружена какая-либо ошибка, запустите режим Repair_Rebuild, чтобы попытаться восстановить базу данных перед использованием режима Repair_Allow_Data_Loss, допускающего потерю информации. Если вы имеете дело с особо крупными базами данных при наличии требования повышенной доступности, то можете выполнять проверку раз в неделю в период снижения активности пользователей.

Многопользовательский режим

В версии SQL Server 2005 стало возможно выполнять команду DBCC CheckDB, не требуя выхода других пользователей из базы данных. Все процессы будут разбиты на отдельные потоки, задействующие все доступные процессоры. Следует отметить, что команда CheckDB интенсивно использует процессор и дисковую подсистему. По этой причине запускайте ее тогда, когда к базе подключено минимальное число пользователей.



Теперь одновременно с командой DBCC CheckDB вы можете выполнять команды DBCC CHECKALLOC, DBCC CHECKTABLE и DBCC CHECKCATALOG.

Если одновременно с выполнением команды DBCC CheckDB с базой данных работают пользователи, то утилита DBCC использует стандартную блокировку схемы. Параметр TabLock несколько снижает гранулярность блокировок. Несмотря на то что работа команды DBCC CheckDB будет менее эффективной, конкуренция в базе данных станет выше, в результате чего пользователи смогут продолжить работу с ней.

```
DBCC CheckDB ('OBXKites') With TabLock;
```

Будет получен следующий результат (приводится с сокращениями):

```
DBCC results for 'OBXKites'.
DBCC CHECKDB will not check SQL Server catalog or Service Broker
consistency because a database snapshot could not be created
or because WITH TABLOCK was specified.
DBCC results for 'syssysrowsetcolumns'.
There are 654 rows in 6 pages for object "syssysrowsetcolumns".
DBCC results for 'syssysrowsets'.
...
DBCC results for 'ProductCategory'.
There are 8 rows in 1 pages for object "ProductCategory".
CHECKDB found 0 allocation errors and 0 consistency errors in
database 'OBXKites'.
DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```



При использовании параметра TabLock в вывод добавляется дополнительная строка *"DBCC CHECKDB will not check SQL Server catalog or Service Broker consistency because a database snapshot could not be created or because WITH TABLOCK was specified"*. Это предупреждение о том, что проверка каталога SQL Server и целостности брокера служб выполняться не будет, поскольку моментальный снимок базы данных не может быть создан либо использован параметр WITH TABLOCK. Это — чисто информационное сообщение, так что не стоит особо волноваться на этот счет. В то же время, когда вы встретите это сообщение, не используя параметр TabLock, проверьте, все ли необходимые службы запущены.

Проверка на уровне объектов

Команда DBCC CheckDB выполняет комплексную проверку целостности структуры базы данных. В то же время можно выполнить и отдельные проверки и при этом получить дополнительную информацию об отдельных объектах. По этой причине лучше выполнять команду DBCC CheckDB ежедневно, а специализированные версии, привязанные к объектам, использовать на этапе отладки.

Если база данных требует восстановления, всегда используйте полноценную версию CheckDB, не останавливаясь на следующих специализированных командах.

- DBCC CheckAlloc (*'база_данных'*). Подмножество команды CheckDB, проверяющее физическую структуру базы данных. Создается отчет с высокой степенью детализации. В нем перечисляется количество экстентов (занимающих 64 Кбайт или восемь страниц данных) и страницы данных, используемые всеми таблицами и индексами.
- DBCC CheckFileGroup (*'файловая группа'*). Команда, аналогичная CheckDB, но проверяющая всего одну заданную файловую группу.
- DBCC CheckTable (*'таблица'*). Выполняет несколько параллельных проверок таблицы.
- DBCC CleanTable (*'база_данных'*, *'таблица'*). Перераспределяет пространство, освобожденное после удаления из таблицы столбца с типом varchar, nvarchar, text или ntext. Фактически эта команда обновляет базу данных; она не включена в состав команды CheckDB, если не используется параметр полного восстановления. Исходя из этого, если обновление тестовых полей базы данных выполняется регулярно, рекомендуется включить эту команду в план ежедневного обслуживания базы данных.

Целостность данных

Над слоем физической структуры базы находится слой данных, проверка которого выполняется с помощью следующих команд. Перечисленные ниже три команды DBCC не входят в состав DBCC CheckDB — их нужно выполнять отдельно.

- DBCC CheckCatalog (*'база_данных'*). Проверка целостности системных таблиц базы данных для обеспечения ссылочной целостности таблиц, представлений, столбцов и типов данных. При нормальных условиях отчет не слишком детализирован. Эта команда не исправляет обнаруженные ошибки. Если в базе данных будут обнаружены ошибки, придется с помощью сценария заново создать базу или таблицу, после чего переместить сохранившиеся данные из старой таблицы в новую. Если ошибки не обнаружены, то отчет не содержит никакой полезной информации.
- DBCC CheckConstraints (*'таблица'*, *'ограничение'*). Проверка целостности указанного ограничения или всех ограничений таблицы. Для проверки ограничений эта команда фактически генерирует и выполняет запрос, после чего выводит отчет обо всех обнаруженных ошибках. Как и в случае с командой CheckCatalog, если ошибки не обнаружены, никакой ценной информации не выводится.
- DBCC CheckIdent (*'таблица'*). Проверяет согласованность значений столбца идентичности заданной таблицы. Если существует проблема, то для исправления ошибки корректируется следующее значение столбца идентичности. Если столбец идентичности поврежден, то его новое значение нарушит ограничения уникальности и первичного ключа, после чего новые строки не смогут быть вставлены в таблицу.

В следующем примере продемонстрировано использование команды DBCC CheckIdent:

```
Use CHA2;
DBCC CheckIdent ('Customer');
```

Будет получен следующий результат:

```
Checking identity information: current identity value '127',
current column value '127'.
DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```

Обслуживание индексов

Индексы являются своеобразным мостом между данными и запросом SQL. Так как данные постоянно вставляются и обновляются, индексы становятся фрагментированными, и статистика распределения данных теряет актуальность. При этом степень заполнения страниц далека от идеала. Обслуживание индексов необходимо для того, чтобы исправить складывающуюся ситуацию и не допустить снижения производительности.



Подробно о создании индексов см. в главе 17.

Фрагментация базы данных

По мере вставки информации на страницы данных и индексов страницы заполняются на 100%. Как только достигается этот порог, SQL Server выполняет разбиение страницы на две, каждая из которых заполнена примерно на 50%. Несмотря на то что такой подход решает проблемы, связанные с отдельными страницами, внутренняя структура базы данных становится фрагментированной.

Чтобы продемонстрировать эффект выполнения команды DBCC, влияющей на фрагментированные таблицы и индексы, нужна сравнительно большая таблица. В следующем сценарии создается подходящая таблица и ее некластеризованный индекс. Кластеризованным первичным ключом является глобальный уникальный идентификатор GUID. По этой причине вставка выполняется в различные части таблицы, что создает первопричину изрядной фрагментации.

```
USE Tempdb;
CREATE TABLE Frag (
  FragID UNIQUEIDENTIFIER NOT NULL DEFAULT NewID(),
  Col1 INT,
  Col2 CHAR(200),
  Created DATETIME DEFAULT GetDate(),
  Modified DATETIME DEFAULT GetDate()
);

ALTER TABLE Frag
  ADD CONSTRAINT PK_Frag
  PRIMARY KEY CLUSTERED (FragID);

CREATE NONCLUSTERED INDEX ix_col
  ON Frag (Col1);
```

Запуск следующей хранимой процедуры приведет к вставке в таблицу ста тысяч строк:

```
CREATE PROC Add100K
AS
SET nocount on;
```

```
DECLARE @X INT;
SET @X = 0;
WHILE @X < 100000
 BEGIN
  INSERT Frag (Col1,Col2)
    VALUES (@X, 'sample data');
  SET @X = @X + 1;
 END
GO
```

В следующем пакете мы вызываем эту хранимую процедуру, заполняя таблицу `Frag` (наберитесь терпения, поскольку выполнение данного пакета может занять несколько минут):

```
EXEC Add100K;
EXEC Add100K;
EXEC Add100K;
EXEC Add100K;
EXEC Add100K;
```

Выполнение команды `DBCC ShowContig (`*`таблица, индекс`*`)` приведет к созданию отчета о фрагментации и плотности данных заданной таблицы и индекса. Имея в наличии полмиллиона строк, таблица `Frag` является достаточно фрагментированной. При этом большая часть страниц заполнена только наполовину, что и следует из отчета:

```
DBCC ShowContig (frag) WITH ALL_INDEXES;
```

В следующих результатах индекс с номером 1 является первичным кластеризованным ключом, так что он также присутствует в отчете, индекс с номером 2 является некластеризованным:

```
DBCC SHOWCONTIG scanning 'Frag' table...
Table: 'Frag' (1227255527); index ID: 1, database ID: 2
TABLE level scan performed.
- Pages Scanned................................: 22056
- Extents Scanned..............................: 2772
- Extent Switches..............................: 22055
- Avg. Pages per Extent........................: 8.0
- Scan Density [Best Count:Actual Count].......: 12.50%
   [2757:22056]
- Logical Scan Fragmentation ..................: 99.24%
- Extent Scan Fragmentation ...................: 12.63%
- Avg. Bytes Free per Page.....................: 2542.0
- Avg. Page Density (full).....................: 68.59%
DBCC SHOWCONTIG scanning 'Frag' table...
Table: 'Frag' (1227255527); index ID: 2, database ID: 2
LEAF level scan performed.
- Pages Scanned................................: 2748
- Extents Scanned..............................: 348
- Extent Switches..............................: 2721
- Avg. Pages per Extent........................: 7.9
- Scan Density [Best Count:Actual Count].......: 12.64%
   [344:2722]
- Logical Scan Fragmentation ..................: 98.07%
- Extent Scan Fragmentation ...................: 99.14%
- Avg. Bytes Free per Page.....................: 3365.3
- Avg. Page Density (full).....................: 58.42%
DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```

Команда `DBCC IndexDefrag` выполняет дефрагментацию страниц как кластеризованных индексов, так и некластеризованных. Она организует узлы для повышения производи-

тельности, уменьшения размеров индекса и повышения коэффициента заполнения индексных страниц. Ее синтаксис следующий:

```
DBCC IndexDefrag (имя_базы, имя_таблицы, имя_индекса);
```

Выполнение операции DBCC IndexDefrag аналогично перестройке индекса, однако имеет то преимущество, что дефрагментация разбивается на серию небольших транзакций, что позволяет снизить вероятность блокировки пользователей, выполняющих вставку и обновление данных.

Следующие две команды выполняют дефрагментацию обоих индексов таблицы Frag:

```
DBCC IndexDefrag ('Tempdb', 'Frag', 'PK_Frag');
```

Вот результат:

```
Pages Scanned Pages Moved Pages Removed
------------- ----------- -------------
22033         15194       6852
```

Вторая команда:

```
DBCC IndexDefrag ('Tempdb', 'Frag', 'ix_col');
```

Результат таков:

```
Pages Scanned Pages Moved Pages Removed
------------- ----------- -------------
2748          1610        1134
```

Теперь с помощью команды DBCC ShowContig проверим результаты дефрагментации индекса. Как мы видим, проблемы логической фрагментации и наполнения страниц, созданные при вставке полумиллиона строк, были решены:

```
DBCC ShowContig (frag) WITH ALL_INDEXES;

DBCC SHOWCONTIG scanning 'Frag' table...
Table: 'Frag' (1227255527); index ID: 1, database ID: 2
TABLE level scan performed.
- Pages Scanned................................: 15204
- Extents Scanned..............................: 1915
- Extent Switches..............................: 1925
- Avg. Pages per Extent........................: 7.9
- Scan Density [Best Count:Actual Count].......: 98.70%
   [1901:1926]
- Logical Scan Fragmentation ..................: 0.60%
- Extent Scan Fragmentation ...................: 15.25%
- Avg. Bytes Free per Page.....................: 38.9
- Avg. Page Density (full).....................: 99.52%
DBCC SHOWCONTIG scanning 'Frag' table...
Table: 'Frag' (1227255527); index ID: 2, database ID: 2
LEAF level scan performed.
- Pages Scanned................................: 1614
- Extents Scanned..............................: 205
- Extent Switches..............................: 207
- Avg. Pages per Extent........................: 7.9
- Scan Density [Best Count:Actual Count].......: 97.12%
   [202:208]
- Logical Scan Fragmentation ..................: 1.05%
- Extent Scan Fragmentation ...................: 99.02%
- Avg. Bytes Free per Page.....................: 41.5
- Avg. Page Density (full).....................: 99.49%
DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```

На данный момент в базе данных tempdb накопилась масса объектов, которые было бы неплохо удалить с помощью следующего программного кода:

```
DROP TABLE Frag;
GO
DROP PROCEDURE Add100K;
GO
```

Статистика индексов

Полезность индекса определяется распределением данных внутри него. Например, если 60% заказчиков живут в Москве, их поиск будет быстрее выполнить путем сканирования таблицы, а не использования индекса. В то же время для поиска единственного заказчика, который живет в Урюпинске, запросу потребуется помощь индекса. Оптимизатор запросов принимает в расчет статистику индексов и на ее основе делает выводы относительно целесообразности использования индексов при выполнении конкретного запроса.

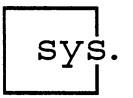
Статистика отображается в некоторых отчетах как индексы с именами, начинающимися с _WA_Sys или heed_.

Команда DBCC Show_Statistic выводит отчет с указанием даты сбора статистики. В этом отчете содержится базовая статистика индекса, включая его полезность. Низкая плотность указывает на высокую избирательность индекса; высокая демонстрирует то, что один узел индекса указывает на несколько строк, следовательно, он менее полезный.

В следующем коде продемонстрировано использование команды Update Statistics:

```
use cha2;
exec sp_help customer;
Update Statistics Customer;
```

Системные хранимые процедуры sp_createstats и sp_updatestats создают и обновляют статистику всех таблиц базы данных.



Чтобы увидеть операционную статистику индекса, выберите в динамическом представлении управления sysdm_db_index_operational_stats. Также вы можете получить статистику использования с помощью представления index_usage_stats и физическую статистику с помощью sysdm_db_index_physical_stats. Представления sysdm_db_index_operational_stats и sysdm_db_index_physical_stats требуют предоставления всех аргументов или заменяющих их значений NULL. Обязательными для ввода являются аргументы идентификаторов базы данных, объекта и индекса, равно как и номер раздела.

Плотность индекса

Плотность индекса определяется процентным отношением числа страниц индекса, содержащих данные. Если плотность индекса низкая, то для получения доступа к индексированным данным серверу придется просмотреть больше страниц. *Коэффициент заполнения* определяет процент индексных страниц, содержащих данные, после создания или дефрагментации индекса. Следует отметить, что плотность индекса немного изменяется с каждой операцией вставки, обновления или удаления.

Команда DBCC DbReIndex полностью перестраивает индекс. По существу, эта команда является аналогом последовательных операций удаления и создания индекса, за тем исключением, что она дает возможность задания коэффициента заполнения. В противоположность этому команда DBCC IndexDefrag восстанавливает исходную фрагментацию индекса, однако не доводит ее до значения заданного коэффициента заполнения.

Следующая команда воссоздает индексы таблицы Frag и устанавливает коэффициент заполнения в 98%:

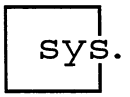
```
DBCC DBReIndex ('Tempdb.dbo.Frag','',98)
```



Плотность индекса оказывает влияние на производительность. Далее в этой главе вы найдете более подробную информацию, посвященную эффективному планированию коэффициента заполнения индекса.

Размер файла базы данных

СУБД SQL Server 7.0 оставила далеко позади используемую в предыдущих версиях методику выделения пространства в файлах с фиксированным размером, называемых *устройствами*. Начиная с версии 7.0 данные и журнал транзакций автоматически увеличиваются в размерах по мере необходимости. И все же управление размерами файлов входит в состав важных задач обслуживания базы данных. Без мониторинга и вмешательства пользователя файлы данных могут разрастись до небывалых размеров. Управление размерами файлов осуществляется с помощью рассмотренных далее команд утилиты DBCC.



Для просмотра информации о размере файла базы данных выберите в динамическом представлении управления `sysdm_db_file_space_usage`.

Мониторинг размеров файлов базы данных

Следует постоянно отслеживать три фактора, связанных с размерами файлов базы данных: текущий размер файлов и его предельно допустимая величина, количество свободных позиций внутри файлов и объем свободного места на диске.

Текущий и максимальный размеры файлов хранятся в системной таблице `sysfiles`:

```
SELECT name, size, maxsize from sysfiles;
```

Будет получен следующий результат:

```
name       size     maxsize
---------- -------- ---------
CHA2       280      -1
CHA2_log   96       268435456
```

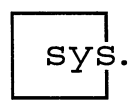
Для определения доли используемого пространства в файле можно воспользоваться системной хранимой процедурой `sp_spaceused`. Команда `DBCC Updateusage` гарантирует точность информации об использовании пространства файла:

```
DBCC Updateusage ('tempdb');
EXEC sp_spaceused;
```

Результат таков:

```
database_name      database_size    unallocated space
------------------ ---------------- ----------------
Tempdb             210.56 MB        73.44 MB

reserved    data          index_size    unused
----------- ------------ ------------- ----------------
138368 KB   122168 KB     14504 KB      1696 KB
```



Чтобы просмотреть, сколько пространства использовалось в текущей сессии, можно выбрать в динамическом представлении управления `sysdm_db_session_space_usage`.

Для определения размера файла журнала транзакций и свободного места в нем используется команда DBCC SQLPerf(LogSpace).

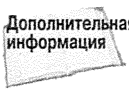
Эта команда возвращает результат следующего вида:

```
Database Name  Log Size (MB)  Log Space Used (%)     Status
-------------- -------------- ---------------------- --------
master         3.3671875      33.207657              0
tempdb         0.7421875      59.473682              0
model          0.4921875      63.194443              0
...
OOD            0.484375       72.278229              0
MS             0.7421875      37.302631              0

DBCC execution completed.
  If DBCC printed error messages,
  contact your system administrator.
```

Для мониторинга свободного пространства в дисковой подсистеме сервера используется процедура xp_fixeddrives, возвращающая такой результат:

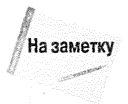
```
drive MB Free
----- -------
C     429
F     60358
```



О процессе конфигурирования файлов данных и журнала транзакций см. в главе 17.

Сжатие базы данных

Если база данных не сконфигурирована для автоматического сжатия в фоновом режиме, то свободное пространство, освобождаемое при удалении неиспользуемых объектов и строк, не возвращается операционной системе. Файлы в данном случае сохраняют максимальный размер, которого достигли данные. Если данные постоянно добавляются и удаляются, регулярное сжатие и наращивание размера файла будет пустой тратой времени. Однако если дисковое пространство представляет собой особую ценность, то после удаления информации нужно выполнять ручное сжатие файла данных. Сжатие может выполняться параллельно с любыми транзакциями.



Использование автоматического сжатия в течение дня, когда пользователи активно работают с данными, может негативно сказаться на общей производительности. Как правило, автоматическое сжатие выполняют в нерабочее время.

Использование команды DBCC ShrinkDatabase позволяет уменьшить размеры файлов базы данных. При этом выполняются две основные задачи.

1. Данные упаковываются в начале файла, а пространство в конце файла остается свободным.
2. Свободное пространство в конце файла обрезается, за счет чего уменьшается размер файла.

Управление этими двумя действиями осуществляется с помощью параметров, описанных ниже.

- Параметр notruncate приводит к выполнению командой DBCC ShrinkDatabase только первого действия — выполняется упаковка данных, но размер файла не изменяется.
- Параметр truncateonly приводит к обрезанию свободного пространства в конце файла, предварительно не выполняя упаковку данных.

- Целевой размер файла может быть задан с помощью указания процента свободного пространства после сжатия файла. Так как автоматическое увеличение размеров является достаточно затратной операцией, полезно оставлять некоторый объем свободного пространства. Если необходимый размер превышает доступное пространство, то размеры файла не изменяются.

В следующем примере выполняется сжатие базы данных OBXKites с 10%-ным запасом свободного пространства:

```
DBCC ShrinkDatabase ('OBXKites', 10);
```

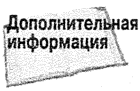
Будет получен следующий результат:

```
DBCC SHRINKDATABASE: File ID 3 of database ID 12 was skipped
because the file does not have enough free space to reclaim.
DbId FileId  CurrentSize MinimumSize UsedPages   EstimatedPages
---- ------- ----------- ----------- ----------- --------------
12   1       216         152         184         184

(1 row(s) affected)

DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```

Как видно из примера, не все файлы имеют достаточно места для перераспределения. Данная команда отображает старые и новые размеры файлов, давая понять, какие изменения произошли в базе данных. Команда `DBCC ShrinkDatabase` влияет на все файлы базы данных, в то время как команда `DBCC ShrinkFile` оказывает влияние только на файл, указанный в аргументах.



База данных может быть сконфигурирована для автоматического сжатия файлов. Подробно об этом процессе настройки см. в главе 34.

Сжатие журнала транзакций

При сжатии базы данных вместе с ней сжимается и журнал транзакций. Использование параметров `notruncate` и `truncateonly` никак не отражается на сжатии журнала транзакций. Если существует несколько файлов журнала транзакций, то SQL Server сжимает их так, будто они представляют собой единый последовательный файл.

Наиболее распространенной проблемой, связанной с журналом транзакций, является постоянное увеличение размера файла. Причиной такого поворота событий может стать старая открытая транзакция. Журнал транзакций состоит из виртуальных разделов. Успех или неудача выполнения операции сжатия журнала зависит от срока действия транзакций в виртуальных разделах и установленных контрольных точек. SQL Server может сжимать журнал транзакций путем удаления данных, которые старше самой давней транзакции в структуре виртуальных журналов.

Для проверки факта существования старых транзакций в журнале используется команда `DBCC OpenTran`:

```
USE OBXKites;
BEGIN TRAN;
  UPDATE Product
    SET ProductDescription = 'OpenTran'
    WHERE Code = '1002';
DBCC OpenTran ('OBXKites');
```

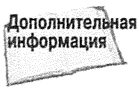
Будет получен следующий результат:

```
Transaction information for database 'OBXKites'.

Oldest active transaction:
  SPID (server process ID): 57
```

```
  UID (user ID) : -1
  Name : user_transaction
  LSN : (19:524:2)
  Start time : Mar 8 2006 4:20:38:890PM
  SID :
  0x0105000000000000515000000b4b7cd2267fd7c3043170a32eb030000
DBCC execution completed. If DBCC printed error messages,
  contact your system administrator.
```

На основании этой информации можно отыскать затерявшиеся транзакции, а затем прервать подключение пользователя. Узел `Current Activity` утилиты Management Studio может предоставить больше информации о подключении с заданным идентификатором. Существует еще одно, более категоричное решение проблемы — можно остановить сервер и перезапустить его, после чего выполнить сжатие базы данных.



На разрастание журнала транзакций оказывают влияние выбранная модель восстановления и порядок резервирования журнала. Более подробную информацию об этих критичных вопросах см. в главе 36.

Когда вы найдете открытую транзакцию, закройте ее. В следующем примере подтверждается транзакция и проверяется, существуют ли в базе данных другие открытые транзакции:

```
COMMIT TRAN;
DBCC OpenTran ('OBXKites');
```

Будет получен следующий результат:

```
No active open transactions.
DBCC execution completed. If DBCC printed error messages,
  contact your system administrator.
```

Дополнительные команды утилиты DBCC

Оставшиеся семь команд утилиты DBCC используются для разрешения проблем, связанных с тестированием хранимых процедур и триггеров.

- `DBCC DropCleanBuffers`. Очистка памяти, занятой буферизированными данными, чтобы они не влияли на производительность запросов во время тестирования.
- `DBCC Inputbuffer (`*`SPID`*`)`. Возвращает последнюю инструкцию, выполненную клиентом с заданным идентификатором SPID. По очевидным причинам эта команда может быть выполнена только членами серверной роли `sysadmin`.
- `DBCC Outputbuffer (`*`SPID`*`)`. Возвращает результат последней инструкции, выполненной клиентом с заданным идентификатором. Подобно `DBCC Inputbuffer`, эта команда может быть выполнена исключительно членами серверной роли `sysadmin`.
- `DBCC PinTable (`*`идентификатор_базы, идентификатор_объекта`*`)`. Закрепляет таблицу в памяти. Старайтесь избегать закрепления таблиц в полном объеме. Гораздо эффективнее по мере необходимости выгружать в кэш отдельные страницы.
- `DBCC UnPinTable (`*`идентификатор_базы, идентификатор_объекта`*`)`. Удаляет таблицу из списка закрепленных в памяти.
- `DBCC ProcCache`. Выводит основную статистику процедурного кэша после того, как запросы и процедуры скомпилированы и сохранены в памяти.
- `DBCC CurrencyViolation`. Данная команда проверяет, сколько раз достигался предел в редакциях с ограниченным числом подключений.

Управление обслуживанием базы данных

SQL Server является центральным узлом команд обслуживания базы данных. СУБД обеспечила администратора баз данных возможностью составления плана обслуживания сервера.

Планирование обслуживания базы данных

Идеальный план обслуживания базы данных включает в себя несколько задач (в указанном порядке).

- **Проверка согласованности**. Следует выполнить команды `DBCC CheckDB` и `DBCC CleanTable` для таблиц, которые подверглись большим текстовым обновлениям. Затем следует проверить целостность структуры базы данных, выполнив команды `DBCC CheckCatalog`, `DBCC CheckConstraints` и `DBCC CheckIdent`.
- **Обновление статистики индексов**.
- **Дефрагментация базы данных**.
- **Перестройка индексов**.
- **Резервное копирование**. Стратегический план резервирования базы данных включает в себя комбинацию полного, дифференциального и транзакционного копирования. При этом в резервировании должны участвовать как системные, так и все существенные пользовательские базы данных.
- **Проверка размеров файлов и высвобождение дискового пространства**.

Все эти задачи можно автоматизировать с помощью заданий агента SQL Server Agent.



С выходом пакета обновлений SP1 в работе планов обслуживания произошли некоторые изменения.

- План обслуживания работает, когда установлен компонент SQL Server Tools или Integration Services. Раньше для этого требовалось установить оба этих компонента.
- В плане обслуживания можно использовать задание очистки для удаления файлов резервных копий из подкаталогов.

Выход пакета SP2 также внес некоторые изменения в планы обслуживания.

- Стала возможной поддержка множества расписаний. Конструктор планов обслуживания поддерживает множество подчиненных планов одного главного, при этом каждый подчиненный план может иметь собственное расписание для своих заданий.
- Планы обслуживания могут работать на множестве серверов, используя главный и целевой серверы; при этом после запуска мастера плана обслуживания требуется некоторое конфигурирование вручную.
- Мастер плана обслуживания включает крайне востребованное задание `Maitenance Cleanup`, существовавшее в SQL Server 2000. Это задание удаляет файлы, оставшиеся от выполнения плана обслуживания.
- Задание `Database Backup` включает возможность определения параметра срока жизни резервной копии, существовавшего в версии SQL Server 2000.
- Задание `Database Backup` не изменяет папку хранения резервных копий, если она была определена в месте, отличном от заданных по умолчанию.

- Задание `Database Backup` не дает возможности ошибочно задать для системных баз данных дифференцированное или транзакционное резервное копирование.
- Задание `History Cleanup` содержит возможность определения интервала срока жизни файлов в часах.
- Задание `Update Statistics` содержит параметры полного сканирования и размера выборки, которые были доступны в версии SQL Server 2000.

Мастер плана обслуживания

Мастер плана обслуживания, встроенный в утилиту Management Studio, поможет автоматизировать базовый план обслуживания базы данных и выполнить все связанные с ним задачи. Все планы обслуживания отображаются в папке `Management\Maintenance Plans` утилиты. Щелкните правой кнопкой мыши на этой папке и выберите в контекстном меню пункт New Maintenance Plan. В открывшемся диалоговом окне введите имя плана обслуживания и щелкните на кнопке OK.

После присвоения имени плану обслуживания Management Studio откроет окно, включающее имя, описание и время запуска плана, а также список выполняемых задач. На рис. 37.1 показан пример плана обслуживания с несколькими введенными задачами.

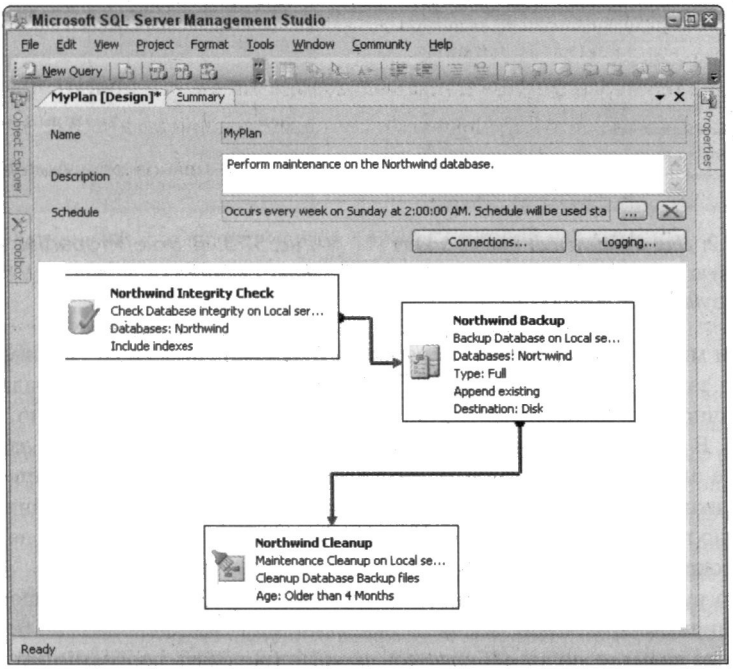


Рис. 37.1. Создание плана обслуживания с помощью ввода нескольких задач и времени их запуска

Добавление задачи

План обслуживания должен содержать минимум одну задачу. К счастью, добавить задачу не так уж и сложно. Для этого достаточно перетащить с панели задач (рис. 37.2) нужный зна-

чок в рабочую область плана обслуживания. После этого задание примет форму прямоугольной области с описанием (см. рис. 37.1).

В данный момент задача еще не сконфигурирована — конфигурация определяется в диалоговом окне Properties (рис. 37.3), содержимое которого зависит от типа конкретной задачи. Например, если вы выберете задачу резервирования базы данных, то увидите в окне параметры выбора типа базы данных, типа устройства резервирования и прочую информацию, связанную с этим типом. С помощью раскрывающегося списка вы можете выбрать другую задачу. В центре окна Properties содержится список параметров выбранной задачи — их можно организовать по категориям или в алфавитном порядке. В нижней части окна Properties отображается описание выделенного свойства. Это описание обычно обеспечивает пользователя достаточной информацией для корректного конфигурирования выбранной задачи.

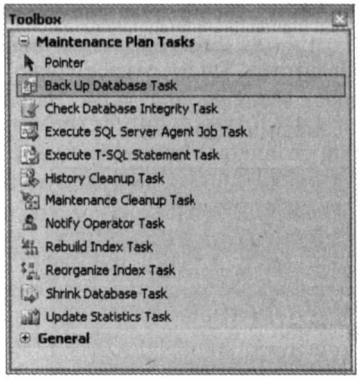


Рис. 37.2. Для добавления новых задач в рабочую область плана выполнения воспользуйтесь панелью задач

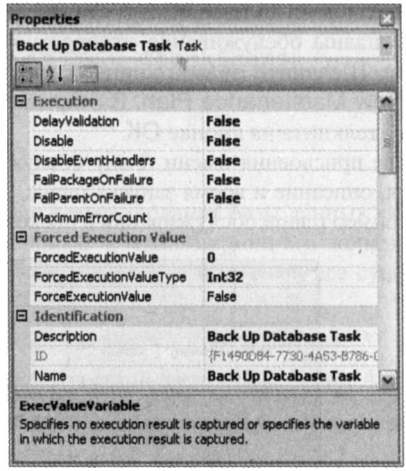


Рис. 37.3. В окне Properties определите специфику задачи

Иногда вам может не потребоваться работать с окном параметров. Например, вы можете создать новую задачу и пропустить окно Properties, дважды щелкнув на задаче. Откроется окно, специфичное для конкретной задачи (на рис. 37.4 показано такое окно для задачи резервирования). В отличие от окна Properties в нем отображаются свойства задачи в контексте. К тому же элементы в нем отображаются серым цветом, пока не будет определена их корректная функциональность. Например, при выборе баз данных для резервирования данное диалоговое окно не даст вам установить какие-либо другие параметры, пока не будет выделена одна или несколько баз.

Достаточно интересно посмотреть на программный код T-SQL, сформированный мастером после установки параметров задачи. В окне Transact SQL невозможно отредактировать команды, в то же время, если вы собираетесь сконфигурировать программный код SQL вручную, то можете предварительно взглянуть на то, как с этой задачей справляется мастер плана обслуживания.

Вы поймете, когда задача сформирована, по внешнему виду прямоугольника в рабочей области конструктора. Задачи, которые не могут быть выполнены корректно, отображаются с крестиком в красном кружке.

После того как определена одна задача, приступайте к формированию следующей. Вы можете создавать задачи в любой последовательности, связывая их в цепочку впоследствии

в рабочей области конструктора. На рис. 37.1 план обслуживания содержал несколько связанных задач. Этот план предполагал проверку целостности базы данных `Northwind`, последующее ее резервирование и обслуживание файлов.

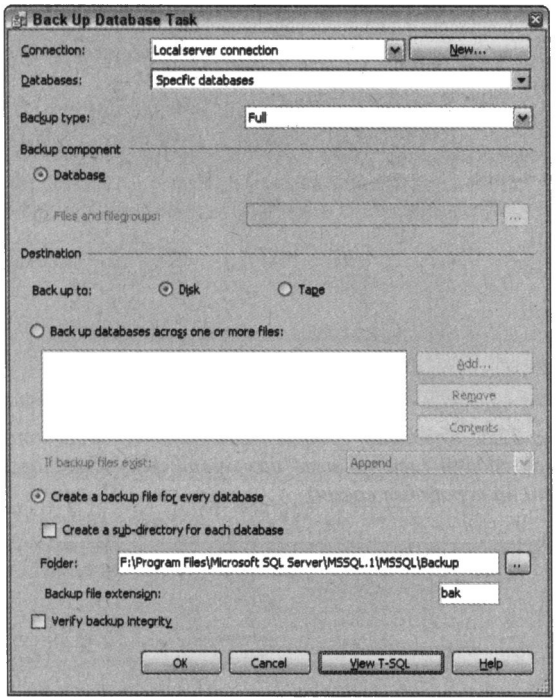


Рис. 37.4. В дополнение к окну **Properties** *для конфигурирования вы можете воспользоваться диалоговым окном, специфичным для конкретной задачи*

Определение графика выполнения

Созданные задачи вы можете запустить на выполнение вручную; также можно потребовать от системы их автоматического запуска по расписанию. Как правило, такие регулярные задачи, как резервирование, лучше автоматизировать. Щелкните на эллипсе рядом с полем Schedule и в открывшемся диалоговом окне (рис. 37.5) создайте график выполнения задачи.

Создайте график задания — например, запланируйте ежедневное, еженедельное и ежемесячное его выполнение. В этом диалоговом окне вы можете также выбрать конкретное время запуска задания и задать интервал, в течение которого задание будет актуальным. Единственное, чего нельзя сделать в этом диалоговом окне, — это запланировать запуск задания вручную. Для настройки запуска задачи вручную щелкните на красном крестике рядом с полем Schedule.

Создание нового подключения

В зависимости от настройки базы данных для выполнения некоторых задач вам может потребоваться создать несколько подключений. Мастер плана обслуживания всегда предлагает подключение к экземпляру по умолчанию, установленному на локальном компьютере. Чтобы добавить другие подключения в текущий план обслуживания (имейте в виду, что данные подключения не распространяются на другие планы), щелкните на кнопке Connections. Откроется диалоговое окно, показанное на рис. 37.6.

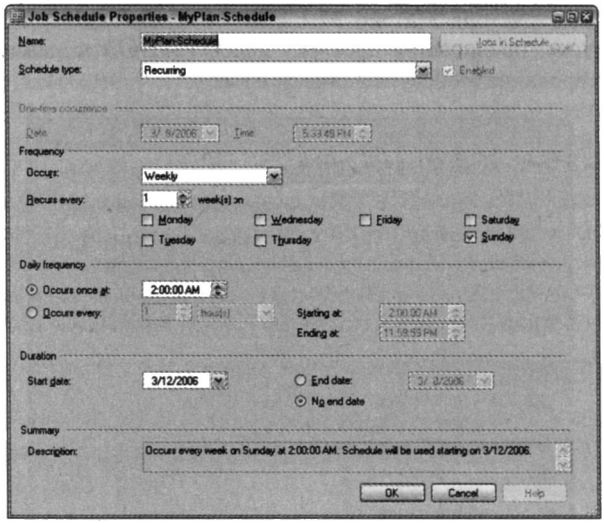


Рис. 37.5. Создайте график, соответствующий характеру задачи (старайтесь назначать ресурсоемкие задачи на нерабочее время)



Рис. 37.6. Если задаче требуются дополнительные подключения, определите их

Щелкните на кнопке Add и добавьте новое подключение. Определение подключения состоит из трех элементов: имени базы данных, имени сервера и регистрационных данных системы безопасности сервера. Старайтесь в имени подключения отражать имя сервера и его экземпляра. Если в будущем вам потребуется устранять какие-либо проблемы, то такое имя, как `Мое подключение`, вряд ли вам поможет быстро выполнить обслуживание базы. Имя сервера и его экземпляра можно ввести вручную или выбрать из списка — в последнем случае нужно щелкнуть на эллипсе рядом с полем Enter or Select Server Name. И в заключение вам потребуется выбрать между встроенной аутентификацией SQL Server и использованием учетной записи Windows. Для того чтобы добавить созданное подключение в список, щелкните на кнопке OK.

Диалоговое окно Manage Connections позволяет редактировать и удалять ранее созданные подключения. При редактировании существующего подключения вы можете изменить имя сервера, а также регистрационные данные; остальные поля в диалоговом окне будут недоступны. Чтобы удалить из списка подключение, потребность в котором отпала, щелкните на кнопке Remove.



Помните, что удаление подключения — это дорога с односторонним движением, к тому же эта операция выполняется довольно быстро. Перед щелчком на кнопке Remove дважды проверьте, правильное ли подключение выделено в списке.

Протоколирование задач обслуживания

Многие из задач обслуживания, автоматизируемые с помощью мастера, будут выполняться в нерабочее время, когда вы вряд ли захотите сидеть за монитором. Но вы можете потребовать от задач обслуживания протоколирования своих действий, чтобы иметь возможность просмотра журнала обслуживания в любое удобное время. Для использования этой функции щелкните на кнопке Logging. Откроется диалоговое окно, показанное на рис. 37.7.

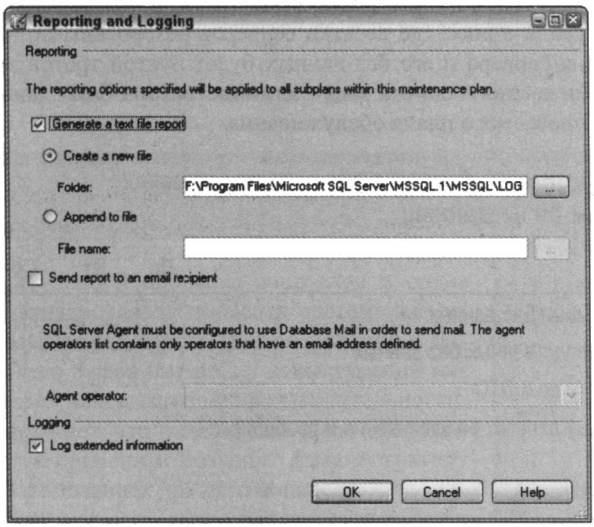


Рис. 37.7. При желании вы можете заставить сервер протоколировать ход выполнения задач обслуживания базы данных

В этом диалоговом окне вы можете потребовать создания файла журнала, а также отправку протокола по электронной почте. Последний вариант работает достаточно хорошо и позволяет сэкономить время на поиск файла на диске. В то же время текстовый файл позволяет накапливать архив, что сложно обеспечить при регистрации хода выполнения задач в электронных сообщениях. После конфигурирования параметров отчетности щелкните на кнопке OK, и план обслуживания начнет регистрировать свои действия.



При желании вы можете потребовать от системы предоставления расширенной информации о действиях, выполняемых в ваше отсутствие. Учтите, что даже небольшой фрагмент информации поможет вам понять, в каком месте конкретная задача обслуживания пошла вкривь и вкось.

Обслуживание базы данных из командной строки

Как правило, обслуживание базы данных выполняется с помощью утилиты Management Studio или автоматизируется с помощью агента SQL Server Agent. В то же время обслуживать

базу данных можно и из командной строки DOS с помощью утилиты SQLMaint. Эта утилита имеет множество параметров, которые позволяют выполнять резервирование, обновлять статистику и запускать команды утилиты DBCC. Более подробную информацию об использовании этой утилиты вы можете получить в SQL Server 2005 Books Online.

Утилита SQLMaint может оказаться полезной в некоторых ситуациях, когда в план обслуживания будут включены некоторые системные функции или сторонние утилиты резервного копирования. Но я все же рекомендую оставаться с агентом SQL Server Agent, разумеется, если не возникают обстоятельства непреодолимой силы, не допускающие его использование.

Мониторинг обслуживания базы данных

Совершенно не достаточно лишь запланировать выполнение задач обслуживания базы данных. В крупных реализациях, где десятки серверов разбросаны по всему миру, обычный мониторинг состояния сервера и его баз данных будет пустой тратой времени. В табл. 37.1 приведен пример стандартного списка задач администратора базы данных, который можно принять за основу создаваемого плана обслуживания.

Таблица 37.1. Ежедневный план работ, выполняемых администратором базы данных

Задача	Пн	Вт	Ср	Чт	Пт	Сб
Резервирование системных баз данных						
Резервирование производственных баз данных						
Запуск SQL Agent, SQL Main и DTC						
Проверка размеров базы данных, ее увеличения и свободного пространства на диске						
Выполнение пакетных заданий						
Выполнение заданий DBCC						
Проверка ошибок в журнале сервера						
Запуск агента журнала репликаций						
Выполнение заданий очистки уже распределенных репликаций						
Проверка последней перезагрузки сервера баз данных						

В зависимости от сложности и количества серверов администратор может создать данный список задач вручную на рабочем листе Excel или в таблице SQL Server.

Резюме

В этой главе мы остановились на вопросах обслуживания базы данных. SQL Server предлагает богатый выбор команд и утилит, которые можно использовать для мониторинга состояния сервера баз данных и выполнения его обслуживания. Планом обслуживания следует охватить все инсталляции SQL Server. Это поможет администратору справиться со своей работой, не пропустив важных элементов обслуживания сервера.

В следующей главе мы поговорим об использовании агента SQL Server Agent, который позволяет автоматизировать выполнение заданий.

Автоматизация обслуживания баз данных с помощью SQL Server Agent

ГЛАВА

38

В этой главе...

Настройка SQL Server Agent

Концепции предупреждений, операторов и заданий

Управление предупреждениями

Управление операторами

Управление заданиями

Автоматизация обслуживания базы данных важна для обеспечения регулярной проверки состояния и оптимизации базы. Автоматизированная проверка состоит из мониторинга размера базы, ее регулярных резервирований и оптимизации, включающего в себя настройку конфигурации индексов для обеспечения максимальной производительности. Автоматизация предполагает, что выполнение задач обслуживания не отберет много времени и позволит администратору сосредоточиться на более важных для него вопросах.

В идеальном случае вам достаточно запланировать регулярные самопроверки сервера баз данных. В случае обнаружения критических ошибок сервер предупредит вас. В версии SQL Server 2005 содержится специальный компонент, способный генерировать предупреждения при возникновении опасных ситуаций. Этот же компонент позволяет запланировать регулярное выполнение задач обслуживания базы данных — например, каждую неделю или в первую субботу месяца. SQL Server Agent — это служба, ответственная за обработку предупреждений и запуск запланированных заданий.

Настройка SQL Server Agent

Настройку SQL Server Agent выполнить достаточно просто, если миновать два подводных камня, один из которых обойти легко, а другой — немного сложнее. Разумеется, мы начнем с более простого. Так как SQL Server Agent является обычной службой Windows, следует обеспечить его автоматический запуск при каждой перезагрузке операционной системы. (Компания Microsoft не обеспечила настройку автомати-

ческого запуска этой службы при инсталляции SQL Server, так что эту настройку придется выполнить вручную.) Эта операция проста, однако ее часто обходят вниманием. (Результатом может стать то, что после нескольких перезагрузок сервера ни одно из запланированных технологических заданий так и не будет запущено и, что еще хуже, не будут сгенерированы некоторые критичные для системы предупреждения.)



Чтобы избежать проблем, связанных со службами, следует сконфигурировать их для автоматического запуска. Откройте консоль служб в папке Администрирование (Administrative Tools) панели управления системы Windows. Щелкните правой кнопкой мыши на службе `SQL Sever Agent` и выберите в контекстном меню пункт Properties. В раскрывающемся списке Startup Type выберите пункт Automatic и щелкните на кнопке OK.

Служба SQL Server Agent маркируется с указанием в скобках имени экземпляра. Если на компьютере установлено несколько экземпляров SQL Server, то отдельная служба SQL Server Agent будет обслуживать конкретный экземпляр.

Как и в любой другой службе, режим запуска SQL Server Agent можно изменить в любой момент в консоли служб. В то же время эту задачу легче выполнить с помощью утилиты SQL Server Configuratyion Manager, запускаемой с помощью выбора в меню Пуск пункта Все программы⇨Microsoft SQL Server 2005⇨Configuration Tools. Окно этой утилиты показано на рис. 38.1.

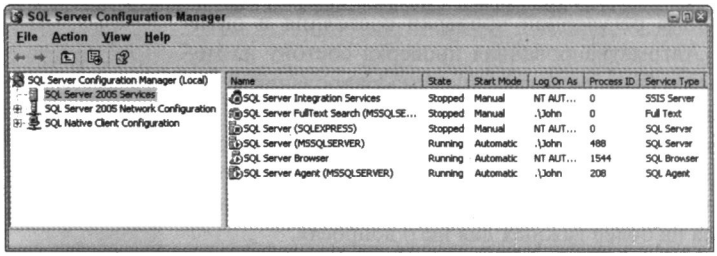


Рис. 38.1. Окно утилиты SQL Server Configuration Manager позволяет легко изменить режим запуска агента SQL Server Agent

Чтобы гарантировать автоматический запуск службы SQL Server Agent, выполните следующие действия.

1. Откройте утилиту SQL Server Configuration Manager.
2. Выделите папку `SQL Server 2005 Services`.
3. Щелкните правой кнопкой мыши на службе, режим которой хотите изменить, и выберите в контекстном меню пункт Properties. Откроется диалоговое окно свойств службы.
4. Перейдите на вкладку Service.
5. Выделите свойство Start Mode и выберите в раскрывающемся списке новый режим запуска.
6. Щелкните на кнопке OK, чтобы изменения вступили в силу.

Чтобы убедиться в запуске службы SQL Server Agent, было бы неплохо выполнить еще одно дополнительное действие. Вот как это сделать.

1. Откройте окно SQL Server Management Studio, выбрав в меню Пуск пункт Все программы⇨Microsoft SQL Server 2005⇨Management Studio.

2. Раскрывайте папки до тех пор, пока не найдете сервер, подлежащий конфигурированию. Если в настоящий момент вы работаете на конфигурируемом сервере, то путь к папке будет следующим: `Console Root/Microsoft SQL Server/SQL Server Group/(local) (Windows NT)`.
3. Разверните папки под выбранным сервером. Одна из них будет носить имя `SQL Server Agent`. Щелкните правой кнопкой мыши на этой папке и выберите в контекстном меню пункт **Properties**. Перейдите на вкладку **General**, в которой вам будут предложены параметры, показанные на рис. 38.2.

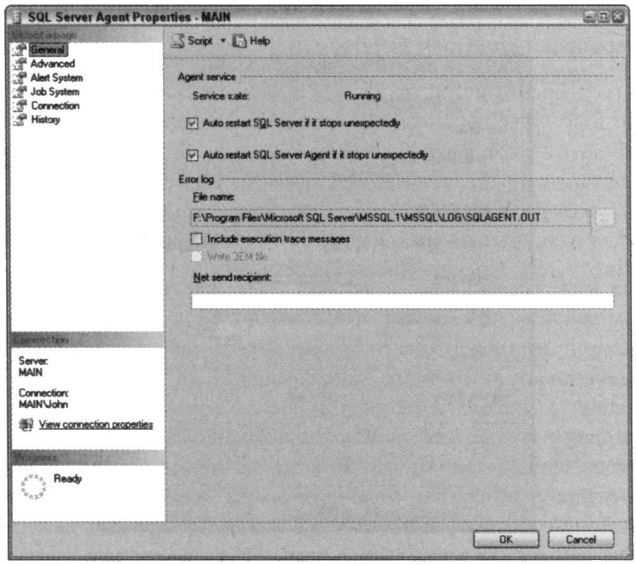


*Рис. 38.2. Вкладка **General** диалогового окна свойств службы SQL Server Agent позволяет сконфигурировать способ запуска службы*

4. Установите оба флажка, начинающиеся словами `Auto restart`. Установка этих флажков гарантирует автоматический запуск SQL Server и SQL Server Agent в случае их неожиданной остановки.

Второй подводный камень скрывается в настройке учетной записи, под которой будет работать служба SQL Server Agent. По умолчанию эта служба запускается в контексте защиты системной учетной записи. Эта запись имеет доступ только к локальным ресурсам. Если вам нужно в запланированных заданиях обращаться к ресурсам локальной сети, используйте учетную запись домена. Например, резервирование базы данных вы можете осуществлять на другом сервере. Также следует иметь учетную запись домена, если планируется отправлять уведомления оператору по электронной почте или на пейджер (об этом мы поговорим немного позже в этой главе). Доменная учетная запись также требуется для обеспечения работоспособности репликации. Как правило, в этих случаях SQL Server Agent настраивают для использования доменной учетной записи Windows, являющейся членом роли `sysadmin`. Только в этом случае у службы будет достаточно прав для выполнения заданий и отправки уведомлений.

Для изменения учетной записи, используемой службой SQL Server Agent, выполните следующие действия.

1. Откройте SQL Server Configuration Manager.
2. Выделите папку `SQL Server 2005 Services`.
3. Щелкните правой кнопкой мыши на записи `SQL Server Agent` и выберите в контекстном меню пункт Properties. Откроется диалоговое окно параметров службы.
4. Перейдите на вкладку Log on (рис. 38.3). В этой вкладке вы можете выбрать одну из встроенных учетных записей или создать новую. Поскольку я использую систему разработки, то пользуюсь специальной учетной записью. Обычно в производственных системах приходится выбирать между записями `Local System`, `Local Service` и `Network Service`.

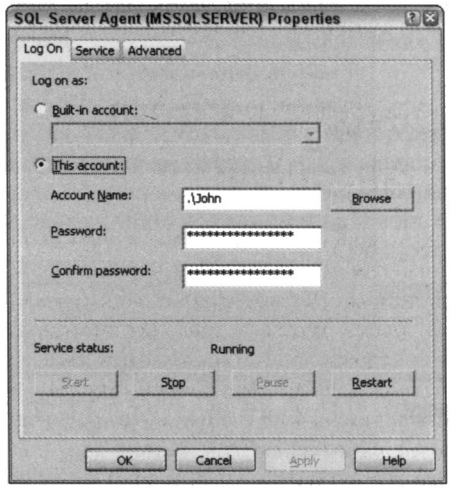


Рис. 38.3. Выберите для службы SQL Server Agent встроенную или другую учетную запись

5. Щелкните на кнопке OK. В диалоговом окне будет отображено предупреждение о том, что для вступления в силу изменений следует перезапустить службу SQL Server Agent.
6. Щелкните на кнопке Yes. Сразу после перезапуска службы изменения вступят в силу.

Заключительным шагом настройки службы SQL Server Agent является создание профиля учетной записи электронной почты для возможности отправки уведомлений на электронный адрес или пейджер. Это требует установки и конфигурирования почтовой службы и информирования SQL Server Agent о способе доступа к этой службе. Проще всего организовать доставку почтовых сообщений с помощью Exchange Server. Если в качестве почтовой службы вы используете Microsoft Exchange, выполните следующие действия.

1. Установите почтовый ящик службы SQL Server Agent на сервере Exchange (обычно этот сервер устанавливают на компьютере, отличном от сервера баз данных). Сконфигурируйте этот почтовый ящик для использования учетной записи домена, под которой запускается SQL Server Agent. Не забудьте сопроводить создаваемый профиль описательным именем — это поможет в дальнейшем избежать случайного удаления этого важного профиля.
2. Установите MAPI-совместимый клиент электронной почты на сервере баз данных (например, можете использовать программу Outlook).

3. Настройте почтовый профиль службы SQL Server Agent с помощью утилиты Mail панели управления. Этот почтовый профиль должен указывать на сервер Exchange и созданный на первом этапе процедуры почтовый ящик.

Теперь нам осталось только указать службе SQL Server Agent, какой ящик использовать при отправке электронных сообщений. Эта операция выполняется в диалоговом окне параметров службы.

1. Щелкните на вкладке Alert System.
2. В группе Mail Session установите флажок Enable Mail Profile. После этого станут доступными все остальные параметры группы.
3. Выберите профиль электронной почты, который настроили для службы. Чтобы проверить корректность настройки профиля, щелкните на кнопке Test.

Концепции предупреждений, операторов и заданий

Предупреждение определяет конкретное действие, совершаемое при выполнении определенного условия или по достижении некоторого состояния. Такое условие может быть настроено с помощью множества системных счетчиков производительности, в частности, достижения базой данных определенного объема, максимального количества подключений или количества взаимоблокировок в секунду. Условие также может быть связано с числом или строгостью ошибок. Когда некоторое заданное условие выполняется, SQL Server Agent может уведомить об этом одного или нескольких операторов и/или запустить на выполнение какое-либо задание.

Операторами называют людей, которые отвечают за обработку критических ситуаций на сервере баз данных. Как уже говорилось в предыдущем разделе, одной из основных задач SQL Server Agent является отправка сообщений операторам для их уведомлении об условиях работы сервера. При этом операторы могут получать сообщения по электронной почте, на пейджер или с помощью службы сообщений Net Send. Вы можете определить время доступности определенных операторов по пейджеру (например, с 9:00 до 17:00). Также вы можете приостановить отправку сообщений определенному оператору, если он ушел в отпуск.

Заданием называется одна операция или группа операций, выполняемых в базе данных. В качестве примеров заданий можно привести резервирование базы данных, реорганизацию индексов и выполнение пакетов службы преобразования данных (DTS). Задания SQL Server Agent также используются и в фоновом режиме для реализации планов обслуживания, созданных в SQL Server Management Studio.

Управление операторами

Подобно учетным записям, которые нужны для пользователей, осуществляющих доступ к базе данных SQL Server, операторы создают, чтобы иметь возможность отправлять им уведомления. Для создания оператора в SQL Server выполните следующие действия.

1. Запустите SQL Server Management Studio и найдите папку `SQL Server Agent` конфигурируемого сервера.
2. Щелкните правой кнопкой мыши на папке `SQL Server Agent\Operators` и выберите в контекстном меню пункт New Operator. В открывшемся диалоговом окне (рис. 38.4) перейдите к вкладке General.

3. Введите имя оператора, его адрес электронной почты, пейджера или службы Net Send (в зависимости от выбранного способа отправки уведомлений). Если выбирается адрес пейджера, следует дополнительно определить время доступности соответствующего оператора. Если вводимый адрес электронной почты допускает двусмысленное толкование, заключите его в квадратные скобки. В качестве альтернативы можете щелкнуть на кнопке с многоточием и выбрать адрес из адресной книги сервера базы данных.
4. Выделите папку `Notifications` и выберите метод доставки каждого из типов уведомлений.
5. Щелкните на папке `Job` и выберите метод оповещения для каждого задания.

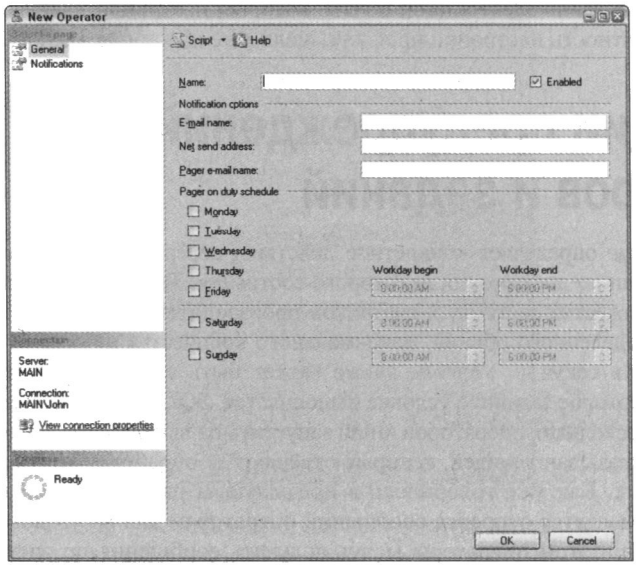


Рис. 38.4. В диалоговом окне New Operator *можно определить, когда оператор доступен для приема уведомлений на пейджер*



При каждом создании нового задания или уведомления вы должны снова заходить на страницу конфигурирования операторов. Убедитесь, что каждому нужному заданию и уведомлению назначены соответствующие операторы.

Если некоторый оператор не доступен для обработки уведомления, вы можете временно отключить его, сняв соответствующий флажок Enabled во вкладке General. Вместо отключения оператора вы можете удалить его адреса доставки уведомлений и заполнить их снова, когда оператор приступит к работе.

Управление предупреждениями

В зависимости от того, как была выполнена настройка SQL Server, вы можете увидеть некоторые из доступных предупреждений. При установке дополнительных средств, таких как репликации, SQL Server добавляет новые предупреждения в список доступных. Например, имена доступных предупреждений репликации начинаются со слова `Replication`. На

рис. 38.5 показан список предупреждений, связанных с репликациями. По умолчанию все эти уведомления отключены, за исключением предупреждений об ошибках, а также предупреждений, которым не назначены операторы. Если вы решили использовать репликацию, то должны назначить операторов, получающих уведомления о недопустимых процессах в сервере. В предыдущем разделе был описан один из методов назначения операторов уведомлениям. В следующих подразделах речь пойдет о том, как создать собственные определения ошибок и сформировать для них предупреждения.

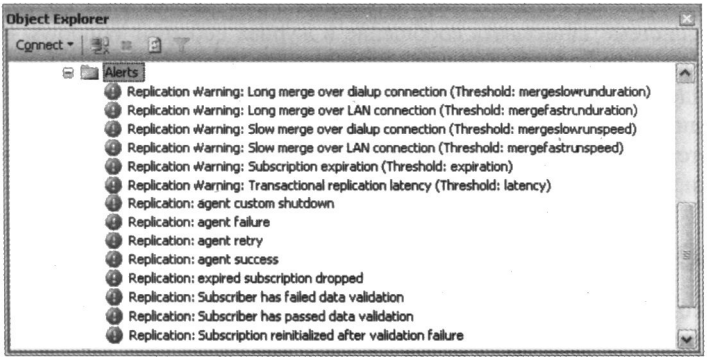
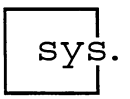


Рис. 38.5. SQL Server при установке поддержки репликации предлагает по умолчанию некоторые предупреждения

Создание сообщений об ошибках, определенных пользователем

Если вы внедряете пользовательские приложения, использующие SQL Server в качестве хранилища данных, то можете определить собственный набор ошибок. Для выполнения этой задачи используется команда `sp_addmessage`. Вы должны передать этой команде номер ошибки (в диапазоне от 50 001 до 2 147 483 647), ее уровень строгости (число от 1 до 25), а также текст сообщения. Эта команда также принимает в качестве аргумента идентификатор языка, установленного на сервере (например, `English`). По умолчанию используется язык, предопределенный в региональных настройках операционной системы. Также следует определить, нужно ли записывать данное сообщение об ошибке в журнал событий Windows (по умолчанию эта запись отключена). В заключение следует переопределить отображаемое сообщение. Если вы попытаетесь создать сообщение с уже существующим номером и при этом не укажете ключевое слово `REPLACE`, то система выдаст сообщение об ошибке с номером 15043, информирующее о необходимости использования этого ключевого слова при замене существующего сообщения новым.



Для получения списка ошибок, поддерживаемых SQL Server, создайте запрос к представлению каталога `sysmessages`. Если вы хотите просмотреть только пользовательские сообщения, то установите фильтр `error>50000`. Вы можете также отфильтровать сообщения по строгости и идентификатору языка. Если сообщение об ошибке записывается в журнал событий Windows, то его поле `dlevel` имеет значение `128`.

Приведем пример использования команды `sp_addmessage`:

```
sp_addmessage 50001, 1, 'Тестовое сообщение.';
```

В данном случае мы добавили новое сообщение с номером ошибки 50001. Строгость этой ошибки равна единице, и отображается текст "Тестовое сообщение". Система не зарегистрирует это сообщение в журнале событий Windows и будет использовать язык, установленный по умолчанию. Теперь предположим, что вы решили записать сообщение в журнал событий Windows. В этом случае команда примет следующий вид:

```
sp_addmessage 50001, 1, 'Тестовое сообщение.', 'Russian', True,
REPLACE;
```

Обратите внимание на то, что теперь в команде явно указан язык сообщения. Значение `True` указывает системе записывать сообщение в журнал событий Windows. К тому же в команду было включено слово `REPLACE`, так как данное сообщение замещает уже существующее.

Когда речь идет о предупреждениях, пользовательские и системные сообщения SQL Server обрабатывает одинаково. Определите номер ошибки и ее строгость, и SQL Server Agent инициирует соответствующую реакцию сервера. В следующем разделе будут рассмотрены вопросы настройки этого типа предупреждений.

Создание предупреждения

Вы можете создать два типа предупреждений. Первый тип инициируется номером ошибки или при достижении ею определенной строгости; второй — счетчиком производительности SQL Server. Вот как настроить оба этих типа предупреждений.

1. Запустите SQL Server Management Studio.
2. Щелкните правой кнопкой мыши на папке `SQL Server Agent\Alerts` и выберите в контекстном меню пункт `New Alert`. Откроется диалоговое окно создания нового предупреждения, показанное на рис. 38.6.
3. Введите имя предупреждения. Поскольку предупреждение может касаться одной или нескольких баз данных и определяться любым количеством условий или состояний,

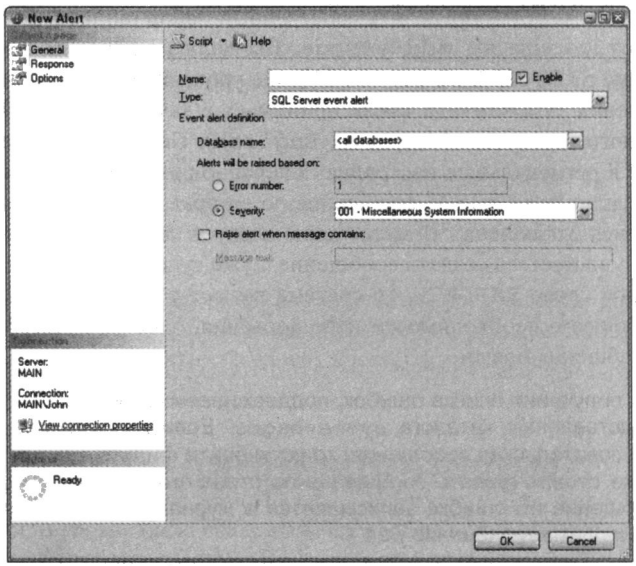


Рис. 38.6. Состояние ошибки является одним из двух событий, которые могут инициировать предупреждение

следует подойти к выбору имени со всей серьезностью. Используйте ключевые слова, позволяющие автоматически идентифицировать ошибку. Например, вы можете использовать в качестве имени "Event:Northwind:Severity Level 16".

4. В списке Type выберите тип создаваемого предупреждения. Это может быть предупреждение о событии SQL Server (вызываемое номером ошибки или ее строгостью), предупреждение, реагирующее на определенные значения счетчиков производительности SQL Server, а также предупреждение интерфейса управления WMI операционной системы. На рис. 38.7 показаны значения, соответствующие созданию предупреждения счетчика производительности. В данном случае предупреждение будет инициироваться при превышении журналом транзакций размера в 2000 Кбайт (или 2 Мбайт). С помощью этой методики можно получить доступ к любому счетчику производительности SQL Server.

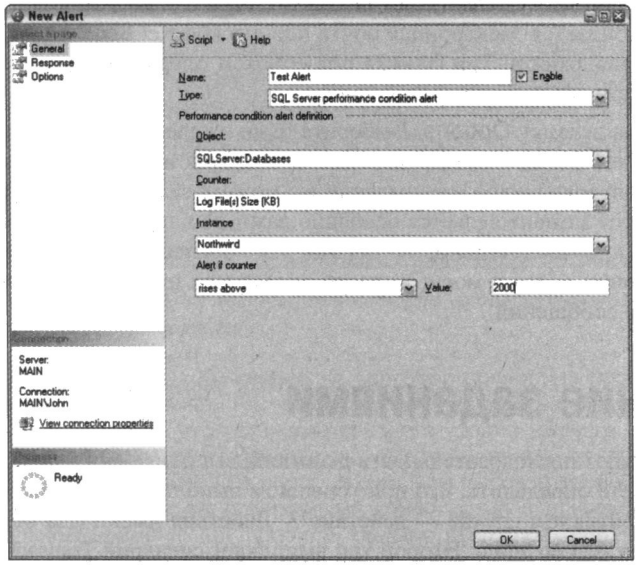


Рис. 38.7. Условие счетчика производительности является одним из факторов, инициирующих предупреждение

5. Сконфигурируйте предупреждение (состав действий зависит от выбранного на предыдущем этапе типа предупреждения).
   - Если создается предупреждение о событии SQL Server, выберите номер ошибки или ее строгость. В первом случае вы можете воспользоваться кнопкой с изображением эллипса для поиска нужного номера ошибки; во втором — сконцентрировать внимание на критических ошибках, которые имеют строгость 19 и выше. Вы можете осуществлять мониторинг как всех баз данных, так и только избранной. Также вы можете ограничить сообщения только теми, которые содержат определенную текстовую строку; фильтр при этом вводится в текстовое поле Error Message Contains This Text.
   - Если создается предупреждение, реагирующее на показания счетчика производительности SQL Server, выберите объект и счетчик, значения которого будете отслеживать. После этого установите пороговое значение счетчика и условие (превышение заданного значения, равенство ему или падение ниже него). В некоторых счетчиках нужно дополнительно определить экземпляр сервера, к которому он

применяется. Например, вы можете отслеживать размер файлов данных как всех баз, так и только избранной.

- Если создается предупреждение о событии интерфейса WMI, следует определить пространство имен данного события (например, `\\.\root\Microsoft\SqlServer\ServerEvents\MSSQLSERVER`). Кроме того, вы должны создать запрос к этому пространству имен. Все запросы WMI используют специальный язык WQL (Windows Management Instrumentation Query Language).

6. Перейдите на вкладку Responce. Определите тип отклика, который следует получить в ответ на предупреждение. На этой вкладке вы можете определить уведомление о событии одного или нескольких операторов, запуск некоторого задания, или и то и другое. Об установке заданий мы поговорим подробно в следующем разделе. Как вы уже догадались, кнопка New Operator открывает окно создания нового оператора, которое мы рассматривали в предыдущем разделе. Обычно в качестве средства отправки сообщения выбирается электронная почта или служба Net Send, если оператор не связан со службой пейджинга. Три флажка под списком операторов позволяют управлять отправкой текстовых сообщений об ошибке.
7. Перейдите на вкладку Options. Выберите один из следующих методов запроса: Email, Pager или Net Send. Стандартный тип сообщения вы можете дополнить произвольным. Для повторяющихся уведомлений вы можете задать время задержки в минутах и секундах. Этот момент является особенно важным в предупреждениях, инициируемых счетчиками производительности, так как эти события имеют тенденцию присутствовать достаточно долго и могут вывести операторов из себя постоянной отправкой одних и тех же сообщений.

Управление заданиями

Заданием называют последовательность действий, выполняемых в определенном порядке. Например, вы можете определить, что при успешном выполнении действия 1 будет выполнено действие 2, в противном случае — действие 3. Действия имеют два основных типа. Первый тип задействует репликацию. Второй тип выполняет сценарий Transact-SQL, или ActiveX (написанный на языке JScript или JavaScript), или команду операционной системы. Следует отметить, что последний тип используется чаще всего. После каждого действия можно определить следующее действие, зависящее от успеха или неудачи предыдущей операции. Вы можете воспользоваться несколькими вариантами:

- перейти к следующему действию;
- перейти к действию с заданным номером в данной последовательности действий;
- завершить выполнение задания, рапортуя о его успехе;
- завершить выполнение задания, рапортуя об ошибке.

При желании вы можете также задать число попыток выполнения действий, предпринимаемых перед тем, как сообщить о безуспешном его выполнении. С заданием может быть ассоциирован один или несколько графиков. Это позволяет автоматически запускать задание в разное время. В графике можно задать единоразовое выполнение задания в назначенное время, а также его выполнение на регулярной основе. Также можно инициировать выполнение задания во время запуска SQL Server Agent или во время простоев процессора. На вкладке Advanced диалогового окна параметров службы SQL Server Agent (рис. 38.8) можно определить, при каких условиях процессор считается простаивающим. При этом следует учитывать,

что средняя загрузка процессора не должна превышать заданный уровень в течение нескольких секунд. В заключение нужно настроить уведомления для информирования об успешном или безуспешном завершении задания.

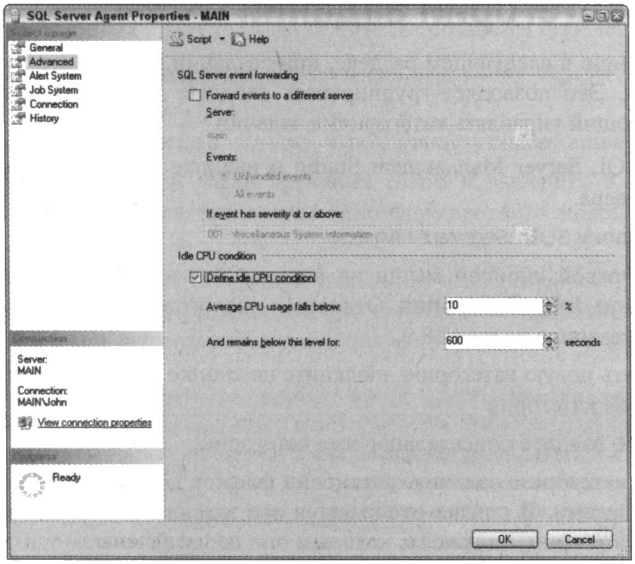


Рис. 38.8. Задайте условия, при которых процессор считается простаивающим


Проверено

Важно осознавать, что сервер не способен полностью простаивать. Даже при грамотно выполненной настройке его загруженность не опускается ниже 2–3%. Из этого следует, что установка среднего уровня загрузки в нулевое значение приведет к тому, что задание не будет выполнено никогда. В то же время не стоит устанавливать этот порог на слишком высоком уровне, чтобы не возникало войны за ресурсы с активными клиентскими заданиями, выполняемыми в реальном времени. Как правило, больше всего подходят уровни, установленные в значения от 5 до 12. Аналогично, и время простоя является важным фактором. Обычно устанавливают значения в пределах от 360 до 600 секунд. В то же время администраторы особо крупных систем могут подумать и о более высоких значениях, чтобы гарантировать наличие состояния простоя, а администраторы малых систем могут максимально эффективно использовать время простоя за счет снижения порога его продолжительности.

Некоторые мастера выполняют основную работу за кулисами. Среди таких мастеров можно упомянуть следующие: мастер плана обслуживания, мастер резервирования, а также мастера импорта и экспорта службы преобразования данных. Все формы репликации также создают задания, выполняемые в фоновом режиме.

Как и в случае с предупреждениями, все задания создаются в диалоговом окне New Job. Создание задания предполагает выполнение пяти действий.

1. Создание определения задания.
2. Настройка каждого выполняемого действия.
3. Определение следующего шага для каждого действия.
4. Конфигурирование графика запуска задания.
5. Формирование сообщений об успешном или ошибочном завершении задания.

В следующих подразделах мы подробно рассмотрим каждое из этих действий. Первый из них посвящен дополнительному этапу создания категории заданий.

Создание категории заданий

Как будет показано в следующем разделе, при создании задания его нужно назначить некоторой категории. Это позволяет группировать сходные задания. Ниже приведен список действий, позволяющий управлять категориями заданий.

1. Запустите SQL Server Management Studio и найдите папку `Management` конфигурируемого сервера.
2. Раскройте папку `SQL Server Agent`.
3. Щелкните правой кнопкой мыши на папке `Jobs` и выберите в контекстном меню пункт Manage Jobs Categories. Откроется диалоговое окно управления категориями заданий, показанное на рис. 38.9.
4. Чтобы создать новую категорию, щелкните на кнопке Add. Откроется диалоговое окно свойств новой категории.
5. В поле Name введите описательное имя категории.
6. Добавьте в категорию задания, установив флажок Show All Jobs и отметив нужные задания флажками. В списке отобразятся все задания, которым автоматически были назначены категории, а также те, которым они не назначены.

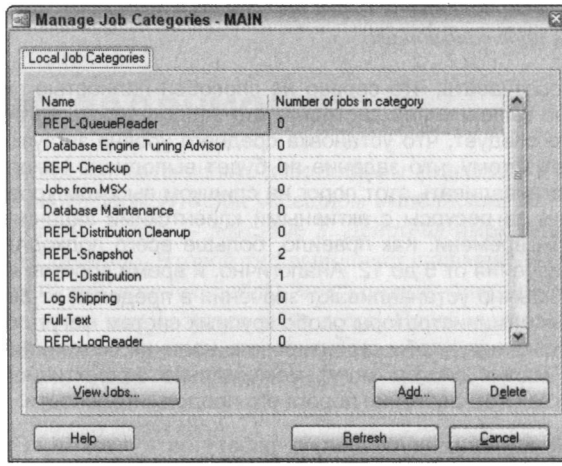


Рис. 38.9. Диалоговое окно Manage Job Categories позволяет управлять принадлежностью заданий к категориям

Диалоговое окно Manage Job Categories можно также использовать и для удаления существующих категорий. Для этого выделите задания, которые хотите удалить, и щелкните на кнопке Delete.

При желании в этом окне вы можете отобразить список заданий, назначенных выбранной категории. Выделите категорию, которую хотите просмотреть, и щелкните на кнопке View Jobs. Добавление дополнительных заданий в категории выполняется с помощью установки флажка задания в столбце Select диалогового окна New Job Category.

Создание определения задания

Главным компонентом определения задания является его уникальное имя — именно оно будет использоваться для обращения к заданию. Например, это имя можно использовать при конфигурировании реакции на предупреждение. Вот как создается определение задания.

1. Запустите SQL Server Management Studio и найдите папку `SQL Server Agent` конфигурируемого сервера.
2. Раскройте папку `SQL Server Agent`, чтобы увидеть содержащиеся в ней элементы.
3. Щелкните правой кнопкой мыши на папке `Jobs` и выберите в контекстном меню пункт New Job. Откроется диалоговое окно формирования нового задания, показанное на рис. 38.10.
4. Во вкладке General присвойте заданию уникальное имя, размер которого не должен превышать 128 символов. Выберите корректную категорию и владельца задания, введите его описание. Владельца существующего задания могут изменить только администраторы.

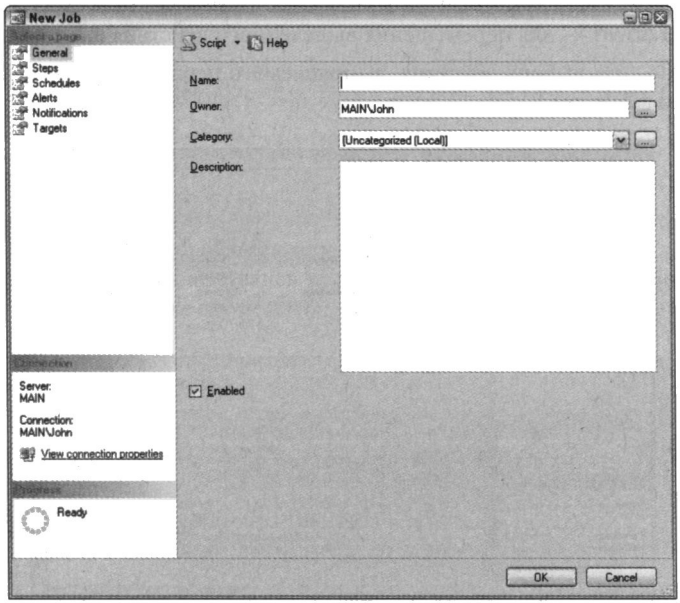


Рис. 38.10. Назначьте новому заданию категорию и владельца



В качестве владельца могут указываться только предопределенные учетные записи. Если вы не нашли запись, с которой хотите идентифицировать задание, выйдите из окна определения задания, щелкнув на кнопке Cancel, после чего пройдите полную процедуру создания новой регистрационной записи. Для этого разверните папку `Security`, находящуюся немного ниже элемента `Management`, щелкните правой кнопкой мыши на элементе `Logins` и выберите в контекстном меню пункт New Login.

5. Выделите папку `Targets`. Выберите элемент Target Local Server для заданий, предназначенных для запуска на одном локальном компьютере. Если задание охватывает множество серверов, выберите элемент Target Multiple Servers, затем выберите тот

сервер, на котором будет выполняться задание. При желании вы можете запустить задание на группе серверов; для этого выделите все соответствующие элементы.

6. Щелкните на кнопке Apply, чтобы создать определение задания. Теперь вы готовы к дальнейшим действиям, которые будут описаны в следующих подразделах.

Настройка действий, выполняемых в задании

После создания определения задания вы можете приступить к созданию составляющих его действий. Для этого перейдите на вкладку Steps (рис. 38.11) диалогового окна параметров задания. На открывшейся странице вы увидите ряд кнопок.

- New — для создания нового действия.
- Insert — для создания действия, предшествующего выделенному.
- Edit — для редактирования выделенного действия.
- Delete — для удаления выбранного действия.
- Move step up — для перемещения выделенного действия вверх по списку.
- Move step down — для перемещения выделенного действия вниз по списку.
- Start step — для выбора действия, выполняемого первым. Первое действие помечается зеленым флажком.

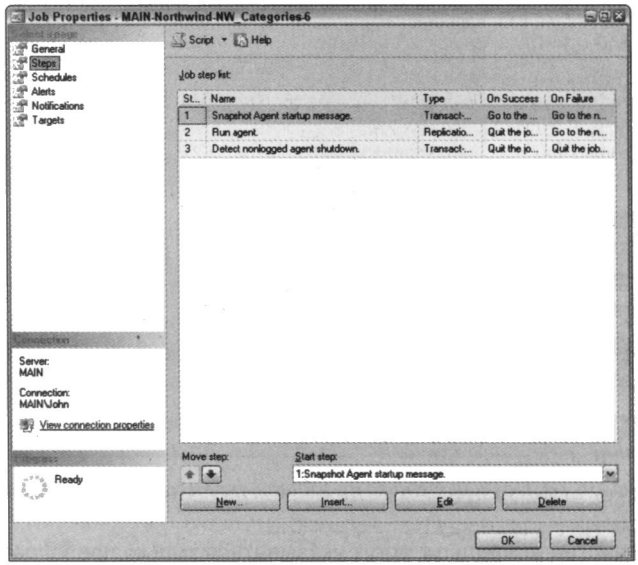


Рис. 38.11. Задание может состоять из одного или нескольких действий; все они создаются на странице Steps

При создании нового действия открывается диалоговое окно, показанное на рис. 38.12. Все действия должны иметь уникальное имя, содержащее до 128 символов. Для трех наиболее распространенных типов действий (сценарии Transact-SQL и ActiveX, а также команды операционной системы) в поле команды вводится выполняемый код. Вы можете также воспользоваться кнопкой Open для загрузки программного кода из файла. Кнопка Parse позволяет проверить синтаксис команды.

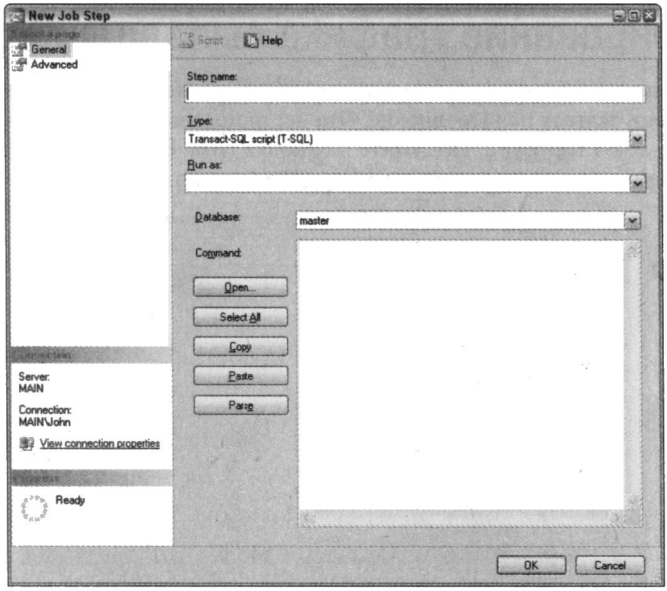


Рис. 38.12. Одно действие может содержать любой код Transact-SQL

После ввода кода для действия перейдите на вкладку Advanced (рис. 38.13) диалогового окна New Job Step и определите, что должно произойти после выполнения данного действия. В этой же вкладке вы можете определить количество попыток выполнения действия, если оно закончилось неудачей, а также промежуток времени между этими попытками (в минутах).

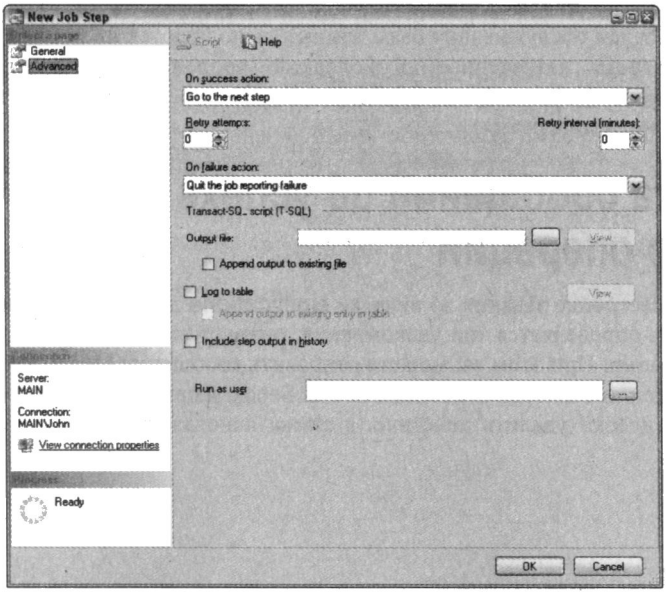


Рис. 38.13. Во вкладке Advanced определяется, что должно произойти после выполнения действия

Конфигурирование графика выполнения задания

После ввода всех действий формируемого задания вы можете определить, когда данное задание должно запускаться на выполнение. Эта настройка выполняется во вкладке Schedule диалогового окна Job Properties. Щелкните на кнопке New Schedule, чтобы открыть диалоговое окно формирования нового графика, показанное на рис. 38.14.

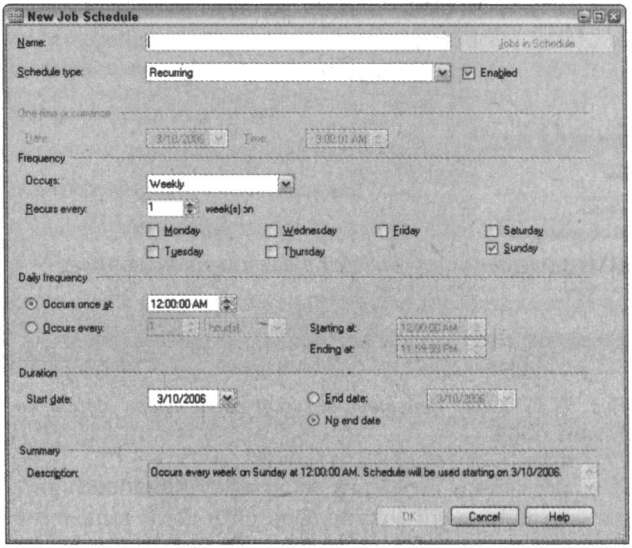


Рис. 38.14. Задание может быть запрограммировано как для единоразового, так и для регулярного выполнения

Большинство задач обслуживания базы данных выполняются на регулярной основе. Если вам не подходит время, назначенное по умолчанию (полдень воскресенья), вы можете сами задать частоту и время начала. Как вы могли увидеть на рис. 38.14, при определении графика у вас полностью развязаны руки.

Обработка сообщений об успехе и неудаче операции

Завершается настройка задания во вкладке Notifications диалогового окна Job Properties (рис. 38.15). Здесь определяется тип уведомления, генерируемого при успешном и неудачном завершении операции. При этом вы можете отправить сообщение оператору (по электронной почте, на пейджер или с помощью сообщения Net Send), запротоколировать соответствующее событие, автоматически удалить действие, а также использовать любую комбинацию этих операций.

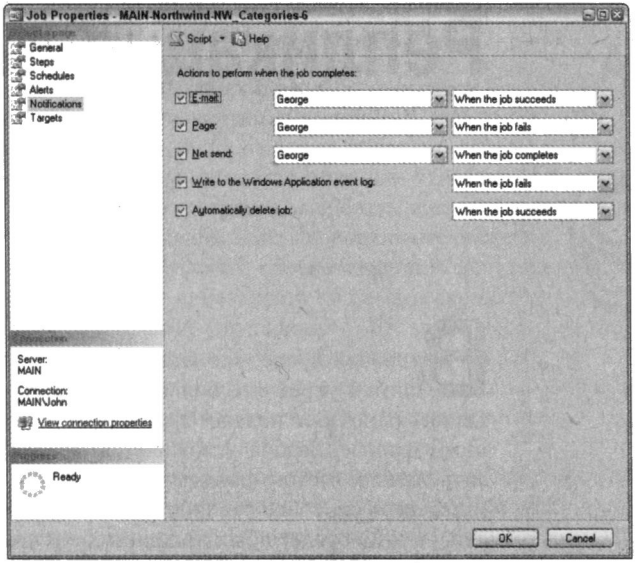


Рис. 38.15. Определите тип уведомления, используемого при удачном или неудачном завершении задания

Резюме

Служба SQL Server Agent станет вашим союзником, помогая не забыть о критичных задачах обслуживания базы данных и о событиях, которые требуют вашего или постороннего вмешательства. Первая цель достигается путем планирования повторяющихся заданий; вторая — с помощью предупреждений.

В этой главе мы рассмотрели вопросы, связанные с настройкой службы SQL Server Agent. Вы узнали, что собой представляют задания, операторы и предупреждения, ознакомились с действиями, необходимыми для управления этими объектами. Короче говоря, теперь вы можете считать себя полностью подготовленным для автоматизации критичных задач обслуживания базы данных.

ГЛАВА

39

Репликация данных

В этой главе...

Эффективное использование репликации

Модель репликации компании Microsoft

Создание четырех типов репликаций

Отличия репликации в SQL Server 2005

Определение издателей и подписчиков

В любой среде, где администратору баз данных приходится управлять больше чем одним экземпляром SQL Server, рано или поздно приходится распространять между ними данные. Задача усложняется, если приходится переносить данные между разными СУБД. SQL Server предлагает множество способов распространения данных; в новой версии список этих средств был дополнительно расширен. В отличие от других СУБД, решения распространения данных от Microsoft доступны бесплатно; в то же время предлагаемое ими многообразие методов и их надежность делают их незаменимыми компонентами любой системы распространения данных.

Зачем реплицировать данные

Администраторам может понадобиться репликация данных по нескольким причинам.

- Поддержка восстановления данных при аппаратных и программных сбоях.
- Требования приложений.
- Выигрыш в производительности.
- Распространение информации.

Поддержка восстановления данных при аппаратных и программных сбоях

Нечувствительность к аппаратным и программным сбоям является той степенью устойчивости, которая обеспечивает живучесть баз данных. Сбои обычно делят на две категории.

- **Локальные**. Эти сбои связаны с отказами программных и аппаратных средств, действиями некомпетентных пользователей и прочими эксцессами.

- **Региональные**. Эти сбои связаны с глобальными разрушениями, вызванными землетрясениями, наводнениями, атаками террористов и др.

В общем случае пользователи снисходительно относятся к региональным сбоям. Обычно клиенты защищаются от таких региональных событий, как отключение электропитания, которое привело к отказу системы. В то же время довольно сложно объяснить клиенту, что он не сможет снять деньги в банкомате по той причине, что некий Вася случайно выдернул из розетки штепсель бухгалтерского сервера. Клиенты, как правило, не понимают локальных событий, которые не оказали влияния на их персональные системы.

Организационный уровень решений, обеспечивающих нечувствительность к сбоям, определяется тем, какое время будет простаивать система и сколько данных будет потеряно. Например, большинство финансовых учреждений в большей мере чувствительны к простоям, чем магазины подержанных товаров. Компании, которым необходима минимизация времени простоя, могут реализовать дорогостоящее решение обеспечения максимальной доступности сервера. Такое решение, как правило, задействует кластеризацию для защиты от локальных сбоев, а также геопространственную кластеризацию для защиты от региональных сбоев. Такое решение обеспечивает автоматическое восстановление работоспособности систем. Каким бы ни был сбой, локальным или региональным, клиент автоматически переключается на другие серверы, не подверженные сбою. При этом затраты на такие переключения минимальны, если вообще существуют.

Кластеризацией называют технологию, обеспечивающую автоматическое восстановление работоспособности систем после отказа. Кластеризация предполагает группировку двух и более серверов (называемых *узлами*) для совместной работы в качестве единого виртуального сервера. Узлы кластера совместно используют общие ресурсы. Кластеризация необходима для автоматического восстановления работоспособности системы. Виртуальный сервер постоянно переключается между ресурсами отдельных узлов. В каждый конкретный момент времени ресурсы для виртуального сервера может поставлять один узел. Если этот узел подвергается сбою или переводится в автономный режим для выполнения задач обслуживания, ресурсы виртуальному серверу начинает предоставлять другой узел. При этом клиенты практически не ощущают задержку во времени.

Большинство финансовых учреждений используют *зеркальное отображение* данных на аппаратном уровне. При этом они используют оборудование от таких производителей, как EMC, Hitachi или Veritas. Эти решения в большей мере масштабируемы, чем кластеризация. Они способны обеспечить устойчивость к отказам регионального уровня и автоматическое восстановление. Кластеризация ограничена расстоянием между узлами (обычно позволяющими получить отклик от опрашиваемого узла в течение 500 миллисекунд).

Другими средствами обеспечения отказоустойчивости в SQL Server 2005 являются зеркальное отображение данных, доставка журналов и репликация. SQL Server 2005 также поддерживает автоматическое восстановление, однако режим высокой доступности повышает время подтверждения транзакций в источнике и требует участия третьего сервера. Зеркальное отображение данных не допускает большое масштабирование, особенно в условиях высокой нагрузки.



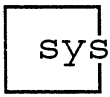
Зеркальное отображение данных, внедренное в SQL Server 2005, предназначено для решения проблем предыдущих версий, связанных с обеспечением надежности хранения информации. Использование этой функции в значительной мере повысило надежность сервера баз данных, равно как и доступность данных.

Доставка журнала является еще одной альтернативой, обеспечивающей отказоустойчивость, но не автоматическое восстановление. В этой модели резервная копия восстанавливается на сервере назначения, затем журнал перебрасывается с сервера источника на сервер на-

значения, после чего применяется к базе данных. Несмотря на то что журнал транзакций можно перебрасывать каждую минуту, на практике редко используется промежуток времени меньше пяти минут. Это значит, что решение, использующее доставку журнала, не столь масштабируемо, как кластеризация, аппаратное и программное зеркальное отображение данных. В то же время доставка журнала хорошо зарекомендовала себя в ситуациях, когда требуется синхронизация двух баз данных, в которых часто выполняются изменения схемы.

Для обеспечения отказоустойчивости может использоваться и репликация. Однако в данном случае существуют некоторые особенности.

- Автоматическое восстановление невозможно.
- Если используется репликация транзакций, то все таблицы должны иметь первичные ключи. Репликация транзакций лучше всего подходит для обеспечения отказоустойчивости, поскольку гарантирует создание точной копии реплицируемых данных.



Чтобы просмотреть метаданные транзакций, реплицированных в публикации, выберите в динамическом представлении управления `sysdm_repl_tranhash`.

- При репликации изменяются объекты на сервере назначения, что иногда неприменимо при восстановлении работоспособности систем. Например, свойство идентичности столбца удаляется, первичные ключи заменяются уникальными индексами, свойство `NOT FOR REPLICATION` не используется, а декларативная ссылочная целостность, равно как и некоторые индексы сервера источника, не реплицируются.
- Системные объекты не реплицируются.
- Изменения в первичных ключах сервера источника не реплицируются.

При использовании репликации для обеспечения отказоустойчивости вы должны сконфигурировать сервер назначения так, чтобы учесть перечисленные выше особенности. Репликация освобождает базу данных назначения для полезной работы, а показатели времени задержки лучше, чем при доставке журнала.

Требования приложений

Репликация часто используется, когда приложению нужно консолидировать данные из одного или нескольких серверов в центральное хранилище. Также во время перемещения между серверами данные могут быть трансформированы. Зачастую администраторам баз данных приходится работать в условиях существования центрального офиса и нескольких филиалов. В этом случае ему приходится консолидировать данные о продажах в центральное хранилище, а также распространять по филиалам централизованную информацию, такую как каталоги продукции. Еще один пример предполагает наличие коммивояжеров, которые в дневное время посещают клиентов, оформляют заказы, а по возвращении в офис синхронизируют информацию с центральным сервером. Классическим примером такой ситуации является служба доставки, которая использует карманные компьютеры (с установленной СУБД SQL Server Everywhere) и выполняет синхронизацию с центральным сервером в точках назначения. Репликация может использоваться для централизованной консолидации данных, для переноса данных между серверами, для обмена данными между экземплярами SQL Server и продуктами других производителей СУБД, такими как Oracle, Sybase и др. С версии SQL Server 2005 стала доступной репликация баз данных Oracle в SQL Server. Репликация может использоваться также и для синхронизации информации в пределах одной базы данных.

Выигрыш в производительности

Репликацию часто используют, когда необходимо перенести функцию отчетности на отдельный сервер, освободив главный. В противоположность другим способам распространения данных (репликации журнала и зеркальному отображению данных), репликация позволяет создать на сервере отчетности полностью отличные индексы и даже таблицы, наполнив их все теми же данными центрального сервера. Вы можете также повысить производительность операций чтения, реплицируя одни и те же данные в банк серверов. Вместо того чтобы замыкать тысячу пользователей на один сервер, можно объединить в единый банк 10 серверов, снизив нагрузку на каждый из них до 100 пользователей. Это позволило бы повысить производительность операций чтения в 10 раз. Еще одним примером повышения производительности с помощью репликации является объединение в глобальную сеть нескольких офисов одной организации. Задержки, связанные с подобной организацией сетей, могут замедлить работу приложений. Несмотря на то что использование серверов Citrix или службы терминалов идеально подходит для подобных распределенных сетей, зачастую более эффективным является подход с использованием серверов SQL Server в удаленных офисах и их репликации с центральным сервером. При этом приложения в удаленных офисах будут обращаться к локальным базам данных. Изменения, выполняемые ими, будут впоследствии реплицированы на центральный сервер. Еще один интересный подход связан с репликацией данных на рабочие узлы, выполнением в них вычислений и последующей репликацией результатов назад, на центральный сервер. Такой подход позволяет в значительной мере выиграть в производительности, так как обработка пакета распределяется среди нескольких рабочих узлов.

Распространение данных

Репликацию часто используют для поддержки больших групп пользователей, связанных с разными серверами. Каждый из этих серверов располагает копией одних и тех же данных, поэтому не имеет значения, на какой из них полагается конкретный пользователь. Распределение данных позволяет получить также и другие преимущества использования репликаций. Во-первых, повышается отказоустойчивость, так как на разных компьютерах существуют копии одних и тех же данных. Даже если один компьютер испытывает потребность в использовании автономного режима работы, его пользователи могут получить доступ к тем же данным, размещенным на других компьютерах. Зачастую подобная избыточность отвечает специфичным потребностям приложений, таким как производительность и доступность. В заключение следует сказать, что разрешение пользователям доступа к распределенным источникам позволит повысить общую производительность системы, так как запросы будут направляться к наименее загруженному компьютеру.

Сравнительная характеристика различных методов распределения данных

Поскольку существует множество методик распределения данных, очень важно осознать, как варьируются их свойства. Это позволит в каждой конкретной ситуации определить, является ли репликация наилучшим решением.

- **Распределенные транзакции**. Этот вариант является программным решением — транзакции применяются как на стороне сервера источника, так и на стороне сервера приемника, обычно в пределах одной транзакции. За это приходится платить опреде-

ленную цену, выражаемую, как правило, задержкой операции и дополнительным объемом программирования. Распределенные транзакции используются, когда обязательна синхронизация в реальном времени; они довольно распространены в финансовых решениях.

- **Связанные серверы**. Связанные серверы аналогичны распределенным транзакциям, однако доступ к связанному серверу осуществляется в базе данных, а не в программном коде. Падение производительности, обусловленное связыванием серверов, не позволяет масштабировать такие решения. Связанные серверы осуществляют доступ к более широкому множеству разнообразных источников данных, чем репликация SQL Server. Обычно этот метод используется для подключения к источникам данных, которые не поддерживаются репликацией или требуют прямого доступа из запросов.
- **Триггеры**. Могут использоваться для распространения данных, однако чаще всего в пределах локального сервера. С ними связаны повышение нагрузки на ресурсы сервера и снижение производительности. Обычно триггеры используют для аудита.
- **Служба интеграции** (ранее известная как служба преобразования данных DTS). Служба интеграции может распространять данные, к тому же она имеет достаточно богатый набор средств преобразования данных в процессе их перемещения с сервера источника на сервер назначения. Служба интеграции может быть использована для распространения данных в гораздо более широкий круг источников данных, чем может позволить репликация SQL Server. Служба интеграции не отслеживает отдельные транзакции; если вы решили ее использовать, значит, вам придется каждый раз полностью заменять весь набор данных или самостоятельно создавать некоторый механизм, позволяющий определять, какие именно данные были изменены в источнике для репликации только их подмножества. Службу интеграции обычно используют в пакетных процессах, которые не накладывают ограничения на задержку операций.



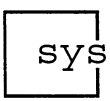
Изоляция моментальных снимков (SNAPSHOT) была нацелена на решение вопросов производительности и доступности, связанных с определенными недостатками SQL Server 2000. Используя эту функцию, вы можете повысить общую производительность и облегчить пользователям получение необходимых им данных.

- **Резервирование/восстановление**. Все перечисленные выше варианты используются для распространения отдельных транзакций; резервирование/восстановление связано с распространением базы данных в целом. Это решение является единственно правильным, когда пользователю нужно скопировать всю базу данных, к которой они должны относиться как к доступной только для чтения, так как при следующем восстановлении архива она будет полностью замещена новой копией. Во время выполнения восстановления из архива пользователи должны отключаться от базы данных. К тому же не существует возможности перенесения работы, выполненной на сервере-приемнике, на сервер-источник. Также не существует возможности распространять только часть данных — вы можете иметь дело только с базой данных в целом.
- **Доставка журнала**. Доставку журнала можно рассматривать как постепенное восстановление из резервной копии. Вначале на сервере назначения восстанавливается полная резервная копия, после чего следуют периодические восстановления журнала транзакций. Журналы могут восстанавливаться по порядку или все вместе. Во время восстановления пользователи должны отключаться от базы-приемника. Не существует возможности восстановления только фрагмента данных — как и в предыдущем случае, вы ограничены только базой данных в целом. Доставка журнала не обеспечивает такой маленькой задержки, как репликация, и не позволяет отправлять на сервер

назначения только подмножество данных. Также вы не имеете возможности манипулировать данными во время их передачи с сервера-источника на сервер-приемник.

- **Зеркальное отображение данных в Microsoft SQL Server**. Введенное в версии SQL Server 2005, зеркальное отображение данных поддерживает один и тот же набор данных на двух серверах. Этот механизм требует наличия третьего сервера (называемого *свидетелем*), который определяет, когда сервер-источник переходит в автономный режим, после чего перенаправляет пользователей на сервер-приемник. Зеркальное отображение данных использует последовательную доставку журнала. Эта технология хорошо зарекомендовала себя при работе с базами данных среднего размера, обеспечивая при этом относительно невысокую загрузку ресурсов.
- **Кластеризация**. Требует специального аппаратного оснащения и разных версий операционной системы. Кластеризация предполагает совместное использование узлами баз данных, СУБД SQL Server и ресурсов. Клиенты подключаются к виртуальному серверу баз данных, который, в свою очередь, обращается за ресурсами к одному из своих узлов. Если основной узел дает сбой, виртуальный сервер переключается на запасной узел, и это практически не заметно клиенту. Кластеризацию нельзя рассматривать как технологию распространения данных — скорее, это решение обеспечения отказоустойчивости и высокой доступности данных. Кластеризация может использоваться для обеспечения отказоустойчивости при репликациях; она как бы "распространяет" один сервер между несколькими физическими узлами.
- **Программное зеркальное отображение данных**. Эти продукты устанавливают драйвер файловой системы, который перехватывает действия над файлами и реплицирует их на сервер назначения. Эти программные продукты потребляют ресурсы главного узла (т.е. сервера источника). Они обеспечивают функциональность, аналогичную аппаратному зеркальному отображению данных, однако с несколько более низкой производительностью. Примером программного решения обеспечения зеркального отображения данных является продукт Double-Take.
- **Аппаратное зеркальное отображение данных**. Аппаратное зеркальное отображение данных основано на компонентах таких производителей, как Hitachi Data Systems, EMC и Veritas. Это оборудование реплицирует изменения, происходящие на байтовом уровне, из одного дискового массива в другой. Эти решения сильно задействуют кэширование, они дорогостоящие и сложные. В то же время они незаменимы, когда требование доступности данных ставится во главу угла.

Приведенный выше список упорядочен по уровню гранулярности. Например, первые три альтернативы распространяют данные на уровне транзакций. Преобразования данных и репликации выполняются на уровнях статей и таблиц. Доставка журнала и резервирование/восстановление выполняются на уровне базы данных. Кластеризация, а также зеркальное отображение данных выполняются на уровне серверов и дисковых массивов. Порядок этих решений также отражает степень подготовки администратора базы данных и уровень затрат, хотя и стоимость программирования распределенных транзакций может быть существенной.



Для просмотра метаданных объектов базы, опубликованных в статьях топологии репликации, выберите `sysdm_repl_articles` в динамическом представлении управления.

Репликация позволяет отображать подмножества таблиц, столбцов и строк. Также в ней могут участвовать следующие объекты:

- таблицы, разбитые на разделы;

- схемы хранимых процедур;
- представления, как простые, так и индексированные (в последнем случае они реализуются либо как индексированные представления, либо как таблицы, представляющие их);
- пользовательские типы и функции;
- полнотекстовые индексы;
- псевдонимы типов данных;
- объекты схемы (ограничения, индексы, триггеры, расширенные свойства, сопоставления и т.п.);
- инструкции DDL (такие как `ALTER TABLE` и т.п.).

Задержка, вызываемая репликациями в производственной среде с полной нагрузкой, обычно не превышает одной минуты. Репликации можно рассматривать как недорогую альтернативу, предлагающую более высокую степень гибкости, чем другие решения.

Модель репликации от Microsoft

Модель репликации, предложенная компанией Microsoft, основана на сервере репликаций от Sybase (собственно. и сам сервер баз данных SQL Server основан на решениях Sybase). Компоненты репликации SQL Server входят во все варианты поставки, кроме SQL Server 2005 Express и SQL 2005 Mobile Edition. В то же время обе эти редакции могут являться подписчиками публикаций SQL Server 2005. (SQL Server 2005 Mobile Edition является новой версией SQL CE, которая запускается на карманных компьютерах и смарт-устройствах. Она может быть только подписчиком публикаций слияния.) Модель Microsoft использует метафоры периодических изданий. (В следующих подразделах будет описан каждый из этих компонентов.) В поставку SQL Server 2000 были включены примеры программ, иллюстрирующих публикации из гетерогенных серверов в среде SQL Server. Все эти функции остались доступными и в SQL Server 2005.

Издатель

Издателем называют сервер-источник, который содержит реплицируемые объекты и данные. Обычно издателем является SQL Server 2005, однако им может быть SQL Server 2000 и Oracle Server. SQL Server поставляется с провайдером репликаций, позволяющим создавать издателей сторонних серверов баз данных.



SQL Server 2005 может выступать шлюзом между сервером Oracle и системой гетерогенных подписчиков. В этой модели можно опубликовать данные Oracle для SQL Server 2005, где впоследствии их переиздать для гетерогенных подписчиков.

Подписчик

Подписчик в модели репликации получает схему объектов и их данные (т.е. *подписку*) от издателя. Каждый подписчик может получать несколько подписок от разных издателей. Подписчиками могут выступать SQL Server 2005, SQL Server 2005 Mobile Edition, SQL Server 2000 и SQL Server 7. Большей своей частью подписки доступны только для чтения, однако существуют некоторые типы репликаций, позволяющие подписчикам публиковать данные у издателя.

Распространитель

Распространителем называют SQL Server, который распространяет схему, данные и транзакции, опубликованные у издателя, среди подписчиков. Распространитель также хранит историческую информацию о подписках. Чаще всего издатель и распространитель находятся на одном компьютере. В то же время при значительной нагрузке некоторые типы репликаций только выиграют, если распространитель будет перенесен на отдельный сервер, т.е. отделен от издателя. Ядром распространителя является *база данных распространения* — хранилище исторической информации и метаданных, используемых репликациями моментальных снимков и транзакций. Издатель может использовать только одну базу данных распространения. Распространитель может содержать несколько баз данных распространения, ориентированных на конкретных издателей или их группы.

Модель с централизованным издателем

Эта топология является самой распространенной. В ней один издатель создает публикации для множества подписчиков. Эта модель чаще всего используется, когда существует один главный офис или хранилище данных, распространяющие материалы по своим филиалам, серверам отчетности, рабочим узлам и т.п.

Модель с централизованным подписчиком

Эта топология обычно используется для консолидации данных: например, от филиалов в центральный офис, на сервер отчетности или в централизованное хранилище данных. В этой модели удаленные издатели могут публиковать для централизованного подписчика как одни и те же, так и разные статьи.

Модель с переизданиями

Переизданием называют ситуацию, когда подписчик одновременно выступает издателем для других подписчиков, публикуя для них статьи, полученные от главного издателя. Эта модель обычно используется для масштабирования решений репликации. Например, в организационной структуре может существовать единый центральный офис, который распространяет информацию в свои региональные офисы, которые, в свою очередь, выступают издателями для своих местных подписчиков. Чтобы модель с переизданиями хорошо работала, нужно четко представлять себе все потоки данных, а также все серверы, порождающие данные или являющиеся их авторизованным источником. Это может привести к созданию полноценной модели одноранговой репликации. Если вы не имеете такого контроля над потоками данных, то дублирующие друг друга данные могут вводиться в разных региональных центрах; также могут образоваться своеобразные "черные дыры", поглощающие информацию, но не переиздающие ее для конечных потребителей.

Одноранговая модель

Эта модель является новой в SQL Server 2005. Она аналогична двусторонней репликации транзакций, однако та ограничена всего двумя узлами. В одноранговой модели SQL Server реплицирует данные всем другим серверам, использующим репликацию транзакций. Если один из узлов, участвующих в такой схеме распространения данных, переходит в автономный режим, то все остальные узлы сохраняют для него информацию, продолжая обмен между собой.

Как только отключенный узел снова становится доступным, с ним выполняется синхронизация. Эта модель репликации создавалась с парадигмой "все всем"; она обеспечивает повышенную производительность операций чтения и записи, а также повышенную доступность, реализованную с помощью масштабирования базы данных на множество узлов. Пользователь может подключиться к любому узлу, участвующему в этой модели, внести изменения, после чего эти изменения отразятся на всех узлах, участвующих в данной топологии. Для успешной работы этой модели данные должны быть разделены, чтобы минимизировать конфликты. Конфликты могут возникать в следующих ситуациях.

- Когда данные с одним и тем же первичным ключом одновременно вставляются на разных узлах, после чего синхронизируются с другими серверами. В этом случае возникает коллизия первичных ключей.
- Когда данные обновляются на одном узле и одновременно удаляются на другом.
- Когда инструкции обновления или удаления применяются к разному количеству строк, будучи выполненными на разных узлах. Это приводит к тому, что база данных будет иметь разный состав данных на разных узлах, вследствие чего ее целостность может быть нарушена.

Статья

Статьей называют минимальную единицу публикации. Статья может представлять собой элемент:

- таблиц;
- разделенных таблиц;
- хранимых процедур (CLR и T-SQL);
- выполнений хранимых процедур (CLR и T-SQL);
- представлений;
- индексированных представлений;
- индексированных представлений в форме таблиц;
- пользовательских типов (CLR);
- пользовательских функций (CLR и T-SQL);
- псевдонимов типов данных;
- полнотекстовых индексов;
- объектов схемы (ограничений, индексов, триггеров, расширенных свойств, сопоставлений и т.п.).

Для представлений, индексированных представлений, хранимых процедур и функций реплицируется только схема; индексированные представления могут реплицироваться в таблицы; также вы можете реплицировать выполнение хранимых процедур. Обычно из административных и логистических соображений одна публикация создается для одной базы данных; при этом в нее включается множество статей. Одной из немногих причин включения статей, относящихся к одной базе данных, в разные публикации является то, что они относятся к разным типам репликаций (например, слияния и транзакций). В SQL Server 2000 было целесообразно разбивать статьи на несколько публикаций по двум причинам.

- **Повышение производительности.** Независимые параметры агента в репликации транзакций могут создать два или более параллельных потока данных, направленных к подписчику.

- **Повышение надежности**. Репликация крупных таблиц может стать источником проблем. Помещение крупных таблиц в собственные публикации позволяет использовать альтернативные методы их развертывания. Также это позволяет восстанавливать их после сбоев, не оказывая влияния на репликации других таблиц, принадлежащих к другим публикациям.

По умолчанию в SQL Server 2005 можно начинать прерванную репликацию снимков баз данных с места останова (если агенту не удалось выполнить репликацию за один прием, в следующий раз он реплицирует только отсутствующие фрагменты, не начиная операцию с самого начала), при этом включен параметр независимости.

В общем случае лучше группировать публикации по таблицам и зависимостям данных.

Принудительная подписка

Принудительная подписка инициируется издателем. Другими словами, именно издатель управляет тем, когда снимок базы данных и последующие репликации будут отправляться подписчику. Отслеживание репликаций выполняется на стороне издателя, там же хранятся метаданные.

Принудительные подписки обычно используют в следующих ситуациях.

- Когда число подписчиков относительно невелико.
- Когда репликация выполняется в пределах локальной сети или интрасети — принудительные подписки не работают через Интернет.
- Когда необходима централизация управления репликацией.

Принудительные подписки являются их простейшей формой, так как их проще всего развертывать. Вследствие этого и используются на практике они чаще всего.

Подписка по запросу

Подписка по запросу управляется на стороне подписчика, получающего снимки базы и данные. Этот тип подписки снижает загрузку сервера издателя (распространителя), так как агенты распространения и слияния запускаются на стороне подписчика. Это также делает подписку по запросу более масштабируемым решением. Подписки по запросу подходят в следующих ситуациях.

- Когда число подписчиков сравнительно велико.
- Когда выполняются репликации через ненадежные коммуникационные каналы (в том числе через Интернет), а также в автономном режиме работы подписчиков.
- Нет необходимости в централизованном узле управления.

Подписки по запросу требуют большей работы по развертыванию и обслуживанию. В то же время они больше подходят для крупных реализаций с использованием репликаций.

Типы репликаций

Существуют три основных типа репликаций:

- репликация снимков баз данных;
- репликация транзакций;
- репликация слияния.

Репликации снимков баз данных и транзакций больше всего подходят в ситуациях, когда на стороне подписчиков базы данных используются только для чтения. Изменения выполняются исключительно на стороне источника данных, после чего реплицируются подписчикам. В то же время оба подтипа репликаций позволяют выполнять изменения на стороне подписчика, после чего направлять их издателю. Репликация слияния создавалась для двусторонней репликации с клиентами, которые часто выходят в автономный режим. Примером такой ситуации является сеть коммивояжеров и представителей, выезжающих для составления заказа к клиентам со своими ноутбуками, периодически синхронизируя информацию в центральном офисе.

Репликация снимков баз данных

Этот тип репликации использует метафору *снимка* из области фотографии; он фиксирует некоторую "точку во времени" состояния данных и высылает ее на сервер назначения. Репликация снимков баз данных идеально подходит при выполнении следующих условий.

- Природа данных не связана со временем и изменяется не часто.
- В дискретные моменты времени происходят глобальные изменения в данных.
- Объем данных не очень большой.

Репликация снимков баз данных не требует наличия первичных ключей во всех реплицируемых таблицах. По этой причине многие администраторы баз данных используют ее вместо репликации транзакций, в которой наличие первичных ключей является обязательным. В то же время часто в таблицах, участвующих в репликациях снимков баз данных, присутствуют ключи-кандидаты. Если вы сможете идентифицировать этих кандидатов и преобразовать их в первичные ключи, то лучшим решением будет использование репликации транзакций.

Репликация снимков баз данных позволяет реплицировать следующие объекты:

- таблицы;
- разделенные таблицы;
- хранимые процедуры;
- представления хранимых процедур;
- индексированные представления;
- индексированные представления в форме таблиц;
- пользовательские типы (CLR);
- пользовательские функции (CLR и T-SQL);
- псевдонимы типов данных;
- полнотекстовые индексы;
- объекты схемы.

Вы можете также реплицировать подмножества таблиц и представлений, например подмножества столбцов или строк. Репликация снимков баз данных использует два компонента:

- агент снимков;
- агент распространения.

Агент снимка данных генерирует сценарий создания объектов и данных для репликации. Созданный сценарий агент записывает в папку файловой системы, называемую *папкой снимков*. Команды также записываются в базу данных распространения, которую агент распространения использует для создания объектов на стороне подписчика; при этом он обращается

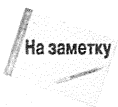
к сценариям, находящимся в папке снимков. Агент распространения направляет данные от издателя к подписчику.

Репликация снимков баз данных не в такой мере масштабируема, как репликация транзакций, особенно если пропускная способность каналов коммуникаций невелика. Когда возможно, лучше используйте репликацию транзакций, так как в этом случае передаются только изменения в базе данных. В случае репликации снимков баз данных каждый раз передается весь массив данных издателя.

Репликация снимков баз данных с непосредственным обновлением

Этот подтип репликации позволяет изменениям, внесенным на стороне подписчика, быть реплицированными у издателя с помощью процесса двухэтапного подтверждения. При этом используется координатор распределенных транзакций от компании Microsoft. Изменения, произошедшие у подписчика, остаются на его сервере до тех пор, пока следующий снимок базы не будет сгенерирован агентом снимков и распределен агентом распределения. В этом процессе первое подтверждение выполняется на стороне издателя, где реплицируются транзакции, выполненные на стороне подписчика, а затем второе подтверждение происходит уже у подписчика. Этот процесс имеет побочный эффект. Он состоит в том, что диапазоны идентичности у подписчика и издателя постоянно находятся в синхронизации. Этот подтип репликации снимков базы данных рекомендуется использовать, когда подавляющее большинство транзакций выполняется на стороне издателя. Следует отдельно отметить два недостатка этого метода репликации.

- Увеличение времени подтверждения транзакции на стороне подписчика, так как перед этим выполняется подтверждение у издателя.
- Если издатель переходит в автономный режим или соединение между издателем и подписчиком обрывается, то транзакция "подвешивается" у подписчика примерно на 20 секунд и только после этого откатывается. Приложение должно быть готово к правильной реакции на такое событие.



Этот вариант репликации нельзя настроить с помощью мастера — только с помощью хранимой процедуры репликации.

Репликация снимков базы данных с очередью обновлений

Этот подтип репликаций использует триггеры реплицируемых таблиц подписчика для регистрации изменений в очереди. Эта очередь впоследствии читается *агентом чтения очереди* (*queue reader agent*), который применяет эти транзакции у издателя. Изменения, внесенные у издателя, остаются у него до следующего запуска агента снимков. В версии SQL Server 2000 существовала возможность использования для хранения очереди MSMQ (Microsoft Message Queue) и таблицы SQL Server. В SQL Server 2005 очереди обслуживаются исключительно сервером баз данных. Репликацию снимков баз данных с очередью обновлений нельзя назвать подходящим вариантом для топологий, имеющих больше десятка подписчиков. Этот вариант репликации также нельзя настроить с помощью мастеров — только с помощью хранимых процедур репликации.

Репликация снимков базы данных с непосредственным обновлением и очередью восстановления

Этот тип репликации позволяет переключать непосредственное обновление в использующее очередь, если сервер издателя по тем или иным причинам нужно перевести в автономный режим. Отметим, что этот вариант репликации также нельзя настроить с помощью мастера — только с помощью хранимых процедур репликации.

Репликация транзакций

Репликация снимков базы данных отправляет подписчикам полный набор данных (или его часть, если использована фильтрация столбцов и строк) при каждой генерации снимка. Этот тип репликации обеспечивает синхронизацию на определенный момент времени. После генерирования снимка и его распространения подписчикам у последних начинается постепенное рассогласование данных с издателем. Эта синхронизация восстанавливается только после следующей репликации. Промежуток времени, который подписчик остается рассогласованным с издателем, называется *задержкой*. Задержка в репликации снимков базы данных довольно велика.

Репликация транзакций для выполнения синхронизации базы данных с подписчиками использует журнал транзакций. Она более масштабируема, чем репликация снимков базы данных, так как реплицируются только изменения. Задержки этой репликации меньше, чем у репликации снимков базы данных, поскольку изменения постоянно применяются к подписчикам.

Репликация транзакций использует три компонента:

- агент снимков базы данных генерирует схему, данные и необходимые репликации метаданных для отслеживания процесса репликации;
- агент распределения распространяет снимок и последующие команды;
- агент чтения журнала читает журнал транзакций базы данных издателя.

Агент снимков

Агент снимков (Snapshot Agent) используют все три типа транзакций. Он генерирует схему реплицируемых объектов и связанных с ними данных, а также необходимые метаданные отслеживания, которые затем помещает в виде текстовых и двоичных файлов в заданную папку снимков. После этого он записывает данные отслеживания в базу данных распространения, которую впоследствии читает агент распространения для синхронизации подписчиков с издателем.

Агент чтения журнала

Агент чтения журнала (Log Reader Agent) ищет транзакции, которые должны быть реплицированы подписчикам. Идентификаторы этих транзакций он записывает в таблицу базы данных распространения, имеющую название `msrepl_transactions`. При этом все транзакции разбиваются на отдельные команды — по одной команде для каждой задействованной строки.

Рассмотрим транзакцию, обновляющую 100 строк на стороне издателя (это может быть инструкция удаления, обновления или вставки). В журнале она будет зарегистрирована как

единая транзакция. После этого агент читает эту транзакцию и создает для нее запись в таблице msrepl_transactions базы данных распространения. Затем он разбивает данную транзакцию на одиночные команды, которые записывает в таблицу msrepl_commands базы распространения (о причинах такой последовательности действий вы узнаете в следующем разделе). Если инструкция достаточно большая, она может быть записана в несколько строк базы данных распространения. Все команды сохраняются в двоичном формате, но их можно прочитать с помощью хранимой процедуры sp_browsereplcmds, которую можно найти в базе распространения. По умолчанию команды, которые репликация транзакций использует для синхронизации подписчика с издателем, оформляются в виде хранимых процедур. Кроме того, у вас есть возможность хранить команды в виде отдельных инструкций SQL DML (INSERT, UPDATE и DELETE), что крайне необходимо в гетерогенной среде репликации.



Одиночной называется инструкция, которая оказывает воздействие на одну строку.

Когда агент чтения записал информацию в базы данных распространения, он помещает в журнал транзакций маркеры, указывающие на регистрацию данных транзакций для распространения. Отмеченные этими маркерами фрагменты журнала транзакций могут быть перераспределены, если вы зарезервируете журнал при полной модели восстановления или будете использовать простую модель восстановления. Заметим, что репликация транзакций работает с любой моделью восстановления.

Агент распространения

Этот агент (Distribution Agent) распространяет команды из базы данных распространения среди подписчиков. Также он удаляет из этой базы данных те команды, которые были распространены всем подписчикам и срок хранения которых вышел за пределы величины, установленной в базе.

Процесс репликации транзакций разбивает транзакции на одиночные команды, поэтому всегда можно подсчитать, сколько строк было обновлено у подписчика в результате распространения операций агентом. Если одиночная команда не оказала влияния ни на одну строку или повлияла на несколько строк (т.е. более одной), то процесс репликации завершается и агент распространения уведомляется о наличии проблемы целостности данных. Этот процесс верификации обеспечивает целостное состояние баз данных издателя и подписчика. Другими словами, подписчик будет всегда иметь те же данные, что и издатель. Агент распространения применяет все одиночные команды (формирующие исходную транзакцию) в контексте транзакции. Если хотя бы одна из них завершается неудачей, то все одиночные команды, составляющие транзакцию, откатываются. Это гарантирует то, что репликация транзакций всегда будет выполняться на уровне целостных транзакций.

Одноранговая репликация

Этот тип репликации был введен в версии SQL Server 2005 и является одной из вариаций репликации транзакций. Все узлы, участвующие в топологии одноранговой репликации, одновременно являются подписчиками и издателями друг друга. Транзакция, выполненная на одном узле, будет реплицирована на все остальные, кроме самого издателя. Эта модель, как правило, используется в приложениях, использующих для масштабирования множество баз данных и серверов. Клиенты могут подключаться к любой из множества баз данных и гарантированно работать с одной и той же информацией. Если из общего банка будет выведен

один из серверов (или база данных), его нагрузка будет равномерно распределена между оставшимися серверами (или базами), участвующими в топологии одноранговой репликации. Все узлы одноранговой репликации содержат идентичные копии данных.



Одноранговая репликация решает задачу, в которой множество распределенных серверов должно оставаться постоянно обновленным.

Двусторонняя репликация транзакций

Двустороннюю репликацию транзакций лучше всего использовать, когда существует возможность разделения объектов, участвующих в решении, для минимизации конфликтов или когда одна из сторон репликации доступна для записи в любой момент времени. В двусторонней репликации участвуют только два узла: один издатель и один подписчик.

Репликация транзакций с непосредственным обновлением

Репликация транзакций с непосредственным обновлением аналогична репликации снимков базы данных с непосредственным обновлением. Однако в данном случае только изменения, произошедшие на стороне издателя, реплицируются у подписчика. В этом процессе участвует исключительно агент распространения. Изменения, происходящие на стороне подписчика, реплицируются издателю с помощью координатора распределенных транзакций (DTC). Если вы выбрали этот вариант репликации, то должны обеспечить надежную связь между издателем и подписчиком. Если связь обрывается или издатель становится недоступным по другим причинам, транзакции, выполненные у подписчика, выждут около 20 секунд, после чего будет выполнен их откат.

Репликация транзакций с очередью обновлений

Репликация транзакций с очередью обновлений аналогична репликации снимков базы данных с очередью обновлений. Однако в данном случае только изменения, произошедшие на стороне издателя, реплицируются у подписчика. В этом процессе участвует исключительно агент распространения. Все изменения, происходящие на стороне подписчика, реплицируются у издателя с помощью агента чтения очереди. Этот тип репликации лучше всего себя проявляет при выполнении следующих условий.

- Количество изменений, выполняемых на стороне подписчика, составляет малую долю всего множества изменений.
- Количество подписчиков не превосходит десяти.

Если хотя бы одно из этих условий не удовлетворяется, данная модель репликации не может быть названа оптимальной, поскольку в процессе работы агента чтения очереди будут возникать частые взаимоблокировки.

Репликация транзакций с непосредственным обновлением и очередью восстановления

Репликация транзакций с непосредственным обновлением оказывается полезной при выполнении следующих условий.

- Подавляющая масса транзакций происходит у издателя.
- Некоторые транзакции порождаются у подписчика, и возникает потребность в их репликации издателю.
- Число подписчиков невелико.
- Необходимо иметь возможность перевода издателя в автономный режим.

В этом случае лучше использовать непосредственное обновление с постановкой в очередь отказов. При использовании такой схемы можно вручную переключаться между немедленным обновлением и использованием очереди обновлений, вызывая хранимую процедуру `sp_setreplfailovermode`. В случае переключения в режим использования очереди используется параметр `queued`, при обратном переключении — `immediate`. Режим немедленного обновления снова включается, когда издатель становится доступным в режиме реального времени.

Репликация транзакций через Интернет

Компания Microsoft рекомендует при выполнении репликаций через Интернет использовать виртуальную частную сеть, защищая тем самым свое подключение. Если этого не сделать, скорее всего, вам придется использовать подписку по запросу с анонимными подписчиками и загружать снимок базы данных на сервер FTP. В этом случае всегда существует вероятность того, что хакеры получат доступ к снимку базы и, возможно, даже нарушат работу сервера.



Виртуальные частные сети (VPN) являются важным подспорьем в бизнесе, в котором приходится передавать данные на большие расстояния. Присутствие Интернета постоянно растет даже в самых отдаленных регионах, что увеличивает отдачу от использования VPN. Если вам интересно узнать подробности о внутренней работе виртуальных частных сетей, посетите сайт

`http://computer.howstuffworks.com/vpn.htm`

Репликация слияния

Репликация слияния предназначена в основном для клиентов с мобильными устройствами и тех, кому часто приходится выходить в автономный режим. Изменения могут выполняться как на стороне издателя, так и на стороне подписчика. Когда запускается агент слияния (Merge Agent), эти изменения синхронизируются, после чего подписчик и издатель становятся обладателями одного и того же множества данных. В репликации слияния используются триггеры во всех таблицах, участвующих в репликации. Использование идентификаторов GUID в столбцах позволяет уникально идентифицировать все строки во всех реплицируемых таблицах. Когда изменения затрагивают одну из реплицируемых таблиц, они записываются также в таблицу метаданных, содержащую список идентификаторов GUID, соответствующих измененным строкам реплицируемых таблиц. Когда запускается агент слияния, он извлекает все идентификаторы из таблиц метаданных, соответствующие строкам, измененным на стороне издателя и подписчика. Если некоторая строка была модифицирована только на одной

из сторон, она извлекается из таблицы, породившей изменения, после чего с помощью инструкций DML выполняются соответствующие вставки, обновления или удаления на противоположной стороне репликации. Если же некоторая строка была изменена на обеих сторонах репликации, то агент слияния трактует сложившуюся ситуацию как конфликт и принимает решения, основываясь на правилах, заложенных при конфигурировании агента. По умолчанию в любом конфликте преимущество отдается издателю. Чтобы изменить методику разрешения конфликтов, вы можете выбрать соответствующее правило при создании публикации. При этом используется мастер новой публикации (New Publication Wizard).

1. Щелкните правой кнопкой мыши на публикации, которую хотите изменить, и выберите в контекстном меню пункт Properties.
2. Выберите папку `Articles`.
3. Щелкните на кнопке Article Properties, установите переключатель в положение Set properties of Highlighted Table article или Set Properties of All Table articles, после чего перейдите на вкладку Resolver (рис. 39.1).

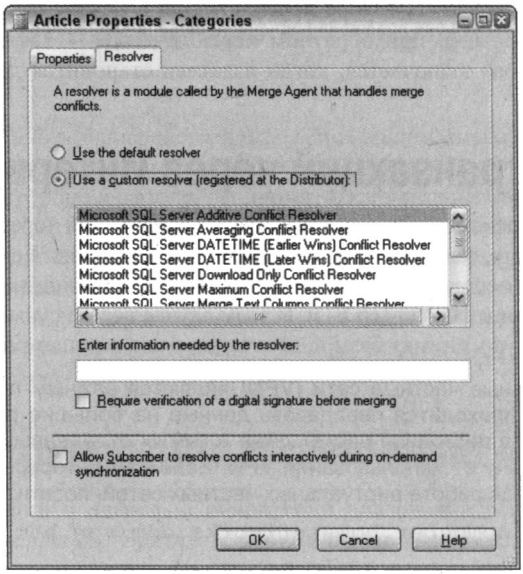


Рис. 39.1. Выбор типа разрешения конфликтов в репликации слияния



Для просмотра метаданных столбцов таблиц, публикуемых репликацией, выберите `sysdm_repl_schemas` в динамическом представлении управления.

По умолчанию выбран пункт Use Default Resolver. Он подразумевает бесспорное преимущество издателя или подписчика, имеющего наивысший приоритет. Доступны также следующие варианты.

- Microsoft SQL Server Additive Coflict Resolver. Значения столбца, определенного в текстовом поле Enter information needed by resolver, суммируются для конфликтующих строк.

- Microsoft SQL Server Averaging Coflict Resolver. Значения столбца, определенного в текстовом поле Enter information needed by resolver, усредняются для конфликтующих строк.
- Microsoft SQL Server DATETIME (Earlier Wins) Coflict Resolver. Строка с самой ранней версией значения столбца, определенного в текстовом поле Enter information needed by resolver, выигрывает в конфликте.
- Microsoft SQL Server DATETIME (Later Wins) Coflict Resolver. Строка с самой поздней версией значения столбца, определенного в текстовом поле Enter information needed by resolver, выигрывает в конфликте.
- Microsoft SQL Server Download Only Conflict Resolver. В конфликте выигрывают данные, загруженные подписчиком.
- Microsoft SQL Server Maximum Coflict Resolver. Строка с самым большим значением столбца, определенного в текстовом поле Enter information needed by resolver, выигрывает в конфликте.
- Microsoft SQL Server Merge Text Columns Conflict Resolver. Данные в текстовом столбце, определенном в текстовом поле Enter information needed by resolver, выигрывают в конфликте.
- Microsoft SQL Server Maximum Coflict Resolver. Строка с самым маленьким значением столбца, определенного в текстовом поле Enter information needed by resolver, выигрывает в конфликте.
- Microsoft SQL Server Priority Conflict Resolver. В конфликте выигрывает строка с наибольшим приоритетом.
- Microsoft SQL Server Subscriber Always Wins Conflict Resolver. В конфликте всегда побеждает подписчик.
- Microsoft SQL Server Upload Only Conflict Resolver. В конфликте всегда побеждает строка, загруженная от подписчика.
- Microsoft SQL Server Stored Procedure Conflict Resolver. Разрешением конфликта управляет хранимая процедура, определенная в текстовом поле Enter information needed by resolver.

Вы имеете также два дополнительных варианта.

- Создать собственную систему разрешения конфликтов на основе хранимых процедур или компонентов COM.
- Использовать расширенную логику, определенную в пространстве имен `Microsoft.SqlServer.Replication.BusinessLogicSupport` объектной модели RMO.

Вы можете также выполнять мониторинг на уровне столбцов. В этом случае одновременное изменение одним узлом адреса, а другим номера телефона клиента в одной и той же строке, позволяет им слить свои данные, не переписав изменения друг друга. Таким образом, в приведенном примере в базах данных и издателя, и подписчиков будет как новый адрес, так и новый телефон. Доступен также и вариант интерактивного разрешения конфликтов. Этот вариант доступен, когда в репликации слияния с публикацией по запросу используется Windows Synchronization Manager и публикация открыта для интерактивного разрешения конфликтов. В этом случае при обнаружении конфликта запускается соответствующий мастер, и пользователь компьютера подписчика имеет возможность как самостоятельно разрешить конфликт, так и выбрать вариант, предлагаемый системой по умолчанию.

Репликация слияния и подписчики SQL CE и SQL Mobile

SQL Server 2005 позволяет пользователям мобильных систем SQL CE и SQL Mobile подписываться на публикации репликаций слияния. Пользователи SQL Mobile могут открыть множество подписок на разные публикации, в то время как пользователям SQL CE предоставляется возможность подписаться только на одну публикацию каждой из баз данных SQL CE.

Репликация слияния через Интернет

Новой в SQL Server 2005 является возможность выполнения репликаций слияния через защищенные подключения Интернета. При этом для синхронизации используется протокол THHPS и процесс, называемый *Web-синхронизацией*. У вас остается возможность размещать подписки и на серверах FTP, однако этот вариант не считается безопасным.

Нововведения в репликациях SQL Server 2005

В следующих разделах будут описаны все нововведения, появившиеся в репликациях SQL Server 2005.

Продолжение репликаций снимков базы данных

В предыдущих версиях SQL Server, если в ходе репликации снимков базы данных возникала ошибка, даже в последних строках последней таблицы, репликация начиналась с самого начала. В SQL Server 2005 при перезапуске агента распространения будут реплицированы только недостающие части снимка базы данных.

Публикации Oracle

SQL Server 2005 позволяет сконфигурировать СУБД Oracle для публикаций подписчикам и распространителям SQL Server 2005. Для этого мастер публикаций помещает в таблицы Oracle соответствующие триггеры. Эти триггеры отслеживают изменения, после чего посылают их подписчикам SQL Server. Сервер Oracle можно активизировать для репликаций снимков базы данных и транзакций. Сервером Oracle не поддерживается репликация слияния. Публикация развертывается с помощью сценария, запускаемого на сервере издателя Oracle, что позволяет удалить на нем все объекты репликации и метаданные.

Повышенная безопасность

SQL Server 2005 с самого начала разрабатывался как система с повышенной безопасностью. Нигде повышенная защищенность не ощущается так, как в репликациях. Ранее все типы репликации использовали общий каталог `Admin`, из которого загружались снимки баз данных. В SQL Server 2005 он был заменен локальным каталогом (например, `\Program Files\Microsoft SQL Server\MSSQL.1\Repldata`), однако у вас есть возможность сконфигурировать этот каталог снимков для доступа подписок по запросу тех пользователей,

у которых нет обычных разрешений доступа к данному локальному каталогу. Все агенты репликации могут запускаться под учетной записью, отличной от используемой агентом SQL Server Agent. Это позволяет назначить агентам репликации более низкий уровень разрешений, чем было возможно в SQL Server 2000. Теперь подписчики репликации слияния могут подключаться к Web-серверу и синхронизироваться с ним с помощью защищенного протокола HTTPS. Этот вариант более защищенный, чем те, которые были доступны в SQL Server 2000.

Одноранговая модель репликации

Как уже говорилось ранее, одноранговая топология репликации транзакций позволяет масштабировать решения с высоким уровнем доступности. Пользователям становится доступной модель "записывай, куда хочешь". Вместо того, чтобы тысяча пользователей одновременно подключалась к одному серверу, выступающему своеобразной "рабочей лошадкой", теперь можно иметь десять серверов с идентичными данными, к каждому из которых будет подключаться не больше ста пользователей. Уменьшение нагрузки выливается в повышение производительности, при этом каждый из серверов баз данных предъявляет более низкие требования к оборудованию, чем единственный сервер. В одноранговой модели все изменения в данных на одном узле распространяются на все остальные узлы. Таким образом, все узлы, участвующие в одноранговой репликации, имеют одни и те же данные. При использовании этой топологии следует минимизировать конфликты за счет использования разделения или разрешения операций записи только на одном узле. Если этого не сделать, агент распространения не справится со своей работой, и все узлы постепенно выйдут из синхронизации.

Чтобы включить репликацию транзакций для участия в одноранговой топологии, создается обычная публикация транзакций (но без непосредственных обновлений). После создания щелкните правой кнопкой мыши на публикации, выберите в контекстном меню пункт Subscription Options и установите для параметра Allow Peer to Peer Subscriptions значение `true`. Разверните подписку у всех подписчиков. После того как все подписки будут развернуты у всех подписчиков, щелкните правой кнопкой мыши на публикации и запустите мастер конфигурации одноранговой репликации.

Репликация инструкций DDL

Теперь вы можете при желании реплицировать все инструкции DDL, в том числе `ALTER TABLE`, `ALTER VIEW`, `ALTER PROCEDURE`, `ALTER FUNCTION` и `ALTER TRIGGER`.

В предыдущих версиях SQL Server, чтобы внести изменения в схему, часто приходилось удалять публикации, хотя SQL Server 2000 и позволял внести некоторые изменения с помощью хранимых процедур репликации `sp_repladdcolumn` и `sp_repldropcolumn`.

Также у вас есть возможность отключить репликацию изменений схемы, щелкнув правой кнопкой мыши на публикации и выбрав в контекстном меню пункт Subscription Options. Установите для параметра Replication Schema Properties значение `false`. Этот параметр доступен во всех типах репликаций.

Репликация полнотекстовых индексов

SQL Server 2005 позволяет реплицировать полнотекстовые индексы. Для этого нужно включить в базе данных подписчика полнотекстовую индексацию и указать это свойство в параметрах статьи. В диалоговом окне статьи щелкните на кнопке Article Properties, установите переключатель в положение Set Prtoperties of Highlited Table Article или Set Properties of All Table Articles и установите для параметра Copy Full-Text Indexes значение `True`.

Разрешение анонимных подписок на все публикации

В SQL Server 2000 существовало два типа подписчиков: именованные и анонимные. Именовалось обычно малое число подписчиков. Метаданные репликаций хранились для всех именованных подписчиков и удалялись при развертывании. Метаданные репликаций не отслеживались для анонимных подписчиков и обычно были одинаковыми для каждого из них. Публикации должны были создаваться отдельно для именованных и анонимных подписчиков. В SQL Server 2005 все публикации могут быть развернуты как у именованных, так и у анонимных подписчиков.

Логические записи в репликации слияния

В SQL Server 2000 обновления, выполняемые в таблицах, связанных с DRI и участвующих в репликациях слияния, могли применяться у издателя и подписчика безо всякого порядка. Например, дочерние записи у издателя могли появиться раньше родительских, что в некоторых случаях приводило к завершению работы агента слияния и необходимости его повторного запуска. В SQL Server 2005 транзакции реплицируются как логические единицы, называемые *логическими записями*, в которых родительские записи реплицируются раньше дочерних.

Предварительно вычисленные разделы

Предварительно вычисленные разделы — это новое средство репликации слияния в SQL Server 2005. В SQL Server 2000 при запуске агента слияния он должен был определить, какие из измененных строк должны быть слиты с конкретным подписчиком. Этот процесс существенно снижал производительность. В SQL Server 2005 стало возможным включать предварительное вычисление разделов, т.е. выборок строк, которые должны быть отправлены определенному подписчику при запуске агента слияния. Вследствие этого агенты слияния стали запускаться быстрее, с меньшим влиянием на производительность.

Обновление уникальных ключей

В SQL Server 2000 обновление столбца первичного ключа сегментировалось на операции удаления и вставки. Это приводило к проблемам при выполнении каскадных удалений в таблицах. В SQL Server 2005 обновление столбца первичного ключа реплицируется как инструкция `UPDATE`, что существенно повышает производительность.

Пользовательская обработка конфликтов с помощью RMO

В версии SQL Server 2000 для разрешения конфликтов с помощью усложненной логики приходилось создавать объекты COM или хранимые процедуры. В SQL Server 2005 сложную логику можно реализовать с помощью пространства имен `Microsoft.SqlServer.Replication.BusinessLogicSupport` объектной модели RMO. Эту объектную модель также можно использовать для обработки ошибок и изменений данных.

Многочисленные улучшения производительности

Снимки базы данных теперь генерируются по умолчанию с параметром конкурентности для репликации транзакций. Вследствие этого сокращается время блокировки при генерации снимка. В SQL Server 2000 при генерировании снимка все реплицируемые таблицы блокировались. Теперь агенты чувствительны к состоянию сети и переподключаются, когда соединение становится доступным. Сбои в работе агентов в SQL Server 2000, связанные с сетевыми отказами, требовали вмешательства администратора базы данных для их перезапуска, когда сетевое соединение восстанавливалось. Также была улучшена поддержка особо крупных объектов в подписках слияния и с обновлением.

Маркеры трассировки

Один из наиболее часто задаваемых вопросов, связанных с репликациями, звучал следующим образом: "Насколько быстро после применения транзакции у издателя она применяется у подписчика?" Другими словами, на сколько отстает подписчик от издателя. В мониторе репликаций SQL Server 2005 имеется специальная вкладка Trace Tokens. Если щелкнуть на кнопке Insert Tracer в этой вкладке, у издателя будет вставлена запись, позволяющая узнать, сколько времени прошло до применения изменения в записи к подписчику (рис. 39.2).

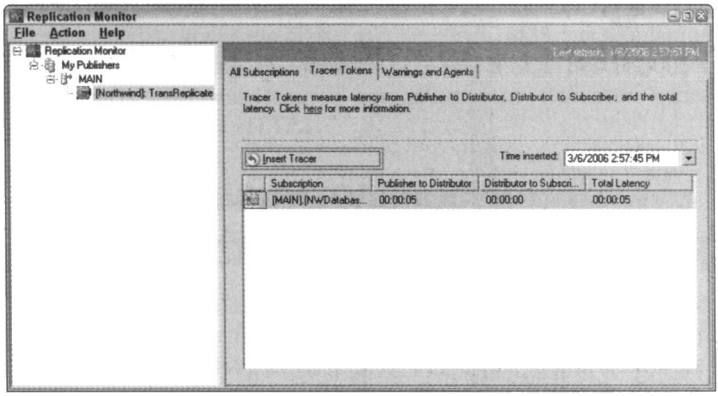


Рис. 39.2. Используйте маркеры для мониторинга задержек обновления между издателем и подписчиком

Распараллеливание транзакций

В SQL Server 2000 все публикации транзакций одной базы данных конкретному подписчику использовали один и тот же агент распространения. В именованных подписках можно было включить несколько агентов распространения, называемых *независимыми*. В этом случае можно было создать два или более параллельных потока данных, отправляемых подписчику. Независимые агенты в SQL Server 2005 создаются по умолчанию, что значительно повышает производительность распространения транзакций.

Загрузка только статей

В SQL Server 2005 появилась новая возможность определять, какие из статей предназначены "только для загрузки" (т.е. эти данные перемещаются только от издателя к подписчику).

Если изменения выполняются в помеченной таким образом таблице и публикуются для репликации слияния, то изменения, внесенные на стороне подписчика, не сливаются с изменениями издателя и остаются только у подписчика. Изменения, выполненные на стороне издателя, сливаются с подписчиком.

Монитор репликаций

Монитор репликаций был полностью переписан и теперь стал центральным пультом управления репликациями. Монитор репликаций больше не интегрирован в другие утилиты — теперь он представляет собой обособленную консоль управления. Запустить монитор репликаций можно из папки `Program Files\Microsoft SQL Server\90\Tools\Binn`, введя имя файла **`sqlmonitor.exe`**. В качестве альтернативы его можно запустить и из интерфейса Management Studio, щелкнув правой кнопкой мыши на узле `Replication` нужного сервера баз данных и выбрав в контекстном меню пункт Launch Replication Monitor. Монитор репликаций (рис. 39.3) очень похож на группу Replication Monitor Group, присутствовавшую в SQL Server 2000. Он позволяет быстро получить оценку состояния всех экземпляров SQL Server, которые выбраны для мониторинга.

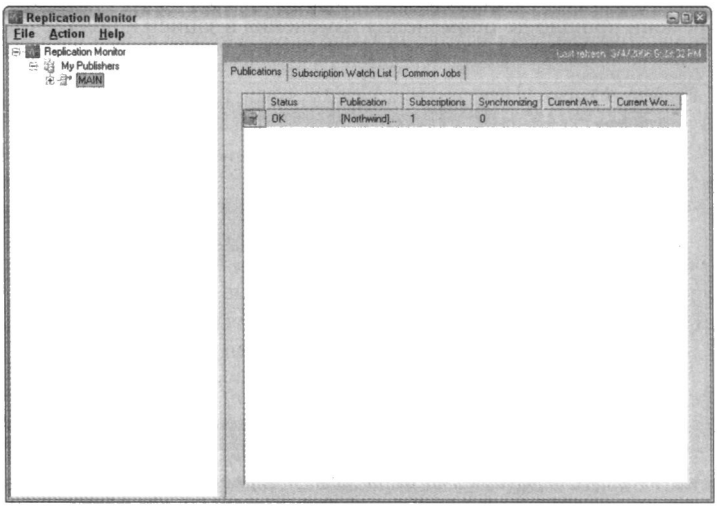


Рис. 39.3. В целях упрощения работы монитор репликаций был перепроектирован

На рис. 39.3 показано, как выполняется мониторинг экземпляра SQL Server 2005, который называется `Main`. Ниже приведен список некоторых улучшений, выполненных в мониторе репликаций.

- **Улучшенная метрика**. В предыдущих версиях SQL Server отображались три состояния заданий репликации: успешно выполненное (`successful`), не выполненное (`failed`) и запущенное повторно (`retrying`). В SQL Server 2005 в метрику добавлен символ предупреждения, отражающий состояние проблем производительности.
- **Минимизация влияния мониторинга на производительность**. В предыдущих версиях SQL Server мониторинг мог при значительной нагрузке привести к блокировке самих репликаций. В связи с этим монитор репликаций был перестроен с нуля, чтобы минимизировать влияние мониторинга на общую производительность.

- **Улучшенный мониторинг репликаций слияния**. Монитор репликаций теперь отслеживает статистику на уровне статей репликаций слияния, чтобы помочь администратору быстро выявить проблемные статьи.
- **Введены маркеры трассировки для измерения задержек в репликациях транзакций**. Эти маркеры позволяют точно отслеживать производительность репликаций транзакций и оценивать их продолжительность.
- **Предупреждения по истечении срока подписки**. Этой возможности не было в SQL Server 2000.

Репликации слияния по протоколу HTTPS

В предыдущих версиях SQL Server невозможно было настроить репликацию через Интернет. Пользователю приходилось использовать виртуальные частные сети или серверы FTP. Виртуальные сети сильно замедляли производительность, и если не использовался клиент VPN от Microsoft (для его запуска нужно было выбрать в меню Start пункт Settings⇨Network and Dial-Up Connections⇨Make a New Connection⇨Connect to a prinate network through the Internet), то достаточно дорого обходились пользователю. Использование серверов FTP считается незащищенным решением, так как пароль FTP передается в Интернете в незашифрованном (т.е. текстовом) виде, если не используется анонимный доступ (в последнем случае загрузить снимок базы может вообще любой желающий). Подключение через Интернет также подразумевает открытие в брандмауэре порта 1433 для входящих подключений, что также подвергает риску систему. Репликация слияния в SQL Server 2005 позволяет сконфигурировать Web-сервер, обеспечивающий синхронизацию клиентов через Интернет посредством защищенного протокола HTTPS.

Повышение производительности и масштабируемости репликаций слияния

SQL Server 2005 предлагает множество улучшений, связанных с повышением производительности и масштабируемости репликаций слияния. Производительность простой репликации слияния повысилась примерно на 20%, однако ее можно удвоить, если использовать преимущества предварительно скомпилированной фильтрации. Статьи, доступные только для загрузки, повышают производительность примерно в пять раз.

SQL RMO

В SQL Server 2000 можно было создавать репликации с использованием хранимых процедур репликации и объектной модели SQL DMO. В SQL Server 2005 была добавлена специальная объектная модель RMO — именно она в настоящее время считается наиболее предпочтительным методом создания объектов репликации (по сравнению с SQL DMO). Модель SQL RMO можно использовать в среде .NET, к тому же она создает меньшую дополнительную нагрузку, чем SQL DMO и хранимые процедуры репликации.

Упрощенные мастера

Мастера репликаций были в значительной мере упрощены. В SQL Server 2000 для развертывания простейшей публикации у подписчика приходилось выполнять около 52 щелчков мышью. В SQL Server 2005 мастера репликаций имеют ряд преимуществ.

- Они содержат примерно на 40% меньше страниц.
- Значительно упрощены диалоговые окна.
- Улучшены автоматически определяемые значения и значения, принятые по умолчанию.
- Уменьшено количество уровней ветвления.
- Все мастера предлагают записать выполненные действия в виде сценария или применить их немедленно.
- Мастера подписок по требованию и принудительных подписок были консолидированы в единый мастер. Мастер новых подписок может одновременно создать множество подписок с разными свойствами публикаций. Это сильно упрощает настройку репликации в условиях небольших реализаций, так как все подписки можно создать, только единожды запустив мастер. В условиях крупных реализаций для упрощения администрирования придется все равно генерировать сценарии.

Инициализация подписчика

В предыдущих версиях SQL Server приходилось развертывать снимок базы данных с помощью запуска агента снимка для генерации схемы и данных в файловой системе, а затем запускать агент распространения или слияния и воссоздавать реплицируемые объекты у подписчика. Также существовала возможность предварительно создать базу данных подписки вручную, после чего выполнить несинхронную подписку. Ручное создание подписки предполагает большой объем работы по настройке ограничений и столбцов идентичности с параметром `NOT FOR REPLICATION` и замещения их с помощью соответствующих хранимых процедур репликации. Запуск агента снимков каждый раз и отправка файлов данных посредством агентов слияния или распространения отнимало много времени, особенно когда объем снимка был большим.

В SQL Server 2005 существует ряд дополнительных вариантов перемещения данных базы.

Резервирование и восстановление

Вы можете выполнить стандартные процедуры резервирования и восстановления базы данных для создания базы данных подписки на стороне подписчика. При развертывании подписчика используйте метод его инициализации.

Копирование базы данных

Вы можете также вручную создать все объекты на стороне подписчика с помощью утилиты массового копирования (`bcp`), службы преобразования данных (DTS) и прочих методов синхронизации данных. Однако вам потребуется проделать солидный объем работ по добавлению предложения `NOT FOR REPLICATION` в столбцы идентичности и ограничения внешних ключей; также вам придется модифицировать хранимые процедуры репликации.

Динамические снимки базы данных

В репликации слияния можно использовать динамические снимки базы данных. Они позволяют фильтровать снимки, основываясь на некотором критерии. Это позволяет отправлять подписчику только те данные, в которых он нуждается, а не весь снимок.

Конфигурирование репликаций

Для конфигурирования репликации в SQL Server 2005 подключитесь к вашему серверу баз данных в Management Studio, щелкните правой кнопкой мыши на папке Replication и выбе-

рите в контекстном меню пункт Configure Distribution. Будет запущен мастер конфигурирования распространения (Configure Distribution Wizard). Щелкните на кнопке Next, чтобы перейти к диалоговому окну Distributor. В этом окне вы можете сконфигурировать SQL Server для использования локального или удаленного распространения.

Использование локального распространителя

Существует простое правило, регламентирующее, когда стоит использовать локальный, а когда удаленный распространитель. При использовании локального распространителя один сервер совмещает роли издателя и распространителя. Удаленный распространитель располагается на отдельном сервере. Принятие решения относительно перехода к удаленному распространителю следует принимать на основании совместной работы агентов чтения журнала и распространения. Если они практически блокируют систему, значит, переход будет полезным.

Для конфигурирования SQL Server на использование локального распространителя выберите в окне мастера конфигурирования распространителя параметр, предложенный по умолчанию, — *Имя_вашего_сервера* will act as your own Distributor — и щелкните на кнопке Next. Если вы используете принудительные подписки, примите предложенный по умолчанию каталог снимков. Если вы используете подписки по требованию, установите общий доступ к папке `C:\Program Files\Microsoft SQL Server\MSSQL.X\MSSQL\ReplData` (где *X* — имя экземпляра SQL Server 2005) и введите в текстовом поле Snapshot Folder путь `\\имя_сервера_издателя\имя_общей_папки` (где *имя_общей_папки* — имя, присвоенное общему ресурсу `C:\Program Files\Microsoft SQL Server\MSSQL.X\MSSQL\ReplData`). Не забудьте предоставить учетной записи, под которой будет запускаться агент распространения в подписках по запросу, право чтения папки снимков. В принципе, общий ресурс вы можете создать и по завершении работы мастера. Щелкните на кнопке Next и примите все параметры, предложенные по умолчанию (из соображений производительности лучше поместить папку снимков на дисковый массив RAID 10). Два раза щелкните на кнопках Next, чтобы включить использование сервером базы данных распространения, еще раз щелкните на кнопке Next, чтобы сконфигурировать текущий сервер в качестве распространителя (или сгенерировать сценарий, который будет использован позже), и еще раз щелкните на кнопке Next, чтобы внесенные изменения вступили в силу. Щелкните на кнопке Finish, чтобы завершить работу мастера.

Локальный распространитель может быть использован другими подписчиками, сконфигурированными соответствующим образом. Эти подписчики смогут использовать базу распространения по умолчанию на локальном распространителе или создать для себя собственного распространителя. Издатель может использовать только одну базу данных распространения и может быть сконфигурирован для использования только одного распространителя.

Использование удаленного распространителя

Если ваш сервер баз данных подвергается сильной нагрузке, подумайте об использовании удаленного распространителя. Если вы решили поступить именно так, то должны сделать следующее:

- сконфигурировать удаленный сервер как распространителя;
- сконфигурировать издателя для использования этого распространителя (подключиться к нему, открыть окно Distribution Properties, перейти на вкладку Publishers и добавить издателя с помощью кнопки Add, при этом ввести пароль администратора);
- знать пароль администратора для подключения к серверу.

В идеальном случае удаленный распространитель должен находиться в той же локальной сети и быть кластеризованным ввиду того, что в автономном режиме журнал транзакций базы данных распространения может достичь гигантских размеров. Если вы остановились на использовании удаленного распространителя, щелкните на кнопке Add, подключитесь к распространителю, введите пароль администратора, установите флажок Configure Distributor (по умолчанию он установлен), щелкните на кнопке Next, а затем на кнопке Finish.

Создание публикаций репликации снимков базы данных

Для создания публикаций репликации снимков баз данных подключитесь к своему серверу SDQL Server в Management Studio, разверните папку публикаций, щелкните правой кнопкой мыши на папке `Local Publications` и выберите в контекстном меню пункт New Publication. В открывшемся мастере новых публикаций щелкните на кнопке Next. Выберите базу данных, которую хотите реплицировать, и снова щелкните на кнопке Next. Выберите установленный по умолчанию тип публикации — Snapshot Publication — и щелкните на кнопке Next. В открывшемся диалоговом окне выберите типы объектов, щелкая на них. При желании вы можете разворачивать папки типов объектов и выбирать отдельные объекты (например, щелкнуть на типе Tables и выбрать для репликации отдельные таблицы).

Если вы углубились до уровня отдельных столбцов, как на рис. 39.4, можете снять выделение с них.

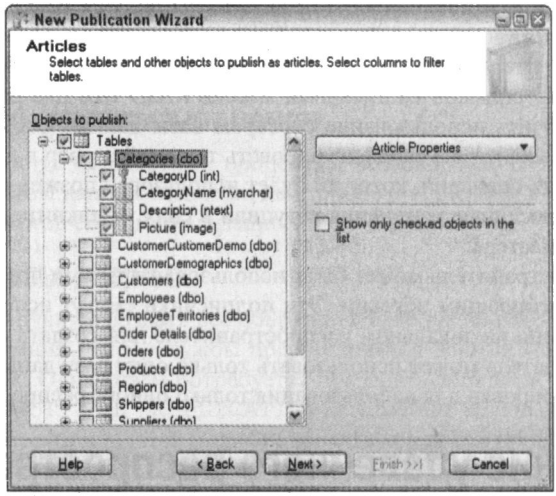


Рис. 39.4. У вас всегда есть возможность выбрать статью и помещаемые в нее столбцы

Вы можете определить, какие столбцы не должны реплицироваться подписчикам, снимая с них выделение. Этот процесс называют *вертикальной фильтрацией* или *вертикальным разделением*. На рис. 39.4 выделены все столбцы — эта установка используется по умолчанию.

Щелкнув на кнопке Article Properties, вы можете изменить параметры статьи (например, имя владельца и даже имя статьи, какой она будет представлена подписчику).

После выбора реплицируемых объектов щелкните на кнопке Next.



Может открыться диалоговое окно Article Issues с предупреждениями о проблемах, которые могут возникнуть при развертывании подписки. Например, если вы публикуете представления или хранимые процедуры, то можете получить предупреждение о том, что следует создать для них базовые объекты в базах данных подписчиков. При репликации индексированных представлений вы можете получить предупреждение о том, что данный тип объектов поддерживается только версией SQL Server 2000 Enterprise Edition и более поздними.

Щелкните на кнопке Next, чтобы открыть диалоговое окно Filter Table Rows. В этом диалоговом окне вам предоставляется возможность установить в реплицируемых таблицах и представлениях *горизонтальную фильтрацию* (или *горизонтальное разделение*). Вы можете принять решение реплицировать только часть строк — например, имеющих отношение к определенному региону страны. Для этого вы должны щелкнуть на кнопке Add и выбрать таблицу или статью, подлежащую фильтрации. После этого следует щелкнуть на столбце, который будет служить основой для фильтрации. В фильтре вы можете использовать множество столбцов, разделяя условия булевыми операторами. Например, условие фильтра может выглядеть следующим образом:

```
SELECT <published_columns> FROM [Person].[Address]
WHERE   [StateProvinceID]=5 and[ModifiedDate] > getdate() -365
```

В приведенном примере выполняется фильтрация клиентов, живущих в штате Калифорния (в предположении, что идентификатор этого штата равен пяти). Фильтр может также включать в себя системные функции, такие как `SUSER()`, `HOST_NAME()`, и системные переменные, такие как `@@ServerName`. Следует обратить внимание, что функция `HOST_NAME()` разрешает имена узлов в пределах одного кластера и инициализирует соответствующий агент. Чтобы это имя разрешалось для подписчиков, следует использовать переменную `@@ServerName` и подписку по требованию. Если используется принудительная подписка, то будет разрешаться имя издателя или распространителя.

Щелкните на кнопке Next, чтобы открыть окно Snapshot Agent. В этом окне вам будет предложено создать расписание, по которому будут генерироваться снимки базы данных. Например, можно настроить подписку так, чтобы снимки генерировались в нерабочее время, поскольку этот процесс является ресурсоемким и может отрицательно сказаться на производительности.



SQL Server 2005 предлагает большую гибкость снимков баз данных. По умолчанию первый снимок генерируется сразу же после завершения работы мастера, т.е. вам не придется ждать, пока данные станут доступными. Однако этот новый подход может повлиять на производительность, так что, если снимок вам не нужен немедленно, вы можете отложить его генерацию на более позднее время.

Щелкните на кнопке Next, чтобы открыть диалоговое окно системы безопасности. В этом окне вы должны указать, какую учетную запись следует использовать для генерации снимка. Здесь значения по умолчанию не предлагаются — выберите ту учетную запись, которую считаете нужной. Щелкните на кнопке Security Settings, чтобы открыть диалоговое окно, показанное на рис. 39.5.



Несмотря на то что может показаться, что для генерации снимка лучше использовать учетную запись агента SQL Server Agent, эта учетная запись по умолчанию предоставляет *слишком* много прав. Используйте ту учетную запись, которую специально сконфигурировали для агента снимков. В дополнение к сокращению потенциальных проблем защиты информации использование отдельной учетной записи позволяет осуществлять ее мониторинг, помогающий оперативно выявить нежелательную активность.

Вы можете запускать агент снимков под учетной записью Windows или SQL Server Agent (по умолчанию в SQL Server 2000 предлагалась именно она). Вы можете выбрать учетную

запись, под которой будет запускаться агент снимков при подключении к издателю. В SQL Server 2000 ею должна была быть запись системного администратора. Некоторые администраторы считают, что это предоставляет агенту снимков слишком много возможностей, поэтому в версии SQL Server 2005 вы имеете возможность выбора для него учетной записи с более низкими привилегиями.

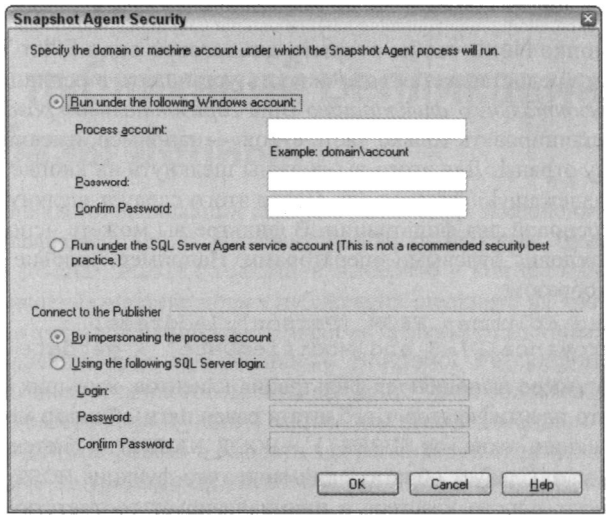


Рис. 39.5. Для настройки учетной записи, используемой при репликации, используйте это диалоговое окно

Рекомендуется использовать учетную запись Windows, так как она имеет более высокий уровень шифрования, чем доступный при использовании учетной записи SQL Server.

Если вы решили генерировать сценарий, то в следующем диалоговом окне можете определить, куда поместить файл. Если вы не генерируете сценарий, откроется заключительное диалоговое окно, в котором публикации нужно присвоить имя. После этого окно индикатора будет информировать вас о ходе процесса создания подписки.

Теперь можете создать подписки на эту публикацию. Этот вопрос мы обсудим в разделе "Создание подписок".

Создание публикаций репликации транзакций

Создание публикаций репликации транзакций очень похоже на создание публикации снимков базы данных. Для этого в Management Studio подключитесь к SQL Server, разверните папку `Replication`, щелкните правой кнопкой мыши на папке `Local Publications` и выберите в контекстном меню пункт New Publicztion. В открывшемся окне мастера публикаций щелкните на кнопке Next. В следующем окне выберите базу данных, публикации которой требуется создать, и щелкните на кнопке Next. В открывшемся окне выбора типа публикации выберите тип Transactional Publication и щелкните на кнопке Next. В следующем окне развернется список типов публикуемых объектов, аналогичный показанному на рис. 39.4.

Выберите типы объектов, которые хотите включить в публикацию. Вы можете публиковать все объекты выбранного типа или развернуть папку типа и выбрать конкретные объекты, например отдельные таблицы. Далее вы можете установить вертикальную фильтрацию, сняв отметки с тех столбцов, которые не следует реплицировать подписчикам.

Чтобы изменить параметры статьи, щелкните на кнопке Article Properties. В открывшемся диалоговом окне (рис. 39.6) вы можете определить, как именно статьи будут реплицироваться подписчикам.

Здесь вы можете назначить другого владельца таблицы и имя подписчика, определить, что должно произойти, если реплицируемый объект уже существует у подписчика, а также выбрать способ доставки транзакций подписчику.

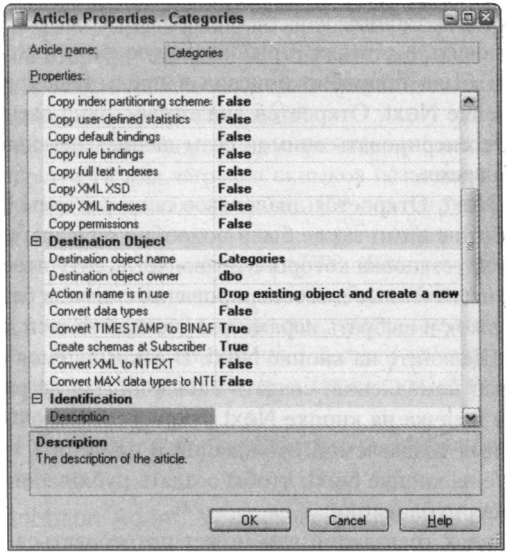


Рис. 39.6. Диалоговое окно параметров статьи

Если реплицируемый объект уже существует у подписчика, вы можете выполнить одно из действий:

- удалить существующий объект и создать новый (данный вариант предлагается по умолчанию);
- сохранить существующий объект неизмененным;
- удалить данные; если в статье установлен фильтр строк, удалить только данные, соответствующие фильтру;
- удалить все данные в существующем объекте.

В большинстве случаев используется первый вариант. В схемах с централизованным подписчиком подумайте об использовании третьего варианта. Если вам нужно сохранить архив, можно использовать второй вариант. Если нужно реплицировать данные в фиксированную схему, используйте последний вариант — удалите все данные в существующем объекте.

Вы можете также использовать сгенерированную системой хранимую процедуру с заданным системой именем или присвоить ей новое по собственному усмотрению. Также вы можете выбрать способ репликации транзакции: использовать хранимую процедуру в формате `CALL` (этот формат реплицирует все столбцы, независимо от того, изменены они или нет, — он доступен только для процедур вставки и удаления) или в формате `XCALL` (этот формат позволяет реплицировать только измененные столбцы — он доступен только в процедурах обновления); создать инструкции DML (`INSERT`, `UPDATE` или `DELETE`) без имен столбцов или с ними или выбрать вариант без репликации инструкций.

При репликации отдельных типов объектов доступны некоторые интересные параметры. Например, существует возможность репликации выполнения хранимой процедуры или репликации индексированного представления в таблицу на стороне подписчика.

После выбора состава реплицируемых объектов и способа репликации щелкните на кнопке Next. В отдельных случаях может быть открыто диалоговое окно Article Issues с предупреждениями системы. Просмотрите эти предупреждения и при необходимости приведите параметры статьи в соответствие с ними. Щелкните на кнопке Next, чтобы перейти к диалоговому окну фильтрации строк таблиц.

Если вы хотите установить в статьях горизонтальную фильтрацию, щелкните на кнопке Add. Фильтрация таблиц была подробно описана в предыдущем разделе. После создания фильтра щелкните на кнопке Next. Откроется диалоговое окно агента снимка базы данных. Выберите, хотите ли вы сгенерировать снимок базы данных немедленно или отложить этот процесс на более позднее время.

Щелкните на кнопке Next. Откроется диалоговое окно системы безопасности агента. Параметры этого окна и работа с ними также были подробно описаны в предыдущем разделе. В этом окне имеется флажок, установка которого позволяет агенту чтения журнала наследовать установки безопасности, назначенные для генерирования снимков базы данных. При желании вы можете снять этот флажок и выбрать параметры учетной записи для агента чтения журнала транзакций вручную. Щелкните на кнопке Next. В открывшемся окне определите, хотите ли вы создать публикацию немедленно, создать сценарий для ее развертывания в будущем или и то и другое. После щелчка на кнопке Next откроется заключительное окно мастера, в котором следует ввести имя создаваемой публикации и проверить правильность выбранных ранее настроек. Щелкните на кнопке Next, чтобы создать публикацию. Откроется диалоговое окно мониторинга создания публикации.

В некоторых публикациях транзакций вам может потребоваться изменить некоторые параметры, так как возможности изменить настройки в процессе создания публикации нет. Чтобы внести эти изменения, раскройте папку локальных публикаций и выберите ту, параметры которой хотите изменить. Щелкните на ней правой кнопкой мыши и в контекстном меню выберите пункт Properties. В открывшемся окне вы увидите семь разделов.

- General (Общие параметры).
- Articles (Параметры статьи).
- Filter rows (Параметры фильтрации строк).
- Snapshot (Параметры снимка базы данных).
- FTP Snapshot (Параметры подключения к серверу FTP).
- Publication Access List (Список доступа к публикации).
- Agent Security (Параметры безопасности агента).

Во вкладке General можно ввести описание публикации. Здесь можно также выбрать действия, которые следует предпринять, если подписчик находится в автономном режиме дольше определенного срока. В этом случае вы можете либо отменить подписку, либо считать ее устаревший (т.е. с истекшим сроком годности). В последнем случае вам придется заново создать и развернуть снимок базы данных. Если вы удаляете подписку, то должны будете воссоздать ее у соответствующего подписчика. Эти параметры являются очень важными, поскольку чем дольше подписчик находится в автономном режиме, тем больше метаданных репликации для него накапливается. Это может вызвать общее снижение производительности на сервере издателя, так что иногда выгоднее во всех отношениях заново создать снимок базы данных и развернуть его.

Во вкладке Article можно внести изменения в статьи, содержащие снимок базы данных. Здесь вы можете добавить или удалить статьи из публикации или изменить их параметры.

Во вкладке Filter Rows вы можете модифицировать существующие фильтры или добавить новые в статьи таблиц.

Во вкладке Snapshot можно определить, где будет размещаться снимок базы данных (в принятой по умолчанию или альтернативной папке, на FTP-сайте или и там, и там), следует ли сжимать снимок в альтернативной папке, а также сценарии, которые будут запускаться до и после развертывания снимка. Предваряющий сценарий можно запустить, чтобы создать нужные учетные записи на стороне подписчика или включить на сервере подписчика полнотекстовое индексирование. Завершающий сценарий пригодится для создания ограничений в таблицах на стороне подписчика и других операций.

Вкладка FTP Snapshot позволяет управлять параметрами, используемыми подписчиками при подключении к серверам FTP для загрузки снимков базы данных. Например, вы можете разрешить доступ по запросу к снимкам на сайте FTP для загрузки, определить, следует ли использовать анонимный доступ, а также указать, какая учетная запись NT может быть использована, если не используется аутентификация FTP. (Следует учесть, что система безопасности NT не применяется; эта учетная запись должна быть локальной, а не доменной; также пароль будет передаваться по Интернету в открытом, текстовом виде.) Здесь же можно задать имя сервера FTP (заметим, что он не обязательно должен находиться на серверах издателя или распространителя), используемого им порта, а также путь из корня FTP (здесь обязательно следует указать префикс `/ftp`).

Во вкладке Snapshot содержится ряд параметров, управляющих некоторыми аспектами работы агента распространения.

- Independent Distribution Agent. Установленное по умолчанию значение `true` означает, что при необходимости может быть запущено несколько агентов распространения, обслуживающих одну и ту же базу данных подписки. В SQL Server 2000 по умолчанию было принято, что при существовании нескольких публикаций в одной базе публикаций, все из которых имеют подписки из одной базы подписок, должен был существовать всего один агент распространения, совместно используемый всеми публикациями для данной базы подписки. В SQL Server 2005 в таком случае может существовать множество агентов распространения, каждый из которых открывает отдельный поток данных, направленный в базу данных подписки, что обеспечивает значительно бо́льшую производительность.
- Snapshot always available. Определяет, следует ли формировать новый снимок базы данных при каждом запуске агента снимков. По умолчанию это так. Если вы работаете только с именованными подписчиками, можете установить для этого параметра значение `false`.
- Allow anonymous subscriptions. Позволяет подписчикам создавать как именованные, так и анонимные подписки.
- Attachable subscription database. Позволяет копировать существующую базу данных подписки другому подписчику для ускорения развертывания. По умолчанию эта возможность отключена; скорее всего, эта возможность будет исключена в будущих версиях SQL Server.
- Allow pull subscriptions. Позволяет создавать подписки по требованию.
- Allow initialization from backup files. Позволяет создавать резервную копию, а затем восстановить ее у подписчика. При дальнейшем конфигурировании подписки нужно будет указать, что снимок базы должен развертываться из резервной копии. Для создания подписки в этом случае следует использовать хранимую процедуру `sp_addsnapshot` следующим образом:

```
exec sp_addsubscription @publication = N'test', @subscriber =
'имя_сервера_подписчика', @destination_db = N'awsub',
@subscription_type = N'Push',
@sync_type = N'initialize with backup', @article = N'all',
@update_mode = N'read only', @subscriber_type = 0,
@backupdevicetype='disk',@backupdevicename='c:\adventure.bak'
```



Большая часть аргументов хранимой процедуры `sp_addsubscription` не имеет значений, заданных по умолчанию. Например, вы обязаны определить аргумент `@publication`. В примере, рассматриваемом в этом разделе, будет показано, какой вид должны иметь передаваемые вами аргументы.

- The Allow non-SQL Server Subscribers. Позволяет выполнять репликацию в гетерогенные источники данных, такие как Oracle, Sybase, DB2 и др.
- Allow data transformations. Этот раздел позволяет запускать пакеты DTS для преобразования транзакций по пути их следования от издателя к подписчику. Этот параметр недоступен в SQL Server 2005, однако он доступен в SQL Server 2000, так что подписчики, использующие SQL Server 2005, смогут использовать это преимущество.
- Schema Replication. В этом разделе определите, хотите ли вы реплицировать инструкции DDL (`ALTER TABLE`, `ALTER VIEW`, `ALTER PROC` и т.п.) подписчикам. По умолчанию для этого параметра установлено значение `true`.
- Allow peer-to-peer subscriptions. Позволяет открыть публикацию для развертывания в одноранговой среде.
- Allow immediate updating. Определяет, открыта ли подписка для непосредственных обновлений.
- Allow queued updating. Определяет, доступна ли подписка для очередей обновлений.

  Если подписка открыта для немедленных обновлений или очередей, то доступны два дополнительных параметра: Report Conflicts Centraly и Conflict Resolution Policy. Первый из них определяет, будут ли все конфликты отправляться издателю или будут протоколироваться в таблицах на стороне подписчиков. Второй параметр определяет решения, принимаемые для разрешения конфликтов: выигрывает издатель, выигрывает подписчик или выигрывает издатель, а подписка реинициализируется.

Во вкладке Publication Access отображается список учетных записей, для которых открыт доступ к публикации. В эту группу должны быть включены все учетные записи, под которыми сконфигурированы агенты распространения и снимков.

Последняя вкладка диалогового окна параметров публикации — Agent Security. Здесь вы можете определить, под какими учетными записями будут запускаться все агенты чтения журнала, снимков, распространения и чтения очереди.

На этом мы завершаем рассмотрение процессов создания и администрирования публикаций транзакций. После того как публикация будет создана, ее нужно развернуть у подписчиков.

Создание публикаций двусторонней репликации транзакций

Двусторонние репликации транзакций лучше всего использовать в парах "издатель–подписчик", где необходим взаимный обмен транзакциями. Без существенных усилий невозможно расширить топологию этой репликации более чем на одну пару "издатель–подписчик". Если вам нужен двусторонний обмен репликациями между несколькими узлами, то лучшим реше-

нием будет использование репликации слияния или одноранговой репликации. Перед тем как развертывать решение двусторонней репликации, следует позаботиться о минимизации вероятности возникновения конфликтов. В идеальном случае в любой момент времени изменения должны выполняться только на одном узле — например, если вы используете двустороннюю репликацию из соображений отказоустойчивости системы, где заранее известно, что активизированным является только один узел. Если же вы собираетесь разрешить одновременное выполнение обновлений на нескольких узлах, следует разделить таблицы с помощью различных диапазонов идентификаторов и приращений или с помощью столбца идентификатора региона.

Для настройки двусторонней репликации транзакций создайте резервную копию базы данных публикаций и восстановите ее у подписчика. После этого модифицируйте столбцы, реализующие свойства идентичности, с тем, чтобы они имели другой домен значений. Например, можно установить стартовые значения у издателя и подписчика, соответственно в `1` и `2`, а также интервал приращения, равный двум. В этом случае нечетные значения идентификатора будут принадлежать издателю, а четные — подписчику. После этого модифицируйте все триггеры, ограничения и столбцы идентичности с параметром `NOT FOR REPLICATION`. Затем создайте публикацию на стороне издателя и пока не синхронизируйте ее с подписчиком. В базе данных публикаций на стороне издателя запустите хранимую процедуру `sp_scriptpublicationcustomprocs '`*`имя_публикации`*`'`. Результатом ее выполнения будет сценарий генерации хранимых процедур репликации, который следует запустить в базе данных подписки. Выполнив эти действия, повторите их на сервере подписчика. После этого можете запускать агенты распространения, если они еще не запущены.

При возникновении конфликтов работа агентов распространения будет остановлена. Причиной таких конфликтов могут быть нарушения согласованности данных и ограничений первичного ключа (например, при применении команды на стороне подписчика не была найдена строка). Вы можете включить протоколирование, чтобы иметь возможность выяснить, какая строка вызвала ошибку, и устранить проблему вручную. Возможно, для этого придется удалить строку.

Создание публикаций Oracle

Для создания публикации Oracle нужно в первую очередь создать учетную запись пользователя с правами администрирования репликации на сервере Oracle. Сценарий для создания такого пользователя вы можете найти в файле `C:\Program Files\Microsoft SQL Server\MSSQL.X\MSSQL\Install\OracleAdmin.sql`. Запустите этот сценарий с помощью PL/SQL и введите необходимую информацию (имя и пароль учетной записи, а также экземпляр сервера).

Теперь вы можете включить сервер Oracle в качестве издателя. Для этого выполните следующие действия.

1. Подключитесь к экземпляру SQL Server, который будет играть роль распространителя или распространителя/подписчика. Сам сервер следует заранее сконфигурировать как распространителя.
2. Щелкните правой кнопкой мыши на папке `Replication` и выберите в контекстном меню пункт Distributor Properties. Перейдите на вкладку Publisher.
3. Щелкните на кнопке Add и выберите пункт Add Oracle Publisher. Откроется диалоговое окно, в котором будет предложено подключиться к серверу Oracle. Это диалоговое окно очень похоже на то, которое открывается при подключении к SQL Server в Management Studio. Если вы точно не знаете имя сервера Oracle, щелкните на раскрывающемся списке Server Instance и выберите нужный сервер из списка.
4. Щелкните на кнопке OK, а затем на кнопке Options.

5. Для аутентификации выберите вариант Oracle Standard Authentication, введите имя и пароль учетной записи администратора репликаций, созданный ранее. Щелкните на кнопке Remember Password. Перейдите на вкладку Connection Properties, содержащую два параметра.
   - Gateway. Используйте этот параметр для высокопроизводительных решений репликации.
   - Complete. Используйте этот параметр, если хотите преобразовать данные или выполнить фильтрацию строк.
6. Щелкните на кнопке Connect.

Теперь, когда пользователь с привилегиями администратора репликаций создан на сервере Oracle, можно приступить к созданию публикации.

1. В Management Studio раскройте папку `Replication` и щелкните правой кнопкой мыши на папке `Local Publications`.
2. В контекстном меню выберите пункт New Oracle Publication, после чего разверните базы данных в папке `Replication`.
3. Щелкните правой кнопкой мыши на папке `Local Publications` и в открывшемся контекстном меню выберите пункт New Oracle Publication.
4. В открывшемся диалоговом окне Oracle Publisher выберите сервер Oracle, на котором хотите создать публикацию, и щелкните на кнопке Next.
5. В диалоговом окне выбора типа публикации выберите пункт Snapshot Publication или Transactional Publication.
6. Щелкните на кнопке Next, чтобы открыть диалоговое окно Articles. Выберите объекты, которые хотите реплицировать. Обратите внимание на присутствие параметров вертикальной фильтрации таблицы и просмотра объектов; при желании вы можете также отобразить типы данных.
7. Щелкните на кнопке Next, чтобы перейти к окну фильтрации строк таблицы. Это диалоговое окно аналогично рассмотренным в предыдущих разделах окнам публикаций транзакций и снимков базы данных.
8. Щелкните на кнопке Next, чтобы перейти к диалоговому окну Snapshot Agent. Примите предложенные по умолчанию значения и щелкните на кнопке Next, чтобы перейти на страницу параметров системы безопасности агента.
9. Сконфигурируйте учетные записи, под которыми будут запускаться агенты чтения журнала и снимков базы данных. Щелкните на кнопке Next, чтобы перейти в окно Actions и определить, хотите ли вы записать публикацию в сценарий.
10. Щелкните на кнопке Next, чтобы открыть заключительное окно мастера публикаций. Присвойте публикации имя, проверьте правильность установки параметров и щелкните на кнопке Finish. После того как публикация будет создана, создайте ее подписчиков.

Создание публикаций одноранговой репликации

Для создания публикации одноранговой репликации прежде всего создайте обычную публикацию транзакций, после чего выполните следующие действия.

1. Щелкните правой кнопкой мыши на созданной публикации транзакций (ее можно найти в папке `Replication\Local Publications`) и выберите в контекстном меню

пункт Properties. Перейдите на вкладку Subscription Options (рис. 39.7) и установите для параметра Allow Peer-to-Peer Subscriptions значение `true` (по умолчанию он имеет значение `false`).

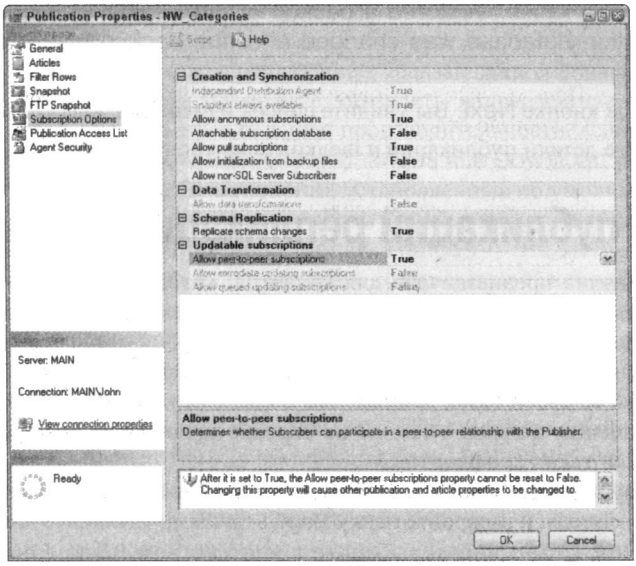


Рис. 39.7. Для включения одноранговой репликации установите для параметра Allow Peer-to-Peer Subscriptions значение `true`

2. Щелкните на кнопке OK, чтобы закрыть диалоговое окно параметров публикации.
3. Создайте резервную копию базы данных и разверните подписки, используя созданный архив.
4. Щелкните правой кнопкой мыши на публикации и выберите в контекстном меню пункт Configure Peer-toPeer Topology.
5. В открывшемся окне мастера щелкните на кнопке Next.
6. Выберите свою публикацию и щелкните на кнопке Next.
7. Выберите всех издателей, которых хотите включить в топологию одноранговой репликации.
8. В раскрывающемся списке выберите базы данных подписки и щелкните на кнопке Next.
9. Сконфигурируйте параметры системы безопасности, используемые для подключения к распространителю и издателю. Это придется сделать для всех баз данных издателя и подписчика, участвующих в одноранговой топологии.
10. Щелкните на кнопке Next, чтобы перейти к окну параметров системы безопасности агента распространения.
11. Сконфигурируйте учетные записи, используемые для подключения к распределителю и подписчику.
12. Щелкните на кнопке Next, чтобы перейти к окну New Peer Initialization. В этом окне вам будет предложено два варианта. Первый из них подходит в ситуациях, когда в базе данных публикации не было внесено изменений с момента конфигурирования подпис-

ки. В данном случае выберите пункт I created the peer database manually, or I restored a backup of the original publication database which has not been changed since the backup was taken. Если же с момента развертывания подписки в базе данных публикации были изменения, то вам потребуется снова создать ее резервную копию. Выберите пункт I restored a backup of the original publication database, and the publication database was changed after the backup was taken и восстановите базу из резервной копии.

13. Щелкните на кнопке Next. Вы увидите список изменений, внесенных в публикацию.
14. Просмотрите детали публикации и щелкните на кнопке Finish.

Создание публикаций репликации слияния

Репликация слияния предназначена для клиентов, которые время от времени или часто отключаются от издателя и вынуждены работать в автономном режиме со своими копиями базы данных. Периодически они должны выполнять синхронизацию своих изменений с издателем (*выгрузка данных*) и получать изменения от него (*загрузка данных*). Чем больше клиент работает автономно со своей базой данных, тем больше вероятность возникновения конфликтов. Репликация слияния должна с самого начала проектирования создаваться с учетом возможности обработки таких конфликтов.

Для создания публикации слияния подключитесь к экземпляру SQL Server, который будет выступать в роли издателя, и раскройте папку `Replication`. Щелкните правой кнопкой мыши на папке `Local Publications` и выберите в контекстном меню пункт New Publication. В открывшемся окне мастера публикаций щелкните на кнопке Next.

Начинается процесс создания публикации с конфигурирования параметров базы данных. Выберите базу данных публикаций и щелкните на кнопке Next. Выберите тип публикации Merge Publication и снова щелкните на кнопке Next. В открывшемся окне (рис. 39.8) выберите типы подписчиков, которые будет обслуживать издатель.

Теперь пришло время конфигурировать статьи. Щелкните на кнопке Next и в открывшемся окне выберите типы статей, которые будете реплицировать. Обратите внимание на невоз-

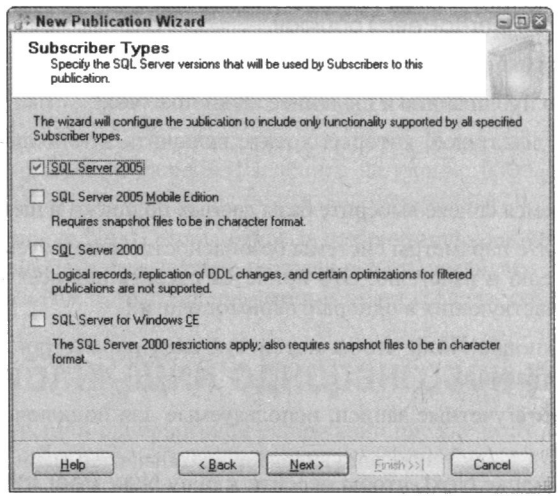


Рис. 39.8. Выберите типы подписчиков, которые будет обслуживать издатель

можность репликации выполнения хранимых процедур и индексированных представлений в виде таблиц. В то же время у вас есть возможность вертикальной фильтрации таблиц и представлений. Для выполнения такой фильтрации раскрывайте соответствующие элементы списка и снимайте флажки с тех столбцов, которые не будут участвовать в репликации. Щелкните на кнопке Article Properties, чтобы открыть диалоговое окно, позволяющее установить параметры всех или выделенных статей таблиц. Доступные параметры практически совпадают с теми, с которыми мы имели дело в репликациях транзакций и снимков базы данных.

Обратите внимание на то, что вы можете управлять направлением синхронизации на уровне отдельных статей. Вы добьетесь большей производительности, если выберете вариант Download only to Subscriber, Allow Subscriber Changes или Download only to Subscriber-prohibit Subscriber Changes, а не Bi-directional, установленный по умолчанию.

К тому же обратите внимание на параметры управления диапазоном идентичности (Identity Range Management). Authomatic Identity Ranges — это метод назначения таких диапазонов идентичности издателю и подписчикам, которые позволяют избежать конфликтов первичных ключей (такие конфликты возникают, когда издатель и подписчики одновременно используют одни и те же ключи). Вы можете назначить издателю один диапазон значений идентичности и совершенно отличные каждому из подписчиков. Например, издателю можно назначить диапазон от 1 до 10000, первому подписчику от 10001 до 11000, второму — от 11001 до 12000 и т.д. Когда значения идентичности у издателя или подписчиков устанавливаются путем умножения процентного порога на диапазон, SQL Server увеличивает диапазон при запуске агента слияния. Этот подход к управлению значениями идентичности позволяет эффективно использовать диапазон идентичности.

После установки параметров публикации щелкните на кнопке Next, чтобы перейти к окну Article Issues. Просмотрите предупреждения системы и при необходимости скорректируйте параметры публикации.

Фильтрация в репликации слияния работает несколько по-иному, нежели в других типах публикаций. Щелкните на кнопке Next, чтобы открыть окно фильтрации строк таблиц. Если вы хотите установить горизонтальную фильтрацию в статьях слияния, щелкните на кнопке Add. Вам будет предложено два варианта: Add Filter и Authomatically Generate Filters. Первый вариант позволяет генерировать фильтры строк отдельных таблиц. Второй вариант позволяет установить фильтр в одной таблице, после чего выполняется автоматическая установка соответствующих фильтров во всех связанных таблицах. При создании фильтра параметр в нижней части диалогового окна позволяет указать, как много подписок будут получать данные из таблицы. Второй параметр (A row from this table will only go to one subscription) предназначен только для предварительно вычисленных разделов. Во время создания статьи он будет аккумулировать больше метаданных репликации, но потребует меньше ресурсов процессора для слияния транзакций, которые будут происходить в будущем у издателя и подписчиков.

Щелкните на кнопке Next, чтобы определить график запуска агента снимков базы данных. Еще раз щелкните на кнопке Next, чтобы установить контекст безопасности, под которым будет запускаться агент снимков. Снова щелкните на кнопке Next, и вы увидите сводный отчет о создаваемой публикации. Введите имя публикации и щелкните на кнопке Finish. Публикация слияния будет создана.

Создание подписок

После создания и конфигурирования нескольких публикаций вы можете создать подписчиков, которые будут их использовать. Подключитесь к подписчику и раскройте папку `Replication`. Щелкните правой кнопкой мыши на папке `Local Subscriptions` и выберите в контекстном меню пункт New Subscriptions. В открывшемся окне мастера подписок щелкните на

кнопке Next. Откроется диалоговое окно, показанное на рис. 39.9. В раскрывающемся списке Publisher выберите издателя. Раскройте базу данных публикаций и выберите ту публикацию, на которую хотите подписаться. Обратите внимание, что каждая публикация имеет значок, указывающий на ее тип. Щелкните на нужной публикации и на кнопке Next.



Рис. 39.9. В этом окне выберите издателя и одну из публикаций

Теперь вы должны принять решение, какую подписку создать: принудительную или по требованию. Для принятия решения руководствуйтесь следующими рекомендациями.

- Если число подписчиков велико, используйте подписку по требованию.
- Если существуют подписчики, подключенные не постоянно, выбирайте подписку по требованию.
- Если ни одно из предыдущих условий не применимо, используйте принудительную подписку.

Далее следует выбрать одного или нескольких подписчиков. Щелкните на кнопке Next, чтобы перейти к окну Subscribers (рис. 39.10). Выделите серверы, которых хотите сделать подписчиками данной публикации, и в соответствующих раскрывающихся списках выберите базы данных подписки. Вы можете включить подписчиков, не отображенных в предложенном списке, щелкнув на кнопке Add Subscriber.

Каждый из подписчиков требует настройки профиля безопасности. Щелкните на кнопке Next, чтобы открыть диалоговое окно Distribution Agent Security. Щелкните на кнопке с изображением эллипса, расположенной рядом с каждым подписчиком, и сконфигурируйте его профиль безопасности. Только после настройки всех необходимых профилей кнопка Next станет доступной.

Определите время, когда каждый из подписчиков доступен для репликации данных, осуществляемой издателем. Для этого щелкните на кнопке Next, чтобы открыть диалоговое окно Synchronization Schedule. В раскрывающемся списке Agent Schedule определите, должен ли агент быть включенным постоянно, запускаться по требованию (т.е. с помощью Windows Synchronization Manager) или согласно установленному графику.

Щелкните на кнопке Next, чтобы перейти в окно инициализации подписки. Если публикация поддерживает обновляемые подписки, то предварительно откроется диалоговое окно

Updatable Subscriptions. В этом случае в раскрывающемся списке Commit at Published выберите Simultaneously Commit Changes, если хотите использовать немедленное обновление, или Queue Change and Commit When Possible, если решили использовать очередь обновлений. Откроется диалоговое окно, предлагающее ввести информацию об учетной записи для связанного сервера, использующего подписку с обновлениями. Вы можете создать учетную запись или выбрать использование связанного сервера между подписчиком и издателем. Щелкните на кнопке Next, чтобы наконец-то открылось окно инициализации подписки.

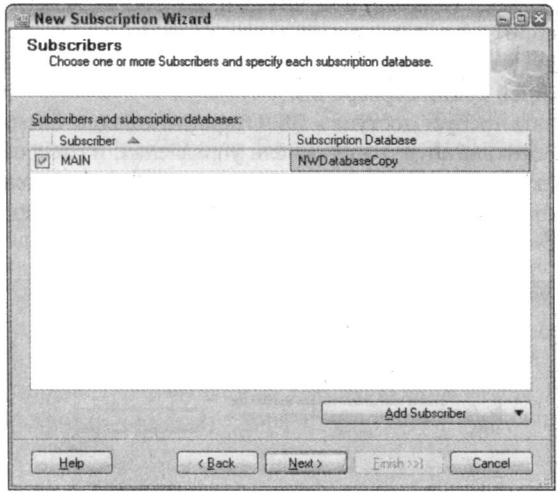


Рис. 39.10. Выберите подписчиков данной публикации; каждый из них должен иметь базу данных, в которой будут храниться реплицируемые данные

Если вы развернули подписку из резервной копии или вручную создали базы данных подписки с помощью схемы и данных объектов, которые хотите реплицировать, то можете вручную переместить базу данных. В противном случае выберите пункт Immediately, если хотите немедленно отправить снимок базы подписчику. Если же вы хотите управлять моментом, когда снимок будет доставлен подписчику, выберите пункт At First Synchronization. Щелкните на кнопке Next, чтобы открыть окно Wizard Actions.

Если вы создаете подписку на публикацию слияния, то перед окном инициализации откроется окно выбора типа подписки. В этом окне вы можете установить приоритет подписки и выбрать ее тип. Если выбрать тип клиентской подписки, то конфликт выиграет первый подписчик, участвующий в слиянии с издателем. Если выбрать серверный тип подписки, то можно выбрать приоритет, назначаемый подписчику. Назначая разные приоритеты разным серверам, на которые выполняется репликация, вы можете управлять тем, какой из серверов выиграет в конфликте. Щелкните на кнопке Next, чтобы открыть окно инициализации подписки.

В этом диалоговом окне вы можете либо немедленно создать подписку, либо создать сценарий для ее последующего развертывания, либо выбрать оба варианта. Если вы выберете вариант немедленного создания подписки, откроется диалоговое окно Subscription Summary; если вариант создания сценария — окно Script File Properties, в котором вам будет предложено ввести имя создаваемого файла сценария и место его размещения. Щелкните на кнопке Finish, чтобы создать подписку и/или сгенерировать файл сценария подписки. Ход процесса создания подписки и ее сценария вы можете отслеживать в окне Creating Subscription. После завершения процесса щелкните на кнопке Close.

Чтобы проверить факт успешного развертывания подписки, щелкните правой кнопкой мыши на подписке в папке `Local Subscription` и выберите в контекстном меню пункт View Subscription Status.

Создание подписок Web-синхронизации

SQL Server для подписок слияния поддерживает Web-синхронизацию. Для включения этой функции вы должны решить, хотите ли получить сертификат от какого-либо уполномоченного источника (например, от `thawte.com`, `verisign.com` или `cacert.com`) или от службы управления сертификатами своей операционной системы. В любом случае сертификат следует установить в вашей копии сервера IIS.

Установка сертификата требует доступа к IIS. Откройте консоль Internet Information Services, находящуюся в папке Administrative Tools панели управления, и раскройте папку `Web Sites`. Щелкните правой кнопкой мыши на Web-сайте, который хотите использовать для подписок, и выберите в контекстном меню пункт Properties. В открывшемся диалоговом окне (рис. 39.11) перейдите на вкладку Directory Security.

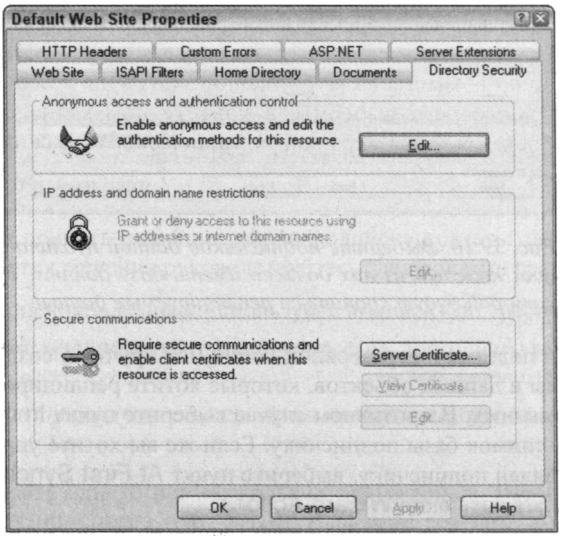


Рис. 39.11. Вкладка Directory Security предлагает установить сертификат на сервер IIS

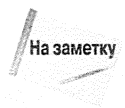
Инициализируйте установку сертификата, щелкнув на кнопке Server Certificate. В открывшемся окне мастера сертификатов щелкните на кнопке Next, а затем на кнопке Create a New Certificate. Щелкните на кнопке Next и выберите пункт Prepare the Request Now. (Если вы используете уполномоченного издателя сертификатов, щелкните на кнопке Send the Request Immediately to an Online Certification Authority.) Щелкните на кнопке Next и в разделе Name and security settings введите имя и длину (в битах) своего сертификата. Щелкните на кнопке Next и введите название своей организации и ее подразделения. Снова щелкните на кнопке Next. Теперь введите имя, которое будет вашим именем DNS, или NetBIOS-имя компьютера. Щелкните на кнопке Next. В диалоговом окне Geographic Information введите название своей страны, региона и города. Щелкните на кнопке Next. Выберите имя файла и путь для своего сертификата. Щелкните на кнопке Next, просмотрите выбранные параметры и снова щелкните на кнопке Next, а затем на кнопке Finish.

Если вы выбрали использование собственноручно созданного сертификата, начните с открытия консоли Certification Authority, расположенной в папке Administrative Tools панели управления. Подключитесь к своему серверу сертификатов, щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт All Tasks⇨Submit a New Request. Найдите ранее созданный файл сертификатов и щелчком откройте его. Раскройте папку Pending Request и щелкните правой кнопкой мыши на сертификате, который найдете в ней. Выберите в контекстном меню пункт Tasks⇨Issue. Раскройте папку Issued Certificates, щелкните правой кнопкой мыши на выпущенном сертификате и выберите в контекстном меню пункт All Tasks⇨Export Binary Data. Выберите вариант действий Save Binary Data to a File и щелкните на кнопке OK. Выберите место, где хотите сохранить файл, и примите предложенное имя. Щелкните на кнопке Save.

Вернитесь в консоль IIS и щелкните на кнопке Server Certificate. Щелкните на кнопке Next, после чего выберите Process the Pending Request and Install the Certificate. Щелкните на кнопке Next. Найдите место, в котором вы только что сохранили сертификат (его именем, скорее всего, будет Binary Certificate-*X*.tmp, где *X* — целое число). Дважды щелкните на сертификате, а затем один раз на кнопке Next. Выберите используемый порт SSL. Дважды последовательно щелкните на кнопке Next, а затем на кнопке Finish. Ваш сертификат установлен.

Теперь IIS готов к использованию в качестве подписчика. Подключитесь к издателю с помощью Management Studio, раскройте папку Replication, а затем папку Local Publications. Щелкните правой кнопкой мыши на публикации, которую хотите активизировать для Web-синхронизации, после чего выберите в контекстном меню пункт Configure Web Synchronization. Откроется окно мастера конфигурирования Web-синхронизации. Ниже описаны последующие действия.

1. Щелкните на кнопке Next. Выберите тип клиентов, которые будут подключаться к вашей репликации слияния. У вас есть выбор между стандартными клиентами SQL Server и мобильными клиентами.
2. Щелкните на кнопке Next. Откроется диалоговое окно Web Server (рис. 39.12). Выберите в нем имя Web-сервера, к которому будут подключаться клиенты репликации слияния. Этот сервер не обязательно должен быть сервером издателя.
3. Выберите существующий виртуальный каталог или пункт Create a New Vitrual Directory в диалоговом окне Web Server. Как правило, вам потребуется использовать новый виртуальный каталог, чтобы отделить свои данные от других. Если вы решили поступить так, щелкните на кнопке Next и перейдите к п. 4. Если вы все же захотите использовать уже существующий каталог, разверните Web-сайт, выберите этот каталог, щелкните на кнопке Next и сразу перейдите к п. 5.
4. В диалоговом окне Virtual Directory Information введите псевдоним, который хотите использовать для своего виртуального каталога, после чего введите путь, такой как c:\inetpub\wwwroot. Щелкните на кнопке Next. Если указанная папка физически еще не существует, то вам будет предложено ее создать. В этом случае щелкните на кнопке Yes.



Если выбранный вами каталог еще не имеет динамическую библиотеку SQL Server ISAPI, необходимую для выполнения Web-синхронизации, вам будет предложено скопировать ее туда. В этом случае согласитесь с предложением (т.е. щелкните на кнопке Yes).

5. Щелкните на кнопке Next, чтобы открыть диалоговое окно Authenticated Access. Здесь вы можете выбрать один из трех режимов аутентификации: Integrated Windows authentication, Digest authentification for Windows domain servers или Basic authentication.

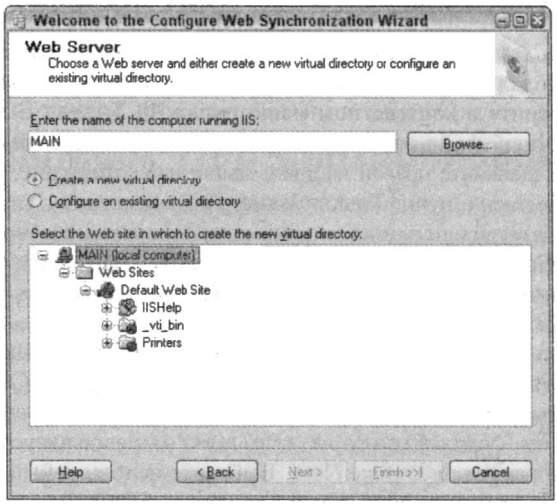


Рис. 39.12. Выберите настройки Web-сервера, которые хотите использовать для подписки



Интегрированную аутентификацию Windows можно использовать только в локальных и корпоративных сетях. Ее нельзя использовать, осуществляя соединение через Интернет, не открывая дополнительные порты, которые в противном случае должны быть защищены. Если вы выбрали интегрированную аутентификацию, щелкните на кнопке Next, чтобы открыть диалоговое окно Directory Access. При использовании интегрированной аутентификации удостоверьтесь, что учетные записи, используемые для подключения к Web-серверу, находятся в списке PAL вашей публикации и являются теми учетными записями, которые будут использовать подписчики для Web-синхронизации с сервером.



Чтобы обеспечить максимально возможный уровень защиты, старайтесь всегда выбирать аутентификацию Windows, когда это возможно. Аутентификацию Basic authentication используйте в последнюю очередь, так как она посылает имя пользователя и пароль в открытом текстовом виде. Используйте аутентификацию Digest authentication в соединении с SSL/HTTPS, при этом предварительно создавая учетные записи в службе активных каталогов. Вам придется добавить имя домена в область действия и убедиться, что учетные записи, используемые для подключения к Web-серверу с помощью служб Web-синхронизации, находятся в списке PAL вашей публикации.

6. Щелкните на кнопке Next, чтобы открыть диалоговое окно Directory Access. Введите имена учетных записей, которые будут использовать подписчики для подключения к Web-серверу с помощью служб Web-синхронизации. Для упрощения управления в этом случае лучше образовать группу.
7. Щелкните на кнопке Next, чтобы открыть окно Snapshot Share Access. Общий ресурс должен быть представлен в виде `\\сервер_издателя\имя_общего_ресурса`. Это общее имя будет отображаться на альтернативную папку хранения снимков базы данных, которую мы сконфигурируем немного позже.
8. Щелкните на кнопке Next и просмотрите созданные параметры настройки. Щелкните на кнопке Finish. Щелкните на кнопке Close, когда мастер завершит конфигурирование Web-синхронизации.

Теперь, когда конфигурирование Web-синхронизации публикации слияния завершено, щелкните на этой публикации правой кнопкой мыши и выберите в контекстном меню пункт Properties. Вначале нужно сконфигурировать снимок базы данных. Перейдите на вкладку Snapshot и в текстовом поле Put files in the following folder введите имя вашей папки снимков. К этой папке на Web-сервере должен быть открыт общий доступ. Если Web-сервер размещен на одном компьютере с SQL Server, то это будет папка, принятая по умолчанию для хранения снимков. Убедитесь, что данный параметр установлен.

Далее следует сконфигурировать виртуальный каталог, предназначенный для хранения снимков базы данных. Перейдите на вкладку FTP Snapshot and Internet и выберите параметр Allow Subscribers to synchronize by connecting to a web server. Адрес Web-сервера будет выглядеть следующим образом: `https://имя_web_сервера/имя_виртуального_каталога`. Именем виртуального каталога будет выступать то имя, которое было назначено в диалоговом окне Web Server мастера конфигурирования Web-синхронизации.

После этого проверьте тип подписки, чтобы убедиться, что пользователи смогут к ней подключиться. Щелкните на кнопке Subscription Options и проверьте, установлено ли для параметра Allow Pull Subscription значение `True`. После этого щелкните на кнопке Publication Access List. Добавьте учетную запись или группу, которую будут использовать подписчики для подключения к Web-серверу и издателю.

Теперь мы полностью подготовились к созданию Web-подписки.

1. Щелкните правой кнопкой мыши на папке `Local Subscriptions` сервера подписчика и выберите в контекстном меню пункт New Subscription.
2. Щелкните на кнопке Next. В раскрывающемся списке выберите нужного издателя, найдите свою публикацию и выделите ее.
3. Щелкните на кнопке Next. В диалоговом окне Merge Agent Location выберите пункт Run each agent at its Subscriber.
4. Щелкните на кнопке Next. Выделите ваших подписчиков, а в раскрывающемся списке Subscription Database выберите базу данных подписки.
5. Щелкните на кнопке Next. В следующем окне выберите учетную запись, под которой агент слияния будет запускаться на сервере подписчика. В поле Connect to the Publisher and Distributor введите имя учетной записи, которое вы задали в списке PAL.
6. Щелкните на кнопке OK, а затем на кнопке Next. В диалоговом окне расписания синхронизации выберите режим Run on Demand Only (Запуск по требованию) или Schedule (Запуск по расписанию). Режим Run Continuously, вероятнее всего, не будет работать при подключении через Интернет.
7. Щелкните на кнопке Next, чтобы открыть диалоговое окно инициализации подписки. Решите, хотите ли вы сгенерировать снимок базы данных немедленно или когда-нибудь позже.
8. Щелкните на кнопке Next. В открывшемся окне выберите Use Web Synchronization.
9. Щелкните на кнопке Next. В диалоговом окне Web Server Information проверьте правильность адреса URL. Введите имя, которое хотите использовать для базовой аутентификации.
10. Щелкните на кнопке Next. Убедитесь, что в типе подписки выбран Client.
11. Щелкните на кнопке Next и затем еще раз на кнопке Next, чтобы сгенерировать снимок базы данных немедленно (или, как вариант, создать сценарий для развертывания подписки в будущем).
12. В диалоговом окне Subscription Summary щелкните на кнопке Finish.

Снимок базы данных будет применен к подписчику. Вы сможете увидеть его состояние, щелкнув правой кнопкой мыши на подписке в папке `Local Subscription` сервера подписчика и выбрав в контекстном меню пункт View Synchronization Status. Чтобы увидеть самую свежую информацию о состоянии подписчика, вам может потребоваться щелкнуть на кнопке Start.

Мониторинг решений репликации

Теперь мониторинг всей топологии репликации доступен в обособленном компоненте Replication Monitor, размещенном в файле `\Program Files\Microsoft SQL Server\90\Tools\Binn\SQLMonitor.exe`. Его также можно запустить, щелкнув правой кнопкой мыши на папке `Replication` в Management Studio и выбрав в контекстном меню пункт Launch Replication Monitor. Если развернуть некоторого издателя в окне этой утилиты, то отобразятся все его публикации. С помощью щелчка правой кнопкой мыши на издателе вы сможете сделать следующее:

- изменить настройки издателя;
- удалить издателя;
- подключиться к распространителю;
- отключиться от распространителя;
- установить профили агентов:
- сконфигурировать предупреждения репликации.

Вы можете щелкнуть правой кнопкой мыши на отдельной публикации, чтобы выполнить следующее:

- повторно инициализировать все подписки;
- сгенерировать снимок базы данных;
- изменить параметры публикации;
- обновить состояние публикации;
- проверить подписки (для репликаций транзакций и слияния).

Чаще всего вам придется иметь дело с настройками профилей агентов.

Профили агентов

Агенты репликации имеют множество переключателей, которые объединены в профили и группы, предназначенные для разных функций. По умолчанию репликация снимков базы данных имеет всего один профиль, а репликация транзакций — пять: Continue on data consistency errors (принуждает агента распределения пропускать ошибки, связанные с нарушением ограничений первичного ключа и отсутствием строк у подписчика); профиль по умолчанию; словесный профиль (предназначен для отладки), потоковый профиль OLE DB (используется при репликации особо крупных двоичных объектов) и профиль Windows Synchronization Manager (используется, когда подписка управляется диспетчером синхронизации Windows). Агенты слияния имеют следующие профили: High-Volume Server-to-Server, проверки количества строк и контрольных сумм, проверки количества строк, медленных подключений (для работы через медленные каналы, такие как телефонные линии), агента словесной истории (для отладки) и диспетчера синхронизации Windows. Для точного управления вы можете создать и собственные профили.

Вернемся к утилите Replication Monitor. Если вы щелкнули на публикации слияния или снимков базы данных, то на правой панели увидите две вкладки:

- All Subscriptions (Все подписки);
- Warnings and Agents (Предупреждения и агенты).

При желании можно отобразить и третью вкладку, предназначенную для публикаций репликации транзакций, — Tracer Tokens (Маркеры трассировки).

Все подписки

Во вкладке All Subscriptions отображаются все подписки и их состояние, их подписчики и базы данных, а также последнее время синхронизации. Эти подписки можно отфильтровать, щелкнув на кнопке Show и выбрав один из следующих вариантов:

- All Subscribers (Все подписчики);
- The 25 worst performing subscriptions (25 подписок с наибольшей производительностью);
- The 50 worst performance subscriptions (50 подписок с наибольшей производительностью);
- Errors and Warning only (Только предупреждения и ошибки);
- Errors only (Только ошибки);
- Warning only (Только предупреждения);
- Subscriptions running (Запущенные подписки);
- Subscriptions not running (Незапущенные подписки).

Эти параметры позволяют повысить наглядность данных о подписках.

Предупреждения и агенты

Во вкладке Warnings and Agents (рис. 39.13) на нижней панели отображаются все агенты. Если щелкнуть правой кнопкой мыши на любом из агентов, можно выполнить следующее:

- просмотреть журнал операций агента;
- запустить агента;
- изменить параметры агента (именно здесь вы можете поупражняться с высокоточным управлением агентом).

Вкладка Warnings and Agents имеет два раздела:

- Warnings (Предупреждения);
- Agent Status (Состояние агента).

Вкладка Warnings используется для инициирования предупреждений при выполнении заданных условий, например, когда задержка превосходит наперед заданное значение или срок годности подписки истекает. Эти предупреждения включаются установкой соответствующих флажков. После включения предупреждений нужно определить, как они должны обрабатываться. Щелкните на кнопке Configure Alerts, выберите предупреждение и щелкните на кнопке Configure. Теперь у вас появится возможность включить предупреждение, инициировать предупреждение, когда сообщение содержит некоторую строку, остановить и запустить агент, отправить сообщение оператору по электронной почте, на пейджер или с помощью службы Net Send, а также определить ведение журнала.

Вкладка Agent Status позволяет просматривать детали задания, останавливать и запускать агента, выбирать для него профиль, а также изменять его параметры. Для установки параметров открывается отдельное диалоговое окно Job Properties (рис. 39.14).

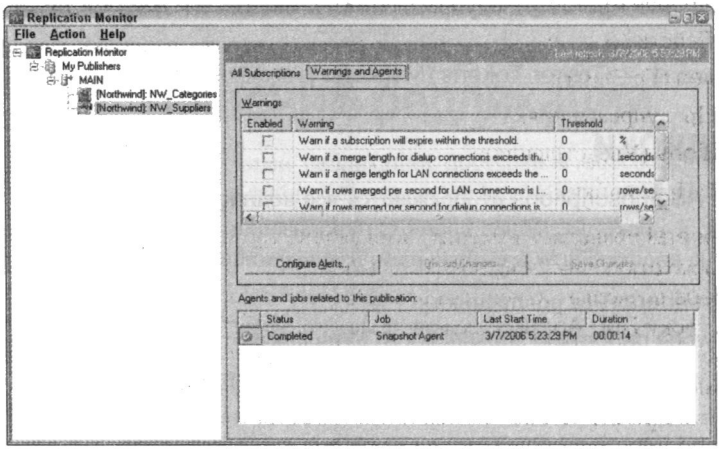


Рис. 39.13. Используйте вкладку Warnings and Agents для конфигурирования уведомлений агента

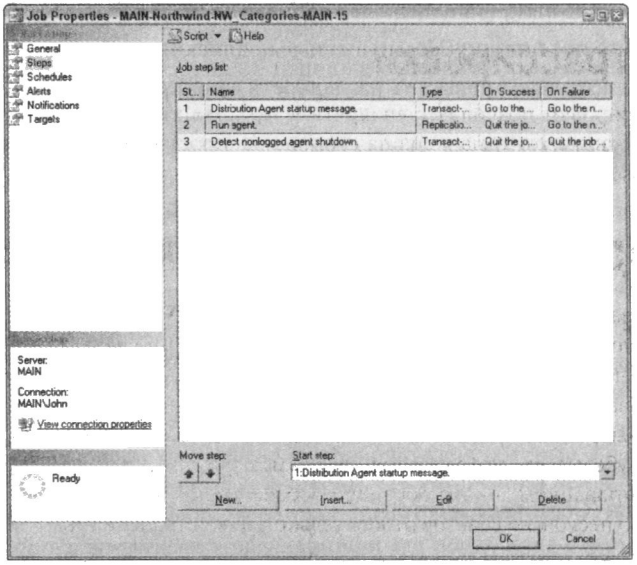


Рис. 39.14. Измените параметры агента, изменяя по мере необходимости действия, выполняемые им при запуске

Изменение параметров агента репликации

Когда необходимо изменить режим работы агента, принятый по умолчанию, это можно выполнить с помощью замены профиля или установки отдельных параметров агента. Для изменения параметров найдите в мониторе репликаций подписку, агента которой следует изме-

нить, щелкните на ней правой кнопкой мыши и выберите в контекстном меню пункт View Details. В диалоговом окне Subscription выберите Actions⇨Distribution Agent Properties. Откроется диалоговое окно параметров заданий, содержащее шесть вкладок:

- General (Общие);
- Steps (Действия);
- Schedules (Графики);
- Alerts (Предупреждения);
- Notifications (Уведомления);
- Targets (Приемники).

Вкладка General позволяет изменить имя задания, его владельца, категорию, описание категории, а также отключить и включить задание.

Вкладка Steps позволяет изменить параметры агента, показанные на рис. 39.14.

Выделите строку Run Agent и щелкните на кнопке Edit. В разделе Command вы можете добавить новые параметры или изменить значения уже существующих. Например, вы можете добавить параметр `-QueryTimeout 300`, который позволит подсистеме репликации выждать 300 секунд, прежде чем пометить агента как сомнительного.

Вкладка Schedules используется для изменения графика работы агента. Вкладка Alerts позволяет создать предупреждения; вкладка Notifications позволяет уведомлять операторов о сбоях работы агентов репликации, а вкладка Targets используется при работе с главным сервером заданий.

Маркеры трассировки

Маркеры трассировки позволяют внедрять в публикацию маркеры, фиксирующие время выполнения репликации этого маркера на стороне распространителя, а затем на стороне подписчиков. Использование этих маркеров позволяет оценить значение задержки репликации.

Производительность репликации

Существуют два аспекта, связанных с повышением производительности репликаций: создание и развертывание снимка базы данных и распространение данных.

По мере увеличения размеров изначального снимка базы его развертывание с помощью метода резервирования/восстановления начинает обеспечивать более высокую производительность. Если вы решили распространять снимок базы с помощью агентов слияния или распространения, убедитесь, что установили для базы подписки модель восстановления с неполным протоколированием, — это обеспечит более высокую производительность загрузки.

Ниже приведено несколько советов относительно распространения транзакций в репликации.

- Минимизируйте использование триггеров и индексов, чтобы максимально облегчить таблицы подписчика.
- Если основная масса операций записи, выполняемых на стороне издателя, выполняется в пакетах, подумайте о репликации выполнения хранимых процедур. Если же большая часть операций записи выполняется в виде одиночных команд над строками, то от репликации выполнения хранимых процедур вы не получите никакой выгоды.
- Установите для параметра `PollingInterval` в агенте распространения значение, равное одной секунде.

А вот несколько советов, касающихся репликаций слияния.

- Помещайте индексы в столбцы, участвующие в условиях объединения.
- Если такое возможно, используйте предварительное вычисление разделов.
- Если такое возможно, используйте статьи, предназначенные только для загрузки.



Для просмотра метаданных всех реплицированных транзакций выберите `sysdm_repl_traninfo` в динамическом представлении управления.

Разрешение проблем репликации

Для разрешения проблем репликации поищите в Replication Monitor издателей и публикации, имеющие значок с белым крестом в красном кружке. На панели All Subscriptions найдите проблемную подписку, щелкните на ней правой кнопкой мыши и выберите в контекстном меню пункт View Details. В нижней части открывшегося окна вы увидите сообщение об ошибке. Если ничего не отображается, закройте окно View Details, щелкните правой кнопкой мыши на сбойном агенте и выберите профиль Verbose Agent. Перезапустите агента. На этот раз в нижней части окна View Details вы увидите более подробное описание ошибок.

Большинство ошибок являются преходящими и легко устраняются с помощью перезапуска агентов слияния или распространения.

Иногда, особенно при работе репликации слияния или использующей обновления, вы можете обнаружить неожиданную потерю данных. В большинстве случаев это связано с конфликтами. Для просмотра этих конфликтов (и возможного отката, если используется репликация слияния) щелкните правой кнопкой мыши на публикации в Management Studio и выберите в контекстном меню пункт View Conflicts.

Если данные все еще не отправлены подписчику, проверьте его состояние синхронизации. Щелкните правой кнопкой мыши на подписке и выберите в контекстном меню пункт View Synchronization Status.

При желании вы можете протоколировать вывод агентов репликации в файлы журналов. Для этого установите в проблемном агенте следующие ключи, после чего выполните его перезапуск:

```
-OutputVerboseLevel 3 -Output c:\Temp\Out.log
```

Некоторые ошибки агента распространения можно пропускать, установив в нем ключ `-Skiperrors` *XXX*, где *XXX* — номер ошибки.

Существует еще несколько команд, позволяющих заглянуть внутрь процессов репликации. Например, хранимая процедура `sp_hrowsereplcmds` показывает содержимое базы данных распространения, а представление `distribution.dbo.MSDistribution_status` — сколько команд было распространено подписчикам и сколько еще ожидает распространения. Хранимая процедура `sp_repltran`, выполненная в базе данных публикаций, отображает команды, ожидающие выборки агентом чтения журнала транзакций.

Пакеты обновлений

С выходом пакетов обновлений SP1 и SP2 ряд изменений коснулся и службы репликаций в SQL Server.

- Зеркальное отображение базы данных можно использовать совместно с репликацией.
- Для таблиц, которые используют ограничения каскадной ссылочной целостности, такие как `ON DELETE CASCADE` или `ON UPDATE CASCADE`, репликация поддерживает операции слияния в предварительно вычисленных разделах.
- В редакции Enterprise Edition можно инициализировать подписку в публикации снимков и транзакций, используя снимок базы данных. Для этого нужно установить для свойства публикации `sync_method` значение `database snapshot` или `database snapshot character`. Снимки баз данных закладывают фундамент устойчивого механизма обработки снимков, способный уменьшить конкуренцию блокировок в базе данных публикаций во время генерации снимка.
- Репликация слияния теперь обеспечена хранимой процедурой, регенерирующей триггеры, хранимые процедуры и представления, которые используются для отслеживания изменений в данных.

Резюме

Репликация — это один из методов распространения данных, и он не всегда лучше других подходит к конкретным потребностям распространения данных. В данной главе мы обсудили альтернативы, равно как и различные типы самой репликации. Для каждого из этих типов было указано, когда его лучше использовать, также был представлен метод развертывания данного решения репликации. В этой главе мы рассмотрели множество новых, введенных в SQL Server 2005, средств репликации, а также новые интерфейсы развертывания и администрирования репликаций.

ГЛАВА

40

Защита баз данных

В этой главе...

Концепция системы безопасности SQL Server

В чем смысл множества регистрационных записей

Безопасность на уровне сервера

Роли уровня сервера, баз данных и приложений

Предоставление, отзыв и запрет доступа

Рекомендуемые модели безопасности

Представления и безопасность

Когда я работал техническим специалистом по системам данных в структуре военно-морских сил США, я почти два года провел в CSTSC (Combat System Technical School Command) в Калифорнии. Это было прекрасное время. Мой класс был одним из последних, тренировавшихся на компьютерах AN-UYK-7. Центральный процессор состоял примерно из пятидесяти плат, собранных на транзисторах. Мы учились выявлять и устранять проблемы такого процессора на логическом уровне. Это было действительно здорово. На одном с нами острове располагалась школа криптографов. Система безопасности в этой школе была очень строгой — учащиеся не имели права выносить за ее пределы практически ничего: ни записей, ни книжек, ни тетрадей. Слава Богу, мы имели возможность встречаться с ними в свободное время. Защита информации никогда меня особо не волновала, однако принципы информационной архитектуры гласят, что информация должна быть защищена.

Как правило, разработчики вначале проектируют базу данных, а уже потом начинают заботиться о ее защите. Несмотря на то что нет никакого смысла думать о системе безопасности, когда сама структура базы находится в постоянно меняющемся состоянии, проект только выиграет, если вы займетесь защитой информации как можно раньше.

Безопасность, равно как и все остальные аспекты проекта базы данных, должна быть тщательно продумана, реализована и протестирована. Система безопасности может оказать влияние на выполнение некоторых процедур, и ее следует принимать во внимание при создании программного кода.

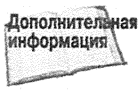
Небольшим организациям вполне подойдет простой план защиты, содержащий несколько пользовательских ролей и единую для всех IT-специалистов роль системного администратора. Крупным компаниям требуется более сложный план защиты, продуманный так же тщательно, как и сама логическая схема базы данных.

Если система безопасности разрабатывалась с привлечением средств аудита утилиты SQL Profiler, то инсталляция SQL Server может быть сертифицирована с уровнем безопасности C2. К счастью, модель безопасности SQL Server хорошо

продумана, доступна для понимания, достаточно логичная и гибкая. Несмотря на то что тактика построения системы безопасности базы данных сводится к созданию пользователей и ролей и последующему назначению им разрешений, стратегия состоит в идентификации прав и ответственностей, связанных с доступом к данным, и последующей разработке плана.



Система безопасности в SQL Server 2005 усовершенствована в нескольких направлениях. В качестве владельца объектов используется схема, разрешения более дисперсные, код может быть запрограммирован для выполнения конкретным пользователем, а данные могут шифроваться.



Возможно, вы слышали, что раньше СУБД SQL Server 2005 называлась SQL Server 2003. Затем компания Microsoft потратила два года на пересмотр программного кода, относящегося к системе безопасности SQL Server. Фокусирование внимания на защите информации повлияло и на изначальные установки. В предыдущих версиях SQL Server устанавливался с большой площадью покрытия, потенциально открытой для множества атак. Теперь SQL Server устанавливается в максимально защищенном состоянии, и системный администратор должен сам разрешать определенные функции, чтобы они могли быть использованы. По умолчанию запрещены даже удаленные подключения.

Новая утилита SQL Server Surface Area Configuration используется для включения функций и компонентов, т.е. для управления площадью покрытия. Она обычно запускается сразу после установки SQL Server. После конфигурирования СУБД администратору редко требуется использовать эту утилиту в дальнейшем. О работе с этой утилитой см. в главе 4.

Концепции защиты

Модель защиты SQL Server большая и сложная. В некотором роде она даже более сложная, чем в Windows. Так как концепции защиты тесно переплетены между собой, лучше начать их рассмотрение с общего обзора модели.

Система безопасности SQL Server основана на концепции *защищаемых объектов (securables)*, т.е. объектов, *на* которые можно назначать разрешения, и *принципалов (principles)*, т.е. объектов, *которым* можно назначать разрешения. Принципалами могут быть пользователи и роли. Пользователи назначаются ролям; при этом как пользователям, так и ролям могут предоставляться разрешения на доступ к объектам (рис. 40.1) Каждый объект имеет своего владельца, и права собственности также влияют на разрешения.

Система безопасности уровня сервера

Пользователь в первую очередь должен быть идентифицирован сервером одним из следующих методов:

- с помощью учетной записи пользователя Windows;
- на основе членства в одной из групп пользователей Windows;
- с помощью отдельной регистрационной записи SQL Server (если на сервере используется смешанная модель безопасности).

На уровне сервера пользователь распознается по своему идентификатору (`LoginID`), который может быть либо регистрационной записью SQL Server, либо именем домена и пользователя Windows.

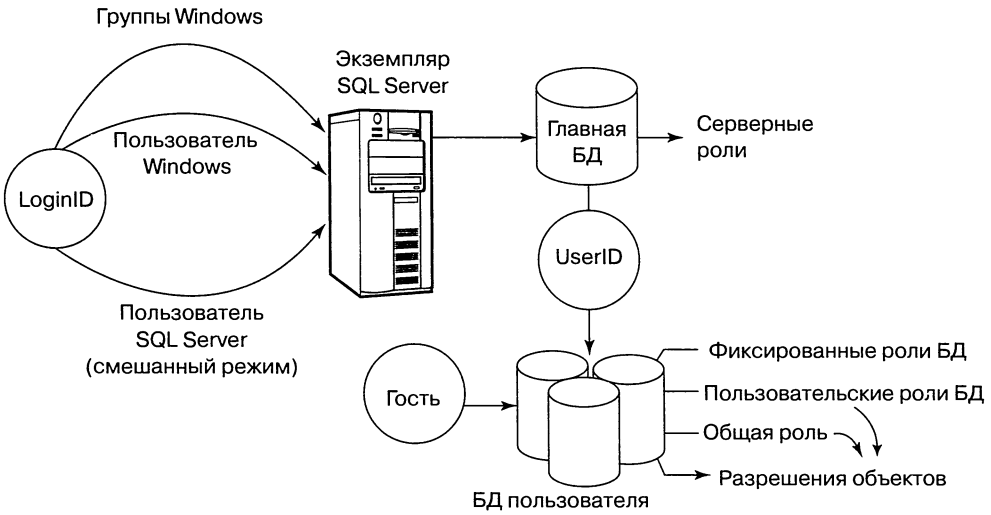


Рис. 40.1. Обобщенная схема системы безопасности SQL Server. Показано, как пользователи вначале аутентифицируются сервером, затем базой данных и объектами базы данных. В кружках представлены элементы идентификации пользователя

Как только пользователь был представлен серверу и идентифицирован, он получает те права, которые были назначены его фиксированной серверной роли администратором. Если пользователь принадлежит роли `sysadmin`, то он имеет полный доступ ко всем функциям сервера, а также ко всем базам данных и объектам на сервере.

Пользователю может быть разрешен доступ к базе данных, и его идентификатор отображается на специфический для базы данных идентификатор пользователя (`UserID`). Если пользователю доступ к базе данных явно не разрешен, он может обратиться к ней как гость (если в конфигурацию сервера предварительно внести некоторые изменения).

Система безопасности уровня базы данных

На уровне базы данных пользователю могут быть предоставлены административные привилегии с помощью прикрепления к фиксированной роли базы данных.

В то же время пользователь пока не имеет доступа к данным. Для этого ему должны быть предоставлены разрешения к отдельным объектам базы данных (например, таблицам, хранимым процедурам, представлениям и функциям). *Пользовательские роли* — это дополнительные роли, служащие в качестве групп. Роли может быть разрешен доступ к объекту базы данных, а пользователю может быть назначена пользовательская роль базы данных. Все пользователи автоматически становятся членами стандартной роли базы данных `public`.

Разрешения к объектам назначаются с помощью инструкций `GRANT` (предоставить), `REVOKE` (отозвать) и `DENY` (запретить). Запрет привилегии замещает собой ее предоставление, а предоставление привилегии замещает собой ее отзыв. Пользователю может быть предоставлено множество разрешений к объекту (индивидуальных, наследованных от роли, обеспеченных принадлежностью к роли `public`). Если какая-либо из этих привилегий запрещена, для пользователя блокируется доступ к объекту. В противном случае, если какая-либо из привилегий предоставляет разрешение, пользователь получает доступ к объекту.

Разрешения объекта достаточно детализированы; существуют отдельные разрешения для каждого из возможных действий (SELECT, INSERT, UPDATE, RUN и т.д.) над объектом. Некоторые фиксированные роли базы данных также влияют на доступ к объекту, например, управляя возможностью считывать информацию и записывать ее в базу данных.

Вполне возможна ситуация, когда пользователь был распознан в SQL Server, но у него нет доступа ни к одной из его баз данных. Также возможно и обратное: пользователю открыт доступ к базам данных, но он не был распознан сервером. Перемещение базы данных и ее разрешений на другой сервер без параллельного перемещения регистрационных записей сервера может привести к возникновению таких "осиротевших" пользователей.

Права собственности на объект

Заключительным аспектом в этом обзоре модели безопасности SQL Server является право собственности на объект. *Владельцем* любого объекта является схема. Схемой по умолчанию является dbo, которую не следует путать с ролью dbo.



В предыдущих версиях владельцами объектов являлись пользователи (точнее, владелец сам представлял собой схему). По сравнению с этим модель SQL Server 2005 в виде *база данных–схема–объекты* имеет целый ряд преимуществ.

Права собственности становятся особенно критичными, когда пользователю предоставляется разрешение на запуск хранимой процедуры, но у него нет прав на таблицы, с которыми работает эта процедура. Если цепочка принадлежности от таблиц до хранимой процедуры согласована, то пользователь имеет право доступа к хранимой процедуре. а сама хранимая процедура имеет доступ к таблицам как ее владелец. Однако если цепочка принадлежности разорвана (т.е. между хранимой процедурой и таблицей существует объект с другим владельцем), то пользователь должен иметь разрешения как к хранимой процедуре, так и к таблицам, и ко всем объектам, находящимся в цепочках между хранимой процедурой и таблицами.

Большая часть операций управления системой безопасности выполняется в утилите Management Studio. В программном коде управление системой безопасности выполняется с помощью инструкций DDL GRANT, REVOKE и DENY, а также нескольких хранимых процедур.

Система безопасности Windows

Поскольку SQL Server работает в операционной среде Windows, одним из аспектов системы безопасности должна быть защита самой операционной системы.

Система безопасности Windows

Базы данных SQL Server часто поддерживают Web-сайты. Таким образом, в общий план обеспечения безопасности следует включить систему безопасности IIS (Internet Information Server) и брандмауэры.

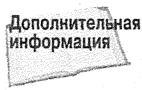
Система безопасности Windows представляет собой отдельную, обширную тему, так что ее рассмотрение выходит за рамки настоящей книги. Если как администратор базы данных вы не обеспечены поддержкой высококвалифицированного персонала, обслуживающего локальную сеть, то вам самому следует приложить определенные усилия, чтобы стать профессионалом в технологиях Windows Server, особенно в области системы безопасности.

Регистрационная запись SQL Server

Не следует путать регистрационные записи SQL Server с учетными записями Windows в этом сервере баз данных. Эти два типа регистрации совершенно отличны.

Пользователям SQL Server не нужен доступ к каталогам, где хранятся файлы базы данных, или к файлам данных уровня Windows, так как реальный доступ к файлам осуществляет процесс SQL Server, а не сам пользователь. В то же время права доступа к файлам нужны самому процессу SQL Server, следовательно, ему нужна учетная запись Windows. И здесь доступны два варианта.

- **Учетная запись локального администратора**. SQL Server может использовать эту учетную запись для получения доступа к компьютеру, на котором установлен. Этот вариант идентичен установке на обособленный сервер — он не способен поддерживать сетевую систему безопасности, необходимую для распределенной обработки.
- **Учетная запись пользователя домена (рекомендуется)**. SQL Server может использовать учетную запись пользователя домена Windows, созданную специально для него. Учетной записи пользователя SQL Server могут быть назначены административные привилегии для локального сервера, и посредством него он может получить доступ к сети для поддержания взаимодействия с другими серверами.



Учетные записи SQL Server изначально конфигурируются в процессе установки сервера баз данных. О процессе инсталляции SQL Server см. в главе 4.

Безопасность сервера

SQL Server использует двухэтапную схему аутентификации. Вначале пользователь аутентифицируется сервером. Как только он будет "находиться внутри сервера", права доступа ему предоставляются отдельными базами данных.

SQL Server хранит всю информацию о регистрационных записях в базе данных `master`.

Режимы аутентификации в SQL Server

При установке SQL Server одним из решений, которые следует принять, является выбор используемого метода аутентификации.

- В **режиме аутентификации Windows** SQL Server полностью доверяет аутентификацию операционной системе.
- В **смешанном режиме** аутентификация Windows и самого сервера сосуществуют независимо друг от друга.

Установленный при инсталляции режим аутентификации можно изменить на странице Security диалогового окна SQL Server Properties в утилите Management Studio (рис. 40.2).

В программном коде установленный режим аутентификации можно проверить с помощью системной хранимой процедуры `xp_loginconfig` следующим образом:

```
EXEC xp_loginconfig 'login mode'
```

Вот пример результата этой процедуры:

```
name         config_value
------------ --------------------
login mode   Mixed
```

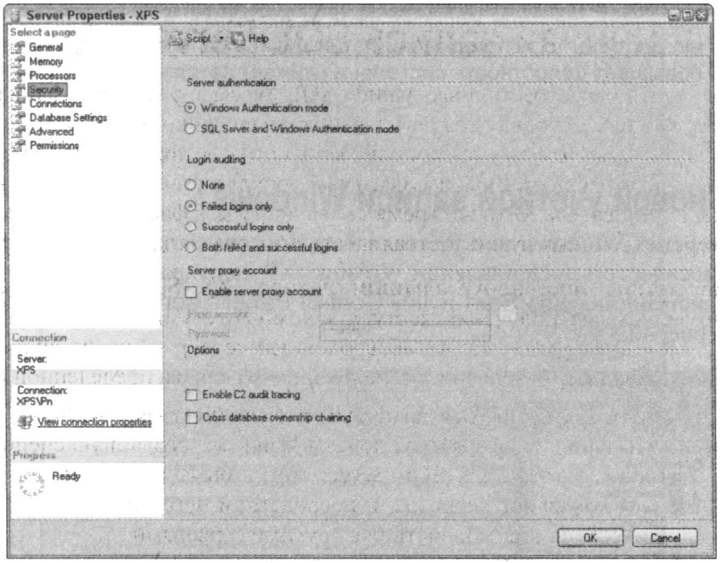


Рис. 40.2. Система безопасности уровня сервера конфигурируется во вкладке Security *диалогового окна* SQL Server Properties

Обратите внимание на то, что хранимая процедура, выводящая отчет об установленном режиме аутентификации, является расширенной. Это связано с тем, что режим аутентификации хранится в реестре Windows в следующем ключе:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
MicrosoftSQLServer\<имя_экземпляра>\MSSQLServer\LoginMode
```

Значением этого ключа является `0` для режима аутентификации Windows и `1` — для смешанного режима.

Установить режим аутентификации можно либо в Management Studio, либо с помощью редактора реестра Windows `regedit.exe`.

Аутентификация Windows

Аутентификация Windows имеет преимущества перед смешанным режимом, так как пользователю не приходится запоминать еще один пароль. Это делает систему безопасности сети более универсальной.

Использование аутентификации Windows означает, что пользователю для доступа к SQL Server достаточно иметь учетную запись Windows. Идентификатор безопасности Windows (SID) передается из операционной системы на сервер баз данных.

Аутентификация Windows очень устойчивая — она распознает не только пользователей Windows, но и их группы.

Когда имя группы Windows передается в SQL Server в качестве регистрационной записи, любой член этой группы может быть аутентифицирован сервером баз данных. Права доступа, роли и разрешения могут назначаться группе Windows, после чего они будут применены ко всем пользователям в этой группе.

SQL Server также известно истинное имя пользователя Windows, вследствие чего приложение может выполнять аудит на уровне пользователей, а также на уровне групп.



Если пользователи Windows уже организованы в группы по функции или уровню доступа, то использование этих групп в качестве пользователей SQL Server повышает целостность системы и снижает затраты на ее администрирование.

Создание новой учетной записи Windows

В разных версиях Windows пользователи и группы создаются и управляются различным образом. В классическом представлении Windows XP для управления пользователями нужно выбрать в системном меню Start пункт Control Panel⇨Administrative Tools⇨Computer Management (рис. 40.3).

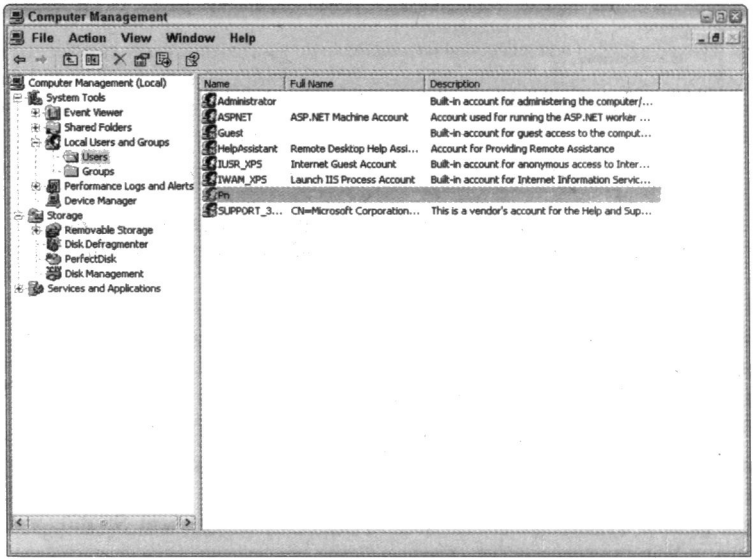


Рис. 40.3. Учетные записи пользователей Windows создаются и назначаются группам с помощью консоли Computer Management

Если учетная запись пользователя уже существует в списке пользователей компьютера или домена Windows, то SQL Server может его распознать. Чтобы добавить новую учетную запись для SQL Server с помощью Object Explorer, выполните следующие действия.

1. Откройте узел `Security⇨Logins` сервера и выберите в контекстном меню пункт New Login.
2. На странице General диалогового окна Login-New (рис. 40.4) с помощью кнопки Search найдите пользователя Windows.
3. Вы также имеете возможность ввести имя пользователя или группы вручную или воспользоваться кнопкой Advanced для выполнения расширенного поиска.

Пользователю может быть назначена база данных, к которой он подключается по умолчанию. Ее имя выбирается в раскрывающемся списке Default Database в нижней части окна. Следует отметить, что назначение базы данных по умолчанию никак не связано с назначением для доступа к ней определенных прав. Права доступа к базам данных назначаются во вкладке Database Access. (О доступе к базам данных мы поговорим в следующем разделе.)

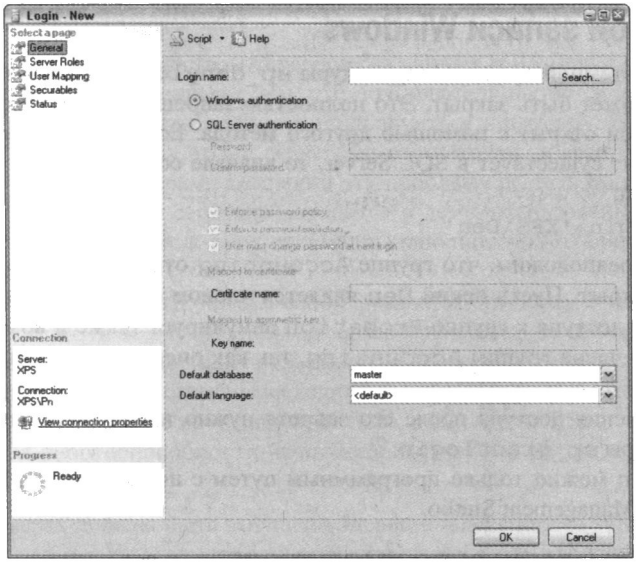
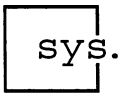


Рис. 40.4. Страница General *диалогового окна* Login-New *используется для создания и редактирования регистрационной записи пользователя на уровне сервера*

Для создания учетной записи пользователя или группы Windows в SQL Server в программном коде используется системная хранимая процедура `sp_grantlogin`. Обязательно передавайте ей в качестве аргумента полное имя пользователя, включающее имя домена, как в следующем примере:

```
EXEC sp_grantlogin 'XPS\Den'
```



Для просмотра учетных записей Windows из программного кода выполните запрос к представлению каталога `sysserver_principals`.

Окно Login-New также используется для управления существующими пользователями. Для открытия версии этого окна, регулирующего разрешения существующего пользователя, дважды щелкните на его имени в узле `Security⇨Logins` или щелкните правой кнопкой мыши и выберите в контекстном меню пункт Properties.

Удаление учетной записи Windows

Удаление учетной записи Windows из SQL Server достаточно просто выполняется в Management Studio. Выберите эту учетную запись в Object Browser и с помощью контекстного меню удалите ее (т.е. выполните команду Delete).

Для удаления учетной записи или группы Windows из SQL Server программным путем используется системная хранимая процедура `sp_revokelogin`:

```
EXEC sp_revokelogin 'XPS\Den'
```

Разумеется, эта операция не удаляет данную учетную запись из операционной системы — только из сервера баз данных.

Запрет учетной записи Windows

С помощью системной хранимой процедуры sp_denylogin доступ любого пользователя в SQL Server может быть закрыт. Это полностью запрещает доступ пользователя к SQL Server, даже если он открыт с помощью другого метода. Если заданная в аргументе группа или пользователь не существует в SQL Server, то вначале создается такая запись, а затем ей запрещается доступ:

```
EXEC sp_denylogin 'XPS\Den'
```

Для примера предположим, что группе Accounting открыт доступ к серверу, а группе Probation — закрыт. Пусть некий Den является членом обеих этих групп. В данной ситуации запрет ему доступа к группе Probation аннулирует также и возможность его доступа к серверу как члена группы Accounting, так как операция deny имеет превосходство над grant.

Для восстановления доступа после его запрета нужно предоставить доступ с помощью хранимой процедуры sp_grantlogin.

Отозвать доступ можно только программным путем с помощью T-SQL. Эта функция не поддерживается в Management Studio.

Установка базы данных по умолчанию

Установка базы данных, используемой по умолчанию, выполняется в разделе Login Properties страницы General. В программном коде для этого используется системная хранимая процедура sp_defaultdb:

```
EXEC sp_defaultdb 'Sam', 'OBXKites'
```

"Осиротевшие" пользователи Windows

Если пользователь Windows был добавлен в SQL Server, а затем был удален из домена Windows, он продолжает существовать как пользователь в севере баз данных, но считается уже *осиротевшим*. Это значит, что, несмотря на то что пользователь формально имеет доступ к серверу, он не имеет доступа к ресурсам сети и, следовательно, ко всему комплексу средств SQL Server.

Системная хранимая процедура sp_validatelogins позволяет найти "осиротевших" пользователей и возвращает их идентификаторы системы безопасности Windows NT и имена учетных записей. В следующем примере пользователю Den был открыт доступ к SQL Server, после чего его учетная запись была удалена из Windows:

```
EXEC sp_validatelogins

SID                                              NT Login
------------------------------------------------ ----------
0x0105000000000005150000000FCE31531A931...       XPS\Joe
```

Это нельзя считать брешью в системе безопасности. Не имея учетной записи Windows с таким идентификатором, пользователь не сможет подключиться к SQL Server.

Для разрешения проблемы "осиротевших" пользователей можно использовать следующую процедуру.

1. Отзовите права доступа пользователя ко всем базам данных с помощью хранимой процедуры sp_revokedbaccess.
2. Отзовите право доступа данного пользователя к серверу с помощью sp_revokelogin.
3. Создайте для пользователя новую учетную запись.

Делегирование полномочий

В корпоративной сети с множеством серверов баз данных и IIS могут возникнуть проблемы подключения к серверу, связанные с тем, что пользователь пытается подключиться к одному из них, а тот, в свою очередь, пытается подключиться к другому. Причиной этой проблемы может быть то, что между этими серверами отсутствуют доверительные отношения. При работе с внутренними серверами компании эту проблему решить просто. Гораздо сложнее дело обстоит, когда один из серверов находится в демилитаризованной зоне Интернета (DMZ), и администратору сети не с руки открывать дополнительную брешь в системе безопасности локальной сети.

Делегирование полномочий представляет собой функцию SQL Server 2005, которая для передачи профилей безопасности между связанными серверами использует протокол Kerberos.

Например, пользователь может получить доступ к серверу IIS, который, в свою очередь, может получить доступ к SQL Server, и последний увидит имя пользователя, даже несмотря на то, что запрос на подключение поступил не от него, а от сервера IIS.

Для обеспечения работоспособности протокола *Kerberos* необходимо выполнение ряда условий.

- На всех серверах должна быть запущена операционная система Windows 2000 или более поздняя, к тому же в домене или доверительном дереве должна быть установлена служба активных каталогов.
- В учетной записи пользователя не должен быть установлен параметр запрета делегирования полномочий (Account is sensitive and cannot be delegated).
- В учетной записи службы SQL Server должен быть установлен параметр доверия к делегированию полномочий (Account is trusted for delegation).
- На сервере, на котором запускается SQL Server, также должен быть установлен параметр доверия к делегированию полномочий (Computer is trusted for delegation).
- Экземпляр SQL Server должен иметь строгое имя службы (SPN), созданное с помощью утилиты `setspn.exe`, доступной в Windows 2000 Resource Kit.

Делегирование профиля безопасности достаточно тяжело настроить, и это может потребовать привлечения к работе администратора домена сети. В то же время распознавание пользователей, проходящих через сервер IIS, является мощным механизмом системы безопасности.

Регистрационные записи SQL Server

Наличие дополнительных регистрационных записей SQL Server уместно тогда, когда аутентификация Windows не доступна или не подходит для создаваемой системы. Эта регистрация имеет обратную совместимость с предыдущими версиями сервера, а также с приложениями, в которых регистрационная запись встроена в программный код.



При внедрении регистрационных записей SQL Server (т.е. смешанного режима аутентификации) автоматически создается пользователь `sa`, являющийся членом фиксированной серверной роли `sysadmin` и имеющий все права доступа к серверу. Зачастую ему забывают присвоить пароль, и эти ворота в системе безопасности открывают путь к серверу любому взломщику. Как правило, наличие такой бреши хакеры проверяют в первую очередь. Исходя из этого, прежде всего нужно отключить пользователя `sa` и назначить административной роли `sysadmin` какого-то другого пользователя (или создать дополнительные роли с административными привилегиями).

Для управления пользователями SQL Server в Management Studio используется то же диалоговое окно Login-New, что и для управления пользователями Windows, однако в данном случае выбирается режим SQL Server Authentication.

В программном коде для этого используется системная хранимая процедура sp_addlogin. Так как в данном случае требуется настройка пользователя, а не просто выбор его из списка, данная задача более сложная, чем выполнение процедуры sp_grantlogin. Среди передаваемых процедуре аргументов обязательным является только имя регистрационной записи.

```
sp_addlogin 'имя', 'пароль', 'база_по_умолчанию',
'язык_по_умолчанию', 'идентификатор_пользователя_сервера',
'параметр_шифрования'
```

Например, следующий программный код создает пользователя SQL Server с именем Den и назначает ему в качестве базы данных по умолчанию учебную базу OBXKites:

```
EXEC sp_addlogin 'Den', 'myoldpassword', 'OBXKites'
```

Параметр шифрования skip_encryption указывает серверу хранить пароль в системной таблице sysxlogins безо всякого шифрования. В то же время SQL Server ожидает, что пароль обязательно будет зашифрован, поэтому он не распознает созданный таким образом пароль. Избегайте использования этого параметра.

Идентификатор пользователя сервера (SID) — это 85-битовое двоичное значение, которое SQL Server использует для идентификации пользователя. Если некоторый пользователь создается на двух серверах как один и тот же, то для второго сервера следует явно задать идентификатор SID, присвоенный первым сервером. Для получения идентификатора SID используется представление каталога sysserver_principals:

```
SELECT Name, SID
  FROM sysserver_principals
  WHERE Name = 'Den'

Name       SID
---------  ----------------------------------
Den        0x1EFDC478DEB52045B52D241B33B2CD7E
```

Обновление пароля

Пароль может быть изменен с помощью системной хранимой процедуры sp_password, например:

```
EXEC sp_password 'myoldpassword', 'mynewpassword', 'Den'
```

Если пароль пуст, то в хранимую процедуру вместо пустой строки (' ') передается ключевое слово NULL.

Удаление регистрационной записи

Для удаления регистрационной записи SQL Server используется системная хранимая процедура sp_droplogin, например:

```
EXEC sp_droplogin 'Joe'
```

Удаление учетной записи приводит к удалению и всех ее настроек безопасности.

Установка базы данных, используемой по умолчанию

База данных, используемая по умолчанию, задается на странице General диалогового окна Login Properties так же, как и для пользователя Windows. В программном коде для этого используется системная хранимая процедура sp_defaultdb:

```
EXEC sp_defaultdb 'Den', 'OBXKites'
```

Серверные роли

SQL Server содержит только фиксированные, предопределенные серверные роли. В основном эти роли открывают доступ к административным задачам, связанным с сервером баз данных. Один пользователь может принадлежать к нескольким ролям.

Следующие роли лучше всего использовать для выполнения конкретных задач администрирования.

- Члены роли `Bulk admin` могут выполнять операции массовой вставки.
- Члены роли `Dbcreators` могут создавать, изменять, удалять и восстанавливать базы данных.
- Члены роли `Diskadmin` могут создавать, изменять и удалять файлы на диске.
- Члены роли `Processadmin` могут прерывать запущенный процесс SQL Server.
- Члены роли `Securityadmin` могут управлять регистрационными записями сервера.
- Члены роли `Serveradmin` могут конфигурировать параметры сервера, включая настройки полнотекстового поиска и останов сервера.
- Члены роли `Setupadmin` могут конфигурировать связанные серверы, расширенные хранимые процедуры и процедуры автозапуска.
- Члены роли `Sysadmin` могут осуществлять любую деятельность в инсталляции SQL Server, независимо от наличия других разрешений. Права роли `sysadmin` даже замещают собой запрет доступа к отдельным объектам.

  SQL Server автоматически создает пользователя `BUILTINS/Administrators`, который включает всех пользователей Windows в группу `Windows Admins` и назначает этой группе роль на сервере баз данных `sysadmin`. Пользователя `BUILTINS/Administrators` лучше удалить или изменить по своему усмотрению.

  Если в SQL Server устанавливается смешанный режим аутентификации, то автоматически создается пользователь `sa`, принадлежащий роли `sysadmin`. Этот пользователь оставлен в SQL Server 2005 только из соображений обратной совместимости с предыдущими версиями сервера.



Отключите или переименуйте пользователя `sa` или по крайней мере назначьте ему пароль, однако не используйте его при работе администраторов и разработчиков баз данных. К тому же удалите пользователя `BUILTINS/Administrors`. Вместо этого лучше использовать аутентификацию Windows и назначить всех разработчиков и администраторов серверной роли `sysadmin`.

Чтобы все доступные возможности, предоставляемые ролью `sysadmin`, вступили в силу, пользователь должен повторно зайти на сервер.

Серверные роли настраиваются в Management Studio на странице Server Roles диалогового окна Login Properties (рис. 40.5).

В программном коде серверная роль назначается пользователю с помощью следующей системной хранимой процедуры:

```
sp_addsrvrolemember
  [ @loginame = ] 'регистрационная_запись',
  [ @rolename = ] 'роль'
```

Например, в следующем программном коде роль `sysadmin` назначается регистрационной записи `XPS\Lauren`:

```
EXEC sp_addsrvrolemember 'XPS\Lauren', 'sysadmin'
```

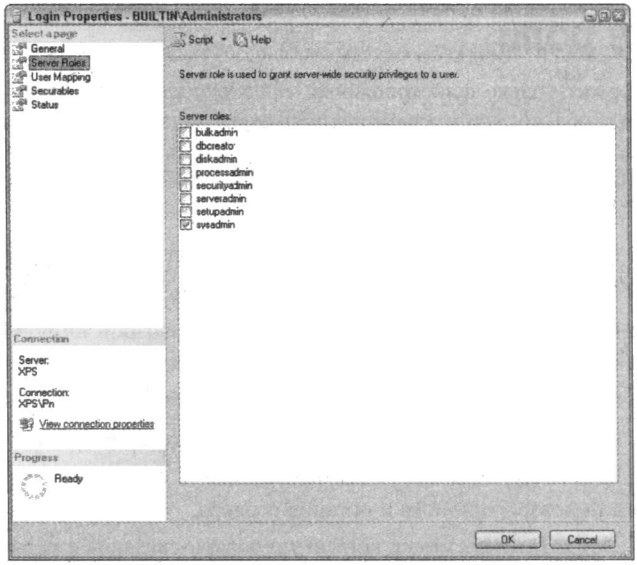
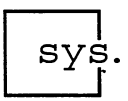


Рис. 40.5. Страница Server Roles *используется для назначения пользователям прав администрирования сервера. В данном примере группе* `Windows Admin` *назначена роль* `sysadmin`

Для удаления членства учетной записи в некоторой серверной фиксированной роли используется следующая процедура

```
EXEC sp_dropsrvrolemember 'XPS\Lauren', 'sysadmin'
```



Для просмотра назначенных ролей в программном коде выполните запрос к представлению каталога `sysserver_principals`, объединенному с `sysserver_role_members`.

Безопасность базы данных

После получения доступа к серверу пользователь может получить доступ к отдельным базам данных. Система безопасности баз данных относительно сложная.

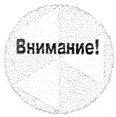
Изначально доступ к базе данных устанавливается либо путем добавления базы данных пользователю, либо добавления пользователя базе данных.

Гостевые учетные записи

Любой пользователь, желающий получить доступ к базе данных, но для которого в ней не создана учетная запись, автоматически получает привилегии гостя, если в базе данных существует учетная запись `guest` (см. рис. 40.1).

При создании базы данных учетная запись `guest` автоматически не создается — ее нужно добавлять в базу данных вручную с помощью программного кода. При этом регистрационную запись `guest` не обязательно определять для сервера баз данных:

```
EXEC sp_adduser 'Guest'
```



Будьте очень осторожны с использованием гостевой учетной записи. Несмотря на то что в некоторых случаях удобнее открывать доступ к базе пользователям без обязательной их регистрации в системе безопасности базы данных, следует учитывать, что разрешения, предоставленные пользователю с учетной записью `guest`, применяются *абсолютно ко всем* пользователям.

Пользователь с учетной записью `guest` должен быть обязательно удален из базы данных, как только в гостевом доступе отпадает необходимость.

Предоставление доступа к базе данных

Доступ пользователям к любой базе данных должен открываться в явном виде. Так как эта схема представляет собой отношение "многие ко многим", доступом к базе можно управлять как на уровне регистрационных записей, так и на уровне баз данных.

Если некоторой учетной записи открыт доступ к базе данных, то ей может быть назначена учетная запись базы. Если ее имя совпадает с именем регистрационной записи сервера, то ей можно назначить другое имя — оно будет служить чем-то вроде псевдонима регистрационной записи, который распознает база данных.

Для предоставления доступа к базе данных на уровне регистрационных записей в Object Explorer используется страница User Mappping диалогового окна Login Properties (рис. 40.6).

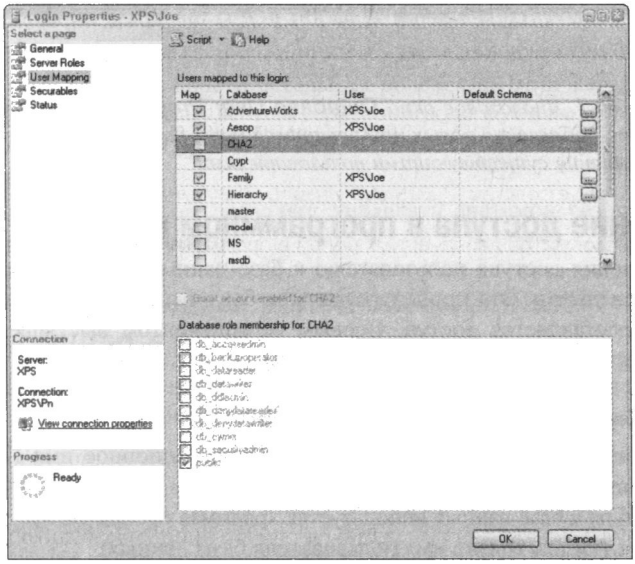


Рис. 40.6. Диалоговое окно Login Properties *используется для предоставления доступа к базе данных регистрационной записи и для назначения ей ролей*



Многие настройки системы безопасности включают в себя множества объектов, таких как пользователи и базы данных, или роли и разрешения объектов. Эти настройки можно установить в разных местах: в диалоговом окне Login Permission, в роли или на странице свойств объекта.

Для предоставления доступа пользователям со стороны базы данных используйте команду New User контекстного меню в узле `Database⇨Security⇨Users`. В поле Login Name открывшегося диалогового окна Database User-New (рис. 40.7) введите имя добавляемой

регистрационной записи. Для поиска регистрационных записей используется кнопка с многоточием. В поле User Name вводится имя пользователя, под которым он будет известен базе данных.

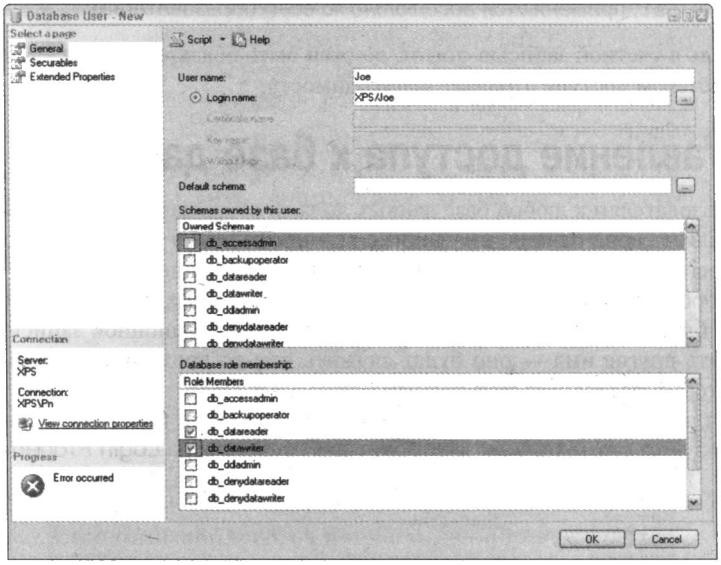


Рис. 40.7. Диалоговое окно Database User Properties *используется для добавления новых пользователей базы данных, а также для управления существующими пользователями*

Предоставление доступа в программном коде

Для предоставления доступа пользователю к базе данных используется хранимая процедура `sp_grantdbaccess`. Эта процедура должна быть вызвана из базы данных, к которой пользователю предоставляется доступ. Первым ее аргументом выступает регистрационная запись сервера, вторым — имя учетной записи базы данных, если оно отличается:

```
USE Family
EXEC sp_grantdbaccess 'XPS\Lauren', 'LRN'
```

После выполнения этой хранимой процедуры регистрационное имя `Lauren` появится в списке пользователей базы данных как `LRN`.

Для снятия доступа к базе данных используется хранимая процедура `sp_revokedbaccess`. При этом требуется указать только имя пользователя базы данных:

```
USE Family
EXEC sp_revokedbaccess 'LRN'
```



Для получения списка пользователей базы данных программным путем создайте запрос к представлению каталога `sysdatabase_principals`.

Фиксированные роли базы данных

SQL Server содержит несколько стандартных (или фиксированных) ролей базы данных. Подобно серверным фиксированным ролям, в основном они предназначены для выполнения

административных задач. Один пользователь может принадлежать нескольким ролям. Среди фиксированных ролей базы данных стоит упомянуть следующие.

- Роль db_accessadmin позволяет авторизировать пользователя для доступа к базе данных, но не для управления системой безопасности уровня базы.
- Роль db_backupoperators позволяет выполнять задачи установки контрольных точек, создания резервных копий, запуска команды DBCC. В то же время она не открывает возможности восстановления базы данных — это разрешено только членам роли sysadmin.
- Роль db_datareaders позволяет читать все данные в базе. Эта роль является эквивалентом разрешения доступа ко всем объектам базы и может быть замещена разрешением deny отдельных объектов.
- Роль db_datawriters позволяет записывать данные в базу. Эта роль является эквивалентом разрешения доступа ко всем объектам базы и может быть замещена разрешением deny отдельных объектов.
- Роль db_ddladmins позволяет выполнять инструкции DDL (CREATE, ALTER и DROP).
- Роль db_denydatareaders блокирует чтение из всех таблиц базы данных. Этот запрет замещает собой любые разрешения на уровне объектов.
- Роль db_denydatawriters блокирует изменение данных во всех таблицах базы данных. Этот запрет замещает собой любые разрешения на уровне объектов.
- Роль db_owner открывает полный доступ ко всем объектам базы данных. Эта роль включает в себя все возможности остальных ролей. Она отличается от роли пользователя dbo и не является эквивалентом серверной роли sysadmin на уровне базы данных, поскольку запрет на уровне объектов замещает собой предоставленное ролью db_owner разрешение.
- Роль db_securityadmins позволяет управлять системой безопасности базы данных — ролями и разрешениями.

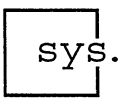
Назначение фиксированных ролей базы данных в Management Studio

Фиксированную роль базы данных можно назначить пользователю в графической среде Management Studio. Для этого можно использовать любую из процедур, описанных ниже.

- Роль пользователю можно назначить в диалоговом окне Database User Properties (см. рис. 40.7), как в случае его создания, так и в случае его существования.
- Роль пользователю можно назначить в диалоговом окне Database Role Properties (рис. 40.8). Чтобы открыть это окно, щелкните правой кнопкой мыши на папке Roles в узле Security базы данных и выберите в контекстном меню пункт Properties.

Назначение пользователей фиксированным ролям базы данных программным путем

В программном коде добавить пользователя в фиксированную роль базы данных можно с помощью хранимой процедуры sp_addrole.



Для просмотра пользователей, назначенных роли, выполните запрос к представлению каталога sysdatabase_role_members, объединенному с sysdatabase_principal.

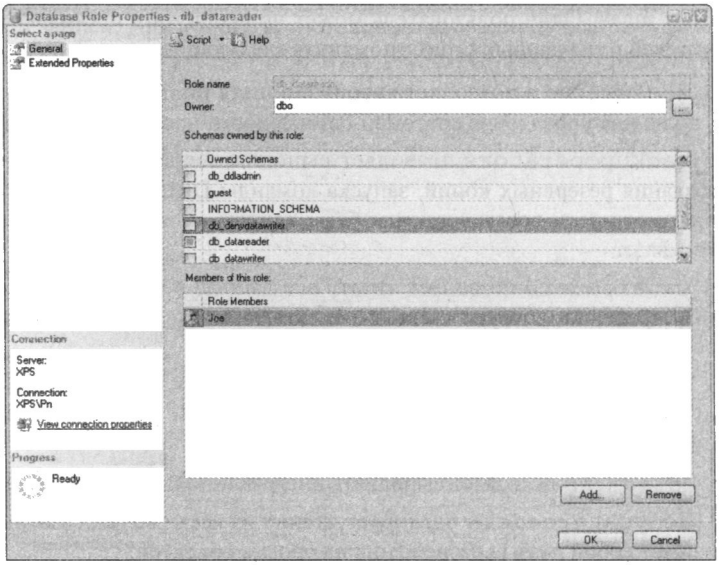


Рис. 40.8. В диалоговом окне Database Role Properties *перечислены все пользователи, назначенные текущей роли. Назначение и открепление пользователей от роли выполняется с помощью кнопок* Add *и* Remove

Разрешения защищаемых объектов

Пользователю базы данных могут быть предоставлены конкретные *разрешения* к определенным защищаемым объектам. Для этого используется страница Securables диалогового окна Database User (рис. 40.9).

Способ конфигурирования разрешений зависит от типа объекта. Каждая база данных представляет собой защищаемый объект и имеет ряд специфичных только для нее разрешений.

Если у вас нет особых оснований регулировать доступ на уровне отдельных инструкций, лучше всего управлять задачами администрирования базы данных с помощью фиксированных ролей базы данных.

Инструкции GRANT, REVOKE и DENY мы подробно рассмотрим в следующих разделах.

Роли приложений

Ролью приложения называется специфичная для базы данных роль, предназначенная для предоставления приложению определенных полномочий, независимо от прав, которыми обладает работающий с ним пользователь. Например, если некоторая программа Visual Basic используется для поиска в таблице `Customer`, к которой не открыт доступ конкретным пользователям, программа может использовать запрограммированную роль приложения для получения доступа к ресурсам SQL Server.



Так как за роль приложения придется расплачиваться потерей контроля над идентичностью пользователя, я выступаю решительно против использования ролей приложений.

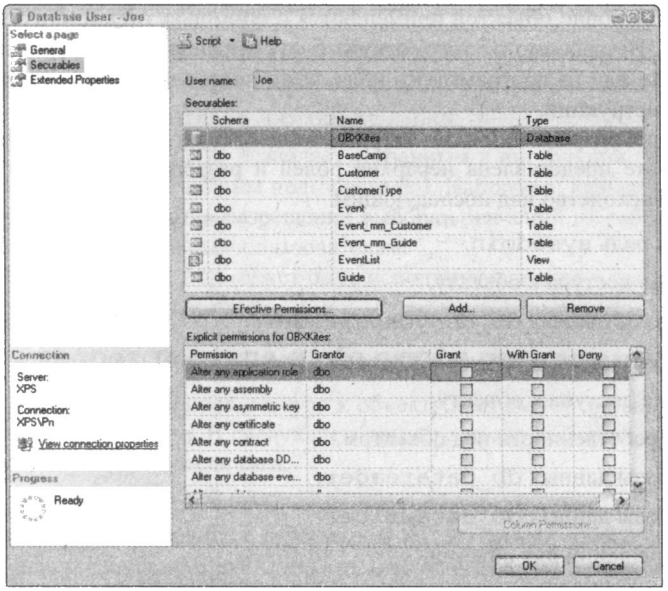


Рис. 40.9. Страница **Securables** *используется для конфигурирования разрешений к отдельным объектам*

Безопасность объектов

Если пользователю открывается доступ к базе данных, ему автоматически предоставляются разрешения ко всем ее объектам. Эти разрешения могут быть открыты непосредственно для пользователя; также они могут быть назначены роли, к которой принадлежит пользователь. Пользователь может быть назначен нескольким ролям, поэтому от пользователя к объекту может существовать множество цепочек разрешений.

Разрешения уровня объекта

Разрешения к объекту назначаются с помощью инструкций SQL DCL `GRANT`, `REVOKE` и `DENY`. Эти инструкции имеют свою иерархию: `DENY` имеет превосходство над `GRANT`, которая, в свою очередь, имеет превосходство над `REVOKE`, т.е. инструкция `GRANT` разрешает доступ к объекту, если он не отменен инструкцией `DENY`.

Существует несколько особых типов разрешений.

- **Разрешение на отбор данных**. Эти разрешения могут применяться к отдельным столбцам.
- **Разрешение на вставку данных**.
- **Разрешение на обновление данных**. Право на изменение существующих данных требует наличия права на их отбор. Разрешения на обновление можно устанавливать на уровне отдельных столбцов.
- **Право на удаление существующих данных**.
- **Право на создание внешних ключей, использующих DRI**.

Разрешения на уровне объектов применяются с помощью трех инструкций DCL: GRANT, REVOKE и DENY. Независимо от того, откуда будут управляться разрешения, из интерфейса Management Studio или из программного кода, важно четко понимать особенности и взаимосвязь этих трех инструкций.

Предоставление разрешений к объекту взаимодействует с серверными ролями и ролями базы данных. Ниже представлена иерархия ролей и разрешений, где каждый предыдущий номер имеет превосходство над последующим.

1. Серверная роль `sysadmin`.
2. Запрещение доступа к объекту,

   или роль базы данных `db_denydatareader`,

   или роль базы данных `db_denydatawriter`.
3. Разрешение доступа к объекту,

   или права собственности над объектом,

   или роль базы данных `db_datareader`,

   или роль базы данных `db_datawriter`.
4. Отзыв разрешений для объекта.



Существует простой способ тестирования системы безопасности. Для этого нужно сконфигурировать на севере смешанный режим аутентификации и создать тестовую учетную запись SQL Server. С помощью анализатора запросов можно создать дополнительные подключения как отдельных пользователей — это гораздо проще, чем каждый раз заново регистрироваться на сервере в Management Studio или каким-либо другим способом.

Если в среде, в которой вы работаете, запрещено использование совместного режима, для проверки системы безопасности щелкните правой кнопкой мыши в меню Пуск на ярлыках утилиты Management Studio или Query Analyzer, выберите в контекстном меню команду Run As и укажите имя пользователя, от имени которого вы собираетесь запустить утилиту. Однако учтите, что это влечет за собой создание подставных пользователей в домене Windows.

Предоставление разрешений из программного кода

Настройка разрешений к объектам представляет собой единственную область системы безопасности, в которой можно обойтись без вызова хранимых процедур.

```
GRANT разрешение, разрешение
  ON объект
  TO пользователь/роль, пользователь/роль
  WITH GRANT OPTION
```

Разрешение может быть одним из следующих ключевых слов: ALL, SELECT, INSERT, DELETE, REFERENCES, UPDATE и EXECUTE. *Роль* и *пользователь* указывают на имя пользователя базы данных, на пользовательскую или стандартную публичную роль. В следующем примере пользователю Den предоставляется разрешение на выборку значений из таблицы Person:

```
GRANT Select ON Person TO Den
```

В следующем примере для роли public открываются все разрешения к таблице Marriage:

```
GRANT All ON Marriage TO Public
```

В одной инструкции может быть перечислено несколько имен пользователей и/или ролей, а также множество разрешений к объекту. В следующем примере пользователям guest и LRN предоставляется разрешение на отбор и обновление значений из таблицы Person:

```
GRANT Select, Update ON Person to Guest, LRN
```

Параметр WITH GRANT OPTION позволяет указанным в инструкции GRANT пользователям передавать свои полномочия другим пользователям. Например, в следующей инструкции пользователю Joe предоставляются разрешения выборки значений из таблицы Person, которое он может передавать остальным пользователям.

Параметр WITH GRANT OPTION может быть использован только в программном коде — в среде Management Studio он недоступен.

Отзыв прав и запрет доступа к объекту в программном коде

Инструкции отзыва и запрета прав доступа к объекту используют в основном тот же синтаксис, что и предоставление прав. В следующем примере право выборки данных из таблицы Marriage отзывается у пользователя Den:

```
REVOKE Select ON Marriage TO Den
```

Если разрешения предоставлялись с параметром WITH GRANT OPTION, то они должны отзываться и запрещаться с параметром CASCADE, чтобы операция была применена и ко всем пользователям, которым были делегированы права данным пользователем. В следующем примере запрещается доступ к таблице Person пользователю Joe, а также всем, кому он сам предоставил разрешения.

Стандартные роли базы данных

Стандартные роли базы данных иногда называют пользовательскими. Они могут быть созданы любым пользователем сервера, имеющим серверную роль sysadmin или роли базы данных db_owner или securityadmins. Эти роли функционально идентичны группам пользователей в операционной системе Windows. Стандартной роли могут быть предоставлены разрешения и принадлежности другим ролям, и этой роли могут принадлежать пользователи.



Самый "прозрачный" план защиты SQL Server предполагает назначение разрешений к объектам и фиксированных ролей стандартной роли и последующее назначение стандартной роли пользователей.

Роль public

Роль public является фиксированной, однако, подобно стандартной роли, ей можно назначать разрешения к объектам. Каждый пользователь автоматически становится членом роли public, и эту связь невозможно удалить. Таким образом, роль public служит своеобразным мерилом минимально допустимых разрешений.



При назначении разрешений роли public будьте особо осторожны, поскольку они будут применены ко всем пользователям, за исключением тех, кто принадлежит серверной роли sysadmin. Предоставление прав скажется на всех пользователях, однако более важно то, что на них скажутся и запреты доступа, примененные к роли public (как мы уже говорили, это не касается роли sysadmin), — в том числе и на владельцах самих этих объектов.

Управление ролями в программном коде

Создание стандартных ролей в программном коде предполагает использование системной хранимой процедуры sp_addrole. Имя роли может состоять не более чем из 128 символов, не может включать в себя обратную косую черту, не может быть пустой строкой или пустым значением. По умолчанию роли принадлежат пользователю dbo, в то же время владельца роли можно определить явно во втором параметре хранимой процедуры. В следующем примере создается роль manager:

```
EXEC sp_addrole 'Manager'
```

Вот результат:

```
New role added.
```

Операцией, противоположной созданию роли, является ее удаление. Роль не может быть удалена, пока ей назначен хотя бы один пользователь. Для удаления роли из базы данных используется хранимая процедура sp_droprole:

```
EXEC sp_droprole 'Manager'
```

Вот результат:

```
Role dropped.
```

После создания роли она может быть назначена пользователям с помощью хранимой процедуры sp_addrolemember. В следующем примере роль manager назначается пользователю Den:

```
EXEC sp_addrolemember 'Manager', Den
```

Вот результат:

```
'Joe' added to role 'Manager'.
```

Аналогично системная хранимая процедура sp_droprolemember удаляет назначение пользователя некоторой роли. В следующей команде пользователь Den будет освобожден от роли manager:

```
EXEC sp_droprolemember 'Manager', Den
```

Вот результат:

```
'Joe' dropped from the role 'Manager'.
```

Иерархическая структура ролей

Если структура системы безопасности сложная, то при организации структуры разрешений можно воспользоваться иерархической структурой стандартных ролей базы данных.

- Рабочий может иметь ограниченный доступ.
- Управляющий может иметь все права рабочего плюс некоторые права к таблицам классификаторов.
- Администратор может иметь все права управляющего плюс право на выполнение задач администрирования базы данных.

Для создания структуры такого типа выполните следующие действия.

1. Создайте роль рабочего и предоставьте ей разрешения.
2. Создайте роль управляющего и добавьте ее как пользователя роли рабочего. Предоставьте роли управляющего дополнительные разрешения.
3. Создайте роль администратора и добавьте ее как пользователя роли управляющего. Предоставьте роли управляющего дополнительные фиксированные серверные роли.

Преимущество такой системы организации системы безопасности заключается в том, что изменения на нижнем уровне иерархии автоматически отражаются и на более высоких уровнях. В результате администрирование можно выполнять только на каком-либо одном уровне, не дублируя его на десятке остальных, связанных с ним.

Безопасность объектов и Management Studio

Разрешениями уровня объектов, применяемыми к самим объектам, а также к пользователям и ролям, можно управлять из разных мест, в том числе и из Management Studio, и это вносит в управление системой безопасности определенную путаницу.

Управление из списка объектов

Для изменения разрешений объекта выполните следующие действия.

1. Щелкните правой кнопкой мыши на имени объекта в соответствующем узле (`tables`, `views` и т.п.) окна Object Browser и выберите в контекстном меню пункт Properties. Откроется диалоговое окно параметров этого типа объектов.
2. Щелкните на кнопке Permissions, чтобы открыть диалоговое окно разрешений объекта.

Как и в случае назначения разрешений с помощью инструкций, можно выбрать разрешения `grant`, `with grant` и `deny`. Список всех объектов базы данных находится в верхней части диалогового окна. Этот список можно использовать для быстрого переключения между объектами, не выходя из окна, и назначения им тех или иных разрешений.

Кнопка Columns в нижней части окна служит для открытия диалогового окна Column Permissions. Выберите пользователя, щелкните на этой кнопке и установите разрешения для этого пользователя на уровне столбцов. На уровне столбцов могут быть назначены только разрешения на выборку и обновления, так как операции вставки и удаления применяются на уровне строк.

Управление из списка пользователей

В списке пользователей базы данных утилиты Management Studio дважды щелкните на имени пользователя или щелкните на нем правой кнопкой и выберите в контекстном меню пункт Properties. Открывшееся диалоговое окно Database User Properties служит для назначения пользователей ролям.

Щелкните на кнопке Properties, чтобы открыть диалоговое окно параметров выбранной роли.

Если теперь щелкнуть на кнопке Permissions, откроется вкладка Permissions диалогового окна Database User Properties. Эта вкладка сходна с одноименной вкладкой диалогового окна Database Object Properties.

К сожалению, список объектов в этом окне не отсортирован, и отсортировать его невозможно с помощью щелчка на заголовке столбца. Это диалоговое окно остро нуждается в функции select all и прочих средствах, предлагаемых формами разрешений программы Access.

Управление из списка ролей

Разрешениями объектов можно также управлять и из списка ролей. Для открытия диалогового окна Database Role Properties дважды щелкните на роли или щелкните на ней правой кнопкой мыши и выберите в контекстном меню пункт Properties. Это диалоговое окно используется для назначения роли пользователей и других ролей, а также для удаления их из роли.

Щелкните на кнопке Permissions, чтобы открыть диалоговое окно разрешений роли. Работа в этой форме аналогична описанной в предыдущих разделах, за исключением того, что она организована с фокусом на роль.

Цепочки принадлежности

В базах данных SQL Server пользователи часто получают доступ к данным, проходя через несколько объектов. Цепочки принадлежности применимы к представлениям, хранимым процедурам и пользовательским функциям. Например:

- в Visual Basic программа может вызывать хранимую процедуру, которая извлекает данные из таблицы;
- отчет может отбирать данные из представления, которое, в свою очередь, основано на некоторой таблице;
- сложные хранимые процедуры могут в процессе работы вызывать несколько других процедур.

Во всех этих случаях пользователь должен иметь разрешения для вызова хранимой процедуры или отбора данных из представления. Должен ли пользователь при этом иметь права доступа к подлежащим таблицам этих процедур и представлений, зависит от цепочки принадлежности, протягивающейся от вызываемого пользователем объекта к таблице, в которой хранятся данные.

Если такая цепочка принадлежности не разорвана, то хранимая процедура может быть выполнена с помощью разрешения своего владельца. В этом случае пользователю достаточно прав на выполнение самой хранимой процедуры — разрешения к исходным таблицам ему не нужны.

Цепочки принадлежности являются прекрасным средством реализации структуры системы безопасности, в которой пользователи не имеют прямых полномочий на таблицы данных и выполняют только разрешенные им хранимые процедуры.

Если цепочка принадлежности разорвана (т.е. в ней появляется другой владелец объекта), SQL Server проверяет наличие разрешений к каждому из объектов, к которому осуществляется доступ.

Приведем примеры.

- Цепочка принадлежности от `dbo.A` к `dbo.B` и далее к `dbo.Person` не разорвана. В этом случае `dbo.A` может вызвать `dbo.B` и таким образом получить доступ к `dbo.Person` как `dbo`.
- Цепочка принадлежности от `dbo.A` к `Sue.C` и далее к `Joe.Purchase` разорвана, поскольку в ней присутствует несколько разных владельцев объектов. В этом случае `dbo.A` вызывает `Sue.C`, используя разрешения `Joe`, и `Sue.C` обращается к данным `Joe.Purchase`, также используя разрешения `Joe`.
- Цепочка принадлежности от `dbo.A` к `dbo.B` и далее к `Joe.Person` также разорвана. Таким образом, `dbo.A` вызывает `dbo.B` с использованием разрешений `dbo`, но `dbo.B` может получить доступ к `Joe.C` только с помощью разрешений `Joe`.

Пример простой модели защиты

За примерами организации разрешений обратимся к учебной базе данных OBXKites. В табл. 40.1 представлены разрешения для стандартных ролей базы данных, а в табл. 40.2 — список некоторых пользователей этих ролей.

Таблица 40.1. Роли базы данных OBXKites

Стандартная роль	Иерархическая структура ролей	Таблицы первичной файловой группы	Таблицы статичной файловой группы	Прочие разрешения
IT	Серверная роль sysadmin	—	—	—
Clerc	—	—	—	Разрешение выполнения нескольких хранимых процедур, которые считывают данные из оперативных таблиц и обновляют их
Admin	Фиксированная роль базы данных db_owner	—	—	—
Public	—	Разрешения отбора данных	—	—

Таблица 40.2. Пользователи базы данных OBXKites

Пользователь	Стандартная роль базы данных
Sammy	Admin
Joe	
LRN	IT
Группа Windows Clerc (пользователи Betty, Tom, Martha и Mary)	Clerc

В этой модели системы безопасности следующие пользователи могут выполнять задачи, описанные ниже.

- Betty как член группы Clerc может выполнять приложение VB, выполняющее хранимые процедуры, извлекающие и обновляющие данные. Поскольку Betty является членом роли Public, он может выполнять запросы извлечения данных.
- LRN как администратор и член группы IT может выполнять все задачи в базе данных, поскольку он принадлежит к фиксированной серверной роли sysadmin.
- Joe как член роли Public может выполнять запросы извлечения данных.
- Будучи членом роли Admin, Sammy может выполнять все хранимые процедуры. Он также может с помощью запросов вручную модифицировать любые таблицы. Как член группы Admin, включающей в себя роль db_owner, Sammy может выполнять любые задачи администрирования данных и изменять данные в любых таблицах.
- Sammy может выполнять резервные копирования, но только LRN имеет право восстанавливать базу данных из архивов.

Уровень C2 системы безопасности

Организации, которым требуется надежная защита базы данных, могут разработать и внедрить систему безопасности уровня C2.

Критерии оценки безопасности Министерства обороны США для компьютерных систем и баз данных варьируются от уровня A (используется очень редко), обозначающего проверенную конструкцию, до уровня D, обозначающего минимальную защиту. Уровень безопасности C2 обозначает *защиту управления доступом*. Она необходима для многих классифицированных данных, данных информационно-поисковых систем и большинства правительственных контрактов.

В сущности, уровень защиты C2 требует наличия нескольких условий.

- Уникальный идентификатор регистрации для каждого пользователя, защищенный от перехвата. Для предоставления доступа к базе данных пользователь должен зарегистрироваться.
- Метод аудита всех попыток каждого пользователя или процесса доступа к данным и их изменения.
- Отсутствие прав доступа по умолчанию ко всем объектам.
- Предоставление доступа к объектам исключительно их владельцу или под его ответственность.
- Ответственность всех пользователей за свой доступ к данным и их изменение.
- Защита данных в памяти от несанкционированного доступа.

Для сертификации SQL Server корпорация Science Applications International Corporation (Сан-Диего) выполнила тестирование для Агентства национальной безопасности США и Национального института стандартов и технологий, которые вовлечены в правительственную программу сертификации безопасности. Процесс тестирования продолжался 14 месяцев и финансировался компанией Microsoft.

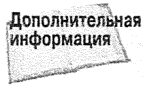
Публикацию *SQL 2005 C2 Admin and Users Guide* можно загрузить с сайта `http://msdn.microsoft.com/library/en-us/secauthz/security/c2_level_security.asp`.

Реализация системы безопасности уровня C2 в SQL Server требует следующего.

- СУБД SQL Server должна быть запущена в операционной системе Windows NT 4 Service Pack 6a.
- Не допускаются репликации слияния, репликации снимков базы данных. федеративные и распределенные базы данных.
- С помощью утилиты SQL Profiler должен осуществляться полный аудит.
- Параметр безопасности C2 должен быть включен — это приводит к останову SQL Server, если файл аудита не функционирует.
- Существуют другие ограничения на размещение и размеры файла аудита.

Представления и безопасность

Популярный, но очень спорный метод реализации системы безопасности основан на использовании представлений, которые проецируют только определенные столбцы или ограничивающие строки с помощью предложения `WHERE` и параметра `WITH CHECK OPTION`, и выдаче разрешений только на эти представления. Этот метод открывает пользователям только ограниченный доступ к данным. Некоторые компьютерные решения даже требуют организации всего доступа пользователей только с помощью представлений; использование этой же методики предполагают тесты сертификации специалистов Microsoft.



О том, как создавать представления и использовать параметр `WITH CHECK OPTION`, см. в главе 14.

Оппоненты использования представлений в качестве средства защиты используют следующие аргументы.

- Представления не откомпилированы и не оптимизированы.
- Защиту на уровне столбцов можно обеспечить и стандартными средствами защиты SQL Server.
- Использование представлений для обеспечения безопасности на уровне строк предполагает, что параметр `WITH CHECK OPTION` должен быть создан в каждом представлении вручную. По мере увеличения категорий уровня строк система потребует ручной настройки.

Криптография

Обычно обеспечения защиты доступа на уровне таблиц достаточно; если же нет — можно понизить уровень защиты до уровня столбцов. В то же время некоторая конфиденциальная информация, такая как номера кредитных карточек и документы особого отдела, требует дополнительной защиты информации, реализуемой с помощью шифрования данных в базе.



В то время как предыдущие версии SQL Server совершенно не решали задачу шифрования данных (возможно, за исключением процесса транспортировки данных клиенту), СУБД SQL Server 2005 лучше оснащена для корпоративной среды, так как способна шифровать данные с помощью ключей, паролей и сертификатов. Все редакции SQL Server поддерживают шифрование.

Введение в криптографию

Шифрование данных представляет собой их перемешивание с секретными ключами для получения копии, называемой *шифрованными данными*. Не имея ключа, эти данные практически невозможно расшифровать.

Симметричное шифрование использует для шифрования и расшифровки данных одни и те же ключи. Несмотря на то что этот метод проще в администрировании, как правило, он расценивается как рискованный, поскольку в копиях ключа нуждается множество людей. Это может и не быть проблемой, если шифрование и дешифровка данных выполняются в пределах одного сервера баз данных.

Асимметричное шифрование рассматривается как более строгое, поскольку один из ключей — *частный* — создается в паре со вторым ключом — *общим*. Если данные зашифрованы с помощью одного ключа из этой пары, их можно расшифровать с помощью другого ключа. Например, если я зашифровал данные с помощью своего частного ключа, вы можете их дешифровать с помощью парного ему общего ключа. Если возникает необходимость поделиться своим общим ключом с несколькими партнерами и требуется, чтобы только они могли дешифровать данные, необходимо двойное шифрование с помощью двух пар частного и общего ключей. Например, я шифрую с помощью своего частного ключа, а затем с помощью общего ключа партнера. Мой партнер выполняет все в обратном порядке: он дешифрует данные с помощью своего частного ключа, а затем с помощью моего общего.



Данные могут быть также зашифрованы и дешифрованы на среднем или клиентском уровне приложения .NET. Этот подход имеет то преимущество, что сервер баз данных никогда не видит незашифрованных данных. Мы с Дарреном Шейфером использовали этот метод при реализации проекта хранения данных кредитных карточек в SQL Server 2000. Даррен создал класс C#.NET, задейст-

вующий класс `System.Security.Cryptography`, для использования криптографического алгоритма Rinjdael. Этот класс работал отменно, и данные были зашифрованы с момента их изначального ввода до момента просмотра авторизированным пользователем отчета.

Шифрование с помощью симметричного ключа дает на выходе меньший массив данных, чем при использовании асимметричного ключа. Один из методов решения этой проблемы заключается в шифровании данных с помощью симметричного ключа, специально сгенерированного для данного набора данных, и последующем шифровании самого симметричного ключа с помощью асимметричного.

Сертификаты аналогичны ключам, но чаще всего выпускаются специальными центрами сертификации, такими как VeriSign, гарантирующими легитимность организации, связанной с сертификатом. В SQL Server можно сгенерировать и локальный сертификат.

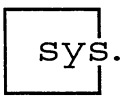
Криптографическая иерархия SQL Server

Система шифрования в SQL Server основана на иерархии ключей. На вершине этой иерархии находится уникальный *служебный мастер-ключ*, используемый в SQL Server для выполнения шифрования мастер-ключей базы данных при их создании.

На следующем уровне иерархии находится *мастер-ключ базы данных*, являющийся симметричным ключом, используемым в SQL Server для шифрования частных сертификатов и асимметричных ключей. Мастер-ключ базы данных создается с помощью инструкции DDL `CREATE MASTER KEY`. После этого созданный ключ шифруется с помощью служебного мастер-ключа и сохраняется как в пользовательской базе данных, так и в базе данных `master`.

```
CREATE MASTER KEY
  ENCRYPTION BY PASSWORD = 'P@$rw0rD';
```

Пароль ключа должен удовлетворять требованиям операционной системы Windows для строгого ключа.



Для просмотра информации о мастер-ключах используются представления каталога `syssymmetric_keys` и `sysdatabases.is_master_key_encrypted_by_server`.

В самой базе данных, ниже уровня ее мастер-ключа, находятся сертификаты и частные ключи.

Когда необходимо выполнить шифрование данных, SQL Server использует четыре метода:

- использование парафраз;
- использование симметричного ключа;
- использование асимметричного ключа;
- использование сертификата.

Шифрование парафразой

Первый метод шифрования данных предполагает использование парафразы. *Парафразы* аналогичны паролям, но на них не накладываются ограничения строгого пароля. Шифрованные данные имеют двоичный вид, поэтому в коде примера для столбца номера кредитной карточки `creditcardnumber` использован тип данных `varbinary`. Необходимая длина для зашифрованных данных определяется исходя из конкретной ситуации.

Реальное шифрование выполняется с помощью функции EncryptbyPassPhrase. Первым ее параметром является парафраза, за которой следуют шифруемые данные. В следующем примере продемонстрировано шифрование данных в инструкции INSERT:

```
CREATE TABLE CCard (
   CCardID INT IDENTITY PRIMARY KEY NOT NULL,
   CustomerID INT NOT NULL,
   CreditCardNumber VARBINARY(128),
   Expires CHAR(4)
   );
INSERT CCard(CustomerID, CreditCardNumber, Expires)
  VALUES(1,EncryptbyPassPhrase('Passphrase',
         '12345678901234567890'), '0808');
```

Зашифрованное значение, хранимое в базе данных, извлекается с помощью обычной инструкции SELECT:

```
SELECT *
  FROM CCard
  WHERE CustomerID = 1;
```

Результат в данном примере будет получен следующий (двоичное значение обрезано):

```
CCardID    CustomerID  CreditCardNumber      Expires
---------- ----------- --------------------- -------
1          1           0x01000000C8CF68C     0808
```

Для дешифровки данных кредитной карточки в обычный читаемый текст используется функция decryptbypassphrase в паре с функцией convert, преобразующей двоичный формат:

```
SELECT CCardID, CustomerID,
  CONVERT(VARCHAR(20), DecryptByPassPhrase('Passphrase',
          CreditCardNumber)),Expires
  FROM Ccard
  WHERE CustomerID = 1;
```

Будет получен следующий результат:

```
CCardID    CustomerID                      Expires
---------- ----------- --------------------- -------
1          1           12345678901234567890 0808
```

Совершенно очевидно, что данные дешифрируются таким же образом, которым ранее были зашифрованы. Если заданная для расшифровки парафраза некорректна, в результате будет получено пустое значение.

Метод использования парафразы имеет одно расширение. В процесс шифрования может быть вовлечен *аутентификатор*. В качестве аутентификатора обычно используют некоторое встроенное значение, неизвестное пользователю. Это в еще большей мере усложняет задачу дешифровки данных, если они были извлечены из базы.

В следующем примере мы добавим аутентификатор в шифрование с использованием парафразы. Код 1 включает аутентификатор, а последним аргументом является фраза аутентификатора:

```
INSERT CCard(CustomerID, CreditCardNumber, Expires)
   VALUES(3,EncryptbyPassPhrase('Passphrase',
         '12123434565678788989',1,
         'hardCoded Authenticator'), '0808');

SELECT CCardID, CustomerID,
  CONVERT(VARCHAR(20),DecryptByPassPhrase('Passphrase',
          CreditCardNumber,1,
          'hardCoded Authenticator')), Expires
  FROM Ccard
  WHERE CustomerID = 3;
```

Будет получен следующий результат:

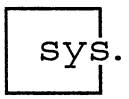
```
CCardID     CustomerID                       Expires
----------- ----------- -------------------- -------
2           3           1212343456567878989 0808
```

Шифрование с помощью симметричного ключа

Использование симметричного ключа предполагает наличие реального объекта, используемого при шифровании, а не пригодной для чтения парафразы. Симметричные ключи создаются в SQL Server с помощью следующей инструкции CREATE:

```
CREATE SYMMETRIC KEY CCardKey
WITH ALGORITHM = TRIPLE_DES
ENCRYPTION BY PASSWORD = 'P@s$wOrD';
```

После того как ключи будут созданы, они отобразятся в узле `Security⇨Symmetric keys` окна Object Explorer утилиты Management Studio,.



Для просмотра информации о симметричных ключах используется запрос T-SQL к представлению каталога `syssymmetric_keys`.

Ключи представляют собой объекты, так что их можно изменить и удалить так же, как и любые другие объекты SQL Server.

Криптографические алгоритмы

Криптографические алгоритмы определяют, как именно данные будут шифроваться с помощью ключа. Существует девять доступных для выбора алгоритмов: DES, TRIPLE_DES, RC2, RC4, RC4_128, DESX, AES_128, AES_192 и AES_256. Они отличаются скоростью и строгостью.

Алгоритм DES (Data Encryption Standard) был выбран в 1976 году официальным криптографическим алгоритмом для правительства США, однако редко использовался, так как на компьютерах того времени процесс шифрования мог занять сутки. Алгоритм TRIPLE_DES использует более длинный и, следовательно, более строгий ключ.

Алгоритм AES (Advanced Encryption Standard), также известный как Rijndael, введен Национальным институтом стандартов и технологий США в ноябре 2001 года. Числа 128, 192 и 256 в имени алгоритма отражают размер ключа в битах. Самым строгим алгоритмом этого семейства является AES_256.



SQL Server использует криптографические алгоритмы Windows. Так что если некоторый алгоритм не установлен в операционной системе, SQL Server не сможет его использовать. Алгоритмы AES не поддерживаются в Windows XP и Windows 2000.

Поскольку симметричные ключи должны передаваться клиенту в открытом виде, сам ключ также может быть зашифрован. SQL Server может шифровать ключи, используя один или множество паролей, другие ключи или сертификаты. Для распространения поколения ключей может быть задействована ключевая фраза.

Временный ключ действителен на протяжении текущей сессии, и, аналогично временным таблицам, его можно идентифицировать по знаку решетки (#). Временные ключи могут использовать идентификаторы GUID для идентификации зашифрованных данных, применяя параметр `indentity_value='passphrase'`.

Использование симметричных ключей

Для использования симметричного ключа его нужно вначале открыть с помощью команды OPEN, которая дешифрует ключ и делает его доступным для SQL Server:

```
OPEN SYMMETRIC KEY CCardKey
  DECRYPTION BY PASSWORD = 'P@s$wOrD';
```

Используя ранее созданную таблицу CCard, в следующем фрагменте кода данные шифруются с помощью ключа CCardKey. Функция encryptbykey принимает идентификатор GUID ключа, который находится с помощью функции key_guid(), а также данные, подлежащие шифрованию:

```
INSERT CCard(CustomerID, CreditCardNumber, Expires)
  VALUES(1,EncryptbyKey(Key_GUID('CCardKey'),
         '1111222233334445555'), '0808');
```

Для дешифровки данных ключ должен быть открыт. Функция decryptbykey идентифицирует корректный ключ из данных, а затем выполняет дешифровку:

```
SELECT CCardID, CustomerID,
  CONVERT(varchar(20), DecryptbyKey(CreditCardNumber))
         as  CreditCardNumber, Expires
  FROM Ccard
  WHERE CustomerID = 7;
```

Будет получен следующий результат:

```
CCardID     CustomerID  CreditCardNumber                 Expires
----------- ----------- -------------------------------- -------
1           7           1111222233334445555              0808
```

На практике рекомендуется закрывать открытый ключ сразу же после транзакции, в которой он использовался:

```
CLOSE SYMMETRIC KEY CCardKey
```

В большинстве приложений вам придется шифровать данные по мере их поступления в базу и дешифровать по мере их извлечения. Если вы хотите переместить данные на другой сервер и дешифровать их там, то оба сервера должны иметь идентичные ключи. Для генерации одного и того же ключа на двух серверах они должны создаваться с использованием одного алгоритма, значения идентичности и ключевой фразы.

Использование асимметричных ключей

Использование асимметричного ключа предполагает выполнение шифрования и дешифровки с помощью пары соответствующих друг другу частного и общего ключей. Генерация асимметричного ключа сходна с генерацией симметричного:

```
CREATE ASYMMETRIC KEY AsyKey
  WITH ALGORITHM = RSA_512
  ENCRYPTION BY PASSWORD = 'P@s$w0rD';
```

SQL Server поддерживает следующие асимметричные алгоритмы: RSA_512, RSA_1024 и RSA_2048. Они различаются длиной частного ключа.

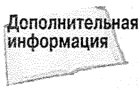
Асимметричные ключи также могут быть сгенерированы из существующих файлов ключей:

```
CREATE ASYMMETRIC KEY AsyKey
  FROM FILE = 'C:\SQLServerBIble\AsyKey.key'
  ENCRYPTION BY PASSWORD = 'P@s$w0rD';
```

Шифрование и дешифрация данных с помощью асимметричного ключа сходны с использованием симметричного ключа, за исключением того, что асимметричный ключ не нужно открывать.

Использование сертификатов

Сертификаты обычно используются для шифрования данных, передаваемых по Интернету между ключевыми точками HTTPS. SQL Server содержит сертификаты для своей "родной" службы HTTP Web Services. Сертификаты получают от специализированных центров сертификации.



Об использовании сертификатов с концевыми точками HTTP см. в главе 32.

Как избежать "инъекций" SQL

Инъекцией SQL называют технологию взлома, которая добавляет в параметр код SQL, который впоследствии выполняется как динамический SQL. Эта технология особо опасна тем, что любой, кто имеет доступ к Web-сайту организации и способен вводить данные в текстовые поля, потенциально может стать источником атак с помощью "инъекций" SQL. Существует множество вредоносных приемов, использующих прикрепляемый код и измененные предложения `WHERE`. Перед тем как приступить к изучению методов защиты, очень важно понять сам принцип работы этой вредоносной технологии. Об этом мы и поговорим в следующих разделах.

Прикрепление вредоносного кода

Добавление терминатора инструкции, другой инструкции SQL и содержимого позволяет взломщику передать программный код в строку выполнения. Например, если передаваемым параметром является

```
123'; Delete OrderDetail --
```

то в динамическую строку SQL будет добавлена инструкция DDL `DELETE`, которая и будет выполнена в пакете с основной:

```
SELECT *
  FROM Customers
  WHERE CustomerID = '123'; Delete OrderDetail --'
```

Терминатор инструкции завершает заложенную программистом операцию, после чего SQL Server рассматривает продолжающийся текст как следующую инструкцию в пакете. Завершающая кавычка могла бы привести к ошибке выполнения, однако эта проблема просто решается путем добавления маркера комментария. Каков результат? Пустая таблица строк заказов (`OrderDetails`).

Среди других популярных прикрепляемых кодов можно упомянуть запуск команды `xp_commandshell` и установку пароля для пользователя `sa`.

Прикрепление `OR 1=1`

Еще одним методом "инъекций" SQL является модификация предложения `WHERE` для выборки большего количества строк, чем было изначально предусмотрено.

Если пользователь вводит в текстовое поле строку

```
123' or 1=1 --
```

то условие `1=1` (которое всегда истинно) внедряется в предложение `WHERE`:

```
SELECT *
  FROM Customers
  WHERE CustomerID = '123' or 1=1 -'
```

Так как в инструкцию отбираются все строки, далее все зависит только от того, как система обрабатывает множества строк. Однако независимо от этого, программа сделает совсем не то, что должна была сделать.

Пароль? Какой пароль?

Еще одной распространенной "инъекцией" кода SQL является комментирование остальной части кода, подлежащего выполнению. Если пользователь вводит в Web-форму

```
UserName: Joe' --
Password : who cares
```

то результирующую инструкцию SQL можно будет прочитать следующим образом:

```
SELECT USerID
  FROM Users
  WHERE UserName = 'Joe'--' AND Password = 'who cares'
```

Включение маркера комментария в поле имени пользователя приводит к тому, что дальнейшая часть предложения WHERE, включая условие для пароля, игнорируется.

Защита от "инъекций" кода SQL

Предотвратить проникновение "инъекций" кода можно несколькими путями.

- Используйте команды EXECUTE AS и тщательно определяйте роли, чтобы инструкции не имели разрешений на удаление таблиц.
- Используйте ссылочную целостность DRI во избежание удаления строк первичной таблицы вместе с удалением строк вторичной таблицы.
- Никогда не смешивайте динамический SQL с данными, вводимыми пользователем в Web-форме. Все параметры обязательно передавайте с помощью хранимых процедур.
- Проверяйте и удаляйте параметры, содержащие терминаторы инструкций, комментарии и символы xp_.
- Остерегайтесь динамического SQL.
- Тестируйте свою базу данных с использованием методик "инъекций", описанных выше.

Проблеме инъекций кода SQL следует уделять повышенное внимание. Если ваше приложение предполагает ввод данных из Интернета и вы не предусмотрите действия, направленные против потенциальных инъекций, разрушение базы данных станет только вопросом времени.

Резюме

В нашу эпоху киберпреступлений как никогда ранее важна безопасность данных. Требование безопасности данных является одним из принципов информационной архитектуры. Несмотря на то что вполне возможно назначить всем пользователям роли системного администратора и игнорировать вопросы безопасности, стоит приложить совсем немного усилий, и функциональность гибкой и мощной системы обеспечения безопасности SQL Server защитит ваши данные во всевозможных ситуациях.

Следующая глава завершает очередную часть книги, посвященную вопросам администрирования SQL Server. В ней мы поговорим об особенностях развертывания и администрирования версии SQL Server Express.

ГЛАВА

41

Администрирование SQL Server Express

В этой главе...

Бесплатное ядро базы данных

Установка в фоновом режиме

Утилита Management Studio Express

Редакция SQL Server 2005 Express Edition, также называемая SQL Server Express, или SSE, является упрощенной версией популярной системы управления базами данных SQL Server компании Microsoft. Любая база данных, созданная в этой версии, может непосредственно использоваться любой другой версией SQL Server. Компания Microsoft сделала все для того, чтобы сделать SSE доступной и полезной версией для всех типов разработчиков, от начинающих до опытных. Функциональность и простота использования, заложенные в SSE, сделала эту СУБД довольно привлекательным выбором для программных приложений.

SSE отлично подходит для ограниченных реализаций, таких как организация рабочего места разработчика баз данных. Например, в состав основного пакета установки не входят утилиты управления; они сформированы в отдельный пакет загрузки. К тому же SQL Server Express использует модифицированную версию утилиты SQL Server Management Studio, которую мы также рассмотрим в настоящей главе.



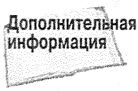
Если вы хотите узнать, над чем компания Microsoft работает в настоящее время, воспользуйтесь блогами на сайте Microsoft Developer Network (MSDN).

Установка SQL Server Express

Для установки SSE в графическом интерфейсе вы можете использовать процедуру, описанную в главе 4. Однако вполне вероятно, что вы вообще не хотите, чтобы пользователи занимались установкой чего-либо, кроме вашего приложения. В этом случае лучше прибегнуть к установке, выполняемой в фоновом режиме. В этом случае пользователю вообще не придется участвовать в процессе, что, в принципе, могло бы вылиться в некорректную установку всего приложения.



Максимальное снижение сложности установки, где это возможно, — вот истинный ключ к сердцу пользователя. Этим вы осчастливите не только пользователя, но и себя, поскольку на порядок повысите шансы установки работоспособной конфигурации приложения.



Функции установки SQL Server Express практически такие же, как и при установке полной версии SQL Server 2005, — отличие состоит только в составе устанавливаемых средств (в частности, в SSE отсутствует служба отчетности). Подробное описание процесса установки SQL Server см. в главе 4.

Копию версии SQL Server Express можно без труда загрузить с Web-сайта компании Microsoft по адресу

```
http://msdn.microsoft.com/vstudio/express/sql
```

Как и в случае полной версии SQL Server, дополнительно можно загрузить учебные базы данных, находящиеся на сайте

```
www.microsoft.com/downloads/details.aspx?familyid=
06616212-0356-46a0-8da2-eebc53a68034&displaylang=en
```

Базы данных устанавливаются с помощью той же методики, что и сам сервер баз данных.



Подробно об использовании учебных баз данных `Pubs` и `Northwind` см. в главе 3. Следует отметить, что с выходом пакета обновлений SP1 база данных AdwentureWorks была несколько переработана и дополнена.

Файл, который вы получите от компании Microsoft (`SQLEXPR.EXE`), будет в сжатом формате. Для того чтобы разархивировать файл, используйте в командной строке переключатель `/X`. В открывшемся диалоговом окне Choose Directory for Extracted Files будет по умолчанию указана текущая папка. Как правило, для разархивированных файлов создают отдельный каталог, чтобы не смешивать их с другими файлами, хранящимися на жестком диске.

В следующих разделах мы обсудим два метода, понижающих сложность фоновой установки сервера. Первый метод основан на использовании параметров командной строки и отлично подходит для небольших приложений и пакетных файлов, так как использование командной строки обычно снижает гибкость. Пользователь не сможет итеративно изменять параметры и тестировать их. Если вам нужно предоставить пользователю большую гибкость, чем позволяет командная строка, используйте второй метод. Для конфигурирования SSE он полагается на файлы INI. Несмотря на то что этот метод требует большей работы с вашей стороны, он обеспечивает установку, максимально подходящую к вашим требованиям. Больше каких-либо существенных различий между упомянутыми типами установки с функциональной точки зрения не существует. Любой из этих методов приведет к полной или выборочной установке SSE.



Установка SSE предполагает наличие среды Microsoft .NET Framework 2.0. Несмотря на то что на вашем компьютере этот продукт, вероятнее всего, установлен, того же нельзя гарантировать для компьютеров пользователей. Вы можете проверить наличие этой среды в их системе путем определения наличия папки `\WINDOWS\Microsoft.NET\Framework\v2.0.50727`. Если этой папки не существует, то на компьютере пользователя следует установить .NET Framework 2.0. Файлы установки предоставлены на сайте

```
www.microsoft.com/downloads/details.aspx?familyid=
0856eacb-4362-4b0d-8eddaab15c5e04f5&displaylang=en
```

Использование параметров командной строки для фоновой установки SSE

Установка с помощью командной строки позволяет использовать для инсталляции пакетные файлы. Вы также можете открыть окно командной строки операционной системы Windows (Пуск⇨Все програмы⇨Стандартные⇨Командная строка), перейти в каталог, в котором разархивировали файлы установки, ввести **SETUP.EXE** и нажать клавишу <Enter>. Также вы можете дважды щелкнуть на ярлыке этого файла в окне Проводника Windows.

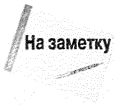
В большинстве рекомендаций предлагается использовать для установки SSE команду START. При этом операционная система открывает окно командной строки для выполнения программы установки. Если вам по определенным причинам не хочется открывать это окно, вы можете этого и не делать. В то же время с целью ознакомления введите в окне командной строки **start /?** и нажмите клавишу <Enter>. Вы увидите множество переключателей, которые можно использовать для настройки режима работы программы setup.exe. Например, вы можете настроить открытие окна установки в свернутом режиме, а также назначать программе более высокий приоритет, чтобы она выполнялась быстрее. В данном случае чаще всего используют переключатель /WAIT, который при использовании в составе команды START указывает операционной системе не закрывать окно командной строки, пока программа установки SSE не завершит свою работу. В общем случае командная строка, использующая START, выглядит следующим образом (все переключатели набираются в одной строке, несмотря на то, что в книге они приводятся в нескольких строках):

```
Start /WAIT ["заголовок"] [/Dpath] [/I] [/MIN] [/MAX] [/SEPARATE |
/SHARED] [/LOW | /NORMAL | /HIGH | /REALTIME | /ABOVENORMAL |
/BELOWNORMAL] [/B] SETUP [параметры_SSE]
```

В отличие от многих приложений командной строки Windows, после ввода команды **SETUP /?** на экран не выводится список доступных переключателей, поэтому ниже мы приводим их описание.

- /qb. Установка в фоновом режиме, при которой пользователь видит на экране основные элементы интерфейса, такие как индикатор процесса.
- /qn. Установка в фоновом режиме, при которой пользователь не видит на экране никаких элементов интерфейса.
- ADDLOCAL=[ALL|*список_функций*]. В этом элементе можно определить список функций SSE, которые следует устанавливать. Если ввести ключевое слово ALL, то будут установлены все функции. Этот параметр вы можете также использовать для дополнения существующей установки новыми функциями. Данный список следует разделять запятыми, не допуская пробелов. Например, параметр ADDLOCAL=SQL_Engine,SQL_Data_Files позволяет установить ядро SQL и отвести место для хранения файлов данных. Этот параметр следует использовать в таком виде, в котором он показан на рис. 41.1. Таким образом, если вы хотите использовать параметр SQL_Data_Files, то должны также использовать и параметр SQL_Engine, устанавливающий службы SSE. Среди доступных в данной версии функций следующие: SQL_Engine (ядро базы данных), SQL_Data_Files (файлы данных), SQL_Replication (репликация), SQL_FullText (полнотекстовый поиск), Client_Components (компоненты клиентской стороны), Connectivity (подключения) и SDK (инструментарий разработчика).
- AGTACCOUNT=*учетная_запись* и AGTPASSWORD=*пароль* указывают учетную запись и пароль, с которыми будет запускаться служба SQL Server Agent (вернее, ее экземпляр).

Доступ к этой службе осуществляется с помощью консоли служб операционной системы, запустить которую можно из папки Администрирование панели управления. В диалоговом окне Properties имя пользователя и пароль вводятся во вкладке Log On.



В будущих версиях SSE, скорее всего, появятся и другие функции, такие как служба отчетности (RS_Server).

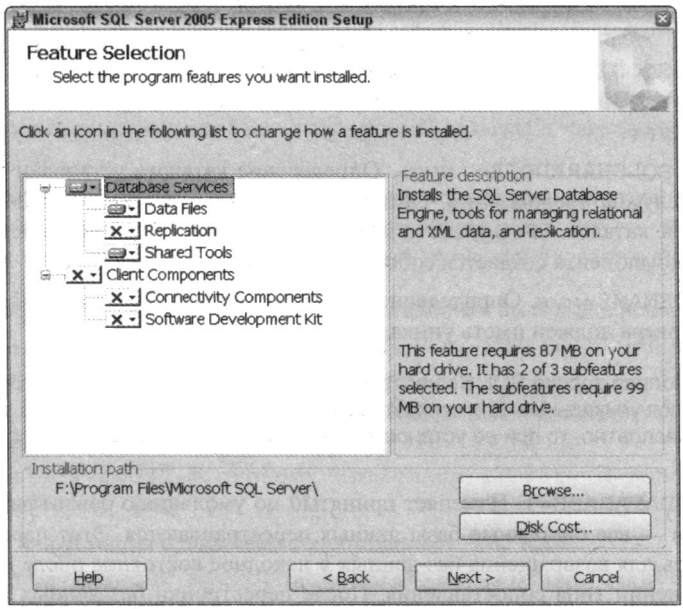


Рис. 41.1. Используйте параметр ADLOCAL для определения состава функций, доступных для установки

- AGTAUTOSTART=[0|1]. Настройка автоматического запуска службы SQL Server Agent. Если для этого параметра установлено значение 1, то служба запускается автоматически, если 0 — нет. Когда запускается служба SQL Server Agent, автоматически запускается и сама СУБД SQL Server, независимо от настроек автозапуска.



Полная версия SQL Server 2005 поддерживает параметры ASACCOUNT и ASPASSWORD для параметров безопасности службы анализа. Как правило, при этом требуется также указать в параметре INSTALLASDATADIR каталог установки. Служба анализа использует сопоставление, установленное по умолчанию, которое можно модифицировать с помощью параметра ASCOLLATION. Поскольку SSE не имеет в своей комплектации службу анализа, то упомянутые параметры в командной строке вы использовать не можете.

- COMPANYNAME=*имя*. Определение названия компании, на которую зарегистрирован сервер баз данных. Этот параметр в основном предназначен для использования при работе с файлом INI; в командной строке он является необязательным.
- DISABLENETWORKPROTOCOLS=[0|1|2]. Определение, какой сетевой протокол активизирует программа установки. Использование значения 0 позволяет включить все три сетевых протокола: общую память, именованные каналы и TCP/IP. Значение 1 яв-

ляется самым безопасным, так как включает только общую память. Значение 2 является стандартом для тех установок, где к серверу будет осуществляться доступ с удаленных компьютеров. Это значение активизирует общую память и протокол TCP/IP.

- ERRORREPORTING=[0|1]. Включает и отключает функцию, которая отправляет отчеты в компанию Microsoft при возникновении фатальных ошибок в процессе инсталляции. По умолчанию в программе установки параметр отправки сообщений отключен; если вы хотите отправлять сообщения об ошибках, включите этот параметр в явном виде.
- INSTALLDATADIR= *путь*. Определение каталога, в котором будут храниться файлы данных.
- INSTALLSQLDIR= *путь*. Определение каталога, в который будет устанавливаться SQL Server. По умолчанию используется каталог \Program Files\Microsoft SQL Server.
- INSTALLSQLSHAREDDIR= *путь*. Определение каталога установки таких компонентов, как служба уведомлений, служба интеграции и Workstation. По умолчанию используется каталог \Program Files\Microsoft SQL Server, в котором для каждого компонента создается собственная папка.
- INSTANCENAME=*имя*. Определение имени экземпляра SSE. Каждый экземпляр (установка) сервера должен иметь уникальное имя.



Полная версия SQL Server требует указания идентификатора продукта, являющегося уникальным для конкретной установки. Так как версия SSE распространяется бесплатно, то при ее установке значение идентификатора не используется.

- REBUILDDATABASE=1. Изменяет принятый по умолчанию режим работы программы установки — все системные базы данных перестраиваются. Этот параметр можно использовать для возвращения баз данных в исходное состояние после повреждения или при изменении типа сопоставления. После перестройки баз данных нужно восстановить базы master, model и msdb. Только после этого можно начать восстанавливать все остальные базы данных.
- REINSTALLMODE=[O|M|U|S]. Восстановление всех существующих инсталляций для добавления отсутствующих файлов и замены поврежденных. Использовать значение O рекомендуется при установке новой версии системы; M — для замены всех специфичных для компьютера ключей реестра; U — для замены всех остальных ключей реестра, связанных с пользователем; S — для переустановки всех "горячих клавиш".
- REMOVE=[ALL|*функция*]. Удаление SQL Server или отдельных компонентов сервера. Для просмотра установленных функций вы можете воспользоваться параметром ADDLOCAL. Этот параметр можно использовать и для установки всех функций, за исключением одной-двух.



Полная версия SQL Server 2005 использует параметры RSACCOUNT и RSPASSWORD для настройки системы безопасности службы отчетности и параметр RSAUTOSTART для настройки автоматического запуска этой службы. Сервер отчетности также требует отдельной конфигурации, устанавливаемой с помощью параметра RSCONFIGURATION. Так, в состав SE не входит служба отчетности, эти параметры в командной строке программы установки не используются.

- SAPW=*пароль*. Определение пароля системного администратора (т.е. пользователя sa) при использования смешанного режима аутентификации.

- SAVESYSDB=[0|1]. Определение, следует ли сохранять системные базы данных во время деинсталляции. Обычно этот параметр используют при обновлении версии пакета SQL Server.
- SECURITYMODE=SQL. Установка смешанного режима аутентификации. Если этот параметр отсутствует, то SSE будет использовать режим аутентификации Windows.
- SQLACCOUNT=*учетная_запись* и SQLACCOUNT=*пароль*. Определение учетной записи и пароля для службы SQL Server. Доступ к этой службе осуществляется с помощью консоли служб операционной системы, запустить которую можно из папки Администрирование панели управления. В диалоговом окне Properties имя пользователя и пароль вводятся во вкладке Log On.
- SQLAUTOSTART=[0|1]. Настройка автоматического запуска службы SQL Server. Если для этого параметра установлено значение 1, то служба запускается автоматически, если 0 — нет.
- SQLALLOCATION=*путь*. Определение настроек сопоставления для уровней сервера, базы данных, столбца и выражения. В общем случае установки по умолчанию работают хорошо, если у вас нет особых требований, таких как многоязыковая поддержка.



Общие сведения о настройках сопоставления вы можете почерпнуть из статьи

```
http://msdn2.microsoft.com/en-us/library(d=robot)/
ms143508.aspx
```

Информация, необходимая для настройки сопоставления в Windows, приведена по адресу:

```
http://msdn2.microsoft.com/en-us/library(d=robot)/
ms143515.aspx
```

а информация о двоичном сопоставлении — по адресу:

```
http://msdn2.microsoft.com/en-us/library(d=robot)/
ms143350.aspx
```

- UPGRADE=SQL_Engine. Выполнение обновления SQL Server (вместо установки). Этот параметр требует дополнительного указания имени экземпляра в параметре INSTANCE.
- USERNAME=*имя*. Имя человека, на которого зарегистрирован продукт. Это значение обычно указывают при использовании файла INI; в командной строке этот параметр необязателен.
- USESYSDB=*путь*. Определение пути к системным базам данных при выполнении обновления. Никогда не включайте в этот путь каталог Data. Например, если данные фактически содержатся в папке \Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data, в параметре следует путь \Program Files\Microsoft SQL Server\MSSQL.1\MSSQL.

Использование файлов INI при установке SSE

При использовании в процессе установки файла инициализации (INI) применяются те же параметры, что и в командной строке. Таким образом, в этом методе применимы все функции, описанные в предыдущем разделе. В то же время в файлах INI используется совсем иная методика определения параметров установки. При этом командная строка должна иметь следующий вид:

```
SETUP.EXE /settings C:\SQLEXPR\TEMPLATE.INI /qn
```

В отличие от установки из командной строки переключатель SETTING указывает на имя и путь к файлу INI. Параметр /qn запрещает открытие окна интерфейса программы установки. Если же вы хотите отобразить такие базовые элементы интерфейса, как индикатор состояния, то можете использовать параметр /qb.

Если вам нужно быстро создать файл инициализации, воспользуйтесь файлом TEMPLATE.INI, входящим в состав файлов SSE. Вы найдете его в папке, в которую разархивировали пакет инсталляции. Этот файл содержит массу документации и все настройки, которые возможно установить в файле INI (рис. 41.2).

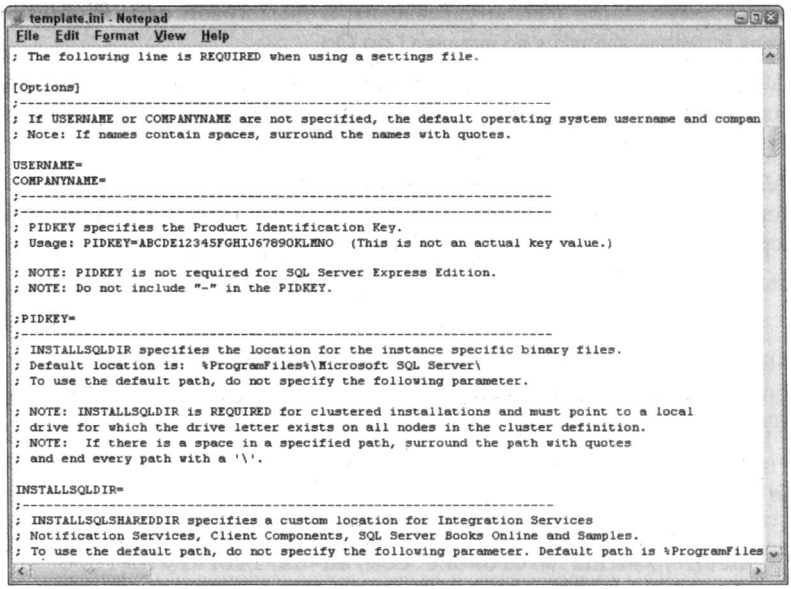


Рис. 41.2. Не стоит начинать файл инициализации с нуля — достаточно заполнить пробелы в файле TEMPLATE.INI

Всмотритесь в детали файла TEMPLATE.INI, используя код, показанный на рис. 41.2. Любой файл INI начинается с раздела [Options]. Установка версии SSE не требует никаких дополнительных разделов; кроме этого, если они присутствуют в файле INI, то будут просто проигнорированы (или вы можете получить сообщение об ошибке).

Файл TEMPLATE.INI содержит множество комментариев. Каждая строка комментария начинается с символа точки с запятой (;). При необходимости вы можете добавить и собственные комментарии для документирования особых требований компании. Также можно создать строки, которые будут использоваться достаточно редко, комментируя и раскомментируя их по мере необходимости. Обязательно документируйте эти элементы, чтобы не забыть, когда их стоит использовать, а когда нет.

Введите необходимые значения в файл TEMLATE.INI. В этот файл компания Microsoft включила все параметры, допустимые в программе SETUP.EXE. Обязательно просмотрите файл шаблона от начала до конца, чтобы случайно не пропустить какой-либо нужный параметр. Также обязательно закомментируйте все параметры, которые версия SSE не поддерживает (например, PIDKEY).

Версия Management Studio для SSE

Установка SSE не включает в себя утилиты администрирования — их нужно загрузить и установить отдельно. Все эти утилиты можно найти на сайте

```
www.microsoft.com/downloads/details.aspx?FamilyId=
82AFBD59-57A4-455E-A2D6-1D4C98D40F6E&displaylang=en
```



С выходом пакета обновлений SP1 появился новый доступный для загрузки пакет SQL Server 2005 Express Edition with Advanced Services, содержащий все, что существовало ранее в SQL Server 2005 Express Edition, вместе с новым инструментом управления SQL Server Management Studio Express. Кроме того, этот пакет содержит средства полнотекстового поиска и службу отчетности. Также вышел отдельный, доступный для загрузки пакет SQL Server 2005 Express Toolkit, включающий в себя утилиту Business Intelligence Developement Studio. Эта утилита позволяет создавать и редактировать серверные отчеты.

Специализированная версия Management Studio для SSE содержит следующие функции.

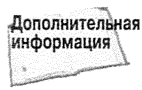
- Единый интегрированный графический интерфейс пользователя для выполнения простых операций с базой данных.
- Мастера для создания объектов (таких как базы данных и таблицы) и управления ими.
- Возможность просмотра объектов базы данных с помощью Object Explorer.
- Редактор запросов, поддерживающий множественные результирующие наборы данных.

После загрузки Management Studio этот продукт нужно установить, для чего следует дважды щелкнуть на файле с расширением `.msi`. В данном случае никаких дополнительных параметров или переключателей не используется.



Версия SSE поддерживает только пять пользователей. Если нужна поддержка большего числа пользователей, используйте одну из полных версий SQL Server.

Версия Management Studio для SSE работает точно так же, как для полной редакции SQL Server. Естественно, эта утилита имеет ряд ограничений, связанных с особенностями редакции SSE. Самым большим видимым отличительным признаком является отсутствие папок `Notification` и `SQL Server Agent`. Также в этой версии утилиты отсутствуют элементы, недоступные в SSE. Во всем остальном утилита Management Studio для редакции Express Edition похожа на своего "большого брата". Это значит, что данный продукт вполне можно использовать для учебных целей.



Более подробно о работе в Management Studio см. в главе 6.

Резюме

Для некоторых решений, использующих базы данных, версия SQL Server Express является отменным выбором. Ее ограничения никак не сказываются на общей производительности, и вы можете использовать эту редакцию как бесплатное приложение к своей программе. В этой главе было описано, как использовать усеченную утилиту Management Studio и выполнять установку СУБД в фоновом режиме.

Бизнес-логика

ЧАСТЬ



Если верить принципам информационной архитектуры, о которых мы говорили в главе 1, то информация важна не только в повседневной работе, но и для анализа накопленных данных. Для этого процесс извлечения, преобразования и загрузки данных (ETL) накапливает оперативную информацию и помещает ее в хранилище, используя шаблоны, специально созданные для анализа. Кубы, запросы MDX и служба отчетности извлекают данные из хранилищ и представляют их для анализа.

Весь процесс анализа исторических и текущих данных является активной частью информационных технологий, носящей собирательное название *бизнес-аналитика*.

В последних трех версиях SQL Server компания Microsoft постоянно наращивала службы бизнес-аналитики, в результате в версии SQL Server 2005 мы имеем возможность пожинать плоды этих усилий. Как служба интеграции, так и простота построения кубов данных и насыщенный интерфейс службы отчетности способны удовлетворить самые взыскательные требования к бизнес-аналитике.

В то же время бизнес-аналитика не может существовать в вакууме. Хранилище данных может зависеть от множества баз, а бизнес-анализ может использовать средства разных производителей. Несмотря на то что исследование сторонних утилит выходит за рамки книги, в этой части мы все-таки затронем самую популярную программу бизнес-анализа — Excel — и рассмотрим ее с точки зрения взаимодействия с SQL Server.

В этой части...

Глава 42
ETL в службе интеграции

Глава 43
Бизнес-логика в службе анализа

Глава 44
Раскрытие данных в службе анализа

Глава 45
Программирование запросов MDX

Глава 46
Создание отчетов в службе отчетности

Глава 47
Администрирование отчетов в службе отчетности

Глава 48
Анализ данных в Excel и Data Analyzer

ГЛАВА

42

ETL в службе интеграции

В этой главе...

Переменные и выражения службы интеграции

Создание потоков управления и данных

Упаковка обработки событий

Отладка

Полный список элементов пакета

Конфигурирование пакетов для различных сред

Развертывание пакета

Службу интеграции часто описывают как инструмент извлечения, преобразования и загрузки данных (ETL). Эти средства обычно ассоциируют с задачами подготовки данных для хранения, анализа и отчетности, однако служба интеграции вышла далеко за эти рамки. Она является надежной средой программирования задач, связанных с анализом и хранением данных.

Многие опытные пользователи уже сталкивались со службой преобразования данных DTS, поэтому предпочитают испытанные средства языка T-SQL изучению ETL, и язык T-SQL представляет собой реального соперника службы интеграции. В то же время те, кто все-таки отважился исследовать средства ETL, несомненно отметили для себя следующие преимущества.

- Простые и быстрые методы перемещения больших массивов данных минимизируют нагрузку на базы данных и упаковывают информацию в таблицы назначения, тем самым снижая объем блокировок и размеры журнала транзакций.
- Способность объединять в цепочку множество задач при осуществлении полного контроля над их порядком и обработкой ошибок. Многие задачи могут выполняться параллельно.
- Создание подключений для чтения и записи любых типов данных, не требующее специального программирования и вызова связанных серверов.
- Распространенные задачи управления данными реализованы без необходимости программирования. Среда создания сценариев .NET расширяема с помощью дополнительных надстроек.
- Результирующие пакеты настраиваются в зависимости от ситуации; они имеют ряд параметров развертывания, конфигурирования и протоколирования.

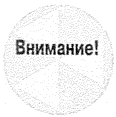
Несмотря на то что программирование на T-SQL и прочих языках может дать в результате ту же функциональность,

большинство проектов требуют значительных усилий и завершаются реализацией минимальной обработки ошибок (нечто вроде сообщения "Напишите мне, если обнаружите ошибку"). Один из моих коллег однажды при развертывании очередной системы всего за два дня получил от пользователей около восьми тысяч таких писем.

Служба интеграции позволяет избежать излишней детализации, создавая возможность заниматься реальным проектированием самодостаточных приложений. Эта служба также преуспела в идентификации проблем и выполнении операций восстановления в теле самого приложения. Многие функции службы интеграции, такие как перенаправление ошибок, сложные ограничения приоритетов, преобразование данных и нечеткий поиск, отлично подходят для "самолечения". Нет ничего лучше, чем создание приложения системной интеграции, за которым не нужно постоянно ухаживать, как за маленьким ребенком.



Служба интеграции (Integration Services) формально является наследником службы преобразования данных DTS, доступной в версии SQL Server 2000. Обе они используют одни и те же базовые концепции, однако служба Integration Services была полностью переписана, как говорится, с нуля. Служба интеграции доступна в редакциях Standard и Enterprise.



Выход пакетов обновлений SP1 и SP2 позволил расширить и улучшить функциональность службы интеграции. В основном в данной главе мы будем рассматривать работу службы интеграции в варианте "чистой" версии SQL Server 2005. Все изменения, связанные с пакетами обновлений, будут описаны в конце главы.

Среда проектирования

Лучше всего начать знакомство со службой интеграции с освоения ее среды проектирования. Запустим утилиту Business Intelligence Development Studio (далее BIDS) и создадим новый проект. Шаблон Integration Services находится в папке `Business Intelligence`. Окно программы показано на рис. 42.1.

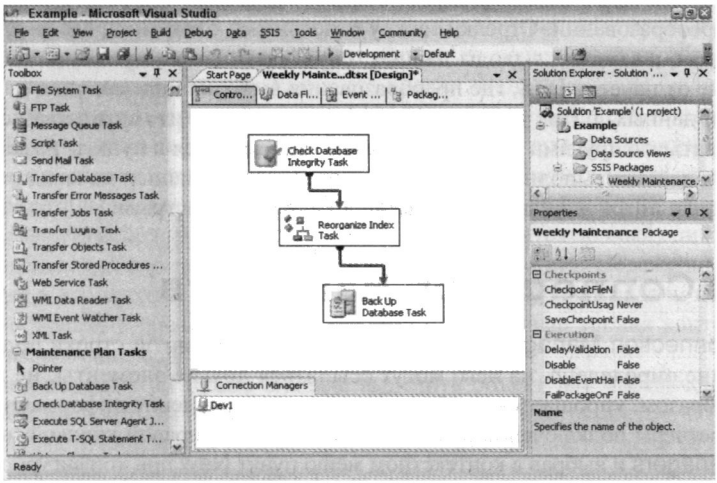
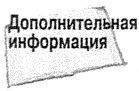


Рис. 42.1. Вкладка Control Flow среды проектирования службы интеграции



Более подробно о работе в программе BIDS см. в главе 6.

Кроме общеизвестных панелей Solution Explorer и Properties, при проектировании пакетов используются и другие (полный их список вы можете увидеть в меню View).

- Connection Managers (Диспетчеры подключений). На этой панели находятся указатели на файлы, базы данных и прочие объекты, существующие в контексте выполнения задач, помещенных в область проектирования. Например, задача Execute SQL Task требует наличия подключения к базе данных.
- Toolbox (Панель элементов). Эта панель содержит список задач, которые можно перетащить в область проектирования. Список доступных задач зависит от конкретной вкладки.
- Control Flow (Поток управления). Это основная область проектирования, на которую помещаются задачи и где выполняется их конфигурирование и упорядочение.
- Data Flow (Поток данных). Одной из задач, которые можно конфигурировать на панели Control Flow, является Data Flow. Она используется для перемещения и преобразования данных. Панель Data Flow используется для конфигурирования одноименного задания.
- Event Handlers (Обработчики событий). События происходят как с пакетом в целом, так и с отдельными его задачами. На эту панель помещают задачи, которые следует выполнить при возникновении определенных событий, таких как `OnError` и `OnPreEcecute`.
- Package Explorer (Обозреватель пакета). На этой панели перечислены все элементы пакета в древовидной структуре. На ней можно найти те сконфигурированные элементы, которые не очевидны в других представлениях.

Выполнение пакета происходит последовательно, при этом первой задачей считается та, которая не имеет предшественников. Далее выполнение продолжается с учетом установленных приоритетов задач. Тем, кто в предыдущих версиях SQL Server работал с пакетами службы DTS, такой порядок знаком. Так для чего же предназначена вкладка Data Flow?

Любому множеству данных во вкладке Control Flow соответствует одна задача. Для просмотра конфигурации задачи следует дважды щелкнуть на ее ярлыке (рис. 42.2). Обратите внимание на то, что здесь не содержится иных задач, кроме источников и приемников данных, а также преобразований. Стрелки между отдельными блоками представляют не порядок выполнения задач, а входы и выходы данных, определяющие их путь от источника к приемнику. В отличие от пакетов DTS, где преобразования выполнялись как один шаг при извлечении или записи данных, поток данных может формироваться из множества источников, разделяться и сливаться с другими потоками, а затем записываться в пункты назначения. Так как преобразования данных выполняются без операций чтения и записи на каждом шаге, хорошо спроектированный поток данных может быть на удивление быстрым.

Вкладка Connection Managers

Вкладка Connection Manager служит своеобразной оболочкой строки подключения. Когда подключение определено, на него могут ссылаться другие элементы, не дублируя друг друга. Таким образом, упрощается конфигурирование и управление информацией.

Новые диспетчеры подключений можно создать, щелкнув правой кнопкой мыши на панели Connection Managers и выбрав в контекстном меню пункт New при конфигурировании задачи, требующей подключения. В списке этого контекстного меню перечислены наиболее популярные типы подключений; все остальные доступны после щелчка на пункте New Connection.

Каждый тип подключения имеет собственные диалоговые окна редактора и параметров, состав которых зависит от конкретного типа. Каждый из этих двух списков может содержать элементы, недоступные в другом списке. Например, редактор OLE DB является единствен-

ным местом, где устанавливается время ожидания подключения, в то время как параметр проверки задержки устанавливается на панели Properties.

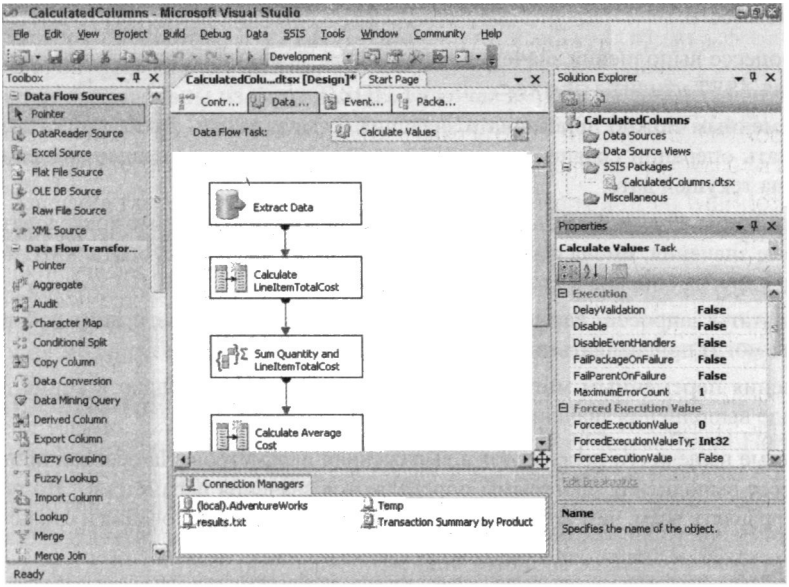


Рис. 42.2. Вкладка Data Flow среды проектирования службы интеграции

Переменные

Как и любая другая среда программирования, служба интеграции поддерживает переменные для управления выполнением процедур, операций с данными и прочих целей. Щелкните правой кнопкой мыши на рабочей области и выберите в контекстном меню пункт Variables. По умолчанию открывшаяся панель содержит только те переменные, в область видимости которых входит выделенный объект, а также его предки. Например, если щелкнуть на пустом месте рабочей области конструктора, объектом будет считаться весь проект и на панели Variables отобразятся только переменные уровня пакета, а при выделении объекта потока управления будут показаны переменные уровня выделенной задачи, а также всего пакета (который является родительским объектом по отношению к выделенному). Если же щелкнуть на кнопке Show User Variables панели инструментов, то отобразится полный список переменных.

Область определения переменной задает область ее видимости. Переменные уровня пакета видимы везде в его пределах, в то время как переменные уровня задачи или обработчика ошибок видимы только в пределах одного объекта. Переменные уровня контейнера видимы во всех объектах, которые содержатся в нем.

Для создания переменной нужно вначале выделить объект для обозначения области видимости, а затем щелкнуть на кнопке Add Variable панели инструментов. Следует обратить внимание, что область видимости переменной невозможно изменить после ее создания — для этого ее нужно удалить и создать заново.

В дополнение к области определения любая переменная имеет пространство имен — по умолчанию `User` или `System`. Пространство имен пользовательских переменных можно изменить, в то время как системные переменные только в редких случаях поддаются изменению. Пространство имен указывается для полной квалификации ссылок на переменную. Например, ссылка на пользовательскую переменную `MyVar` будет иметь следующий вид: `User::MyVar`.

Использование переменных

Значения переменным можно присваивать вручную на панели Variables, но существуют и другие возможности.

- В процессе выполнения значения переменным можно присвоить с помощью переключателя `/SET` утилиты `dtexec`.
- Переменным можно присваивать значения в выражениях. Такие выражения могут содержать операции сравнения, другие переменные и вычисленные значения, основанные на текущей дате.
- Контейнеры `For` и `Foreach` могут присваивать переменным последовательные числовые значения, имена файлов в папке, названия узлов документа XML, а также элементы других списков и коллекций.
- Результаты запросов, как в виде скалярных значений, так и в виде наборов данных, также могут присваиваться переменным.
- Значения переменным могут присваивать сценарии, но только в конце выполнения (в ходе выполнения значения фиксированы).

Переменные чаще всего используют в выражениях параметров. На основании этих выражений в процессе выполнения программы определяются значения параметров. Это позволяет переменным управлять всем, чем угодно, — от текста запросов, до включения и отключения задач.

Выражения

Выражения в службе интеграции используются для вычисления значений в циклах, разбиения потоков данных, присвоения значений переменным и назначения параметров заданий. Язык, используемый для определения выражений, имеет полностью новый синтаксис — нечто вроде комбинации T-SQL и C#. К счастью, вам на помощь придет инструмент Expression Builder, который можно запустить во множестве мест. Следует отметить ряд ключевых моментов.

- Переменные начинаются с символа коммерческого "эт" (@) и могут содержать пространство имен. Таким образом, ссылка `@[User::foo]` указывает на пользовательскую переменную `foo`. Ссылки на столбцы идентифицируются источником данных. Например, ссылка `[RowFileSource].[CustomerName]` указывает на столбец `CustomerName`, который можно прочитать в файле `RowFileSource`. Квадратные скобки являются необязательными для имен, не содержащих пробелы и другие специальные символы.
- Операторы сходны с языком C, например, символы `==` указывают на операцию сравнения, символы `&&` заменяют логическое AND, а восклицательный знак заменяет операцию на противоположную (например, `!<` или `!=`). Символ вопросительного знака заменяет функцию `iif`. Например, результатом выражения `@[User::foo]==17&&CustonerID<100` будет значение `true`, если переменная `foo` имеет значение `17`, а столбец `CustomerID` — значение меньше `100`.
- Строковые константы заключаются в двойные кавычки, а специальные символы предваряет обратная косая черта (например, `\n` или `\t`).
- Оператор `cast` представляет собой заключенную в скобки целевую переменную, после чего следует преобразуемое значение. Например, выражение `(DT_I4)"193"` преобразует строку в целое 4-байтовое число, а выражение `(DT_STR, 10, 152)@[User::foo]` преобразует значение переменной `foo` в 10-символьную строку, использующую кодовую страницу 1252. Для кодовой страницы нет значений по умолчанию, так что вам придется самому заучить номер кодовой страницы своего языка.



Не стоит путать кодовую страницу с региональными настройками. Кодовая страница отображает символы в соответствующие им коды. Если вам интересно ознакомиться с кодовыми страницами, воспользуйтесь следующими ресурсами:

`www.i18nguy.com/unicode/codepages.html`
`www.micrrosoft.com/typography/unicode/cscp/htm`

- Функции наследованы из мира T-SQL, включая хорошо знакомые функции работы с датами (`GETDATE`, `DATEADD`, `YEAR`), строками (`SUBSTRING`, `REPLACE`, `LEN`), а также математические функции (`CEILING`, `SIGN`).

Элементы конфигурирования

В службе интеграции совместно работает множество элементов, используя задания потока выполнения и компоненты потока данных. В этом разделе будут описаны общие концепции и настройки, а также уникальные свойства, характерные для конкретных элементов.

Поток управления

Конфигурирование потока управления во вкладках Control Flow и Event Handler осуществляется путем перетаскивания элементов (задач и контейнеров) в рабочую область конструктора, конфигурирования каждого из элементов и последующего связывания их в цепочку выполнения. Отдельные элементы конфигурируются с помощью частично пересекающихся окон Properties и Editor. Щелкните правой кнопкой мыши на элементе и выберите в контекстном меню пункт Edit. Откроется диалоговое окно редактора, состоящее из нескольких страниц (наполнение этих страниц зависит от типа выделенной задачи).

Все редакторы содержат страницу Expressions, на которой можно конфигурировать свойства с помощью выражений, а не статичных значений. Существующие выражения можно просмотреть и модифицировать непосредственно на странице; также можно щелкнуть на кнопке с эллипсом и тем самым активизировать инструмент Expression Builder. Дополнительные выражения можно редактировать непосредственно на странице или открыть щелчком на эллипсе диалоговое окно Expression Editor, показанное на рис. 42.3. Выберите устанавливаемый параметр в левом столбце, а в правом введите выражение. Если есть необходимость, воспользуйтесь редактором выражений, который открывается щелчком на эллипсе.

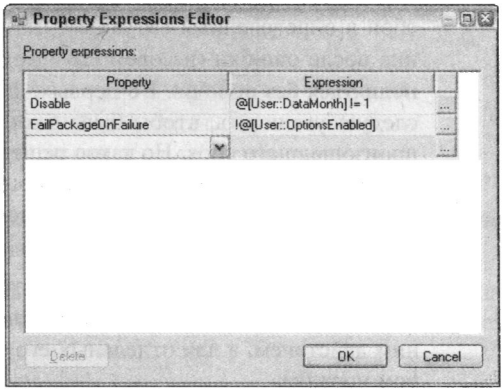


Рис. 42.3. Окно редактора выражений свойств объектов

Несмотря на множество индивидуальных свойств объектов, большая их часть повторяется в разных элементах, в том числе в пакетах, контейнерах и отдельных задачах. Среди общих свойств можно упомянуть следующие.

- **Delay Validation**. Обычно все задачи в пакете проверяются перед выполнением, чтобы избежать неполной работы. И правда, зачем тратить полчаса на поиск случайной опечатки в названии переменной? В то же время эту операцию можно отложить до момента выполнения, установив для этого параметра значение `true`. Этот параметр важен в задачах, которые ссылаются на объекты, которые при запуске задачи еще не существуют и создаются в процессе выполнения.

- **Disable**. Когда для этого параметра установлено значение `true`, задание не выполняется. Этот переключатель также доступен в контекстном меню объекта задания. Отключенные задачи отображаются более темным цветом.
- **DisableEventHandler**. Установка этого параметра отключает обработку ошибок. Обычно его устанавливают, когда обработка ошибок, выполняемая родительским контейнером, не подходит для конкретного объекта.
- Свойства обработки ошибок лучше рассматривать в группе.
  - **FailPackageOnFailure**. Если для этого параметра установлено значение `true` и возникает ошибка в одном из заданий пакета, выполнение всего пакета прерывается.
  - **FailParentOnFailure**. Если для этого параметра установлено значение `true`, выполнение родительского контейнера прерывается, если возникает ошибка в одной из его задач. Если задача не включена в контейнер в явном виде (т.е. с помощью выражений `For Loop`, `Foreach Loop` или `Sequence`), то она неявно обволакивается невидимым контейнером `TaskHost`, выступающим в роли родителя. По умолчанию этот параметр имеет значение `false`.
  - **MaximumErrorCount**. Максимальное количество ошибок задач или контейнеров, по достижении которого выполнение пакета прекращается. По умолчанию для этого параметра установлено значение 1.

  Принимая во внимание значения параметров по умолчанию уровня пакета, контейнера или задания, любая задача, завершающаяся ошибкой, приводит к прекращению выполнения пакета. Причиной тому является параметр `MaximumErrorCount`. Это справедливо независимо от неверных ветвей выполнения, определенных порядком выполнения. Увеличение значения этого параметра для задачи позволит включить ветвление.

  Учитывая такой режим работы, спрашивается, где более уместны параметры `FailOn`? В качестве примера рассмотрим контейнер с двумя заданиями, одно из которых дает сбой в определенных ситуациях (назовем его `Try`), а второе выполняет восстановление после ошибки (назовем его `Recover`). В то же время сам контейнер должен выполняться без ошибок. Во-первых, значение параметра `MaximumErrorCount` пакета следует увеличить, чтобы была возможность запустить задание восстановления после произошедшего сбоя. Но такое решение приведет к игнорированию ошибок и второго задания. Чтобы исправить ситуацию, установите параметр `FailOnFailure` в задании `Recover`. Таким образом, выполнение пакета будет прервано при возникновении ошибки в задании `Recover`.
- **LoggingMode**. По умолчанию для этого параметра установлено значение `UseParent Setting`. Таким образом, протоколирование для всего пакета можно определить одним действием, а для отдельных его элементов включать и отключать в индивидуальном порядке.
- Для определения последовательности операций, таких как внесение изменений во множество таблиц, можно использовать транзакции. При этом все задачи либо будут, либо не будут выполнены вместе. Следующие параметры регулируют транзакции в пакете.
  - **IsolationLevel**. Установка уровня изоляции транзакций в одно из следующих значений: `Unspecified`, `Chaos`, `ReadUncomited`, `ReadCommited`, `RepeatableRead`, `Serializable` или `Snapshot`. По умолчанию принято значение `Serializable`.
  - **TransactionOption**. Этот параметр может принимать одно из следующих значений: `NotSupported` (если элемент не участвует в транзакции), `Supported` (если

родительский контейнер требует транзакцию, в которой данный элемент участвует) или `Required` (если родительский контейнер не запустил транзакцию, данный контейнер запустит ее).

Если транзакция начата родительским контейнером, то в ней могут участвовать все дочерние элементы, в которых параметр `TransactionOption` имеет значение `Suported` или `Required`.

Организация потока управления

Как уже говорилось ранее, порядок выполнения задач определяется ограничениями первенства. Выберите любое задание или контейнер и перетащите значок стрелки, прикрепив его к элементу тупым концом. Выполняйте это действие до тех пор, пока все элементы не будут связаны, как нужно. Все несвязанные задачи будут запускаться в непредсказуемом порядке, часто даже параллельно друг другу. Для каждого элемента по умолчанию определен параметр `On Success`. Если дважды щелкнуть на нем, откроется диалоговое окно редактора Precedence Constraint Editor (рис. 42.4).

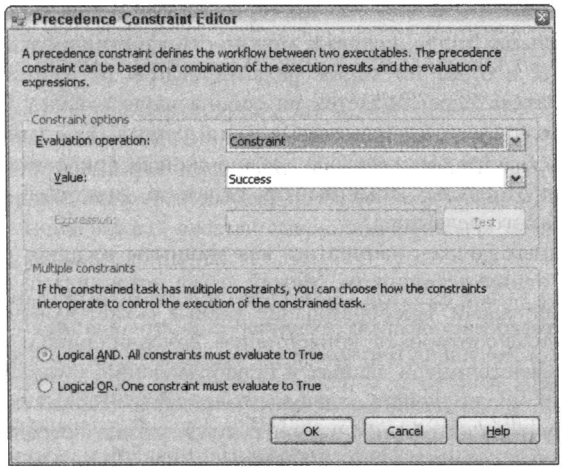


Рис. 42.4. Редактор Precedence Constraint Editor

В верхней части окна в разделе Constraint options определяется, когда ограничение должно активизироваться (все зависит от концепций операции).

- **Ограничение**. На основании результата выполнения предыдущей операции — успешном, ошибочном или законченном (выполнение является завершенным, независимо от того, произошла ли ошибка).
- **Выражение**. На основании вычисления введенного выражения, принимающего значение `true` или `false`.

Эти концепции комбинируются четырьмя способами: одно ограничение, одно выражение, выражение с ограничением, а также выражение или ограничение. Все это позволяет создавать довольно гибкие конструкции. В качестве примера рассмотрим задание, подсчитывающее в предварительно загруженной таблице количество обработанных строк. Выполнение задачи может пойти двумя путями: успешным, когда число обработанных строк соответствует количеству загруженных, и ошибочным, когда либо задание завершилось ошибкой, либо количество обработанных строк меньше необходимого.

В нижней части редактора в области Multiple constraints определяется, как следует поступать со множеством входящих стрелок. Если выбирается значение AND (оно принято по умолчанию), то перед выполнением задания должны выполниться все ограничения. Если выбирается OR, то удовлетворение любого входящего ограничения влечет за собой выполнение задания. Чаще всего используется логическое "И" (т.е. AND). Логическое "ИЛИ" (т.е. OR) обычно используется тогда, когда рабочие потоки разбиваются и сливаются. Например, поток управления может разделяться в зависимости от успеха выполнения некоторого задания, а затем после обработки ошибки ход выполнения входит в обычное русло. Если бы использовалось логическое "И", то в точке слияния ветвей требовалось бы одновременно и успешное, и ошибочное выполнение, что невозможно по определению.

Стрелки, представляющие порядок выполнения заданий, визуально отражают тип ограничения. Зеленая стрелка указывает на успешное выполнение, красная — на ошибочное, синяя — на завершение выполнения. Ограничения, использующие выражения, обозначаются значком `f(x)`. Ограничения логического "И" представляют собой сплошную линию, а логического "ИЛИ" — пунктирную.

Поток данных

В отличие от остальных задач, которые можно сконфигурировать в потоке управления, задание потока данных в ответ на запрос редактирования не открывает диалоговое окно. Вместо этого пользователь переключается на специальную вкладку Data Flow, где может просмотреть и сконфигурировать параметры потока данных. Каждый из компонентов, отображаемых в рабочей области конструктора, можно сконфигурировать на панели Properties. Это специфичный для отдельных компонентов редактор, имеющий к тому же множество компонентов расширенного редактора.

Каждый поток данных должен начинаться как минимум из одного источника и заканчиваться также хотя бы одним источником. Между этими источниками находятся компоненты преобразования данных задачи. Преобразование данных может включать в себя всевозможные операции, в том числе сортировку, конвертацию, консолидацию и т.п.

Кроме компонентов источников данных и преобразований, существуют также и зеленые стрелки потока данных, соединяющие отдельные элементы. Перед тем как поместить на рабочую область конструктора какой-либо элемент, нужно создать стрелку, поставляющую ему данные, — именно от нее зависит состав метаданных, необходимых для конфигурирования. Аналогичный процесс связан и с красными стрелками, которые направляют данные после ошибочной отработки компонента.

Редактор потоков данных используют для формирования и конфигурирования путей движения информации. Двойной щелчок на пути приводит к открытию специализированного редактора, содержащего три вкладки.

- General. В этой вкладке находятся имя, описание и аннотация. Несмотря на то что аннотация, предлагаемая по умолчанию, вполне адекватна, более сложные и взаимосвязанные потоки могут потребовать дополнительных сведений.
- Metadata. В этой вкладке отображаются метаданные каждого столбца информационного потока, в том числе тип данных и компонент источника. Эта информация доступна только для чтения, так что если нужно внести изменения, то следует оперировать с исходящими компонентами пути. Также можно использовать элементы преобразования данных.
- Data Viewers. К пути можно подключить разные средства просмотра данных с целью отладки и тестирования.

Так как поток данных существует в пределах одного задания потока управления, сбой любого компонента приведет к сбою всего задания.

Обработчики ошибок

Обработчики ошибок можно определить для любого из множества событий любого задания или контейнера потока управления. Их можно использовать для дополнительного протоколирования, обработки ошибочных ситуаций, программирования инициализации, а также множества других задач. Если при возникновении ошибки в некотором компоненте соответствующий обработчик ошибки не определен, служба интеграции проверяет родительские контейнеры, вплоть до уровня пакета, на предмет наличия такого обработчика и использует его. Если в каком-либо родительском элементе был создан обработчик ошибок, то во всех его дочерних элементах можно либо установить для параметра `DisableEventHandler` значение `true`, либо создать собственный обработчик.

Чтобы спроектировать обработчик ошибок, перейдите на вкладку Event Handler и выберите в верхнем левом списке элемент потока управления, а в правом верхнем списке — событие. После этого щелкните на активизированной ссылке в рабочей области конструктора, чтобы инициализировать событие. Логика обработки ошибки программируется точно так же, как и любой другой компонент потока управления.

Выполнение пакета в среде разработки

После того как часть пакета сформирована, ее можно протестировать непосредственно в среде разработки. Щелкните правой кнопкой мыши на пакете в Solution Explorer и выберите в контекстном меню пункт Execute Package. Пакет будет запущен в режиме отладки. В ходе выполнения пакета успешно выполненные задачи будут перекрашены в рабочей области из белого цвета (еще не выполненные) в желтый (выполняющиеся) и затем в зеленый или красный (в зависимости от успеха выполнения).



Существуют и другие удобные методы выполнения пакетов в утилите BIDS — главное убедиться, что на выполнение запускается корректный объект. Выберите в меню пункт Start Debugging, или нажмите клавишу <F5>, или щелкните на соответствующей кнопке панели инструментов — главное убедитесь, что выполняемый пакет был назначен автоматически запускаемым. Для этого щелкните на нем правой кнопкой мыши в Solution Explorer. К тому же решения, содержащие множество проектов, могут привести к непредсказуемым результатам (например, к развертыванию базы данных службы анализа), независимо от того, какой из объектов был выбран для отладки как исходный. Поверьте, что даже на этапе разработки непреднамеренный запуск шестичасовой задачи, такой как развертывание данных или построение куба, надолго испортит вам настроение.

После того как отладка началась, во вкладке Execution Results будут отображаться сообщения о ходе процесса, включая время выполнения каждого из компонентов. После завершения работы пакета он остается в режиме отладки; при этом все еще можно просмотреть информацию о его переменных и состоянии. Чтобы вернуться в режим конструктора, щелкните на кнопке Stop панели инструментов Debug, или выберите в меню пункт Debug⇨Stop Debugging, или нажмите комбинацию клавиш <Shift+F5>.

В любом задании, контейнере или пакете можно устанавливать контрольные точки. Для этого нужно щелкнуть правой кнопкой мыши на объекте и выбрать в контекстном меню пункт Edit Breakpoints. Откроется диалоговое окно (рис. 42.5), позволяющее установить контрольную точку на любом событии, связанном с данным объектом. Чаще всего выбирают события `PreExecute` (перед выполнением) или `PostExecute` (после выполнения). Если вы хотите установить контрольную точку перед выполнением некоторого компонента, выделите данный объект и нажмите клавишу <F9>. Любая контрольная точка может быть проигнорирована в нескольких ситуациях:

- до *n*-го по счету выполнения компонента;
- начиная с *n*-го по счету выполнения компонента;
- во всех случаях, кроме перечисленных выполнений компонента.

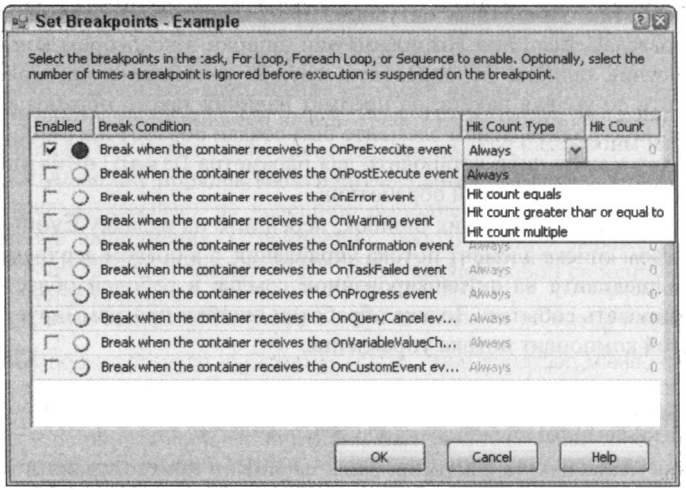


Рис. 42.5. Диалоговое окно установки контрольных точек

Когда выполнение приостанавливается в контрольной точке, в окне Locals вы можете увидеть текущие значения переменных, в окне Output — полезные сообщения и предупреждения, а в окне Execution Results — детали хода выполнения всех заданий.

Аналогом контрольных точек является окно Data Viewer. Дважды щелкните на интересующем вас пути. Далее в ходе выполнения пакета окно DataViewer будет заполняться данными; при этом выполнение пакета будет приостанавливаться. Для его продолжения нужно щелкнуть в этом окне на кнопке Go (продолжить выполнение) или Detach (отключить просмотр данных в этой точке).

Контрольные точки можно также поместить и в любой точке компонента сценария. Откройте сценарий и в интересующей вас строке установите такую точку. В этом случае служба интеграции остановит процесс именно в требуемом месте.

Элементы пакетов службы интеграции

При проектировании пакетов службы интеграции может использоваться множество элементов. Их индивидуальные черты будут подробно описаны в настоящем разделе. Общие концепции и свойства, характерные для всех элементов, были описаны в предыдущем разделе.

Диспетчеры подключений

Диспетчер подключений представляет собой оболочку строки подключения и параметров, которые следует поддерживать в подключении во время выполнения пакета. Когда определено подключение, на него можно создавать ссылки в других элементах пакета, не дублируя это определение. Таким образом, упрощается управление этой информацией и перенастройка для альтернативных платформ.

База данных

Определение подключения к базе данных с помощью одного из доступных диспетчеров требует установки нескольких ключевых параметров.

- **Provider** (поставщик). Драйвер, используемый для подключения к базе данных.
- **Server** (сервер). Сервер или имя файла, содержащие базу данных, к которой осуществляется доступ.
- **Initial Canalog** (исходный каталог). Выбор базы данных по умолчанию в источнике, содержащем множество баз.
- **Security** (защита). Используемый метод аутентификации, а также необходимые имя пользователя и пароль.

Чаще всего выбирают диспетчер подключений OLE DB, позволяющий использовать множество "родных" поставщиков СУБД SQL Server, Oracle, Jet (Access), а также длинный список других типов источников. Среди других диспетчеров подключений можно назвать следующие.



Основным достоинством службы интеграции является скорость. Среда ADO.NET предлагает больше средств, но в большинстве случаев это совершенно не то, что заставляет пользователей обращаться к службе интеграции. Большинство разработчиков по той же причине отдают предпочтение OLE DB, а также потому, что с помощью подключения ADO.NET нельзя сконфигурировать выход — только вход.

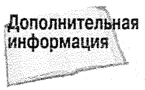
- **ADO**. Реализует уровень абстракции ADO (т.е. уровень команд, наборов данных и т.п.), базирующийся на поставщике OLE DB. ADO не используется встроенными элементами службы интеграции, но может понадобиться для специализированных задач, использующих этот интерфейс (например, для старых приложений, написанных в среде Visual Basic 6.0).
- **ADO.NET**. Реализует для выбранного подключения к базе данных уровень абстракции ADO.NET (т.е. уровень именованных параметров, объектов чтения данных и наборов данных). Несмотря на то что ADO.NET не может похвастаться такой же скоростью, как OLE DB, подключение ADO.NET позволяет выполнять сложные параметризованные сценарии, обеспечивать в памяти наборы данных для циклов `Foreach`, а также поддерживать произвольные задачи, запрограммированные с помощью таких языков, как C# и VB.NET.
- **ODBC**. Позволяет конфигурировать диспетчер подключений на основе ODBC DSN. Этот диспетчер может оказаться полезным, когда для данного источника недоступны поставщики .NET или OLE DB.
- **Analysis Services**. При доступе к существующей базе данных службы анализа этот диспетчер является аналогом OLE DB, который использует поставщика Analysis Services 9.0. В качестве альтернативы в том же решении может использоваться неразвернутая база данных службы анализа — для поддержки новой базы данных были разработаны полезные средства для пакетов. Если по какой-то причине требуется один из старых поставщиков OLAP, к нему можно получить доступ с помощью диспетчера подключений OLE DB.
- **SQL Server Mobile**. Позволяет подключиться к мобильной базе данных в файле `.SDF`.

При выполнении отдельных заданий подключение, описанное в диспетчере, открывается и закрывается в каждой из задач. Такой режим работы, настроенный по умолчанию, обеспечивает безопасную изоляцию задач, не позволяя одной задаче вмешиваться в подключения

последующих. Если нужно оставлять подключение открытым между заданиями, следует установить для параметра `RetainSameConnection` значение `true`. При достаточном внимании можно совместно использовать одну сессию подключения для нескольких задач, вручную управлять транзакциями, передавать временные таблицы и т.п.

Источники данных и представления источников данных

Диспетчеры подключений можно создавать на основе самих источников данных. Этот метод может пригодиться программистам, которые разрабатывают пакет в составе проекта, в котором несколько пакетов требуют синхронизации своих диспетчеров подключений. Также этот метод будет полезен, когда базы данных источников содержат большое количество таблиц или незнакомых структур.



Более подробно об источниках данных и их представлениях мы поговорим в главе 43.

Для создания диспетчера подключений нужно щелкнуть правой кнопкой мыши на панели Connection Manager и выбрать в контекстном меню пункт New Connection from Data Source. При этом информация о подключении будет унаследована от источника данных. Сам источник данных может быть составной частью проекта (но не решения), содержащего моделируемый пакет. При настройке заданий, использующих данный диспетчер, выбирайте его обычным образом из списка доступных подключений. Если было создано несколько представлений источника данных, то они будут перечислены как дочерние объекты этого источника и содержать, возможно, меньшее количество таблиц. Если представление источника данных было создано на основе множества таблиц из разных источников, то при выборе представления будут видимы только объекты первичного источника данных. Именованные запросы доступны для пакетного использования — это достаточно удобный механизм в случае, когда конкретное представление должно довольно часто использоваться в пакете.

Диспетчеры подключений к источникам данных создавались исключительно для удобства проектирования. После своего развертывания пакеты сами выступают в роли источников данных, как будто бы и не существует вовсе ссылок на исходные источники.

Файл

Напомним, что если на какой-либо файл или папку существует ссылка, она должна быть доступна не только на этапе проектирования, но и после развертывания пакета. При определении путей к глобальной информации и конфигурации пакетов используйте универсальное соглашение об именах (UNC). UNC — это метод идентификации пути для обеспечения доступа к объекту из любого места локальной сети, где бы ни был запущен пакет. В общем случае путь к файлу должен иметь следующий вид:

`\\имя_сервера\имя_общего_ресурса_сети\путь\файл.расширение`

Некоторые диспетчеры конфигурирования файлов перечислены ниже.

- **Flat File**. Текстовый файл, в который помещена таблица, вместе с параметрами региональных настроек и заголовками. Этот так называемый "плоский" файл может иметь один из трех форматов.
  - **С разделителями**. Данные в файле отделены друг от друга с помощью разделителей. Обычно в роли разделителя столбцов выступает запятая, а в роли разделителя строк — последовательность символов возврата каретки и перевода строки, т.е. {CR}{LF}.
  - **С фиксированной шириной**. Данные в файле имеют фиксированные размеры; при этом не используется никаких разделителей строк или столбцов. При открытии

в обычном текстовом редакторе, таком как Notepad, все данные представляются так, будто все они составляют одну строку.

- **Строчный**. Данные в файле интерпретируются с помощью фиксированной ширины всех столбцов, за исключением последнего, завершаемого символом перевода строки.

Только файлы, использующие формат с разделителями, способны интерпретировать строки с нулевой длиной как пустые.

- **Multiple Flat File**. Этот диспетчер аналогичен `Flat File`, однако позволяет выбирать множество файлов, индивидуально или с помощью символов макроподстановки. Элементам службы интеграции такие данные представляются в виде единой таблицы.
- **File**. Идентифицирует файл или папку файловой системы, не определяя содержимое. Такие указатели на файлы используются многими элементами службы интеграции. Например, задания `File System` и `FTP` используют этот диспетчер для манипуляций файлами, а задания Execute SQL — для идентификации файла, из которого следует прочитать инструкцию SQL. Конкретный тип использования (`Create file`, `Existing file`, `Create folder`, `Existing folder`) гарантирует создание корректного указателя на файл.
- **Multiple Files**. Функционально идентичен диспетчеру `File`, однако позволяет выбирать множество файлов, либо индивидуально, либо с помощью символов макроподстановки.
- **Excel**. Идентифицирует файл, содержащий группу ячеек, которые можно интерпретировать как таблицу, использующую одну или две строки для заголовков, а все следующие строки для данных.

Специализированные диспетчеры подключений

Кроме диспетчеров подключений к базам данных и файлам, существует еще ряд и других типов.

- **FTP**. Определяет подключение к FTP-серверу. В большинстве случаев для определения такого подключения достаточно указания имени сервера и регистрационных данных. Этот диспетчер используется для перемещения и удаления файлов, а также для создания и удаления каталогов на серверах FTP.
- **HTTP**. Определяет подключение к Web-службам. Введите URL-адрес файла определения WSDL Web-службы. Например, адрес

  `http://MyServer/reportserver/reportservice.asmx?wsdl`

  указывает на файл определения службы отчетности на сервере `MyServer`. Этот диспетчер используется заданием `Web Service` для доступа к методам Web-службы.
- **MSMQ**. Определяет подключение к очереди сообщений Microsoft Message Queue и используется задачей `Message Queue` для приема и отправки сообщений.
- **SMO**. Определяет имя и метод аутентификации для использования заданиями перемещения баз данных (`Transfer Objects`, `Transfer Logins` и т.п.).
- **SMTP**. Определяет имя сервера SMTP, используемого в задании отправки почты. Старые версии серверов SMTP могут не поддерживать все команды, необходимые для отправки сообщений электронной почты из службы интеграции.
- **WMI**. Определяет подключение к серверу, необходимое для задания Windows Management Instrumentation, позволяющего накапливать протоколируемые данные о событиях.

Элементы потока управления

Вкладка Control Flow предлагает среду для создания общего потока заданий в пакете. Следующие элементы являются конструктивными блоками рабочего потока.

Контейнеры

Контейнеры реализуют работу важных конструктивных элементов пакетов службы интеграции, включая итерации групп заданий и изоляцию обработки событий и ошибок.

В дополнение к контейнерам конструктор службы интеграции позволяет создавать группы заданий. Создание группы выполняется с помощью выделения нескольких элементов потока управления, щелчка правой кнопкой мыши на любом из выделенных объектов и выбора в контекстном меню пункта Group. Выделенные задания обрамляются блоком группы и могут быть свернуты в единый элемент. Следует особо отметить, что группа не имеет собственных свойств и не может участвовать в иерархии заданий контейнера. Короче говоря, группу можно рассматривать как визуальный элемент, не оказывающий влияния на выполнение пакета.

В службе интеграции доступны следующие типы контейнеров.

- **TaskHost**. Этот контейнер не виден в пакете. Он неявно обслуживает все задачи, не вошедшие ни в один контейнер тем или иным образом. Знание о наличии этого невидимого контейнера поможет вам понять режим обработки ошибок и событий, принятый по умолчанию.
- **Sequence**. Объединяет задания в один блок, не реализуя каких-либо итеративных функций. Он просто обеспечивает заданиям единый контекст обработки ошибок и событий, а также доступ к определенным на уровне контейнера переменным; позволяет на этапе отладки отключить весь контейнер.
- **For Loop**. Обладает всеми свойствами контейнера `Sequence`, дополнительно обеспечивая циклическое выполнение включенных в него заданий (подобно циклам в C#). Например, если свойству InitExpression этого контейнера присвоить выражение `@LoopCount=0`, свойству EvalExpression — `@LoopCount<3`, а свойству AssignExpression — `@LoopCount=@LoopCount+1`, то содержимое контейнера будет выполнено три раза, при этом переменная `@LoopCount` будет последовательно принимать значения `0`, `1` и `2`.
- **Foreach Loop**. Содержимое контейнера выполняется итеративно, однако на основании различных списков элементов:
  - **File** — на основании списка файлов, содержащихся в заданном каталоге (при этом могут использоваться символы макроподстановки);
  - **Item** — на основании всех элементов списка, созданного вручную;
  - **ADO** — на основании всех строк переменной, содержащей набор записей ADO или набор данных ADO.NET;
  - **ADO.NET Schema Rowset** — на основании всех элементов набора строк схемы;
  - **Nodelist** — на основании всех узлов, содержащихся в результирующем наборе XPath;
  - **SMO** — на основании списка серверных объектов (в частности, заданий, баз данных и файловых групп).

Список, на основании которого выполняется итерация, определяется на странице Collections, а затем отображается на конкретную переменную. Например, цикл `File` требует наличия одной строковой переменной, отображенной на индекс `0`, а цикл `ADO` требует *n* переменных (т.е. для каждого столбца) с индексами от `0` до *n*`-1`.

Задания потока управления

В поток управления могут быть включены следующие задачи.

- **ActiveX Script**. Позволяет включить в поток старые сценарии, написанные на языках Visual Basic и Java; для новых сценариев лучше использовать задание `Script`. Если такое возможно, подумайте о переводе старых сценариев на новые рельсы.
- **Analysis Services Execute DDL**. Отправка сценариев ASSL в службу анализа для создания, изменения и обработки кубов и структур раскрытия данных. Часто такие сценарии могут создаваться в редакторе сценариев утилиты Management Studio.
- **Analysis Services Processing Task**. Идентифицирует базы данных службы анализа, список обрабатываемых объектов и операционные параметры.
- **Bulk Insert**. Реализует быстрый механизм загрузки "плоских" файлов в таблицы базы данных без преобразований. Определите файл источника и таблицу назначения — это минимальная конфигурация. Если в качестве источника выступает файл с разделителями, определите разделители строк и столбцов. В противном случае создайте файл формата, описывающий композицию источника данных. Определите столбцы сортировки, соответствующие кластеризованным ключам таблицы назначения, что позволит минимизировать фрагментацию таблицы. Ошибочные строки не перенаправляются; их обнаружение приводит к ошибочному завершению задания.



Пользователи, знакомые со службой преобразования данных (DTS) в версии SQL Server 2000, будут пытаться установить такие параметры, как `Batchsize` (размер пакета), чтобы повысить быстродействие и поддерживать размеры журнала транзакций на минимальной отметке. Однако следует заметить, что в службе интеграции изменение этих параметров не желательно, так как часто приводит к эффекту, противоположному тому, к какому приводил в DTS.

- **Data Flow**. Обеспечивает гибкую структуру для загрузки, преобразования и хранения данных, конфигурируемую во вкладке Data Flow. Подробнее о компонентах потока данных мы поговорим чуть позже в этой главе.
- **Data Mining Query**. Запуск запросов прогнозирования относительно существующих учебных моделей раскрытия данных. Во вкладке Mining Model укажите подключение к базе данных службы анализа и имя структуры раскрытия данных. Во вкладке Build Query введите запрос DMX; при этом, если необходимо, щелкните на кнопке Build New Query, чтобы запустить построитель запросов. Запрос DMX может быть параметризованным; при этом имена параметров помещаются в строку запроса в форме *@имя_параметра*. Если используются параметры, во вкладке Parameter Mapping отобразите их имена (опуская префикс @) на соответствующие имена переменных. Результаты обрабатываются либо путем направления их переменным во вкладке Result Set, либо таблицам базы данных во вкладке Output (или то и другое одновременно).
  - Результирующие наборы данных, состоящие из одной строки, могут направляться непосредственно переменным. При этом во вкладке Result Set для определенного столбца результирующего набора назначается целевая переменная и выбирается тип результата `Single Row`.
  - Многострочные результирующие наборы данных могут сохраняться в переменных типа `Object` для дальнейшего использования в контейнерах `Foreach` или для других целей. Во вкладке Result Set укажите в имени столбца нуль, определите имя переменной и установите тип `Full Result Set`.

- Независимо от отображения на переменные, как однострочные, так и многострочные результирующие наборы данных могут быть направлены в таблицу. При этом во вкладке Output определяется подключение к базе данных и имя нужной таблицы.

- **Execute DTS 2000 Package**. Позволяет запускать на выполнение старые пакеты DTS в составе рабочего потока службы интеграции. Укажите местонахождение пакета DTS, регистрационную информацию и способ отображения внутренних переменных на внешние. Дополнительно, если пакет DTS идентифицирован, он может быть загружен как часть пакета службы интеграции.

- **Execute Package**. Выполняет заданный пакет службы интеграции. Это позволяет разбивать крупные пакеты на ряд мелких, предназначенных для многократного использования. Привлечение дополнительных пакетов накладывает на системные ресурсы дополнительную нагрузку, поэтому дважды подумайте, прежде чем создавать дочерние пакеты. Например, использование для одного файла одного-двух пакетов вполне разумно, в то время как использование отдельного пакета для каждой строки — определенно нет. Дочерние пакеты участвуют в транзакции, если для этого сконфигурировано задание `Execute Package`. Переменные, доступные для задания `Execute Package`, также доступны и для дочерних пакетов, если предусмотреть в них соответствующую конфигурацию, отображающую каждую переменную родительского пакета на локальную переменную дочернего (если такое необходимо).

- **Execute Process**. Выполнение внешней программы или пакетного файла. В параметре Executable задайте выполняемую программу, обязательно включая расширение и полный путь к файлу, если тот не содержится в системной переменной среды `PATH`. Поместите в параметр Arguments все переключатели и аргументы, которые обычно следуют за именем программы в командной строке. Установите дополнительные параметры, такие как WorkingDirectory и SuccessValue, чтобы служба интеграции знала о работоспособности процесса. Параметр StandardInputVariable позволяет передать в приложение текст переменной, помещая ее в поток StdIn. Этот параметр используется для таких программ, как `find` и `grep`. Параметры StandardOutputVariable и StandardErrorVariable позволяют приложению помещать сообщения (обычные и об ошибках) в переменные.

- **Execute SQL**. Выполнение сценария или запроса SQL, при необходимости возвращая их результаты в переменные. На странице General редактора установите параметры ConnectionType и Connection в базу данных, относительно которой будет выполняться запрос. Параметр SQLSourceType определяет способ ввода запроса.

  - **DirectInput**. Ввод запроса в параметре SQLStatement либо с клавиатуры, либо щелчком на эллипсе, чтобы открыть таким образом текстовое поле, либо щелчком на кнопке Browse для извлечения запроса из файла, либо щелчком на кнопке Build Query для открытия специализированного построителя запросов.

  - **File Connection**. Задайте имя файла, из которого во время выполнения пакета будет прочитан запрос.

  - **Variable**. Определите имя переменной. содержащей выполняемый запрос.

  Запрос можно сделать динамическим либо с помощью параметров, либо с помощью свойства SQLExpression страницы Expressions редактора. Использование выражений — несколько более сложный, но гораздо более гибкий метод, чем использование параметров. Последние предназначены только для подстановки в предложения `WHERE`, причем (за исключением подключений ADO.NET) только в довольно простых запросах. Если используются параметры, то запрос вводится с маркером для каждого

используемого параметра, после чего этот параметр отображается на некоторую переменную во вкладке Parameter Mapping. Маркеры параметров и их отображение варьируются для разных диспетчеров подключений.

- **OLE DB**. Создайте запрос, помечая символом вопросительного знака местоположение каждого параметра. При этом соблюдайте очередность: используйте `0` для первого параметра, `1` — для второго и т.д.
- **ODBC**. То же, что и OLE DB, за исключением того, что нумерация параметров начинается с единицы, а не с нуля.
- **ADO**. Маркируйте местоположение каждого параметра символом вопросительного знака, за которым следует некоторое нечисловое значение. В данном случае это порядок, в котором значения будут подставляться в параметры.
- **ADO.NET**. Напишите запрос, как если бы параметры были переменными, объявленными в T-SQL (например, `SELECT name FROM table WHERE id=@ID`), а затем обратитесь к параметру по имени для отображения.

- Параметр **ResultSet** (на странице General) определяет, как результаты запроса возвращаются переменным.
  - **None**. Результаты не заносятся в переменные.
  - **Single row**. Результаты одиночных запросов (т.е. обрабатывающих одну строку) сохраняются в переменных напрямую. Во вкладке Result Set каждому возвращаемому столбцу назначаются переменные назначения. Как и в случае с входными параметрами, имена результатов зависят от типа диспетчера подключения. В подключениях ADO, ADO.NET и OLE DB столбцы отображаются в прямом порядке, начиная с нуля. В ODBC также используется числовое отображение, но начинающееся с единицы. К тому же подключения ADO и OLE DB позволяют столбцам отображаться по имени, а не порядковому номеру.
  - **Full result set**. Многострочные результирующие наборы данных сохраняются в переменных типа `Object` для дальнейшего использования в циклах `Foreach` и прочих конструкциях. Во вкладке Result Set отобразите результат с именем `0` (нуль) на объектную переменную, а тип результата установите в Full Result Set.
  - **XML**. Результаты сохраняются в документах объектной модели XML для дальнейшего использования в циклах `Foreach` и прочих конструкциях. Во вкладке Result Set отобразите результат с именем `0` (нуль) на объектную переменную, а тип результата установите в Full Result Set.
- **File System Task**. Этот элемент реализует множество операций над файлами (копирование, перемещение, удаление, переименование и установка атрибутов) и папками (копирование, создание, удаление, удаление содержимого, перемещение). Файлы или папки источника и назначения можно определить с помощью диспетчера подключений либо с помощью строковой переменной, содержащей полный путь. Не забудьте при конфигурировании диспетчера подключения установить соответствующий тип использования. Установите параметр OverwriteDestination или UseDirectoryIfExists, чтобы определить порядок замещения уже существующих объектов.
- **FTP**. Поддержка самой распространенной функциональности FTP, включая отправку, получение и удаление файлов, а также создание и удаление каталогов. С помощью диспетчера подключения FTP определите серверы. Любой удаленный файл или путь должен быть определен с помощью непосредственного ввода либо строковой переменной. В именах файлов допускаются символы макроподстановки. В параметре

OverWriteFileAtDest определите, следует ли замещать уже существующий файл, а в параметре IsAsciiTransfer режим передачи —двоичный файл или ASCII.

- **Message Queue**. Отправка или получение сообщений из очереди MSMQ. Задайте подключение к очереди, тип сообщения и операцию: получение или отправку.
- **Script**. Это задание позволяет внедрять в пакет программный код Visual Basic .NET. Среди параметров этого задания стоит упомянуть следующие.
  - **PrecompileScriptIntoBinaryCode**. При сохранении сценарий компилируется конструктором, что увеличивает вероятность корректного выполнения в любой среде. В то же время за это приходится расплачиваться увеличением размера файла пакета.
  - **ReadOnlyVariables/ReadWriteVariables**. В этих параметрах перечисляются через запятую переменные, доступные только для чтения или для чтения и записи. Попытка обращения к любой переменной, не вошедшей в данный список, приведет к ошибке выполнения. Имена переменных чувствительны к регистру символов, так что `myvar` и `MyVar` — две совершенно разные переменные.
  - **EntryPoint**. Имя класса, содержащего точку входа в сценарий. Обычно не имеет смысла изменять имя, предложенное по умолчанию (`ScriptMain`). При этом генерируется оболочка программного кода:

```
Public Class ScriptMain
  Public Sub Main()
     ' Программный код вводится здесь
     Dts.TaskResult = Dts.Results.Success
  End Sub
End Class
```

После выполнения сценарий должен вернуть результат `Dts.TaskResult`, сообщающий об успешном или ошибочном выполнении программного кода (значение `Dts.TaskResults.Success` или `Dts.TaskResults.Failure` соответственно). Ссылки на переменные описываются с помощью коллекции `Dts.Variables`. Например, чтобы сослаться на переменную `MyVar`, нужно использовать свойство `Dts.Variables("MyVar").Value`. Учтите, что эта коллекция чувствительная к регистру символов, поэтому ссылка на `"myvar"` не вернет значение переменной `MyVar`. Объект `Dts` является экземпляром класса `ScriptObjectModel`, который имеет ряд других полезных членов: `Dts.Connections` для обращения к диспетчерам подключений; методы `Dts.Events.Fire` — для генерации событий и метод `Dts.Log` — для выполнения записи в журнал. Более подробно эта тема обсуждается в статье "Interacting with the Package in the Script Task" утилиты SQL Server 2005 Books Online.

- **Send Mail**. Отправка текстового электронного сообщения на сервер SMTP. Заполните в диспетчере подключений `SMTP` все необходимые поля: To, From и т.п. Если сообщение должно быть отправлено по множеству адресов, разделите их запятыми (но не точками с запятыми). Источник тела сообщения указывается в параметре MessageSourceType: `Direct Input` — если текст сообщения будет вводиться в поле `MessageSource`; `File Connection` — если текст будет считан из файла в ходе выполнения; `Variable` — если содержимое будет извлекаться из строковой переменной. Вложения в письмо вводятся как разделенные вертикальной чертой полные спецификации. Отсутствие какого-либо из файлов вложения приводит к ошибке выполнения задания.
- **Transfer Database**. Копирование или перемещение полной базы данных из экземпляра SQL Server 2005 или SQL Server 2000 в экземпляр SQL Server 2005. Выберите или более

быстрый метод — `DatabaseOffline` (который отсоединяет базы данных, копирует файлы, а затем снова подключает к базе), или более медленный — `DatabaseOnline` (использующий для создания целевой базы данных объектную модель SMO). Идентификация серверов источника и назначения осуществляется с помощью диспетчеров подключений. Для метода `DatabaseOffline` определите имена баз данных источника и назначения, а также пути к соответствующим файлам данных. Метод `DatabaseOnline` требует ту же информацию, а также *путь к общему сетевому ресурсу* для каждого файла источника и назначения. Определение путей UNC к сетевым ресурсам является самым распространенным методом, однако пакеты могут напрямую обращаться к файлам, расположенным локально на том же сервере. Использование метода `DatabaseOnline` требует, чтобы все объекты, связанные с базой данных (такие как регистрационные записи), были предоставлены перед перемещением базы данных.

- **Transfer Error Messages**. Передача между серверами произвольных сообщений об ошибках. Серверы источника и назначения определяются с помощью менеджеров подключений `SMO`; также укажите список передаваемых сообщений.
- **Transfer Jobs**. Копирование заданий SQL Agent из SQL Server 2000 или 2005 в экземпляр SQL Server 2005. Серверы источника и назначения определяются с помощью менеджеров подключений `SMO`; также укажите список перемещаемых заданий. Все ресурсы, необходимые заданиям (например, базы данных), должны быть доступны для успешного копирования.
- **Transfer Logins**. Копирование регистрационных записей из SQL Server 2000 или 2005 в экземпляр SQL Server 2005. Серверы источника и назначения определяются с помощью менеджеров подключений `SMO`; также укажите список перемещаемых регистрационных записей. Этот список должен состоять из выбранных регистрационных записей сервера источника или всех регистрационных записей, имеющих доступ к выбранным базам данных (см. параметр `LoginsToTransfer`).
- **Transfer Master Stored Procedures**. Копирование всех дополнительных хранимых процедур из базы данных `master` одного сервера в базу `master` другого. Определите серверы источника и назначения с помощью менеджеров подключений `SMO`. Также вы можете либо выбрать все дополнительные хранимые процедуры, либо копировать только избранные.
- **Transfer Objects**. Копирование любых объектов уровня базы данных из SQL Server 2000 или 2005 в экземпляр SQL Server 2005. Определите серверы источника и назначения с помощью менеджеров подключений `SMO`, а также базы данных на каждом из серверов. Для каждого из типов объектов выберите либо копирование всех подобных объектов, либо создайте список объектов и параметров копирования (таких как DropObjectFirst или CopyIndexes).
- **Web Service**. Вызов Web-службы и сохранение результатов в файле или переменной. Определите диспетчер подключения `HTTP` и файл для сохранения информации WSDL. Если диспетчер подключений указывает непосредственно на файл WSDL (например, `http://MyServer/MyService/MyPage.asmx?wsdl` для Web-службы `MyService`, находящейся на сервере `MyServer`), воспользуйтесь кнопкой Download WSDL, чтобы создать локальную копию этого файла. В противном случае вручную перепишите и создайте локальный файл WSDL. Установка для параметра OverwriteWSDLFile значения `true` приведет к сохранению последнего описания Web-службы в локальном файле при каждом запуске задания.

Когда информация о подключении будет задана, переключитесь на страницу Input и выберите службу и выполняемый метод, а также введите необходимые параметры метода. На странице Output определите диспетчер доступа к файлу или переменную. При выборе особое внимание уделите типу данных — он должен быть совместимым с результатом, возвращаемым Web-службой.

- **WMI Data Reader**. Выполнение запроса Windows Management Instrumentation к серверу для извлечения журнала событий, конфигурации и прочей служебной информации. Выберите диспетчер подключения WMI и определите запрос WQL (например, `SELECT * FROM win32_ntlogevent WHERE logfile - 'system' AND timegenerated>'20050911'` для отбора всех системных событий, начиная с 11 сентября 2005 года), вводимый вручную, извлекаемый из файла или содержащийся в строковой переменной. Выберите формат вывода, установив для параметра OutputType значение `'Data table'` для разделенного запятыми списка значений, значение `'Property name and value'` — для одной комбинации "параметр–значение" для каждой строки или `'Property value'` — для одного значения свойства в каждой строке без имени. С помощью параметров `DestinationType` и `Destination` вы можете отправить результаты запроса в файл или строковую переменную.
- **WMI Event Watcher**. Этот элемент аналогичен `WMI Data Reader`, но вместо возвращения данных ожидает возникновения события, определенного в запросе WQL. Когда происходит событие или истекает срок его ожидания, служба интеграции генерирует событие `WMIEventWatcherEventOccured` или `WMIEventWatcherEventTimeout`. Для каждого из этих событий определите действие (занесение в журнал и генерация события или только протоколирование) и диспозицию задания (вернуть успешное завершение, вернуть ошибку или снова ожидать). В свойстве Timeout установите время ожидания (значение 0 указывает на отсутствие пределов).
- **XML**. Выполнение операций над документом XML, в том числе сравнение двух документов, их слияние, применение разности, проверка документа относительно определения DTD, выполнение запросов XPath и преобразований XSLT. Выберите документ источника как непосредственный ввод, файл или строковую переменную, а результат — как файл или строковую переменную. Установите остальные параметры, характерные для выбранного типа.

Задания плана обслуживания

Задания плана обслуживания предлагают те же элементы, которые использовались для создания планов в дополнительных средствах пакета. Эти задания используют диспетчер подключений ADO.NET для идентификации обслуживаемого сервера, однако любая база данных, указанная в диспетчере, будет замещена базами, идентифицированными в каждом из заданий плана. Все вопросы относительно действий, выполняемых любым из заданий, разрешаются щелчком на кнопке View T-SQL задания.



Более подробно о задачах обслуживания базы данных см. в главе 37.

Доступные задачи описаны ниже.

- **Back Up Database**. Создание резервной копии одной или нескольких баз данных средствами SQL Server.
- **Check Database Integrity**. Выполнение команды `DBCC CHECKDB`.

- **Execute SQL Server Agent Job**. Запуск выбранного задания службы SQL Server Agent посредством хранимой процедуры `sp_start_job`.
- **Execute T-SQL Statement**. Упрощенный вариант выполнения инструкции T-SQL. Результаты выполнения и наборы данных не возвращаются. Для более сложных запросов следует использовать задание **Execute SQL**.
- **History Cleanup**. Удаление устаревших элементов из журналов резервирования/восстановления, плана обслуживания и службы SQL Server Agent.
- **Maintenance Cleanup**. Сжатие старых планов обслуживания, резервных копий и прочих файлов.
- **Notify Operator**. Выполнение хранимой процедуры `sp_notify_operator`, посылающей сообщение избранному дежурному оператору, определенному в SQL Server.
- **Rebuild Index**. Перестройка всех индексов таблиц и/или представлений выбранной базы данных с помощью инструкции `ALTER INDEX REBUILD`.
- **Reorganize Index**. Реорганизация либо всех, либо только избранных индексов в выбранной базе данных с помощью инструкции `ALTER INDEX...REORGANIZE`. Как правило, приводит к уменьшению объема крупных объектов данных.
- **Shrink Database**. Выполнение команды сжатия базы данных `DBCC SHRINKDATABASE`.
- **Update Statistics**. Выполнение инструкции обновления статистики `UPDATE STATISTICS` для столбцов и/или индексов выбранных баз данных.

Компоненты потока данных

В этом разделе будут описаны отдельные компоненты, которые можно сконфигурировать в задании потока данных, в том числе источники данных, участвующие в потоке, преобразования данных и приемник, получающий трансформированные данные. Общую информацию о конфигурировании потока данных см. выше в этой главе.

Источники

Источники потока данных поставляют строки данных, которые проходят через задание `Data Flow`. Щелчок правой кнопкой мыши на источнике в рабочей области конструктора покажет, что для любого источника доступны два варианта редактирования: Edit (basic) и Show Advanced Editor (хотя порой щелчок на варианте стандартного редактора приводит к открытию расширенного). Последовательность действий в процессе конфигурирования источника данных представлена страницами базового редактора.

- Connection Manager. Определение таблиц, файлов, представлений и запросов, поставляющих данные для этого источника. Некоторые источники способны принимать имя таблицы или строку запроса из переменной.
- Columns. Выберите столбцы, которые должны передаваться в поток данных. При желании вы можете изменить имена столбцов в потоке.
- Error Output. Определите, что нужно делать с каждым из столбцов при возникновении ошибки. Любой тип ошибок можно игнорировать, можно прерывать работу компонента (предусмотрено по умолчанию), а также направлять проблемную строку в поток ошибок.

Расширенный редактор обладает теми же возможностями, что и стандартный, только в несколько ином формате. К тому же расширенный редактор позволяет более точно управлять

входными и выходными столбцами, в том числе их именами и типами данных. Также расширенный редактор позволяет сортировать строки, отправляемые в поток данных. Для этого во вкладке Input and Ouput Properties выберите корневой узел дерева и установите для параметра IsSorted значение `True`. После этого выберите каждый из выходных столбцов (т.е. столбцов потока данных), участвующих в сортировке, и установите их параметры SortKeyPosition в последовательные целые числа, начиная с единицы. Если сортировка по определенному столбцу должна осуществляться по убыванию, задайте отрицательное значение параметра SortKeyPosition. Например, если в столбцах `Date` и `Category` присвоить этому параметру значения `-1` и `2` соответственно, сортировка по дате будет выполнена по убыванию, а по категории — по возрастанию.

Доступными источниками являются следующие.

- **OLE DB**. Самый предпочтительный метод чтения данных базы. Он требует наличия диспетчера подключений `OLE DB`.
- **DataReader**. Для чтения данных базы использует диспетчер подключений `ADO.NET`. Обязательна явная строка запроса, определяющая извлекаемые данные.
- **Flat File**. Требует наличия диспетчера подключений `Flat File`. В файлах с разделителями строки нулевой длины трансформируются в пустые значения потока данных, если для параметра RetainNulls установлено значение `true`.
- **Excel**. Использует диспетчер подключений Excel и либо рабочий лист, либо именованный диапазон в качестве таблицы. Инструкция SQL может быть создана с помощью кнопки Build Query, позволяющей отобрать подмножество строк. Типы данных назначаются всем столбцам на основе идентификации нескольких первых строк.
- **Raw**. Читает файл, записанный в депозитарии `Integration Services Raw File` (см. следующий раздел) в обработанном формате. Это делает данный метод извлечения данных предельно быстрым. Так как данные уже были единожды обработаны, данный метод не требует обработки ошибок и конфигурирования выходного потока. Имя входного файла задается напрямую, без использования диспетчера подключений.
- **XML**. Читает простой файл XML и подставляет его в поток данных в виде таблицы, используя либо встроенную схему (т.е. заголовок файла XML, описывающий имена столбцов и типы данных), либо файл XSD (т.е. файл определения схемы XML). Для доступа к файлу не используется диспетчер подключений. Вместо этого задайте имя входного файла, файла схемы (если таковой присутствует) и определите, встроена ли схема в файл XML (установите для параметра UseInlineSchema значение `true` или установите соответствующий флажок в стандартном редакторе).
- **Script**. Компонент сценария может выступать в роли источника, приемника или узла преобразования потока данных. Используйте сценарии в качестве источника тестовых данных или для форматирования сложных внешних источников. Например, плохо отформатированный текстовый файл с помощью сценария можно прочитать и разобрать на составные части, оформив их в виде столбцов. Вначале перетащите компонент сценария в рабочую область конструктора и выберите в открывшемся диалоговом окне тип компонента Source. На странице Inputs and Outputs редактора добавьте необходимое количество выходов, переименовав их по своему усмотрению. В каждом выходе определите соответствующие столбцы, уделив особое внимание выбору типа данных.

  На странице Script редактора перечислите переменные, доступные в сценарии только для чтения или для чтения и записи. Для этого разделите эти переменные запятыми в параметрах ReadOnlyVariables и ReadWriteVariables соответственно. Оставьте для параметра PreCompile предложенное по умолчанию значение `True`, если размер па-

кета на диске не вызывает тревоги. Щелкните на кнопке Script и обратите внимание в программном коде на то, что программируемый главный метод замещает собой метод CreateNewOutputRows, как показано в следующем примере:

```
Public Class ScriptMain
  Inherits UserComponent
  '-- Создание 20 строк случайных целых чисел от 1 до 100
  Public Overrides Sub CreateNewOutputRows()
    Randomize()
    Dim i As Integer
    For i = 1 To 20
      Output0Buffer.AddRow()
      Output0Buffer.RandomInt = CInt(Rnd() * 100)
    Next
  End Sub
End Class
```

Этот пример работает для единственного выхода с именем по умолчанию Output 0, содержащего один целочисленный столбец RandomInt. Обратите внимание на то, что все выходы представлены как *имя*+"Buffer", при этом внедренные пробелы из имени удалены. Новые строки добавляются с помощью метода AddRow, а обращение к столбцам производится как к свойствам выхода. Также представлено дополнительное свойство для каждого из столбцов — суффикс _IsNull (например, Output0Buffer.RandomInt_IsNull) для маркировки значения как пустого.

Чтение данных из внешнего источника требует выполнения некоторых дополнительных действий, в том числе идентификации диспетчера подключений на странице Connection Managers, на которые будут выполняться ссылки в сценарии. К тому же в сценарии должны быть замещены и некоторые другие методы: AcquireConnections и ReleaseConnections — для открытия и закрытия подключений, а также PreExecute и PostExecute — для открытия и закрытия любых наборов данных, объектов DataReader и т.п. Полноценные примеры программного кода и методы диспетчеров подключений вы можете найти в статье "Examples of Specific Types of Script Components" утилиты SQL Server Books Online.

Приемники

Приемники потока данных представляют собой место, куда записываются данные, преобразованные заданиями потока данных. Их конфигурация аналогична источникам, к тому же они используют те же основной и расширенный редакторы и три общих этапа конфигурирования.

- **Connection Manager**. Здесь определяются конкретные таблицы, файлы, представления и запросы, в которые записываются данные. Некоторые приемники допускают нахождение имени таблицы в переменной.
- **Columns**. Отображение столбцов потока данных на соответствующие столбцы приемника.
- **Error Output**. Здесь определяются действия над строкой, если вставка ее в приемник привела к ошибке. Строку можно игнорировать, можно завершить работу компонента по ошибке (принято по умолчанию), а также перенаправить проблемную строку в поток ошибок.

Доступны следующие приемники.

- **OLE DB**. Запись строк в таблицу, представление или инструкцию SQL, для которых существует драйвер OLE DB. Имена представлений и таблиц можно задать напрямую или хранить в строковой переменной, при этом для каждого можно при желании ис-

пользовать так называемую ускоренную загрузку, которая способна уменьшить время выполнения за счет применения определенных параметров и характера самого набора данных. Параметры ускоренной загрузки приведены ниже.

- **Keep Identity**. Когда таблица назначения содержит столбец идентичности, установка данного параметра позволяет этому идентификатору быть замещенным вставляемыми значениями (`SET IDENTITY_INSERT ON`), или этот столбец должен быть исключен из отображаемых, чтобы СУБД SQL Server могла сгенерировать новые значения идентичности.
- **Keep nulls**. Этот параметр позволяет загружать пустые значения вместо (как обычно) предусмотренных по умолчанию.
- **Table lock**. Во время выполнения устанавливается блокировка на уровне таблицы.
- **Check constraints**. Активизация ограничений проверки (например, допустимого диапазона значений) для вставляемых строк. Следует отметить, что остальные типы ограничений (на уникальность, запрет пустых значений, а также ограничений первичного и внешнего ключей) не могут быть отключены. Загрузка данных с отключенными ограничениями проверки приведет к тому, что SQL Server пометит их как ненадежные.
- **Rows per batch**. Явное указание размера пакета учитывается при построении плана запроса, однако не изменяет размер транзакций, используемых для помещения строк в таблицу назначения.
- **Maximum insert commit size**. Аналогично параметру `BatchSise` задания `Bulk Insert`, этот параметр указывает на максимальное количество строк, включаемое в одну транзакцию. Если этот параметр оставить в нулевом значении (как принято по умолчанию), то весь набор данных загружается в пределах одной транзакции.



Те, кто знаком со службой преобразования данных DTS, будут пытаться установить такие параметры, как количество строк в пакете и максимальный размер транзакций, для увеличения скорости и минимизации размеров журнала транзакций. Однако изменение значений, принятых в этих параметрах по умолчанию, в службе интеграции не является необходимым и часто оказывает эффект, противоположный тому, который достигался в службе DTS.

- **SQL Server**. Этот приемник использует тот же механизм ускоренной загрузки, что и задание `Bulk Insert`, однако ограничен тем, что пакет должен выполняться на том же сервере, где находится целевая таблица или представление. При некоторых обстоятельствах скорость может превосходить достижимую в задании `OLE DB`. Параметры этого задания такие же, как были описаны для `Bulk Insert`.
- **DataReader**. Поток данных становится доступным для объекта `DataReader` модели ADO.NET. Этот объект может быть открыт другими приложениями, в частности службой отчетности, для чтения выходных данных пакета.
- **Flat File**. Записывает поток данных в файл, определенный в диспетчере подключений `Flat File`. Так как этот файл описан в диспетчере подключений, для объекта назначения доступен ограниченный набор параметров. Вы можете указать, следует ли замещать существующий файл, следует ли применить к заголовку файла сопровождающий текст.
- **Excel**. Отправка строк из потока данных в рабочий лист или именованный диапазон рабочей книги с помощью диспетчера подключений `Excel`. Следует заметить, что рабочий лист Excel способен содержать не более 65536 строк данных, первая из которых отводится для хранения информации заголовка. Все строки должны поддерживать

кодовую таблицу Unicode, поэтому любые типы DT_STR следует предварительно преобразовать в DT_WSTR.

- **Raw**. Запись строк из потока данных в формат службы интеграции, подходящий для быстрой загрузки компонентом источника Row. При этом не используется диспетчер подключений. Вместо этого выбирается режим доступа либо явным указанием имени файла, либо его предоставлением в строковой переменной. Установите параметр записи WriteOptions в одно из следующих значений.
  - **Append**. Добавление данных в существующий файл, предполагая, что они соответствуют уже содержащимся в нем.
  - **Create always**. Всегда создается новый файл.
  - **Create once**. Файл создается один раз, а затем дополняется. Этот параметр полезен в циклах, в которых постоянно выполняется запись в один и тот же приемник в пределах пакета.
  - **Truncate and append**. Сохранение метаданных существующего файла, но замена в нем информационных данных.
- **Recordset**. Запись потока данных в переменную. Сохраненная как набор записей, переменная объекта может использоваться как источник для циклов Foreach и прочих операций в пределах пакета.
- **SQL Server Mobile**. Запись строк из потока данных в таблицу базы данных SQL Mobile. Конфигурация выполняется с помощью указания диспетчера подключения SQL Server Mobile, указывающего на соответствующий файл .SDF, а также имени таблицы во вкладке Component Properties.
- **Dimension Processing** и **Partition Processing**. Эти задания позволяют наполнить кубы службы анализа без предварительного заполнения реляционного источника данных, на которых они построены. Укажите нужный диспетчер подключений Analysis Services, а затем выберите размерность или раздел, который вас интересует. После этого выберите один из следующих режимов обработки.
  - **Add/Incremental**. Минимальная обработка, необходимая для добавления новых данных.
  - **Full**. Полная переработка структуры и данных.
  - **Update/Data-only**. Замена данных без обновления структуры.
- **Data Mining Model Training**. Предоставление тестовых данных для существующей структуры раскрытия данных с целью подготовки их для запросов прогнозирования. Определите диспетчер подключений Analysis Services и целевую структуру раскрытия в этой базе данных. Во вкладке Columns отобразите тестовые данные на соответствующие атрибуты структуры раскрытия.
- **Script**. Сценарий может также использоваться и в качестве приемника данных; при этом процесс конфигурирования в точности совпадает с тем, который был описан для источника. Обычно сценарий используют в качестве приемника тогда, когда форматирование вывода не может обеспечить ни один из стандартных приемников. Например, файл, подходящий для ввода в программу на языке COBOL, может быть сгенерирован из стандартного потока данных. Для этого переместите компонент сценария в рабочую область конструктора, выбрав из всплывающего окна выбора типа компонента вариант Destination. Идентифицируйте интересующие вас столбцы и сконфигурируйте параметры, как было описано выше. После щелчка на кнопке Script для доступа к про-

граммному коду вы увидите главную процедуру, носящую имя Input с номером выхода и суффиксом _ProcessInputRow (например, Input0_ProcessInputRow). Отметим, что объект строки передается в эту процедуру в качестве аргумента, обеспечивая столбец ввода информацией для каждой строки (например, Row.MyColumn и Row.MyColumn_IsNull). Конфигурация подключения и подготовительные операции те же, что были описаны для одноименного источника. Полноценные примеры программного кода и методы диспетчера подключений вы можете найти в статье "Examples of Specific Types of Script Components" ресурса SQL Server Books Online.

Преобразования

Находясь между источником и приемником, элементы преобразования обеспечивают трансформацию данных из того, что было прочитано из источника, в то, что на самом деле необходимо. Любая трансформация требует на входе один или несколько потоков данных и обеспечивает на выходе информацией один или несколько приемников или других элементов преобразований. Подобно источникам и приемникам, многие преобразования реализуют некоторый метод конфигурирования потока строк, преобразование которых завершилось ошибкой. Многие элементы преобразования обеспечены стандартными и расширенными редакторами, позволяющими конфигурировать компонент, при этом в базовом редакторе доступны обычные конфигурации.

Стандартные преобразования, доступные в задании потока данных, описаны ниже.

- **Aggregate**. Подобно предложению GROUP BY в языке SQL, позволяет генерировать максимальное, минимальное, среднее и другие итоговые значения из потока данных. Ввиду своей природы преобразование Aggregate не внедряется в поток, а дает результат в виде консолидированных строк. Работа с этим элементом начинается во вкладке Aggregations, где выбираются консолидируемые столбцы. Если необходимо включить нескольких копий одного и того же столбца, нужно щелкнуть на нем несколько раз на нижней панели. Затем для каждого из выбранных столбцов задается выходное имя (Output Alias), выполняемая операция (например, Group by, Count и т.п.) и любые флаги сравнения (например, Ignore case). Для разных столбцов могут задаваться разные параметры производительности в виде либо точных (Distinct Count Keys), либо приблизительных чисел (Distinct Count Scale). Эти значения подсчитывают количество различных (т.е. уникальных) преобразуемых значений. Предполагается несколько диапазонов.
  - **Low**. Приблизительно 500 тысяч значений.
  - **Medium**. Приблизительно 5 миллионов значений.
  - **High**. Примерно 25 миллионов значений.

  Аналогичные параметры могут быть заданы и для столбцов, по которым выполняется группировка (Group by). Для этого следует развернуть раздел Advanced вкладки Aggregations и ввести точное (Keys) или приблизительное (Keys Scale) число различных значений, подвергающихся обработке. В качестве альтернативы ключи производительности могут быть заданы во вкладке Advanced для всего компонента, включая объем выделяемой расширенной памяти, если такая потребуется.
- **Audit**. Добавляет в поток данных столбцы контекста выполнения, позволяя записывать информацию аудита (откуда и когда поступила информация) непосредственно в данные. Доступны следующие столбцы: PackageName, VersionID, ExecutionStartTime, MachineName, UserName, TaskName и TaskId.

- **Character Map**. Позволяет строкам потока данных быть преобразованными с помощью целого ряда операций: `Byte reversal`, `Full width`, `Half width`, `Hiragana`, `Katakana`, `Linguistic casting`, `Lowercase`, `Simplifyed Chineese` и `Uppercase`. В редакторе выберите столбец, подлежащий преобразованию, добавляя его с помощью нижней панели необходимое число раз. Далее для каждого столбца указывается пункт назначения: новый столбец (New column) или замена текущего столбца (Inplace change). После этого выберите операцию для изменяемого столбца.
- **Conditional Split**. Позволяет строкам потока данных разделяться между различными результатами в зависимости от их содержимого. Элемент конфигурируется путем ввода выражений и результирующих имен в редакторе. Когда этот элемент преобразования получает строку, каждое из выражений вычисляется по порядку, и первое из них, которое окажется истинным, получит строку данных. Если ни одно из выражений не окажется истинным, то эту строку получит результат, назначенный по умолчанию (его имя вводится в нижней части редактора). После конфигурирования при подключении потока данных открывается всплывающее окно выбора выходного потока, в котором пользователю придется произвести выбор. Не отображенные результаты игнорируются, что может привести к потере данных.
- **Copy Column**. Добавление в поток данных копии существующего столбца. В редакторе выберите копируемые столбцы, добавляя по мере необходимости несколько копий с помощью нижней панели. Каждому новому столбцу впоследствии следует присвоить соответствующее имя (Output Alias).
- **Data Conversion**. Добавление в поток данных копии столбца с преобразованием типа данных (при необходимости). В редакторе выберите преобразуемые столбцы, создавая по мере надобности несколько их копий. Каждому новому столбцу необходимо присвоить соответствующее имя и тип данных; при этом преобразования кодовых страниц не допускаются. При использовании расширенного редактора можно активизировать независимый от региональных настроек алгоритм ускоренного разбора строк. Для этого нужно установить для параметра `FastParse` значение `true`.
- **Data Mining Query**. Запускает запрос DMX для каждой строки потока данных, позволяя строкам быть ассоциированными с предсказаниями (например, определение вероятности того, что новый заказчик совершит покупку). Конфигурирование выполняется путем выбора диспетчера подключений `Analysis Services`, структуры раскрытия данных и модели раскрытия, к которой будет выполняться запрос. Во вкладке Query щелкните на кнопке Build New Query и отобразите столбцы потока данных на столбцы модели (по умолчанию отображение основывается на именах столбцов). Затем в нижней части панели выберите столбцы, добавляемые в поток, и присвойте выходному потоку подходящее имя (Alias).
- **Derived Column**. Использование выражений для генерирования значений, которые могут быть либо добавлены в поток данных, либо заменить значения в существующих столбцах. В редакторе спроектируйте выражения службы интеграции, при необходимости используя преобразования типов данных. Определите, будут ли новые значения замещать существующие столбцы или формировать новый столбец. Присвойте новому столбцу подходящее имя и тип данных.
- **Export Column**. Запись особо больших типов данных (`DT_TEXT`, `DT_NTEXT` или `DT_IMAGE`) в файл, определяемый именем, содержащемся в потоке данных. Например, большие текстовые объекты могут извлекаться в разные файлы для включения в Web-сайт или индексации. В редакторе выберите по два столбца для каждого определяемого извлечения: столбец с крупными данными и столбец, содержащий имя файла. Один

файл может получить любое количество объектов — для определения режима помещения данных в файлы определите параметр Append/Truncate/Exists.

- **Fuzzy Grouping**. Идентификация дублирующихся строк в потоке данных с использованием точного сравнения для любых типов данных и нечеткого сравнения для строковых данных типов DT_STR и DT_WSTR. Данное задание конфигурируется с указанием ключевых столбцов потока данных, на основе которых определяется уникальность. В качестве результата этого преобразования в выходной поток может быть добавлено несколько столбцов.
  - **Входной ключ** (имя по умолчанию — _key_in). Последовательный номер, предназначенный для идентификации каждой входящей строки.
  - **Выходной ключ** (имя по умолчанию — _key_out). Ключ _key_in строки, с которой данная строка совпадает, или ее собственный ключ _key_in, если совпадений не найдено. Одним из способов исключения из потока данных дублирующихся строк является связывание этого компонента с элементом Conditional Split, в котором задано условие [_key_in] == [key_out].
  - **Мера подобия** (имя по умолчанию — name_score). Мера подобия всей строки (в диапазоне от нуля до единицы) первой строке множества дубликатов.
  - **Групповой вывод** (имя по умолчанию — <*столбец*>_clean). Для каждого выбранного ключевого столбца это значение из первой строки набора дубликатов (т.е. значение из строки, на которую указывает ключ _key_out).
  - **Сходный вывод** (имя по умолчанию — Similarity_<*столбец*>). Для каждого выбранного ключевого столбца это его мера подобия по отношению к первой строке набора дубликатов.

  В редакторе выберите диспетчер подключений OLE DB, где преобразование будет иметь разрешения на создание временной таблицы. После этого сконфигурируйте все ключевые столбцы, назначая им имена обычного, группового и сходного вывода. В дополнение для каждого столбца определите следующие параметры.
  - **Match Type**. Для каждого строкового столбца выберите между точным (Exact) и нечетким (Fuzzy) сравнением. (Для столбцов других типов всегда используется точное сравнение.)
  - **Minimum Similarity**. Наименьшая мера подобия, допустимая для принятия решения о соответствии строк. Если в столбцах, в которых выполняется нечеткое сравнение, оставить значение 0 (нуль), подобие будет управляться ползунком во вкладке Advanced редактора.
  - **Numerals**. Определите, должны ли предваряющие и замыкающие нули и пробелы приниматься во внимание при сравнении. Принятое по умолчание значение Neither предполагает, что это не так.
  - **Comparison Flags**. Выберите установки, соответствующие типу сравниваемых строк.
- **Fuzzy Lookup**. Аналогично преобразованию Lookup, за исключением случая, когда четкий поиск завершается неудачей, нечеткий будет пытаться выполнить поиск во всех строковых столбцах (DT_STR и DT_WSTR). Определите диспетчер подключений OLE DB и имя таблицы, в которой будет выполняться поиск, а также новый или существующий индекс, который будет использован для кэширования информации о нечетком поиске. Во вкладке Columns определите отношения между потоком данных и таблицей, на которую сделана ссылка, а также то, какие столбцы таблицы будут до-

бавляться в поток данных. Во вкладке Advanced выберите меру подобия, по достижении которой будет приниматься решение о соответствии. Чем меньше это число, тем более либеральным будет поиск. В дополнение к заданным столбцам будут добавляться следующие метаданные.

- **_Similarity**. Отчет о подобии всех сравниваемых значений.
- **_Confidence**. Отчет об уровне правдоподобия того, что выбранное соответствие является корректным. в сравнении с остальными возможными соответствиями в таблице классификаторов.
- **_Similarity_<*имя_столбца*>**. Подобие каждого отдельного столбца.

Расширенный редактор позволяет дополнительно установить для каждого столбца параметры `MinimumSimilarity` и `FuzzuComparisonFlags`.

- **Import Column**. Чтение объектов крупных типов данных (`DT_TEXT`, `DT_NTEXT` и `DT_IMAGE`) из файла, определяемого именем, содержащимся в потоке данных, с добавлением текста или изображений в новый столбец потока данных. Этот элемент конфигурируется в расширенном редакторе. При этом во вкладке Input Columns идентифицируется столбец, содержащий имена файлов. После этого во вкладке Input and Output Properties создается новый выходной столбец для каждого столбца имен файлов, в который будут заноситься все прочитанные данные. Присвойте новым столбцам соответствующие имена и типы данных. Запомните в свойствах столбца недоступный параметр ID и найдите параметры соответствующего входного столбца с именем файлов. Установите значение параметра `FileDataColumnID` входного столбца в значение идентификатора `ID` выходного столбца, чтобы связать воедино столбцы имени файла и содержимого. Также установите для свойства `ExpectBOM` значение `true` для всех данных с типом `DT_NTEXT`, подлежащих считыванию, которые записывались с маркерами порядка байтов.
- **Lookup**. Поиск строк в таблице базы данных, совпадающих с потоком данных, и включение ее избранных столбцов в поток — нечто вроде объединения между потоком данных и таблицей. Например, идентификатор товара может быть добавлен в поток с помощью выполнения поиска в классификаторе товаров по названию. Идентифицируйте диспетчер подключений `OLE DB` и ссылку на таблицу, представление или запрос, которые будут участвовать в поиске. Во вкладке Columns установите соответствия между столбцами потока данных и таблицы классификаторов, перетаскивая линии между соответствующими элементами. После этого выберите столбцы таблицы классификаторов, которые должны быть добавлены в поток данных, изменяя при необходимости на нижней панели их имена.

  Во вкладке Advanced вы можете оптимизировать производительность памяти преобразования `Lookup`. По умолчанию вся таблица классификаторов считывается в память в начале преобразования для минимизации операций ввода-вывода, что на практике может отнимать достаточно много времени при работе с большими таблицами. Установите флажок Enable Memory Restrictions, чтобы кэшировать только часть таблицы, а затем установите размер кэша, подходящий для того количества строк, которые вы хотите загружать в одном пакете. Если для набора данных более эффективна другая форма, можно также модифицировать инструкцию SQL.
- **Merge**. Комбинирует строки двух отсортированных потоков данных в единый поток. Например, если некоторый поток строк разделен ошибочными строками или компонентом преобразования `Conditional Split`, его можно снова соединить. В обоих вышестоящих потоках должна быть выполнена сортировка по одним и тем же ключе-

вым столбцам, а типы данных сливаемых столбцов должны быть совместимы. Конфигурация выполняется с помощью перетаскивания двух разных входов к элементу слияния и отображения столбцов в редакторе. Комбинирование неотсортированных потоков данных выполняется элементом `Union All`.

- **Merge Join**. Реализует функциональность оператора `JOIN` языка SQL для потоков данных, отсортированных по столбцам объединения. Конфигурирование выполняется перетаскиванием входов к элементу, однако при этом следует уделять внимание тому, какой из входов находится слева, а какой справа (этот будет существенно для внешних объединений). Выбор типа объединения выполняется в редакторе. Там же следует установить отображения объединяемых столбцов и выбрать столбцы. которые будут направлены на выход.
- **Multicast**. Копирование всех строк входного потока данных во множество выходных. Как только один из выходов подключается к некоторому компоненту, тут же появляется следующий выход, доступный для подключения. Конфигурируются только имена выходов.
- **OLE DB Command**. Выполнение инструкции SQL (типа `UPDATE` или `DELETE`) для всех строк потока данных. Конфигурирование выполняется определением диспетчера подключения OLE DB. Далее следует перейти к вкладке Component Properties и ввести инструкцию SQL, используя символы вопросительных знаков для параметров (например, `UPDATE MyTable SET Col1=? WHERE Col2=?`). Во вкладке Column Mappings ассоциируйте столбцы потока данных с параметрами инструкции SQL.
- **Percentage Sampling**. Разделение строк потока данных случайным образом на основе введенного процентного соотношения плотности выходных потоков. Например, этот элемент может использоваться для разделения потока на множества для обучения и тестирования элемента раскрытия данных. В редакторе укажите приблизительный процент для выхода `Selected` — остальные строки будут направлены на выход `Unselected`. Если на вход подается один и тот же поток данных, то данное преобразование отберет один и тот же набор строк.
- **Pivot**. Денормализация потока данных, аналогично тому, как работают перекрестные (сводные) таблицы программы Excel, перенося значения атрибутов в столбцы. Например, поток данных с тремя столбцами, `Quarter` (квартал), `Region` (регион) и `Revenue` (доход), можно развернуть так, что столбцами окажутся `Quarter`, `Western Region` (западный регион) и `Eastern Region` (восточный регион), т.е. поток будет развернут по регионам.
- **Row Count**. Подсчет количества строк в потоке данных и помещение результата в переменную. Конфигурируется путем заполнения поля имени переменной (`VariableName`).
- **Row Sampling**. Этот элемент практически идентичен преобразованию `Percentage Sampling`, за исключение того, что вводится количество строк, которые должны попасть в выборку, а не их процент.
- **Script**. Перетащите компонент сценария в рабочую область конструктора, выбрав в открывшемся диалоговогом окне тип компонента Transformation. В редакторе выделите те столбцы, которые будут доступны в сценарии, при этом разделяя их на доступные только для чтения (`ReadOnly`) и доступные для чтения и записи (`ReadWrite`). Во вкладке Inputs and Outputs добавьте столбцы выхода, которые будут заполняться сценарием, выше и ниже входных столбцов.

  На странице редактора Script укажите переменные, доступные для чтения, а также для чтения и записи в сценарии, разделив их запятыми в параметрах `ReadOnlyVariables`

и ReadWriteVariables соответственно. Оставьте для параметра PreCompile значение true, как предложено по умолчанию, если, естественно, вас не тревожит размер пакета на диске. Щелкните на кнопке Script, чтобы раскрыть сам программный код. Заметим, что первичный метод замещает метод *<имя_входа>*_ProcessInputRow, как показано в следующем примере:

```
Public Class ScriptMain
    Inherits UserComponent
    Public Overrides Sub Input0_ProcessInputRow(ByVal Row As
Input0Buffer)
  'Система источника отмечает отсутствие дат, вставляя старые даты
  If Row.TransactionDate < #1/1/2000# Then
     Row.TransactionDate_IsNull = True
     Row.PrimeTimeFlag_IsNull = True
  Else
     'Установка флага выполнения транзакции в рабочее время
     If Weekday(Row.TransactionDate) > 0 _
     And Weekday(Row.TransactionDate) < 6 _
     And Row.TransactionDate.Hour > 7 _
     And Row.TransactionDate.Hour < 19 Then
        Row.PrimeTimeFlag = True
     Else
        Row.PrimeTimeFlag = False
     End If
  End If
 End Sub
End Class
```

В этом примере используется один вход, доступный для чтения и записи (TransactionDate), и один выход (PrimeTimeFlag), а имя входа оставлено, как принято по умолчанию (Input 0). Каждый столбец представляется как свойство объекта Row и дополняется суффиксом _IsNull для тестирования и установки пустого значения. Эта процедура вызывается для каждой строки потока данных.

- **Slowly Changing Dimension**. Сравнение данных в потоке с таблицей измерений и формирование измерения на основе ролей, назначенных конкретным столбцам. Этот компонент необычен тем, что не имеет редактора. Вместо него запускается мастер, позволяющий последовательно определить роли столбцов и взаимодействие с таблицей измерений. В результате выполнения мастера множество компонентов будет помещено в рабочую область конструктора для выполнения задачи обслуживания измерения.
- **Sort**. Сортировка строк потока данных по выбранным столбцам. Конфигурация осуществляется выбором соответствующих строк. После этого на нижней панели выберите тип сортировки, ее порядок и установите флажки, соответствующие сортируемым данным.
- **Term Extraction**. Создание нового потока данных, основанного на терминах, найденных в текстовом столбце с кодировкой Unicode (с типом данных DT_WSTR или DT_NTEXT). Такой поток применяется при раскрытии данных, когда строки конкретного типа используются для генерации списка часто используемых терминов. Впоследствии эти термины могут использоваться в компоненте Term Lookup для идентификации сходных строк. Например, текст сохраненного документа RSS может быть использован для поиска сходных документов в большом наборе. Для конфигурирования следует выбрать столбец, содержащий текст с кодировкой Unicode. Если создан список исключаемых терминов, идентифицируйте соответствующие таблицу и столбец во вкладке Exclusions. Вкладка Advanced управляет алгоритмом извлечения; среди прочего

определяется, должно ли слово быть одиночным или находиться в составе фразы (артикли, местоимения и прочие служебные части речи никогда не включаются), алгоритм подсчета, минимальная частота, необходимая для извлечения, и максимальная длина фразы.

- **Term Lookup**. Выполнение своеобразного объединения столбца потока данных с текстом Unicode с таблицей терминов, созданной компонентом `Term Extraction`. Для каждого совпадающего термина создается строка в выходном потоке данных. Выходной поток также содержит два дополнительных столбца: `Term` и `Frequency`. В первом из них содержится шаблон из таблицы сопоставления, а во втором — количество вхождений данного шаблона в строку столбца потока данных. Это преобразование конфигурируется определением диспетчера подключений `OLE DB` и таблицы, содержащей список терминов. Во вкладке Term Lookup выбираются входные столбцы, проходящие через компонент для формирования выходного потока. После этого на верхней панели определяется соответствие между столбцом входного потока и столбцом `Term` таблицы терминов.
- **Union All**. Этот компонент комбинирует несколько входных потоков данных в единый выходной; при этом требуется совместимость типов соответствующих столбцов. Конфигурирование осуществляется с помощью подключения всех входных потоков, участвующих в формировании выходного. После этого выполняется отображение столбцов каждого из входящих потоков на столбцы выходного.
- **Unpivot**. Нормализация потока данных с помощью преобразования столбцов в значения атрибутов. Например, поток данных с одной строкой для каждого квартала и столбцами объема доходов по каждому из регионов может быть преобразован в выходной поток с тремя столбцами: номера квартала, региона и объема доходов.

Обслуживаемые и управляемые пакеты

Служба интеграции позволяет создавать приложения с относительно небольшими усилиями. Это довольно весомое преимущество с точки зрения процесса разработки решений, но оно может стать источником проблем при отсутствии надлежащего планирования. При написании работоспособных и управляемых приложений будьте внимательны, независимо от реализации. К счастью, служба интеграции оснащена множеством средств, поддерживающих долгосрочную работоспособность и управляемость.

Перед этапом разработки очень важным моментом является проектирование, особенно если работа над приложением службы интеграции осуществляется в первую очередь, так как практические результаты часто используются на последующих этапах, в частности протоколирование, аудит и общая структура данных. Возможно, ключевым преимуществом разработки приложений с использованием службы интеграции является возможность централизации всей обработки данных в одном месте, с четким разграничением последовательности действий и возможностью обработки возникающих ошибок. Централизация существенно повышает работоспособность приложения по сравнению с традиционным подходом "сценарий — здесь, программа — там, а хранимая процедура — вообще неизвестно где". Во время проектирования не забудьте уделить внимание и ряду других вопросов.

- Идентифицируйте повторяющиеся фрагменты для их последующего многократного использования. Многие повторяющиеся задания выполняют над объектами одни и те же действия и используют одни и те же метаданные — именно они являются первыми кандидатами на помещение в отдельные субпакеты.

- Ключом к успеху в процессе эксплуатации является подходящая стратегия протоколирования. Спросите себя, кто будет отвечать за устранение возникшей ошибки и как он о ней узнает. Например, как ответственный узнает, что некоторый пакет должен был быть запущен, но этого по некоторой причине не произошло? Какой уровень протоколирования достаточен? (Учтите, что "больше" не всегда "лучше". Слишком много ненужных деталей способно только замаскировать истинную проблему.) Какой тип информации о состоянии пакета и среды необходим для выявления причины возникновения ошибки?
- Концепции аудита могут оказаться полезными для операций согласования и восстановления после ошибок. Какой тип данных можно ассоциировать с данными, создаваемыми в пакете? Если необходим большой объем информации, подумайте о создании многоуровневого протокола, помещая в обработанные записи только идентификаторы.
- Какие детали пакетов будут изменяться при запуске пакетов на разных платформах? Какой режим хранения (реестр, XML, SQL и т.п.) будет более эффективным при распространении данных конфигурации? Если возможно, используйте конфигурирование на уровне системы, а не на уровне пакета — это упростит распространение и сопровождение настроек.
- Определите порядок восстановления после сбоев пакета. Спросите себя, потребуется ли вмешательство вручную для того, чтобы снова запустить пакет. Например, пакет, загружающий данные, может потребовать использования транзакций, чтобы после восстановления после сбоя не образовались дублирующиеся строки.
- Для "долгоиграющих" пакетов продумайте логику расстановки контрольных точек, с которых можно заново запускать пакет.
- Определите наиболее вероятные точки возникновения ошибок в пакете. Подумайте, какие действия придется предпринять для исправления ошибки (реально, а не теоретически). Если возможно, внесите эти действия в пакет, используя потоки ошибочных данных и ограничения, чтобы не нагружать лишней работой обслуживающий персонал.

Правильный стиль программирования также способен повысить обслуживаемость пакета. Давайте пакетам, заданиям, компонентам и прочим видимым объектам осмысленные имена. Свободно используйте примечания, чтобы обслуживающий персонал смог понять неочевидные решения и причины их использования. Это также поможет и разработчикам, которые в будущем будут развивать пакет. Обязательно используйте программы управления версиями, чтобы поддерживать историю пакета и связанные версии файлов.

Протоколирование

Поскольку множество пакетов предназначено для несопровождаемых операций, генерация журнала выполнения является отличным методом отслеживания операций и накопления отладочной информации. Щелкните правой кнопкой мыши в рабочей области конструктора пакетов и выберите в контекстном меню пункт Logging. Во вкладке Providers and Logs добавьте поставщика для каждого типа вывода, который будет протоколироваться (допустимо множество поставщиков). Во вкладке Details определите события, для которых будут создаваться записи в журнале. Расширенное представление (рис. 42.6) также позволяет выбрать столбцы, включаемые в каждую запись журнала.

Древовидное представление на левой панели является контейнером иерархии пакета. Флажки отвечают за параметр `LoggingMode` каждого объекта: пустой флажок соответствует режиму отключения, отмеченный — включенному режиму, а серый позволяет наследовать настройки протоколирования от родительского объекта. Если выделить какой-либо элемент в

дереве, то отобразятся его детали. Следует отметить, что поставщики могут быть сконфигурированы только для пакета в целом, а параметры всех объектов с установленным параметром `UseParentSettings` будут недоступны в отличие от параметров родительских объектов.

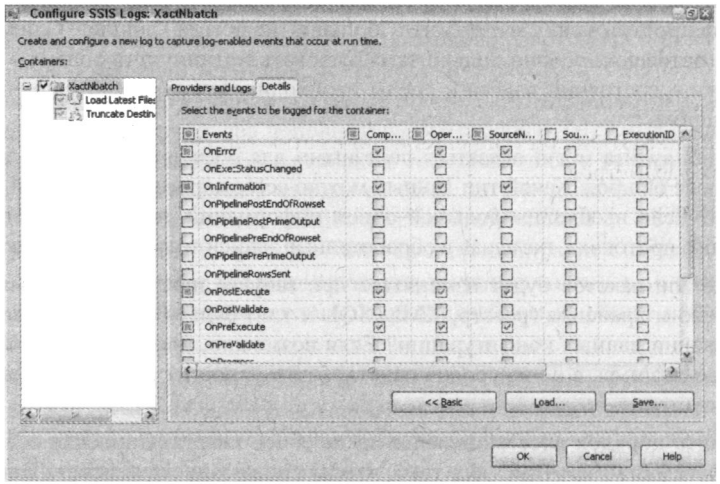


Рис. 42.6. Расширенное представление вкладки Advanced

Стандартные поставщики журнала описаны ниже.

- **Text File**. Запись в текстовый файл с разделителями. Конфигурирование выполняется в соответствующем диспетчере подключений.
- **SQL Profiler**. Запись в файл `.TRC`, который можно просмотреть в утилите SQL Profiler. Этот метод может пригодиться, когда требуется наряду с журналом пакета просмотреть другую информацию о производительности сервера. Конфигурирование выполняется в соответствующем файловом диспетчере подключений.
- **SQL Server**. Запись в таблицу `dbo.sysdtslog90` базы данных, обозначенной в соответствующем диспетчере подключений OLE DB. Для обслуживания таблицы может быть выбрана любая база данных. Если схема таблицы не существует, то она будет создана при первом использовании.
- **Event Log**. Запись в журнал событий Windows того компьютера, на котором выполняется пакет. Конфигурирование в данном случае не требуется.
- **XML File**. Запись в файл XML. Конфигурирование выполняется в соответствующем диспетчере подключений.

Вкладка Details окна конфигурации протоколирования также имеет расширенный режим. позволяющий выбрать регистрируемые столбцы для каждого из событий. Когда необходимая конфигурация комбинаций "событие–столбцы" будет выстроена, ее можно сохранить в качестве шаблона и использовать в других пакетах.

Конфигурация пакета

Конфигурации пакетов облегчают перемещение пакетов между серверами и платформами. Это позволяет установить свойства пакетов, основанные на специфичных для каждой среды конфигурациях. Например, имена серверов и входные каталоги могут изменяться при перемещении пакета из среды разработки в среду эксплуатации.

Щелкните правой кнопкой мыши в рабочей области конструктора пакетов и выберите в контекстном меню пункт Package Configurations. В открывшемся окне отобразится список конфигураций, применяемый к пакету в порядке перечисления. Чтобы добавить новую конфигурацию, убедитесь, что конфигурации активизированы, и щелкните на кнопке Add. Когда запустится мастер конфигурации пакетов (Package Configuration Wizard), выберите необходимый тип конфигурации (размещение хранилища). В сущности, следует рассматривать три категории.

- **Registry & Environment Variable**. Эти типы позволяют хранить только один параметр.
- **XML File & SQL Server Table**. Каждый из этих типов позволяет хранить любое число настроек параметров.
- **Parent Package Variable**. Осуществляет доступ из вызываемого пакета к содержимому одной переменной.

Большинство типов конфигураций позволяет идентифицировать место хранения либо непосредственно, либо с помощью переменной среды. Переменные среды могут оказаться полезными, когда в разных средах место хранения (например, файловый каталог) отличается. После определения типа конфигурации и места хранения параметр Select Properties to Export позволяет выбрать те параметры, которые будут изменяться в разных средах. В заключение просмотрите выбранные настройки, присвойте имя конфигурации и завершите работу мастера.

Конфигурации могут многократно использоваться в разных пакетах, если имена объектов, содержащих настраиваемые параметры, идентичны. В частности, пакеты, использующие одинаковые имена своих диспетчеров подключений, могут совместно использовать конфигурацию, определяющую сервер и имена файлов. Чтобы использовать готовую конфигурацию в дополнительных пакетах, выберите ее тип, а затем определите то же место хранения (например, файл XML), что и в исходном пакете. Когда откроется диалоговое окно, предупреждающее, что такая конфигурация уже существует, выберите опцию Reuse Existing.

Перезапуск из контрольной точки

Разрешение перезапуска пакета из контрольных точек позволяет не повторять выполнение заданий, завершившихся успешно. Учтите приведенные ниже правила использования точек перезапуска.

- Только задания потока управления определяют точки перезапуска; все задания потока данных видны как единый рабочий элемент, независимо от того, сколько компонентов они содержат.
- При сбое любая транзакция, находящаяся в процессе выполнения, откатывается, так что точка перезапуска должна находиться перед этой транзакцией. Таким образом, если весь пакет выполняется как единая транзакция, перезапуск должен всегда осуществляться с первой операции пакета.
- Любые контейнеры цикла следует запускать с самого начала.
- Конфигурация, используемая при перезапуске, сохраняется в файле контрольной точки, а не в текущем файле конфигурации.

Включение контрольных точек выполняется с помощью установки следующих параметров пакета.

- **CheckpointFilename**. Имя файла, в котором сохраняется информация о контрольной точке.
- **CheckpointUsage**. Значение `IfExists` указывает начинать выполнение с контрольной точки, если такой файл существует, или с первой операции пакета в противном случае.

Значение Always указывает на завершение выполнения пакета, если файл контрольной точки не существует.

- **SaveCheckPoints**. Для параметра сохранения контрольных точек следует установить значение True.

К тому же для параметра FailPackageOnFailure должно быть установлено значение True в пакете и во всех заданиях и контейнерах, выступающих в роли точек перезапуска.

Развертывание пакетов

Утилита Business Intelligence Development Studio является идеальной средой разработки и отладки пакетов службы интеграции, однако это не самое эффективное место выполнения пакетов. Не устанавливая пакет на сервер, его можно выполнить без дополнительной нагрузки среды разработки с помощью утилиты dtexec или dtexecui. Запустите утилиту dtexecui из командной строки и установите необходимые параметры выполнения, после чего либо щелкните на кнопке Execute, либо перейдите к странице Command Line, чтобы скопировать соответствующие переключатели командной строки утилиты dtexec.

Установка пакета на сервере назначения целесообразна тогда, когда пакет будет использоваться многократно. После установки пакет становится известен службе интеграции, которая может быть подключена к SQL Server Management Studio для мониторинга. Служба интеграции также кэширует компоненты, выполняемые в пакетах, для сокращения времени их загрузки.

Установка пакетов

Можно создать специальную утилиту развертывания, устанавливающую все пакеты проекта на сервере назначения. Эта утилита конфигурируется щелчком правой кнопкой мыши на проекте (а не на пакете) в Solution Explorer и выбором в контекстном меню пункта Properties. Перейдите на страницу Deployment Utility открывшегося диалогового окна и установите для параметра CreateDeploymentUtility значение True. Просмотрите настройки параметра DeploymentOutputPath, чтобы определить, где будет сохранен пакет установки, по отношению к каталогу проекта. Установка параметра AllowConfigurationChanges позволяет в процессе инсталляции изменять параметры конфигурации. Сохраните настройки, а затем выберите в меню пункт Build⇨Build *имя_проекта*. Пакет развертывания будет создан.

После создания утилиты развертывания подключитесь к серверу назначения и дважды щелкните на манифесте пакета (*имя_проекта*.SSISDeploymentManifest), чтобы установить пакеты на целевом сервере.

С помощью утилиты Management Studio можно установить на сервере и отдельные пакеты проекта. Подключитесь к целевому серверу, а затем в окне Object Explorer — к локальному экземпляру службы интеграции. В узле File System или MSDB щелкните правой кнопкой мыши и выберите в контекстном меню пункт Import Package. Отмечу, чтобы исходный пакет был сохранен либо в файле, либо в базе данных msdb SQL Server. Аналогичная функциональность доступна и в утилите командной строки dtutil.

Выполнение пакетов

После установки на сервере назначения пакет может быть запущен на выполнение несколькими способами.

- Найдите установленный пакет в SQL Server Management Studio, щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Run Package. В результате для выбранного пакета будет запущена утилита `dtexecui`.
- Запустите утилиту `dtexecui`, которая позволит настроить целый ряд параметров выполнения пакета. Также она сформирует соответствующие переключатели командной строки для утилиты `dtexec`.
- В определении задания службы SQL Server Agent создайте действие с типом SQL Server Integration Services Package и источником SSIS Package Store.

Выбор метода выполнения и места размещения пакета оказывает значительное влияние на производительность и отчетность. При использовании утилит `dtexecui/dtexec` можно снять нагрузку с текущего сервера баз данных, если перенести обработку на другой сервер или рабочую станцию. Эти варианты также по умолчанию предполагают словесную обратную связь во время выполнения, что очень полезно для отслеживания хода выполнения и осмысления ошибок. В то же время этот сценарий предполагает и значительные сетевые потоки. Например, загрузка последовательности файлов данных с файл-сервера на SQL Server через посредство рабочей станции удвоит нагрузку на сеть (так как файл должен вначале быть перемещен с файл-сервера на рабочую станцию и уже затем с рабочей станции на SQL Server) по сравнению с запуском пакета непосредственно на сервере баз данных.

Запуск пакета с помощью службы SQL Server Agent приведет к его выполнению непосредственно на сервере баз данных. Это минимизирует нагрузку на сеть, однако может вызвать проблемы, если SQL Server не имеет ресурсов, адекватных запросам пакета (в частности оперативной памяти). К тому же средства отчетности SQL Server Agent ограничены протоколированием, установленным в самом пакете. В данном случае протоколирование в пакете нужно настраивать со всей тщательностью, чтобы на выходе получить адекватную информацию для управления и отладки.

Изменения в службе интеграции, связанные с выходом пакетов обновлений

Выход пакета обновлений SP1 внес в службу интеграции следующие коррективы.

- Задание `Web Service` теперь поддерживает параметры.
- Мастер импорта-экспорта теперь обрабатывает многотабличные сценарии.
- Интерфейс `IDtsPipelineEnvironmentService` теперь позволяет компонентам пользовательских потоков данных получить программный доступ к родительскому потоку данных.
- Улучшено взаимодействие со службой анализа. Источник `DataReader` в службе интеграции поддерживает тип данных `System.Object`, преобразуя столбцы, имеющие его, в тип `DT_NTEXT` службы интеграции, который можно преобразовать далее в более подходящий.
- Увеличена производительность многих преобразований, в том числе сортировки. Повышено удобство работы конструктора. К примеру, можно щелкнуть правой кнопкой мыши на рабочей области вкладки Data Flow и выбрать в контекстном меню пункт Execute Task. Это позволит выполнить только задание `Data Flow`. Для этого уже не нужно переключаться на вкладку Control Flow.

- Добавлена основанная на реестре политика обработки подписей пакетов службы интеграции во время выполнения пакета. К примеру, администратор может запретить загрузку неподписанных и сомнительных пакетов. Так как эти настройки содержатся в реестре, администратор может распространить их в пределах домена, используя настройки политики безопасности системы Windows.
- В диалоговом окне Advanced Editor источник `Flat File` теперь имеет новое свойство `UseBinaryFormat`, которое поддерживает загрузку упакованных десятичных данных в канал для обработки сценарием или пользовательского преобразования.
- В диалоговом окне Advanced Editor источник `DataReader` теперь имеет новое свойство `CommandTimeout`, которое можно использовать для изменения времени ожидания долго выполняющихся операций.
- Для создания или изменения выражений свойств переменных теперь можно открыть диалоговое окно Expression Buider из окна Properties.
- Отныне можно прикреплять примечания к ограничениям приоритета.

Выход пакета обновлений SP2 внес в службу интеграции следующие коррективы.

- Отныне можно разрешить проблемы взаимодействия пакетов с внешними источниками данных, включая регистрацию и выбирая событие `Diagnostic` пакета. Многие ошибки пакетов возникают в процессе взаимодействия с поставщиками внешних данных. Однако эти поставщики не возвращают в службу интеграции сообщения, достаточные для решения проблемы. Компоненты службы интеграции, перечисленные ниже, предназначены для записи сообщений в журнал до и после каждого вызова поставщика внешних данных. Это сообщение содержит имя вызванного метода; например, метод `Connect` объекта `Connection` поставщика OLE DB или метод `ExecuteNonQuery` объекта `Command`. Эти сообщения можно просмотреть, включив протоколирование и выбрав событие `Diagnostic` пакета.
- Служба интеграции и мастер импорта и экспорта SQL Server теперь поддерживают поставщика Microsoft Office 12.0 Access Database Engine OLE DB Provider для подключения к источникам данных Microsoft Office Access 2007 и Excel 2007.

  Для подключения к источникам данных Access 2007 и Excel 2007 нельзя использовать поставщик Microsoft Jet OLE DB Provider. Для подключения к источнику данных Excel 2007 следует использовать диспетчер подключений OLE DB, а также источник и приемник OLE DB. Для подключения к источнику данных Excel 2003 и более ранних версий продолжайте использовать диспетчер подключений Excel, а также источник и приемник Excel.
- Значением по умолчанию свойства `BypassPrepare` задания `Execute SQL` теперь является `True`. В более ранних версиях значением по умолчанию было `False`, и это означало, что инструкции были всегда подготовлены. В SP2 запросы по умолчанию не считаются подготовленными. Это позволяет избежать многочисленных ошибок при работе с некоторыми поставщиками, когда вы пытаетесь подготовить инструкцию, использующую маркер параметра (`?`).
- Для выбора переменных для свойств некоторых компонентов потока данных можно использовать комбинированные списки.

  В более ранних версиях имена переменных приходилось вводить в текстовые поля. (Например, имена переменных требуют наличия свойства `OpenRowsetVariable` источника OLE DB.) Комбинированный список содержит все доступные переменные, в том числе и системные.

- Преобразование Lookup выводит в отчет количество кэшированных строк.

  В режиме полного кэширования преобразование Lookup использует функцию, активируемую таймером, для отчета о количестве кэшированных строк. В предыдущих версиях преобразование не всегда выводило этот отчет. В SP2 преобразование Lookup оснащено новым информационным сообщением, специально предназначенным для этой цели. Это сообщение отображается в окне Progress и протоколируется событием OnInformation.

- Задание ExecuteSQL теперь имеет свойство ParameterSize для строковых параметров.

  В предыдущих версиях задание Execute SQL завершалось ошибкой при использовании подключения ADO.NET для выполнения хранимых процедур, возвращающих строковый параметр. Причиной этому было то, что по умолчанию размер строкового параметра был установлен в нуль. В SP2 параметры, используемые в задании ExecuteSQL, оснащены свойством ParameterSize, которое используется для выделения памяти под строковый выходной параметр. Компонент Script выдает ошибку, когда свойство *<имя_столбца>*_IsNull установлено в значение True.

- Свойство IsNull столбцов компонента Script выводит предупреждение при своем некорректном использовании.

- Программный код, генерируемый компонентом Script, для каждого столбца входа и выхода имеет свойства *<имя_столбца>* и *<имя_столбца>*_IsNull. Назначение последнего — разрешить пользователю вводить в столбец пустые значения с помощью свойства *<имя_столбца>*_IsNull, присваивая ему значение True. Для ввода непустого значения пользователь должен присвоить его свойству *<имя_столбца>*. В предыдущих версиях, если значение свойства *<имя_столбца>*_IsNull устанавливалось в False, оно неявно заменялось значением True. В SP2 компонент Script выдает предупреждение о некорректном использовании.

Резюме

Служба интеграции является прекрасной средой разработки приложений, эффективно перемещающих большие массивы данных и предусматривающих их многошаговую трансформацию с надежной обработкой ошибок. Благодаря таким свойствам, как простота установки, функции аудита и протоколирования, а также возможность создания специфичных для разной среды конфигураций, приложения службы интеграции без проблем можно эксплуатировать сразу после создания.

Многие компании тратят массу времени и усилий на разработку приложений T-SQL с интеграцией других языков программирования там, где аналогичные задачи с легкостью может решить служба интеграции. Те, кто приложат немного усилий для освоения службы интеграции, получат многократную отдачу от упрощения процесса разработки и повышения производительности конечного продукта.

Лучше всего начинать с небольшого, простого проекта, а затем постепенно осваивать возможности службы. Как и в любой среде программирования, первые ваши приложения не будут совершенными, но в скором времени вы удивитесь, как раньше могли обходиться без службы интеграции.

ГЛАВА

43

Бизнес-логика в службе анализа

В этой главе...

Обзор хранилищ данных

Создание базы данных службы анализа

Конфигурирование измерений

Тонкая настройка измерений

Конфигурирование упреждающего кэширования

Конфигурирование поддержки целостности данных

Проработав долгое время с различными компаниями и их системами данных, со временем я начал замечать явный прогресс в их решениях анализа и отчетности. В первое время запросы выполнялись непосредственно к базам данных оперативной обработки транзакций (OLTP), однако этот подход конфликтовал с повседневным использованием баз и обычно в значительной мере ограничивал доступ ввиду ограничений безопасности.

Часто следующим этапом было ежедневное создание копии базы данных OLTP, предназначенной исключительно для обслуживания аналитических запросов. Такие попытки использования баз данных OLTP для аналитической обработки данных (OLAP) были проблематичными сразу по нескольким причинам.

- Структуры данных OLTP оптимизированы для разовых, атомарных транзакций, в то время как системы OLAP оптимизированы для работы с крупными массивами данных. Таким образом, выполнение запросов было мучительно долгим.
- Базы OLTP обычно хранят текущие данные, в то время как базы OLTP интересны с точки зрения выявления исторических тенденций.
- Структуры данных OLTP понятны ограниченному кругу экспертов в компании, в то время как базы OLAP наиболее эффективны для представления информации широкой аудитории.

Наиболее распространенным подходом к созданию баз данных OLTP, доступных для запросов анализа, было внедрение таблиц сводных данных. При надлежащем проектировании такой подход был способен решить ряд проблем быстродействия и наличия исторических данных. В то же время он все равно оставался понятным достаточно узкому кругу лиц. Целостность интерпретации также представляла проблему, так как сводные таблицы создавались в разное время и для разных целей.

Эти две концепции — целостной интерпретации и доступности для широкой аудитории — и являются ключевыми для успешной реализации систем оперативного анализа данных в частности и бизнес-аналитики в целом. Для принятия правильных производственных решений необходимо понимание общей информационной картины. Единственными альтернативами этому являются чужой опыт и интуиция.

В этой главе будет описано, как использовать службу анализа в приложениях бизнес-аналитики, сопровождающих процесс принятия решений в компании.

Хранилища данных

Служба анализа в SQL Server 2005 расширила функциональность так, что некоторые сценарии теперь не требуют наличия хранилища данных для поддерживаемых ими операций. Тем не менее при работе со службой анализа очень важно понимать основные концепции хранения данных.

Схема "звезда"

Хранилища данных — это стандартный подход к структурированному хранению реляционных данных OLAP. В его основе лежат понятия измерений и мер. Измерение — это разделение на категории, или группировка данных, а мера — консолидируемое значение. Например, в задаче консолидации объема продаж по кварталам и подразделениям мерой является объем продаж, а измерениями — квартал и подразделение.

Решение относительно того, какие измерения и меры использовать в хранилище данных, должно основываться на производственной необходимости, сводя воедино все типы вопросов, ответ на которые потребуется, и семантику данных, которые будут содержаться в хранилище. Опросы конечных пользователей и анализ существующих отчетов и метрик поможет создать первое приближение понимания задачи. Однако в большинстве случаев для полноценной идентификации потребностей необходим так называемый пилотный проект.

После этого производственные потребности обеспечат базис для построения схемы "звезда" или "снежинка", являющейся каркасом хранилища (рис. 43.1).

Свое название схема "звезда" получила благодаря своей форме — центральная таблица фактов и множество таблиц измерений окружают ее, как лучи звезды. Каждое из измерений связано с таблицей фактов отношением внешнего ключа.

Таблица фактов состоит из двух типов столбцов: ключей, связанных с остальными измерениями схемы "звезда", и столбцов интересующих мер.

Каждая таблица измерений состоит из первичного ключа, по которому строится отношение с таблицей фактов, и одного или нескольких атрибутов, разбивающих данные по категориям данного измерения. Например, измерение заказчика может содержать атрибуты имени, электронного адреса и индекса. В общем, измерение представляет денормализацию данных в системе OLTP. Например, измерение заказчика в AdventureWorksDW управляется, среди прочего, из таблиц `Sales.Individual` и `Person.Contact` базы данных AdventureWorks, а также полей. выделенных из столбца XML, описывающего демографические данные.

Иногда имеет смысл ограничить денормализацию, организуя зависимость одной таблицы измерений от другой, таким образом, изменяя схему "звезда" на схему "снежинка". В качестве примера на рис. 43.2 показано, как измерение товаров базы данных AdventureWorks было организовано в схему "снежинка". Информация категорий товаров и подкатегорий могла быть включена непосредственно в таблицу `DimProduct`, но вместо этого для создания системы категорий было создано несколько таблиц. Схема "снежинка" полезна для сложных измерений, в которых в противном случае возникают проблемы целостности, такие как назначение подкатегорий категориям (см. рис. 43.2), а также для больших измерений, где поднимается вопрос объема.

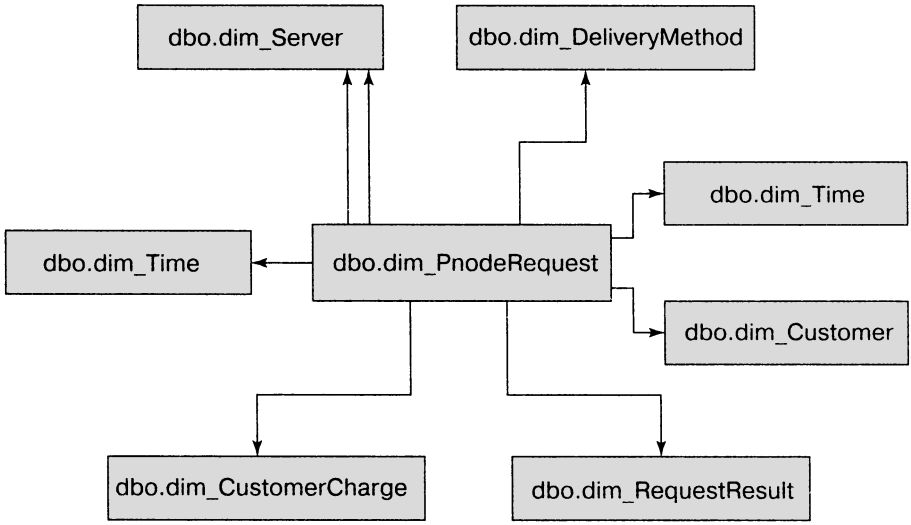


Рис. 43.1. Простая схема "звезда"

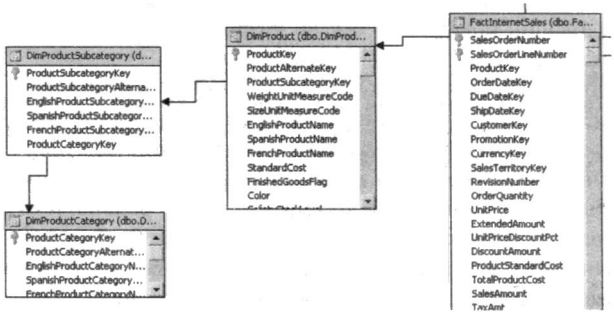


Рис. 43.2. Измерение со схемой "снежинка"



Обычно схема "снежинка" непривлекательна, так как вносит сложность и замедляет операции SQL, однако служба анализа устраняет большинство этих проблем. Если измерение можно сделать более цельным, используя схему "снежинка", поступайте именно так, за исключением следующих случаев:

- когда процедура публикации данных в схему "звезда" слишком сложная или когда замедление чувствительно;
- когда проектируемая схема будет использоваться в расширенных запросах SQL, которые усложнятся или замедлятся при использовании схемы "снежинка".

Единообразие

Определение схемы "звезда" позволяет быстро выполнять запросы OLAP к хранилищу данных, однако повышенные требования к единообразию делают необходимым соблюдение следующих правил.

- При загрузке данных в хранилище некорректные и пустые значения должны замещаться своими печатными эквивалентами. Это позволит единоразово исследовать

семантику данных и затем сделать ее доступной для широкой аудитории, реализуя единообразную интерпретацию в пределах организации. Зачастую это подразумевает добавление вручную строк в таблицы измерений. чтобы они охватывали все возможные случаи (например, `Неизвестно`, `Недоступно` и т.п.).

- Строки в таблице фактов никогда не должны удаляться. Также недопустимы любые другие операции, которые могут привести к разным результатам запросов, выполненных в разное время. Часто это подразумевает отложенный импорт выполняющихся транзакций.
- Избегайте использования ключей OLTP для связывания таблиц фактов и измерений, несмотря на то, что это часто достаточно удобно. Подумайте о схеме "звезда", содержащей финансовые данные, когда обновляется пакет бухучета с изменением при этом всех идентификаторов заказчиков. Если отношения в хранилище основать на столбце идентичности, то любые изменения в базе OLTP могут быть отражены небольшими изменениями данных в измерении заказчиков. Если же будет использован ключ OLTP, то придется преобразовать всю таблицу фактов.

Внимательные читатели, наверное, заметили, что все эти правила единообразия связаны с размерами таблицы фактов. В общем случае таблицы фактов увеличиваются только при вставке в операционные таблицы новых строк — операции удаления и обновления не оказывают влияния на их размеры. На самом деле при оценке размеров базы данных можно игнорировать размер таблиц измерений и учитывать только размер таблиц фактов.



Поддержка в хранилище больших объемов исторических данных не подразумевает хранение данных в одном и том же месте. С начала проектирования планируйте архивирование данных с помощью разделения таблиц фактов, дополняющего стратегию разделения, используемую в службе анализа. Например, большая таблица фактов может быть разбита на ежемесячные разделы, поддерживая двухлетний исторический период.

Загрузка данных

При использовании схемы "звезда" или "снежинка" добавление новых данных начинается в точках и перемещается внутрь, добавляя строки в таблицу фактов в последнюю очередь, чтобы удовлетворить ограничениям внешних ключей. Чаще всего для заполнения таблиц хранилища данных используется служба интеграции, однако в примерах, приводимых в настоящей главе, в качестве иллюстрации приводятся инструкции вставки языка SQL.

Загрузка измерений

Подход к загрузке измерений варьируется от природы данных источника. В самом благоприятном случае, когда фактические данные источника зависят от внешнего ключа к таблице, содержащей данные измерений, сканировать нужно только данные измерений. В следующем примере используется естественный первичный ключ в источнике данных — код товара — для идентификации строк в таблице товаров `Product`, которые еще не были добавлены в таблицу измерений:

```
INSERT INTO Warehouse.dbo.dimProduct (ProductCode, ProductName)
  SELECT stage.Code, stage.Name FROM Staging.dbo.Products stage
    LEFT OUTER JOIN Warehouse.dbo.dimProduct dim
       ON stage.Code=dim.ProductCode
  WHERE dim.ProductCode is NULL
```

Часто данные измерений источника не связаны внешним ключом с таблицей фактов источника. В этом случае загрузка таблицы измерений требует полного сканирования таблицы файлов с целью обеспечения согласованности. В следующем примере выполняется сканирование фактических данных, извлечение соответствующих описаний из таблицы измерений, когда таковые доступны, или использование описания "Неизвестно" в противном случае:

```
INSERT INTO Warehouse.dbo.dimOrderStatus (OrderStatusID, OrderStatusDesc)
  SELECT DISTINCT o.status, ISNULL(mos.Description,'Неизвестно')
    FROM Staging.dbo.Orders o
      LEFT OUTER JOIN Warehouse.dbo.dimOrderStatus os
        ON o.status=os.OrderStatusID
      LEFT OUTER JOIN Staging.dbo.map_order_status mos
        ON o.status = mos.Number
    WHERE os.OrderStatusID is NULL
```

Последний вариант предполагает наличие одной таблицы, которая содержит фактические данные и измерения. Именно такой случай открывает двери для несогласованности отношений между атрибутами измерений. В следующем примере выполняется добавление новых кодов товаров, появившихся в данных источника. При этом ставится заслон проникновению различных вариантов именования одного и того же товара путем выбора его с помощью итоговой функции. Не используя в данном случае функции MAX, запрос мог бы вернуть несколько строк для одного и того же кода товара:

```
INSERT INTO Warehouse.dbo.dimProduct (ProductCode, ProductName)
  SELECT stage.Code, MAX(stage.Name)
    FROM Staging.dbo.Orders stage
      LEFT OUTER JOIN Warehouse.dbo.dimProduct dim
       ON stage.Code=dim.ProductCode
    WHERE dim.ProductCode is NULL
```

Загрузка таблиц фактов

Когда все измерения заполнены, можно приступить к загрузке таблицы фактов. Первичные ключи измерения обычно имеют одну из следующих форм:

- естественный ключ, основанный на данных измерения (например, код товара);
- суррогатный ключ, не имеющий какой-либо связи с данными (например, столбец идентичности).

Суррогатные ключи используют чаще всего; они отлично адаптируются к данным, получаемым из множества источников, однако при загрузке каждый из суррогатных ключей требует наличия объединения. Для примера предположим, что наша простая таблица фактов связана с таблицами измерений `dimTime`, `dimProduct` и `dimCustomer`. Если `dimCustomer` и `DimProduct` используют суррогатные ключи, загрузка может выглядеть так:

```
INSERT INTO Warehouse.dbo.factOrder
    (OrderDate, CustomerID, ProductID, OrderAmount)
  SELECT o.Date, c.CustomerID, p.ProductID, ISNULL(Amount,0)
    FROM Staging.dbo.Orders o
      INNER JOIN Warehouse.dbo.dimCustomer c
        ON o.CustCode = c.CustomerCode
      INNER JOIN Warehouse.dbo.dimProduct p
        ON o.Code = p.ProductCode
```

Поскольку таблица `dimTime` сама связана с таблицей фактов по значению даты, то для определения этого отношения не требуется объединения. Меры могут быть преобразованы в читаемый вид, при этом избегая пустых значений. В рассматриваемом примере, если встретится пустое значение объема продаж, его лучше преобразовать в нуль.



Процесс ETL (извлечения, преобразования и загрузки) состоит из большого числа относительно простых действий, модифицируемых по мере изменения источника данных. Централизуйте логику процесса ETL, насколько это возможно, документируйте ее неочевидные аспекты и осуществляйте строгий контроль. Если процесс потребует сопровождения, то это упростит раскрытие всех компонентов и истории их изменения. В этом отношении лучшими инструментами являются Integration Services и SourceSafe.

Знакомство со службой анализа

Лучший способ познакомиться со службой анализа и хранилищами состоит в использовании утилиты Business Intelligence Development Studio (BIDS) для создания базы данных службы анализа и ассоциированных таблиц на базе структур, определенных в учебной базе данных AdventureWorks. Начнем с идентификации и создания базы данных хранилища SQL Server. Затем откроем утилиту BIDS и создадим новый проект службы анализа.

Щелкните правой кнопкой мыши на узле `Cubes` в Solution Explorer и выберите в контекстном меню пункт New Cube. На первой странице открывшегося мастера кубов (Select Build Method) выберите пункт Build cube without data source (Построение куба без источника данных), установите параметр Use a Cube Template и выберите из списка шаблон, соответствующий вашей редакции SQL Server. Пройдите все остальные страницы мастера, принимая предложенные по умолчанию параметры, за исключением страницы Define Time Period, на которой отметьте несколько дополнительных периодов (например, `Year/Quarter/Month`), чтобы сделать измерение времени более интересным.

На заключительной странице мастера выберите опцию Generate Schema Now для запуска мастера генерации схем. Пройдите по страницам мастера, оставив без изменения предложенные по умолчанию параметры и указав только место размещения схемы. По завершении работы мастера будут созданы все объекты службы анализа и связанные с ними. Даже если сгенерированная схема не в полной мере удовлетворяет текущим потребностям, она представит нам довольно интересный образец.

Мастер кубов позволяет выполнить множество настроек определенной шаблоном структуры, и все необходимые изменения можно выполнить в проекте службы анализа вручную. Регенерация схемы может быть выполнена в любое время щелчком правой кнопкой мыши на проекте в Solution Explorer и выбором в контекстном меню пункта Generate Relation Schema.

Более сложные системы могут быть построены путем изначального создания необходимых измерений. Для этого, когда возможно, используется шаблон, а затем по мере необходимости создаются измерения и атрибуты. После этого можно использовать мастер кубов без шаблона, ввести необходимые меры и включить предварительно созданные измерения.

Архитектура службы анализа



Несмотря на то что служба анализа является составной частью SQL Server с версии 7.0, версия 2005 предложила полностью обновленный продукт. Многие из ключевых концепций остались неизменными, однако детали их реализации отличаются настолько, что все функции, описываемые далее в этой главе, можно считать новыми.

Служба анализа основана на концепции хранилища данных. Она предназначена для представления информации в многомерном формате, в отличие от двухмерной парадигмы реляционных баз данных. Спрашивается, как же в службе анализа реализована многомерность? При выборке набора реляционных данных запрос идентифицирует значения по координатам столбцов и строк. Многомерное хранилище полагается на выбор одного или нескольких

элементов из каждого измерения для идентификации возвращаемых значений. Также результирующий набор данных в реляционной базе данных представляет собой последовательность строк и столбцов, в то время как многомерная база может быть организована вдоль множества осей, зависящих от того, что определено в запросе.

Вместо двухмерной таблицы служба анализа для хранения данных использует многомерный куб. Этот куб представляет собой сущность, доступ к которой осуществляется посредством *многомерных выражений (Multidinesional Expression или MDX)* — эквивалента запросов SQL в службе анализа.

Служба анализа также предлагает удобные средства для определения вычислений в выражениях MDX, которые, в свою очередь, обеспечивают новый уровень целостности информационных потоков бизнес-аналитики.



Детали создания запросов и вычислений в выражениях MDX описаны в главе 45.

Служба анализа использует комбинацию стратегий простого и упреждающего кэширования с целью повышения производительности запросов, которая на порядок выше, чем у обычных запросов к хранилищам данных. Например, существующий запрос консолидации данных за последние шесть месяцев, где история за каждый месяц насчитывает 130 миллионов строк, в службе анализа займет пару секунд. В то же время эквивалентный запрос к хранилищу данных потребует больше семи минут.

Унифицированная модель измерений

Унифицированная модель измерений (UDM) определяет структуру многомерной базы данных.

В сердце UDM лежит представление источника данных, которое идентифицирует, какие реляционные таблицы будут поставлять данные в службу анализа, а также отношения между этими таблицами. К тому же эта модель поддерживает присвоение дружественных имен включаемым в представление таблицам и столбцам. На основе представления источника данных определяются группы мер и измерения в соответствии с фактами и измерениями хранилища данных. После этого кубы определяют отношения между измерениями и группами мер, формируя базис для многомерных запросов.

Сервер

Унифицированная модель измерений поддерживается как часть сервера службы анализа (рис. 43.3).

Данные могут храниться в многомерном хранилище OLAP (MOLAP), которое многократно повышает скорость запросов, однако требует предварительной обработки данных источника. Эта обработка обычно осуществляется в форме запросов SQL, управляемых унифицированной моделью измерений, которые направляются в базу данных для извлечения соответствующей информации. В качестве альтернативы данные в хранилище MOLAP могут направляться непосредственно из потока службы интеграции.

В дополнение к хранению мер на уровне деталей служба анализа может хранить предварительно вычисленные итоговые данные, называемые *агрегациями*. Например, если агрегации по месяцам и товарным линиям создаются как часть процесса обработки первичных данных, то запросам, требующим эту комбинацию значений, нет никакой необходимости обращаться к исходным данным и подводить итоги по ним — они могут напрямую использовать существующие агрегации.

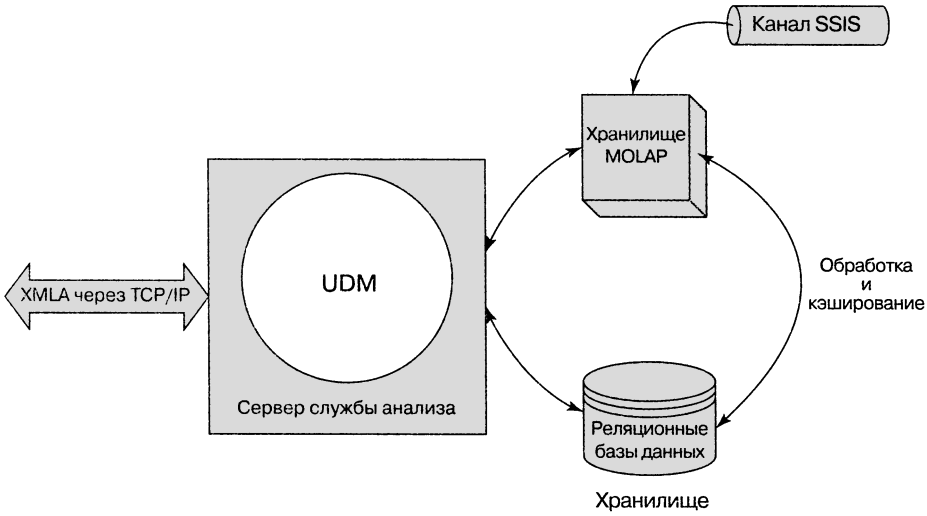


Рис. 43.3. Сервер службы анализа

В принципе, данные могут оставаться и в реляционной базе (хранилище ROLAP), что связано с ускорением времени обработки, но за счет замедления выполнения запросов в несколько раз. Без агрегаций запросы к хранилищу ROLAP являются эквивалентом обычных запросов SQL. Агрегации могут быть вычислены и для хранилищ ROLAP, однако это потребует обработки всех первичных данных. Таким образом, хранилища MOLAP являются оптимальным вариантом. Реляционная база в этом контексте не ограничена только сервером SQL Server — она может использовать любой источник данных, для которого существует поставщик OLE DB.

Упреждающее кэширование обеспечивает компромисс между скоростью хранилища MOLAP и необходимостью предварительной обработки. В данном случае запросы посылаются к хранилищу MOLAP всегда, когда это возможно, за исключением запросов к последним реляционным данным, которые еще не были обработаны для помещения в хранилище MOLAP.

В качестве ключевого протокола служба анализа использует XML for Analysis (XMLA).

Клиент

Клиент взаимодействует со службой анализа точно так же, как и с любой другой Web-службой, — посредством протокола Simple Object Access Protocol (SOAP). Клиентские приложения скрывают детали XMLA и SOAP, используя для доступа к службе анализа различные интерфейсы.

- Все языки программирования семейства .NET используют интерфейс ADOMD.NET.
- Приложения Win32 (например, написанные на C++) могут использовать интерфейс OLE DB для драйвера OLAP (`MSOLAP90.DLL`).
- Остальные приложения COM (например, написанные на VB6, VBA) могут использовать интерфейс ADOMD.

В то время как сервер может общаться на языке XMLA посредством протокола TCP/IP, клиенты имеют возможность использовать протокол HTTP, если соответствующим образом настроенный сервер IIS доступен для перевода.

Помимо приложений доступ к службе анализа может быть осуществлен посредством некоторых инструментов, включая приведенные ниже.

- Утилита Business Intelligence Development Studio, предназначенная для определения структуры базы данных.
- Утилита SQL Server Management Studio — для управления сервером и выполнения запросов к нему.
- Служба Reporting Services, которая может основывать определения отчетов на данных службы анализа.
- Средства и надстройки программы Excel, предназначенные для выполнения запросов и анализа данных.

Также доступен широкий спектр сторонних инструментов, использующих средства службы анализа.

Создание базы данных

База данных службы анализа создается путем идентификации данных, включаемых в нее, определения отношений между этими данными, определения структуры измерений и построения одного или нескольких кубов, комбинирующих меры и измерения. В этом разделе будет описан общий процесс, но с акцентом на сборе данных, необходимых для определения базы данных. В последующих разделах мы рассмотрим многие особенности измерений и кубов.

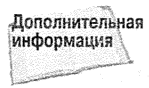
Утилита Business Intelligence Development Studio

Процесс создания базы данных службы анализа начинается с открытия нового проекта в утилите Business Intelligence Development Studio (BIDS). Каждому проекту соответствует база данных, которая будет создана на сервере назначения при развертывании проекта.



Наряду с открытием проекта службы анализа также можно непосредственно открыть существующую базу данных в BIDS. Несмотря на то что это — полезное средство проверки конфигурации запущенного сервера, изменения все равно должны выполняться в проекте, развертываться на сервере разработки, тестироваться и только затем развертываться на производственном сервере. Храните проект и связанные с ним файлы под контролем программы управления версиями.

Перед попыткой развертывания новой базы данных обязательно определите сервер назначения. Для этого щелкните правой кнопкой мыши на проекте в Solution Explorer и выберите в контекстном меню пункт Properties. На странице параметров развертывания выберите сервер назначения с интересующей вас конфигурацией (например, сервер разработки или эксплуатационный сервер). Будьте внимательны при установке и старайтесь не допустить развертывания базы данных на неверном сервере.



Более подробно о работе утилиты Business Intelligence Development Studio см. в главе 6.

Источники данных

Для каждой исходной базы данных или другого источника, необходимого базе данных службы анализа, определите отдельный источник данных. Каждый источник данных содержит строку подключения, параметры аутентификации, а также свойства чтения определенного множества данных. Такой источник можно определить для любых данных, для которых

существует поставщик OLE DB. Это позволяет службе анализа использовать множество типов источников данных, отличающихся от традиционных реляционных баз.

Запустите мастер новых источников данных, щелкнув правой кнопкой мыши на папке Data Sources в Solution Explorer и выбрав в контекстном меню пункт New. После окна приветствия откроется страница, содержащая список подключений. Выберите соответствующее подключение, если таковое существует.

Если подходящее подключение еще не существует, создайте новый диспетчер подключений, щелкнув на кнопке New. В самом диспетчере подключений выберите подходящего поставщика, отдавая предпочтение OLE DB в интересах повышения производительности. После этого введите имя сервера, информацию об аутентификации, имя базы данных, а также прочие параметры, необходимые для конкретного поставщика. Во вкладке All просмотрите все параметры и протестируйте подключение, прежде чем щелкнуть на кнопке OK и завершить процесс создания.

Пройдите через остальные окна мастера, введя соответствующую регистрационную информацию для среды назначения и имя источника данных.

При управлении множеством проектов в пределах одного решения может оказаться полезным базировать регистрационную информацию одного проекта на аналогичной информации другого. В этом случае, вместо выбора подключения в соответствующем окне мастера, выберите опцию Create a data source based on another object. В открывшемся окне мастера Data sources from existing object будут предложены две альтернативы.

- Вариант Creating a data source based on an existing data source минимизирует число мест, в которых информация о подключении должна быть отредактирована в случае ее изменения.
- Вариант Create a data source based on an analysis services project позволяет двум проектам совместно использовать одни и те же данные. Это аналогично использованию для доступа к данным поставщика OLE DB службы анализа, однако в данном случае базы данных могут проектироваться одновременно без дополнительных сложностей развертывания.

Представление источника данных

В то время как источник данных описывает, где искать таблицы данных, представление источника данных определяет, какие доступные таблицы использовать и какие отношения установить между ними. С представлением источника данных также ассоциированы метаданные, такие как дружественные имена и вычисления, выполняемые с таблицами и столбцами.

Создание представления источника данных

Для создания представления источника данных выполните следующие действия.

1. Добавьте в представление источника данных необходимые таблицы и именованные запросы.
2. Определите логичные первичные ключи для таблиц, которые их не имеют.
3. Определите отношения между связанными таблицами.
4. Назначьте таблицам и столбцам дружественные имена и вычисления.

Стоит начать создание представления источника данных с запуска мастера. Щелкните правой кнопкой мыши на папке Data Source Views и выберите в контекстном меню пункт New. Открывшийся мастер имеет несколько страниц.

- Select a Data Source. Выберите один или несколько источников данных, которые следует включить в представление. Если таких источников несколько, то первым должен быть источник SQL Server.
- Name Matching. Эта страница открывается только в том случае, если в базе данных источника не существуют внешние ключи, реализующие отношения, основанные на соглашении об общих именах. Режим поиска соответствия можно включить, установив параметр NameMatchingCriteria при создании представления. При этом соответствия по именам будут отыскиваться сразу же при включении в представление новых таблиц.
- Select Tables and Views. Перетащите используемые в представлении объекты с левой панели (доступные объекты) на правую (используемые объекты). Чтобы сократить список доступных объектов, введите часть имени таблицы в поле Filter и щелкните на кнопке Filter. Для добавления объектов, связанных с включенными в представление, выделите один или несколько таких объектов и щелкните на кнопке Add Related Tables. То же диалоговое окно (только с заголовком Add/Remove Tables) используется и при изменении представления источника данных после его создания.
- Completing the Wizard. В этом окне задайте имя для представления источника данных.

После того как представления источника данных будет создано, состав таблиц в нем можно изменить. Для этого следует щелкнуть правой кнопкой мыши на диаграмме и выбрать в контекстном меню пункт Add/Remove Tables. Этот же метод можно использовать и для добавления таблиц из других источников данных.

Аналогично представлению SQL, в представление источника данных можно добавлять именованные запросы. Они будут вести себя так, будто являются таблицами. Чтобы открыть конструктор запросов, либо щелкните правой кнопкой мыши на диаграмме и выберите пункт контекстного меню New Named Query. либо щелкните правой кнопкой мыши на таблице и выберите пункт Replace Table/with New Named Query. В открывшемся конструкторе можно определить содержимое именованного запроса. Если результирующий набор данных именованного запроса похож на некоторую таблицу. то лучше ее заменить им. Это связано с тем, что именованный запрос будет служить значением по умолчанию для запроса, воссоздающего заменяемую таблицу. Использование именованных запросов устраняет необходимость создания представлений в источнике данных и позволяет сосредоточить все метаданные в одной модели.

По мере добавления таблиц в представление источника данных первичные ключи и уникальные индексы источника импортируются в качестве первичных ключей модели. Внешние ключи и выбранные совпадающие имена (о соглашении об общих именах см. выше) автоматически импортируются в качестве отношений между таблицами. В случае, когда первичные ключи или отношения не импортированы, их нужно определить вручную.

В таблицах, не имеющих первичных ключей, выберите один или несколько столбцов, которые будут формировать ключ, щелкните на одном из них правой кнопкой мыши и выберите в контекстном меню пункт Set Logical Primary Key. Когда все необходимые первичные ключи будут созданы, все таблицы, не имеющие нужных отношений, могут быть связаны перетаскиванием соответствующих столбцов между таблицами. Если новое отношение допустимо, то оно будет добавлено в модель без какого бы то ни было предупреждения; в противном случае откроется диалоговое окно Edit Relationship. Проще всего в последнем случае щелкнуть на кнопке Reverse и скорректировать направление отношения (рис. 43.4); также можно, в зависимости от типа ошибки, предпринять дополнительные действия. Чаще всего при работе с множеством источников данных возникает проблема несоответствия типов. Например, ключ в одной базе может быть 16-разрядным целым числом, а в другой — 32-разрядным целым. В данной ситуации проблема решается путем создания именованного запроса, преобразующего 16-разрядное целое в его 32-разрядный эквивалент.

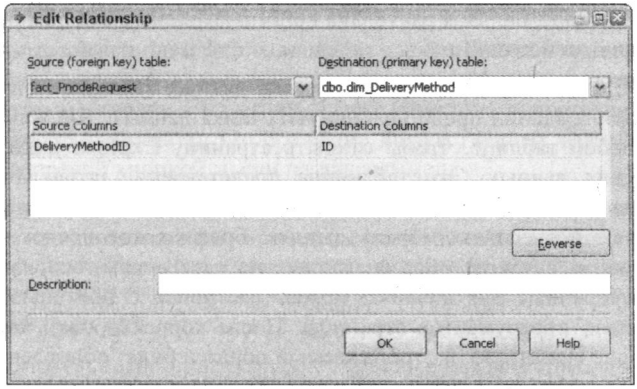


Рис. 43.4. Диалоговое окно Edit Relationship

Диалоговое окно Edit Relationship также можно открыть двойным щелчком на существующей связи, щелчком правой кнопкой мыши на диаграмме или с помощью панели инструментов или меню. Обязательно определите все необходимые отношения, в том числе между разными столбцами таблицы фактов и аналогичными столбцами таблицы измерений (например, столбцы `OrderDate` (дата заказа) и `ShipDate` (дата доставки) связаны с таблицей измерений `Time`), так как это включает ролевую функциональность измерений при создании куба.

Управление представлением источника данных

По мере увеличения числа таблиц, участвующих в представлении источника данных, ими становится все сложнее управлять. Лучше всего справиться с этой сложностью, разбив таблицы на множество диаграмм. Панель диаграмм, находящаяся в верхнем левом углу страницы Data Source View, изначально содержит только одну диаграмму — `<All Tables>`. Для создания новой диаграммы щелкните правой кнопкой мыши на панели диаграмм и выберите в контекстном меню пункт New. После этого перетащите таблицы с панели Tables, находящейся в левом нижнем углу, на новую диаграмму. В качестве альтернативы можете щелкнуть правой кнопкой мыши на новой диаграмме и с помощью диалогового окна Show Tables включить таблицы, содержащиеся в настоящее время на общей диаграмме `<All Tables>`. Не путайте диалоговое окно Show Tables, определяющее отображение представления источника данных на диаграмме, с диалоговым окном Add/Remove Tables, которое формирует состав таблиц самого представления.

Среди других инструментов управления источником данных стоит упомянуть следующие.

- **Панель** Tables. На этой панели перечислены все таблицы, включенные в представление источника данных. Щелкните на любой таблице, и она будет показана и выделена на текущей диаграмме (если она на ней существует).
- **Диалоговое окно** Find Table. Открывается из панели инструментов или меню. В нем перечислены только таблицы, содержащиеся на текущей диаграмме. Это окно позволяет ускорить процесс поиска нужного объекта на диаграмме. Щелкните на имени таблицы, и она будет показана и выделена на диаграмме.
- **Инструмент** Locator. Находится он на пересечении вертикальной и горизонтальной полос прокрутки и позволяет быстро прокручивать текущую диаграмму Щелкните и перетаскивайте этот инструмент для быстрого перемещения по диаграмме.
- **Переключатель** Switch Layout. Для переключения между прямоугольной и диагональной раскладками щелкните правой кнопкой мыши на диаграмме. Прямоугольная раскладка организована по таблицам и незаменима для общего понимания множества

взаимосвязей. Диагональная раскладка организована по столбцам и подходит для просмотра деталей отношений.

- **Страница** Explore Data. Просмотр образца данных таблицы может оказаться очень полезным при создании представления источника данных. Щелкните правой кнопкой мыши на любой таблице, чтобы открыть страницу Explore Data, представляющую табличный вид данных. Это табличное представление позволяет непосредственно проверить данные, в то время как представления сводной таблицы и сводной диаграммы позволяют исследовать модель данных. Графическое представление отображает набор диаграмм, разбивая образец данных по категориям, основанным на столбцах. Столбцы, выбранные для анализа, можно настроить с помощью кнопки Sampling Options панели инструментов страницы. После корректировки характеристик щелкните на кнопке Resample, и отображаемый образец будет обновлен.

Представление источника данных можно представить себе как кэш используемых им схем, открывающий ответную среду моделирования. Подобно любому другому кэшу, представление может стать устаревшим. Если изменилась схема, используемая представлением, щелкните правой кнопкой мыши на диаграмме и выберите в контекстном меню пункт Refresh — все изменения в схеме источника данных сразу же отобразятся в представлении. Функция обновления представления также доступна на панели инструментов и в меню. Если открыть диалоговое окно Refresh Data Source View, то в нем будут перечислены все изменения, влияющие на представление источника данных. Перед тем как принять эти изменения, просмотрите список на предмет наличия удаленных таблиц. Если удаленные таблицы будут найдены, отмените изменения, после чего проверьте саму схему на предмет переименованных или реструктурированных таблиц — возможно, отсутствующие данные можно заменить эквивалентными. Только после разрешения всех конфликтов можете попытаться выполнить обновление представления. Например, можно щелкнуть правой кнопкой мыши на переименованной таблице и выбрать в контекстном меню пункт Replace Table/with Other Table, после чего выбрать новую таблицу. Такой подход позволит избежать при обновлении потери взаимосвязей таблиц и прочей контекстной информации.

Тонкая настройка представления источника данных

Одной из сильных сторон унифицированной модели измерений является то, что запросам совсем не обязательно знание структуры и взаимосвязей исходных таблиц. В то же время зачастую даже имя таблицы предоставляет пользователю важную информацию о семантике. Например, ссылка на столбец `accounting.hr.staff.employee.hourlu_rate` дает понять, что в нем содержится информация о почасовой ставке работника, содержащейся на бухгалтерском сервере `accounting`, при этом используются база данных `hr`, схема `staff` и таблица `employee`. Так как этот источник данных скрыт за унифицированной моделью измерений, эта семантика будет потеряна. Представление источника данных позволяет любому столбцу или таблице присвоить дружественные имена. Представление также содержит параметр описания для каждой таблицы, столбца и отношения. Дружественные имена и описания позволяют скрыть существующую семантику и. в случае необходимости, заменить ее другой.


Проверено

Сделайте представление источника данных единым центром управления метаданными. Если столбец должен быть переименован в ходе запроса, присвойте ему дружественное имя в представлении источника данных, вместо того, чтобы переименовывать атрибут меры или измерения. Эти два имени отображаются рядом в представлении источника данных, что поможет в будущем программистам понять порядок использования данных. Используйте параметр описания для неочевидных примечаний, описывающих результаты исследований, которые понадобились при создании и модификации модели.

Назначение дружественных имен и описаний таблицам и столбцам осуществляется путем выделения соответствующего элемента и ввода на панели Properties соответствующей информации. Добавление описаний к отношениям происходит аналогично: выделите связь и скорректируйте параметры на панели Properties. Также описание отношения можно ввести и в диалоговом окне Edit Relationship. Отображение дружественных имен можно включать и отключать щелчком правой кнопкой мыши на диаграмме.



Приложения и отчеты, основанные на данных службы анализа, скорее всего, будут впоследствии изменяться в организации заказчика решения. Назначайте такие дружественные имена, которые приняты для данных объектов в принимающей организации. Это позволит ускорить адаптацию продукта и сделать его более понятным.

В представление источника данных можно также включить и множество простых вычислений. Считается хорошим тоном помещать в представление вычисления, которые зависят только от одной строки, одной таблицы или одного именованного запроса, а многострочные и многотабличные вычисления отдавать на откуп выражениям MDX. В именованные запросы вычисления добавляются путем их программирования непосредственно в запросе. Чтобы добавить вычисления в таблицу, щелкните на ней правой кнопкой мыши и выберите в контекстном меню пункт New Named Calculation. Введите имя вычисления и любое выражение, которое сможет интерпретировать используемый поставщик данных.

Создание куба

Представление источника данных формирует базис для создания кубов, которые, в свою очередь, представляют данные пользователям. Запуск мастера кубов с автопостроением (Cube Wizard with Auto Buld), как правило, позволяет получить неплохой черновой вариант куба. Щелкните правой кнопкой мыши на папке Cubes и выберите в контекстном меню пункт New. Откроется мастер, предлагающий пройти несколько последовательных страниц.

- Select Build Method. В качестве метода построения выберите вариант Build the cube using a data source (создать куб, используя источник данных), а также установите флажки Auto Build (Автопостроение) и Create attributes and hierarchies (Создать атрибуты и иерархии).
- Select Data Source View. На этой странице выделите соответствующее представление источника данных, на котором будет основан куб.
- Detecting Fact and Dimension Tables (Обнаружение таблиц фактов и измерений). Мастер подготавливает значения по умолчанию для следующей страницы.
- Identify Fact and Dimension Tables (Идентификация таблиц фактов и измерений). На этой странице вы должны выбрать, какие таблицы содержат факты (меры), а какие — измерения, а какие и то и другое. Во вкладке Tables эта информация представлена в виде простого списка, а во вкладке Diagram — в виде диаграмм, составленных из разноцветных блоков таблиц (синие блоки представляют измерения, желтые — факты, а зеленые содержат оба типа информации). Здесь также могут быть определены и временны*е* измерения, однако этот процесс требует детального понимания их конфигурирования. Большинство пользователей предпочитают не заниматься временны*ми* измерениями в мастере. (Временны*е* измерения мы рассмотрим в следующем разделе.)
- Review Shared Dimensions. Перетащите измерения, включаемые в куб, с левой панели (доступные измерения) на правую (измерения куба). В этом списке перечислены как существующие измерения, так и добавленные мастером. Иногда мастер может принять не-

корректное решение относительно того, как комбинировать таблицы в измерения, — такие измерения лучше исключить на данном этапе и добавить вручную позже.

- Select Measures (Выбор мер). На этой странице отметьте только те столбцы, которые сослужат полезную службу в качестве мер. По умолчанию мастер включает в таблицу фактов все, что не используется в отношениях между таблицами. По этой причине в этой таблице может оказаться слишком много столбцов. Столбцы организованы в алфавитном порядке в каждой из обработанных таблиц фактов. Имена мер можно редактировать, однако обычно потребность в изменении имени является признаком того, что все-таки стоило присвоить дружественные имена в представлении источника данных.
- Detecting Heirarchies (Раскрытие иерархий). Мастер будет пытаться обнаружить иерархические взаимосвязи в столбцах (атрибутах) таблицы измерений во всех добавленных измерениях.
- Review New Dimensions (Формирование новых измерений). На этой странице отметьте те атрибуты и иерархии, которые должны быть включены в новые измерения. Включайте только те элементы, которые позволяют хорошо структурировать данные по категориям (например, по названиям, кодам и описаниям). Исключите те элементы, которые не могут помочь в структурировании, например, непонятные пользователю идентификаторы или даты вставки объекта.
- Completing the Wizard. На последней странице мастера введите имя куба.

После завершения работы мастера будет создан куб и, возможно, несколько новых измерений. Количество созданных измерений зависит от количества таблиц, отмеченных как поставляющие информацию об измерениях, и от того, сколько этих измерений было определено ранее.

Измерения

В дискуссии о схеме "звезда" мы уже говорили, что измерения представляют собой полезные средства разбиения данных на категории, используемые при консолидации, — нечто вроде атрибутов группировки, используемых в запросах SQL. Измерения, созданные мастером, как правило, дают неплохое первое приближение того, что действительно нужно. Таким образом, перед развертыванием базы для эксплуатации эти измерения нужно подправить.

Обстоятельное изучение свойств измерений — довольно сложная тема. К счастью, большая часть работы выполняется в ходе относительно простой установки. В этом разделе мы вначале поговорим о базовой функциональности, после чего исследуем более сложные вопросы.

Конструктор измерений

Откройте любое измерение в Solution Explorer, чтобы вызвать конструктор, показанный на рис. 43.5. Этот конструктор выводит информацию в трех представлениях, разделенных вкладками.

- Dimension Structure. Представляет собой базовую рабочую область конструктора для определения измерений. Наряду с вездесущими панелями Solution Explorer и Properties в этом представлении открываются три дополнительные панели. На панели Data Source View, расположенной по центру, отображается фрагмент представления источника данных, на основе которого построено измерение. На панели Attributes в левой нижней части окна перечислены все атрибуты, включенные в измерение. Панель Hierarchies and Levels в левой верхней части окна позволяет организовать атрибуты в стандартную иерархическую структуру.

- Translations. На данной вкладке определяется альтернативная версия данных и объектов на другом языке.
- Browser. На этой вкладке отображаются данные измерения, которое было последним развернуто на целевом сервере анализа.

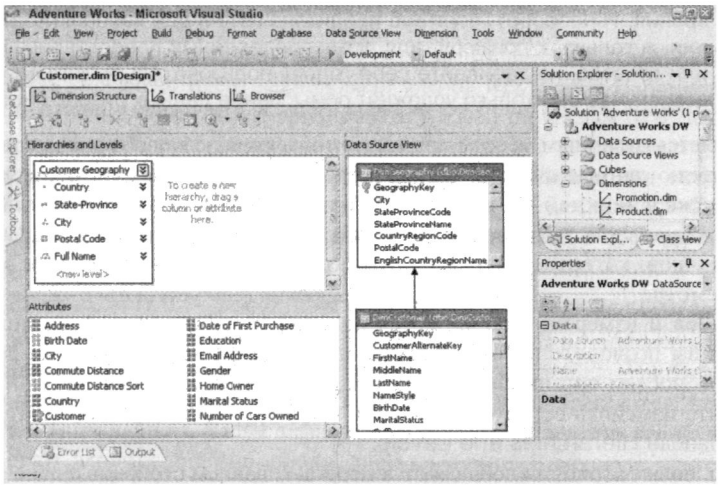


Рис. 43.5. Конструктор измерений с открытым измерением `Customer` *базы AdventureWorks*

В отличие от источников данных и их представлений, кубы и измерения должны развертываться, прежде чем их работа может быть исследована (например, путем просмотра данных). Процесс развертывания измерения состоит из двух частей. На первом этапе — построения — определение измерения (или изменения, внесенные в определение определения) отправляется на целевой сервер анализа. При этом в окне вывода отображается индикатор выполнения. На втором этапе — обработки — сервер службы анализа выполняет запросы к исходным данным и заполняет измерения. Ход этого процесса отображается в окне Deployment Progress, которое обычно выглядит как вкладка панели Properties. Утилита BIDS пытается создать или обработать только измененную часть проекта, чтобы минимизировать время развертывания.

Атрибуты и иерархии

Панель Attributes заполняется одним атрибутом для каждого из столбцов таблиц источников, которые было разрешено создать мастеру. Значок ключа указывает на ключевой атрибут, соответствующий первичному ключу источника данных, используемому для связывания с таблицей фактов. В каждом измерении должен существовать ровно один ключевой атрибут.

После развертывания каждый атрибут, который не был специально отключен, становится *иерархией атрибутов* для просмотра и выполнения запросов. Иерархия атрибутов в общем случае содержит два уровня: `All`, представляющий все возможные значения атрибута, и уровень, названный именем самого атрибута, перечисляющего отдельно каждое значение.

Панель Hierarchies and Levels позволяет создать *пользовательские иерархии*, определяющие принятые в организации уровни организации атрибутов. В качестве примера на рис. 43.5 показана пользовательская иерархия, которая вначале была представлена в браузере как список стран, который может быть далее расширен как список регионов, затем городов и т.д.



Пользовательские иерархии реализуют пути раскрытия различных уровней иерархии атрибутов пользователем, который интерактивно исследует содержимое куба. По этой причине достаточно важно заложить те пути раскрытия, которые имеют смысл для потенциального пользователя (а не разработчика). Потратьте некоторое время на исследование того, как различные пользователи мыслят себе представление данных, а затем адаптируйте структуру к их точке зрения. И наоборот, если куб предназначен не для интерактивного использования, а только, к примеру, для предопределенных представлений в приложении или отчетов, не забивайте себе голову пользовательскими иерархиями. В данном случае они только усложнят осмысление данных разработчиком.

Браузер является отличным местом для того, чтобы понять, как оба типа иерархий представляют пользователю данные службы анализа. Выберите исследуемую иерархию из раскрывающегося списка в верхней части представления, а затем последовательно раскрывайте узлы каждого из уровней, постепенно углубляясь в структуру данных. Обратите внимание на различие значков, которыми маркируются пользовательские иерархии и иерархии атрибутов при просмотре.

Мастера кубов и измерений попытаются идентифицировать и создать пользовательские иерархии, если вы позволите им это сделать, однако их решение может не соответствовать реальным потребностям организации. Перетаскивайте атрибуты с панели Attributes в иерархию, если хотите изменить ее; чтобы начать новую иерархию, перетащите атрибут в пустое пространство панели Hierarchies and Levels.

Аналогично, новые атрибуты добавляются перетаскиванием столбцов с панели Data Source View. Если таблица, столбцы которой вы хотите добавить в список атрибутов, не отображается, щелкните правой кнопкой мыши на панели Data Source View и воспользуйтесь командой Show Table для включения интересующей вас таблицы. Как только атрибут из таблицы будет включен в измерение, представление Show Only Used Table будет отображать новую таблицу. Естественно, каждый из атрибутов должен иметь обратную связь с таблицей фактов с помощью того же ключевого атрибута, даже если этот ключ участвует в схеме "снежинка".

Столбцы источника атрибутов и упорядочение

Столбцы представления источника данных назначаются параметрам KeyColumns и NameColumn атрибута и используются для указания на источник заполнения атрибута данными. В процессе обработки служба анализа включает эти столбцы ключа и имени в инструкцию `SELECT DISTINCT`, которая выполняется относительно исходных данных с целью заполнения атрибута. Значение параметра KeyColumns определяет, какие элементы будут включены в качестве членов атрибута. Необязательное значение NameColumn присваивает ключу экранное имя, если само имя ключа описывает его неадекватно. Для большинства атрибутов при создании вполне достаточно одного ключевого атрибута. Например, атрибут `Address` в измерении `Customer`, как правило, представляет собой обычную текстовую строку, не имеющую ассоциированного идентификатора или кода. В данном примере вполне достаточно назначить столбец `Address` в качестве значения KeyColumn без присвоения значения NameColumn.

Рассмотрим некоторые более сложные сценарии.

- **Атрибуты с идентификатором/кодом и именем**. Данный случай типичен для таблиц измерений с первичным ключом. Подход в этом случае зависит от того, имеет ли смысл для пользователей, обращающихся к измерению с запросом, данный идентификатор или код. Если код общеизвестен, оставьте поле NameColumn пустым, чтобы избежать сокрытия кода. Здесь поля идентификатора и имени будут представлены в модели как отдельные атрибуты. Если же код является внутренним значением приложения или хранилища, скройте его, определив в одном атрибуте оба параметра: KeyColumns и NameColumn.

- **Идентификатор/код существует без соответствующего имени**. Если идентификатор или код имеет всего несколько значений, назначьте его параметру NameColumn, добавив для этого именованное вычисление в представление источника данных. Если же идентификатор/код имеет множество значений или их непредсказуемый состав, подумайте о создании объединенной в схему "снежинка" новой таблицы измерений, содержащей имя.
- **Неуникальные ключи**. Очень важно, чтобы присвоенные значения параметров KeyColumns уникально идентифицировали членов измерения. Например, таблица временного измерения может идентифицировать месяцы по числам от 1 до 12, которые не являются уникальными ключами в разрезе множества лет. В данном случае для обеспечения уникальности можно включить в состав ключа как месяц, так и год. Если используется подобный множественный ключ, то значение параметра NameColumn является обязательным. Скорее всего, в этом случае придется добавить в представление источника данных именованное вычисление для синтеза из используемых в ключе столбцов месяца и года понятного пользователю описания (например, `Ноябрь, 2008`).

В приведенном выше сценарии с неуникальным ключом можно попытаться использовать результаты именованного вычисления в качестве ключевого столбца атрибута, если по нему не потребуется упорядочение. Числовые данные месяца и года требуются для поддержания членов атрибута в календарном, а не в алфавитном порядке. Параметр атрибута OrderBy позволяет сортировать атрибут либо по ключу, либо по имени. В качестве альтернативы параметры сортировки AttributeKey и AttributeName позволяют сортировать членов текущего атрибута на основании ключа или имени другого атрибута, если этот второй атрибут был определен как свойство члена текущего атрибута. Свойства членов мы подробно рассмотрим в следующем разделе.

Для изменения параметра KeyColumns щелкните на его значении, а затем на эллипсе с многоточием рядом с ним. Откроется окно редактора DataItem Collection Editor, показанное на рис. 43.6. На левой его панели отображены все члены текущего ключа. Если выделить какой-либо из этих членов, то на правой панели отобразятся его свойства. С помощью кнопок Add и Remove можно изменить количество членов ключа. Назначение столбца члену выполняется путем щелчка на текущем значении параметра Source, а затем на эллипсе. В открывшемся диалоговом окне Object Binding нужно выбрать тип связывания, а также соответствующие таблицу и столбец.

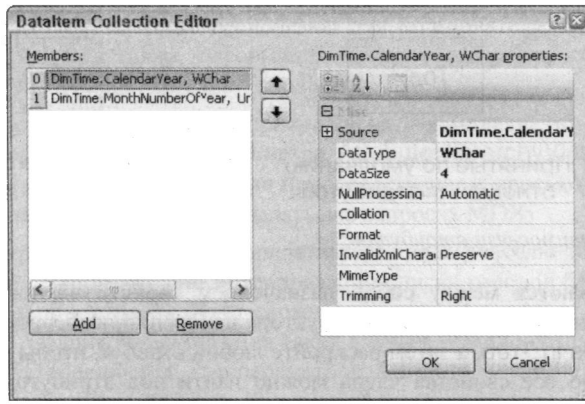


Рис. 43.6. Окно DataItem Collection Editor используется для просмотра членов и установки их свойств

Для того чтобы добавить и изменить параметр NameColumn атрибута, выберите соответствующее значение из раскрывающегося списка. Если выбрать в списке открывшегося диалогового окна Object Binding значение (none), то назначение имени будет очищено.

Взаимосвязи атрибутов

Одним из важных средств, с помощью которых служба анализа достигает высокой скорости выполнения запросов, является предварительное вычисление итогов. Эти итоги, или агрегации, создаются в соответствии с атрибутами измерения. Например, итоги по годам и месяцам могут быть предварительно вычислены вдоль оси времени.



Тщательно проанализируйте все измерения и убедитесь, что все взаимосвязи установлены правильно. Поверьте, стоит ли повышение производительности ваших усилий.

Как агрегацию, так и обработку запросов можно сделать более эффективной, если взаимосвязи между атрибутами отражены в структуре измерения. Рассмотрим в качестве примера простое измерение времени, в котором существуют атрибуты года, квартала, месяца и дня, ссылающиеся на таблицу фактов (ключевой атрибут). По умолчанию каждый атрибут в измерении непосредственно связан с ключевым атрибутом. Результирующая система взаимосвязей показана на рис. 43.7. Агрегации всех не ключевых уровней суммируют все итоги за день. Противоположностью этому является правильно организованная система взаимосвязей, где агрегации уровня месяца ссылаются на итоги уровня дня, агрегации уровня квартала — на итоги уровня месяца, а агрегации уровня года — на итоги уровня квартала. Чем больше и сложнее измерение, тем большее влияние на производительность оказывает его правильная организация.

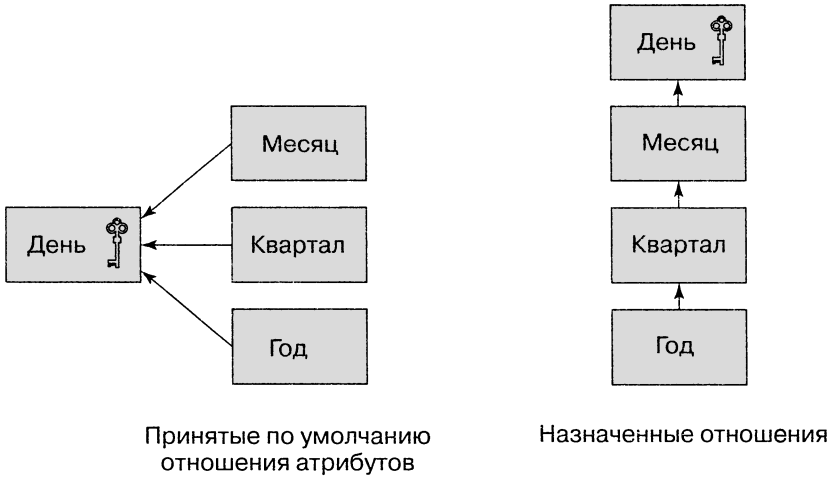


Рис. 43.7. Взаимосвязи атрибутов

Атрибуты связываются между собой назначением свойств членов. Щелкните правой кнопкой мыши на панели Attributes в конструкторе измерений, выберите пункт контекстного меню Show Attributes in Tree, а затем раскройте любой атрибут, чтобы увидеть его свойства члена. По умолчанию все свойства члена можно найти под атрибутом, соответствующим первичному ключу таблицы, из которой они были прочитаны.

Свойство члена должно иметь только одно значение для каждого значения родительского атрибута. Таким образом, год может быть свойством члена квартала, но страна не может

быть свойством члена города, поскольку города с одним и тем же названием порой имеются в разных странах. Именно по этой причине по умолчанию свойству члена назначается столбец первичного ключа, так как такое назначение правильно по определению.

Щелкните на свойстве члена, чтобы просмотреть его ассоциированные параметры. Особенно нас интересует тип отношения, который может принимать два значения.

- `Rigid`. Значение атрибута и свойство члена имеют статичную взаимосвязь, неизменную во времени. Если это отношение со временем изменится, то процессор выдаст ошибку. Этот тип отношения более эффективен, чем `Flexible`, поскольку служба анализа может удерживать агрегации при обработке измерения. Таким образом, квартал закрепляется за месяцем, а штат — за городом.
- `Flexible`. Этот тип используется для значений атрибута и значений свойств члена, которые изменяются в отношении во времени. В этом случае при обработке измерения агрегации обновляются, чтобы учесть произошедшие изменения. Например, подразделение как свойство члена работника должно быть изменяемым, чтобы допустить возможность перевода сотрудника из одного подразделения в другое.

Изменение взаимосвязей в измерении выполняется путем раскрытия атрибутов и перетаскивания свойств членов между ними. Перетаскивание имени атрибута в список свойств члена также допустимо, однако оно приводит к добавлению новой взаимосвязи вместо перенаправления существующей. Избегайте конфигурирования атрибута как свойства члена больше чем один раз, если, конечно, это не требует обеспечение другой функциональности, такой как вычисления или упорядочение.

Взаимосвязи между атрибутами определяют *естественную иерархию*. В отличие от иерархии атрибутов и пользовательской иерархии, этот третий тип незаметен для пользователя измерения. Естественная иерархия определяет, как создаются внутренние структуры агрегаций и индексов.

Видимость и организация

Большинство кубов имеет большое число атрибутов измерений, которые могут ошеломить пользователя. При этом может помочь присвоение им знакомых имен. Однако лучшим способом борьбы с перегрузкой атрибутами является их тщательный отбор. Не стоит включать в атрибуты ту информацию, которая не заинтересует пользователя. В процессе отсеивания обычно используют специальные стратегии, которые приведены ниже.

- Удаление атрибутов, которые бесполезны для пользователя. Среди них непонятные человеку элементы, а также информация об альтернативных языках, которая может быть определена в представлении Translation.
- Для атрибутов, которые должны присутствовать в модели, но используются крайне редко, можно установить для параметра AttributeHierarchyVisible значение `False`. Эти атрибуты будут недоступны при непосредственном просмотре данных куба пользователем, однако на них можно ссылаться в запросах MDX.
- Некоторые атрибуты могут быть представлены пользователю только в пользовательской иерархии. Например, при интерпретации списка городов без знания информации о соответствующих странах будет сложно отличить Париж, находящийся в штате Техас, США, от столицы Франции. Для таких случаев обычно создают пользовательскую иерархию и устанавливают для параметров AttributeHierarchyVisivle соответствующих атрибутов значение `False`.
- Атрибуты, которые не будут использоваться в запросах, однако должны присутствовать в качестве свойств членов, например для сортировки или вычислений, могут быть

полностью отключены. Установите для параметра AttributeHierarchyEnabled значение `False` и обратите внимание на то, что значок атрибута теперь потускнел. Также установите для параметра AttributeHierarchyOptimizedState значение `NotOptimized`, а для параметра AttributeHierarchyOrdered — значение `False`, чтобы сократить время обработки службой анализа данного измерения.

Когда список видимых атрибутов и пользовательских иерархий скомпонован, объедините измерения в папки, более удобные, чем видимые иерархии. Объединение атрибутов выполняется путем ввода имени папки в параметр AttributeHierarchyDisplayFolder; у пользовательских иерархий есть аналогичный параметр — DisplayFolder. В общем случае все эти параметры можно оставить незаполненными для наиболее часто используемых иерархий в измерении, чтобы эти элементы отображались на корневом уровне.



Хорошая организация измерений является ключом к успеху интерактивных приложений, поскольку большинство пользователей может просто накрыть с головой волна доступных атрибутов. Исключение неиспользуемых атрибутов не только поможет упростить просмотр пользователем данных, но и значительно повысит производительность. Особенно это касается кубов с фактическими вычислениями, поскольку, чем больше атрибутов, тем большее число ячеек должно рассматривать каждое вычисление.

Список проверки установок

После создания основного измерения с помощью мастера кубов или мастера измерений выполните предложенные в следующем списке первоочередные проверки. Такой уровень внимания адекватен важности обстоятельств.

- Убедитесь, что имена атрибутов понятны и недвусмысленны в контексте всех измерений модели. Если возникнет потребность внести изменения, измените имена в представлении источника данных и регенерируйте измерение, чтобы сохранить целостность в масштабах модели. В качестве альтернативы можете изменить имена атрибутов и уровни пользовательских иерархий напрямую.
- Пересмотрите все пользовательские иерархии, созданные мастером, и при необходимости скорректируйте уровни. Добавьте отсутствующие иерархии. Замените имена, присвоенные по умолчанию, смысловыми альтернативами.
- Удалите ненужные атрибуты и скорректируйте область их видимости.
- Скорректируйте взаимосвязь атрибутов, чтобы она соответствовала данным, содержащимся в них.
- Пересмотрите все источники атрибутов (параметры KeyColumns и NameColumn) и упорядочение. Для просмотра результатов почаще используйте представление браузера.
- Организуйте измерения с массой иерархий в папки.

Изменение данных в измерениях

Правильная обработка изменений в данных измерений может представлять собой достаточно сложную задачу, и здесь немаловажную роль играет отношение компании к отслеживанию исторических данных. Например, если сотрудник изменил фамилию, важно ли знать предыдущее и настоящее значения соответствующего атрибута? А как поступать с изменившимся адресом клиента? А с изменением кредитной ставки?

Служба анализа самоотверженно отражает изменения, внесенные в данные таблиц измерений при обработке измерения. Она не имеет возможности самостоятельно отслеживать ис-

торию изменений измерений — все должно выполняться на уровне исходных таблиц. Для отслеживания истории таблиц измерений обычно используют четыре стандартных сценария.

- **Slowly Changing Dimension Type 1**. История не отслеживается, поэтому изменения применяются к измерению, распространяясь на весь временной диапазон. Например, когда кредитный рейтинг клиента падает с хорошего до плохого, нет никакой возможности узнать, когда это изменение произошло и, вообще, имел ли клиент когда-либо иной рейтинг. В этом режиме отслеживания сложно объяснить, почему в прошлом квартале товар отпускался клиенту без предоплаты. В то же время этого простого подхода вполне достаточно для многих измерений. При реализации базы данных службы анализа на данных OLTP, а не хранилища данных, это единственно возможный вариант, поскольку базы данных OLTP редко отслеживают исторические сведения.
- **Slowly Changing Dimension Type 2**. Каждое изменение в данных источника отслеживается как история с помощью дополнительных строк в таблице измерения. Например, когда имя заказчика впервые появляется в таблице OLTP, в таблицу измерения для него вводится строка и соответствующие строки фактов связываются с ней. Позже, когда информация о клиенте изменяется в исходной таблице, существующая строка данного клиента в таблице измерения помечается как устаревшая; при этом создается новая строка с новыми данными атрибутов. С этой новой строкой будут ассоциированы все последующие строки таблицы фактов, имеющие отношение к данному клиенту.
- **Slowly Changing Dimension Type 3**. Комбинирует концепции первых двух типов. Отслеживается история некоторых, но не всех изменений, основанная на заданных правилах бизнес-логики. Например, повышение должности сотрудника в пределах одного подразделения может расцениваться как изменение типа 1 (только обновление), а его перевод в другое подразделение — как тип 2 (вставка новой строки измерения).
- **Rapidly Changimg Dimension**. Может случиться так, что какой-то атрибут (или их группа) изменяется настолько часто, что второму подходу приходится генерировать в таблице измерения слишком много строк. Такие атрибуты часто связаны с состоянием и приоритетами. Данный подход предполагает вывод стремительно изменяющегося атрибута в отдельное измерение, напрямую связанное с таблицей фактов. Таким образом, вместо отслеживания изменений в дополнительных строках таблицы измерения таблица фактов в каждой строке будет содержать соответствующее состояние или приоритет.

Несмотря на то что управление изменениями измерений должно осуществляться на уровне хранилища данных, мастер измерений может помочь в создании измерения, поддерживающего функциональность типа 2. Создайте измерение без использования источника данных, а затем определите, что оно будет изменяющимся. После этого мастер займется конфигурацией столбцов, которые будут использованы при поддержке исторических данных.

- Первичный ключ конфигурируется как столбец идентичности (иногда называемый *суррогатным ключом*). Он будет использоваться для обратной связи с таблицей фактов.
- Столбец `Original_ID`, иногда называемый *альтернативным ключом*, обычно является первичным ключом источника данных. Например, это может быть табельный номер сотрудника или идентификатор клиента в исходной таблице OLTP.
- Столбец `Start_Date` указывает на дату начала действия строки измерения.
- Столбец `End_Date` указывает на дату конца действия строки измерения.
- Столбец `State` указывает на активное или неактивное состояние строки.

Сгенерированная мастером схема предполагает, что дата конца действия текущей строки устанавливается в 31 декабря 9999 года, так что столбец состояния, сформированный как вычисляемый, программируется следующим образом:

```
case when [Customer_SCD_End_Date]='12/31/9999'
  then 'Active'
  else 'Inactive' end
```

После того как измерение будет сконфигурировано мастером, добавьте в него дополнительные необходимые атрибуты, а затем щелкните на ссылке панели Data Source View измерения для генерации соответствующей схемы.

За пределами обычных измерений

Концепция измерения, описываемая до сих пор, касалась в основном общей функциональности, характерной для большинства типов измерений. В то же время не было дано объяснений самому понятию "тип измерения". Некоторые первоисточники подразделяют измерения на два типа: стандартные и раскрытия данных. Эти общие типы вобрали в себя все остальные. Каждое измерение имеет параметр типа, которому назначаются такие значения как `Time`, `Geography`, `Customer`, `Accounts` или `Regular`, которые соответствуют тому, что не содержится в списке. Прочие характеристики измерения, такие как структура отношений родительских и дочерних объектов, разрешение записи или ссылок на измерение из других баз данных, также можно рассматривать как определители различных типов измерений.

Сразу отметим, что в настоящей главе мы ограничимся рассмотрением только стандартных измерений и будем рассматривать понятие "тип" только в контексте свойств измерения. В то же время, читая другие первоисточники, важно понимать, насколько всеобъемлющим является понятие типа измерения.

Временное измерение

Практически каждый куб нуждается во временном измерении, однако в большинстве эксплуатируемых кубов это измерение реализовано из рук вон плохо. К счастью, мастер измерений автоматически создает временное измерение и соответствующую ему таблицу, а также заполняет эту таблицу данными. Щелкните правой кнопкой мыши на папке измерения на панели Solution Explorer и выберите в контекстном меню пункт New Dimension. На страницах открывшегося мастера следует выполнить следующие действия.

- Select Build Method. На этой странице выберите метод построения Build the dimension without using a data source, установите флажок Use a Dimension Template и выберите Time из списка шаблонов.
- Define Time Periods. Выберите диапазон дат и периоды, которые должны использоваться в измерении.
- Select Categories. В дополнение к стандартному календарю выберите и сконфигурируйте другие календари, которые должны быть внедрены в измерение.
- Completing the Wizard. При необходимости измените имя измерения, при этом не устанавливайте флажок Generate schema now.

Проанализируйте созданную мастером структуру измерения. Обратите внимание на то, что для параметра измерения Type установлено значение `Time` и все атрибуты также имеют соответствующий временной тип: дни, месяцы, кварталы и т.п. Выполните описанные в предыдущем разделе необходимые проверки и подстройки измерения, корректируя их по мере необходимости. Параметры KeyColumns и NameColumn в данном случае не требуют пристального внимания, в то же время имена, присвоенные элементам иерархий, следует адапти-

ровать для целевой аудитории. Взаимосвязи атрибутов также потребуют настройки. После выполнения всех настроек щелкните на ссылке на панели Data Source View, чтобы создать таблицу временного измерения, задав для нее подходящее имя и место размещения.

Временные измерения можно создавать на основе уже существующих таблиц измерений, используя при этом мастер кубов или измерений. Сложность этого подхода заключается в необходимости определить тип атрибута для каждого из столбцов временного измерения. При интеграции произвольной временной таблицы с помощью мастера сгенерируйте аналогичную таблицу измерения — она послужит вам путеводителем в настройке.

Назначение подходящих типов атрибутам согласно документации может открыть для приложений, использующих куб, новые возможности. Также эти свойства используются некоторыми функциями в самой службе анализа, в том числе и вычислениями бизнес-аналитики.

Серверное временное измерение является альтернативой обычному, полагающемуся на реляционные таблицы. Оно создается самой службой анализа и не допускает такую гибкость, как традиционный подход. В то же время оно может стать удобным подспорьем в создании простых кубов или быстрых прототипов. Для создания серверного временного измерения запустите мастер измерений, как было описано выше, однако в данном случае оставьте не установленным флажок Use a Dimansion Template, а на второй странице мастера выберите вариант Server Time Dimension.

Так как серверное временное измерение не имеет связанной таблицы измерения, оно не отображается в представлении источника данных, так что в нем не может быть описана связь с таблицей фактов. Вместо этого в данном случае для создания связи с выбранными таблицами фактов (также называемыми *группами мер*) следует использовать представление измерений конструктора кубов.

Прочие типы измерений

В дополнение к временному измерению служба анализа распознает больше десяти других типов измерений, в том числе `Customers` (заказчики), `Accounts` (расчеты) и `Products` (товары). Многие из этих типов могут быть созданы с помощью шаблонов, аналогично процессу, описанному для временного измерения. В каждом типе измерения существует множество соответствующих типов атрибутов, отражающих роль этих атрибутов в измерении. Назначение типов измерениям и атрибутам полезно для целей документирования, а также вследствие возможностей, которые они открывают.

Использование доступных шаблонов при создании измерений, соответствующих им таблиц и пакетов загрузки позволяет ускорить процесс конфигурирования и может послужить разработчику хорошим подспорьем.

Измерения типа "родитель–потомок"

Большинство измерений организовано в иерархии, имеющие фиксированное число уровней, однако такое ограничение не подходит для многих практических решений. Например, даже минимальное изменение в структуре организации может вызвать добавление в организационную диаграмму нового уровня. Реляционные базы данных решают эту проблему с помощью рекурсивных таблиц, а служба анализа — с помощью измерений типа "родитель–потомок".

Рекурсивные таблицы содержат два ключевых столбца, например идентификаторы сотрудника и его начальника. Для построения организационной диаграммы в этом случае нужно начать с президента компании и последовательно исследовать его непосредственных подчиненных, раскрывая, в свою очередь, списки их подчиненных и т.д. Часто такое отношение реализуется с помощью ключей идентификатора сотрудника (первичный ключ) и идентификатора его начальника (внешний ключ). Когда такое отношение существует в таблице источника, мастер измерений предложит соответствующее ему отношение типа "родитель–потомок". В примере

с таблицей сотрудников атрибут идентификатора сотрудника будет конфигурироваться с параметром Usage, для которого установлено значение `Key`, в то время как в атрибуте идентификатора начальника для этого же параметра будет установлено значение `Parent`. Среди других важных параметров конфигурации измерения типа "родитель–потомок" можно упомянуть следующие.

- **RootMemberIf**. Этот параметр атрибута родителя указывает службе анализа, как идентифицировать верхний уровень иерархии. Среди допустимых значений параметра `ParentIsBlank` (в поле родителя содержится пустое значение или нуль), `ParentIsSelf` (значения полей родителя и потомка совпадают) и `ParentIsMissing` (строка, на которую ссылается поле родителя, не найдена). По умолчанию установлено значение `ParentIsBlankSelfOrMissing` (допускающее все три вышеупомянутых варианта).
- **OrderBy**. Способ организации отображения иерархии.
- **NamingTemplate**. По умолчанию каждый уровень иерархии использует простую систему именования: `Level 01`, `Level 02` и т.д. Изменить схему именования можно, щелкнув на эллипсе в параметре NamingTemplate атрибута родителя и выбрав в открывшемся диалоговом окне шаблон. Уровням можно присвоить собственные имена, а также использовать для них схемы с нумерацией (в последнем случае символом звездочки отмечается местоположение номера в имени).
- **MembersWithData**. Этот параметр атрибута родителя управляет отображением не листовых членов вместе с данными. По умолчанию эти значения родителей отображаются на собственном уровне, а затем повторяются на листовом (для параметра установлено значение `NonLeafDataVisible`). Например, если просматривать куб с использованием измерения сотрудников типа "родитель–потомок", с отображением объемов продаж торговыми представителями, то имя управляющего отобразится вначале на собственном уровне, а затем будет повторено на уровне сотрудника, чтобы можно было ассоциировать сотрудника с его управляющим. Альтернативное значение, `NonLeafDataHidden`, не повторяет имя родителя и данные, связанные с ним. Это может вносить разнобой в некоторые представления, так как общая сумма не будет соответствовать сумме всех строк. В частности, в примере с применением измерения сотрудников к кубу объемов продаж общие итоги будут отличаться на объем продаж, совершенных управляющими самостоятельно.
- **MembersWithDataCaption**. Если для параметра MembersWithData установлено значение `NonLeafDataVisible`, то этот параметр атрибута родителя указывает службе анализа, как следует именовать генерируемые листовые члены. По умолчанию этот параметр пуст; в таком случае генерируемые листовые члены будут иметь то же имя, что и их родители. Если вы хотите изменить такой режим, введите любую строку с использованием звездочки, которая при отображении замещается именем родителя. Например, значение параметра `* (mgr)` приведет к тому, что имя каждого менеджера будет дополняться строкой `" (mgr)"`.
- **UnaryOperatorColumn**. Эта дополнительная функция сведения часто используется в измерениях расчетов, позволяя значениям, связанным с разными типами счетов, в случае необходимости добавляться к родительским итогам или вычитаться из них. Установленный в атрибуте родителя, этот параметр идентифицирует столбец в таблице источника данных, определяющий, как данное значение участвует в формировании итогов. Ожидается, что данный столбец будет содержать знак "плюс" в элементах, которые следует прибавить, знак "минус" — в вычитаемых элементах и тильду — в игнорируемых. Этот столбец также может содержать символ звездочки для умноже-

ния значения на текущий частичный итог или символ косой черты для деления значения на частичный итог, однако данные операторы дают разные результаты в зависимости от того, какие значения аккумулируются первыми. Для управления порядком операций можно в измерение типа "родитель–потомок" добавить еще один столбец в форме атрибута с типом последовательности. Пустые операторы интерпретируются как операторы суммирования.

После конфигурирования отношения "родитель–потомок" атрибут родителя обеспечивает многоуровневое представление данных измерения. В то же время доступны все остальные атрибуты измерения — они работают в обычном режиме. Приведенный выше список операций подстройки измерения применим и для типа "родитель–потомок", за исключением того, что атрибут родителя предпочтительнее корректировать в самом измерении, а не в представлении источника данных, учитывая его уникальный характер использования.

Тонкая настройка измерений

Когда измерение настроено, становится доступным множество параметров, с помощью которых можно управлять его режимом работы; то же относится и к атрибутам. В этом разделе будут описаны как широко распространенные, так и менее очевидные настройки.

Уровень иерархии (All) и члены, заданные по умолчанию

Уровень иерархии `(All)`, помещаемый в ее корень по умолчанию, представляет всех членов этой иерархии. Во время запроса этот уровень позволяет включить всех членов иерархии, не перечисляя каждого из них в отдельности. На самом деле любая иерархия, не участвующая в запросе явно, включается в нее неявно, используя уровень `(All)`. Например, запрос, возвращающий объем продаж товаров в разрезе регионов, неявно делит итоги по всем годам, по всем месяцам, по всем заказчикам и т.п.

По умолчанию уровень `(All)` называется `All`, что вполне практично и чего достаточно для большинства приложений. Однако этому уровню можно присвоить и другое имя, указав его в значении параметра AttributeAllMemberName измерения или в параметре AllMemberName пользовательской иерархии. Например, верхний уровень измерения сотрудников можно именовать как `Everyone`.

Независимо от своего имени, член `(All)` является одновременно членом, заданным по умолчанию, неявно включаемым в любой запрос, для которого данное измерение явно не определено. Член, заданный по умолчанию, можно изменить с помощью параметра измерения `DefaultMember`. Однако к изменению данного параметра следует подходить с осторожностью. В частности, если установить этот параметр для атрибута года в `2005`, то любой запрос, в котором явно не указан год, будет возвращать данные только для этого года. Установка членов по умолчанию может также вызвать конфликт. Например, если параметр DefaultMember для года установить в значение 2005 год, а для месяца — в август 2004 года, запросы, в которых явно не будут указаны месяц и год, вообще не вернут результатов.

Члены по умолчанию обычно устанавливают, когда не все данные, включенные в куб, часто используются. В качестве примера рассмотрим куб, в который включены все продажи, подавляющее количество которых были успешными, и только небольшая часть их сорвалась из-за возникновения каких-либо проблем (например, проблем с кредитной линией заказчика). В данном случае подавляющее большинство запросов будет интересовать данные об объемах успешных продаж. К завершившимся неудачей сделкам могут обратиться только работники, проводящие маркетинговый или другой анализ. Таким образом, установка члена по умолчанию в измерении статуса транзакции продажи в успешный вариант упростит запросы, выполняемые абсолютным большинством пользователей.

Еще одно решение заключается в полном исключении уровня (All) с помощью установки для параметра атрибута IsAggregatable значения false. Когда уровень (All) исключен, следует определить параметр DefaultMember; в противном случае уровень, заданный по умолчанию, будет выбираться случайным образом во время выполнения запроса. К тому же этот атрибут должен участвовать в пользовательских иерархиях только на верхнем уровне.

Группировка членов измерения

Создание групп членов, или *дискретизация*, представляет собой процесс группировки значений многозначного атрибута в своеобразные блоки данных. Этот подход может оказаться особенно полезным при представлении больших массивов последовательных значений, например расстояний коммуникаций или годовых доходов. Эта функция включается для атрибута путем установки для параметра DiscretizationBucketCount количества создаваемых групп и выбора метода дискретизации (параметр DiscretizationMethod) из списка. Автоматический метод дискретизации (значение Automatic) приведет к подобающей группировке в большинстве приложений. Установка автоматического метода отдает выбор алгоритма на откуп службе анализа, которая сама выполняет анализ группируемых данных. Если результаты автоматической группировки вас не устраивают, воспользуйтесь другими методами, за исключением UserDefined, который зарезервирован для локальных кубов. После создания группировка не обязательно будет статичной — при каждой обработке куба будет осуществляться перегруппировка с учетом изменений в исходных данных.

Атрибут, в котором используется группировка, не должен иметь свойств членов. Это значит, что другие атрибуты не могут ссылаться на дискретный атрибут в качестве источника своих агрегаций. Если дискретизируемый атрибут должен участвовать в естественной иерархии (например, является ключом или сильно влияет на производительность), подумайте о создании еще одного атрибута, основанного на том же столбце и обеспечивающего групповое представление.

Уделите внимание конфигурированию исходных столбцов атрибута и их упорядочению, поскольку параметр OrderBy будет определять, в каком порядке данные будут рассматриваться при создании групп и в каком порядке будут отображаться сами эти группы.

Кубы

Куб объединяет в себе все элементы проектирования и преподносит их пользователю, комбинируя источники данных, их представления, измерения, меры и вычисления в едином контейнере. В этом смысле куб является унифицированной моделью измерений (UDM).

Куб может содержать данные (меры) из множества таблиц фактов, организованных в группы мер. Для представления службе анализа данные обычно моделируются с помощью нескольких кубов и баз данных, и это имеет свои преимущества как с точки зрения разработчика, так и с точки зрения конечного пользователя. Пользователи, которым нужен только небольшой срез того, что содержит куб, могут удовлетвориться определением *проекции* (нечто подобное представлению службы анализа), отображая только то, что имеет для них смысл. С точки зрения разработчика, ограничение числа кубов и баз данных позволяет свести к минимуму число связанных измерений и мер.

Использование мастера кубов было рассмотрено в предыдущих разделах как в направлении "сверху вниз" (см. раздел "Знакомство со службой анализа"), используя модель куба для генерации соответствующих реляционных пакетов и пакетов службы интеграции, так и в направлении "снизу вверх" (см. раздел "Создание куба"). После того как структура куба создана, ее можно настроить с помощью конструктора кубов (Cube Designer).

Откройте любой куб в Solution Explorer, чтобы вызвать окно конструктора, показанное на рис. 43.8. Как видите, конструктор кубов содержит несколько вложенных представлений, которые будут описаны далее.

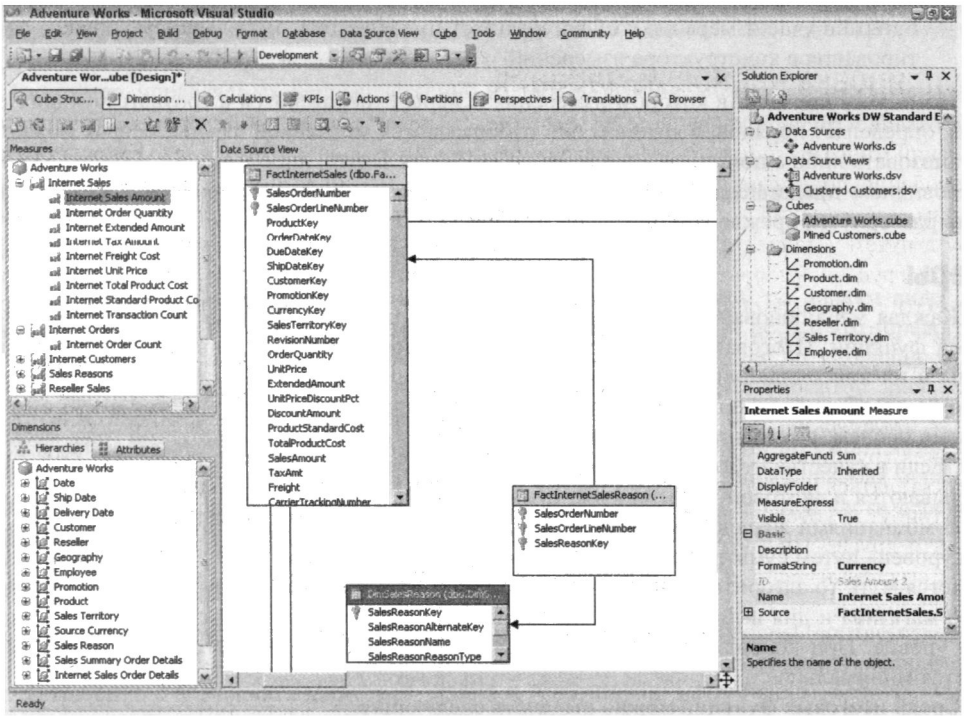


Рис. 43.8. Конструктор кубов

Представление Cube Structure

Представление Cube Structure является главной панелью проектирования. Наряду с вездесущими панелями Solution Explorer и Properties следующие три панели представляют структуру куба.

- Панель Data Source View, находящаяся в центре, отображает фрагмент представления источника данных, на основе которого построен куб. Каждая таблица окрашена в определенный цвет, соответствующий ее функциональности: в желтый окрашены таблицы фактов, в синий — измерений, в белый — те, которые нельзя причислить ни к одной из этих категорий. Обычно на этой панели отображается только часть представления; чтобы отобразить все таблицы, нужно выбрать в контекстном меню пункт Show All Tables или Show Tables. Можно также использовать диаграммы, определенные в представлении источника данных. Для этого нужно выбрать пункт Copy Diagram From контекстного меню. Для переключения между древовидным представлением данных таблиц и отношений и диаграммой можно использовать панель инструментов.
- На панели Measures, расположенной в верхней левой части окна, перечисляются все меры куба, организованные по группам. Как панель инструментов, так и контекстное меню позволяет переключаться между табличным и древовидным отображением мер.

- На панели Dimensions, расположенной в нижней левой части окна, перечислены все измерения, ассоциированные с кубом. Эта панель содержит две вкладки. Во вкладке Hierarchies отображаются все измерения и пользовательские иерархии в этом измерении. Во вкладке Attributes отображаются все измерения и содержащиеся в них атрибуты. Каждое измерение в обеих вкладках имеет ссылку, используемую для его редактирования в конструкторе измерений.

Поскольку порядок, в котором измерения и меры отображаются в соответствующих списках, определяет порядок, в котором они отображаются на главной панели, список можно реорганизовать либо с помощью кнопок Move Up/Move Down, либо методом перетаскивания в древовидном представлении. Подобно конструктору измерений, изменения в кубе должны быть развернуты, прежде чем они будут отображены.

Меры

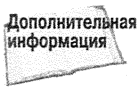
Каждая мера основана на столбце представления источника данных и на некоторой итоговой функции. Итоговая функция определяет порядок обработки данных фактической таблицы и их консолидации. В качестве примера рассмотрим простую таблицу фактов, содержащую столбцы значений дня, магазина и объема продаж. Она считывается в куб с мерой объема продаж, измерением магазинов и временным измерением с атрибутами года, месяца и дня. Если в качестве итоговой функции для меры объема продаж используется `Sum`, то строки считываются в листовой уровень куба, суммируя все строки исходной таблицы с одинаковыми комбинациями значений магазина/дня. После этого они консолидируются на более высокий уровень путем суммирования суточных объемов продаж. Однако если итоговой функцией является `Min`, то алгоритм будет совершенно иной. Из всех строк с дублирующейся комбинацией магазина и дня выбирается наименьший объем продаж, который сохраняется на листовом уровне. При консолидации на более высокий уровень среди листовых значений также ищется минимальное.

Среди итоговых функций можно выделить следующие.

- **Sum**. Сумма значений всех потомков.
- **Min**. Минимальное значение среди всех потомков.
- **Max**. Максимальное значение среди всех потомков.
- **Count**. Количество соответствующих строк таблицы фактов.
- **Distinct Count**. Количество уникальных значений в столбце (например, число уникальных заказчиков).
- **None**. Не выполняется никакой консолидации. Любое значение, не прочитанное непосредственно из таблицы фактов, будет пустым.
- **AverageOfChildren**. Среднее значение потомков, не считая пустых.
- **FirstChild**. Значение первого потомка.
- **FirstNonEmpty**. Значение первого непустого потомка.
- **LastNonEmpty**. Значение последнего непустого потомка.
- **ByAccount**. Консолидация варьируется в зависимости от значений в измерении расчетов. Параметр типа измерения должен быть установлен в значение `Accounts`, и один из атрибутов измерения должен иметь параметр Type, установленный в `AccountType`. Столбец, соответствующий атрибуту с установленным типом `AccountType`, содержит определенные строки, идентифицирующие тип счета и, исходя из этого, метод консолидации, используемый службой анализа.

Лучше всего добавлять новые меры путем щелчка правой кнопки мыши на панели Measures и выбора в контекстном меню пункта New Measure. В открывшемся диалоговом окне определите итоговую функцию и комбинацию таблицы/столбца. Новая мера сразу же будет добавлена в соответствующую группу мер. Группы мер создаются для каждой таблицы фактов плюс для определенных мер подсчета уникальных значений. Эти группы соответствуют разным запросам SQL, выполняемым для извлечения данных куба.

Кроме мер, управляемых непосредственно из таблиц фактов, можно добавить вычисляемые меры либо с помощью соответствующего мастера, либо непосредственно в представлении вычислений.



Более подробно о вычисляемых мерах мы поговорим в главе 45.

Меры могут быть представлены пользователю в сгруппированном по папкам виде. Для этого нужно ввести в параметре DisplayFolder имя папки, в которой должна отображаться мера. Также считается хорошим тоном назначать каждой мере формат по умолчанию. Для этого в параметре FormatString либо выбирается один из стандартных форматов, либо непосредственно вводится пользовательский формат.

Каждый куб может иметь при необходимости меру, используемую по умолчанию. Она используется для возвращения запросам, в которых явно не указана запрашиваемая мера. Для установки меры, используемой по умолчанию, выберите имя куба в верхней части дерева на панели Measure и установите параметр DefailtMeasure, выбрав нужную меру из списка.

Измерения куба

Иерархии и атрибуты в каждом измерении могут быть либо отключены (соответственно с помощью параметров Enabled и AttributeHierarchyEnabled), либо сделаны невидимыми (соответственно с помощью параметров Visible и AttributeHierarchyVisible), если это уместно в контексте конкретного куба (соответствующие примеры сценариев см. в разделе "Видимость и организация"). Доступ к этим параметрам осуществляется путем выделения соответствующего атрибута или иерархии на панели Dimensions и работы с панелью Properties. Эти параметры специально предназначены на роль измерения в кубе и не изменяют структуру самого измерения.

Измерения можно добавить в куб щелчком правой кнопкой мыши на панели Dimensions и выбором в контекстном меню пункта New Dimension. После добавления измерения в куб обратитесь к представлению Dimension Usage, чтобы проверить, правильно ли новое измерение связано со всеми группами мер.

Представление Dimension Usage

Представление Dimension Usage отображает таблицу, в которой показано, как каждое из измерений связано с группами мер. Измерения и группы мер являются заголовками строк и столбцов соответственно; таким образом, каждая ячейка таблицы определяет отношение между соответствующей парой измерения и группы мер. Раскрывающийся список в левом верхнем углу позволяет скрывать строки и столбцы для упрощения просмотра больших представлений.



Информация, которая должна была быть разделена на разные кубы в версии Analysis Services 2000, в настоящей версии может быть запросто включена в один куб. Вместо разделения кубов представление Dimension Usage позволяет по-разному применять измерения куба к разным группам мер.

Конструктор кубов создает связи по умолчанию, основываясь на отношениях в представлении источника данных, что в большинстве случаев вполне адекватно. В то же время некоторые связанные объекты требуют особого подхода, поскольку не управляются представлением источника данных. Щелкните на эллипсе в любой ячейке таблицы, чтобы открыть диалоговое окно Define Relationship, в котором можно выбрать тип отношения. Разные типы отношений требуют разной информации отображения. Об этом мы поговорим в следующих разделах.

Отсутствие отношения

В базах данных, имеющих больше одной таблицы фактов, скорее всего, будут присутствовать измерения, не связанные с некоторыми группами мер. В парах групп мер/измерений такие ячейки таблицы выделяются серым цветом; они не имеют каких-либо аннотаций. Если запускается запрос, определяющий измерение, не связанное с конкретной мерой, то она по умолчанию игнорируется.

Обычные отношения

Обычным называют отношение таблицы фактов непосредственно к таблице измерения, подобно схеме "звезда". В диалоговом окне Define Relationship выберите атрибут гранулярности (Granularity) в качестве атрибута измерения, непосредственно связанного с группой мер, — обычно это ключевой атрибут измерения. В таблице в нижней части окна выберите имена столбцов таблицы фактов, соответствующие ключевым столбцам атрибута гранулярности.

Связь измерения с группой мер с помощью неключевого атрибута также работоспособна, однако в данном случае ее следует рассматривать в контексте естественной иерархии измерения (см. раздел "Взаимосвязи атрибутов"). Естественную иерархию можно представить себе как дерево с ключевым атрибутом в корне. Любой атрибут, находящийся на уровне или выше связанного, будет связан с группой мер и работать, как положено. Все атрибуты, находящиеся ниже связанного или расположенные на другой ветви, не будут иметь связи (см. раздел "Отсутствие отношения").

Фактические отношения

Фактическими называют отношения, управляемые непосредственно в таблице фактов, когда в ней содержатся данные как фактов, так и измерений. Этот тип связи не требует каких-либо дополнительных настроек. Только одно измерение может иметь фактическое отношение с конкретной группой мер. На практике это подразумевает наличие не более одного фактического измерения для каждой таблицы фактов, которое содержит все данные измерения в этой таблице фактов.

Ссылочные отношения

Когда таблицы измерений соединены с таблицей фактов в схеме "снежинка", это измерение может быть реализовано как одно измерение, имеющее обычное отношение с таблицей фактов, или как одно обычное измерение и одно или несколько ссылочных. Ссылочные измерения опосредованно связаны с группой мер через другие измерения. Одно измерение, имеющее обычное отношение, реализовать значительно проще. Однако если ссылочное измерение можно создать и использовать с помощью множества цепочек различных обычных измерений (например, измерение "Географическое положение" может быть использовано в измерениях "Магазин" и "Заказчик"), то этот вариант может быть более эффективным с точки зрения хранения и обработки данных. Ссылочные отношения могут связывать измерения в цепочки любой длины.

Создается ссылочное отношение в диалоговом окне Define Relationship с помощью выбора измерения-посредника, связанного с группой мер. После этого выбираются атрибуты, по которым связываются ссылочное измерение и измерение-посредник. Обычно для повышения производительности устанавливают флажок Materialize.

Отношения "многие ко многим"

Отношения, рассмотренные ранее, имели тип "один ко многим": один магазин может иметь множество транзакций продаж, в одной стране может находиться множество заказчиков. В качестве примера отношения "многие ко многим" можно рассмотреть процесс отслеживания продаж книг по автору и книге. Каждый автор может написать множество книг, а каждая книга может иметь множество авторов. Отношение "многие ко многим" позволяет моделировать такое отношение в службе анализа, однако это требует особой конфигурации, начиная с представления источника данных (рис. 43.9). Отношение "многие ко многим" можно реализовать с помощью использования дополнительной таблицы фактов, выступающей в роли посредника. В этой таблице перечисляется все множество пар, участвующих в отношении между измерениями "многие ко многим" и обычными. В более простых приложениях обычное измерение можно опустить и установить прямое отношение между промежуточной и основной таблицами фактов.

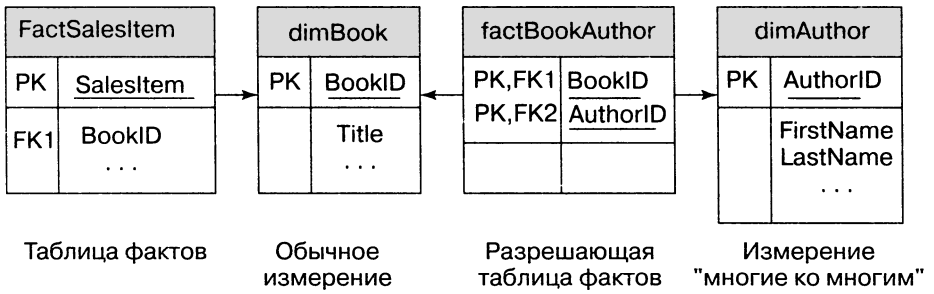


Рис. 43.9. Пример отношения "многие ко многим"

В диалоговом окне Define Relationship обязательным является только параметр имени группы мер, создаваемой в промежуточной таблице фактов для конфигурирования отношения "многие ко многим". Все остальные параметры конфигурации управляются представлением источника данных.

Отношения "многие ко многим" оказывают побочное влияние на запросы, генерируя результирующие наборы данных, которые нельзя консолидировать напрямую. В рассмотренном нами примере книжного магазина предположим, что множество проданных книг имело несколько авторов. В этом случае запрос, отображающий книги в разрезе авторов, будет дублировать некоторые названия. В результате этого суммирование значений списка даст результат, гораздо больший реально проданного числа книг. Часто такой эффект прогнозируют заранее и учитывают в анализе, однако некоторые приложения требуют написания сценариев вычислений для достижения требуемого результата во всех представлениях куба.

Представление KPI

Ключевым индикатором производительности (KPI) называют серверное вычисление, определяющее наиболее значимые метрики. Эти метрики (такие как общая прибыль или охват клиентов) часто используют в инструментальных панелях и прочих инструментах отчетности для распространения в пределах всей организации. Использование KPI для обслуживания такой метрики помогает обеспечить единообразие вычислений и презентаций.

В представлении KPI каждый индикатор состоит из нескольких компонентов.

- Фактическое значение метрики, введенное как выражение MDX, вычисляющее ее.

- Назначение метрики — например, какой объем прибыли предусмотрен в бюджете. Назначение метрики также вводится в виде выражения MDX, вычисляющего соответствующее значение.
- Состояние метрики, сравнивающее фактическое и целевое значения. Статус вводится как выражение MDX, возвращающее значение в диапазоне от `-1` (очень плохо) до `+1` (очень хорошо). В приложениях, представляющих данные KPI, может использоваться графика, что позволяет сделать презентацию единообразной в пространстве всех приложений.
- Тренд метрики, показывающий, в каком направлении она меняется. Подобно состоянию, тренд вводится как выражение MDX, возвращающее значение в диапазоне от `-1` до `+1` с подкреплением графикой.

При определении ключевых индикаторов производительности можно с помощью панели инструментов переключаться между режимами формы и браузера, отображающего результаты. Панель Calculation Tools, находящаяся в левой нижней части окна, отображает метаданные куба и список функций MDX, которые можно использовать при создании выражений MDX методом перетаскивания элементов. Вкладка Templates содержит шаблоны некоторых наиболее распространенных ключевых индикаторов производительности.

Представление Actions

Определение действий позволяет клиенту запускать параметризованные команды, в том числе раскрывающие детали данных, запускающие отчет службы Reporting Services, обрабатывающие URL-адреса или командные строки. Действия могут быть специфичными для отображаемых данных, в том числе для отдельных ячеек и членов измерений, что позволяет выполнить более детальный анализ и даже интеграцию приложения анализа в более крупную среду управления данными.

Представление Partitions

Разделы — это единицы хранения службы анализа. Они хранят данные группы мер. Изначально конструктор кубов создает один раздел MOLAP для каждой группы мер. Как уже говорилось в разделе "Архитектура сервера", MOLAP является самым предпочтительным режимом хранения для подавляющего большинства сценариев. В то же время установка размеров разделов и агрегаций является ключом к эффективной обработке данных и выполнению запросов.

Размеры разделов

Проектирование куба обычно начинается с использования малого, но представительного среза данных. В то же время реальные производственные массивы данных достаточно велики — кубам приходится консолидировать миллиарды и более строк в каждом квартале. Стратегия разделения необходима для управления данными как в реляционных базах, так и в базах данных службы анализа, начиная с оценки объема данных, которые нужно поддерживать для аналитической или оперативной обработки, и определения размеров разделов, которые будут хранить эти данные.

Объем данных, который нужно поддерживать для оперативного доступа, всегда представляет собой компромисс между доступностью исторических данных и стоимостью их хранения. Когда стратегия поддержки данных определена, доступно множество способов разделения данных на управляемые цепочки, однако чаще всего используют подход, основанный на времени. Как правило, в одном разделе хранят данные, накопленные за один отчетный пери-

од, такой как год, месяц или квартал. Размер раздела важен с точки зрения его заполнения данными, так как он определяет интервал времени, необходимый для обработки раздела. Старайтесь удерживать время обработки на уровне не больше нескольких часов. С другой стороны, размер важен и с точки зрения процесса удаления, поскольку определяет объем данных, удаляемых за одну операцию.

Поддержание соответствия размеров разделов и правил удержания между реляционной базой данных и службой анализа является простым и эффективным подходом. Чем больше количество строк, импортируемых ежедневно, тем меньший размер раздела (например, неделя или день) может требоваться для ускорения изначальной обработки. Если система агрегации данных единообразна во всех разделах, служба анализа позволит объединять данные малых разделов, поддерживая общий объем на управляемом уровне.


Проверено

Перед тем как запустить приложение в производство, следует обстоятельно рассмотреть стратегии удержания данных, их обработки и разделения. Когда приложение уже работает, любые изменения могут обойтись очень дорогой ценой, учитывая большой объем накопленных данных.

Создание разделов

Ключом к точному разделению данных является включение любой строки данных в массив только один раз. Так как куб для отчетности использует комбинацию всех разделов, дублирование строк в разных разделах может исказить результат. Самой распространенной ошибкой является создание нового раздела без удаления раздела, созданного конструктором по умолчанию. В этом случае созданный раздел будет содержать одну копию всех данных источника, а раздел, заданный по умолчанию, — вторую, в результате чего куб будет просто удваивать истинные результирующие значения.

Представление Partition состоит из одной сворачиваемой панели для каждой группы мер; в каждой панели содержится таблица, в которой перечислены существующие в настоящее время разделы соответствующей группы мер. Если выделить какой-либо раздел, то его параметры отобразятся на панели Properties.

Процесс добавления раздела начинается со щелчка на ссылке New Partition, чтобы открыть последовательность диалоговых окон мастера разделов (Partition Wizard).

- Specify Source Information. Выберите соответствующую группу мер (по умолчанию выбрана та группа, которая была выделена в момент запуска мастера). Если нужная таблица источника включена в представление источника данных, то она отобразится в списке Available Tables и может быть выбрана. В противном случае выберите соответствующий источник данных в списке Look In и щелкните на кнопке Find Tables, чтобы добавить в список таблицы из выбранного источника. При желании в поле Filter tables можете ввести часть имени таблицы источника, чтобы сократить список отображаемых таблиц.
- Restrict Rows. Если таблица источника содержит только те строки, которые должны быть включены в раздел, пропустите данную страницу. Если таблица источника содержит больше строк, чем должно быть включено в раздел, установите флажок Specify query to restrict rows, и в поле Query будет автоматически сформирован полный запрос `SELECT`, в котором отсутствует только предложение `WHERE`. Введите отсутствующие ограничения и щелкните на кнопке Check, чтобы проверить синтаксис.
- Processing and Storage Locations. Предложенные по умолчанию установки подойдут для большинства ситуаций. Если необходимо, выберите другие параметры, чтобы сбалансировать нагрузку на диски и серверы.

- Completing the Wizard. На этой странице введите имя раздела — чаще всего используют имя группы, за которым следует некоторый признак среза (например, `Internet_Orders_2004`). Если агрегация еще не была определена, определите ее сейчас. Если агрегация была определена для другого раздела, скопируйте ее оттуда, чтобы поддерживать единообразие разделов.

После того как раздел был добавлен, его имя и источник можно отредактировать, щелкнув на соответствующей ячейке в таблице разделов представления Partition.

Проектирование агрегаций

Наилучший компромисс между временем обработки, размерами разделов хранилища и производительностью запросов достигается путем определения только тех агрегаций, которые помогают ответить на самые распространенные запросы к кубу. Основанная на статистике использования оптимизация в службе анализа отслеживает все выполняемые относительно куба запросы, а затем моделирует агрегации, поддерживающие эти запросы. Однако накопление представительной истории выполнения запросов требует достаточно длительного периода эксплуатации куба. Поэтому можете воспользоваться мастером проектирования агрегаций (Aggregation Disign Wizard), который поможет вам создать консолидации, основанные на аналитических решениях. Этот мастер последовательно проведет вас по следующим страницам.

- Specify Storage and Caching Options. На этой странице определяется способ хранения и обновления отдельных разделов. Представленные параметры могут быть установлены на уровне всего куба (т.е. для всех его разделов), что обеспечит их единообразие, а затем модифицированы для особых случаев на этой странице. Более подробно об этом речь пойдет в разделе "Хранилище данных".
- Specify Object Columns. Укажите для каждого раздела точное количество строк, и таблица измерений будет сама управлять тем, как будут вычисляться агрегации. Щелкните на кнопке Count, чтобы выполнить подсчет текущего количества строк. Если текущий источник данных отличается от того, для которого создается проект (т.е. является небольшим тестовым набором данных), то можете ввести числа вручную.
- Set Aggregation Options. На этой странице выполняется фактическое проектирование агрегаций. Как правило, здесь определяют минимально необходимое число агрегаций, оставляя на откуп оптимизатору службы анализа проектирование агрегаций на основе статистики использования. Параметры в левой части страницы указывают конструктору, когда следует остановить создание новых агрегаций, а график в правой части отображает оценку соотношения объема хранилища к выигрышу в производительности.

  В проектировании агрегации не существует строгих правил, однако стоит прислушаться к следующим рекомендациям.

  - Если объем хранилища не является главным ограничением, установите изначальный выигрыш в производительности на уровне 10–20%. В более сложных кубах достаточно сложно оценить благоразумное количество агрегаций (и соответствующее время выполнения). В простых кубах можно допустить множество агрегаций, однако производительность будет все равно настолько высока, что лишние агрегации не принесут ощутимой выгоды.
  - Удерживайте общее количество агрегаций на уровне менее 200 (этот показатель отображается в нижней части окна, непосредственно над индикатором процесса).
  - Найдите на графике соотношения объема хранилища и производительности явный перелом и остановитесь на нем.

- Completing the Wizard. На этой странице вы можете указать мастеру только сохранить проект агрегаций или сохранить и развернуть его.



Лучшие агрегации — те, которые созданы на основе анализа использования. Накапливайте исторические сведения в журнале запросов и периодически используйте его для оптимизации агрегаций каждого из разделов. Протоколирование запросов можно включить в разделе Log/QueryLog окна параметров службы анализа. Установите для параметра `CreateQueryLogTable` значение `true`, в параметре `QueryLogConnectionString` определите строку подключения к журналу, а в параметре `QueryLogTableName` задайте имя журнала.

Проекции

Проекцией называют представление куба, которое скрывает элементы и функции, не соответствующие поставленной задаче. Конечному пользователю проекции представляются как дополнительные кубы. Таким образом, каждая группа пользователей в компании может иметь собственный куб, являющийся специализированным представлением одних и тех же данных.

Проекции создаются путем щелчка правой кнопкой мыши или использования панели инструментов. В результате создается новый столбец. Замените имя, предложенное по умолчанию в верхней части столбца, более подходящим, а затем снимите флажки с тех элементов, которые не нужны для данной проекции. Для проекции также можно выбрать меру по умолчанию — для этого во второй строке таблицы выберите тип объекта `DefaultMeasure`.

Хранилища данных

Стратегия, выбранная для хранилища данных и его компонентов, определяет не только метод хранения куба, но и характер его обработки. Параметры хранилища данных можно установить на трех различных уровнях, при этом настройки родителя определяют значения, принятые по умолчанию для потомков.

- **Уровень куба**. Начните с установки параметров хранилища данных на уровне куба, чтобы все дочерние объекты (измерения, группы мер и разделы) имели значения по умолчанию. Чтобы открыть диалоговое окно Cube Storage Settings, выделите куб в конструкторе кубов, а затем щелкните на эллипсе в его свойстве Proactive Caching.
- **Уровень группы мер**. Эти параметры устанавливают в том случае, когда значения по умолчанию, установленные в кубе, не подходят для данной группы мер. Диалоговое окно параметров Measure Group Storage Settings открывается либо путем щелчка на эллипсе в параметре Proactive Caching группы мер, либо путем выбора ссылки Storage Settings в представлении разделов, когда не выделен ни один конкретный раздел.
- **Уровень объекта (конкретного раздела или измерения)**. Параметры хранилища устанавливаются для одного конкретного объекта. Диалоговое окно параметров Dimension Storage Settings открывается путем щелчка на эллипсе в параметре Proactive Caching измерения в конструкторе измерений, а окно Partition Storage Settings — путем выбора соответствующего раздела и щелчка на ссылке Storage Settings в представлении разделов. Диалоговое окно Parition Storage Settings также является первой страницей мастера проектирования агрегаций.

Все диалоговые окна настройки хранилищ имеют практически один и тот же состав параметров — они отличаются только областью применения настроек. Главная страница диалогового окна содержит ползунок, позволяющий выбрать одну из предварительно сконфигурирован-

ных систем настроек — от наиболее (крайнее левое положение) до наименее оперативной (крайнее правое положение). Перетащите ползунок в некоторое положение и просмотрите параметры, соответствующие данной настройке. Кроме этих встроенных конфигураций, диалоговое окно Storage Options позволяет настроить широкий диапазон различных режимов работы.

По умолчанию выбран режим работы хранилища Pure MOLAP. Эти настройки отлично подходят для традиционных приложений работы с хранилищами данных, так как разделы могут обрабатываться той же процедурой, которая загружает в хранилище большие массивы данных. Этот вариант также хорошо использовать для архивных разделов хранилища, в которых добавления и обновления данных маловероятны. В то же время, если раздел создан на основе часто изменяющихся данных источника (например, непосредственно на таблицах OLTP), упреждающее кэширование позволит автоматически управлять обновлением разделов.

Упреждающее кэширование управляется множеством параметров диалогового окна Storage Options, однако все они предназначены для одной и той же базовой процедуры. Служба анализа уведомляется каждый раз, когда в данных источника выполняются какие-либо изменения, затем выжидается пауза в обновлениях, после чего выполняется перестройка кэша (раздела). Если новое уведомление об обновлении данных поступило до завершения процесса перестройки, то перестройка начинается с начала. Если процесс перестройки занимает время, большее установленного предела, то кэш возвращается в состояние ROLAP до завершения перестройки.

Параметры, управляющие процессом перестройки, доступны во вкладке General диалогового окна Storage Options (предварительно установите в ней флажок Enable Proactive Caching, чтобы разрешить обновления).

- Silence Interval. Этот параметр определяет продолжительность отсутствия уведомлений после последнего обновления, когда запускается процесс перестройки. Приемлемое значение этой настройки зависит от профиля использования таблицы; этот период времени должен быть достаточно длинным, чтобы убедиться в окончательном завершении работы пакета обновлений.
- Silence Override Interval. По истечении заданного в этом параметре интервала времени начинается перестройка, независимо от наличия интервала простоя, установленного в параметре Silence Interval.
- Latency. Этот параметр определяет промежуток времени, прошедший после получения первого уведомления об обновлении, по истечении которого запросы преобразуются в ROLAP. Данный параметр гарантирует, что данные, возвращаемые службой анализа, не отстанут от реального положения вещей в источнике данных более чем на установленное время. Естественно, это может привести к существенной дополнительной нагрузке на сервер и базу данных в период использования запросов ROLAP, в зависимости от объема и сложности запросов, поступивших в данный период.
- Rebuild Interval. Этот параметр задает периодичность обновления кэша в случае, когда уведомления об обновлениях не поступают. По истечении заданного промежутка времени после последней перестройки кэша будет инициирована новая перестройка. Данный параметр можно использовать независимо от наличия изменений в данных (например, не прослушивать изменения, а запускать перестройку через каждые четыре часа) или как резервный вариант обновления кэша, так как невозможно дать гарантии поступления всех уведомлений об обновлениях.
- Bring online immediately. Установка этого флажка позволяет сделать новый куб доступным для запросов ROLAP сразу же после создания, не дожидаясь завершения процесса построения MOLAP. Это позволит повысить доступность за счет увеличения нагрузки на реляционные базы данных и серверы, которым придется обрабатывать запросы ROLAP.

- Enable ROLAP aggregations. Создает представления для поддержки агрегаций ROLAP.
- Apply storage settings to dimension. Этот параметр доступен только на уровне кубов и групп мер. Установка этого флажка позволяет применить те же настройки хранилища ко всем измерениям, связанным с данным кубом или группой мер.

Во вкладке Notifications определяется, как служба анализа будет уведомляться об изменениях, которые вносятся в реляционном источнике данных. Параметры уведомлений могут быть установлены для каждого объекта (раздела или измерения) индивидуально. Эти параметры имеют смысл только тогда, когда режим перестройки при изменениях данных включен (т.е. установлены параметры Silence Interval и Silence Override Interval).

Уведомления SQL Server

Уведомления SQL Server используют события трассировки для сообщения службе анализа, когда в таблице источника данных выполняется обновление или вставка. Так как доставка уведомлений не гарантирована, то этот подход обычно дополняют периодической перестройкой. Это гарантирует, что пропущенные события будут все равно отражены в кубе после периодической перестройки. Включение трассировки событий требует, чтобы служба анализа была подключена к SQL Server под учетной записью с достаточными привилегиями.

Разделы, непосредственно связанные с исходной таблицей и не использующие именованные запросы или запросы связывания в представлении источника данных, не требуют отслеживания таблиц. Для всех остальных разделов следует определить список таблиц, изменения в которых влияют на актуальность их данных. В измерениях также следует создавать списки отслеживаемых таблиц.

Уведомления, инициируемые клиентом

Уведомления, инициируемые клиентом, позволяют приложениям, изменяющим таблицы данных, непосредственно уведомлять службу анализа о таких событиях. При этом приложение отправляет на сервер команду `NotifyTableChange` с указанием измененной таблицы. Во всем остальном обработка таких уведомлений выполняется так же, как и обработка обычных уведомлений SQL Server.

Уведомления опроса

Уведомления опроса работают с источниками данных, отличными от SQL Server, однако они распознают только операции добавления в таблицу новых строк. Если в таблице выполняются также и операции обновления данных, скомбинируйте уведомления опроса с параметром периодической перестройки, чтобы учесть пропущенные обновления.

Механизм опроса работает с помощью запуска запросов, возвращающих некоторые метки из таблицы источника; он отмечает, когда эти метки изменились. Например, если исходная таблица `factSales` использует постоянно увеличивающийся первичный ключ `SalesID`, для опроса можно использовать следующий запрос:

```
SELECT MAX(SalesID) FROM factSales
```

Конфигурирование выполняется путем установки периодичности выполнения опроса и ввода соответствующего запроса. Если объект основан на нескольких таблицах (подобно многотабличному измерению), то можно использовать многотабличные запросы.

Уведомления, основанные на опросах, могут помочь в реализации дифференциальных обновлений. Дифференциальные обновления приобретают важность, когда размер раздела становится

особенно большим, что влечет за собой выход времени обработки и объема используемых ресурсов за приемлемые рамки. Дифференциальная обработка основана на запросе, возвращающем только те данные, которые были добавлены с момента последней обработки раздела. Продолжая приведенный выше пример, в запросе можно использовать символы вопросительного знака для параметров последнего обработанного значения и текущего возвращенного значения. В этом случае для возвращения только последних внесенных данных можно использовать следующий запрос:

```
SELECT * FROM factSales
  WHERE SalesID > COALESCE(?,-1) AND SalesID <= ?
```

Функция COALESCE вставлена на случай обработки пустой таблицы, когда данные ранее не обрабатывались. Для включения дифференциальных обновлений установите флажок Enable Incremental Updates и введите запрос обработки и отслеживаемую таблицу для каждого запроса опроса.

Целостность данных

Функции обеспечения целостности данных в службе анализа решают проблемы несогласованности, которые в противном случае могли бы вызвать неточное представление данных. Служба анализа рассматривает эту несогласованность в двух аспектах.

- **Обработка пустых значений**. Когда в источнике данных встречаются пустые значения, должны ли они интерпретироваться как нули или остаться пустыми значениями?
- **Ошибки ключей**. Когда ключи отсутствуют, дублируются или каким-либо другим образом не обеспечивают отображение между таблицами. Например, как интерпретировать идентификатор клиента, встречающийся в таблице фактов, если не существует соответствующего элемента в измерении клиентов?

Базирование традиционных хранилищ данных на кубах позволяет минимизировать проблемы целостности данных, поскольку практически все они решаются во время загрузки хранилища.



Ключевым достоинством OLAP в целом и модели UDM в частности является согласованная интерпретация данных. Настройки целостности данных и централизованные вычисления являются примерами множества средств, с помощью которых модель UDM унифицирует интерпретацию данных их потребителями. Решайте эти вопросы на этапе проектирования хранилища данных и/или универсальной модели измерений, и тогда вы сможете получить действительно полезный продукт. Рассматривайте этот этап как строительство "объекта данных", наполненного скрытой информацией и открытыми средствами ее использования.

Обработка пустых значений

Режим интерпретации пустых значений зависит от настройки параметра NullProcessing конкретного объекта.

В мерах этот параметр является частью определения источника и может принимать несколько значений.

- **ZeroOrBlank**. В числовых данных пустые значения преобразуются в нули, а в символьных — в пробелы.
- **Automatic**. То же, что и ZeroOrBlank.
- **Error**. Сервер инициирует ошибку и отвергает запись.
- **Preserve**. Сохраняет пустые значения без изменения.

Для обоснованного выбора значения этого параметра лучше всего посмотреть, как должно вычисляться среднее значение в данной мере. Если в усреднении лучше использовать только непустые значения, то лучше установить значение `Preserve`. В противном случае значение `NullOrZero` обеспечит усреднение, рассматривающее пустые значения как нулевые.

В измерениях параметр NullProcessing может принимать и дополнительное значение.

- **UnknownMember**. Пустое значение интерпретируется как неизвестный член.

В измерениях параметр NullProcessing используется в нескольких контекстах.

- Каждый атрибут `NameColumn` измерения, если он определен, содержит параметр NullProcessing как часть определения источника. Этот параметр используется, когда при формировании измерения встречаются пустые имена столбцов.
- Каждая коллекция `NameColumns` измерения содержит параметр NullProcessing для каждого своего столбца. Этот параметр используется, когда при формирование измерения встречаются пустые ключевые столбцы.
- Каждая ячейка в представлении Dimension Usage, связанная с группой мер или измерением обычным отношением, содержит во вкладке Advanced диалогового окна Measure Group Bindings параметр NullProcessing. Этот параметр используется, когда связанный столбец в таблице фактов оказывается пустым.

В атрибутах измерений по умолчанию установлено значение этого параметра `Automatic`, что в большинстве случаев приводит к созданию строк с пробелами. Как правило, лучше устанавливать значение `UnknownMember`, если ожидается наличие пустых значений, или `Error`, в противном случае.

В отношениях измерений используемое по умолчанию значение `Automatic` на практике довольно опасно. Если в ключевом столбце встретилось пустое значение, то оно будет преобразовано в нуль, что может оказаться допустимым ключом в большинстве измерений, в результате чего этому члену будут соответствовать пустые записи. Оптимальное решение предполагает присвоение этому параметру значения `UnknownMember`, если ожидается наличие пустых значений, или `Error`, если не ожидается. В качестве альтернативы можно создать член измерения для нулевого значения и присвоить ему имя "Invalid", что будет соответствовать режиму автоматической обработки пустых значений.

Параметр UnknownMember

Выбор варианта `UnknowMember` для параметра NullProcessing или при обработке ошибок требует конфигурирования неизвестного члена (`Unknown`) в соответствующем измерении. Когда в измерении включено использование неизвестного члена, он должен быть добавлен во все атрибуты этого измерения. Параметр измерения UnknownMember может иметь одно из трех возможных значений.

- **None**. Неизвестный член отключен в измерении, и любая попытка присвоить ему значение завершится ошибкой. Это значение принято по умолчанию.
- **Visible**. Неизвестный член включен и доступен для запросов.
- **Hidden**. Неизвестный член включен, однако напрямую не доступен для запросов. В то же время уровень измерения (`All`) будет содержать вклад неизвестного члена, а функция MDX `UnknownMember` может обеспечить к нему прямой доступ.

По умолчанию неизвестный член имеет имя `Unknown`; его можно изменить, установив значение параметра UnknownMemberName измерения.

Конфигурирование ошибок

При возникновении ошибок целостности данных и некоторых других параметр ErrorConfiguration определяет, как эти ошибки будут обработаны. Изначально для этого параметра установлено значение `(default)`, однако если его изменить в списке на `(custom)`, то станут доступны восемь параметров. Параметр ErrorConfiguration присутствует во множестве объектов, однако обычно его устанавливают только в измерениях и группах мер.

Параметры конфигурирования обработки ошибок описаны ниже.

- **KeyDuplicate**. Это событие генерируется, когда при построении измерения встречаются дубликаты ключей. По умолчанию параметр имеет значение `IgnoreError` (игнорировать ошибку), однако доступны и другие значения: `ReportAndContinue` (создать отчет и продолжить) и `ReportAndStop` (создать отчет и остановить выполнение). Установка параметра в значение `IgnoreError` приведет к вставке в измерение всех значений атрибутов; при этом служба анализа случайным образом выберет то из них, которое будет ассоциировано с ключом. Например, если таблица измерения товаров содержит две строки товаров с идентификатором `73`, один с названием `Apple`, а другой — `Orange`, то и первый и второй появятся как члены атрибута названия товара, но только один из этих атрибутов будет ассоциироваться с данным товаром в транзакциях. И наоборот, если в обеих строках именем будет `Apple`, то в измерении имени товаров будет присутствовать только один атрибут `Apple`, и пользователь куба даже не заметит, что произошло дублирование записей в исходной таблице.
- **KeyErrorAction**. Это событие генерируется, когда возникает ошибка `KeyNotFound`, т.е. когда в ассоциированной таблице не найдено ключевое значение. В группах мер такое случается, когда таблица фактов содержит ключ измерения, который не может быть найден в таблице измерений. Этот параметр может принимать значения `ConvertToUnknown` (преобразовать в неизвестный член; принято по умолчанию) и `DiscardRecord` (отвергнуть запись).
- **KeyErrorLimit**. Количество ошибок ключа, допустимых до инициирования действия, указанного в параметре KeyErrorLimitAction. По умолчанию установлено значение 0; это значит, что действие будет выполнено после первой же ошибки ключа. Установка значения `-1` не ограничивает число допустимых ошибок.
- **KewErrorLimitAction**. Действие, инициируемое при превышении количеством ошибок ключа значения, заданного в параметре KeyErrorLimit. По умолчанию обработка прекращается (значение `StopProcessing`). Можно также использовать значение `StopLogging`, при котором обработка продолжается, но регистрация ошибок в журнале прекращается.
- **KeyErrorLogFile**. Имя файла, в который будут заноситься ошибки ключа.
- **KeyNotFound**. Этот параметр определяет, как ошибки отсутствия ключа взаимодействуют с параметром KeyErrorLimit. По умолчанию используется значение `ReportAndContinue`, которое включает эти ошибки в общий подсчет. Значение `ReportAndStop` позволяет занести ошибку в журнал и немедленно завершить обработку, независимо от настройки параметров KeyErrorLimit и KeyErrorAction. Значение `IgnoreError` может оказаться полезным, когда ожидается наличие множества ошибок отсутствия ключа; это допускает отображение на неизвестный член, проверяя относительно параметра KeyErorLimit только другие ошибки.
- **NullKeyConvertedToUnknown**. Концепция этого параметра идентична KeyNotFound, но все пустые ключи преобразуются в неизвестный член (значением параметра

NullProcessing должно быть UnknownMember) вместо генерации ошибки KeyNotFound. По умолчанию принято значение IgnoreError.

- **NullKeyNotAllowed**. Концепция этого параметра идентична KeyNotFound, однако пустые ключи не допускаются (параметр NullProcessing принимает значение Error) вместо генерации ошибки KeyNotFound. По умолчанию принято значение ReportAndContinue.

Те же параметры можно установить на уровне сервера, чтобы определить значения по умолчанию. Их можно также определить для конкретного процесса обработки и для обеспечения особого подхода к определенным данным.

Пакеты обновлений

Выход пакетов обновлений SP1 и SP2 внес некоторые коррективы в работу службы анализа.

- Оптимизированные запросы к проекциям стали обладать таким же быстродействием, как и запросы к базовому кубу этих проекций.
- Проект AdventureWorks Analysis Services был обновлен. Если ранее вы уже развертывали его в экземпляре SQL Server, то теперь вы должны его повторно развернуть, чтобы увидеть все изменения.
- Версия Microsoft Office 2007 требует установки SQL Server 2005 Analysis Services SP2 для поддержки всех своих функций бизнес-аналитики. Если установить версию службы анализа без пакета обновлений SP2, то все средства Microsoft Office, ориентированные на этот пакет, будут отключены.
- Если пользовательская иерархия не определена как естественная, на экран выводится предупреждение.
- Функции MDX Drilldown имеют новый аргумент, позволяющий задавать прохождение только заданных кортежей.
- В метод MDX CREATE добавлено новое свойство SCOPE_ISOLATION. Это свойство позволяет разрешать вычисления, определенные запросами, в пределах сессий перед вычислением куба, а не после него.
- Добавлена поддержка множества вложенных таблиц.
- Повышена производительность прогнозов, использующих наивный байесовский алгоритм, за счет кэширования часто используемых атрибутов.
- Добавлена ограниченная поддержка обозревателей раскрытия данных моделями локального раскрытия.
- Перераспределение элементов управления обозревателей раскрытия данных теперь зависит от ADOMD.NET.

Резюме

Любая организация явно или неявно использует средства бизнес-аналитики, так как любой компании требуется маркетинговая оценка происходящих на рынке событий. Единственной альтернативой этому является принятие решений на основе интуиции, а не данных. Построение решений бизнес-аналитики обычно начинается с простых запросов к данным OLTP и заканчивается мешаниной чисел. Служба анализа является стратегическим центром компании Microsoft, предназначенным для удовлетворения потребностей компаний в бизнес-

аналитике и превращения беспорядочного нагромождения результатов ручных запросов в целостную систематизированную структуру поддержки принятия решений.

Служба анализа позволяет создавать быстрые, согласованные и подходящие для исследуемой предметной области хранилища данных, удобные как с точки зрения разработчика, так и конечного пользователя. В остальных главах этой части вы можете узнать о загрузке и анализе данных бизнес-аналитики, выполнении запросов к ним, а также о составлении аналитических отчетов.

Раскрытие данных в службе анализа

ГЛАВА

44

В этой главе...

Обзор процесса раскрытия данных

Создание структур и моделей раскрытия данных

Оценка точности модели

Развертывание функциональности раскрытия данных в приложениях

Алгоритмы и представления раскрытия данных

Интеграция раскрытия данных в OLAP

Ответы на многие производственные вопросы можно получить, выполняя запросы к базе данных. Например, можно узнать, какие страницы Web-сайта являются самыми популярными и какие заказчики являются самыми активными. Другие, порой более важные, вопросы требуют более глубоких исследований, например, каковы самые распространенные пути прохождения по Web-сайту и каковы общие характеристики самых активных покупателей. Раскрытие данных позволяет получить ответы на менее очевидные вопросы.

Термин *раскрытие данных* (data mining) часто неверно интерпретируется, и на эту тему существует множество анекдотов. В нашей книге понятие "раскрытие данных" не связано с чем-либо, выполняемым посредством интуиции, простых запросов или статистических методов. Это алгоритмическое извлечение неочевидной информации из больших массивов данных.

Служба анализа реализует алгоритмы извлечения информации, предназначенной для нескольких категорий задач.

- **Сегментация**. Извлечение элементов групп со сходными характеристиками. Например, создание профилей самых активных покупателей или выделение подозрительных значений на странице ввода данных.
- **Классификация**. Разделение элементов на категории. Например, определение, какие категории покупателей откликнулись на рекламную кампанию или какие электронные сообщения с наибольшей вероятностью являются спамом.
- **Ассоциация** (эту задачу также иногда называют *анализом рыночного портфеля*). Определение, какие элементы имеют тенденцию встречаться вместе. Например, какие Web-страницы обычно просматривают на сайте в одном сеансе или что еще обычно приобретают клиенты, купившие конкретный товар.
- **Оценка**. Оценка значения. Например, оценка объема продаж в расчете на одного клиента или оценка долговечности компонента оборудования.

- **Прогнозирование**. Предположение, как будет выглядеть временная последовательность в будущем. Например, предсказание, когда все доступное пространство жесткого диска будет заполнено или какой объем продаж ожидать в следующем квартале.
- **Анализ последовательности**. Определение, какие элементы имеют тенденцию появляться вместе в определенном порядке. Например, каков путь среднестатистического пользователя по сайту или в каком порядке обычно покупают товары.

Эти категории помогут вам понять, каковы сферы применения раскрытия данных. Однако по мере ознакомления с данной тематикой и накопления опыта для вас откроется множество других приложений раскрытия данных.

Процесс раскрытия данных

Традиционный процесс *раскрытия данных* выглядит следующим образом. Модель раскрытия данных "обучается" на тестовых наборах данных, для которых известен результат. После этого настроенная модель используется для получения результатов на основе новых данных по мере их поступления. Использование раскрытия данных требует выполнения нескольких действий, только некоторые из которых непосредственно связаны со службой анализа.

- **Изучение предметной области и данных**. Определение вопросов, ответы на которые необходимо получить, и данных, необходимых для формирования этого ответа. Данные должны быть уместны для решаемой задачи и иметь приемлемую точность. Только в этом случае можно ожидать правдоподобные ответы на поставленные вопросы.
- **Подготовка данных**. В зависимости от конкретной ситуации подготовка данных к раскрытию может быть простой, а может довести до изнеможения. В этом процессе желательно учесть некоторые моменты.
  - Следует избегать строк с низким качеством данных. Понятие качества данных специфично для каждой предметной области, но оно обычно предполагает достаточный объем выборок и отсутствие значений, выходящих за приемлемый диапазон (т.е. описывающих невозможные или крайне маловероятные ситуации).
  - Следует максимально очистить данные, т.е. устранить дублирования, некорректные и несогласованные значения, масштабирование, форматирование и т.п.
  - Служба анализа принимает одну первичную таблицу выбора и, возможно, одну или несколько дочерних вложенных таблиц. Если источник данных распределен по множеству таблиц, требуется выполнить денормализацию с помощью представлений или предварительной обработки.
  - Неравномерные временные ряды могут только выиграть от применения сглаживания.
  - В процессе моделирования могут оказаться полезными управляемые атрибуты. Обычно они представляют собой либо значения. вычисленные на основе других атрибутов (например, `Прибыль=Доход-Затраты`), либо дискретные диапазоны значений (например, "Высокий доход", "Низкий доход" и т.п.).

  Некоторые типы подготовки данных можно выполнить в представлении источника данных службы анализа с помощью именованных запросов и именованных вычислений. Когда такое возможно, настоятельно рекомендуется избегать переработки наборов данных, если изменения становятся необходимыми.

  В заключение необходимо разбить подготовленные данные на два множества: набор данных обучения, предназначенный для настройки модели, и набор данных тестиро-

вания, который будет использован для оценки точности модели. Преобразования Row Sampling и Percentage Sampling (см. главу 42) службы интеграции можно использовать для разделения наборов данных случайным образом. Обычно для тестирования отбирается 10–20% строк.

- **Моделирование**. Модели в службе анализа создаются с помощью предварительного определения структуры раскрытия данных, в которой определены таблицы, участвующие в качестве входа. После этого в структуру добавляются модели раскрытия данных (разные алгоритмы). И наконец, все модели в структуре проходят процесс обучения с использованием учебных данных.
- **Оценка**. Оценка точности и полезности моделей-кандидатов. Этот процесс упрощается за счет использования диаграммы Mining Accuracy Chart службы анализа. Для получения оценки точности модели и сравнения ее с потребностями производства используют тестовый набор данных.
- **Развертывание**. Интеграция запросов прогнозирования в приложения.



Более детальное описание процесса раскрытия данных можно найти на сайте `www.crisp-dm.org`.

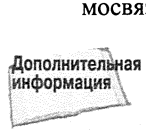
Несмотря на то что описанный процесс типичен для задач раскрытия данных, он не охватывает все возможные ситуации. Иногда исследование набора данных самодостаточно, и обеспечивает лучшее понимание данных и их взаимосвязей. Процесс в данном цикле сводится к повторению действий подготовки, моделирования и оценки. На другом конце спектра находится ситуация, когда для выполнения задачи приложению достаточно создавать и обучать модель, а затем выполнять к ней запрос, например, для выявления значений в наборе данных, не свойственных основному потоку. Независимо от ситуации, понимание типового процесса поможет вам в создании адаптации, подходящей для конкретной задачи.

Моделирование в службе анализа

Для создания структуры раскрытия данных откройте проект службы анализа в утилите Business Intelligence Development Studio (далее BIDS). После развертывания проект создаст базу данных службы анализа на сервере назначения.

Начинается процесс моделирования с указания службе анализа, где размещены данные для обучения и тестирования.

- Определите источники данных, ссылающиеся на данные, которые будут использованы в моделировании.
- Создайте представления источников данных, включающие все обучающие таблицы. Если используются вложенные таблицы, то источник данных должен отражать взаимосвязи между состояниями и вложенными таблицами.



О создании и управлении источниками данных и их представлениями см. в главе 43.

Мастер раскрытия данных

Мастер раскрытия данных (Data Mining Wizard) проводит пользователя через процесс определения новой структуры раскрытия данных и первой модели в этой структуре. Для запуска

мастера щелкните правой кнопкой мыши на узле `Mining Structure` в Solution Explorer и выберите в контекстном меню пункт New Mining Model. Мастер раскрытия данных содержит несколько последовательных страниц.

- Select Definition Method. На этой странице можно выбрать либо реляционные данные, либо куб учебных данных. В данном случае выберите реляционные данные. (Различия между структурами раскрытия, основанными на реляционных данных и на кубах OLAP, описаны в разделе "Интеграция OLAP".)
- Select Data Mining Technique. Выберите алгоритм, который будет использоваться в первой модели раскрытия создаваемой структуры. (Наиболее распространенные алгоритмы описаны в разделе "Алгоритмы".)
- Specify Table Types. Выберите таблицу состояний, содержащую учебные данные, и все связанные вложенные таблицы. Вложенные таблицы всегда связаны с таблицей состояний отношениями "один ко многим". В качестве примера можно привести список заказов, используемый в качестве таблицы состояний, и ассоциированные строки заказов во вложенной таблице.
- Specify the Training Data. Разбейте столбцы на категории, согласно их использованию в структуре раскрытия. Если какой-либо столбец не включен ни в одну из категорий, то он исключается из структуры. Доступные категории приведены ниже.
  - **Key**. Выберите столбцы, которые уникально идентифицируют строки данных обучения. По умолчанию первичный ключ отображается в представлении источника данных со значком ключа.
  - **Predictable**. Идентифицируйте все столбцы модели, которые должны прогнозироваться.
  - **Input**. Пометьте все столбцы, которые будут использованы в прогнозировании, — как правило, в их состав входят и сами прогнозируемые столбцы. Кнопка Suggest может помочь в процессе выбора, если прогнозируемые столбцы были выбраны и перечислены по важности, основанной на выборке учебных данных. Однако старайтесь избегать тех данных, которые с малой вероятностью могут повторяться в эксплуатационных данных. Например, идентификатор клиента, его имя или адрес могут быть достаточно эффективными на этапе обучения модели, однако как только модель будет построена для поиска конкретных идентификаторов или адресов, маловероятно, что новые вводимые клиенты когда-нибудь будут совпадать по данным атрибутам. И наоборот, значения пола и профессии, вероятнее всего, будут часто повторяться в записях о новых клиентах.
- Specify Columns' Content and Data Type. Просмотрите и скорректируйте при необходимости типы данных (`Boolean`, `Date`, `Double`, `Long`, `Text`), проверьте и исправьте типы содержимого. В этом процессе может помочь щелчок на кнопке Detect для вычисления непрерывных числовых данных на основе дискретных. Доступные типы содержимого приведены ниже.
  - **Key**. Содержит значение, которое либо в одиночку, либо в совокупности с другими ключами уникально идентифицирует строку в учебной таблице.
  - **Key Sequence**. Выступает в качестве ключа и определяет порядок строк в таблице. Используется для упорядочения строк в алгоритме последовательной кластеризации.
  - **Key Time**. Выступает в качестве ключа и определяет порядок строк в таблице на основании временной оси. Используется для упорядочения строк в алгоритме временных рядов.

- **Continuous**. Непрерывные числовые данные, часто являющиеся результатом некоторых вычислений или измерений, такие как возраст, цена или высота.
- **Discrete**. Данные, которые можно представить в виде списка значений, такие как модель, профессия или метод доставки.
- **Discretized**. Служба анализа преобразует непрерывные данные в набор дискретных диапазонов (например, диапазоны возраста 1 год–10 лет, 11–20 лет, 21 год–30 лет и т.д.). При выборе этого типа содержимого после завершения работы мастера следует установить еще некоторые параметры столбца. Откройте структуру раскрытия, выберите столбец, а затем установите параметры, определяющие характер дискретизации: `DiscretizationBucketCount` (количество диапазонов дискретизации) и `DiscretizationMethod` (метод дискретизации).
- **Ordered**. Определяет порядок учебных данных, однако без назначения важности значений, используемых для упорядочения. Например, если значения `5` и `10` используются для упорядочения двух строк, то `10` будет следовать за `5`, но это не значит, что `10` вдвое лучше `5`.
- **Cyclical**. Аналогичен типу `Ordered`, однако повторяется в цикле, подобно дням недели или месяцам года.

■ Completing the Wizard. На этой странице задается имя всей структуры раскрытия и первой модели раскрытия в этой структуре. Установите флажок Allow Drill Thru, чтобы активизировать прямую проверку учебных состояний в представлениях раскрытия данных.

После завершения работы мастера будет создана новая структура раскрытия с одной моделью; эта структура будет открыта в конструкторе раскрытия данных Data Mining Designer. Изначальное представление конструктора позволяет добавлять в структуру новые столбцы и удалять из нее ненужные, а также изменять параметры столбцов, такие как тип содержимого или метод дискретизации.

Представление Mining Models

Представление Mining Models конструктора раскрытия данных позволяет конфигурировать различные алгоритмы раскрытия на основе данных, представленных в структуре раскрытия. Для того чтобы добавить новые модели, выполните следующие действия (рис. 44.1).

1. Щелкните правой кнопкой мыши на панели матрицы структуры/модели и выберите в контекстном меню пункт New Mining Model.
2. Присвойте модели имя.
3. Выберите используемый алгоритм.

В зависимости от определения структуры могут оказаться доступными не все алгоритмы. Например, алгоритм последовательной кластеризации `Sequence Clustering` требует наличия столбца с типом `Key Sequence`, а алгоритм временных рядов `Time Series` — столбца с типом `Key Time`. К тому же не все алгоритмы используют столбцы одинаково — например, некоторые алгоритмы игнорируют непрерывные столбцы (в данном случае подумайте об использовании дискретизации в таких столбцах).

Каждая модель раскрытия имеет свойства и параметры алгоритмов. Выделите модель (столбец) для просмотра и изменения свойств, общих для всех алгоритмов, на панели Properties, в том числе Name, Description и AllowDrilThru. Щелкните правой кнопкой мыши на модели, выберите в контекстном меню пункт Set Algorithm Parameters и измените параметры алгоритма, заданные по умолчанию.

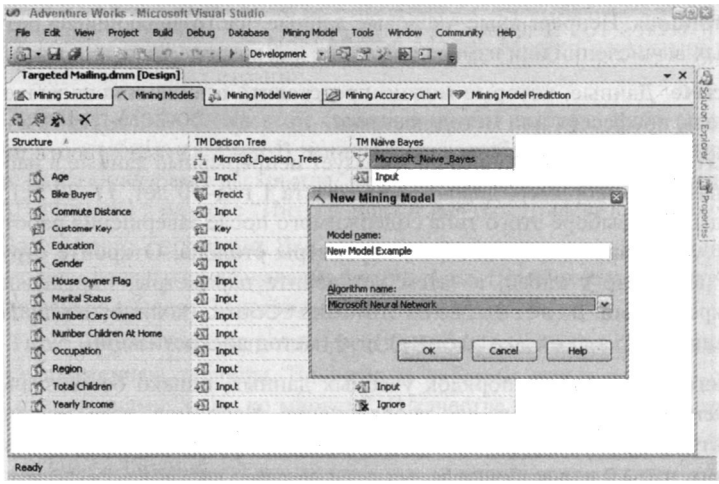


Рис. 44.1. Добавление новой модели в существующую структуру

Когда определения структуры и модели будут созданы, структура должна быть развернута на сервере назначения для обработки и обучения модели. Процесс развертывания модели состоит из двух частей. На первом этапе (построения) определение структуры (или изменений в ней) отправляется на целевой сервер анализа. На панели вывода можно отслеживать ход построения. На втором этапе (обработки) сервер службы анализа выполняет запросы к учебным данным и обучает модель.

Перед первым развертыванием проекта следует определить сервер назначения. Для этого щелкните правой кнопкой мыши на проекте в Solution Explorer, содержащем структуру раскрытия, и выберите в контекстном меню пункт Properties. Перейдите к вкладке Deployment и введите соответствующее имя сервера, одновременно корректируя имя базы данных назначения (по умолчанию имя создаваемой базы данных службы анализа совпадает с именем проекта).

Разверните структуру, выбрав пункт Process или Process Mining Structure and All Models либо в меню Mining Model, либо в контекстном меню. После обработки конструктор переключится в представление Mining Model Viewer, в котором доступно несколько режимов отображения, зависящих от того, какие модели включены в структуру. Специфичные для алгоритма представления помогут понять правила и взаимосвязи, раскрываемые моделью (подробнее об этом — в разделе "Алгоритмы").

Оценка модели

Оценка обученной модели позволяет определить, какая из моделей выполняет прогнозирование более достоверно, и решить, приемлема ли данная точность для рассматриваемой задачи. Представление диаграммы точности раскрытия снабдит вас средствами выполнения такой оценки.

Диаграммы, отображаемые в данном представлении, включаются после предоставления данных во вкладке Column Mapping. Прежде всего убедитесь, что оцениваемая структура раскрытия выделена в левой таблице. Щелкните на кнопке Select Case Table в правой таблице и выберите таблицу либо учебных, либо тестовых данных. Объединения между выбранной таблицей и структурой раскрытия будут установлены автоматически, если имена столбцов совпадают; в противном случае их отображение придется сконфигурировать вручную методом перетаскивания. Проверьте, все ли неключевые столбцы структуры раскрытия участвуют в объединениях.

После того как источник данных определен, перейдите к вкладке Lift Chart и выберите в списке Chart Type тип диаграммы `Lift Chart` (рис. 44.2). Так как данные источника (как

учебного, так и тестового) содержат прогнозируемые столбцы, линейный график позволит сравнить прогноз каждой из моделей с фактическим выходом. График отображается относительно осей `%Correct` и `%Population`, поэтому, когда проверяется 50% популяции, совершенная модель должна корректно прогнозировать ровно 50% данных. На график автоматически добавляются две вспомогательные линии: `Ideal Model`, соответствующая наилучшим из возможных показателей, и `Random Guess`, показывающая, как часто случайно выбранные значения оказываются корректными.

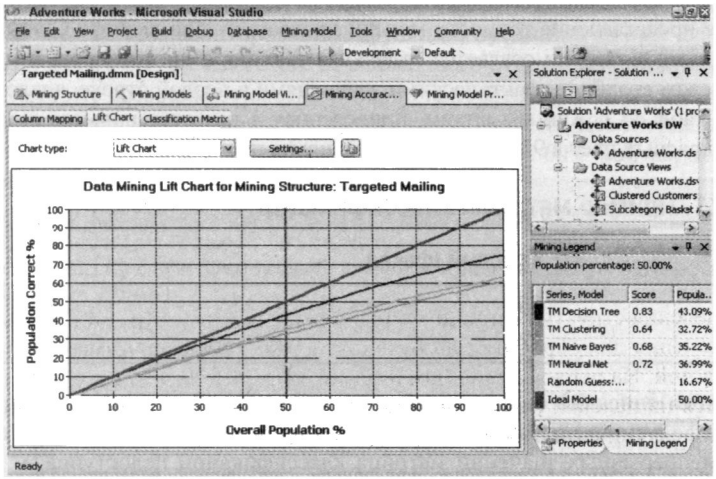


Рис. 44.2. Вкладка Lift Chart

Полезно вначале посмотреть на график, создаваемый учебными данными, и только затем использовать тестовые данные. Хорошо спроектированная модель с адекватными данными будет формировать линии, близкие к идеальной модели, с относительно единообразными показателями для обоих наборов данных. Среди наиболее распространенных проблем можно выделить следующие.

- Модели, хорошо зарекомендовавшие себя на учебных данных и не столь хорошо на тестовых, были плохо обучены. Возможны следующие варианты.
  - Неслучайное разделение данных на учебные и тестовые. Если использованный метод разделения данных был основан на вероятностных алгоритмах, то снова выполните разделение, чтобы получить другие наборы данных, и повторите процесс обучения.
  - Входные столбцы слишком специфичны для некоторого состояния (идентификаторов, имен и т.п.). Скорректируйте структуру раскрытия для игнорирования элементов, содержащих значения, которые встречаются в учебных наборах данных и не встречаются в тестовых или эксплуатационных наборах данных.
  - В учебном наборе данных содержится слишком мало строк (состояний) для создания точной характеристики популяции. Чтобы получить лучшие результаты, поищите дополнительные источники данных. Если таковые недоступны, лучшие результаты можно получить, ограничив число особых состояний, рассматриваемых алгоритмом (например, можно увеличить значение параметра `MINIMUM_SUPPORT`).
- Если все модели более близки к линии случайной выборки, чем к линии идеальной модели, значит, входные данные плохо коррелируют с прогнозируемыми значениями.

Диаграмма выгоды (profit chart) является расширением *линейного графика* (lift chart) и поможет в вычислении максимальной отдачи от маркетинговой кампании по отношению к затраченным усилиям. Щелкните на кнопке Settings и установите количество проекций (фиксированное и в расчете на стоимость каждого состояния), а также ожидаемый выход от успешно идентифицированного состояния, затем в списке типов диаграмм выберите `Profit Chart`. На результирующем графике отобразится выгода по отношению к задействованному проценту популяции. Таким образом, вы будете иметь представление о том, какую часть популяции включать в попытки, либо максимизируя выгоду, либо находя точку перелома.

Простейшее представление точности модели предлагается во вкладке Classification Matrix, в которой для каждой модели создается отдельная таблица с прогнозированными выходными значениями в левом столбце и фактическими значениями в первой строке, аналогично примеру, показанному в табл. 44.1. Как мы видим, приведенная в качестве примера модель правильно предсказывает красный цвет в 95 случаях и неправильно предсказывает синий в 37 случаях.

Таблица 44.1. Пример матрицы классификации

Прогноз	Красный (факт.)	Синий (факт.)
Красный	95	21
Синий	37	104

В приведенном описании оценки модели, использующей средства службы анализа и утилиты BIDS, мы фокусировали внимание на прогнозировании дискретных значений. При прогнозировании непрерывных значений матрица классификации недоступна, и линейный график предлагает несколько отличное сравнение фактических и предсказанных значений. К тому же некоторые алгоритмы, такие как временны*х* рядов, вообще не поддерживают диаграммы точности раскрытия.

Независимо от состава средств, доступных в среде разработки, важно выполнить оценку обученной модели, используя тестовый набор данных, специально зарезервированный для этой цели. После этого выполняйте коррекцию определений данных и модели до тех пор, пока результаты не удовлетворят требованиям, выдвигаемым производственной задачей.

Развертывание

Существует несколько методов внедрения функций раскрытия данных в приложения.

- Непосредственное создание XMLA, взаимодействующего со службой анализа с помощью протокола SOAP. Это позволяет использовать полную функциональность раскрытия данных ценой углубленного программирования.
- Объектная модель AMO (Analysis Management Objects) реализует среду создания и управления структурами раскрытия и прочими метаданными; однако она не предназначена для запросов прогнозирования.
- Язык DMX (Data Mining Extensions) позволяет реализовать большинство задач создания модели и ее обучения, а также выполнять запросы прогнозирования. Инструкции DMX могут быть отправлены службе анализа с помощью следующих интерфейсов:
  - AMOMD.NET — для управляемых языков (семейства .NET);
  - OLE DB — для программ на C++;
  - ADO — для остальных языков программирования.

DMX представляет собой SQL-подобный язык, адаптированный к задачам и структурам раскрытия данных. С целью выполнения запросов прогнозирования относительно обучаемой

модели в языке предусмотрено специальное объединение прогнозирования (PREDICTION JOIN). Как будет показано в следующем примере, этот тип объединения связывает модель раскрытия и набор данных, которые должны предсказываться. Так как запрос DMX выполняется относительно базы данных службы анализа, на модель [TM Decision Tree] можно сослаться напрямую, в то время как состояния должны быть собраны с помощью выполнения функции OPENQUERY относительно реляционной базы данных. Соответствующие столбцы сравниваются в предложении ON, подобно стандартному реляционному объединению, в то время как предложения WHERE и ORDER BY функционируют, как обычно. В язык DMX также добавлено множество специализированных функций, предназначенных для задач раскрытия данных, таких как Predict и PredictProbability (эти функции продемонстрированы в примере), которые возвращают соответственно наиболее вероятный выход и его вероятность. В целом в приведенном примере возвращается список идентификаторов, имен и вероятностей потенциальных клиентов, которые с более чем 60%-ной вероятностью купят велосипед, с сортировкой по убыванию вероятности:

```
SELECT t.ProspectAlternateKey,t.FirstName,t.LastName,
  Predict([TM Decision Tree].[Bike Buyer]),
  PredictProbability([TM Decision Tree].[Bike Buyer]) as Prob
FROM [TM Decision Tree]
PREDICTION JOIN
  OPENQUERY([Adventure Works DW],
  'SELECT
     ProspectAlternateKey, FirstName, LastName, MaritalStatus,
     Gender, YearlyIncome, TotalChildren, NumberChildrenAtHome,
     Education, Occupation, HouseOwnerFlag, NumberCarsOwned,
     StateProvinceCode FROM dbo.ProspectiveBuyer') AS t
ON
  [TM Decision Tree].[Marital Status] = t.MaritalStatus AND
  [TM Decision Tree].Gender = t.Gender AND
  [TM Decision Tree].[Yearly Income] = t.YearlyIncome AND
  [TM Decision Tree].[Total Children] = t.TotalChildren AND
  [TM Decision Tree].[Number Children At Home]
       =   t.NumberChildrenAtHome AND
  [TM Decision Tree].Education = t.Education AND
  [TM Decision Tree].Occupation = t.Occupation AND
  [TM Decision Tree].[House Owner Flag] = t.HouseOwnerFlag AND
  [TM Decision Tree].[Number Cars Owned] = t.NumberCarsOwned AND
  [TM Decision Tree].Region = t.StateProvinceCode
WHERE PredictProbability([TM Decision Tree].[Bike Buyer]) > 0.60
  AND Predict([TM Decision Tree].[Bike Buyer])=1
ORDER BY PredictProbability([TM Decision Tree].[Bike Buyer]) DESC
```

Еще одной полезной формой объединения прогнозирования является одиночный запрос, в котором данные поставляются непосредственно приложением, а не читаются из реляционной таблицы. Эта форма продемонстрирована в следующем примере. Так как имена в точности совпадают с теми, которые используются в модели раскрытия, в примере использовано выражение NATURAL PREDICTION JOIN, не требующее наличия предложения ON. Этот пример возвращает вероятность того, что заданный потенциальный клиент купит велосипед (т.е. [Byke Byer]=1):

```
SELECT
  PredictProbability([TM Decision Tree].[Bike Buyer],1)
FROM [TM Decision Tree]
NATURAL PREDICTION JOIN
(SELECT 47 AS [Age], '2-5 Miles' AS [Commute Distance],
   'Graduate Degree' AS [Education], 'M' AS [Gender],
   '1' AS [House Owner Flag], 'M' AS [Marital Status],
```

```
2 AS [Number Cars Owned], 0 AS [Number Children At Home],
'Professional' AS [Occupation], 'North America' AS [Region],
0 AS [Total Children], 80000 AS [Yearly Income]) AS t
```

Утилита BIDS окажет вам помощь в создании запросов DMX посредством построителя запросов, содержащегося в представлении Mining Model Prediction. Как в диаграмме точности раскрытия, выделите модель и таблицу, к которым будет выполняться запрос, или щелкните на единственной кнопке панели инструментов, чтобы установить значения. В нижней части таблицы определите столбец SELECT и функции прогнозирования. SQL Server Management Studio также предлагает тип запросов DMX с панелью метаданных, предназначенной для перетаскивания имен столбцов структуры раскрытия и функций прогнозирования.

Среди доступных функций прогнозирования следующие.

- `Predict`. Возвращает ожидаемый выход прогнозируемого столбца.
- `PredictProbability`. Возвращает вероятность ожидаемого выхода или особое состояние, если такое определено.
- `PredictSupport`. Возвращает количество состояний обучения, на которых основан ожидаемый выход, или специальное состояние, если такое определено.
- `PredictHistogram`. Возвращает вложенную таблицу со всеми возможными выходами для данного состояния, перечисляя для каждого вероятность, поддержку и прочую информацию.
- `Cluster`. Возвращает кластер, которому назначено данное состояние (функция специфична для алгоритма кластеризации).
- `ClusterProbability`. Возвращает вероятность того, что состояние принадлежит заданному кластеру (функция специфична для алгоритма кластеризации).
- `PredictSequence`. Прогнозирование следующего значения в последовательности (функция специфична для алгоритма последовательной кластеризации).
- `PredictAssociations`. Прогнозирование ассоциативного членства (функция специфична для алгоритма ассоциаций).
- `PredictTimeSeries`. Прогнозирование будущего значения временного ряда (функция специфична для алгоритма временных рядов).

Алгоритмы

При работе со средствами раскрытия данных полезно знать основы алгоритмов раскрытия и области их применения. В табл. 44.2 приведены основные сведения об использовании различных алгоритмов для категорий задач, перечисленных в начале главы.

Таблица 44.2. Использование основных алгоритмов раскрытия данных

Тип задач	Основные алгоритмы
Сегментация	Кластеризация, последовательная кластеризация
Классификация	Деревья решений, наивный Байесовский, нейронные сети, логистическая регрессия
Ассоциация	Ассоциативные правила, деревья решений
Оценка	Деревья решений, линейная регрессия, логистическая регрессия, нейронные сети
Прогнозирование	Временные ряды

Тип задач	Основные алгоритмы
Последовательный анализ	Последовательная кластеризация

Пусть приведенная таблица служит вам путеводителем, однако учтите, что далеко не все задачи раскрытия данных попадают в один из перечисленных типов — для некоторых из них существуют специализированные алгоритмы. В то же время, имея под рукой такой полезный инструмент, как диаграмма оценки точности, довольно просто определить, какой из алгоритмов дает лучший результат в каждой конкретной задаче.

Алгоритм дерева решений

Алгоритм `Decision Trees` является наиболее точным для множества задач. Он начинает свою работу с создания узла `All`, соответствующего всем учебным состояниям (рис. 44.3). Затем выбирается атрибут, наилучшим образом разбивающий состояния на группы, после чего в каждой из получившихся групп также выполняется поиск атрибута, наилучшим образом выполняющего разбиение подмножеств, и т.д. Целью работы алгоритма является создание листовых узлов с единственным прогнозируемым выходом. Например, если задача сводится к определению покупателей, потенциально мотивированных купить велосипед, листовой уровень будет содержать либо всех потенциальных покупателей велосипедов, либо всех, кто не способен купить велосипед, однако ни в коем случае не их комбинацию.

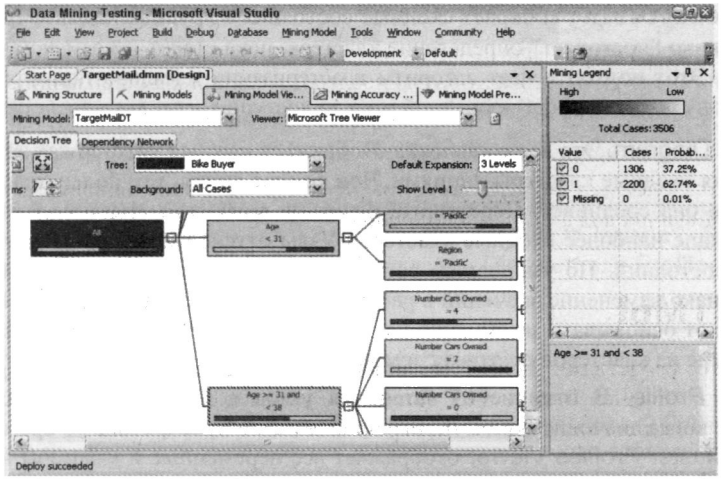


Рис. 44.3. Представление Decision Tree Viewer

Представление Decision Tree Viewer, показанное на рис. 44.3, графически отображает результирующее дерево. Как мы видим, в данном примере первым выбранным атрибутом стал возраст (группы "до 31 года", "от 31 до 38" и т.д.). В группе "до 31 года" решающим оказался атрибут региона, в то время как в группе "от 31 до 38" — количество приобретенных автомобилей. На панели Mining Legend отображаются детали выбранного узла, включая то, как выборки были разбиты переменной прогнозирования (в данном случае 2 200 покупателей и 1 306 воздержавшихся от покупки) как по количеству, так и по вероятности.

Для деревьев решений также доступно представление Dependency Network Viewer, отображающее входные и прогнозируемые столбцы в качестве узлов, которые соединены стрелками, определяющими отношения между предсказывающим и прогнозируемым элементами. Переместите ползунок вниз, и вы увидите наиболее значимые предсказания. Щелкните на любом узле, чтобы выделить все его взаимосвязи.

Линейная регрессия

Алгоритм `Linear Regression` реализован как один из вариантов дерева решений. Он является наилучшим выбором для непрерывных данных, связь между которыми близка к линейной. Результатом регрессии является уравнение вида:

$$Y=B_0+A_1*(X_1+B_1),$$

где `Y` — прогнозируемый столбец; X_1 — входной столбец; A_1 и B_1 — константы, определяемые регрессией. Так как этот алгоритм является частным случаем дерева решений, то для него доступны те же представления раскрытия данных. В то время как представление Tree Viewer отображает только единственный узел `All`, на панели Mining Legend показано само уравнение прогнозирования. Это уравнение можно использовать как в "чистом" виде, так и подставлять в запросы к модели раскрытия данных, направляемые с помощью функции `Prediction`. Представление Dependency Network Viewer графически интерпретирует константы весов, использованные в уравнении.

Кластеризация

Алгоритм `Clustering` собирает сходные состояния в группы, называемые *кластерами*, а затем итеративно "подгоняет" определение кластера, пока станет невозможно добиться большей точности. Этот подход делает алгоритм кластеризации идеально подходящим для задач сегментации популяции. В сформированной модели доступно несколько представлений данных.

- Cluster Diagram. Это представление отображает каждый кластер как затененный узел, соединяя сходные кластеры линиями. Чем темнее линия, тем более подобны кластеры, которые она соединяет. Перетащите ползунок вниз, и вы увидите только линии, соединяющие наиболее сходные кластеры. Узлы оттенены, чтобы представить большее число состояний. По умолчанию состояния подсчитываются на основе всей популяции, однако изменение значений в раскрывающихся списках Shading Variable и State позволяют основывать затемнение на значении конкретной переменной (например, на том, какие из кластеров содержат домовладельцев).
- Cluster Profile. В отличие от затенения узлов в представлении Cluster Diagram Viewer, когда значение одной переменной должно проверяться во времени, представление Cluster Profiles Viewer отображает все переменные и кластеры в одной матрице. Каждая ячейка в этой матрице является графическим представлением распределения переменной в кластере (рис. 44.4). Дискретные переменные отображаются в виде вертикальных столбцов, показывающих, сколько состояний содержит каждое из возможных значений переменной. Непрерывные переменные отображаются в виде диаграмм с ромбиками, где центр каждого ромба соответствует среднему значению, а верхний и нижний его углы соответствуют стандартному отключению. Таким образом, чем выше ромбик, тем менее единообразны значения в кластере. Щелкните на ячейке, и на панели Mining Legend отобразятся все характеристики распределения для комбинации кластера/переменной. Также можете поместить указатель мыши над ячейкой, и в экранной подсказке отобразится та же информация. На рис. 44.4 в экранной под-

сказке отображены характеристики распределения переменной профессии для всей популяции, а на панели Mining Legend показаны характеристики распределения переменной количества детей в кластере 3.

- Cluster Characteristics. В этом представлении отображается список характеристик, формирующих кластер, и вероятностей вхождения в кластер каждой из них.
- Cluster Discrimination. Аналогично предыдущему представлению, но в нем показано, какие характеристики выгодно отличают один кластер от другого. Это представление также позволяет сравнить кластер с собственным компонентом, ясно показывая, что содержится в данном кластере, а что нет.

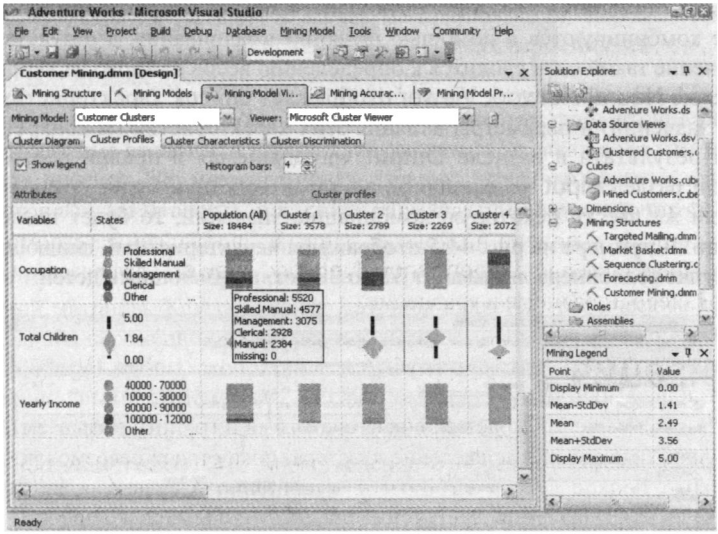


Рис. 44.4. Представление Cluster Profiles Viewer

Для лучшего понимания кластеров, образованных в конкретной модели, имеет смысл переименовать каждый из них в нечто более описательное, нежели "Cluster *n*". Для этого щелкните правой кнопкой мыши на кластере в представлении Diagram или Profiles Viewer и выберите в контекстном меню пункт Rename Cluster.

Последовательная кластеризация

Алгоритм `Sequence Clustering`, как понятно по его названию, собирает состояния в кластеры, однако на основании не атрибутов, а последовательности событий. Например, последовательности посещенных в сессии Web-страниц можно использовать для определения наиболее популярных путей прохождения по Web-сайту.

Природа этого алгоритма требует размещения входных данных во вложенной таблице, при этом родительская строка должна содержать идентификатор сессии или заказа, а вложенная таблица — порядок событий в этой сессии (например, строки заказа). К тому же ключевой столбец вложенной таблицы должен в структуре раскрытия иметь тип содержимого `Key Sequence`.

В ходе обучения модели доступны все четыре представления для кластеров, описанные выше. Дополнительное представление, State Transition Viewer, отображает переходы между элементами (например, Web-страницами) наряду со связанной с ними вероятностью. Пере-

тащите ползунок вниз, и вы увидите только наиболее вероятные переходы. Выделите узел, и будут выделены все возможные переходы от него к вероятным последователям. Короткие стрелки, не соединенные со вторым узлом, отмечают состояние, которое может быть последователем самого себя.

Нейронные сети

Этот известный алгоритм (Neural Network) в общем случае медленнее своих "собратьев", однако зачастую позволяет решать более сложные задачи. Нейронная сеть строится с использованием трех уровней нейронов: входного, скрытого (среднего) и выходного; при этом выход каждого слоя становится входом следующего. Каждый нейрон принимает входные сигналы, которые комбинируются с помощью функций взвешивания, а результат отправляется на выход. Обучение такой сети сводится к определению весов каждого из нейронов.

Представление Neural Network Viewer отображает список характеристик (комбинаций переменных и значений) и параметры влияния этих характеристик на конкретный результат. Выберите два результата в разделе Output, находящемся в правой верхней части окна (рис. 44.5). Если раздел Input не заполнять, то характеристики будут сравниваться для всей популяции; если же ввести некоторую комбинацию значений, то будет исследована только часть популяции. В примере на рис. 44.5 отображены характеристики, влияющие на принятие решения о покупке велосипеда людьми от 32 до 38 лет, не имеющими детей.

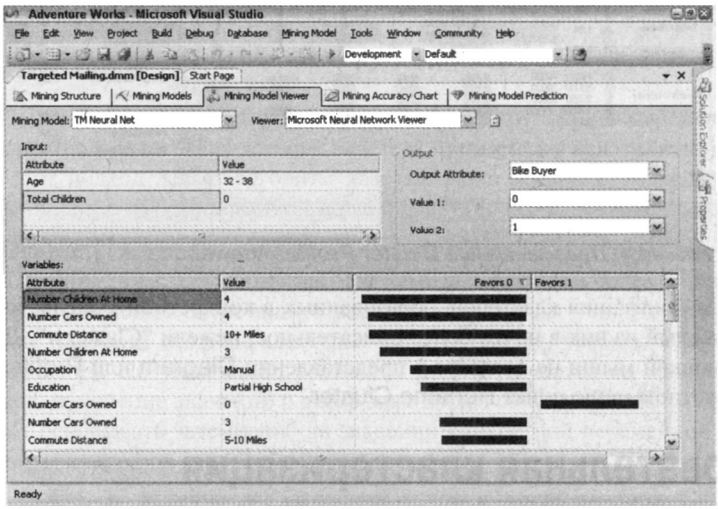


Рис. 44.5. Представление Neural Network Viewer

Логистическая регрессия

Алгоритм `Logistic Regression` является частным случаем алгоритма нейронных сетей, в котором отсутствует скрытый слой нейронов. Несмотря на то что логистическая регрессия может использоваться во множестве задач, она в большей мере подходит для задач оценки, для которых могла бы подойти линейная регрессия; однако ввиду дискретности прогнозируемого значения линейный подход имеет тенденцию предсказывать значения, выходящие за пределы допустимого диапазона (например, предсказание более чем стопроцентной вероятности появления определенной комбинации входных сигналов).

Поскольку данный алгоритм управляется алгоритмом нейронных сетей, ему доступно то же представление (Neural Network Viewer).

Наивный Байесовский алгоритм

Алгоритм `Naive Bayes` является очень быстрым, к тому же его точность вполне адекватна для многих приложений. В то же время он не способен оперировать непрерывными переменными. Источником его "наивности" является предположение о независимости всех входных сигналов. Например, вероятность того, что женатый человек купит велосипед, определяется вычислением частоты появления комбинаций этих двух признаков в учебном наборе данных; при этом остальные столбцы в расчет не принимаются. Вероятностью нового состояния является нормализованное произведение отдельных вероятностей.

Для построенной модели доступно несколько представлений.

- Dependency Network. В этом представлении в качестве узлов отображаются входные и прогнозируемые столбцы, а стрелки указывают на то, на основе которого из них составляется прогноз. Переместите ползунок вниз, чтобы отобразить только самые существенные основы прогноза. Щелкните на узле, чтобы выделить все его отношения.
- Attribute Profile. Это представление функционально идентично представлению Cluster Profile Viewer. В нем отображаются все переменные и прогнозируемые результаты в виде одной матрицы. Каждая ячейка матрицы является графическим представлением распределения переменной в конкретном выходном сигнале. Щелкните на ячейке, и на панели Mining Legend отобразятся все характеристики распределения для данной комбинации "выход/переменная". Если поместить указатель мыши над некоторой ячейкой, то вся эта информация отобразится в экранной подсказке.
- Atribute Characteristics. Это представление отображает список характеристик, связанных с выбранным результатом.
- Atribute Discrimination. Аналогично представлению Characteristic Viewer, однако показывает, какие характеристики отличают один результат от другого.

Ассоциативные правила

Работа алгоритма `Association Rules` заключается в поиске атрибутов, которые имеют свойство появляться вместе в определенных состояниях; при этом в расчет берется достаточная частота таких событий. Такие атрибуты группируются в так называемые *наборы элементов*, которые затем используются для создания правил генерации прогнозов. Несмотря на то что данный алгоритм может быть использован во многих задачах, лучше всего он подходит к задаче формирования рыночного портфеля. В общем случае данные для формирования рыночного портфеля подготавливаются с использованием вложенных таблиц — родительская строка является транзакцией (например, заказом), а вложенные таблицы содержат отдельные ее элементы (например, строки заказа). Три представления позволяют проанализировать обученную модель.

- Itemsets. Это представление отображает список наборов элементов, раскрытых в учебных данных; при этом для каждого набора данных указывается его ассоциированный размер (количество элементов в наборе) и поддержка (количество учебных состояний, в которых это множество появляется). В представлении содержатся некоторые элементы фильтрации, в том числе текстовое поле Filter Itemset, с помощью которой можно выполнить поиск любой введенной строки (например, `"Region=Europe"` позволит отобразить только наборы элементов, содержащие эту строку).

- Rules. По составу и композиции элементов это представление аналогично Itemsets, однако перечисляет не наборы элементов, а правила. Каждое правило имеет форму `A,B⇨C`, а это означает, что состояния, содержащие правила `A` и `B`, вероятнее всего, содержат и `C` (например, люди, купившие макароны и соус, скорее всего, купят и сыр). В каждом из правил указывается его вероятность появления, а также важность (т.е. мера полезности) для выполнения прогнозов.
- Dependency Network. Это представление аналогично одноименному представлению в других алгоритмах; в нем узлы представляют элементы в анализе рыночного портфеля. Следует отметить, что эти узлы склонны влиять на предсказание друг друга (отмечается двунаправленными стрелками). Ползунок позволяет скрывать менее вероятные (но не менее важные) ассоциации. Если выделить некоторый узел, будут выделены также и связанные с ним узлы.

Временные ряды

Алгоритм `Time Series` позволяет предсказать будущие значения ряда последовательных точек данных (например, Web-трафик на следующее полугодие на основании исторических данных). В отличие от ранее описанных алгоритмов прогноз не требует наличия новых состояний, на которых он будет основан, — нужно лишь указать количество итераций, на которое следует удлинить ряд. Входные данные должны содержать *временной ключ* для обеспечения временного атрибута алгоритма. Временные ключи могут определяться с помощью дат, а также числовых столбцов с типом `double` и `long`.

После запуска алгоритма генерируется дерево решений для каждого прогнозируемого ряда. Это дерево решений определяет одну или несколько областей в прогнозировании, а для каждой из них — соответствующее уравнение, которое можно увидеть в представлении Decision Tree Viewer. Например, если некоторый узел имеет метку `Widget.Sales-4<10000`, ее можно интерпретировать следующим образом: "Использовать уравнение этого узла, когда объем продаж данного товара четырьмя этапами ранее был меньше десяти тысяч рублей". При выборе узла на панели Mining Legend отображается связанное с ним уравнение, а если поместить над узлом указатель мыши, то это же уравнение отобразится в экранной подсказке. Обратите внимание на раскрывающийся список Tree в верхней части представления. Он позволяет проверять модель на разных временных рядах. В каждом узле также отображается ромбовидная диаграмма, ширина которой отражает вариации прогнозируемого атрибута в данном узле. Другими словами, чем уже ромбик, тем точнее прогноз.

Второе представление, Charts, используемое в данном алгоритме, выводит на график фактические и прогнозируемые значения выбранного ряда в разрезе времени. Выберите в раскрывающемся списке в верхнем правом углу графика ряд, который следует вывести на диаграмму. Кнопка Abs используется для переключения между абсолютными и относительными (т.е. процентными) значениями. Флажок Show Deviations управляет отображением на диаграмме графика ошибок, показывающего отклонения прогнозируемых значений, а элемент Prediction Steps позволяет управлять количеством отображаемых прогнозов. Поместите указатель мыши над горизонтальным фрагментом, который вас интересует, так, чтобы он стал выделенным, а затем щелкните мышью — данный фрагмент диаграммы будет увеличен. Отмена масштабирования выполняется с помощью соответствующих кнопок масштабирования на панели инструментов.

Так как прогнозирование не основано на состояниях, представление Mining Accuracy Chart для данного алгоритма не работает. Поэтому вам следует вывести значения последующих периодов из учебных данных и сравнить предсказанные значения с результатами, полученными на тестовых данных.

Интеграция OLAP

Раскрытие данных в качестве входных данных может использовать куб, а не реляционные таблицы (см. первую страницу мастера раскрытия данных). В этом случае режим работы, в отличие от режима работы реляционных данных, имеет несколько существенных отличий.

- Несмотря на то что таблицы могут включаться из множества разных источников данных, куб и структура раскрытия, которая на него ссылается, должны быть определены в одном и том же проекте.
- Если таблица определена с помощью одного измерения и связанными с ним группами мер, при необходимости использования дополнительных атрибутов раскрытия данных добавляйте их с помощью вложенных таблиц.
- При выборе ключей структуры раскрытия для реляционной таблицы обычно выбирают первичный ключ. Старайтесь выбирать ключи структуры раскрытия из как можно более высокого уровня данных измерения, обладающего наименьшей гранулярностью. Например, при формировании квартального прогноза следует в качестве ключевого атрибута времени выбирать именно квартал, а не ключевой столбец временного измерения (которым, вероятнее всего, окажется день или час).
- Значения по умолчанию типа данных и содержимого для куба данных имеют тенденцию быть менее достоверными, так что проверяйте и по мере необходимости корректируйте параметры типа.
- Некоторые атрибуты измерения, основанные на датах или числах, могут появиться в интерфейсе раскрытия данных с текстовым типом. Для того чтобы понять причину такого режима работы, нужно заглянуть в теорию. Дело в том, что когда создается представление, требуется определение параметра столбца Key. При необходимости может быть определен и параметр Name, в котором ключевым значениям присваиваются имена, более понятные для конечного пользователя (например, значение `June 2005` будет более понятным, чем `2005-06-01T00:00:00`). В то же время раскрытие данных будет использовать данные с типом Name вместо типа Key, что в результате довольно часто приводит к появлению в структуре раскрытия неожиданных текстовых типов данных. Иногда текстовые данные хорошо справляются с работой, однако в остальных случаях (особенно это касается атрибутов типов `Key Time` и `Key Sequence`) это может сделать невозможным создание структуры раскрытия или привести к ее некорректной работе после создания.

  Разрешение этой проблемы требует либо удаления из атрибута измерения параметра столбца Name, либо добавления в измерение копии столбца, в котором уже будет отсутствовать параметр Name. Если потребуется вторая копия атрибута, то его можно пометить как невидимый, чтобы не запутать конечного пользователя.
- Фрагмент данных куба, который будет использоваться для обучения, определяется с помощью среза куба структуры раскрытия. Скорректируйте этот срез, чтобы исключить состояния, которые не стоит использовать при обучении (например, товары с перебойными поставками и будущие периоды времени). Подумайте также о резервировании части данных для выполнения оценки модели (например, обучение можно выполнять на первых восемнадцати из последних двадцати четырех месяцев, заключительные же шесть месяцев оставить для сравнения прогнозируемых и фактических значений).
- Линейный график не может использоваться для тестовых данных куба. Таким образом, оценка модели требует либо тестирования данных на базе реляционной таблицы, либо использования некоторой стратегии, которая не полагается на представление Mining Accuracy Chart.

Использование куба в качестве источника данных раскрытия может оказаться довольно эффективным, открывая доступ к большим массивам данных в процессе обучения и тестирования и обеспечивая возможности создания измерения или даже целого куба, основанного на обучаемой модели.

Резюме

Раскрытие данных позволяет проанализировать данные намного глубже, чем это можно сделать средствами отчетности, а служба анализа рационализирует весь процесс. Несмотря на то что данные нужно предварительно подготавливать, модели раскрытия скрывают статистические и алгоритмические детали процесса, давая пользователю возможность сфокусироваться исключительно на анализе и интерпретации информации.

Кроме того, обученные модели могут использоваться в приложениях для распределения критичных ресурсов, прогнозирования трендов, идентификации сомнительных данных и множества других целей.

Программирование запросов MDX

ГЛАВА

45

В этой главе...

Основы адресации в кубах

Инструкции MDX SELECT

Распространенные функции MDX

Именованные наборы и вычисляемые члены MDX

Добавление в измерения куба именованных наборов, вычисляемых членов и бизнес-логики

Многомерные выражения MDX являются в службе анализа аналогом того, чем в реляционных базах данных являются запросы SQL. Они реализуют как функции определения (DDL), так и функции запросов (DML), однако базируются на совершенно ином фундаменте. Как вы уже поняли, они возвращают многомерные наборы данных, а не двухмерные, однако, что более важно, запросы MDX не содержат предложений объединения JOIN, так как сам куб содержит явные связи между всеми консолидируемыми данными. Для определения структуры и содержимого результата выполняются манипуляции иерархически организованными данными измерений.

Обучение основам написания запросов MDX у студентов происходит не особенно быстро; в основном это касается тех, кто уже имел опыт работы с другими базами данных. У множества начинающих наблюдается тенденция застревать на базовом уровне запросов. Как правило, причиной этому становится непонимание всего десятка понятий и концепций. Поэтому ряд этих концепций мы выведем в самом начале главы: это кортежи (наборы взаимосвязанных величин), множества (наборы данных), части измерений и некоторые другие.

Чтобы не потерпеть поражение в атаке на MDX, рекомендуется использовать следующий стиль работы.

1. Ознакомьтесь с разделом "Основы запросов SELECT" в начале этой главы, после чего попрактикуйтесь с предложенными примерами, пока не почувствуете себя более уверенно.
2. Внимательно перечитайте раздел "Основы запросов SELECT", пока основные концепции и терминология прочно не закрепятся в вашем сознании. Эти основы позволят вам сознательно знакомиться с дополнительными возможностями в разделе "Расширенные запросы SELECT" и осваивать их на практике.
3. Читая раздел "Сценарии MDX", начните самостоятельно определять множества и вычисления в структуре куба.

Дополнительная информация

Об основах создания кубов, к которым выполняются запросы MDX, см. в главе 43.

Новинка 2005

Несмотря на то что служба анализа входила в состав SQL Server с версии 7.0, для версии 2005 этот продукт был полностью переработан. Многие основные концепции унаследованы из предыдущих версий, однако синтаксис был в значительной мере расширен, а многие ограничения сняты.

Основы запросов SELECT

Подобно SQL, инструкции SELECT в MDX являются средством извлечения данных. Базовой формой этой инструкции является следующая:

```
SELECT { множество1 } ON COLUMNS, { множество2 } ON ROWS
FROM куб
WHERE ( множество3 )
```

Этот запрос вернет простую таблицу, в которой *множество1* и *множество2* определяют заголовки столбцов и строк, *куб* является источником данных, а *множество3* ограничивает фрагменты куба, которые консолидируются в таблице.

Адресация в кубе

Множество представляет собой список, состоящий из одного или нескольких кортежей. Рассмотрим пример, показанный на рис. 45.1. Он ограничен тремя иерархиями измерений (товаром, годом и мерой), что позволяет отобразить множество графически. Запрос MDX консолидирует отдельные блоки в кубе, называемые *ячейками*, в геометрию, определенную запросом. Отдельные ячейки куба адресуются с помощью кортежей.

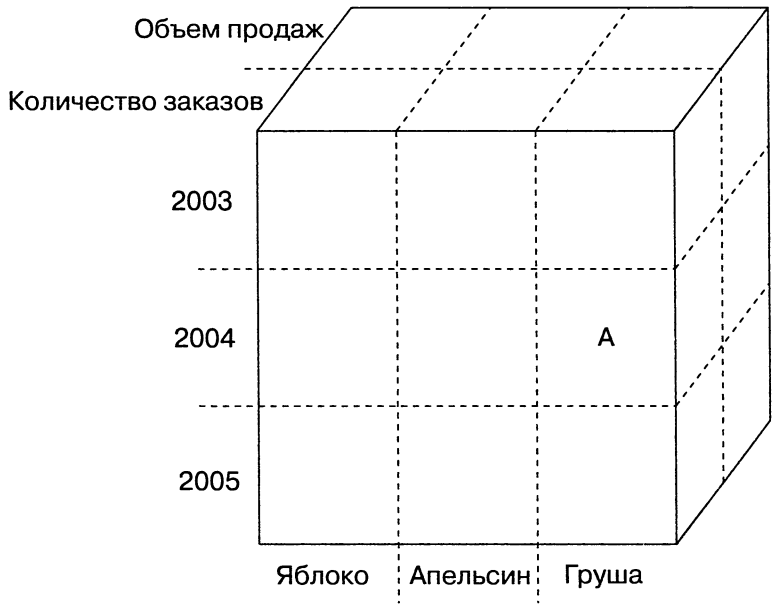


Рис. 45.1. Простой куб с тремя иерархиями измерений

Элементы кортежей каждой из иерархий формируют координаты в кубе, выбирая все ячейки на пересечении этих координат. Например, координаты (груша, 2004, количество заказов) адресуют ячейку, обозначенную на рис. 45.1 символом A. Кортежи могут также адресовать и группы ячеек, используя общий уровень измерения (All). Например, координаты (груша, все годы, количество заказов) указывают на три ячейки: ячейку, помеченную символом A, находящуюся непосредственно над ней (груша, 2003, количество заказов), а также ячейку под ней (груша, 2005, количество заказов). На самом деле, даже когда в кортеже явно не указано измерение, MDX использует член по умолчанию этого измерения (как правило, это уровень All), чтобы восполнить недостающую информацию. Таким образом, адрес (объем продаж) является эквивалентом адреса (все товары, все годы, объем продаж).

Множества создаются на основе кортежей. Таким образом, множество { (груша, 2004, количество заказов) } состоит из одной ячейки, в то время как множество { (яблоко, апельсин) } — из двенадцати, т.е. всех тех, в которых товаром не является груша. Естественно, синтаксис MDX несколько более формализован, чем в приведенном примере, а кубы на практике являются более сложными, однако концепции адресации остаются неизменными.

Структура измерения

Каждое измерение является предметной областью, которая может использоваться для организации результатов запроса MDX. В любом измерении существуют иерархии, в которые включены отдельные понятия данной предметной области. Например, измерение заказчиков может иметь иерархии по городам, странам и почтовым индексам, описывающим местонахождение клиента. Каждая из иерархий, в свою очередь, имеет один или несколько уровней, которые содержат членов или данные измерения. Члены измерения используются для построения множеств, формирующих базис запросов MDX.

Ссылки на измерения

Обращение к одному из компонентов измерения выполняется с помощью перечисления через точку его родословной. Вот несколько примеров:

```
[Customer] -- измерение Customer
[Customer].[Country] -- иерархия Country
[Customer].[Country].[Country] -- уровень Country
[Customer].[Country].[Country].&[Germany] -- член Germany
```

Несмотря на то что технически дозволено не заключать каждый из идентификаторов в квадратные скобки, множество имен в кубах содержат пробелы и специальные символы. По этой причине имеет смысл взять за правило заключать в квадратные скобки все идентификаторы. Символ амперсанда (&) перед именем члена указывает на ссылку по ключу. На каждый член измерения можно ссылаться как по имени, так и по ключу, хотя рекомендуется все же последний вариант. В рассмотренном примере ключ и имя имели одно и то же значение. По этой причине ссылка [Customer].[Country].[Country].[Germany] указывала бы на тот же член измерения. В то же время другие иерархии могут иметь более запутанные ключи. Например, ссылка [Customer].[Customer].[Customer].&[20755] вполне может быть эквивалентом [Customer].[Customer].[Customer].[Mike White].

В дополнение к ссылкам на отдельные члены измерения большинство измерений имеют общий уровень, который ссылается на всех членов измерения. По умолчанию именем этого уровня является [All], однако создатель куба мог назначить ему и другое имя. Например, общий уровень в измерении Customer базы AdventureWorks имеет имя [All Customers]. На общий уровень можно создать ссылку как из измерения, так и из иерархии:

```
[Customer].[All Customers] - Ссылка на общий уровень из измерения
[Customer].[Country].[All Customers] - Ссылка на общий уровень из иерархии
```

Кортежи и простые множества

Как уже говорилось в разделе "Адресация в кубе", кортежи создаются путем перечисления одного члена из каждой иерархии; также для некоторых иерархий члены можно явно не задавать — в этом случае в кортеж будет неявно включен общий уровень этой иерархии. Скобки используются для группировки списка членов кортежа. Например, следующий кортеж ссылается на все ячейки, содержащие объемы Интернет-продаж для заказчиков из Германии в разрезе всех времен, всех территорий и т.д.:

```
([Customer].[Country].[Country].&[Germany],
[Measures].[Internet Sales Amount])
```

Если некоторый простой кортеж определен только с одним членом иерархии, то скобки можно опустить, поэтому всех заказчиков из Германии можно определить просто как

```
[Customer].[Country].[Country].&[Germany]
```

Проще всего построить множество, перечисляя один или несколько кортежей в фигурных скобках. Например, используя простые кортежи без скобок, все множество немецких и французских заказчиков можно определить так:

```
{[Customer].[Country].[Country].&[France],
 [Customer].[Country].[Country].&[Germany]}
```

Базовая инструкция SELECT

Простые множества позволяют создать инструкцию SELECT базового вида. Запрос из следующего примера возвращает объемы Интернет-продаж клиентам из Германии и Франции за 2003 и 2004 календарные годы:

```
SELECT
{[Customer].[Country].[Country].&[France],
 [Customer].[Country].[Country].&[Germany]} ON COLUMNS,
{[Date].[Calendar Year].[Calendar Year].&[2003],
 [Date].[Calendar Year].[Calendar Year].&[2004]} ON ROWS
FROM [Adventure Works]
WHERE ([Measures].[Internet Sales Amount])
```

Будет получен следующий результат:

	France	Germany
CY 2003	$1026324,97	$1058405,73
CY 2004	$922179,04	$1076890,77

В этом примере множества размещены вдоль двух осей, значения которых становятся заголовками строк и столбцов. Предложение WHERE ограничивает результаты запроса только теми ячейками, в которых содержатся объемы Интернет-продаж. Такой частный случай обычно называют *срезом*, так как он ограничивает область определения запроса всего одним сечением куба. Срез можно представить себе как определение вклада в запрос некоторой иерархии, которая не является частью определения какой-либо оси.

Для оси может быть определено любое количество уровней заголовков. Для этого в каждый кортеж, участвующий в построении оси, нужно включить больше одной иерархии, не использующей значения по умолчанию. В следующем примере создаются два заголовка строк путем перечисления элементов двух иерархий в каждой строке кортежа: товарной линии и типа первопричины продаж.

```
SELECT
{[Customer].[Country].[Country].&[France],
 [Customer].[Country].[Country].&[Germany]} ON COLUMNS,
{([Product].[Product Line].[Product Line].&[S],
 [Sales Reason].[Sales Reason Type].[Sales Reason   Type].&[Marketing]),
([Product].[Product Line].[Product Line].&[S],
 [Sales Reason].[Sales Reason Type].[Sales Reason Type].&[Promotion]),
([Product].[Product Line].[Product Line].&[M],
 [Sales Reason].[Sales Reason Type].[Sales Reason Type].&[Marketing]),
([Product].[Product Line].[Product Line].&[M],
 [Sales Reason].[Sales Reason Type].[Sales Reason Type].&[Promotion])
 } ON ROWS
FROM [Adventure Works]
WHERE ([Measures].[Internet Sales Amount],
       [Date].[Calendar Year].[Calendar Year].&[2004])
```

Будет получен следующий результат:

		France	Germany
Accessory	Marketing	$962,79	$349,90
Accessory	Promotion	$2241,84	$2959,86
Mountain	Marketing	$189,96	$194,97
Mountain	Promotion	$100209,88	$126368,03

Такое отображение иерархий на заголовки обеспечивает управление геометрией результирующего набора данных, однако порождает ограничение на создание множеств вдоль осей — иерархии, определенные для кортежей в множестве, должны быть единообразными во всех элементах множества. В приведенном выше примере не допускается, чтобы некоторые кортежи использовали иерархию `Product Category` вместо `Product Line`. Несоответствия между кортежами обычно вызывают появление пустых заголовков ячеек, которые MDX не знает, как обрабатывать.

Еще одним ограничением на создание множеств для запросов MDX является то, что каждая иерархия может указываться только для одной оси или определения среза. Если иерархия `Calendar Year` (календарный год) явно указана в определении строки, то она не может снова появиться в срезе. Это ограничение применяется в пределах исключительно одной иерархии. Если некоторая другая иерархия также содержит календарный год (например, иерархия `Calendar` в базе AdventureWorks), то она может быть указана для одной оси, в то время как иерархия `Calndar Year` — для другой.

Меры

Меры являются значениями, для представления которых создавался куб. Все они доступны запросам MDX как члены всегда присутствующего измерения `Measures`. Это измерение не имеет иерархий или уровней, поэтому ссылка на любую меру выполняется непосредственно из уровня измерения: `[Measures].[`*имя_меры*`]`. Если в запросе явно не указано ни одной меры, то используется мера, принятая в кубе по умолчанию.

Генерация множеств из функций

Разработка программного кода MDX довольно скоро превратится в утомительное занятие, если каждое множество создавать вручную. Для генерации множеств на основе данных куба можно использовать массу функций (некоторые из самых популярных перечислены ниже).

Для экономии места каждый из создаваемых примеров будет формировать наборы строк, соответствующие следующему запросу:

```
SELECT
{[Measures].[Internet Sales Amount],
 [Measures].[Internet Total Product Cost]} ON COLUMNS,
{ } ON ROWS
FROM [Adventure Works]
```

- `.Members`. Эта функция перечисляет всех членов либо иерархии, либо уровня. При использовании с уровнем будут перечислены все его члены (например, `[Date].[Calendar].[Month].Members` вернет все календарные месяцы). При использовании с иерархиями будут перечислены все члены каждого из уровней (например, `[Date].[Calendar].Members` вернет все годы, семестры, кварталы, месяцы и дни).
- `.Children`. Перечисление всех потомков данного члена (например, `[Date].[Calendar].[Calendar Quarter]&[2002]&[1].Children` вернет все месяцы первого квартала 2002 года).
- `Descendants(`*начало*`[,`*глубина*`[,`*флаг*`]])`. Получение списка всех потомков, их потомков и т.д. некоторого члена или набора членов. В качестве аргумента *начало* укажите члена или набор членов, *глубина* — это либо имя конкретного уровня, либо количество уровней ниже члена *начало*. По умолчанию, если аргумент *глубина* явно определен, перечисляются только наследники данного уровня. Аргумент *флаг* может скорректировать этот режим работы, позволяя отображать уровни ниже и выше заданного, а также самого заданного уровня в любой комбинации. Этот флаг может принимать следующие значения: `SELF`, `AFTER`, `BEFORE`, `BEFORE_AND_AFTER`, `SELF_AND_AFTER`, `SELF_AND_BEFORE`, `SELF_BEFORE_AFTER`. Приведем несколько примеров.
  - `Descendants([Date].[Calendar].[Calendar Year].&[2003])`. Перечисляет год, семестры, кварталы, месяцы и дни в 2003 году.
  - `Descendants([Date].[Calendar].[Calendar Year].&[2003], [Date].[Calendar].[Month])`. Перечисляет месяцы в 2003 году.
  - `Descendants([Date].[Calendar].[Calendar Year].&[2003],3,SELF_AND_AFTER)`. Перечисляет месяцы и дни в 2003 году.
- `LastPeriods(`*n*`, `*член*`)`. Возвращает *n* периодов, заканчивающихся членом *член* (например, `LastPeriods(12,[Date].[Calendar].[Month].&[2004]&[6]` вернет месяцы с июля 2003 года по июнь 2004 года). Если значение *n* отрицательно, будут возвращены будущие периоды, начиная с *член*.
- `TopCount(`*множество*`, `*n*` [,`*числовое_выражение*`])`. Возвращает первые *n* элементов множества, отсортированные по *числовому_выражению* (например, `TopCount([Date].[Calendar].[Month].Members,5,[Measures].[Internet Sales Amount])` вернет первые пять по объему продаж Интернет-месяцев). Стоит ознакомиться также и с аналогичными функциями: `BottomCount`, `TopPercent` и `BottomPercent`.

В отличие от функции `TopCount` и ее "побратимов" большинство функций, работающих с множествами, не предполагают выполнения сортировки — они возвращают члены в том порядке, в котором они хранятся в кубе. Для сортировки можно воспользоваться специальной функцией `Order`. Ее синтаксис следующий: `Order(`*множество*`, sort_by [ , { ASC | DESC | BASC | BDESC } ])`. Укажите множество, которое следует сортировать, и, по желанию, порядок сортировки. Параметры `ASC` и `DESC` указывают на сортировку элементов в пределах уровней иерархии. Например, сортировка месяцев в иерархии `Calendar` базы Adventure-

Works с использованием одного из этих параметров приведет к упорядочению месяцев в пределах квартала (следующий уровень иерархии), но не в пространстве всего множества. Параметры BASC и BDESC позволяют выполнить сортировку без учета границ, устанавливаемых родителями члена.

Сгенерированные множества часто содержат членов, для которых недоступны данные мер. Эти члены могут быть подавлены путем дополнения определения оси спереди ключом NON EMPTY. В следующем примере будут извлечены объемы продаж в разрезе торговых представителей по месяцам 2004 года. Ключ NON EMPTY здесь использован для заголовков столбцов, поскольку для некоторых месяцев 2004 года куб вообще не содержит данных. Также этот ключ может оказаться полезным и для определений строк, поскольку не каждый сотрудник является торговым представителем. В приведенном примере использована функция Order для упорядочения торговых производителей по их объемам продаж в 2004 году. Обратите внимание на то, что sort_by является кортежем, определяющим объемы продаж в 2004 году. Если опустить кортеж [Date].[Calendar].[Calendar Year].&[2004], то сортировка проводилась бы по объемам продаж в пределах всего периода накопления данных.

```
SELECT
NON EMPTY {Descendants([Date].[Calendar].[Calendar Year].&[2004], 3)} ON
COLUMNS,
NON EMPTY {
  Order(
    [Employee].[Employee].Members,
    ([Date].[Calendar].[Calendar Year].&[2004],
     [Measures].[Reseller Sales Amount]),
     BDESC
  )
} ON ROWS
FROM [Adventure Works]
WHERE ([Measures].[Reseller Sales Amount])
```

Результат приведенного запроса будет следующим:

	January 2004	February 2004	...	June 2004
All Employees	$1662547,32	$2700766,80		$3415479,07
Linda C. Mitchell	$117697,41	$497155,98		$282711,04
Jae B. Park	$219443,93	$205602,75		$439784,05
...	...			
Stephen B. Jiang	$70815,36	(null)		$37652,92
Amy E. Alberts	$323,99	$42041,96		(null)
Syed E. Abbas	$3936,02	$1376,99		$4197,11

Все эти сгенерированные множества содержат единственную иерархию. Спрашивается, как сгенерировано множество заголовков? Функция Crossjoin генерирует перекрестное произведение любого количества множеств, в результате создавая одно большое множество с кортежами, составленными из всевозможных комбинаций множеств источника. Например, Crossjoin([Product].[Product Line].[Product Line].Members,[Sales Territory].[Sales Territory Country].[Sales Territory Country].Members) будет содержать два уровня заголовков, перечисляющих товарную линию (Product Line) и страну продаж (Sales Territory Country). В качестве альтернативы для генерации перекрестного произведения между множествами можно использовать оператор перекрестного объединения (*). Заключение списка множеств, разделенных запятыми, в скобки даст

тот же эффект. Однако эта конструкция в некоторых контекстах может выглядеть странно. Например, конструкция

```
([Customer].[Country].[Country].&[Germany],
{[Date].[Calendar Year].[Calendar Year].&[2003],
 [Date].[Calendar Year].[Calendar Year].&[2004]})
```

выглядит как кортеж, однако MDX неявно преобразует первый член (клиенты из Германии) в множество, а затем сгенерирует перекрестное произведение, возвращая множество из двух кортежей.

Использование SQL Server Management Studio

Имена объектов в кубе могут быть достаточно длинными, что усложняет их безошибочный ввод. К счастью, утилита SQL Server Management Studio предлагает удобный графический интерфейс, в котором имена функций MDX и объектов можно вставлять методом перетаскивания. Начнем с открытия нового запроса MDX службы анализа и выберем на панели инструментов соответствующую базу данных службы анализа, а также куб назначения в верхнем левом углу окна запроса. Вкладка Metadata (рис. 45.2) автоматически наполнится всеми мерами, измерениями и прочими объектами данного куба. После этого запросы MDX можно создавать, перетаскивая объекты на панель сценария или переключившись во вкладку Functions, таким же образом перетаскивать определения функций.



Более подробно об утилите SQL Server Management Studio см. в главе 6.

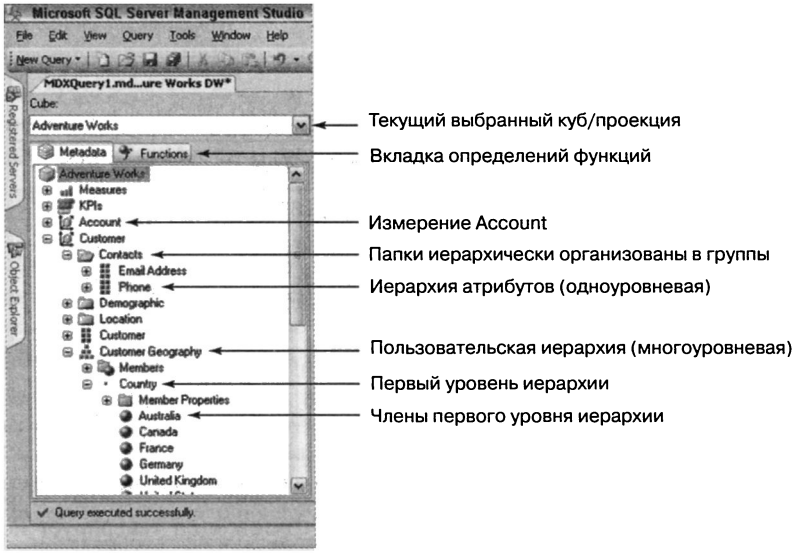


Рис. 45.2. Вкладка Metadata утилиты SQL Server Management Studio

Разработчик куба может при желании сгруппировать иерархии измерения в папки, также показанные на рис. 45.2. Папки являются удобным средством организации длинных списков иерархий и совершенно не оказывают влияния на структуру куба и методику написания запросов MDX.

Расширенные запросы SELECT

Помимо базового процесса генерации таблиц, описанного ранее в этой главе, синтаксис инструкции SELECT, показанный ниже, также включает часто используемые функции.

```
[ WITH <вычисление | множество> [ , <вычисление | множество> ... ] ]
SELECT [ <множество> on 0
    [ , <множество> on 1 ... ] ]
FROM <куб> | <подкуб>
[ WHERE ( <множество> ) ]
```

Инструкция SELECT может возвращать от 0 до 128 осей, где первые пять имеют следующие псевдонимы: ROWS, COLUMNS, PAGES, SECTIONS и CHAPTERS. В качестве альтернативы оси могут именоваться и с указанием порядковых номеров: AXIS(0), AXIS(1) и т.д.



По мере усложнения запросов пропорционально возрастает необходимость их ясности и документирования. Разбивайте длинные запросы на несколько строк и используйте отступы, чтобы визуально организовать вложенные аргументы. Используйте комментарии о назначении определенных элементов запроса, отделяя их двумя тире (--) или двумя косыми чертами (//) в конце строки. Можно также использовать и многострочные комментарии, применяя следующий синтаксис: /* *комментарий* */.

Подкубы

Подкуб определяется в предложении FROM, где другая инструкция SELECT с именем другого куба заключается в скобки. Этот механизм работает практически так же, как и управляемые таблицы в SQL, за исключением того, что управляемые таблицы включают только явно идентифицированные столбцы, а подкуб содержит все иерархии результата, за исключением имеющих ограниченное членство. В следующем примере создается подкуб, содержащий десять лучших товаров и пять месяцев с наиболее высокими объемами продаж (в расчет при этом принимаются все продажи в регионе США). После этого количества заказов суммируются по дням недели и подкатегориям:

```
SELECT
    {[Date].[Day Name].Members} on Columns,
    {[Product].[Subcategory].[Subcategory].Members} ON ROWS
FROM (SELECT
       {TOPCOUNT([Product].[Model Name].[Model Name].Members, 10,
          [Measures].[Internet Sales Amount])} ON COLUMNS,
       {TOPCOUNT([Date].[Calendar].[Month], 5,
          [Measures].[Internet Sales Amount])} ON ROWS
      FROM [Adventure Works]
      WHERE ([Customer].[Country].&[United States]))
WHERE ([Measures].[Internet Order Count])
```

Подкубы могут пригодиться для разбиения сложной логики на управляемые сегменты, а также при создании приложений, в которых требуется либо единообразное представление изменяющихся популяций, либо альтернативные представления фиксированной популяции.

Предложение WITH

Предложение WITH позволяет создавать множества и вычисляемые члены. Несмотря на то что определенная функциональность может быть обеспечена непосредственно с помощью определений осей, считается хорошим стилем использовать множества и вычисления для разбиения логики запроса на более понятные и простые в проектировании сегменты.



Множества и вычисления также могут быть определены в составе куба (подробно об этом — в следующем разделе "Сценарии MDX"). Если какой-либо элемент будет использоваться для чего-то большего, нежели подмножество запроса, создавайте его как часть куба. Это сделает его доступным глобально и централизованно настраиваемым.



В предыдущих версиях MDX требовалось, чтобы все элементы `WITH` заключались в одиночные кавычки (например, `WITH [MySet] AS '`*`определение`*`'`). Теперь эти кавычки не являются обязательными и в приводимых ниже примерах будут опущены.

Множества

Добавьте в предложение `WITH` именованное множество, используя синтаксис *`имя_множества`* `AS` *`определение`*, где *`имя_множества`* — любой допустимый идентификатор, а *`определение`* определяет множество, допустимое для использования в оси или предложении `WHERE`. В следующем примере моделируются три множества для исследования девятимесячных трендов товаров, имеющих более чем 5%-ный рост объемов продаж в 2004 году:

```
WITH
  SET [ProductList] AS
    Filter( [Product].[Product].[Product].Members,
     ([Date].[Calendar Year].&[2004],
      [Measures].[Internet Ratio to All Products])>0.05
     )
  SET [TimeFrame] AS
   LastPeriods(9,[Date].[Calendar].[Month].&[2004]&[6])
    SET [MeasureList] AS {
     [Measures].[Internet Order Count],
     [Measures].[Internet Sales Amount]
    }
SELECT
    {[MeasureList]*[ProductList]} ON COLUMNS,
    {[TimeFrame]} ON ROWS
FROM [Adventure Works]
```

Данный запрос дает следующий результат:

	Internet Order Count	...	Internet Order Count	Internet Sales Amount	...	Internet Sales Amount
	Mountain-200 Silver, 38		Mountain-200 Black, 46	Mountain-200 Silver, 38		Mountain-200 Black, 46
Octover 2003	29	...	29	$67279,71	...	$66554,71
November 2003	28		31	$64959,72		$71144,69
December 2003	32		42	$74239,68		$96389,58
January 2004	28		36	$64959,72		$82619,64
February 2004	36		34	$83519,64		$78029,66
March 2004	35		33	$81199,65		$75734,67
April 2004	45		34	$104399,55		$78029,66
May 2004	48		50	$111359,52		$114749,50
June 2004	62		44	$143893,38		$100979,56

В приведенном примере использовалась функция `Filter`, которая ограничивает множество товаров только теми, которые имели рост объема продаж больше 5%. Эта функция имеет следующий общий синтаксис: `Filter(`*`множество, условие`*`)`.



Пожалуй, самая важная оптимизация запроса выполняется с помощью ограничения размера множеств, причем выполнить это нужно как можно раньше — перед выполняемыми перекрестными объединениями и вычислениями.

Вычисляемые члены

Несмотря на то что синтаксис вычисляемых членов аналогичен синтаксису множеств (*`имя_члена`* `AS` *`определение`*), имя члена должно подходить существующей иерархии, как показано в следующем примере:

```
WITH
  MEMBER [Measures].[GPM After 5% Increase] AS
    ( [Measures].[Internet Sales Amount]*1.05 -
      [Measures].[Internet Total Product Cost] ) /
      [Measures].[Internet Sales Amount], FORMAT_STRING = 'Percent'
  MEMBER [Product].[Subcategory].[Total] AS
     [Product].[Subcategory].[All Products]
SELECT
  {[Measures].[Internet Gross Profit Margin],
   [Measures].[GPM After 5% Increase]} ON 0,
  NON EMPTY{[Product].[Subcategory].[Subcategory].Members,
    [Product].[Subcategory].[Total]} ON 1
FROM [Adventure Works]
WHERE ([Date].[Calendar].[Calendar Year].&[2004])
```

Данный запрос даст следующий результат:

	Internet Gross Profit Margin	GPM after 5% Increase
Bike Racks	62,60%	67,60%
Bike Stands	62,60%	67,60%
Bottles and Cages	62,60%	67,60%
...	...	...
Touring Bikes	37,84%	42,84%
Vests	62,60%	67,60%
Total	41,45%	46,45%

В приведенном примере исследовался текущий рост прибыли и рост прибыли "что, если" по подкатегориям товаров, включая общий объем по всем подкатегориям. Обратите внимание на то, что для согласования с другими иерархиями, использующимися на оси запроса, созданы специальные имена. Необязательный модификатор `FORMAT_STRING` устанавливает формат отображения вычисляемых элементов. Куб источника содержит форматы по умолчанию для каждой меры, в то же время меры, созданные вычислениями, вероятнее всего, потребуют особого форматирования. Итог `[Product].[Subcategory].[Total]`, подобно другим окончательным и промежуточным итогам, может быть основан на родительском члене (в данном случае это уровень `[All]`) для обеспечения соответствующего значения:

```
WITH
  SET [Top20ProductList] AS
    TOPCOUNT([Product].[Product].[Product].Members,
    20,
   ([Date].[Calendar].[Calendar Year].&[2004],
    [Measures].[Internet Order Count]))
  SET [NotTop20ProductList] AS
     Order(
    Filter(
     {[Product].[Product].[Product].Members - [Top20ProductList] },
      NOT IsEmpty([Measures].[Internet Order Count])),
     [Measures].[Internet Order Count],BDESC)
  MEMBER [Measures].[Average Top20ProductList Order Count] AS
    AVG([Top20ProductList],[Measures].[Internet Order Count])
  MEMBER [Measures].[Difference from Top20 Products] AS
    [Measures].[Internet Order Count] -
    [Measures].[Average Top20ProductList Order Count]
  MEMBER [Product].[Product].[Top 20 Products] AS
    AVG([Top20ProductList])
SELECT
    {[Measures].[Internet Order Count],
     [Measures].[Difference from Top20 Products] } ON COLUMNS,
    {[Product].[Product].[Top 20 Products],
     [NotTop20ProductList]} ON ROWS
  FROM [Adventure Works]
  WHERE ([Date].[Calendar].[Month].&[2004]&[6])
```

Будет получен следующий результат:

	Internet Order Count	Difference from Top20 Products
Top20 Products	176	0
Hydration Pack - 70 oz	76	-100
Mountain-200 Silver, 38	62	-114
...	...	...
Touring-3000 Yellow, 54	4	-172
Touring 3000 Yellow, 58	4	-172
Mountain-500 Black, 40	2	-174

В этом примере сравнивается среднее количество проданных за июнь 2004 года товаров в верхней двадцатке самых ходовых, с уровнем продаж остальных товаров, продававшихся в этом же месяце. Этот вымышленный пример помог продемонстрировать множество концепций.

- `Top20ProductList`. Создает список двадцати самых ходовых по количеству заказов товаров за весь 2004 год.
- `NotTop20ProductList`. Создает список всех товаров, не вошедших в двадцатку самых ходовых. Оператор "исключение" (~) используется для удаления двадцати самых продаваемых товаров из общего списка. Этот список фильтруется для исключения также пустых членов, а затем упорядочивается согласно уменьшению количества заказов.
- `Average Top20ProductList Order Count`. Вычисление среднего количества заказов среди двадцати самых ходовых товаров. Аналогичные итоговые функции типа `SUM`, `MIN`, `MAX` и `MEDIAN` используют один и тот же синтаксис: `AVG(множество [, числовое_выражение])`. На практике это вычисление, скорее всего, было бы

реализовано в составе другого вычисления; здесь же оно было включено для демонстрации зависимости одного вычисления от другого.

- `Difference from Top20 Products`. Разность между количеством заказов конкретного товара и средним значением верхней двадцатки.
- `Top 20 Products`. Создается как часть иерархии товаров для получения строки, отображающей среднее значение верхней двадцатки. Так как эта строка должна отображаться для массы мер, *`числовое_выражение`* опущено — оно будет вычислено в контексте отображаемой ячейки.

Параметры измерений

При выполнении запросов к кубу нужно хорошо понимать некоторые вопросы, связанные с параметрами измерений и куба, так как они непосредственно влияют на выполнение запросов.

- `MdxMissingMemberMode`. Этот параметр измерения, для которого установлено значение `true`, позволяет запросу игнорировать некорректные члены, являющиеся частью запроса, не генерируя ошибку. Например, если ось определена как `{ [Product].[Product].[Mountain-100 Silver, 38], [Product].[Product].[Banana] }`, а `Banana` не является корректным названием товара, ошибка не будет сгенерирована. Вместо этого в результате будут перечислены горные велосипеды, а совсем не фрукты. Если установить для этого параметра значение `false`, то для ошибочных имен членов будут генерироваться ошибки. Сценарии MDX (вычисления, описанные в определении куба) всегда вызывают ошибку для отсутствующих членов, независимо от установок данного параметра.
- `IgnoreUnrelatedDimensions`. Если равно `true`, это свойство группы мер заставляет MDX игнорировать измерения, не связанные с опрашиваемой группой мер. Например, измерение для сотрудников в базе AdventureWorks никак не связано с мерами для Интернета, поскольку торговые представители не участвуют в Интернет-продажах. По этой причине в результатах запроса

```
SELECT {[Measures].[Internet Sales Amount]} ON COLUMNS,
       {[Employee].[Employee].[Employee].Members} ON ROWS
  FROM [Adventure Works]
```

  будут перечислены все сотрудники, и для каждого из них будет указан общий объем Интернет-продаж. Это является следствием удовлетворения следующим требованиям: перечислить всех сотрудников и игнорировать несвязанные измерения при подсчете Интернет-продаж. Альтернативное значение этого параметра (`false`) приведет к тому, что все сотрудники также будут перечислены, но их объем продаж будет иметь пустое значение `null`. По умолчанию для этого параметра установлено значение `true`, и это решение является более гибким; в то же время следует быть особо внимательным при написании запросов MDX.
- Если для указанной иерархии был определен член по умолчанию, результаты будут ограничены только связанными с ним значениями, если в запросе явно не указано другое значение этой иерархии. Например, если членом по умолчанию для года является `[Date].[Calendar].[Calendar Year].&[2003]`, то ссылка в запросе на `[Date].[Calendar].[Month].&[2004]&[6]` без указания иерархии календарного кода приведет к отсутствию результатов запроса. Чтобы извлечь данные для июня 2004 года, нужно либо сослаться на уровень `[All]` календарного года, либо, если разработчик куба подавил этот уровень, — на член `[2004]` иерархии года. Определений чле-

нов по умолчанию обычно избегают, однако в некоторых ситуациях они могут оказаться полезными. В таком случае при создании запросов следует учитывать все иерархии, которые имеют членов по умолчанию.

- Автоматическая проверка существования и наличие значения. Помещение множества `{[Date].[Calendar Year].[Calendar Year].Members * [Date].[Calendar].[Month].Members}` на ось запроса приведет к перечислению 2001 года со своими месяцами, 2002 года со своими месяцами и т.д. Спрашивается, почему же в результате перекрестного объединения не было получено настоящее перекрестное произведение множеств? Дело в том, что служба анализа автоматически определяет, какие члены иерархий сосуществуют, и возвращает только приемлемые комбинации. Этот режим работы называют *автоматической проверкой существования*; он функционирует только с иерархиями одного измерения. Параметр `NON EMPTY` используется для того, чтобы далее ограничить множества только теми комбинациями, которые имеют соответствующие значения мер.

Сценарии MDX

Множества и вычисления, подобные описанные в настоящей главе, могут быть созданы непосредственно в кубе. Для этого в утилите BIDS откройте конструктор кубов для интересующего вас куба и переключитесь на вкладку Calculations.



Более подробно об особенностях процесса проектирования куба см. в главе 43.

Куб содержит единственный декларативный сценарий, который описывает все вычисления и множества, несмотря на то, что по умолчанию разработчик представляет сценарий как последовательность форм (рис. 45.3). Даже если в кубе не существует вычисляемых членов или множеств, в нем может существовать обособленная инструкция `CALCULATE`, указывающая кубу заполнять нелистовые ячейки.

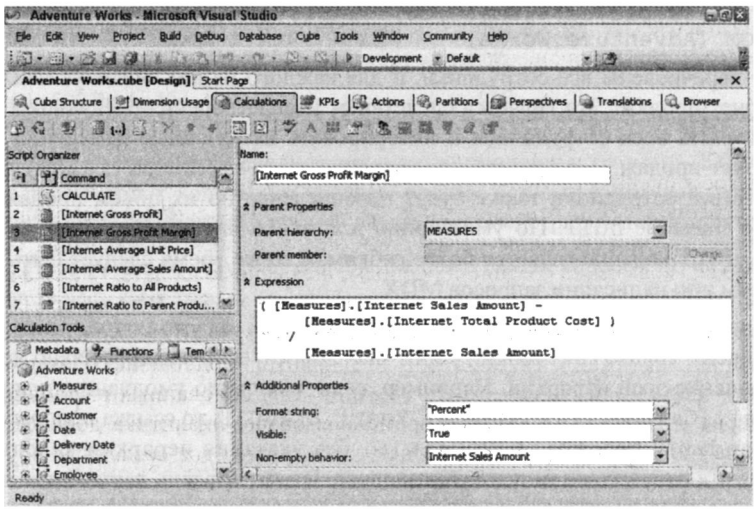


Рис. 45.3. Вкладка Calculations конструктора кубов утилиты BIDS

Вычисляемые члены и именованные множества

Щелкните на существующем вычисляемом члене (см. рис. 45.3) или на кнопке New Calculated Member панели инструментов. Откроется форма, содержащая несколько параметров.

- Name. Имя вычисляемого члена без указания родительской иерархии.
- Parent Hierarchy. Иерархия, к которой должен быть добавлен данный член. Для мер это будет просто `Measures`; для других иерархий используйте встроенные средства навигации для поиска нужной комбинации *измерение*`.`*иерархия*.
- Parent Member. Этот параметр используется только в многоуровневых иерархиях — определите родителя вычисляемого члена. В сущности, этот параметр позволяет определить полный путь, по которому нужно пройти в многоуровневой иерархии для доступа к данному вычисляемому члену.
- Expression. Формула, вычисляющая значение члена; эквивалент выражения, используемого в определении `WITH MEMBER`.
- Format String. Необязательная строка форматирования; обычно определяется для мер.
- Visible. Вычисления обычно делают невидимыми, если они формируют основу для других вычислений, но сами по себе они не имеют для конечного пользователя никакой ценности.
- Non-empty behaviour. Выберите одну или несколько мер для определения способа разрешения параметра `NON EMPTY` в определении оси. Если данный параметр не задан, то вычисляемый член должен вычисляться во всех возможных ячейках, чтобы определить существование значения. Если параметр установлен, то для проверки ячейки на отсутствие значения будут использоваться только перечисленные меры.
- Color and Font Expressions. Отображаемые атрибуты могут быть изменены, в предположении, что клиент использует программное обеспечение, поддерживающее соответствующие режимы отображения, основанные на запросах MDX. Например, значения в бюджете, попадающие в допустимые пределы, могут отображаться зеленым цветом, а выходящие за них — красным.

Множества определяются аналогично вычисляемым членам, однако для них нужно указывать только имя и определяющее выражение.

Добавление бизнес-аналитики

Мастер Business Intelligence Wizard конструктора кубов может добавить в куб вычисления из стандартных шаблонов. Среди этих шаблонов — преобразования валют, основанные на курсах, комбинирование значений на основе диаграммы счетов, а также вычисления, основанные на времени, такие как скользящие средние и периоды, основанные на дате. О назначении и требованиях каждого из шаблонов можно узнать в утилите Books Online. В то же время, ввиду своего широкого распространения, вычисления, основанные на времени, будут описаны в этом разделе.

Использование времени в бизнес-аналитике требует подобающего конфигурирования временного измерения с назначением типов атрибутов, основанных на таблице измерения (а не на серверном измерении). Обычно временная бизнес-аналитика добавляется в куб на последнем этапе цикла его разработки, когда уже доступны все созданные вручную вычисляемые члены. Для запуска мастера откройте интересующий вас куб в конструкторе и щелкните на кнопке Add Business Intelligence панели инструментов. Открывшийся мастер проведет вас по ряду страниц.

- Choose Enchancement. Выберите вариант Define time inteligence.
- Choose Target Hierarchy and Calculations. Вычисления, определенные мастером, будут применены только к одной временной иерархии. Если куб имеет несколько ролей (например, дата заказа и дата доставки) или типов календаря (например, календарный и фискальный), то создание вычислений для разных иерархий потребует отдельных запусков мастера. Как правило, в качестве целевой иерархии выбирают многоуровневую иерархию.

  В верхней части страницы укажите иерархию назначения, а затем выберите вычисления, создаваемые в ней (например, `Twelve Month Moving Average`).
- Define Scope of Calculations. Выберите меры, которые будут консолидироваться (усредняться, суммироваться и т.п.) временными вычислениями.
- Completing the Wizard. Просмотрите изменения, которые мастер выполнит в кубе.

Этот мастер добавляет в куб следующее: именованное вычисление во временной таблице представления источника данных, новую иерархию во временном измерении, содержащую вычисляемые члены, а также сценарий MDX, определяющий эти вычисляемые члены. Результаты вычислений доступны запросам, которые комбинируют иерархию назначения с иерархией, содержащей вычисляемые члены. Результат может иметь следующий вид: значение, если таковое было вычислено; пустое значение, если для выполнения вычисления недостаточно данных; или `"NA"`, если вычисление неприменимо к ячейке (например, вычисление 12-месячного среднего в ячейке, соответствующей одному году).

Резюме

Выражения MDX реализуют практически такой же способ проектирования определений и запросов в базах данных службы анализа, какой доступен в реляционных базах с помощью языка SQL. В отличие от SQL, MDX приспособлен к многомерным данным; он позволяет определять множества вдоль нескольких осей, определяющих геометрию результирующего множества ячеек. Для генерирования, упорядочения и фильтрации множеств доступны встроенные функции. Для создания строительных блоков более сложных и объемных запросов можно применять ключевое слово `WITH`.

MDX можно также использовать как основу для вычислений и создания множеств в определениях кубов. Такие сценарии MDX великолепно подходят для размещения постоянно используемых вычислений и группировок, что делает их доступными для любых запросов.

Комбинация описанных возможностей формирует необычайно мощную и эффективную среду выполнения запросов и анализа данных.

Создание отчетов в службе отчетности

ГЛАВА

46

В этой главе...

Что такое отчет

Процесс создания отчета

Работа с данными

Проектирование композиции отчета

Служба отчетности (Reporting Services) предлагает разработчику мощный инструментарий создания отчетов. Конструктор отчетов (Report Designer) в Visual Studio 2005 реализует удобную среду проектирования для разработчиков, в то время как построителя отчетов (Report Builder) будет вполне достаточно для пользователей, не отягощенных техническими знаниями. В этой главе будет продемонстрировано, как создавать отчеты в Visual Studio 2005.

Создание хороших отчетов требует наличия у разработчика навыков работы с различными, часто несвязанными, технологиями. Отчеты перебрасывают мост между людьми, ответственными за принятие решений, и базой данных, которая невероятными усилиями разработчика была сделана максимально понятной, согласованной, цельной и устойчивой. Учитывая это, можно сформулировать ряд требований к отчету, к достижению которых должен стремиться разработчик.

- Скорость и доступность.
- Точность и своевременность.
- Подобающий уровень детализации — не слишком высокий, но и не слишком низкий.
- Единообразное, целостное и простое для интерпретации форматирование и представление данных.

В этой главе будет исследована "анатомия" отчета, продемонстрированы действия, необходимые для его создания, а также рассмотрены дополнительные средства службы отчетности, позволяющие удовлетворить практически любые потребности разработчиков.

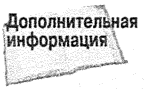
Что такое отчет

Отчет в службе отчетности состоит из источников и наборов данных, параметров и композиции элементов отчета. В настоящем разделе будут отдельно описаны все эти компоненты.

Язык определения отчетов (RDL)

Язык определения отчетов (Report Definition Language, или RDL) представляет собой открытую схему XML, используемую для представления информации об извлечении данных и композиции отчета. Например, схема RDL содержит элементы, определяющие источники данных отчета, наборы данных, а также параметры, с помощью которых данные становятся доступными в отчете. Схема RDL также содержит элементы, управляющие композицией отчета и его форматированием, в частности элементы заголовка и тела отчета, а также его колонтитулов, меток и таблиц.

Определение отчета в службе отчетности является ничем иным, как обычным файлом XML, удовлетворяющим спецификации RDL. Компания Microsoft предлагает два средства создания определений отчетов RDL, позволяющих не заниматься этим вручную: Visual Studio 2005 и Report Builder. В этой главе основное внимание мы сосредоточим на создании отчетов с помощью Visual Studio 2005. Утилита Report Builder является частью пакета диспетчера отчетов (Report Manager). Она обеспечивает конечных пользователей, не являющихся техническими специалистами, возможностью создавать и модифицировать отчеты.



В главе 47 предоставлена дополнительная информация о конфигурировании и развертывании отчетов.

Одной из самых прекрасных сторон службы отчетности является ее способность расширять схему RDL. Так как схема RDL является открытой, можно привлечь расширенные, дополнительные сценарии с помощью добавления в схему атрибутов и элементов. Эти действия можно выполнить как вручную, так и программным путем, используя классы пространства имен `System.XML`.

Создавать, развертывать и выполнять отчеты можно и программным путем. Это снимает какие-либо ограничения с разработки средств создания отчетов, с их генерации на лету и с интеграции отчетов в приложения. Например, с помощью объекта `XMLTextWriter` в программе можно создать определение отчета RDL, затем использовать Web-службу сервера отчетности для его развертывания, а также для обращения к отчету из приложения. Web-служба Report Server также содержит методы для управления практически всеми аспектами сервера отчетности.

Источники данных

Источник данных содержит информацию о подключении к базе данных или файлу, в том числе тип источника, строку подключения и регистрационные реквизиты. Наличие источника данных требуется для извлечения информации для отчета. Источник может быть определен и сохранен в одном отчете или совместно использоваться в проекте несколькими отчетами. Подобно отчету, общие источники данных могут быть развернуты на сервере отчетности. Как общие, так и специфичные для отчета источники данных можно изменить после развертывания на сервере отчетности с помощью утилит Report Manager и SQL Server Management Studio.

Типы источников данных

Несмотря на возможность расширения возможностей службы отчетности с помощью создания дополнительных расширений данных, в ее исходном виде доступны следующие типы источников:

- Microsoft SQL Server;
- Microsoft SQL Server Analysis Services;
- Oracle;
- OLE DB (Object Linking and Embedding for Databases);
- XML (Extensible Markup Language);
- ODBC (Open DataBase Connectivity);
- Report Server Model.

Наиболее распространенные строки подключения к источникам данных

Для подключения к SQL Server 2005 выберите тип источника данных Microsoft SQL Server. Следующая строка подключения является примером соединения с учебной базой данных AdventureWorks в экземпляре SQL Server 2005 с именем SQL2005 на локальном компьютере и с использованием аутентификации Windows:

```
Data Source=localhost\SQL2005;Initial Catalog=AdventureWorks
```

Для подключения к службе анализа Analysis Services 2005 выберите в качестве типа источника данных Microsoft SQL Server Analysis Services. Следующая строка подключения является примером соединения с базой данных AdventureWorksDB в экземпляре SQL2005 локального сервера службы анализа:

```
Data Source=localhost\sql2005;Initial Catalog=AnalysisServicesDB
```

Для подключения к серверу Oracle должен быть установлен клиент Oracle. Выберите в качестве типа источника данных Oracle и используйте следующую строку подключения:

```
Data Source=имя_сервера
```



Тип данных XML является новым в SQL Server 2005. В качестве строки подключения к ресурсу XML, в том числе и Web-службе, файлу XML или приложению, возвращающему данные XML, используется адрес URL. Для подключения к источнику данных XML выберите в качестве типа источника `XML` и введите адрес URL в строке подключения. В следующем примере показана строка подключения к Web-службе сервера отчетности на локальном компьютере:

```
http://localhost/reportserver$sql2005/reportservice2005.asmx
```

В следующей строке показано, как подключиться к файлу XML `StateList.XML`, расположенному на локальном Web-сервере:

```
http://localhost/StateList.xml
```

Процесс создания запроса к данным XML описан в разделе "Работа с источниками данных XML".

Использование выражений в строке подключения

Строки подключения могут содержать выражения, что позволяет формировать их уже в процессе выполнения программы. Например, следующая строка определяет подключение к серверу и базе данных, определенным в параметре:

```
="data source=" & Parameters!ServerName.Value
& ";initial catalog=" & Parameters!Database.Value
```

Добавление в отчет параметров для сервера и базы данных позволяет пользователю самому определять источник данных для запроса.

Установка регистрационных данных подключения к источнику

Регистрационные данные, необходимые для доступа к источнику, можно установить для использования аутентификации Windows, аутентификации на уровне базы данных (т.е. смешанного режима), а можно не выбрать ни одного из них. Эти регистрационные данные могут храниться в базе данных сервера отчетности или конфигурироваться пользователем по запросу во время выполнения отчета. Оптимальный выбор в данном случае зависит от конкретной конфигурации сети и настроек системы безопасности.



Более подробно о конфигурировании регистрационных данных источника мы поговорим в разделе "Администрирование системы безопасности" главы 47.

Наборы данных службы отчетности

В службе отчетности набором данных называют результаты запроса, используемого для определения данных, доступных отчету. Не следует путать набор данных службы отчетности с набором данных ADO.NET. Набор данных может быть создан с помощью мастера отчетов (Report Wizard) или вкладки Data окна конструктора отчетов.

Создание набора данных требует наличия источника данных. При этом вы можете либо выбрать существующий общий источник данных, либо создать новый, специфичный для данного отчета.

Один отчет может содержать множество определений наборов данных, каждый из которых открывает доступ к отличному источнику данных. Это значит, что один отчет может содержать данные из множества баз и источников данных XML. Наборы данных могут содержать текст инструкций SQL, вызовы хранимых процедур и запросов XML. Наборы данных также могут содержать параметры и фильтры, ограничивающие данные, возвращаемые в отчете.

Параметры запроса и отчета

Благодаря параметрам пользователь имеет возможность определять критерии отчета и его форматирование. Параметры запроса включаются в определение набора данных и используются для отбора и/или фильтрации данных. Например, параметр `@CategoryID` может быть использован для возвращения подкатегорий выбранной категории, что и продемонстрировано в следующем примере:

```
Select * From Subcategory Where CategoryID = @CategoryID
```

Параметры запроса могут быть также использованы для определения значений, передаваемых хранимой процедуре. В следующем примере вызывается хранимая процедура `uspSubcategories`, содержащая параметр идентификатора категории:

```
EXEC uspSubcategories @CategoryID
```

Параметры отчета используются для управления форматированием и режимом работы отчета. Например, добавление булева параметра `ShowDetails` позволит управлять видимостью строк отчета, отображающих детали. Параметры позволяют управлять практически всеми элементами структуры отчета. Параметры запроса могут быть связаны с параметрами отчета во вкладках Parameters и Filters окна свойств набора данных.



Служба отчетности в SQL Server 2005 допускает динамическую группировку (критерии для групп в таблице могут создаваться с использованием параметров отчета).

Доступно множество вариантов представления параметров в отчете. Например, установка типа параметра в `Boolean` приведет к отображению параметра в виде переключателя с состояниями `True` и `False`; а установка в `Integer` и добавление списка доступных значений — к формированию раскрывающегося списка.



Теперь параметры отчета можно сконфигурировать так, чтобы они допускали выбор множества значений. На рис. 46.1 показан результат создания параметра цвета с установленным свойством многозначности. При этом также можно выбрать значения, установленные по умолчанию.

Для определения значений параметра и установок по умолчанию доступны три настройки: `None`, `Non-queried` и `From Query`. При выборе варианта `None` в отчет добавляется текстовое поле, позволяющее пользователю вводить любое значение. При выборе вариантов `Non-queried` и `From Query` в отчет добавляется раскрывающийся список, позволяющий выбрать любое из доступных значений. Отличие этих вариантов состоит в том, что в первом случае значения, содержащиеся в списке, вводит автор отчета, а во втором он наполняется значениями из некоторого набора данных.

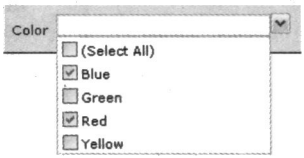


Рис. 46.1. Возможность выбора множества значений параметра представляет собой значимую новую функцию службы отчетности в SQL Server 2005

Содержимое и композиция отчета

Содержимое и композиция (раскладка) отчета определяются во вкладке Layout конструктора отчетов. Любой отчет содержит разделы заголовка, колонтитулов и тела. Для удовлетворения требованиям форматирования доступно множество различных элементов отчета.



В версии SQL Server 2005 стала возможной разработка пользовательских элементов отчета, которые можно добавлять на панель инструментов и использовать в отчетах для обеспечения функциональности, выходящей за пределы обеспечиваемой встроенными в Visual Studio элементами. Пользовательские элементы отчета могут быть связаны с источником данных и использованы для сортировки, группировки, фильтрации и вычисления выражений механизмом обработки отчетов.

В табл. 46.1 перечислены элементы отчета, включенные в состав Visual Studio 2005.

Таблица 46.1. Элементы отчетов в Visual Studio 2005

Элемент отчета	Описание
Textbox (Текстовое поле)	Добавление в отчет содержимого, не находящегося в таблице или матрице. Текстовое поле может содержать статический текст или выражение
Line (Линия), Rectangle (Прямоугольник)	Добавление в отчет визуальных разделителей
Image (Изображение)	Помещение в отчет изображения. Источник изображения может быть встроенным или содержаться в проекте, базе данных или в Интернете. Visual Studio 2005 содержит специальный мастер, максимально облегчающий вставку изображений

Элемент отчета	Описание
Table (Таблица)	Вставка в отчет деталей набора данных. Этот элемент отчета может быть связан с набором данных; к тому же он содержит множество параметров управления группировкой, сортировкой, экспортом и представлением данных. В отчете может содержаться несколько таблиц, что обеспечивает возможность включения в него информации из множества источников и наборов данных
Matrix (Матрица)	Сводное (перекрестное) представление информации из набора данных. Например, элемент матрицы может отображать общие объемы продаж по регионам и периодам. В этом элементе можно организовать множество критериев столбцов, строк и деталей. Работа с матрицей во многом сходна с работой с таблицей
List (Список)	Список, связанный с набором данных. Содержимое списка повторяется для каждой строки набора данных или для каждой групповой строки, если определен критерий группы. Тело списка представляет шаблон для отображаемых элементов отчета
Chart (Диаграмма)	Этот элемент позволяет включить в отчет множество типов диаграмм. Параметры этого элемента обеспечивают максимальное управление типом и форматированием диаграммы
Subreport (Подотчет)	Этот элемент используется для выполнения в отчете вложенного отчета

Процесс создания отчета

Процесс создания отчета включает в себя создание нового проекта службы отчетности в Visual Studio 2005, определение источника данных для отчета, добавление отчета в проект, создание набора данных, извлекаемого из источника, а также форматирование содержимого отчета. В этом разделе будут описаны все основные задачи, стоящие при создании отчета, а также приведены примеры, которые позволят вам самостоятельно выполнить этот процесс.

Создание проекта службы отчетности в Visual Studio 2005

Visual Studio 2005 содержит средства, необходимые для создания и развертывания отчетов службы отчетности. Для создания нового проекта в этом инструментарии выполните следующие действия.

1. Откройте Visual Studio 2005.
2. Выберите в меню пункт File⇨New⇨Project, чтобы открыть диалоговое окно создания нового проекта (рис. 46.2). (В качестве альтернативы можете щелкнуть на ссылке Create Project в разделе Recent Projects титульной страницы Visual Studio 2005, которая открывается по умолчанию при загрузке программы.)
3. На панели типа проекта в левой части окна выберите пункт Business Intelligence Projects.
4. На панели Templates в правой части окна выберите шаблон Report Server Project.
5. Присвойте проекту имя.

6. Определите место размещения проекта. Для создания отдельной папки, в которой будет храниться файл решения, с подпапкой для проекта отчета установите флажок Create directory for solution. Именем создаваемой папки будет имя решения, введенное в поле Solution Name, а именем подпапки — имя проекта отчета, введенное в поле Name. Для создания и решения, и проекта в одной папке снимите флажок Create directory for solution.
7. Щелкните на кнопке OK, и новый проект службы отчетности будет создан.

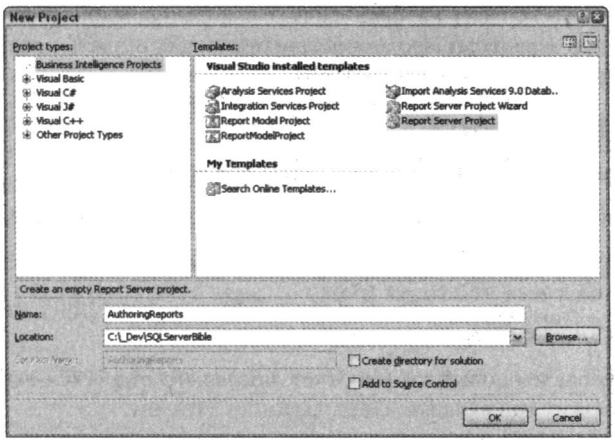


Рис. 46.2. Для создания нового проекта в Visual Studio 2005 используется диалоговое окно New Project

Создание отчета

Для создания отчета в Visual Studio 2005 доступны два метода: использование специального мастера и добавление пустого отчета. Следующие действия описывают общий процесс создания отчета, независимо от выбранного метода.

1. Добавьте в проект отчет, выбрав в меню пункт Project⇨Add New Item. Выберите шаблон Report Wizard, чтобы в создании отчета помог мастер, или Report — для создания пустого отчета.
2. Создайте источник данных.
3. Создайте один или несколько наборов данных.
4. Спроектируйте компоновку элементов отчета.
5. Добавьте и сконфигурируйте параметры отчета.
6. Воспользуйтесь вкладкой Preview для предварительного просмотра отчета.

Использование мастера для создания отчета

Мастер отчетов (Report Wizard) является отличным средством ускорения создания простых отчетов. Этот мастер поможет пользователю быстро выбрать (или создать) источник данных, спроектировать запрос, поставляющий данные, выбрать тип отчета и его стиль, а также присвоить имя отчету. Весь этот процесс займет не больше минуты, а на выходе вы получите готовый к запуску отчет.

Если с помощью мастера вам удалось удовлетворить всем требованиям, выдвигаемым к отчету, считайте, что вам повезло. На практике создание отчета с помощью мастера является не более чем отправной точкой — после этого запускают конструктор отчетов (Report Designer) и начинают вручную подгонять отчет под выдвинутые требования.

Создание отчета с нуля

Создание отчета без использования мастера предполагает добавление отчета в проект, определение одного или нескольких наборов данных и моделирование раскладки элементов отчета. В следующем примере будет продемонстрировано, как создать простой товарный отчет на основе данных учебной базы AdventureWorks.

Создание пустого отчета

Для создания пустого отчета выполните следующие действия.

1. Выберите в меню Visual Studio 2005 пункт Project⇨Add New Item.
2. В диалоговом окне добавления нового элемента проекта выберите шаблон Report.
3. Присвойте отчету имя `Product List`.
4. Щелкните на кнопке Add.

В проект будет добавлен новый пустой отчет, после чего откроется окно конструктора отчетов. Это окно содержит три вкладки: Data, Layout и Preview.

Создание общего источника данных

На рис. 46.3 показано диалоговое окно Shared Data Source, сконфигурированное для базы данных AdventureWorks. Для создания нового общего источника данных выполните следующие действия.

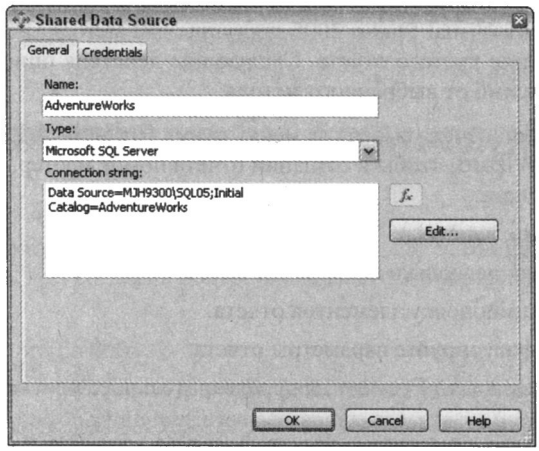


Рис. 46.3. Создание общего источника данных

1. Для того чтобы открыть диалоговое окно Add New Item, выберите в меню пункт Project⇨Add New Item.
2. Выберите шаблон Data Source и щелкните на кнопке Add. Откроется диалоговое окно Shared Data Source.

3. Присвойте источнику данных имя `AdventureWorks`.
4. В раскрывающемся списке Type выберите тип источника данных Microsoft SQL Server.
5. Введите строку подключения к базе данных AdventureWorks, воспользовавшись кнопкой Edit.
6. Щелкните на кнопке OK, и новый источник данных будет добавлен в проект.

Создание набора данных

Для создания набора данных, содержащего информацию о товарах, выполните следующие действия.

1. В конструкторе отчетов перейдите к вкладке Data (рис. 46.4).
2. В раскрывающемся списке Dataset выберите пункт `<New Data Set...>`.
3. Присвойте набору данных имя `Products`.
4. Выберите источник данных `AdventureWorks`.
5. Щелкните на кнопке OK. Несмотря на то что запрос SQL можно ввести и непосредственно в этом диалоговом окне, гораздо проще воспользоваться для этого хорошо знакомым конструктором запросов (Query Designer), содержащимся во вкладке Data конструктора отчетов.

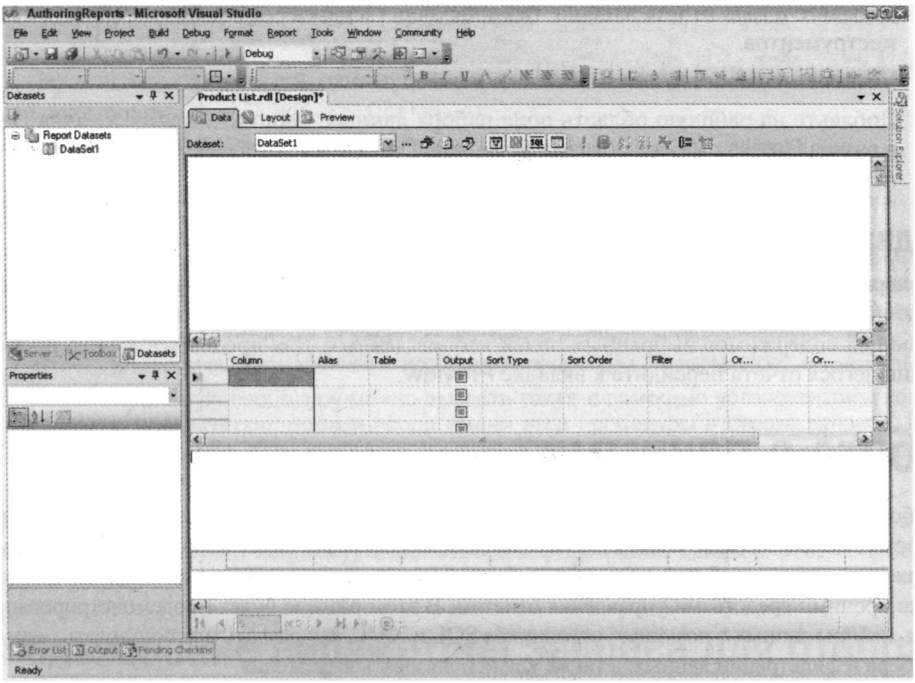


Рис. 46.4. Вкладка Data конструктора отчетов Visual Studio 2005 содержит хорошо знакомый расширенный конструктор запросов, облегчающий создание инструкций SQL

6. Введите следующую инструкцию SQL (представленный в качестве примера запрос возвращает информацию о товаре, в том числе названия категории и подкатегории, а также идентификатор товара из базы данных AdventureWorks):

```
SELECT P.ProductID, P.[Name], P.ProductNumber, P.MakeFlag,
    P.Color, P.[Size], P.ListPrice, P.StandardCost,
    P.ProductSubcategoryID, SubCat.[Name] AS Subcategory,
    SubCat.ProductCategoryID, Cat.[Name] AS Category
  FROM Production.Product P
    INNER JOIN Production.ProductSubcategory SubCat
      ON P.ProductSubcategoryID = SubCat.ProductSubcategoryID
    INNER JOIN Production.ProductCategory Cat
      ON SubCat.ProductCategoryID = Cat.ProductCategoryID
```

7. Выполните запрос с целью его проверки, просмотрите полученный результат и обновите список полей, доступных для отчета. Вы должны увидеть набор данных, содержащий список товаров, а также названий их категорий и подкатегорий. Все эти данные будут доступны в отчете.

Отображение данных в отчете

Результаты запроса могут быть отображены в отчете с помощью элементов списка, таблицы или матрицы. Работа с этими элементами будет подробно описана далее в этой главе. Для отображения результатов набора данных выполните следующие действия. В них создается тело отчета, выполняется связывание с набором данных, и в таблицу добавляются поля данных.

1. Перейдите к вкладке Layout в конструкторе отчетов или выберите в меню пункт Report⇨View⇨Layout.
2. Добавьте в тело отчета элемент таблицы, перетаскивая его в рабочую область с панели инструментов.
3. Установите свойство `DataSetName` в набор данных `Products`.
4. Добавьте на рабочую область поля набора данных, перетаскивая их из окна Dataset в раздел Details рабочей области. Если окно Datasets не отображается, выберите в меню пункт View⇨Datasets.

Предварительный просмотр отчета

Теперь наш отчет списка товаров содержит информацию, полученную запросом из общего источника данных `AdventureWorks`. Композиция отчета состоит из одной таблицы, отображающей содержимое выбранных полей набора данных. Для предварительного просмотра получившегося отчета перейдите к вкладке Preview.

Работа с данными

Работа с данными в службе отчетности SQL Server 2005 вооружила создателей отчетов возможностью использования множества источников данных различных типов. Используя параметры и выражения, можно снабдить пользователей отчетов элементами управления содержимым отчета, а также прочими средствами управления отчетом. В этом разделе будет продемонстрировано, как создать наборы данных с помощью источников SQL и XML, как создать и использовать параметры для отбора и фильтрации данных, а также как использовать выражения.

Работа с SQL в конструкторе отчетов

Панель инструментов конструктора отчетов содержит кнопку, позволяющую переключаться между стандартным (Generic Query Designer) и расширенным (Advanced Query Designer) конструкторами запросов. Стандартный конструктор состоит из двух панелей, SQL и Results, в то

время как расширенный добавляет к ним еще две: Giagram и Grid. На рис. 46.5 показано окно расширенного конструктора запросов, в котором отображается запрос `Products`, созданный ранее в этой главе. Обратите внимание на то, что окно Datasets, находящееся в левой части, содержит набор данных `Products`, в том числе поля, возвращаемые запросом SQL.

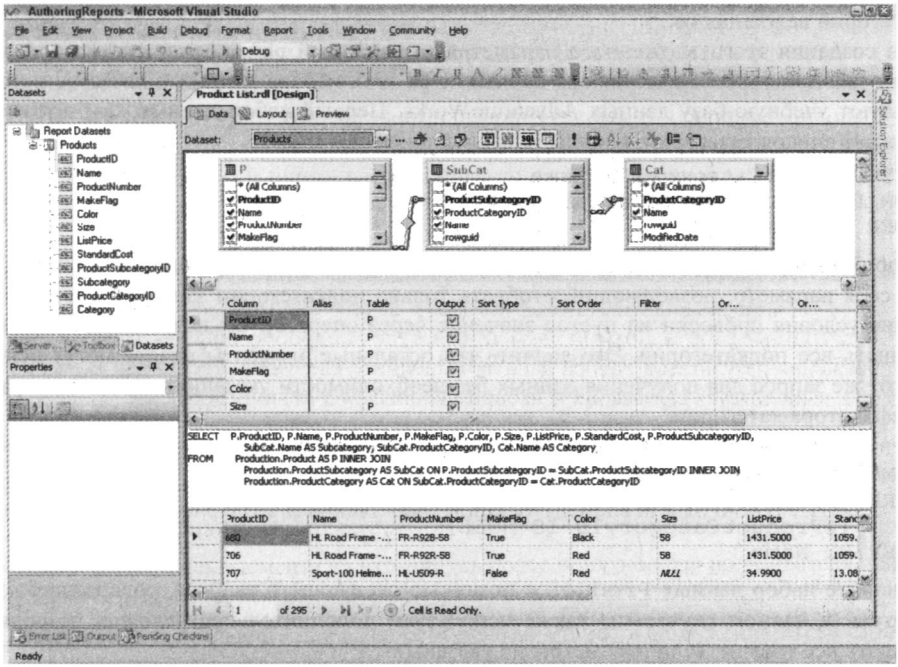


Рис. 46.5. Расширенный конструктор запросов упрощает процесс написания инструкций SQL



Для улучшения использования SQL Server и минимизации обслуживания отчета создайте хранимые процедуры и вызывайте их в отчете. Например, набор данных `Products` может возвращать хранимая процедура `uspProducts`. Вместо написания инструкции SQL непосредственно в отчете можно установить тип команды в `text` и вызвать хранимую процедуру следующим образом: `EXEC uspProducts`.

Хранимую процедуру можно вызвать также с помощью установки типа команды в `StoredProcedure` и ввода имени этой процедуры в строке запроса. В данном случае, если хранимая процедура имеет параметры, следует использовать вкладку Parameters страницы Dataset Properties.

Независимо от того, куда включена инструкция SQL, в отчет или в хранимую процедуру, параметры запроса обычно используют для отбора и фильтрации данных. Этот вопрос будет освещен в следующем разделе.

Использование параметров запроса для отбора и фильтрации данных

Как уже говорилось ранее, параметры запроса можно использовать для отбора и фильтрации данных для отчета. Например, в следующей инструкции параметры `CategoryID` и `SubcategoryID` передаются хранимой процедуре `uspProducts`:

```
EXEC uspProducts @CategoryID, @SubcategoryID
```

Вложенные параметры отчета

В предлагаемом примере продемонстрировано, как использовать параметры в наборах данных SQL и как создавать параметры, зависимые от выбора других параметров. Например, выбор категории `Bikes` параметра `Category` наполняет параметр `Sybcategory` списком подкатегорий велосипедов.

Для создания этого вложенного параметра в созданный ранее отчет списка товаров мы добавим два дополнительных набора данных. Оба этих набора данных в качестве источника используют учебную базу данных AdventureWorks. Первый набор данных, `CategoryList`, возвращает список категорий, используя следующий запрос SQL:

```
SELECT ProductCategoryID, Name
  FROM Production.ProductCategory
  ORDER BY Name
```

Второй набор данных, `SubcategoryList`, возвращает список подкатегорий. Он включает в себя параметр, позволяющий отобрать только подкатегории заданной категории. Добавление условия проверки на пустое значение через оператор `OR` позволяет этому запросу возвращать все подкатегории. Это значит, что остальные элементы отчета могут использовать этот же запрос для получения данных без необходимости указания в качестве параметра идентификатора категории.

```
SELECT ProductSubcategoryID, ProductCategoryID, Name
  FROM Production.ProductSubcategory
  WHERE (ProductCategoryID = @ProductCategoryID) OR
        (@ProductCategoryID IS NULL)
  ORDER BY Name
```

Обновите набор данных `Product` с помощью следующего запроса, содержащего новые параметры (в данном случае мы также используем дополнительное условие, позволяющее возвращать все товары, если параметры не указаны):

```
SELECT P.ProductID, P.[Name], P.ProductNumber, P.MakeFlag,
    P.Color, P.[Size], P.ListPrice, P.StandardCost,
    P.ProductSubcategoryID, SubCat.[Name] AS Subcategory,
    SubCat.ProductCategoryID, Cat.[Name] AS Category
  FROM Production.Product P
  INNER JOIN Production.ProductSubcategory SubCat
    ON P.ProductSubcategoryID = SubCat.ProductSubcategoryID
  INNER JOIN Production.ProductCategory Cat
    ON SubCat.ProductCategoryID = Cat.ProductCategoryID
  WHERE (P.ProductSubcategoryID = @ProductSubcategoryID
    OR @ProductSubcategoryID IS NULL)
    AND (SubCat.ProductCategoryID = @ProductCategoryID
    OR @ProductCategoryID IS NULL)
```



Создание наборов данных, поддерживающих необязательные параметры, может пригодиться во множестве ситуаций, однако будьте бдительны: такой запрос может вернуть невообразимый объем данных. Чтобы застраховаться от подобных ситуаций, вы можете потребовать от пользователя обязательно выбрать параметр до выполнения запроса.

Для конфигурирования параметров запроса выберите в меню пункт Report⇨Report Parameters. На рис. 46.6 показано диалоговое окно Report Parameters, в котором выделен параметр `ProductSubcategoryID`.

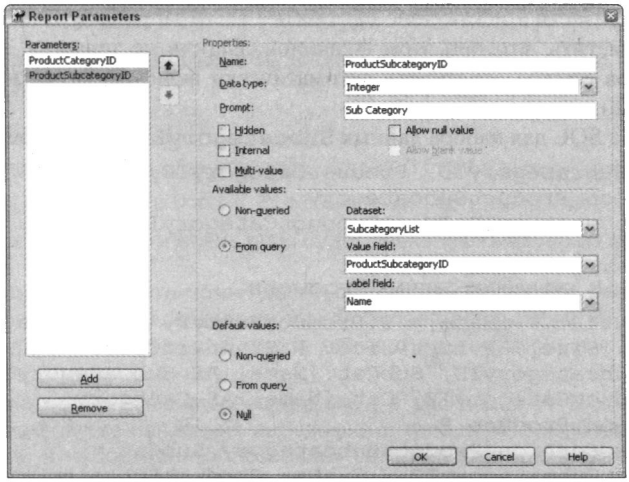


Рис. 46.6. Диалоговое окно **Report Parameters** *позволяет осуществлять расширенное управление параметрами отчета*

Для конфигурирования параметров отчета выполните следующие действия.

1. Выделите параметр `ProductCategoryID`.
2. Измените тип данных на `Integer`.
3. Измените подпись элемента на что-нибудь более понятное для пользователя, например `Категория`.
4. Установите переключатель в разделе Available Values в положение From Query. Это позволит заполнить список доступными значениями из набора данных.
5. Выберите в списке набор данных `CategoryList`.
6. В раскрывающемся списке Value Field выберите элемент `ProductCategoryID`.
7. В раскрывающемся списке Label Field выберите элемент `Name`.
8. Аналогичным образом сконфигурируйте параметр `ProductSubcategoryID`, согласно значениям, показанным на рис. 46.6.

Воспользуйтесь предварительным просмотром отчета. Следует заметить, что параметр подкатегории отключен до тех пор, пока вы не выберете значение для параметра категории. После выбора категории в параметре подкатегории отображается список соответствующих подкатегорий. Щелкните на кнопке View Report, и в созданном отчете отобразятся только товары из выбранной подкатегории.

Многозначные параметры отчета

В приведенном выше примере было показано, как создавать отчет, имеющий вложенные параметры. Давайте расширим этот отчет, позволив пользователям выбирать множество значений каждого из параметров и включать в отчет все товары, соответствующие выбранным критериям.

Изменение параметров в многозначные требует внести некоторые небольшие изменения в запросы для наборов данных `SubcategoryList` и `Products`. В частности, предложение `WHERE` теперь должно содержать оператор `IN` вместо `EQUALS`. К тому же мы должны устранить возможность возвращения всех строк, когда параметр не определен, так как в противном

случае это может привести к искажению синтаксиса всего запроса SQL во время выполнения. Однако следует заметить, что при этом функциональность не теряется, так как для многозначных параметров служба отчетности автоматически включает возможность выбора всех значений (Select All).

Измените запрос SQL для набора данных `SubcategoryList` следующим образом:

```
SELECT ProductSubcategoryID, ProductCategoryID, Name
  FROM Production.ProductSubcategory
  WHERE ProductCategoryID IN (@ProductCategoryID)
  ORDER BY Name
```

Измените запрос и для набора данных `Products`:

```
SELECT P.ProductID, P.[Name], P.ProductNumber, P.MakeFlag,
    P.Color, P.[Size], P.ListPrice, P.StandardCost,
    P.ProductSubcategoryID, SubCat.[Name] AS Subcategory,
    SubCat.ProductCategoryID, Cat.[Name] AS Category
  FROM Production.Product P
  INNER JOIN Production.ProductSubcategory SubCat
    ON P.ProductSubcategoryID = SubCat.ProductSubcategoryID
  INNER JOIN Production.ProductCategory Cat
    ON SubCat.ProductCategoryID = Cat.ProductCategoryID
  WHERE P.ProductSubcategoryID IN (@ProductSubcategoryID)
    AND SubCat.ProductCategoryID IN (@ProductCategoryID)
```

Для конфигурирования многозначного параметра выполните следующие действия.

1. Выберите в меню пункт Report⇨Report Parameters.
2. Выделите параметр `ProductCategoryID` и установите флажок Multi-value.
3. Выделите параметр `ProductSubcategoryID` и установите флажок Multi-value.
4. Щелкните на кнопке OK, чтобы сохранить изменения.

Запустите отчет и выберите категории `Accessories` и `Clothing` — в списке подкатегорий отобразятся элементы обеих выбранных категорий (рис. 46.7). Выделите несколько подкатегорий и запустите отчет — теперь он будет содержать все товары выбранных подкатегорий и не будет стеснен выбором только одного значения параметра.

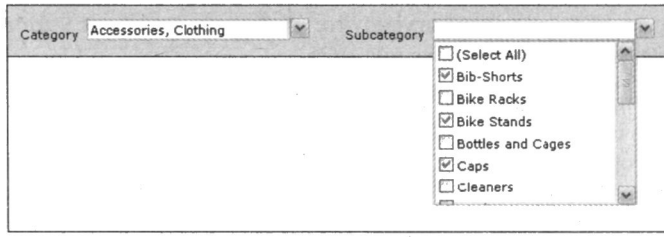


Рис. 46.7. В этом примере продемонстрированы многозначные вложенные параметры

Добавление в набор данных вычисляемых полей

После определения набора данных в него можно добавить новые поля и определить для них соответствующие выражения. Например, в набор данных товаров можно добавить поле `Margin` и установить для него выражение `ListPrice - StandardCost`. Чтобы добавить это поле, выполните следующие действия.

1. Откройте окно Datasets, выбрав в меню пункт View⇨Datasets.
2. Щелкните правой кнопкой мыши на наборе данных `Products` и выберите в контекстном меню пункт Add.
3. Присвойте полю имя `Margin`.
4. Установите переключатель в положение Calculated Field и введите следующее выражение:
   `=Fields!ListPrice.Value - Fields!StandardCost.Value`
5. Щелкните на кнопке OK, и новое поле будет добавлено в набор данных.



Несмотря на то что можно добиться того же результата, включая вычисления в запрос SQL, предложенный подход более удобный при вызове хранимых процедур, в которые вы не хотите включать вычисления или которые не имеете права изменять.

Также можно включать выражения и в раскладку отчета; однако это требует многократного написания выражений в теле отчета. Добавление вычисляемых полей в набор данных позволяет централизовать определение выражений и упростить создание и сопровождение отчета.

Работа с источниками данных XML

Служба отчетности в SQL Server 2005 предоставляет возможность использования XML в качестве источника данных для отчетов. Например, вы можете получать такие данные от Web-служб и извлекать из фактов XML, после чего отображать их в отчете.

В следующем примере продемонстрировано создание отчета на основе данных из двух источников XML: файла XML и Web-службы. Этот отчет использует файл XML для заполнения параметра `StockSymbol`. Выбранный стандартный символ акции передается в качестве параметра Web-службе, которая загружает последнюю доступную информацию о выбранных акциях. Следующий файл XML содержит некоторые символы акций и названия ассоциированных с ними компаний:

```
<?xml version="1.0" encoding="utf-8" ?>
<StockSymbols>
<Symbol Value="AAPL" Name="Apple Computer, Inc. (AAPL)" />
<Symbol Value="MSFT" Name="Microsoft Corporation (MSFT)" />
<Symbol Value="ORCL" Name="Oracle Corporation (ORCL)" />
<Symbol Value="STK" Name="Storage Technology Corp. (STK)" />
<Symbol Value="SUNW" Name="Sun Microsystems, Inc (SUNW)" />
</StockSymbols>
```

Для создания нового набора данных, соответствующего файлу `StockSymbols.xml`, выполните следующие действия.

1. Создайте файл `StockSymbols.xml`, содержащий приведенный выше код, и сохраните его в корневом каталоге своего Web-сервера.
2. Создайте новый набор данных `dsStockSymbols`.
3. Сконфигурируйте источник данных с типом XML и следующей строкой подключения:
   `http://localhost/StockSymbols.xml`
4. В качестве запроса формирования набора данных используйте следующий код:

```
<Query>
<ElementPath IgnoreNamespaces="true">
StockSymbols/Symbol
</ElementPath>
</Query>
```

5. Выберите режим аутентификации Windows (этот режим выбран по умолчанию при создании нового источника данных).
6. Выполните введенный запрос, чтобы проверить успешность подключения. Это также позволит обновить определение набора данных в отчете.

Теперь создайте параметр отчета `Symbols` и используйте набор данных `dsStockSymbols` для заполнения его значениями. Для этого выполните следующие действия.

1. Откройте диалоговое окно параметров отчета, выбрав в меню пункт Report⇨Report Parameters.
2. Щелкните на кнопке Add, чтобы добавить новый параметр.
3. Присвойте параметру имя `Symbols`, установите тип данных в `String` и введите для него подпись типа "Символ акции".
4. В разделе Available Values установите переключатель в положение From Query.
5. В раскрывающемся списке Dataset выберите набор данных `dsStockSymbols`.
6. В раскрывающемся списке ValueField выберите `Value`.
7. В списке Label Field выберите `Name`.
8. Щелкните на кнопке OK, чтобы сохранить новые параметры и закрыть диалоговое окно Report Parameters.

При выполнении отчета пользователю будет предложено выбрать символ акций для параметра, чтобы получить соответствующий курс. Теперь нам осталось только создать набор данных, извлекающий информацию из Web-службы. Корпорация CDYNE предлагает Web-службу для извлечения отсроченных курсов акций. Метод `GetQuote` этой Web-службы возвращает курс акций и требует предоставления двух параметров: `StockSymbol` (символ акции) и `LicenseKey` (ключ лицензии). Первый параметр можно ассоциировать с одноименным параметром отчета, а второму параметру в данном случае нужно назначить значение 0 — этот ключ предлагается компанией для тестовых задач. Для создания набора данных `Stock Quote` выполните следующие действия.

1. Создайте новый набор данных `dsStockQuote`.
2. Сконфигурируйте источник данных с типом XML и следующей строкой подключения:
   ```
   http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx
   ```
3. В качестве запроса формирования набора данных используйте
   ```
   <Query>
   <Method Namespace="http://ws.cdyne.com/"
   Name="GetQuote}" />
   <SoapAction>http://ws.cdyne.com/GetQuote</SoapAction>
   </Query>
   ```
4. Во вкладке Parameters окна свойств набора данных (чтобы открыть это окно, щелкните на кнопке с многоточием рядом с раскрывающимся списком DataSet) добавьте параметр StockSymbol и выберите `Symbols` в качестве его значения. Добавьте второй параметр, LicenseKey, и установите для него значение 0 (не используйте при этом знак равенства). Проверьте правильность регистров символов в именах параметров.
5. Выполните запрос, чтобы убедиться в работоспособности подключения и правильности параметров. При этом также будет обновлено определение набора данных в отчете.

Добавьте содержимое в тело отчета, чтобы отображать курсы акций. Воспользуйтесь предварительным просмотром, и вся мощь средств XML предстанет перед вашими глазами.

Работа с выражениями

Выражения являются мощным ресурсом для создателей отчетов. Например, с помощью выражений можно консолидировать данные, выполнять вычисления, форматировать текст и управлять композицией отчета. На рис. 46.8 показан редактор выражений, входящий в состав Visual Studio 2005.

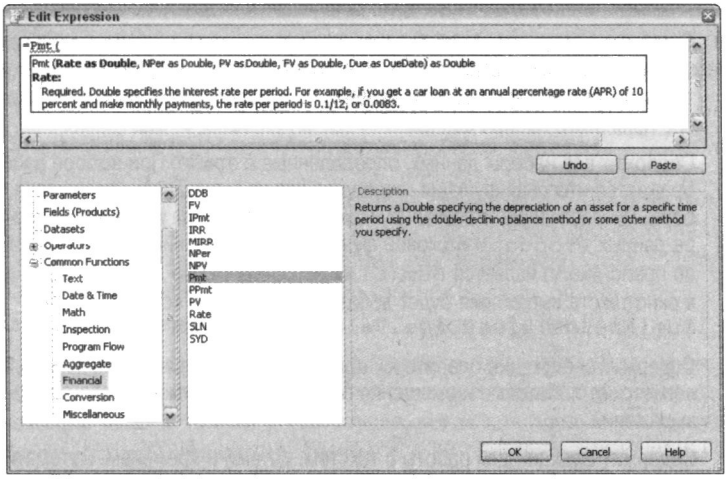


Рис. 46.8. Редактор выражений теперь оснащен дополнительными средствами, такими как IntelliSense и список доступных функций

Новинка 2005

Функциональность редактора выражений была расширена с целью облегчения работы при создании выражений. Теперь он содержит список доступных функций, а также средства IntelliSense, обеспечивающие правильное моделирование инструкций, проверку синтаксиса и отображение оперативной информации о параметрах.

Редактор выражений можно открыть с помощью выбора элемента `<Expression...>` в списке значений параметра, а также щелчком правой кнопкой мыши на объекте с последующим выбором в контекстном меню пункта Expression. Также можно щелкнуть на функциональной кнопке с надписью fx, расположенной рядом с параметром на странице свойств. Редактор выражений содержит окно текста выражения, дерево категорий, элементы категорий и панель описаний. Дважды щелкните на значении на панели категорий или описаний, и в окно текста выражения будет вставлен соответствующий код.

В табл. 46.2 перечислены категории, доступные в редакторе выражений.

Таблица 46.2. Категории, доступные в редакторе выражений

Категория	Описание
`Constants`	Категория констант доступна не для всех элементов и параметров отчета. В зависимости от параметра, для которого создается выражение, в этой категории будут перечислены только значения, применимые к данному свойству. Например, при редактировании выражения для установки цвета фона строки таблицы категория `Constants` будет содержать список доступных цветов, а также элементы, позволяющие создавать дополнительные цвета

Категория	Описание
`Globals`	Содержит функции доступа к информации об отчете и его выполнении, такие как `ExecutionTime` (время выполнения), `PageNumber` (номер страницы), `TotalPages` (общее число страниц в отчете) и `UserID` (идентификатор пользователя)
`Parameters`	Содержит список параметров, определенных в отчете
`Fields`	Содержит список полей набора данных, находящихся в области определения выбранного элемента или параметра отчета. Например, при редактировании выражения для ячейки таблицы, связанной с набором данных `Products`, в этой категории будут перечислены все поля этого набора данных
`Datasets`	Содержит все наборы данных, определенные в отчете. При выборе набора данных отображается итоговая функция, определенная по умолчанию для каждого поля этого набора. По умолчанию итоговая функция зависит от типа данных поля. Например, в наборе данных `Products` итоговой функцией по умолчанию для поля `ListPrice` (цена по прайс-листу) является `Sum(ListPrice)`. После двойного щелчка на этом поле в окно текста выражения будет добавлен следующий код: `Sum(Fields!ListPrice.Value,"Products")`
`Operators`	Содержит следующие операторы: арифметические, сравнения, конкатенации, логические/побитовые и смещения битов, которые можно использовать при создании выражений
`Common Functions`	Содержит функции для работы с текстом, датами и временем, математические, управления потоком команд, итоговые, финансовые и преобразования данных

Область определения выражения

Итоговые функции позволяют задавать область, которую охватывают вычисления. Под областью определения понимается либо имя набора данных, либо имя группы или диапазона данных, содержащих элемент отчета, в котором используется итоговая функция.

В качестве примера рассмотрим отчет о продажах в разрезе товаров. В нем содержится таблица с группой, имеющей имя `grpProduct`. Чтобы добавить в этот отчет столбец Running Total, который инициализирует накопительный итог для каждой группы товаров, можно использовать следующее выражение:

```
=RunningValue(Fields!LineTotal.Value, Sum, "grpProduct")
```

Для создания аналогичного столбца накопительного итога, не инициализируемого в каждой группе товаров, используется следующее выражение:

```
=RunningValue(Fields!LineTotal.Value, Sum)
```

Область определения выражения может также быть важной при добавлении выражения в текстовое поле. Так как элемент текстового поля в отчете не может быть непосредственно связан с набором данных, выражение должно включать область определения набора данных. Следующее выражение вычисляет сумму значений поля `LineTotal` (итог по строке) в наборе данных `Sales` (продажи):

```
=Sum(Fields!LineTotal.Value, "Sales")
```

Оформление отчета с помощью выражений

В приведенных ниже примерах продемонстрированы некоторые наиболее распространенные выражения, используемые в отчетах. Используя следующее выражение в качест-

ве свойства строки деталей элемента таблицы, можно выделить цветом фона AliceBlue все нечетные строки:

```
=IIf(RowNumber(nothing) mod 2 = 1, "AliceBlue", "White")
```

Целесообразно добавлять в отчет дату и время его выполнения. Следующее выражение позволит включить в отчет строку типа "Отчет выполнен: Monday, August 15, 2005 в 2:24:33 pm":

```
="Отчет выполнен: " & Globals!ExecutionTime.ToLongDateString & " в " &
Globals!ExecutionTime.ToLongTimeString
```

Выражения можно использовать для форматирования текста. Следующее выражение вычисляет сумму по полю LineTotal набора данных Sales и форматирует результат как строку денежного типа, такую как 4 231 205,23р.:

```
=FormatCurrency(Sum(Fields!LineTotal.Value, "Sales"), 2, true,
                true, true)
```

Иногда не имеет смысла отображать некоторые элементы отчета; при этом решение принимается на основе выбранных параметров. Для переключения свойства видимости некоторого элемента отчета или даже ячейки таблицы для параметра Visibility.Hidden используют выражение, подобное следующему:

```
=IIf(Parameters!CategoryID.Value = 10, true, false)
```

С помощью выражений можно даже спроектировать текст команды, формирующей набор данных. Выражение, приведенное в следующем примере, позволяет включить в инструкцию SELECT предложение WHERE только в том случае, когда для параметра идентификатора подкатегории (SubcategoryID) указано значение:

```
="Select * From Products " &
IIf(Parameters!SubcategoryID.Value = -1, "",
 WHERE ProductSubcategoryID = " & Parameters!SubcategoryID.Value) &
                              " Order By ProductName"
```

Проектирование композиции отчета

Конструктор отчетов в Visual Studio 2005 содержит большой набор средств, помогающих создать композицию элементов отчета. В этом разделе будут описаны основы проектирования отчетов и продемонстрированы способы создания отчетов, группировки и сортировки данных, а также добавления в отчет диаграмм.

Основные элементы композиции отчета

Вкладка Layout конструктора отчетов содержит множество средств, позволяющих форматировать даже самые сложные отчеты. Эта вкладка разделена на три раздела: верхний колонтитул, тело и нижний колонтитул. В табл. 46.3 описаны назначение каждого из этих разделов и режимы их работы. Проектирование композиции отчета сходно с работой с формами Windows. Элементы отчета добавляются путем их перетаскивания с панели Toolbox в рабочую область конструктора отчетов.


Проверено

Так как отчет фиксирует состояние набора данных в конкретный момент времени, довольно важно включать в него ответы на ведущую пятерку вопросов идентификации: "Кто? Что? Где? Когда? Зачем?". Кто запустил отчет? Какие критерии были заданы при генерации отчета? Когда данный отчет был создан? Зачем создавался данный отчет? Единообразная вставка ответов на эти вопросы во все отчеты позволит избежать впоследствии недоразумений и неопределенности.

Таблица 46.3. Разделы отчета

Раздел	Описание
Header (Верхний колонтитул)	По умолчанию содержимое верхнего колонтитула помещается на каждую страницу. Это отличное место для размещения заголовков и описаний назначения отчета. Параметры `PrintOnFirstPage` и `PrintOnLastPage` определяют, следует ли применять верхний колонтитул соответственно к первой и последней страницам отчета
Body (Тело отчета)	Если отчет содержит параметры, может оказаться разумным добавить в верхнюю или нижнюю часть тела отдельный раздел, отображающий значения параметров, использованных при генерации отчета, а также, возможно, краткое описание назначения отчета. Добавление этих деталей в верхнюю или нижнюю секцию тела отчета обеспечит только одно воспроизведение этой информации, а не на всех страницах отчета. Проектирование тела отчета напоминает создание форм Windows и Web-форм. При запуске отчета на выполнение все его элементы займут свои позиции в отчете относительно друг друга
Footer (Нижний колонтитул)	Подобно верхнему колонтитулу, нижний воспроизводится по умолчанию на всех страницах и может быть отключен для первой и последней страниц. В этот раздел полезно включать информацию о том, кто и когда выполнил отчет, версию отчета, а также номера страниц

Проектирование верхнего колонтитула

Чтобы добавить в отчет верхний колонтитул, выполните следующие действия.

1. Чтобы отобразить раздел верхнего колонтитула, щелкните правой кнопкой мыши на левом поле отчета и выберите в контекстном меню пункт Page Header. Имейте в виду, что все добавленное в этот раздел содержимое будет удалено, если отключить отображение этого раздела.
2. Добавьте в верхний колонтитул текстовое поле с панели Toolbox.
3. Введите заголовок отчета и отформатируйте текстовое поле. Содержимое заголовка можно сделать динамическим — т.е. зависящим от содержимого отчета и выбранных параметров. Для этого можно воспользоваться выражениями. Практически все визуальные аспекты текстового поля также допускают использование выражений.

Проектирование нижнего колонтитула

В нижнем колонтитуле принято отображать информацию о том, кто и когда выполнил отчет, а также номера страниц. Для вывода этого типа информации также можно использовать выражения. Чтобы добавить в отчет нижний колонтитул, выполните следующие действия.

1. Чтобы отобразить раздел верхнего колонтитула, щелкните правой кнопкой мыши на левом поле отчета и выберите в контекстном меню пункт Page Header.
2. Добавьте в раздел три текстовых поля, используя следующие выражения:

```
=User!UserID
=Globals!PageNumber & " из " & Globals!TotalPages
=Globals!ExecutionTime
```

3. Вставьте над текстовыми полями линию, чтобы визуально отделить нижний колонтитул.

Добавление и форматирование элемента таблицы

Элемент Table используют для помещения в отчет информации из набора данных. В качестве примера добавим в отчет элемент таблицы, отображающий данные о товарах из набора данных `Products`.

1. В раздел тела отчета добавьте элемент таблицы из панели Toolbox.
2. Установите параметр DataSetName в набор данных `Products`. Если в отчете определен только один набор данных, то элемент таблицы будет к нему подключен автоматически. Если же в отчете определено больше одного набора данных, то эту установку придется выполнить вручную.
3. Перетащите поля из набора данных в раздел деталей таблицы. Заголовки столбцов автоматически будут установлены в имена полей. Следует заметить, что помещение полей данных в разделы колонтитулов таблицы приведет к формированию выражения, либо выводящего значение из первой строки набора данных, либо вычисляющего сумму числовых значений поля в пределах набора данных.
4. Отформатируйте таблицу, удаляя и добавляя столбцы, а также устанавливая различные параметры, управляющие представлением данных. При желании вы можете вставить дополнительные таблицы, списки, изображения и прочие элементы.
5. Выделите левую границу верхнего колонтитула таблицы и установите для параметра RepeatOnNewPage значение `true`. Это приведет к отображению заголовков столбцов таблицы на всех страницах отчета.
6. Щелкните правой кнопкой мыши на левой границе нижнего колонтитула таблицы и выберите в контекстном меню пункт Table Footer. В результате выполнения этой операции раздел нижнего колонтитула таблицы будет удален.

На рис. 46.9 представлена общая композиция созданного отчета, включая колонтитулы и таблицу. На рис. 46.10 показаны результаты выполнения отчета.

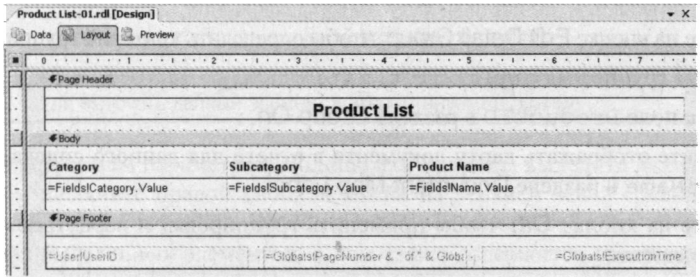


Рис. 46.9. Композиция отчета с отформатированными колонтитулами и телом отчета

Добавление и форматирование элемента списка

Использование в отчете элемента списка (List) является отличным способом включения в него сводных данных. В качестве примера рассмотрим сводный отчет о продажах, содержащий таблицу данных, сгруппированных по номерам и датам заказов. Элемент List может быть использован для добавления в отчет итогов продаж по товарам. Сортировка списка в порядке убывания позволит быстро выявить десятку самых продаваемых товаров. Таким образом, теперь отчет на основе одного и того же набора данных сможет представить две точки зрения на них.

Product List

Category	Subcategory	Product Name
Components	Road Frames	HL Road Frame - Black, 58
Components	Road Frames	HL Road Frame - Red, 58
Accessories	Helmets	Sport-100 Helmet, Red
Accessories	Helmets	Sport-100 Helmet, Black
Clothing	Socks	Mountain Bike Socks, M
Clothing	Socks	Mountain Bike Socks, L
Accessories	Helmets	Sport-100 Helmet, Blue
Clothing	Caps	AWC Logo Cap
Clothing	Jerseys	Long-Sleeve Logo Jersey, S
Clothing	Jerseys	Long-Sleeve Logo Jersey, M
Clothing	Jerseys	Long-Sleeve Logo Jersey, L
Clothing	Jerseys	Long-Sleeve Logo Jersey, XL
Components	Road Frames	HL Road Frame - Red, 62
Components	Road Frames	HL Road Frame - Red, 44
Components	Road Frames	HL Road Frame - Red, 48
Components	Road Frames	HL Road Frame - Red, 52
Components	Road Frames	HL Road Frame - Red, 56
Components	Road Frames	LL Road Frame - Black, 58
Components	Road Frames	LL Road Frame - Black, 60
Components	Road Frames	LL Road Frame - Black, 62
Components	Road Frames	LL Road Frame - Red, 44
Components	Road Frames	LL Road Frame - Red, 48
Components	Road Frames	LL Road Frame - Red, 52
Components	Road Frames	LL Road Frame - Red, 58

Рис. 46.10. Выполненный отчет с отформатированными колонтитулами и телом отчета

Для добавления и форматирования описанного выше списка выполните следующие действия.

1. Добавьте элемент List из панели Toolbox в тело отчета.
2. Откройте страницу параметров списка, щелкнув правой кнопкой мыши на элементе списка и выбрав в контекстном меню пункт Properties.
3. Щелкните на кнопке Edit Detail Group, чтобы определить критерий группировки в списке.
4. Присвойте группе имя `grpProductList`.
5. Выделите поле `ProductID` в разделе Group On.
6. Если хотите отображать карту документа в отчете для данного списка, выберите поле `ProductName` в разделе Document Map Label.
7. Щелкните на кнопке OK, чтобы применить группировку и вернуться на страницу параметров списка.
8. Перейдите к вкладке Sorting и введите следующее выражение:
   ```
   =Sum(Fields!LineTotal.Value, "grpProductList")
   ```
9. Установите порядок сортировки `Descending` (по убыванию).
10. Щелкните на кнопке OK, чтобы применить изменения параметров к списку.
11. Добавьте в элемент списка два текстовых поля, используя для их значений следующие выражения:
    ```
    =Fields!ProductName.Value
    =Sum(Fields!LineTotal.Value, "grpProductList")
    ```
12. Установите для параметра Format текстового поля, вычисляющего общий объем продаж, значение `C`, чтобы отображать значения в денежном формате.

13. Добавьте необходимые дополнительные элементы, а затем примените к ним необходимое форматирование.
14. В режиме предварительного просмотра должен отобразиться список товаров, отсортированный в порядке убывания общего объема продаж.

Использование страниц параметров таблицы и матрицы

На страницах параметров элементов таблицы и матрицы определяются такие аспекты функционирования, как связывание с данными, видимость, навигация, фильтры, сортировка (только в таблицах), группировка и состав выводимых данных. В табл. 46.4 сведены все параметры, доступные во вкладках страниц свойств этих элементов. Несмотря на то что состав параметров для таблиц и матриц немного отличается, процесс работы с ними в основном сходный.

Таблица 46.4. Страницы свойств элементов матрицы и таблицы

Вкладка	Описание работы
General	Присвойте матрице или таблице имя, подключите набор данных, определите текст экранных подсказок и разрывы страниц
Visibility	Определите видимость элементов в процессе выполнения отчета. Видимость отдельных элементов может определяться на основе состояния других элементов отчета
Navigation	Включите ссылку на карту документа, чтобы иметь возможность быстро перемещаться в пространстве данных таблицы или матрицы
Sorting (Доступна только для таблиц)	Установите порядок сортировки данных в таблице
Groups	Добавьте и отредактируйте определения групп и измените их порядок
Filters	Определите критерии фильтрации. Если необходимо, используйте выражения для включения в критерии значений параметров отчета
Data Output	В этой вкладке осуществляется управление форматом экспорта. Укажите, следует ли включать данный элемент в экспортируемые данные

Для открытия страницы параметров таблицы или матрицы выделите любую ячейку в этом элементе, а затем щелкните правой кнопкой мыши на верхней левой границе и выберите в контекстном меню пункт Properties. В окне параметров имеется значок, открывающий страницы свойств для выделенных элементов, в нем также содержится раскрывающийся список всех элементов отчета, реализующий альтернативный метод выбора объектов.

Группировка и сортировка данных в элементах таблицы и матрицы

Элементы таблицы и матрицы в отчете предлагают высокий уровень управления группировкой и сортировкой с помощью окна Grouping and Sorting Properties (рис. 46.11). Это окно выглядит одинаково для этих двух типов элементов.

Чтобы открыть окно Grouping and Sorting Properties, щелкните правой кнопкой мыши на границе строки таблицы (или на заголовке столбца матрицы) и выберите в контекстном меню пункт Edit Group. В качестве альтернативы можете воспользоваться вкладкой Groups

диалогового окна параметров таблицы или матрицы. В окне Grouping and Sorting Properties определяется порядок фильтрации и сортировки, а также параметры видимости и вывода данных для выбранной группы.

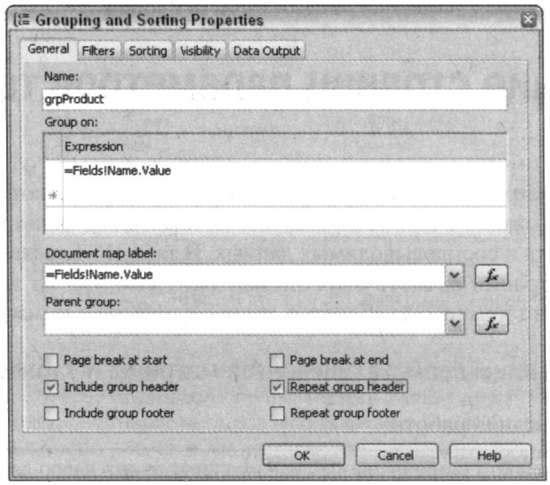


Рис. 46.11. Диалоговое окно **Grouping and Sorting Properties** *используется для создания групп в таблицах и матрицах и управления этими группами*

Во вкладке General этого окна доступны следующие параметры, определяющие группу.

- Name. Это имя будет использоваться при определении области определения выражений.



Для упрощения написания выражений в таблицах и матрицах очень важно снабдить каждую группу собственным именем.

- Group on. Выберите поля набора данных, по которым будет выполняться группировка или построение выражений.
- Document map label. При желании выберите значение, включаемое в карту документа.
- Parent Group. Используется для определения рекурсивной иерархии данных, в которой все отношения между родителями и потомками реализованы в самих данных (примером может служить организационная диаграмма компании).
- **Флажки разрывов страницы**. Управление разрывами страниц и видимостью колонтитулов группы.

Вкладки Filters, Sorting, Visibility и Data Output аналогичны соответствующим вкладкам диалоговых окон свойств таблицы и матрицы, описанных ранее, однако параметры, содержащиеся в них, оказывают влияние только на выбранную группу. Элементы матрицы и таблицы могут содержать несколько групп. Для управления порядком этих групп можно использовать диалоговые окна параметров таблиц и матриц, описанные ранее.

Самым значительным отличием между матрицами и таблицами является то, что матрицы могут содержать определения групп как для строк, так и для столбцов, в то время как таблицы — только для строк. Теперь, когда вы знаете, как создавать группы и управлять ими, вы полностью подготовлены к работе с таблицами и матрицами.

Установка порядка сортировки в таблице

Для того чтобы установить порядок сортировки деталей в элементе таблицы отчета, выполните следующие действия.

1. Откройте диалоговое окно параметров элемента таблицы.
2. Перейдите к вкладке Sorting.
3. В столбце Expression выделите поля, по которым должна выполняться сортировка, а в столбце Direction — ее порядок. С помощью кнопок со стрелками, направленными вверх и вниз, вы можете изменять порядок применения критериев сортировки, а с помощью кнопки Delete — удалять ненужные критерии.
4. Щелкните на кнопке OK, чтобы применить изменения.

Описанные действия позволяют установить порядок сортировки, применяемый в отчете по умолчанию. Чтобы добавить возможность управления сортировкой в отчете в процессе его выполнения, можно воспользоваться новой функцией интерактивной сортировки.



Служба отчетности в SQL Server 2005 содержит функцию интерактивной сортировки (Interactive Sort), позволяющую создателю отчета только определять критерии сортировки для каждого столбца. При выполнении отчета столбцы с включенной интерактивной сортировкой будут содержать в заголовке ссылку, позволяющую конечному пользователю в реальном времени изменять соответствующий порядок сортировки.

Для определения критерия интерактивной сортировки для столбца таблицы выполните следующие действия.

1. Щелкните правой кнопкой мыши на ячейке заголовка столбца, к которому нужно применить критерий сортировки, и выберите в контекстном меню пункт Properties.
2. Перейдите к вкладке Interactive Sort.
3. Установите флажок Add an interactive sort action to this textbox.
4. Выберите поле, по которому будет выполняться сортировка, и введите выражение.
5. Если таблица содержит группы, вы можете выбрать, какова будет область определения выражения сортировки: конкретная группа или весь диапазон данных.
6. Щелкните на кнопке OK, чтобы применить внесенные изменения.

Повторите описанный процесс для всех столбцов, возможность интерактивной сортировки по которым вы желаете предоставить пользователю.

Добавление в таблицу групп

В этом разделе будут описаны действия по добавлению в отчет `Products`, созданный ранее, группировки по категориям, а затем по подкатегориям.

1. Откройте окно параметров элемента таблицы и перейдите к вкладке Groups. Обратите внимание на то, что список групп пока пуст.
2. Для создания новой группы щелкните на кнопке Add. Откроется диалоговое окно Grouping and Sorting Properties.
3. Присвойте группе имя `CategoryGroup`.
4. В столбце Expression параметра Group on выберите поле `Category`.
5. Если хотите. чтобы отчет включал карту документа. позволяющую пользователю быстро переходить в пределах отчета, используя элементы древовидного представления,

выберите в параметре Document Map Label поле `Category`. Очень важно выбрать поле, связанное отношением "один к одному", с полем, использованным для группировки. Например, выполните группировку по полю `CategoryID`, а для карты документа используйте поле `Category`.

6. При создании группы верхнего уровня не определяйте значение для параметра Parent Group.
7. С помощью доступных флажков установите разрывы страниц, параметры видимости колонтитулов, а также их режим отображения на страницах.
8. При создании группы вы должны также указать поле, по которому будет сортироваться группа. Для этого перейдите к вкладке Sorting и выберите поле `Category`.
9. Щелкните на кнопке OK, чтобы закрыть диалоговое окно Grouping and Sorting Properties и вернуться в окно параметров таблицы. Обратите внимание на то, что в списке Group List теперь отображается имя группы `CategoryGroup`.
10. Чтобы добавить группу подкатегорий, снова щелкните на кнопке Add и сконфигурируйте группу согласно рис. 46.12. При этом не забудьте определить сортировку по полю `Subcategory`.
11. Щелкните на кнопке OK, чтобы применить изменения.

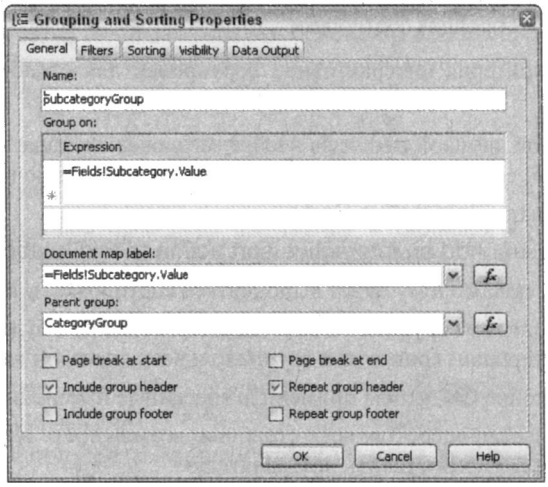


Рис. 46.12. Добавление групп в таблицу выполняется с помощью диалогового окна Grouping and Sorting Priperties

Форматирование таблиц с группами

После создания групп для категорий и подкатегорий таблица будет содержать две новые строки в раскладке каждой из групп. Числа на левой границе строк групп указывают на их порядок. Чтобы узнать имя группы, нужно выделить левую границу и посмотреть на значение параметра Grouping/Sorting.

На рис. 46.13 показаны отформатированные группы, а на рис. 46.14 — выполненный отчет.

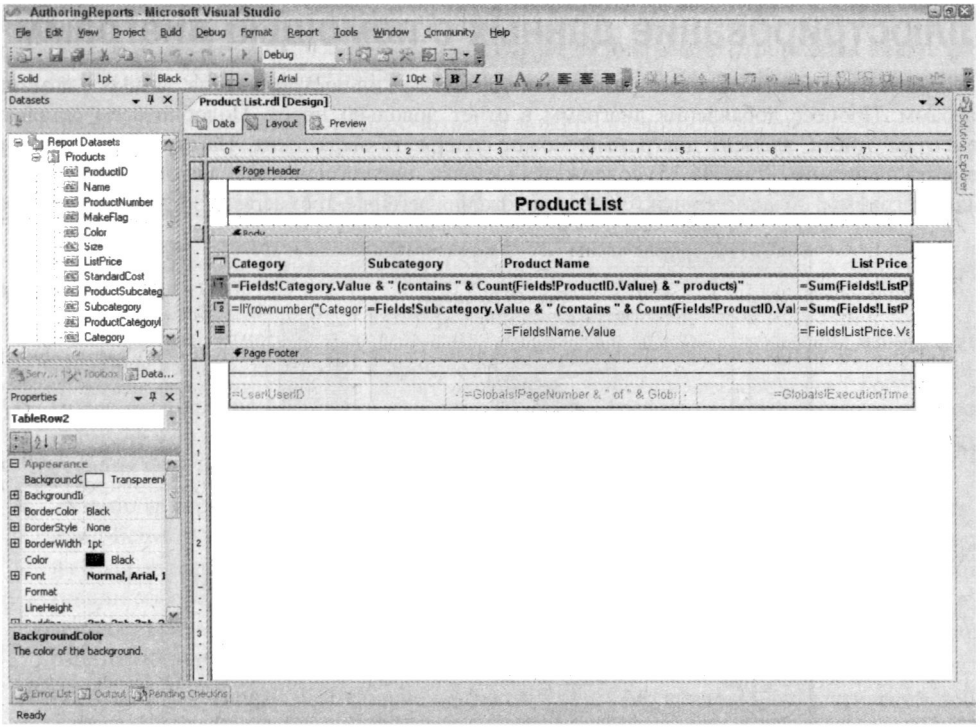


Рис. 46.13. Расширение раскладки таблицы путем добавления групп

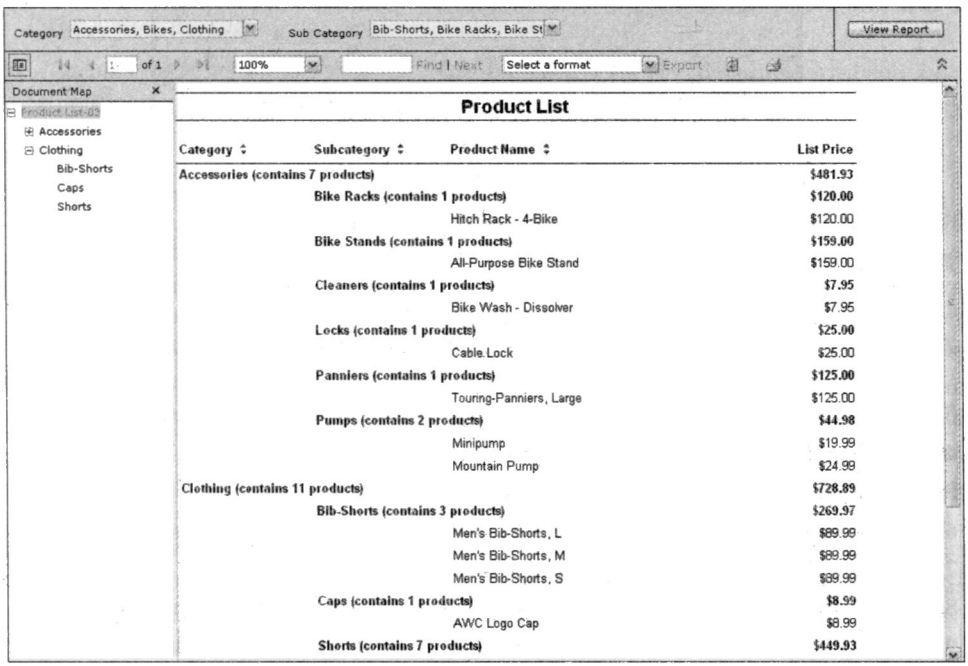


Рис. 46.14. Отчет с группировкой и карта документа

Иллюстрирование данных с помощью диаграмм

Служба отчетности обладает развитыми средствами иллюстрирования данных с помощью диаграмм. Процесс добавления диаграмм в отчет довольно прост. Подобно всем остальным элементам отчета, элемент диаграммы может содержать выражения. На странице параметров элемента диаграммы (рис. 46.15) содержатся вкладки, предназначенные для определения типа и стиля диаграммы, области данных, стиля и положения легенды, трехмерных эффектов и фильтров.

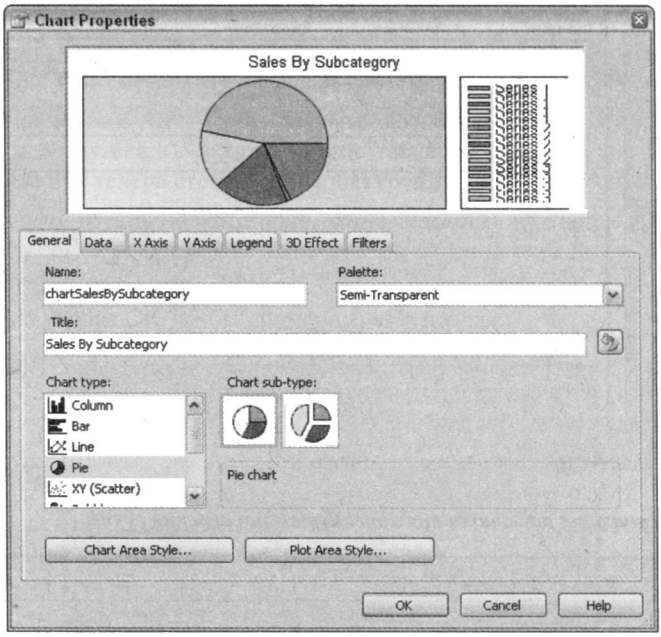


Рис. 46.15. В диалоговом окне Chart Properties *определяются и настраиваются диаграммы*

Добавление в отчет круговой диаграммы

Для добавления в отчет диаграммы выполните следующие действия.

1. Добавьте в отчет элемент `Chart` с панели Toolbox.
2. Щелкните правой кнопкой мыши на элементе диаграммы и выберите в контекстном меню пункт Properties.
3. В списке типов диаграмм выберите тип круговой диаграммы (`Pie Chart`).
4. Введите заголовок диаграммы.
5. Перейдите к вкладке Data.
6. Выберите диапазон данных, которые будут отображаться на диаграмме.
7. Щелкните на кнопке OK, чтобы сохранить изменения.
8. Выделите диаграмму, чтобы отобразить окна данных, значений и категорий.
9. Перетащите поля из окна набора данных в соответствующие им окна данных значений и категорий. Например, если набор данных содержит поле общей суммы по строке

заказа и поле соответствующей категории товара, первое из них перетащите в окно Data, а второе — в окно Series. В этом случае на диаграмме отобразятся общие объемы продаж в разрезе категорий товаров.

Теперь в окне предварительного просмотра вы можете посмотреть полученные результаты.

Пакеты обновлений и служба отчетности

Выход в свет пакетов обновлений SP1 и SP2 внес некоторые коррективы в работу службы отчетности.

- В построителе отчетов стало возможно добавлять в отчеты статические изображения, такие как логотип компании и прочую графику.
- Новое расширение обработки данных предлагает графический конструктор запросов, который можно использовать для доступа к InfoCube и запросам в отчетах, использующих источники данных SAP NetWeaver Business Intelligence.
- Экземпляр сервера отчетности можно интегрировать с Windows SharePoint Services 3.0 или Microsoft Office 2007 SharePoint Server для хранения, защиты, организации доступа и управления элементами сервера отчетности с сайта SharePoint. Эти функции интеграции обеспечивают специальную надстройку Report Services, устанавливаемую в экземпляр SharePoint. В упомянутую надстройку включен также компонент просмотра отчетов Report Viewer Web Part.
- В службу отчетности был включен флажок Select All, который автоматически добавляется при создании списка доступных значений многозначного параметра отчета. Если вы обновили версию до SP1, этот флажок станет недоступным. В пакете обновлений SP2 этот флажок был снова восстановлен. Эта функция поддерживается в SQL Server 2005 Express Edition SP2.
- Для доступа к источнику данных Hyperion System 9.3 BI+ Enterprise Analytics служба отчетности предлагает специальный поставщик Microsoft .NET Data Provider for Hyperion Essbase. Это новое расширение обработки данных обеспечено графическим конструктором запросов, позволяющим создавать запросы MDX. Этот поставщик данных отправляет запросы MDX в Hyperion System 9 BI+ Analytic Provider Services 9.3, который, в свою очередь, создает структуры, необходимые для извлечения данных из хранилища OLAP Hyperion Essbase. Поставщик Microsoft .NET Data Provider for Hyperion Essbase требует установленной системы Hyperion System 9.3 Beta 2. Эта система должна быть установлена на сервере источника данных до того, как хранилище Hyperion Essbase сможет быть использовано в качестве источника данных службы отчетности.
- Поддерживается генерация моделей отчетов из источников данных Oracle версии 9.2.0.3 и более поздних; при этом для создания моделей могут использоваться Report Manager, Management Studio и Model Designer.
- Клиент Oracle должен быть установлен на сервере отчетности и на всех клиентских компьютерах, которые будут осуществлять удаленный доступ к серверу отчетности. Каталог клиента Oracle должен быть включен в системный путь, при этом к данному каталогу должен быть разрешен доступ Windows-службе Report Server и Web-службе Report Server.

Резюме

Служба отчетности в SQL Server 2005 предлагает богатый выбор средств, позволяющих разработчикам отчетов обеспечить конечных пользователей, принимающих бизнес-решения, доступом к интерактивным отчетам. Шаблон проекта Reporting Services Project в программе Visual Studio 2005 содержит мощный инструментарий, облегчающий создание полезных отчетов. Для быстрого создания простых отчетов можно воспользоваться мастером отчетов. Также доступно множество дополнительных средств, позволяющих удовлетворить практически любые потребности в отчетах.

Отчеты могут использовать разнообразные источники данных, при этом каждый отчет может включать данные из нескольких источников. Параметры запросов и отчетов позволяют пользователям управлять критериями отбора, сортировки и фильтрации данных в отчете. С помощью выражений можно управлять практически всеми аспектами отчета, что позволяет для представления данных выполнять сложные вычисления непосредственно в отчете.

Visual Studio 2005 содержит множество новых функций и расширений, некоторые из которых приведены ниже.

- Новый источник данных XML позволяет включать в отчеты данные, содержащиеся в файлах XML, а также возвращаемые Web-службами и приложениями в этом формате.
- Критерии группировки данных в таблице могут содержать параметры и выражения, что позволяет создавать в отчетах динамические группы.
- Многозначные параметры отчета позволяют пользователю выбрать для каждого из параметров отчета несколько значений.
- Теперь можно создавать и внедрять на панель Toolbox пользовательские элементы отчета, что расширяет границы возможного в отчетах.
- В дополнение к списку доступных в отчете функций редактор выражений содержит функцию IntelliSense, являющуюся путеводителем в синтаксисе выражений, а также функции автозавершения инструкций.
- Новая функция интерактивной сортировки позволяет создателю отчета определить критерии сортировки столбцов таблицы и отдать управление процессом сортировки в руки конечному пользователю.

Конструктор отчетов в Visual Studio 2005 содержит множество элементов создания композиции отчета. Элементы списка, таблицы, матрицы и текстового поля обладают большой гибкостью в плане отображения данных, а элемент диаграммы позволяет включать в отчеты визуальное представление данных. В дополнение к управлению данными внутри отчета, параметры и выражения могут использоваться для управления практически всеми визуальными аспектами отчета.

Администрирование отчетов в службе отчетности

ГЛАВА

47

В этой главе...

Развертывание отчетов

Конфигурирование службы отчетности в диспетчере отчетов

Служба отчетности в SQL Server 2005 содержит Web-приложение ASP.NET, которое называется Report Manager (далее — *диспетчер отчетов*). Это приложение предлагает средства, необходимые для администрирования сервера отчетности и развертывания элементов, таких как отчеты, общие источники данных и модели данных. Диспетчер отчетов также содержит интерфейс пользователя, организующий доступ к отчетам, управление ими и их выполнение. По умолчанию диспетчер отчетов размещен в Web-каталоге `http://localhost/reports`.

В настоящей главе будут описаны способы развертывания отчетов и администрирования сервера отчетности с помощью диспетчера отчетов (рис. 47.1). Существуют три стратегии развертывания отчета: с использованием Visual Studio 2005, с использованием диспетчера отчетов и программное развертывание с помощью Web-службы Reporting Services. В этой главе также рассматривается построитель отчетов (Report Builder), позволяющий конечному пользователю самому создавать отчеты.

Развертывание отчетов службы отчетности

В предыдущей главе речь шла о том, как создавать отчеты в Visual Studio 2005. Здесь же мы исследуем стратегии, используемые при развертывании отчетов на сервере отчетности.

Развертывание отчетов с помощью Visual Studio 2005

Развертывание отчетов в Visual Studio 2005 требует некоторых конфигурационных настроек в проекте службы отчет-

ности. После этого отчеты (и другие ресурсы, такие как источники данных и изображения) могут быть развернуты в индивидуальном порядке; также может быть развернут и проект в целом.

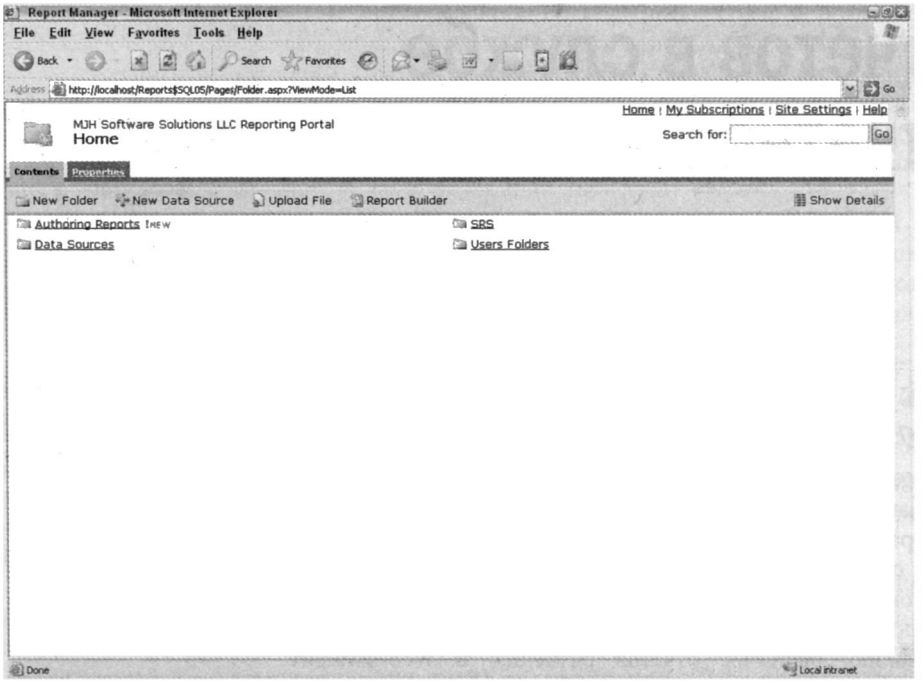


Рис. 47.1. Диспетчер отчетов входит в состав службы отчетности. Он предоставляет пользователям доступ к отчетам, позволяет управлять ими и выполнять их. Администраторам он позволяет развертывать отчеты, управлять системой безопасности и конфигурировать сервер отчетности

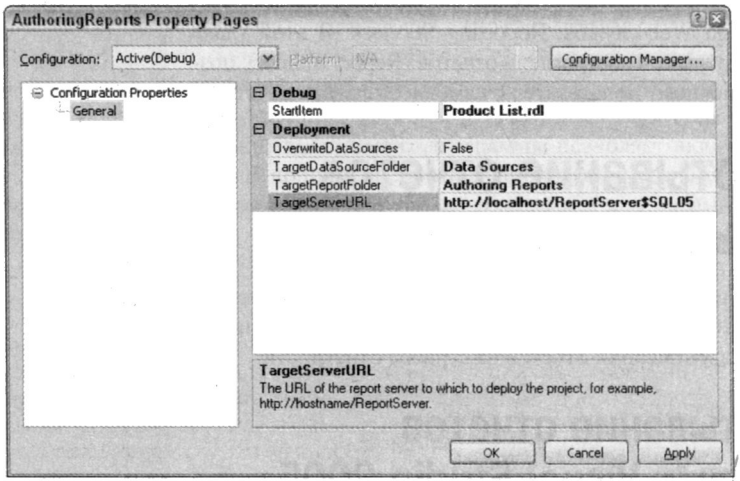


Рис. 47.2. Используйте настройки в окне Property *проекта службы отчетности программы Visual Studio 2005 для конфигурирования параметров развертывания*

Для конфигурирования параметров развертывания проекта службы отчетности откройте страницу Poroperties проекта, щелкнув правой кнопкой мыши на названии проекта и выбрав в контекстном меню пункт Properties. На рис. 47.2 показана страница параметров проекта службы отчетности, а в табл. 47.1 описаны параметры развертывания, доступные в проекте.

Таблица 47.1. Параметры развертывания проекта службы отчетности

Параметр	Описание
`OverwriteDataSources`	Если вы хотите заменить источники данных на сервере отчетности источниками из проекта, установите для этого параметра значение `True`. По умолчанию установлено значение `False`, предотвращающее замещение источника данных. Это может оказаться полезным, если источник данных, используемый в процессе разработки, отличается от используемого на сервере отчетности
`TagretDataSourceFolder`	Путь к папке, в которой следует разместить общие источники данных. Использование этого параметра позволяет хранить определения общих источников данных централизованно и применять их в отчетах, находящихся в разных папках (или проектах в процессе разработки). Использование общего подключения минимизирует задачи администрирования объектов источников данных. Если не установить этот параметр, то общие источники данных будут развернуты в папке, указанной в параметре `TargetReportFolder`, т.е. в месте развертывания отчета (что не рекомендуется)
`TargetReportFolder`	Путь к папке, в которой следует развернуть отчеты. В Visual Studio вам следует создать проект для каждой из папок, в которой вы собираетесь развертывать отчеты. Например, если вы хотите развернуть два отчета в папках `Sales` и `Customers`, создайте в Visual Studio 2005 два проекта и укажите в их настройках соответствующие папки. Для управления размещением отдельных отчетов их можно перемещать между проектами. Также можно использовать переменную `TargetReportFolder` для развертывания подпапок на сервере отчетности. Например, установка для этого параметра значения `Sales\Regional` приведет к созданию удобного места для хранения отчетов о продажах в регионах
`TargetServerURL`	Здесь указывается адрес URL сервера отчетности, на котором будет выполняться развертывание. По умолчанию здесь указано местонахождение локального сервера отчетности: `http://localhost/ReportServer`. Если именем экземпляра SQL Server является `SQL05`, то локальным сервером будет `http://localhost/ReportServer$SQL05`



Проекты службы отчетности в Visual Studio 2005 теперь содержат параметры папки целевого источника данных и целевой папки отчетов. В версии службы отчетности в SQL Server 2000 существовал только параметр `TargetFolder`, содержащий папку, в которую устанавливались все объекты проекта. Теперь совместно используемые источники данных могут развертываться в общей папке и использоваться множеством отчетов, размещенных в разных папках. Это позволяет минимизировать усилия, необходимые для администрирования объектов источников данных на сервере отчетности, а также упрощает сам процесс развертывания.

Развертывание одного отчета или источника данных

Когда в проекте службы отчетности настроены параметры развертывания, отчет или источник данных может быть развернут с помощью щелчка правой кнопкой мыши на его имени и выбора в контекстном меню пункта Deploy. О ходе процесса развертывания информирует индикатор в строке состояния окна Visual Studio 2005. Если во время развертывания были обнаружены ошибки, то они отобразятся в окне Error List.

Этот метод развертывания объектов на сервере отчетности будет удобным при обновлении на сервере отчетности только избранных, а не всех объектов.

Развертывание проекта сервера отчетности

Для развертывания всех объектов проекта службы отчетности вначале проверьте, правильно ли сконфигурированы его параметры, после чего щелкните правой кнопкой мыши на проекте и выберите в контекстном меню пункт Deploy. Следует отметить, что во избежание замещения объектов источника данных на сервере можно использовать параметр проекта `OverwriteDataSources`. Это очень важная возможность, поскольку источники данных на сервере разработки и в эксплуатационной среде часто требуют разного конфигурирования.

Развертывание отчетов с помощью диспетчера отчетов

Несмотря на то что Visual Studio 2005 предлагает простой способ развертывания отчетов на сервере отчетности, этот метод не является единственным. Существует возможность развертывания и конфигурирования отдельных объектов службы отчетности с помощью диспетчера отчетов (Report Manager). В диспетчер отчетов включены функции, позволяющие создавать новые папки и источники данных; также в нем реализована возможность выгрузки и обновления определений отчетов (файлов `.rdl`), источников данных отчетов (файлов `.rds`), а также всех других типов файлов, которые вы хотите сделать доступными на сервере отчетности (например, документы PDF и Word, презентации Power Point, рабочие книги Excel и т.д.).

Для развертывания отчета с помощью диспетчера отчетов выполните следующие действия.

1. Откройте приложение Report Manager в Web-браузере. По умолчанию оно размещено по адресу `http://localhost/reports`.
2. Выберите папку, в которой собираетесь развернуть отчет.
3. Щелкните на кнопке Upload File.
4. Введите путь к файлу или найдите его с помощью кнопки Browse.
5. Введите имя отчета.
6. Если хотите переписать существующий отчет с таким же именем, установите флажок Overwrite item, если таковой существует.
7. Щелкните на кнопке OK, чтобы выполнить выгрузку на сервер файла и вернуться к содержимому папки. Новый отчет теперь отобразится в списке и будет помечен как новый, чтобы привлечь внимание пользователя.

Желательно попытаться выполнить отчет и убедиться, что источник данных задан корректно. Вам может потребоваться использовать ссылку Data Source во вкладке Properties отчета, чтобы выбрать общий источник данных или определить для отчета дополнительный источник. Во вкладке Properties вы можете также проверить и другие ссылки, чтобы установить значения по умолчанию для параметров, сконфигурировать выполнение отчета, а также настройки его журнала и защиты.

Развертывание отчетов программным путем с использованием Web-службы Reporting Services

Web-служба Reporting Services предлагает методы развертывания отчетов и позволяет создавать и использовать прикладные программы для выполнения этих задач. Напомним, что диспетчер отчетов является всего лишь примером интерфейса пользователя ASP.NET, который использует Web-службу Reporting Services для управления сервером отчетности. Все функции, доступные в диспетчере отчетов, доступны также и для разрабатываемого приложения, использующего Web-службу Reporting Services.

Чтобы создать такое приложение, создайте в Visual Studio 2005 проект и добавьте в него ссылку на Web-службу Reporting Services. По умолчанию эта Web-служба на локальном компьютере находится по адресу `http://localhost/ReportServer/ReportService.asmx`. После добавления этой ссылки вы получите доступ к классам, позволяющим пользовательскому приложению выполнить на сервере отчетности практически любые операции, в том числе развертывание и выполнение отчетов. Класс службы отчетности `OnTheWebService` содержит методы, позволяющие создавать (и удалять) папки, отчеты. источники данных, расписания и подписки, а также выполнять множество других операций на сервере отчетности. Он также содержит методы для представления отчетов в любом из форматов, перечисленных в табл. 47.6.

Конфигурирование службы отчетности в диспетчере отчетов

Диспетчер отчетов является Web-приложением, входящим в состав SQL Server 2005 Reporting Services. Он реализует функции администрирования сервера отчетности, а также функции доступа к отчетам и их выполнения. Диспетчер отчетов является интерфейсом пользователя ASP.NET, построенным на Web-службе Reporting Services. По умолчанию диспетчер отчетов размещен по адресу `http://localhost/reports`. В настоящем разделе будет описано, как использовать это приложение для администрирования службы отчетности.

Конфигурирование настроек сайта службы отчетности

Страница Site Settings (рис. 47.3) содержит настройки, позволяющие создавать пользовательские папки, управлять журналом отчетов и установками предельного времени ожидания выполнения, а также протоколированием выполнения отчета. На этой странице также содержатся ссылки на конфигурирование системы безопасности сайта, на создание и управление общими расписаниями и управление заданиями. Для конфигурирования настроек сайта на сервере отчетности щелкните на ссылке Site Settings в заголовке диспетчера отчетов.

Включение личных папок

По умолчанию возможность создания личных папок (My Reports) отключена. Включение этой функции открывает для каждого пользователя личную папку `My Reports`, в которой он может публиковать свои отчеты, создавать связанные отчеты и управлять их содержимым. Каждый пользователь получает в свое распоряжение отдельную папку, подобную папкам `Мои документы` в операционной системе Windows. Служба отчетности содержит роль по умол-

чанию `My Reports`, обеспечивающую защищенный доступ к одноименной папке. Позже в этой главе будет показано, как настраивать существующие роли и создавать новые; сейчас же отметьте для себя, что вы можете определить роль, применимую к папке `My Reports`.

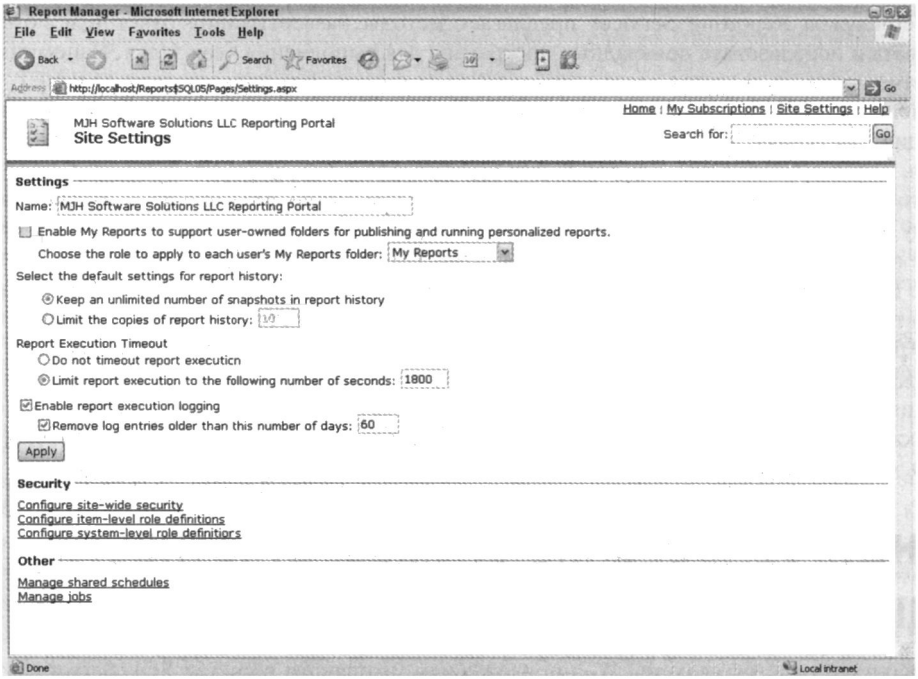


Рис. 47.3. Страница Site Settings диспетчера отчетов используется для администрирования настроек уровня сайта

Включение параметра My Reports приводит к образованию личного пространства пользователя, где он может сохранять отчеты, созданные с помощью построителя отчетов (Report Builder). Этот новый инструмент, ставший доступным в службе отчетности SQL Server 2005, позволяет конечным пользователям создавать собственные отчеты на основе предопределенных моделей данных.



Служба отчетности в SQL Server 2005 содержит построитель отчетов, позволяющий конечным пользователям создавать собственные отчеты. Сотрудники компаний, ответственные за принятие производственных решений, могут воспользоваться этим великолепным инструментом для доступа к данным и их анализа с целью своевременного и более точного принятия решений. Одновременно это приложение минимизирует сложности, связанные с созданием отчетов. В организационном аспекте процесс выглядит следующим образом. Проектированием, разработкой и администрированием моделей для построителя отчетов занимается технический персонал информационного подразделения компании, после чего он обеспечивает готовыми моделями рядовых сотрудников — т.е. непосредственных потребителей информации.

Конфигурирование журнала отчетов

Поддержка журнала отчетов включена по умолчанию; при этом можно как ограничить число хранимых снимков отчетов, так и не устанавливать каких-либо пределов. По умолча-

нию число снимков ограничено, однако этот параметр может быть изменен на уровне конкретных отчетов. Чтобы изменить этот параметр для какого-либо отчета, откройте для него вкладку Properties, а затем щелкните на ссылке History. У выбранного отчета вы можете изменить следующие настройки.

- Включить ручной режим создания журнала отчетов (этот режим установлен по умолчанию). Для этого во вкладке History отчета щелкните на ссылке New Snapshot.
- Сохранить снимок при каждом выполнении отчета (по умолчанию отключено).
- Использовать специфичное для отчета или общее расписание для добавления снимков отчетов в журнал (по умолчанию отключено).
- Выбрать количество хранимых снимков отчетов. Если выбрать вариант Use Default Settings (он установлен по умолчанию), то к отчету будут применяться соответствующие параметры всего сайта; в противном случае они будут замещены значением, установленным в контексте данного отчета.

Конфигурирование настроек выполнения отчетов

Аналогично настройкам журнала отчетов, установленным по умолчанию, параметр Report Execution Timeout позволяет установить время ожидания выполнения запроса на уровне всего сайта (по умолчанию этот лимит составляет 1800 секунд, т.е. 30 минут). На странице Site Settings вы можете изменить время ожидания или вообще снять этот лимит (опция `Do not timeout report execution`). Чтобы установить лимит времени ожидания выполнения на уровне отдельного отчета, перейдите к вкладке Properties отчета и щелкните на ссылке Execution. На открывшейся странице вы можете выбрать режим запуска самого последнего отчета или открытия готового снимка последнего отчета. Здесь же для выбранного отчета можно определить предельное время ожидания выполнения.

Если установлен флажок Enable report execution logging, то в таблицу `ExecutionLog` базы данных `ReportServer` записывается строка при каждом выполнении отчета. Несмотря на то что в построителе отчетов отсутствует интерфейс доступа к этим записям, потенциально они доступны. Если данная информация покажется вам полезной, вы можете создать собственный интерфейс доступа к ней. Страница Site Settings также позволяет удалять записи из журнала отчетов, которые были сформированы раньше заданного количества дней от текущей даты.

Администрирование системы безопасности

Модель системы безопасности службы отчетности использует Active Directory для предоставления доступа к диспетчеру отчетов и его отдельным элементам (папки, отчеты, общие источники данных и т.д.). Администрирование системы безопасности осуществляется путем назначения пользователей и их групп ролям. Роли содержат избранные задания, позволяющие выполнять в приложении диспетчера отчетов определенные действия. Существуют два типа предопределенных ролей и заданий: системного уровня и уровня элементов.

Роли системного уровня

При установке сервера отчетности создаются две роли системного уровня: `System Administrator` и `System User`. В табл. 47.2 приведены задания, выполнение которых разрешено этим ролям по умолчанию. Для изменения предопределенных ролей и создания новых щелкните на ссылке Configure System-level Role Definition на странице Site Settings.

Таблица 47.2. Системные роли, созданные по умолчанию

Задание	System Administrator	System User
Выполнение определений отчетов	X	X
Генерация событий		
Управление заданиями	X	
Управление параметрами сервера отчетности	X	
Управление системой безопасности сервера отчетности	X	
Управление ролями	X	
Управление общими расписаниями	X	
Просмотр параметров сервера отчетности		X
Просмотр общих расписаний		X

По умолчанию определения ролей можно изменять, выбрав соответствующую роль на странице System Roles. При необходимости можно создать дополнительные роли, для чего следует щелкнуть на кнопке New Role страницы System Roles и выбрать задачи, разрешенные для новой роли.

Открытие системного доступа пользователям и группам

По умолчанию группа `BUILTIN\Administrators` назначена роли `System Administrator`. Для открытия системного доступа для других пользователей и групп Active Directory выполните следующие действия.

1. Щелкните на ссылке Configure Site-wide Security страницы Site Settings.
2. Щелкните на кнопке New Role Assignment открывшейся страницы System Role Assignment.
3. Введите имя пользователя или группы, например `myDomain\jdoe` или `myDomain\SRSAdministrators`.
4. Выберите одну роль или более, которые будут назначены пользователю или группе.
5. Щелкните на кнопке OK, чтобы сохранить изменения.



Очень важно поддерживать согласованность независимо от подхода, используемого для назначения разрешений к диспетчеру отчетов и его элементам. При выборе метода управления и способов назначения доступа внимательно оцените рабочую среду в целом и определите, каким образом будет управляться доступ: на уровне групп службы активных каталогов Active Directory или отдельных пользователей. Если вы уже затратили усилия на формирование групп пользователей Active Directory в своей организации, то лучше воспользоваться готовыми группами и администрировать доступ к серверу отчетности на их уровне.

Например, если в Active Directory существует группа, созданная для бухгалтеров, для этой группы можно создать отдельную системную роль, открывающую доступ всем ее членам к серверу отчетности. Возможно, при этом вы захотите создать отдельную папку `Accounting` и поместить в нее все бухгалтерские отчеты, при этом открыв к ней доступ только сотрудникам бухгалтерии. При создании такой папки скорректируйте назначения роли, чтобы к папке имели доступ только члены группы бухгалтеров.

Независимо от используемой стратегии (на уровне пользователей или групп), поддержание единообразия позволит в будущем уменьшить затраты на поддержку системы и разрешение ошибок.

Роли уровня элементов

Роли уровня элементов используются для управления заданиями, доступными для работы с папками, отчетами, общими источниками данных, моделями и прочими ресурсами диспетчера отчетов. В табл. 47.3 приведены роли уровня элементов, создаваемые по умолчанию при установке сервера отчетности. Для доступа к ролям уровня элементов щелкните на странице Site Settings на ссылке Configure Item-level role definitions. В качестве альтернативы можно щелкнуть на ссылке Security во вкладке Property любого элемента, находящегося ниже корневого уровня в диспетчере отчетов, после чего щелкнуть на ссылке Edit, чтобы отредактировать текущие установки, или на кнопке New Role Assignment.

Таблица 47.3. Роли уровня элементов, заданные по умолчанию

Задание	Content Browser	Manager	My Reports	Report Publisher	Builder
Получение отчетов		X			X
Создание связанных отчетов		X	X	X	
Управление всеми подписками		X			
Управление источниками данных		X	X	X	
Управление папками		X	X	X	
Управление отдельными подписками	X	X	X		X
Управление моделями		X		X	
Управление журналом отчетов		X	X		
Управление отчетами		X	X	X	
Управление ресурсами		X	X	X	
Управление системой безопасности на уровне элементов		X			
Просмотр источников данных		X	X		
Просмотр папок	X	X	X		X
Просмотр моделей	X	X			X
Просмотр отчетов	X	X	X		X
Просмотр ресурсов	X	X	X		X

Создание дополнительных ролей уровня элементов выполняется точно так же, как и создание ролей системного уровня. Необходимые для этого действия описаны в следующем разделе.

Управление защитой на уровне элементов

По умолчанию все ресурсы уровня элементов наследуют установки системы защиты своих предков. Если настройки системы безопасности элемента не изменялись (т.е. продолжают наследоваться), то на странице Security этого элемента будет находиться кнопка Edit Item Security. После изменения настроек системы безопасности (т.е. разрушения связки наследования) на странице Security элемента будет открываться двухуровневый доступ к пользова-

телям и группам. Для выполнения этих задач пользователь должен принадлежать к роли с разрешенной задачей настройки системы безопасности отдельных элементов. Выполните следующие действия.

1. Во вкладке Properties элемента, параметры системы безопасности которого собираетесь изменить, щелкните на ссылке Security.
2. Если элемент все еще наследует настройки системы безопасности своего предка, щелкните на кнопке Edit Item Security. Откроется окно, предупреждающее, что после изменения настроек защиты элемента будет разорвана связь наследования. Чтобы продолжить, щелкните на кнопке OK. Обратите внимание на то, что теперь вы можете также удалить назначения ролей, установленные у предка.
3. Щелкните на кнопке New Role Assignment.
4. Введите группу или имя пользователя в форме *домен\учетная_запись*.
5. Выберите одну или несколько ролей, которые следует назначить пользователю или группе. Если вам нужно просмотреть или отредактировать задачи, связанные с ролью, щелкните на соответствующем имени роли. На этой странице вы также можете создать новую роль.
6. Щелкните на кнопке OK, чтобы сохранить новые назначения.

Для того чтобы разрешить доступ пользователя к ресурсу уровня элемента, вы должны иметь разрешения системного уровня. Также проверьте, имеют ли пользователи и группы, которым открывается доступ на уровне элементов, доступ на системном уровне.

Модифицированные настройки системы безопасности элемента автоматически применяются и ко всем дочерним элементам, наследующим эти настройки. Чтобы восстановить наследование настроек элементом с измененными параметрами системы безопасности, щелкните на кнопке Revert to Parent Security. В отобразившемся предупреждении о том, что существующие настройки элемента будут заменены настройками предка, следует щелкнуть на кнопке OK.

Работа со связанными отчетами

Связанный отчет представляет собой ярлык фактического отчета. Это позволяет конфигурировать параметры, выполнение, ведение журнала и систему безопасности отчета, независимо от параметров фактического отчета. Если на сервере отчетности разрешено создание пользовательских папок для публикации и запуска персонализированных отчетов, пользователи могут создать связанный отчет и сконфигурировать его параметры в соответствии со своими текущими потребностями.

Связанные отчеты также реализуют возможность администрирования системы безопасности отчета, при этом ограничивая доступность параметров для пользователей, групп и ролей. В качестве примера можно привести отчет об объемах продаж по регионам, содержащий необязательный параметр, ограничивающий обрабатываемые данные одним конкретным регионом. Некоторым пользователям должна быть предоставлена возможность выполнения отчета по всем регионам, в то время как остальным должен быть открыт доступ только к информации по их региону. В этом случае вы можете ограничить доступ к фактическому отчету только кругом лиц, которым разрешен просмотр информации по всем регионам. После этого можно создать связанные отчеты для каждого из регионов и ограничить доступ к ним пользователям, обладающим конкретными привилегиями. В каждом связанном отчете скройте параметр региона и установите для него значение по умолчанию, соответствующее конкретному региону. Затем вы можете создать папки для каждого региона и сохранить в них связан-

ные отчеты. Таким образом, система безопасности может управляться на уровне папок, содержащих множество связанных отчетов, сохраненных в них, устраняя необходимость в администрировании системы безопасности отдельно в каждом связанном отчете.

Создание связанных отчетов

Для создания связанного отчета перейдите к вкладке Properties отчета, для которого хотите создать ссылку, и щелкните на кнопке Create Linked Report. Присвойте связанному отчету имя и введите его описание и путь к месту размещения. Щелкните на кнопке OK. Для администрирования параметров, процедур выполнения, журнала и системы безопасности связанного отчета перейдите к его вкладке Properties. Для связанного отчета также можно создавать подписки и исторические снимки.

Для упрощения анализа зависимостей и управления связанными отчетами желательно включить в имя или описание связанного отчета имя фактического отчета, на котором он построен. В этом случае, введя имя фактического отчета в критерии поиска, поле которого находится в заголовке диспетчера отчетов, можно получить список всех связанных с ним отчетов.

Вооружаемся подписками

Возможность создания подписок на отчеты представляет собой исключительно важную функцию службы отчетности, которая автоматизирует доставку отчетов. Служба отчетности поддерживает оба типа подписки: принудительную и по требованию. Например, при выполнении отчетов по расписанию подписка может рассылать электронные сообщения с вложенным в них содержимым отчетов (принудительная подписка), с отчетом, находящимся непосредственно в теле сообщения (принудительная подписка), или со ссылкой на отчет, хранящийся на сервере сообщений (подписка по требованию). В качестве альтернативы отчет может быть сохранен в некоторой общедоступной папке, к которой имеют доступ пользователи. В этом случае они могут извлечь или экспортировать данные из файлов, содержащихся в этой папке.

Подписки на отчеты требуют наличия запущенной службы SQL Server Agent на компьютере службы отчетности. Эта служба выполняет задания и мониторинг сервера баз данных, генерирует предупреждения и позволяет автоматизировать некоторые задачи администрирования сервера отчетности. Перед созданием подписки убедитесь, что эта служба запущена. К тому же для успешного создания подписки на отчеты должна быть сохранена регистрационная информация, необходимая для подключения к источнику данных отчета.

Выделите отчет, на который хотите подписаться, и щелкните на кнопке New Subscription в заголовке элемента отчета. На рис. 47.4 показаны параметры, доступные для создания подписки, доставляемой по электронной почте. Вы можете сконфигурировать параметры доставки и обработки подписки, ее расписание, а также значения параметров отчета. Более подробно о каждой из этих настроек мы поговорим в следующем разделе, в котором будет показано, как создавать подписки, управляемые данными. Такие подписки позволяют устанавливать эти параметры, используя результаты запроса.

Создание подписки, управляемой данными

Подписка, управляемая данными, позволяет устанавливать параметры отчета и его доставки динамически, на основе результатов некоторого запроса. Это отличный способ организации доставки отчетов и управления его содержимым. Например, вместо создания связанных отчетов для каждого из регионов (см. предыдущий раздел) отчет, управляемый данными,

может получать список рассылки и соответствующих подписчиков из регионов из некоторого источника данных, после чего автоматически генерировать и отправлять отчеты с содержимым, уместным для конкретного получателя.

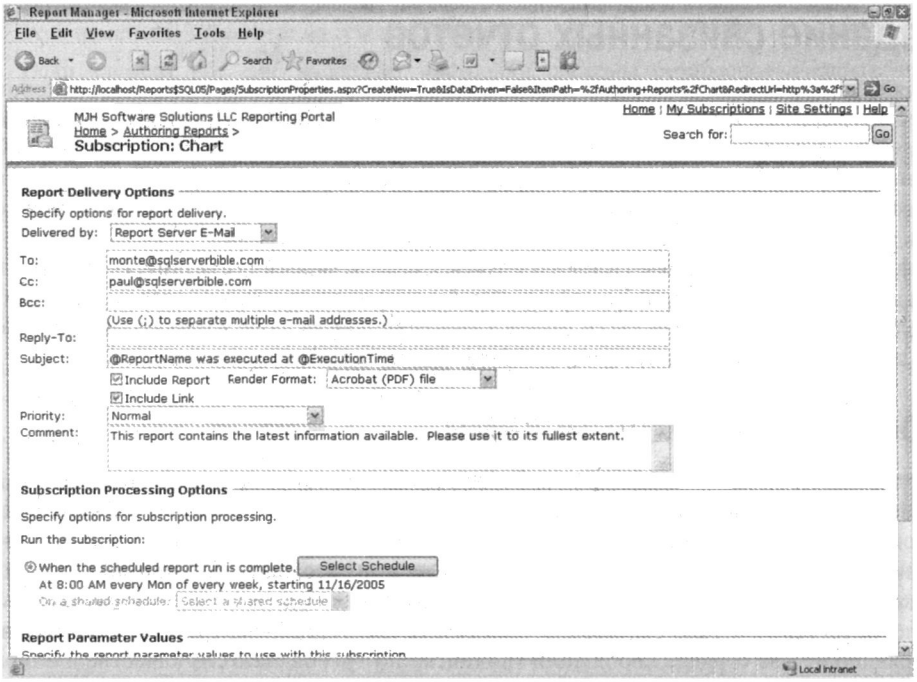


Рис. 47.4. С помощью конфигурирования этих настроек создается подписка на отчет, доставляемый по электронной почте

Для создания подписки, управляемой данными, перейдите на вкладку Subscriptions отчета и щелкните на кнопке New Data-driven Subscription. Вы получите возможность последовательно пройти процесс создания подписки, управляемой данными. Подписки, управляемые данными, можно отправить по электронной почте или поместить в файлы, к которым открыт общий доступ. В любом случае вы должны указать источник данных, содержащий динамические данные для отчета, и написать запрос к нему, возвращающий соответствующие значения.

На рис. 47.5 показаны параметры, предназначенные для определения команд или запросов, возвращающих данные для подписки, управляемой данными. Эти данные могут быть доступны из множества источников данных, включая Microsoft SQL Server, Oracle и XML. Возвращаемые командой или запросом значения могут быть использованы для выполнения отчета, как показано на рис. 47.6.

В дополнение к динамическому назначению параметров доставки отчета, поля запроса могут поставлять значения и для параметров самого запроса. Это позволяет управлять содержимым отчетов на уровне подписчиков. В табл. 47.4 приведены параметры доставки, доступные для подписки, организованной средствами электронной почты, а в табл. 47.5 приведены параметры доставки, доступные для подписок, организованных на основе общедоступных файлов.

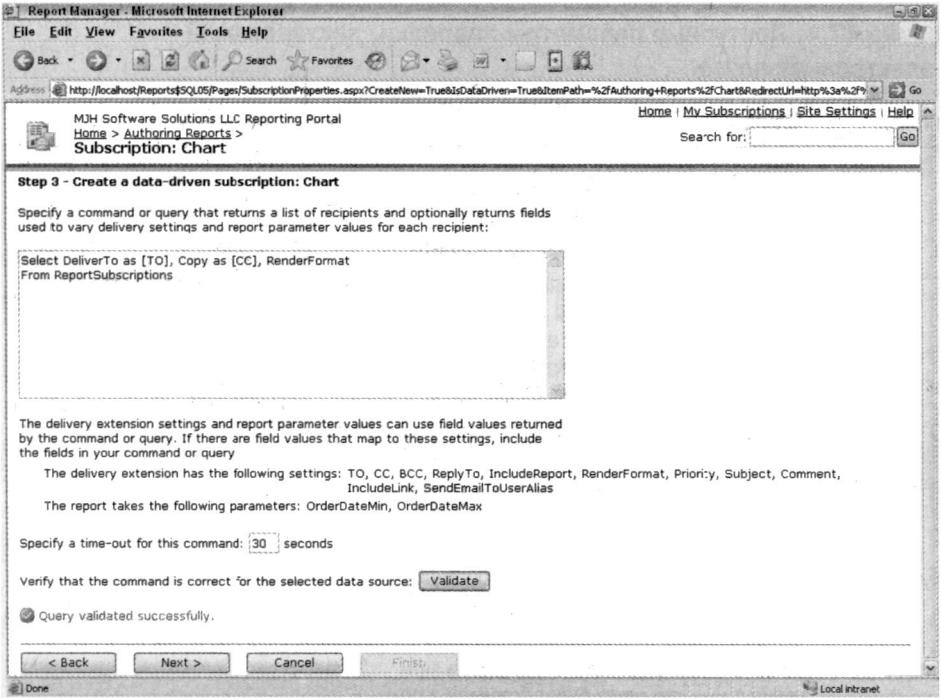


Рис. 47.5. Функция подписки, управляемой данными, позволяет организовать формирование и доставку отчетов пользователям на основании данных из другого источника

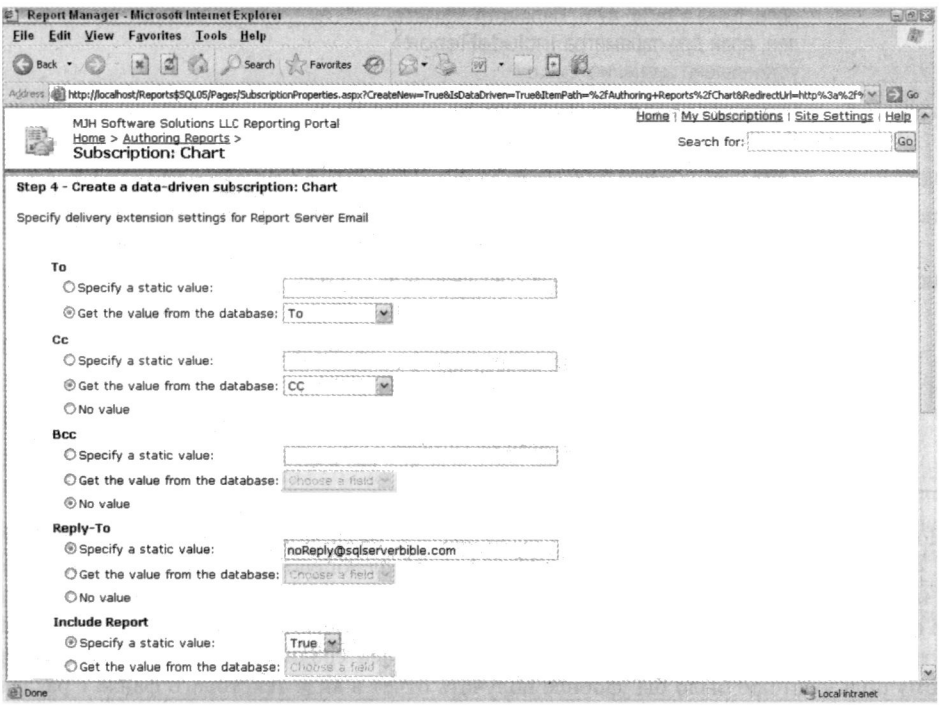


Рис. 47.6. Для управления выполнением отчета вы можете ввести статические данные или использовать значения из базы данных

Таблица 47.4. Доступные параметры подписки, доставляемой по электронной почте

Имя поля	Описание	Пример значения
TO	Список адресов электронной почты, по которым будет отправляться отчет. В качестве разделителей адресов используются точки с запятой. Обязательный параметр	`myself@xyz.com;myboss@xyz.com`
CC	Список адресов электронной почты, по которым будут отправляться копии отчета. В качестве разделителей адресов используются точки с запятой. Необязательный параметр	`mycoworker@xyz.comn`
BCC	Список адресов электронной почты, по которым будут доставляться скрытые копии отчета. В качестве разделителей адресов используются точки с запятой. Необязательный параметр	`mysecretinformet@xyz.com`
ReplyTo	Адрес электронной почты, по которому будут отправляться ответы. Необязательный параметр	`reportReplies@xyz.com`
IncludeReport	Если установить для этого параметра значение `True`, то отчет будет вставлен в сообщение. Для управления форматом используется параметр RenderFormat	`True`
RenderFormat	Формат отчета. Допустимые значения перечислены в табл. 47.6. Параметр обязателен, если для параметра IncludeReport установлено значение `True`	`PDF`
Priority	Приоритет электронного сообщения: `High`, `Medium` или `Low`	`High`
Subject	Тема электронного сообщения	`Отчет о продажах`
Comment	Текст, включаемый в тело электронного сообщения	`Отчет о продажах за день. Пожалуйста, ознакомьтесь`
IncludeLink	Если для этого параметра установлено значение `True`, в тело сообщения добавляется ссылка на отчет, находящийся на сервере отчетности. Следует заметить, что это ссылка на фактический отчет с параметрами выполнения, используемыми для данной подписки, а не ссылка на готовый снимок отчета	`True`

Подписки могут генерировать отчет во множестве различных форматов, перечисленных в табл. 47.6. Это обеспечивает большую гибкость, позволяющую удовлетворить разные подходы к использованию отчетов. Например, одному пользователю было бы удобнее получать отчет в формате PDF, сохраняющем нетронутым все использованное в отчете форматирование; при этом файл может быть без проблем распространен среди сотрудников. В то же время другому пользователю было бы удобнее получать отчет в виде текстового файла с разделите-

лями (CSV) для импорта в другую систему. Форматы Excel и CSV удобны для пользователей, желающих работать с данными в программе Excel; при этом в первом случае программа попытается восстановить исходное форматирование отчета, в то время как во втором данные отчета будут импортированы без какого-либо форматирования.

Таблица 47.5. Доступные параметры подписки, организованной с помощью общедоступных файлов

Имя поля	Описание	Пример значения
FILENAME	Имя файла, который будет записан в общую папку	`MyReport_1`
FILEEXTN	Если установить для этого параметра значение `True`, то к имени файла будет добавлено расширение, соответствующее заданному формату вывода	`True`
PATH	Путь UNC к общей папке, в которую будет записан файл	`\\`*`компьютер\общая_папка*
RENDER_FORMAT	Формат отчета. Список доступных форматов приведен в табл. 47.6	`PDF`
USERNAME	Реквизит пользователя, необходимый для доступа к общему ресурсу	*`домен\пользователь`*
PASSWORD	Реквизит пароля, необходимый для доступа к общему ресурсу	*`пароль`*
WRITEMODE	Допустимые значения: `None`, `AutoIncrement` (Автоматическое дополнение) и `OverWrite` (Переписать)	`AutoIncrement`

Таблица 47.6. Доступные форматы отчета

Значение	Описание
`MHTML`	Web-архив
`HTML3.2`	Web-страница, доступная для большинства браузеров
`HTML4.0`	Web-страница для Internet Explorer 5.0 и более поздних версий программы
`CSV`	Файл с разделителями
`EXCEL`	Формат файлов Excel
`PDF`	Формат файлов Adobe Acrobat
`XML`	Файл XML с данными отчета
`IMAGE`	Графический файл TIFF

Подписки, управляемые данными, позволяют использовать такие же параметры расписания и триггеров, как и обычные подписки. После создания подписки, управляемой данными, она появится в списке подписок на странице My Subscriptions. На этой странице можно просмотреть информацию о подписках, включая тип триггера, дату и время последнего запуска, состояние подписки, а также отредактировать подписку.

Резюме

Для развертывания отчетов на сервере отчетности доступны три метода. Каждый метод служит конкретному назначению и занимает свое место в цикле разработки проекта. Развертывание отчетов с помощью Visual Studio 2005 позволяет развернуть весь проект за одну операцию. Использование диспетчера отчетов в этом процессе обеспечивает бо́льшую гибкость, позволяя пользователям и администраторам управлять конфигурацией развертывания и параметрами самого сервера отчетности, не привлекая средства Visual Studio 2005. В качестве альтернативы клиентские приложения могут использовать Web-службу Reporting Services для автоматизации развертывания отчетов.

Диспетчер отчетов предоставляет конечным пользователям изрядную долю функций службы отчетности, а администраторам — средства эффективного управления сервером отчетности. С помощью диспетчера отчетов администраторы могут без труда управлять содержимым сервера отчетности, системой безопасности и автоматизировать доставку отчетов конечным пользователям.

Анализ данных в Excel и Data Analyzer

ГЛАВА

48

В этой главе...

Создание сводных таблиц Excel, использующих многомерные источники данных

Создание сводных таблиц Excel, использующих реляционные источники данных

Создание сводных диаграмм

Представление реляционных данных с помощью диапазонов данных Excel

Анализ многомерных данных в Data Analyzer

Служба отчетности реализует один из методов создания отчетов, выявляющих тенденции, исключения и прочие важные аспекты данных, хранимых в SQL Server. Отчеты могут создаваться с определенным уровнем интерактивности, однако даже интерактивные отчеты предлагают лишь базовый уровень анализа данных.

Преимуществом анализа данных является возможность выявить неочевидные тенденции и взаимосвязи и взглянуть на данные и их комбинации с нетрадиционной точки зрения. Часто такой анализ является прелюдией к раскрытию данных или даже его составной частью.

В этой главе будет описан анализ данных с использованием программ Excel и Data Analyzer компании Microsoft. Программа Excel предлагает сводные таблицы и сводные диаграммы как для реляционных, так и для многомерных данных, использующие метод перетаскивания для интерактивного формирования представлений. Также Excel допускает формирование именованных диапазонов данных, связующих электронные таблицы и реляционные данные. Data Analyzer концентрируется на многомерных данных, предлагая их табличное и графическое представления.

Интерес к анализу данных в организациях обычно сфокусирован на относительно небольшом их подмножестве, при этом большая часть сотрудников удовлетворяет свой интерес отчетами, создаваемыми другими работниками. Тех, кто заинтересован в анализе данных, объединяет следующее:

- они понимают ценность данных в успехе своей работы (или чувствуют себя беспомощными, если данные отсутствуют);
- они способны создавать средства автоматизации своей работы (например, проблемно-ориентированные электронные таблицы);

- они в общих чертах представляют себе анализируемые данные (например, понимают отличие товарной линии от категории товаров).

Соревнование в анализе данных среди сотрудников организации может оказать позитивное влияние на весь производственный процесс, так как данные становятся более доступными, а количество необходимых отчетов сокращается.

В качестве альтернативы средства анализа данных могут реализовать инструментарий быстрого создания прототипов отчетов, что уменьшает требования, выдвигаемые при создании сложных отчетов.

Сводные таблицы Excel

Идея *сведения данных* основана на отображении итогов, размещенных на перекрестиях категорий, размещенных в заголовках строк и столбцов. По мере помещения категорий в область заголовков таблица автоматически обновляется, группируя данные по всем выбранным в текущий момент категориям (рис. 48.1).

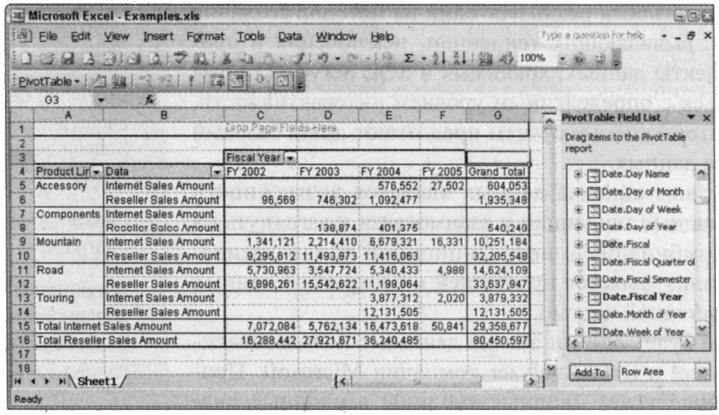


Рис. 48.1. Сводная таблица Excel



Все рисунки и описания функций Excel основаны на версии Microsoft Excel 2003. Эквивалентные функции доступны во всех версиях, начиная с Excel 2000, хотя некоторые детали могут отличаться.

Доступные поля данных, отображенные в списке Field List, можно перетаскивать в любую из четырех областей таблицы.

- Data. Эта область расположена в центре таблицы и содержит итоговые значения данных, например суммы объемов Интернет-продаж (см. рис. 48.1).
- Row headers. Данные категорий, обеспечивающие группировку данных в строки. Они отображаются в левой части таблицы (на рис. 48.1 это категория `Country`).
- Column headers. Данные категорий, обеспечивающие группировку по столбцам, — они отображаются в верхней части таблицы (на рис. 48.1 это `Fiscal Year`).
- Page. Область, отвечающая за установку общего фильтра данных, консолидируемых в сводной таблице. Она не оказывает никакого влияния на композицию таблицы. На рис. 48.1 эта область не показана — она отмечена в таблице меткой Drop Page Fields Here.

В дополнение к фильтрам, применяемым ко всей таблице, в любом поле заголовков столбца или строки можно выбрать любое подмножество значений категорий. После некоторой практики процесс создания нужного представления в среде сводной таблицы станет чрезвычайно эффективным. Единственным сдерживающим фактором станет только скорость работы исходного источника данных.

При создании сводной таблицы на основе реляционного источника данных отображаемые данные кэшируются в рабочей книге Excel. Таким образом, анализ больших объемов данных потребует создания файлов рабочих книг больших размеров. При использовании многомерных источников данных (кубов) скорость перестройки сводных таблиц при выполнении в них изменений зависит в основном от быстродействия выполнения запросов сервером службы анализа. Для работы с многомерными источниками данных в автономном режиме потребуется создание локального файла куба.

Подключение к многомерным источникам данных

В то время как сводные таблицы могут создаваться на основе списков данных, сгенерированных в Excel, они могут использовать и внешние источники данных. Доступ к этим данным требует создания подключения к ним. В качестве примера мы создадим сводную таблицу, выбрав в меню Excel пункт Данные⇨Сводная таблица (Data⇨PivotTable). Откроется мастер сводных таблиц, содержащий несколько последовательных страниц.

1. В разделе выбора типа источника данных установите переключатель в положение Во внешнем источнике данных (External Data), а в разделе вида создаваемого отчета — в положение Сводная таблица (PivotTable) или Сводная диаграмма (со сводной таблицей) (PivotTable/PivotChart). Щелкните на кнопке Далее (Next).
2. Щелкните на кнопке Получить данные (Get Data). Откроется диалоговое окно выбора источников данных, в котором следует перейти к вкладке Кубы OLAP (OLAP Cubes). Если нужный куб уже существует в списке, выберите его, щелкните на кнопке Далее и сразу переходите к п. 3.
   - Добавьте в список новый куб, выбрав пункт <Новый источник данных> (<New Data Source>) и щелкнув на кнопке OK. После этого откроется диалоговое окно создания нового источника данных, показанное на рис. 48.2.

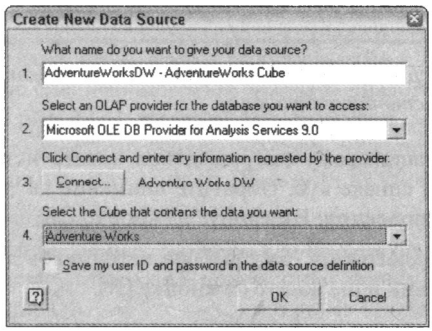


Рис. 48.2. Диалоговое окно создания нового источника данных

   - Введите имя источника данных и в качестве поставщика выберите Microsoft OLE DB Provider for Analysis Services 9.0.



Так как источник данных специфичен для каждого куба, лучше всего в имени источника данных отражать названия базы данных службы анализа и название самого куба. При желании в имя можно также добавить и имя сервера и среды, если опрашиваемая база данных существует во многих местах. Например, имя источника данных может быть следующим:
`Development-AdventureWorksDW-AdventureWorks Cube`.

- Щелкните на кнопке Content, чтобы с помощью мастера многомерных подключений выбрать сервер и базу данных.
- На первой странице открывшегося мастера (рис. 48.3) введите имя сервера. Обычно регистрационная информация не требуется, так как служба анализа, как правило, использует доверительные подключения. Если был создан локальный файл куба, установите переключатель в положение Cube File и выберите нужный файл с расширением `.cub`. (Процесс создания локального файла куба и извлечения в него данных подробно описан в документации к инструкции MDX `CREATE GLOBAL CUBE` в утилите Books Online.)
- На второй странице мастера выберите базу данных, содержащую интересующий вас куб.

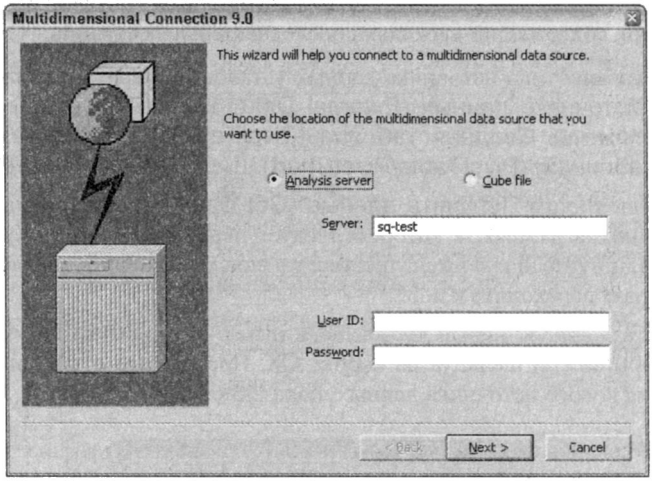


Рис. 48.3. Диалоговое окно создания подключения к многомерному источнику данных

- Вернувшись к диалоговому окну создания нового источника данных, выберите в раскрывающемся списке куб. Обратите внимание на то, что в список кубов включены также и их проекции. Щелкните на кнопке OK.
- Вернувшись к диалоговому окну выбора источников данных, выберите только что созданный источник и щелкните на кнопке OK.
- Вернувшись к мастеру сводных таблиц, щелкните на кнопке Далее, чтобы перейти к п. 3.

3. Проверьте правильность размещения создаваемой сводной таблицы (рабочий лист и координаты левого верхнего угла) и щелкните на кнопке Готово (Finish), чтобы завершить работу мастера.

Как видите, при первом доступе к кубу Excel собирает некоторые фрагменты информации о нем; следующие же операции создания сводных таблиц на основе этого же куба потребуют выполнения простого трехшагового процесса. После завершения работы мастера займитесь компоновкой полученных данных, как будет описано в разделе "Проектирование сводных таблиц".

Подключение к реляционным источникам данных

Для получения данных из реляционного источника вы также можете воспользоваться мастером сводных таблиц и диаграмм.

1. Выберите в меню Excel пункт Данные⇨Сводная таблица.
2. Щелкните на кнопке Получить данные, чтобы открыть диалоговое окно выбора источников данных. В этом диалоговом окне обратите внимание на флажок Использовать мастер запросов (Use the Query Wizard), находящийся в нижней части. Если этот флажок установлен, то запустится мастер запросов, предлагающий выбрать источник данных и последовательно проводящий пользователя по процессу создания запроса к одной таблице. Если снять этот флажок, то аплет Microsoft Query позволит создать более сложные запросы. В нашем примере мы установим данный флажок.
3. В открывшемся окне мастера запросов перейдите к вкладке Базы данных (Databases), чтобы начать создание нового запроса, или к вкладке Запросы (Queries), чтобы начать процесс с ранее сохраненного запроса. Если вы выберете существующий запрос, сразу переходите к п. 4.

   Вкладка Базы данных заполнена всеми источниками реляционных данных, ранее созданными в мастере или определенными в аплете Источники данных (ODBC) папки Администрирование панели управления Windows. Если нужной базы данных в списке нет, выберите пункт <Новый источник данных> (<New Data Source>). Заполнение открывшегося диалогового окна создания нового источника данных требует выполнения нескольких дополнительных действий.

   - Присвойте источнику данных имя. Так как источники данных неразрывно связаны с базами данных, неплохо в имени отразить имя базы данных и, при необходимости, имя сервера, на котором она размещена (например, `Development-AdventureWorks`).
   - Выберите подходящий драйвер (например, SQL Server Native Client для базы данных SQL Server).
   - Щелкните на кнопке Связь, чтобы подключиться к выбранному серверу и базе данных с помощью специфичного для драйвера диалогового окна (пример для SQL Server показан на рис. 48.4). Введите имя сервера и регистрационные данные, после чего расширьте диалоговое окно, щелкнув на кнопке Параметры (Options), и выберите необходимую базу данных. Щелкните на кнопке ОК.
   - Вернувшись к диалоговому окну создания нового источника данных, если нужно, выберите в источнике данных таблицу, используемую по умолчанию. Эта таблица будет автоматически выбираться при каждом использовании источника данных. Щелкните на кнопке ОК.
   - Вернувшись к окну выбора источников данных, выберите только что созданный источник и щелкните на кнопке ОК. В зависимости от установки флажка Использовать мастер запросов будет открыт либо мастер запросов, либо аплет *Microsoft Query*. Особенности использования каждого из этих инструментов при создании запросов будут описаны в следующих разделах.

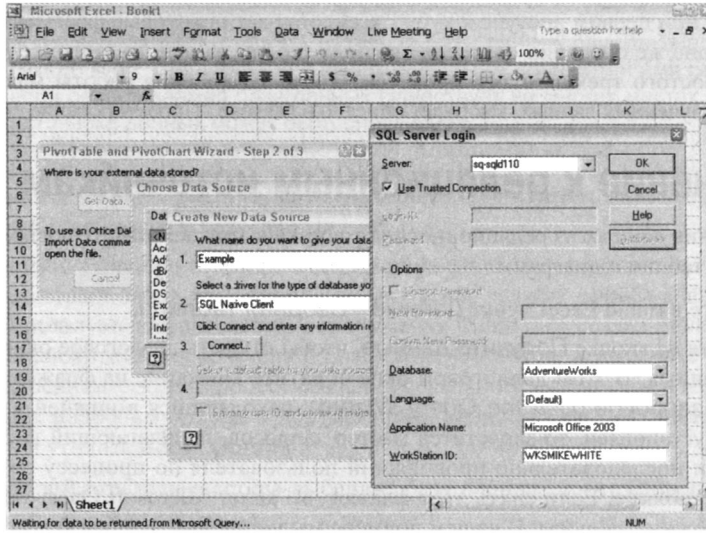


Рис. 48.4. Диалоговое окно параметров подключения к SQL Server

Когда управление будет возвращено в мастер сводных таблиц, рядом с кнопкой Получить данные отобразится сообщение о получении данных.

4. Проверьте правильность размещения создаваемой сводной таблицы (рабочий лист и координаты левого верхнего угла) и щелкните на кнопке Готово (Finish), чтобы завершить работу мастера.
5. Проверьте правильность размещения создаваемой сводной таблицы (рабочий лист и координаты левого верхнего угла) и щелкните на кнопке Готово (Finish), чтобы завершить работу мастера.

После завершения работы мастера займитесь компоновкой полученных данных, как описано ниже, в разделе "Проектирование сводных таблиц".

Мастер запросов

Как было описано выше, мастер запросов запускается после того, как был выбран источник данных, при условии, что флажок Использовать мастер запросов также был установлен. Этот мастер проведет вас по четырем страницам определения запроса к одной таблице; при этом вы не сможете создать ни одного объединения или обращения к хранимой процедуре. Вместо этого в формах, предлагаемых мастером, вам потребуется выбрать таблицу, столбцы, условия фильтрации и порядок сортировки. Для работы с мастером каких-либо знаний о языке SQL не требуется. По завершении работы мастера вам будет предложено либо сохранить запрос для последующего использования, либо продолжить работу над ним в аплете Microsoft Query. Следует отметить, что щелчок на кнопке Отмена в мастере запросов также приведет к переключению на аплет Microsoft Query, где придется продолжить процесс формирования запроса.

Microsoft Query

Аплет Microsoft Query запускается после выбора источника данных и снятия флажка Использовать мастер запросов. Этот аплет реализует графическую среду, подобную интерфейсу программы Microsoft Access, с областью для графического включения таблиц и соз-

дания отношений между ними, областью создания условий фильтрации и областью просмотра результатов запроса. В этой среде конструктора доступен ряд полезных операций.

- Щелкните на кнопке SQL панели инструментов, и вы получите возможность вручную ввести запрос, не обращаясь к графическому интерфейсу. Обычно эту возможность используют при создании сложных запросов, в частности, использующих хранимые процедуры, вложенные запросы и предложения `HAVING`.
- Выберите в меню пункт Таблица⇨Добавить таблицу (Tables⇨Add Tables) или щелкните на одноименной кнопке панели инструментов. Выбранные в открывшемся диалоговом окне таблицы будут добавлены на верхнюю панель конструктора запросов с определенными по умолчанию отношениями. С помощью перетаскивания названий полей вы можете создать дополнительные отношения; вы также можете удалить те линии отношений, которые считаете ненужными.
- Выключатели Таблицы (Tables) и Критерии (Criteria) меню Вид (View) позволяют отображать и скрывать соответствующие панели окна аплета.
- Установка критерия выполняется с помощью выбора имени столбца в верхней строке и ввода одной или более строк критериев в следующих строках. Для создания одного критерия просто введите его. Чтобы применить оператор подобия `LIKE` с использованием шаблона, введите его, как в следующем примере: `LIKE 'A%'` (в данном примере будут отобраны все записи, начинающиеся с символа `A`). Если для отбора следует использовать несколько значений, введите их в отдельных строках.
- Используйте переключатель Автоматический режим (Automatic Query) меню Записи (Records) для включения и отключения повторного запуска запроса после каждого его изменения. В "долгоиграющих" запросах отключение этого режима позволит сэкономить массу времени. При отключенном автоматическом режиме перезапуск запроса выполняется с помощью пункта меню Записи⇨Выполнить запрос (Records⇨Query Now).
- Выбрав пункт меню Записи⇨Сортировать (Records⇨Sort), можно установить порядок сортировки результатов запроса.

Проектирование сводных таблиц

Простейшим способом создания и удаления полей в перекрестной таблице является их перетаскивание в нее из панели списка полей. Для того чтобы такое стало возможным, текущая ячейка должна находиться в сводной таблице. Щелкните правой кнопкой мыши в сводной таблице и выберите в контекстном меню пункт Show Field List или Hide Field List, чтобы отобразить или скрыть список полей соответственно.

Формат списка полей сводной таблицы реляционного источника данных представляет собой список имен полей, в то время как для многомерных источников отображаются список измерений в верхней части и список мер — в нижней. Меры могут отображаться только в средней части (данных) сводной таблицы, в то время как измерения можно помещать в заголовки строк, столбцов и страницы. Для упрощения идентификации измерения и меры помечаются различными значками.

Каждое поле, помещенное в сводную таблицу, закрепляется заголовком поля, представляющим собой раскрывающийся список. Раскрывающиеся списки заголовков строк и столбцов содержат дерево значений, в котором элементы можно выбрать несколькими способами. Щелчок на элементе Show All Items позволяет установить флажки у всех невыбранных значений. Снятие флажка с элемента исключает соответствующее значение из таблицы, а двойной

флажок приводит к отображению всех потомков выбранного элемента. На рис. 48.5 приведен пример выделения элементов раскрывающегося списка значений. Дочерние элементы и двойные флажки недоступны в реляционных источниках данных и одноуровневых измерениях.

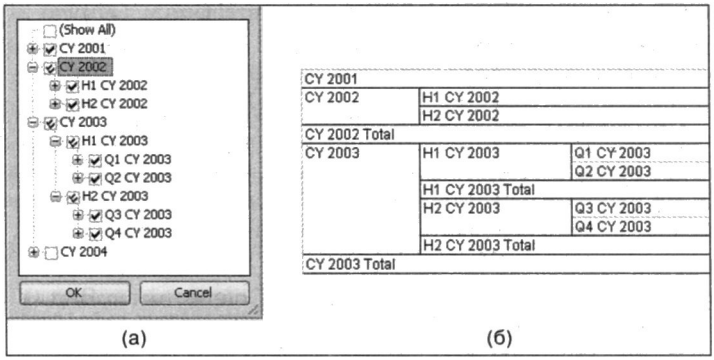


Рис. 48.5. Выбор значений в заголовках строк/столбцов (а) и получившиеся в результате заголовки (б)

Все поля в центральной области (данных) отображаются под единственным заголовком — Data. Этот заголовок можно использовать для удаления ненужных полей данных, сняв соответствующие флажки в раскрывающемся списке.

Раскрывающиеся списки, связанные с полем страницы, позволяют применять фильтры ко всей таблице. В примере на рис. 48.6, *а* сводная таблица ограничивается отображением данных, связанных с клиентами из Германии. Многомерные источники данных позволяют включать в фильтр множество значений, как показано на рис. 48.6, *б*.

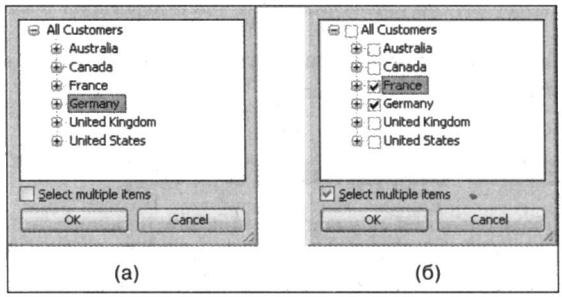


Рис. 48.6. Выбор полей фильтра страницы в одиночном (а) и множественном (б) режимах

При работе со сводными таблицами воспользуйтесь следующими советами.

- В многоуровневых измерениях после двойного щелчка на элементе отобразятся все его потомки. Отображение/сокрытие всех потомков некоторого уровня выполняется с помощью щелчка правой кнопкой мыши на заголовке поля и выбора соответствующей команды в подменю Group and Show Detail.
- В сводных таблицах, основанных на реляционных источниках данных, двойной щелчок на ячейке в области данных приведет к созданию нового рабочего листа, отображающего строки деталей, на основе которых было сформировано выбранное итоговое значение.

- Для открытия диалогового окна PivotTable Field дважды щелкните на заголовке поля (или щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Field Settings). В этом окне можно выполнить ряд действий.
  - Изменить экранное имя поля.
  - Выбрать подходящий числовой формат для поля данных. В отличие от прямых команд форматирования выбранные в окне форматы сохраняются при обновлениях сводной таблицы.
  - Изменить для полей данных, основанных на реляционных источниках, используемую итоговую функцию, например на `Min`, `Max`, `Average`, `Sum`, `Count` и т.п.
  - Выбрать альтернативное отображение полей данных с помощью щелчка на соответствующей кнопке параметров, например Running Total (Накопительный итог), % of Total (Процент относительно итога) и т.п.
  - Включить и отключить отображение промежуточных итогов для полей заголовков строк и столбцов.
  - Выбрать щелчком на кнопке Advanced полей заголовков строк и столбцов режим сортировки и порядок отображения наибольших (наименьших) значений.
  - Откорректировать параметры раскладки заголовков строк и столбцов. Доступны табличная раскладка и организованная с помощью выделения, отображение итогов над или под деталями, разрывы страниц и т.д.
- Щелкните правой кнопкой мыши на сводной таблице и выберите в контекстном меню пункт Table Options. Откроется диалоговое окно, позволяющее скорректировать специфичные для таблицы настройки, такие как отображение общих итогов и параметры форматирования.
- Щелкните правой кнопкой мыши на сводной таблице и выберите в контекстном меню пункт Refresh Data. Сводная таблица будет обновлена текущими данными, а список полей — названиями текущих полей (в случае изменения структуры источника данных).
- На панели инструментов сводной таблицы в раскрывающемся списке PivotTable выберите пункт Format Report, чтобы открыть доступ к списку предопределенных форматов, которые можно применить ко всей сводной таблице. Эти форматы помогут одним действием придать таблице такой внешний вид, которого достаточно сложно было бы добиться с помощью ручного форматирования.

Проектирование сводных диаграмм

Работа сводных диаграмм во многом сходна с обычными диаграммами Excel, за исключением того, что они неразрывно связаны со сводными таблицами. Когда сводная таблица изменяется, это сразу же отражается на диаграмме, и наоборот. Сводная диаграмма создается автоматически, если на первой странице мастера сводных таблиц переключатель был установлен в соответствующее положение. Также сводную диаграмму можно создать и вручную, щелкнув правой кнопкой мыши на сводной таблице и выбрав в контекстном меню пункт Сводная диаграмма (PivotChart). По умолчанию сводная диаграмма создается на отдельной вкладке (рабочем листе), однако при желании ее можно скопировать и вставить на любой другой лист.

В то время как обособленные сводные таблицы часто содержат множество заголовков строк и столбцов, сводные таблицы, связанные с диаграммами, должны быть относительно простыми, чтобы облегчить чтение диаграмм. В качестве примера на рис. 48.7 показана свод-

ная диаграмма, основанная на сводной таблице с одним заголовком строк (время) и одним заголовком столбцов (ряд данных). Несмотря на то что сводная таблица, связанная с этой диаграммой, отображает все три уровня временного измерения, на диаграмме показан только нижний уровень, во избежание проблем масштабирования.

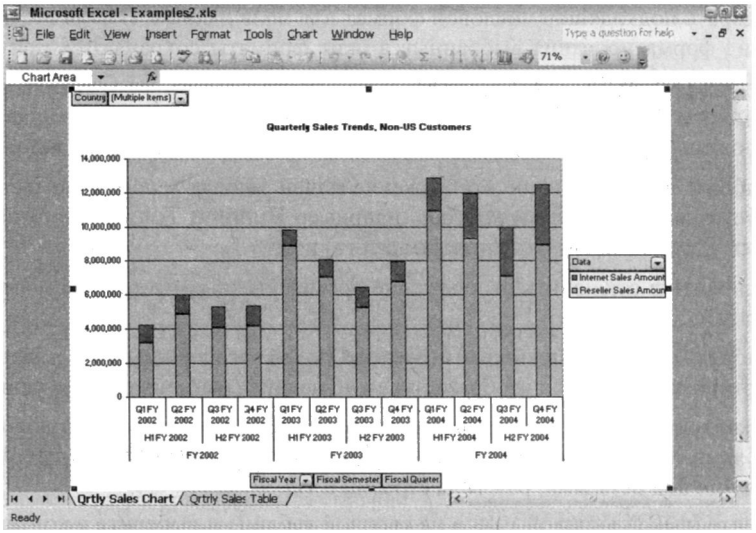


Рис. 48.7. Пример сводной диаграммы

На сводной диаграмме отображаются те же заголовки полей, что и в ассоциированной сводной таблице. Заголовки строк отображаются вдоль нижней, горизонтальной оси, заголовки столбцов — вдоль левой, вертикальной оси, а заголовки страниц — вверху. Раскрывающиеся списки заголовков позволяют выполнить практически такое же конфигурирование данных, как и в сводной таблице. При отображенном списке полей можно изменить и содержимое диаграммы. Если нужно, заголовки полей можно скрыть, щелкнув правой кнопкой мыши на одном из заголовков и выбрав в контекстном меню пункт Hide PivotChart Field Buttons. Подобно обычной диаграмме Excel, тип диаграммы, ее параметры и команды форматирования доступны в контекстном меню.

Диапазоны данных Excel

Определение диапазонов данных аналогично созданию сводной таблицы, основанной на реляционном источнике данных, за исключением того, что возвращенные данные отображаются в виде детального списка. Это позволяет отобразить в строках и столбцах рабочего листа содержимое таблиц, результаты хранимой процедуры или любого запроса. Такой список данных называют *диапазоном*.

Чтобы определить новый диапазон данных, выберите в меню Excel пункт Данные⇨Импорт внешних данных⇨Создать запрос (Data⇨Import External Data⇨New Database Query), после чего создайте запрос (см. выше раздел "Подключение к реляционным источникам данных").

После того как диапазон данных будет помещен на рабочий лист, щелчок на нем правой кнопкой мыши открывает контекстное меню с рядом доступных операций.

- Обновить данные (Refresh Data). Перезапуск запроса и помещение в диапазон самых свежих данных.

- Изменить запрос (Edit Query). Открытие мастера запросов или аплета Microsoft Query для изменения содержимого или организации возвращаемых данных.
- Свойства диапазона данных (Data Range Properties). Открывает возможность изменить график обновления данных, параметры раскладки и форматирования, определить режим применения изменений в процессе обновления, и т.д.

Анализатор данных

Анализатор данных (Data Analyzer) предлагает удобный подход к отображению многомерных данных. Вместо использования для определения заголовков строк и столбцов, как в сводной таблице, каждое измерение отображается в отдельном окне (рис. 48.8). После того как нужное представление было спроектировано, оно может быть сохранено и использоваться совместно группой пользователей. Набор таких представлений позволяет создать единообразную рабочую среду, обеспечивающую заранее сконфигурированные средства просмотра самых свежих данных и являющуюся отправной точкой для дальнейшего анализа данных.

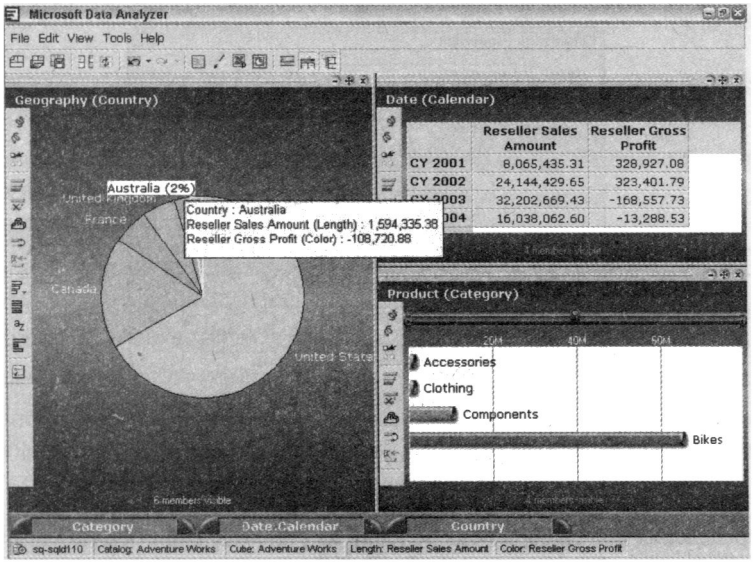


Рис. 48.8. Анализатор данных, отображающий многомерные данные

Каждое из окон представления может отображать данные в виде круговой диаграммы, гистограммы или таблицы. В каждом представлении можно консолидировать данные по измерениям с помощью одной или двух мер. Первая мера, называемая *мерой длины*, используется для определения длины столбцов гистограммы или толщины среза круговой диаграммы. Вторая, называемая *мерой цвета*, определяет раскраску полос гистограммы или секторов круговой диаграммы, от красного цвета (для наименьших значений) до зеленого (для наибольших значений). В примере на рис. 48.8 в качестве меры длины был выбран показатель количества проданных единиц товара (Unit Sales), а в качестве меры цвета — прибыль (Profit). Это отображено в строке состояния в нижней части окна. Точные значения отображаемых на диаграмме данных можно узнать, наведя указатель мыши на соответствующий столбец гистограммы или сектор круговой диаграммы.

Создание представления

Создание нового представления начинается с выбора в титульном диалоговом окне программы пункта Create New View или с выбора пункта меню File⇨New. На странице Connections выберите существующее подключение или создайте новое, щелкнув на кнопке Add. Если создается новое подключение, то откроется диалоговое окно Connection Properties (рис. 48.9), определяющее для представления конкретную комбинацию сервера, базы данных и куба. Создание подключения предполагает выполнение следующих действий.

1. Введите имя подключения и сервера.
2. При необходимости изменения строки подключения щелкните на кнопке Advanced. Доступ к службе анализа в SQL Server 2005 требует указания в строке подключения параметра `provider=Msolap.3`.
3. Щелкните на кнопке Connect, чтобы подключиться к серверу и перезаполнить списки Catalog (database) и Cube.
4. Выберите нужную комбинацию каталога и куба.

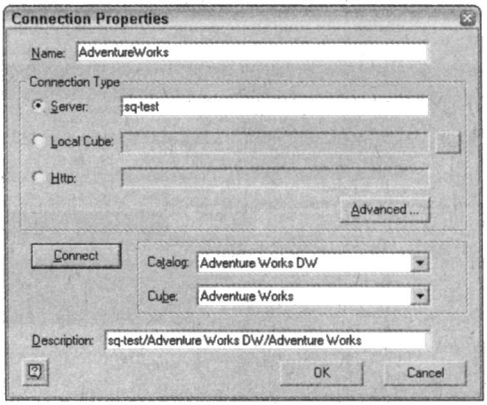


Рис. 48.9. Диалоговое окно настроек подключения в Data Analyzer

После определения или выбора подключения на странице Dimensions мастера отобразится список всех измерений выбранного куба. Выберите измерения, которые должны отображаться в новом представлении, не забывая, что для каждого выбранного измерения будет открываться отдельное окно. В заключение на странице Measures мастера выберите меры, которые будут влиять в представлении на длину столбцов гистограммы и цвета секторов круговой диаграммы.

Форматирование представления

Параметры, определяющие степень полезности представления, варьируются в зависимости от поставленной задачи, однако в общем случае руководствуются следующим принципом: чем проще, тем лучше. Для обеспечения максимальной простоты в каждом представлении прежде всего нужно определиться с его назначением. Всегда лучше создать дополнительные представления, чем слишком загромождать одно. Также имейте в виду, что окна представлений можно сворачивать, а окно измерений является постоянно готовым резервом

свободного пространства в окне программы. При настройке форматирования рекомендуется прислушаться к следующим советам.

- Изменяйте тип отображения в окне с помощью переключателя, находящегося в нижней левой части каждого окна.
- Перетаскивайте окна, чтобы изменить их изначальное упорядочение по горизонтали.
- Если щелкнуть правой кнопкой на каком-либо элементе измерения, откроется контекстное меню со следующими доступными командами.
  - Filter by <name> (Включение в критерий фильтрации). Ограничение отображаемых в других окнах данных только относящимися к выбранному члену. Аналогичный эффект можно получить, щелкнув на члене левой кнопкой мыши. Таким способом может быть выбран только один член измерения, однако в критерий фильтрации могут быть добавлены члены других измерений с помощью выполнения аналогичной операции в соответствующих этим измерениям окнах.
  - Remone <name> from filter (Удаление из критерия фильтрации). Оставить все члены, кроме выбранного, в критерии фильтра.
  - Hide <name>. Выбранный член удаляется из критерия фильтра и из списка отображаемых.
  - Drill down <name> (Раскрытие дочерних объектов). Отобразить потомков выбранного члена многоуровневого измерения.
- Панель инструментов, отображаемая в каждом окне, также предлагает несколько полезных функций.
  - Drill UP/Down. Отобразить предков или потомков текущих членов.
  - Default members. Снять все фильтры и восстановить скрытые члены в текущем окне.
  - Filter by All Visible. Снять все фильтры, игнорируя скрытые члены.
  - Hide Members Not in Filter. Скрыть отфильтрованные члены в текущем окне.
  - Filter by Criteria. Отфильтровать члены на основе значения некоторой меры куба. Например, можно включить в представление только те магазины, объем продаж в которых превысил заданную сумму.
  - Reverse Filter. Отразить состояние всех отфильтрованных и неотфильтрованных членов.
  - Hide Empty. Скрыть пустые члены.
  - Length/Color/Name. Отсортировать членов по выбранному атрибуту.
  - Natural. Использовать порядок сортировки, установленный в кубе.
  - Properties. Установить сортировку членов по любой мере куба.

Резюме

Сводные таблицы и диаграммы являются удобными инструментами анализа данных. Как функциональные элементы популярнейшей программы Excel, они могут быть более понятными и привычными персоналу компании, чем альтернативные средства. К тому же многие компании и так приобретают лицензии Excel для других производственных нужд, что это позволяет сократить объем инвестиций в анализ данных. Некоторые компании даже создают

собственную инфраструктуру отчетности именно на основе сводных таблиц Excel, чтобы иметь возможность выполнять неочевидный анализ после доставки отчета.

Программа Data Analyzer является еще одним полезным инструментом анализа данных. В ней могут быть разработаны полезные представления, к которым впоследствии при необходимости могут обращаться конечные пользователи. Кроме того, эта программа позволяет проектировать более отшлифованные представления, чем те, что создаются с помощью сводных таблиц/диаграмм Excel.

Стратегии оптимизации

ЧАСТЬ

VI

Эта книга начиналась с описания принципов информационной архитектуры, в одном из которых подчеркивалось, что информация должна быть *всегда доступной*. Теория оптимизации объясняет зависимости между различными технологиями оптимизации. В этой, заключительной, части мы применим теорию оптимизации в практике при создании стратегий оптимизации.

- **Модель схемы**. На первом уровне теории оптимизации находится схема (см. главы 1, 2 и 17). Прозрачная схема, без излишних сложностей, позволяет создавать эффективные запросы.
- **Пакетные запросы**. На втором уровне лежит оптимизация запросов. Во второй части книги десять глав было посвящено созданию пакетных запросов, а в главе 20 было показано, в чем заключается их превосходство над итеративным программированием.
- **Индексация**. Индексы служат мостом между схемой и схемой и запросами, поэтому их эффективность зависит от качества схемы и пакетных запросов. В главе 50 мы проанализируем планы выполнения запросов и рассмотрим вопросы настройки индексов.
- **Конкуренция**. Блокировка записей таблиц может превратиться в сплошной кошмар, если модель схемы, запросов и индексов не сокращает по максимуму время выполнения транзакций. Вопросу конкуренции будет посвящена глава 51.
- **Повышенная масштабируемость**. Когда и схема, и транзакции оптимизированы, база данных обеспечивает высокую производительность, а требования к аппаратному обеспечению учтены, особую эффективность приобретают такие технологии, как разделение таблиц. В главе 53 мы займемся вопросами масштабирования особо крупных баз данных.

В этой части...

Глава 49
Измерение производительности

Глава 50
Анализ запросов и настройка индексов

Глава 51
Управление транзакциями и блокировкой

Глава 52
Обеспечение высокой доступности

Глава 53
Масштабирование особо крупных баз данных

Глава 54
Разработка высокопроизводительных поставщиков доступа к данным

ГЛАВА

49

Измерение производительности

В этой главе...

Установка координат измерений

Просмотр данных производительности

Определение ключевых индикаторов производительности

В основе всех стратегий оптимизации лежат измерения производительности. Без конкретных цифр любая оценка производительности была бы чисто субъективной. Только цифры помогают выявить тенденции и узкие места и учитывать эти показатели в настройке базы данных. Теория оптимизации, в основе которой лежат целостные измерения, позволяет проектировать высокопроизводительные базы данных.

Для измерения производительности SQL Server доступно несколько великолепных методов.

- В монитор производительности (Performance Monitor) системы Windows при установке SQL Server встраивается несколько дополнительных счетчиков, позволяющих собирать информацию о производительности сервера баз данных в журналах для дальнейшего анализа.
- Утилита SQL Server Profiler позволяет с предельной детализацией отслеживать практически все события в сервере базы данных. Эту утилиту можно было бы назвать пределом мечтаний оптимизаторов баз данных.
- Динамические представления управления позволяют вскрыть множество деталей в работе сервера баз данных SQL Server 2005.
- Ядро базы данных возвращает статистику времени выполнения запросов и операций ввода-вывода.
- Пакеты T-SQL и хранимые процедуры могут использовать функцию `GetDate()` для вычисления времени выполнения команд с точностью до трех миллисекунд и занесения полученных значений в журнал.

Измерение точности

Нельзя написать о производительности, не затронув вопрос измерения точности. Одним из самых раздражающих факторов является то, что управляющий персонал считает своим долгом вкладывать деньги в оборудование и гораздо реже за-

ботится об оптимизации баз данных, а вопрос измерения точности вычислений вообще считает излишним. Если ваше руководство мыслит по-другому, то это, скорее, исключение.

Процесс измерения точности включает в себя ряд предсказаний и объяснений. В то время как вопрос тестирования базы данных или приложения, обрабатывающих десяток строк, можно воспринимать как профессиональную шутку, использование корректных сценариев тестирования является единственным доступным методом доказательства точности приложения.

Сценарий тестирования измеряет точность базы данных на основе сравнения фактического выполнения запросов с ожидаемыми результатами. Сложный запрос, обрабатывающий десятки миллионов строк, достаточно проблематично проверять на точность. В то же время тщательно продуманный сценарий, обрабатывающий все возможные комбинации данных пары десятков строк, позволяет довольно легко получить ответы на вопросы о точности вычислений. Сценарий тестирования может быть реализован как повторяющийся сценарий, создающий учебную базу данных и наполняющий ее данными. Пример такого сценария приведен в файле `CreateScenario.sql` на сайте книги.

Сценарий тестирования `TestScenario.sql` выполняет хранимые процедуры и представления на уровне абстракции данных и сравнивает результаты с заранее известными результатами. В идеальном случае сценарий тестирования базы данных разрабатывается еще до создания уровня абстракции данных. После этого создается уровень абстракции данных, позволяющий получить корректные ответы на запросы сценария тестирования.



Проверка сценария является одним из самых значительных тестов базы данных. Если база данных не прошла проверку на корректность данных, ее нельзя считать приемлемой для выполнения реальных производственных задач.



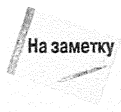
Программист и консультант Скотт Амблер (Scott Ambler) является ярым апологетом разработки программ, основанной на тестировании. Более подробно с его работами вы можете ознакомиться на сайте `www.ambysoft.com`.



Разработанное мною приложение SQL Data Quality Analyzer способно оценить качество производственных данных, хранимых в базах данных SQL Server. Оно определяет, когда некоторая строка не соблюдает установленные правила, и отмечает ее для проверки. Также это приложение отмечает, когда строка была проверена вручную, и удовлетворяет установленным правилам. При желании вы можете загрузить эту программу с сайта `www.SQLServerBible.com` и проверить ее действенность на практике.

Использование монитора производительности

Монитор производительности включает в себя две консоли: System Monitor и Performance Logs and Alerts. Некоторые серверы помещают ярлык этого приложения в папку `Администрирование` панели управления системы Windows. В качестве альтернативы этот монитор можно запустить в утилите SQL Server Profiler, выбрав в ней пункт меню Tools⇨Performance Monitor.



Само название программы *монитор производительности* (Performance Monitor) способно ввести в заблуждение. `PerfMon.exe` — это приложение, которое поставляется с предыдущими версиями операционной системы Windows. Новый монитор производительности является полноценной *консолью* управления, и так его было бы правильнее и называть, однако почему-то компания Microsoft

решила назвать это приложение именно *монитором*, так что нам ничего не остается, кроме как смириться с этим. Подробнее об этом приложении вы можете узнать в книгах, посвященных Windows XP.

Монитор системы

Монитор системы, точнее, программа `sysmon.exe`, хорошо знаком каждому, кто когда-либо занимался администрированием серверов, находящихся под управлением серверной версии Windows. Монитор системы содержит множество *счетчиков*, консолидирующих данные о внутренних характеристиках сервера. В некотором смысле он напоминает монитор кардиографа, однако измеряет режим работы не сердца человека, а операционной системы и сервера баз данных (рис. 49.1).

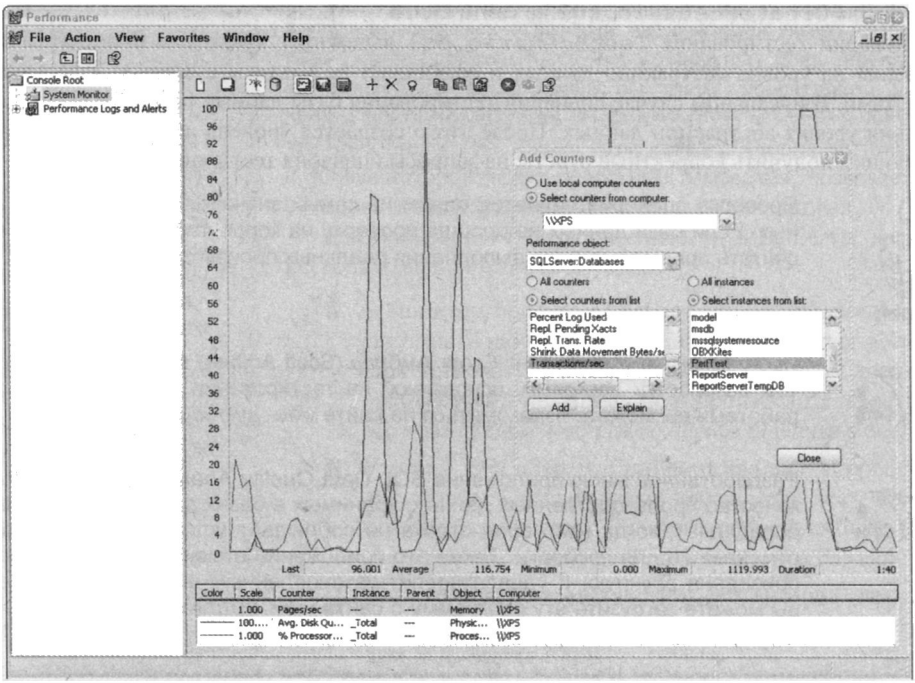


Рис. 49.1. Приложение монитора системы может пригодиться для оценки общей активности SQL Server

Дополнительные счетчики производительности добавляются в монитор системы по одному, с помощью щелчка на кнопке со знаком "плюс" на панели инструментов. Счетчики производительности можно устанавливать как на локальном, так и на удаленном сервере; таким образом, совершенно не обязательно запускать эту программу непосредственно на компьютере сервера баз данных. Показатели счетчиков могут отображаться в виде линейного графика, гистограммы или в реальном времени.

Все счетчики организованы по объектам и иногда по экземплярам объектов. В примере, показанном на рис. 49.1, объект `SQL Server:Databases` содержит множество счетчиков, включая счетчик количества транзакций, обрабатываемых в секунду. Показатели этого счетчика можно просмотреть как для всех баз данных, так и только для избранных экземпляров.



Ядро базы данных SQL Server является далеко не единственным поставщиком информации для монитора системы — свои счетчики в него добавляют службы анализа и отчетности, серверы .NET, ASP, BizTalk и многие другие.

Обычно новый счетчик отображается в виде прямой линии в верхней или нижней части диаграммы, так как для его более детального просмотра необходимо выполнить масштабирование. В диалоговом окне параметров программы, которое открывается с помощью команды Properties контекстного меню, вы можете скорректировать масштаб всех графиков, диапазон значений выбранного счетчика и его представления на экране.

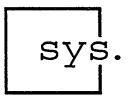
Несмотря на то что существуют сотни доступных счетчиков монитора системы, в табл. 49.1 перечислены только самые популярные из них, непосредственно относящиеся к серверу баз данных SQL Server.

Таблица 49.1. Ключевые счетчики мониторинга производительности

Объект	Счетчик	Описание	Область применения
`SQLServer:Buffer Manager`	Мера использования кэша	Процент чтений, выполняемых из кэшированных в памяти данных	SQL Server обычно хорошо справляется с работой предварительного кэширования данных в память. Если показатели не превышают 95%, то больший объем памяти может существенно повысить производительность
`Processor`	Мера загрузки процессора	Процент общей загрузки процессора	Если мера загрузки процессора постоянно находится на уровне больше 60%, существенно повлиять на производительность могут дополнительные и более быстродействующие ядра процессора
`SQLServer:SQL Statistics`	Количество пакетных запросов в секунду	Обработка пакетов SQL	Неплохой индикатор активности пользователей
`Physical Disk`	Средняя длина очереди дисковых операций	Общее количество дисковых операций (чтения и записи), ожидающих в очереди. Это индикатор общей пропускной способности диска, на которую влияют количество приводов, задействованных в RAID-массиве. Согласно мнению компании Microsoft, длина этой очереди не должна превышать количество дисков плюс 2. (Не забудьте при просмотре этого показателя проверить масштаб.)	Пропускная способность диска является ключевым фактором производительности. Разделение базы данных по множеству дисковых подсистем может положительно сказаться на производительности

Объект	Счетчик	Описание	Область применения
`SQLServer:SQL Statistics`	Количество отказов автоматической настройки параметров в секунду	Количество запросов, для которых СУБД SQL Server не кэшировала план выполнения в память. Эта мера является индикатором плохо написанных запросов (не забывайте проверять масштабирование)	Локализация и корректировка плохо написанных запросов позволяет повысить производительность
`SQLServer: Locks`	Среднее время ожидания блокировок (в миллисекундах)	Может стать причиной серьезных проблем производительности. Время ожидания блокировок, а также общее количество и величина задержек являются хорошими индикаторами качества политики блокировок, использованной в базе данных	Если возникают проблемы, связанные с блокировкой, следует еще раз проверить структуру индексов и программный код транзакций
`SQLServer:User Connections`	Количество подключений пользователей	Текущее количество подключений	Неплохой индикатор потенциальной ценности базы данных
`SQLServer: Databases`	Количество транзакций в секунду	Текущее количество транзакций в секунду	Хороший индикатор активности базы данных

Следует отметить, что еще один индикатор, `SQLServer:Wait Statistics`, способен помочь в выявлении узких мест в сервере баз данных.



Полный список счетчиков SQL Server и их текущие значения можно узнать с помощью динамического представления управления `sysdm_os_performance_counters`. Это достаточно удобно, поскольку позволяет получить статистические данные непосредственно в программном коде T-SQL.

Вы и сами можете создать собственные счетчики с помощью языка T-SQL и передавать данные из базы в системный монитор. Это может оказаться полезным для отображения производительности транзакций или количества строк, вставляемых генератором данных. Существует десяток полезных пользовательских счетчиков. В следующем простом примере показано приращение значения счетчика:

```
DECLARE @Counter Int
SET @Counter = 0
While @Counter < 100
  BEGIN
    SET @Counter = @Counter + 1
    EXEC sp_user_counter1 @Counter
    WAITFOR Delay '00:00:02'
  END
```



Я использую монитор системы для получения общей картины состояния сервера баз данных и определения, с какими вопросами могу потенциально столкнуться в работе. На основании этой информации я могу воспользоваться утилитой SQL Profiler для более детального ознакомления с проблемой и ее решения.

Конфигурацию монитора системы, включая все счетчики, можно сохранить в файле — для этого следует выбрать в меню команду File⇨Save As. Если необходимо восстановить эту конфигурационную информацию, нужно выбрать в меню пункт File⇨Open. С помощью этой технологии вы можете экспортировать конфигурацию монитора системы на другие серверы.

Однако существует одна тонкость. Если счетчик осуществляет мониторинг локального сервера, то и на другом сервере он будет отслеживать состояние локального компьютера. Если же счетчик предназначен для мониторинга удаленного сервера, то даже будучи установленным на другом компьютере, он будет отслеживать все тот же сервер, независимо от того, где открыт файл конфигурации монитора системы. Так как администраторы баз данных крайне редко работают за тем же компьютером, на котором установлена СУБД SQL Server, этот факт может представлять проблему. Если данная проблема встанет перед вами, отправьте мне письмо по электронной почте — возможно, к тому моменту я напишу собственный монитор системы, в котором решу эту и многие другие проблемы.

Протоколы счетчиков производительности

Монитор производительности также содержит встраиваемый модуль Performance Logs and Alerts, включающий в себя протоколы счетчиков (Counter Logs), оповещения трассировки (Trace Alerts) и обычные предупреждения. В этом разделе мы сосредоточим внимание на первом из этих компонентов. Протоколы счетчиков используют все те же счетчики сервера, что и монитор системы, однако вместо графического отображения на экране их показателей в реальном времени данные записываются в журнал. Это значит, что записанные данные можно будет впоследствии проанализировать и даже воспроизвести в утилите SQL Server Profiler (об этом прекрасном инструменте мы подробно поговорим в следующем разделе).

Параметры конфигурации протоколов производительности перечислены в узле Counter Logs монитора производительности. Чтобы увидеть результирующие файлы, достаточно заглянуть в выходной каталог.

Для создания нового протокола производительности следует выбрать в контекстном меню узла Counter Logs пункт New Log Settings. После ввода имени протокола откроется диалоговое окно, показанное на рис. 49.2, в котором можно выбрать регистрируемые счетчики. Добавление нового объекта приводит к перечислению для него всех счетчиков; в то же время можно вручную изменить их состав для более точных целей, подобно тому, как это выполняется в мониторе системы.

Для протоколов счетчиков можно составить расписание; их можно запускать и останавливать вручную из контекстного меню Log или с помощью соответствующих кнопок панели инструментов.

Если создание протокола было определено в виде текстового файла с разделителями, вы сможете открыть его с помощью программы Excel. Каждый столбец представляет собой последовательность показаний некоторого счетчика, а каждая строка является совокупностью показаний одной выборки.

Использование SQL Server Profiler

Одним из моих самых любимых инструментов является SQL Server Profiler, который часто называют просто Profiler. Эта утилита отображает данные о любом количестве детализированных событий SQL Server. Эти события сервера можно просматривать в окне Trace Properties (рис. 49.3), а также записывать в файл или таблицу для последующего анализа. Для регистрации всех событий или их избранного подмножества можно установить фильтры.

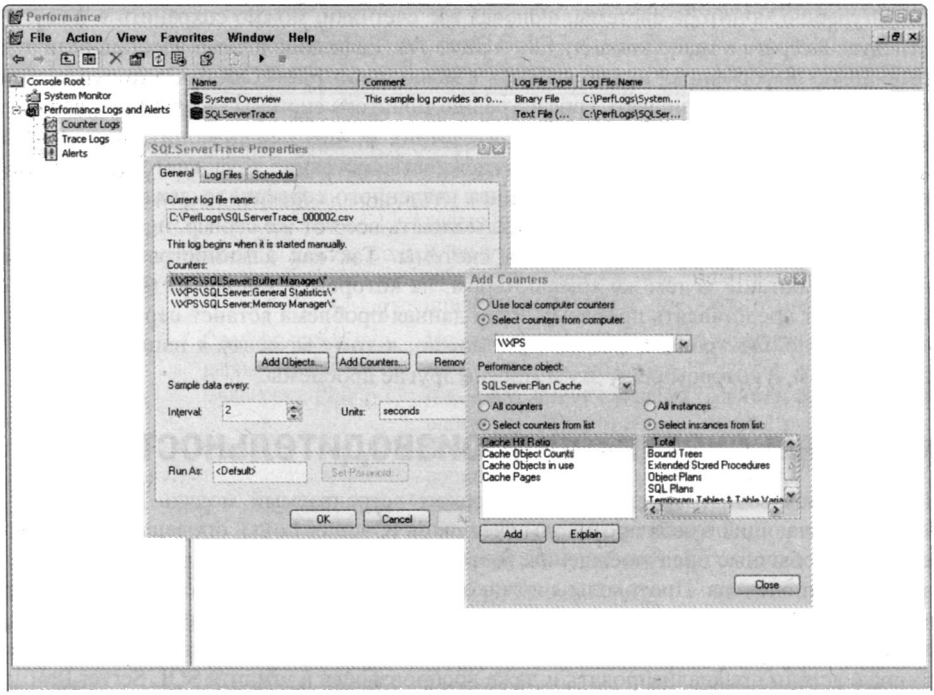


Рис. 49.2. В данном примере протокол счетчиков будет регистрировать информацию о производительности SQL Server в каталоге `C:\PerfLogs`

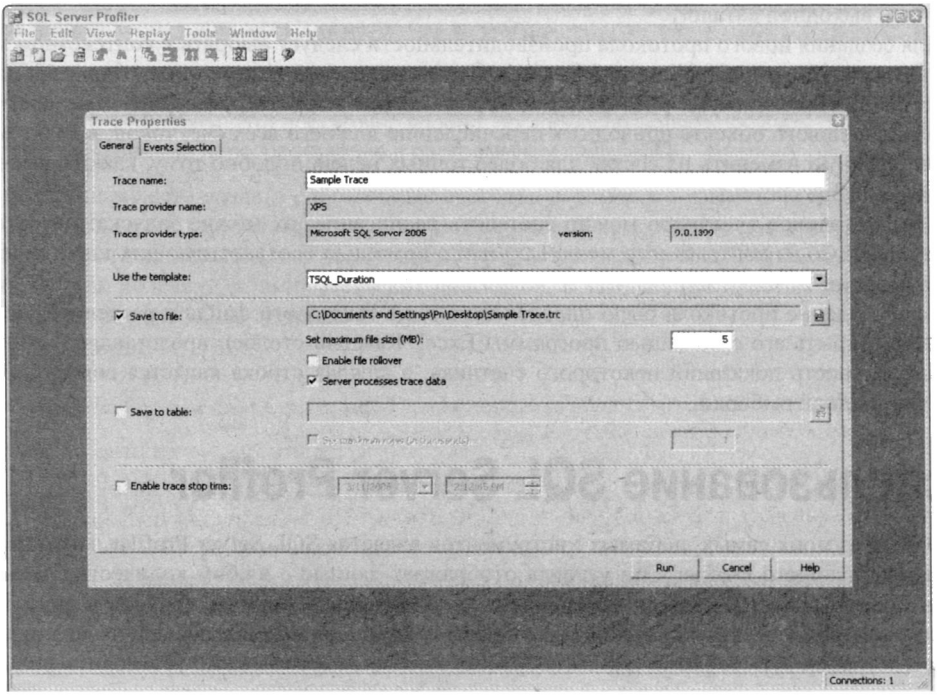


Рис. 49.3. В этом окне SQL Server Profiler использован шаблон T-SQL Duration *для отображения времени выполнения всех инструкций T-SQL*

Утилиту SQL Server Profiler можно запустить из меню Tools программы Management Studio или непосредственно из папки SQL Server 2005 системного меню Пуск. Для просмотра действий вам необходимо либо определить новую трассировку, либо воспользоваться уже существующим файлом.



С выходом пакета обновлений SP1 SQL Server Profiler получил возможность выполнять трассировку крупных массивов данных на больших компьютерах. Кроме того, ранее при мониторинге службы анализа время отображалось в единицах универсального синхронизированного времени (UTC). Теперь используется локальное системное время. Ранее успешно воспроизведенные события подсчитывались некорректно, в результате пользователь получал некорректную статистику. Эта проблема была устранена.

Определение новой трассировки

Когда создается новая трассировка (либо с помощью команды меню File⇨New Trace, либо с помощью кнопки New Trace панели инструментов), создается также новое подключение к SQL Server и открывается диалоговое окно Trace Properties (рис. 49.4). Во вкладке General этого окна настраивается трассировка (в частности, имя, местоположение файла и т.п.), а во вкладке Events Selection определяются регистрируемые события, данные и фильтры. Если трассировка запущена, то эти параметры можно просматривать, но не изменять. Конфигурация трассировки может быть сохранена в виде шаблона, чтобы облегчить создание новых трассировок в будущем.

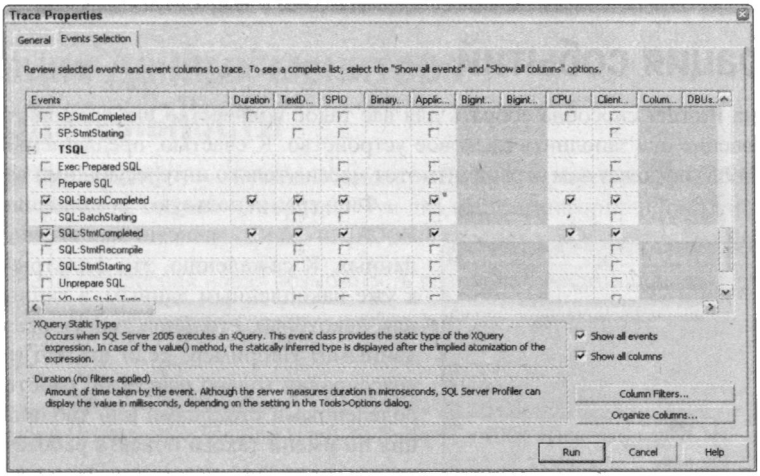


Рис. 49.4. Вкладка Events Selection окна Trace Properties позволяет отобрать события, отслеживаемые утилитой Profiler

Трассировку можно просматривать в реальном времени, однако эти данные могут одновременно записываться в файл или таблицу SQL Server. Это полезно для последующего интеллектуального анализа, сравнения с данными счетчиков монитора системы или для импорта в утилиту Database Engine Tuning Advisor.

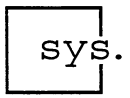
Когда показания записываются в файл, для повышения производительности они объединяются в цепочки по 128 Кбайт; аналогично, при записи в таблицу данные группируются по несколько строк.



Чтобы сохранить данные, полученные программой Profiler, для последующего анализа, используйте высокопроизводительный файловый метод, а также серверную трассировку (о ней мы поговорим позже). Если вы хотите анализировать данные с помощью инструкций T-SQL, используйте точно такой же подход, но после завершения сеанса трассировки откройте полученный файл в утилите Profiler и выберите в меню команду File⇨Save As⇨Table.

Отбор событий

Во вкладке Events Selection определяется состав действий, выполняемых сервером баз данных, которые будут регистрироваться утилитой Profiler. Подобно монитору производительности, Profiler может отслеживать множество ключевых событий SQL Server. Для упрощения настройки отбора можно использовать шаблоны, предлагаемые по умолчанию.



Событие `SQL Batch Completed` основано на выполнении пакетов T-SQL в целом (разделенных терминаторами пакетов), а не отдельных их инструкций. Исходя из этого, Profiler регистрирует данные всего об одном событии, независимо от длины пакета. Для регистрации выполнения отдельных инструкций DML используйте событие `SQL Statement Complete`.

Для воспроизведения трассировки можно использовать далеко не все события. Например, событие `SQL Batch Start` может быть воспроизведено, а событие `SQL Batch Complete` — нет.

В зависимости от событий для трассировки становятся доступными разные данные. Несмотря на то что столбец данных SPID кажется необязательным, это впечатление обманчиво — он обязателен.

Фильтрация событий

Программа Profiler способна собрать для вас такое количество информации, которая легко может в мгновение ока заполнить дисковое устройство. К счастью, предлагаемый программой фильтр (рис. 49.5) поможет вам ограничить этот массив только интересующими вас данными.

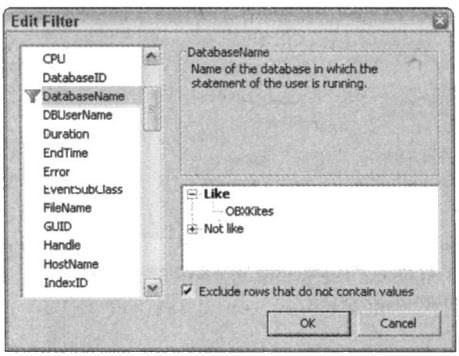


Рис. 49.5. Диалоговое окно Edit Filter является аналогом предложения `WHERE` в задачах трассировки, ограничивая множество отслеживаемых событий. В данном примере мы ограничиваем это множество только событиями, происходящими в базе данных OBXKites

Фильтры используют комбинацию операторов `EQUAL` и `LIKE`, зависящую от типа извлекаемых данных. К сожалению, эти фильтры применяются к уже накопленным данным, а данные, собранные для некоторых столбцов, могут оказаться неожиданными. Например, если вы хотите ограничить трассировку только пакетами, работающими с определенными столбцами или таблицами, фильтрация по имени такого объекта работать не будет. В то же время использование фильтра `LIKE` с символами макроподстановки в столбце `text data` приведет к тому, что Profiler отберет только те пакеты, которые содержат имя заданной таблицы.

Еще один популярный фильтр, устанавливаемый в Profiler, отсеивает только те пакеты, продолжительность выполнения которых превышает заданное время. Это позволяет отследить все слишком долго выполняемые пакеты.

Установка флажка Exclude system ID позволяет отфильтровать только пользовательские объекты.

Организация столбцов

Чтобы добавить столбец для группировки, перед тем как щелкнуть на кнопке Add, выберите пункт group by в правом столбце (он не показан на рисунке). Приоритетом столбцов в группах можно управлять с помощью кнопки Up, увеличивая его по отношению к другим столбцам. Любой столбец, по которому выполняется группировка, отображается в окне трассировки первым, и по мере добавления новых столбцов они присоединяются к соответствующим группам.

Использование трассировки

Как только информация о событиях будет собрана, ее можно просмотреть в окне трассировки программы Profiler, несмотря на то, что считается хорошим тоном накапливать эти данные в таблице базы данных. Эти данные можно затем проанализировать, а также выполнить над ними другие действия, доступные для данных обычных таблиц SQL.

После того как файл записан, его можно открыть в SQL Server Profiler и просмотреть данные. Эти данные можно переписать в таблицу для дальнейшего обобщающего анализа с помощью команды File⇨Save As.

Программа SQL Profiler способна воспроизводить записанные трассировки, однако ограничения, налагаемые на воспроизведение, делают данную возможность малополезной в большинстве баз данных.

К тому же весь файл трассировки может быть передан в качестве рабочего материала в Database Tuning Advisor для тонкой настройки различных запросов.

Интеграция данных монитора производительности

Каждый из инструментов — System Monitor и Profiler — предлагает собственную уникальную точку зрения на состояние сервера. Эти два набора информации можно объединить, чтобы проанализировать весь сценарий с обеих точек зрения в SQL Server Profiler.



Эта прекрасная новая возможность позволяет увидеть общую картину происходящего на сервере.

Чтобы увидеть эту многостороннюю картину, следует одновременно создать журналы производительности в Performance Monitor и в SQL Server Profiler. Ниже описано, как это сделать.

1. Сконфигурируйте System Monitor именно с теми счетчиками, которые захотите просмотреть впоследствии. Обязательно при этом выбирайте правильный масштаб и прочие параметры. После этого сконфигурируйте Counter Log с точно такими же параметрами.
2. Сконфигурируйте Profiler с правильным составом отслеживаемых событий. Не забудьте про столбцы времени начала и окончания, чтобы впоследствии инструмент Profiler смог интегрировать два журнала. Сформируйте для трассировки сценарий SQL и закройте Profiler.
3. Вручную запустите Counter Log. Выполните код T-SQL, чтобы запустить трассировку на стороне сервера.

4. Выполните на сервере код с данными, которые хотите проанализировать.
5. Когда тест будет завершен, остановите Counter Log и серверный код трассировки.
6. Запустите Profiler и откройте сохраненный файл трассировки.
7. Импортируйте журнал Counter Log с помощью команды меню File⇨Import Performance Data.

На импорт журнала утилита Profiler отреагирует открытием новой панели, отображающей диаграмму монитора системы (рис. 49.6). После выбора события в Profiler или времени на диаграмме в мониторе системы оба этих представления будут автоматически синхронизироваться. Неплохо, неправда ли?



Чтобы увидеть все эти действия воочию, загрузите демонстрационный ролик с сайта книги.

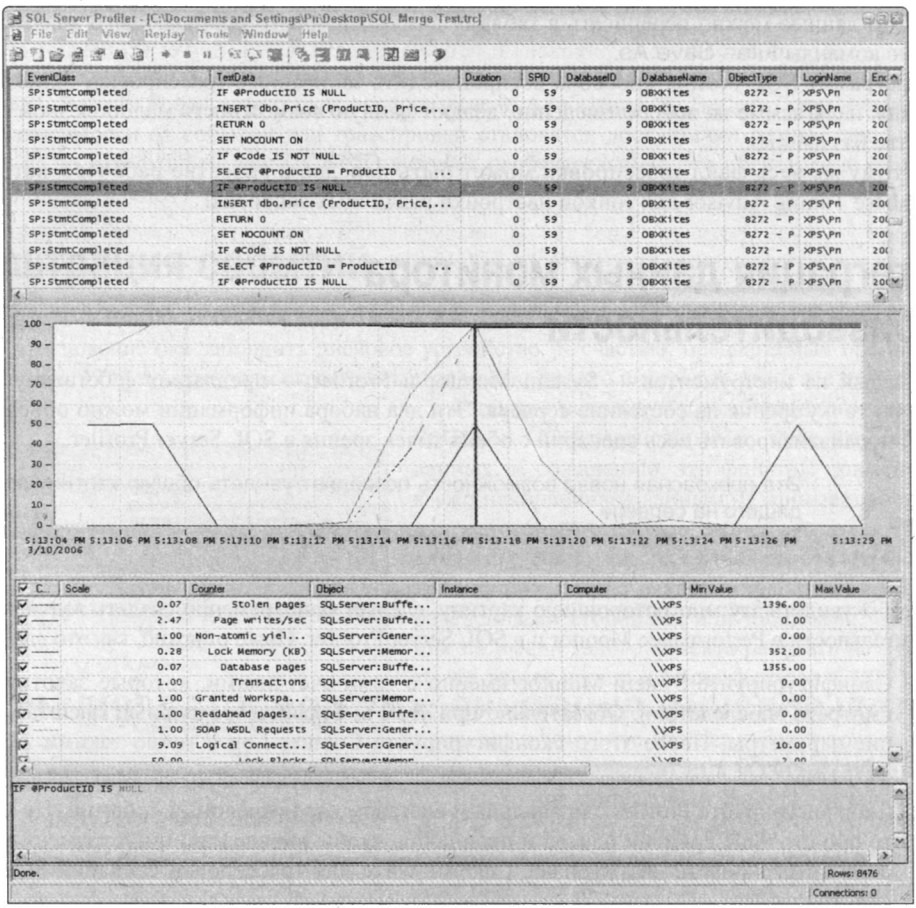


Рис. 49.6. Утилита SQL Server Profiler способна интегрировать данные монитора производительности и синхронизировать их с отслеженными событиями

Использование SQL Trace

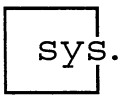
Программу SQL Profiler обычно используют интерактивно, и для непостоянного сбора данных ее вполне достаточно. Однако длительные трассировки способны с легкостью накопить сотни тысяч записей, что может вызвать вполне конкретные проблемы на рабочей станции, выполняющей трассировку. Решение сводится к созданию журнала трассировки непосредственно на сервере. Такая трассировка вызовет небольшую дополнительную нагрузку на сервер; при этом файлы будут записываться блоками по 128 Кбайт.



В промышленных системах, осуществляющих трассировку на стороне сервера, запись данных в файл на сервере является лучшим способом сбора информации о производительности при минимизации дополнительной нагрузки на сервер.

Трассировка, выполняемая на сервере, может быть определена и реализована с помощью набора системных хранимых процедур. Программный код вы можете написать самостоятельно или с помощью программы SQL Server Profiler.

Когда трассировка сконфигурирована и протестирована в SQL Server Profiler, выберите в меню команду File⇨Export⇨Trace Definition⇨For SQL Server 2005, чтобы сгенерировать сценарий T-SQL, способный выполнять трассировку на стороне сервера.



Чтобы узнать, какие трассировки запущены на сервере, выполните запрос к динамическому представлению управления `systraces`. Когда вы посмотрите на результаты этого запроса, то увидите дополнительную трассировку. Первый номер всегда имеет так называемая трассировка по умолчанию, которая собирает данные для журналов SQL Server, — ее невозможно остановить.

Для остановки серверной трассировки используйте системную хранимую процедуру `sp_trace_setstatus`. Первый ее аргумент (`traceid`) является идентификатором трассировки, а второй определяет характер действия. Нулевое значение параметра действия приводит к остановке трассировки, единица — к ее запуску, а двойка — к закрытию и удалению. В следующем коде останавливается трассировка с номером 2:

```
EXEC sp_trace_setstatus 2, 0
```

Использование Transact-SQL

SQL Server предлагает несколько средств исследования данных производительности с помощью T-SQL.

Использование динамических представлений управления

Динамические представления управления позволяют оценить текущее внутреннее состояние SQL Server и могут предоставить пользователю массу информации, бо́льшая часть которой исключительно полезна для оптимизации.

В частности, `sysdm_exec_cached_plans`, `sysdm_exec_query_stats`, `sysdm_exec_query_plan()` и `sysdm_exec_requests` предоставляют разнообразную информацию о текущих процессах.

Использование функции GetDate()

Вычисление длительности выполнения хранимой процедуры в T-SQL является тривиальной задачей и исключительно полезна для регистрации информации о производительности. Установка переменной datetime в результат функции GetDate() в начале процедуры является решением этой задачи:

```
CREATE PROC PerfTest
AS
SET NoCOunt ON
DECLARE @Duration DATETIME
SET @Duration = GetDate()
-- Программа
SET @Duration = GetDate() - @Duration
INSERT PerfLog (ProcName, ExecDateTime, Duration)
   VALUES('PerfTest', GetDate(), @Duration)
RETURN
```

Использование статистики

SQL Server также способен интерактивно предоставлять статистику при выполнении отдельных запросов в утилите Management Studio. Установка статистики в состояние on указывает серверу передавать статистику о времени выполнения запросов или операций ввода-вывода вместе с результатами запросов.

```
USE OBXKites
Set statistics io on
SELECT LastName + ' ' + FirstName as Customer, Product.[Name],
Product.code
  FROM dbo.Contact
    JOIN dbo.[Order]
      ON Contact.ContactID = [Order].ContactID
    JOIN dbo.OrderDetail
      ON [Order].OrderID = OrderDetail.OrderID
    JOIN dbo.Product
      ON OrderDetail.ProductID = Product.ProductID
  WHERE Product.Code = '1002'
  ORDER BY LastName, FirstName

Set statistics io off
Set statistics time on
SELECT LastName + ' ' + FirstName as Customer
  FROM dbo.Contact
  ORDER BY LastName, FirstName
Set statistics time off
go

Set showplan_all on
go
SELECT LastName
  FROM dbo.Contact
go
Set showplan_all off
go

Set showplan_xml on
go
```

```
SELECT LastName
  FROM dbo.Contact
go
Set showplan_xml off
go
```

Ключевой индикатор производительности базы данных

Ключевой индикатор производительности (далее KPI) базы данных является единственным полноценным показателем, способным оценить общую производительность базы на уровне управления в течение времени.

Единообразие является критичным фактором в оценке любого KPI. Определение базового уровня, а затем постоянное повторение абсолютно того же измерения является единственным способом выявления тенденций и оценки результатов оптимизации. При этом тесты должны повторяться через регулярные промежутки времени.

Ключевой индикатор производительности должен сравнивать данные, полученные в "чистой" тестовой среде, а также фактические данные, принимая во внимание эффект масштабирования системы. Чтобы получить реальный индикатор KPI, команда разработчиков должна предъявить особые требования к тестам и методам вычисления. Я рекомендую масштабировать результаты трех последовательных тестов, чтобы получить в результате каждого одну треть и чтобы их сумма, т.е. изначальная полноценная метрика, составляла единицу, как показано на рабочем листе и диаграмме рис. 49.7.

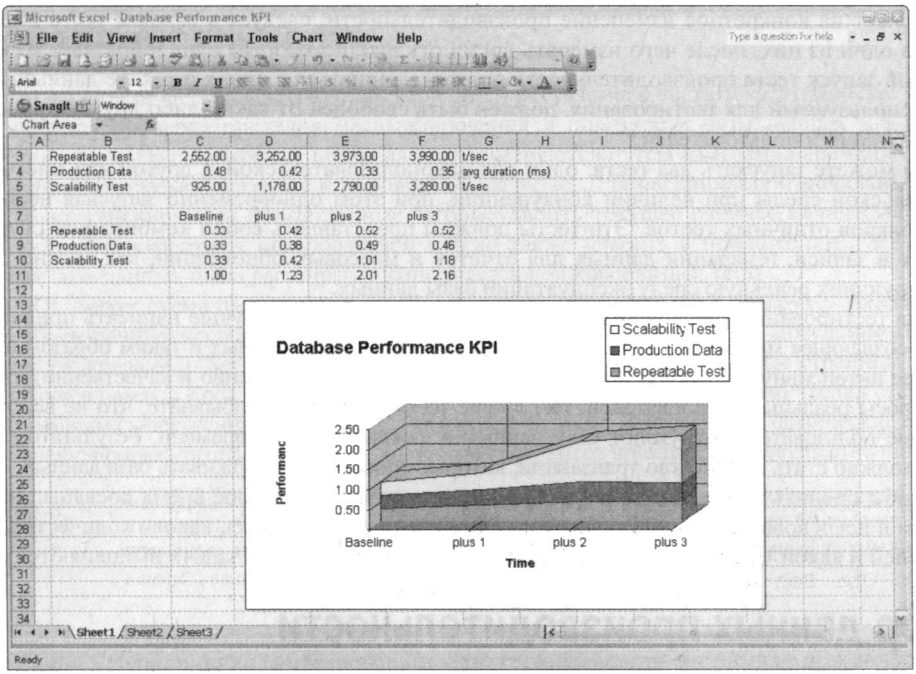


Рис. 49.7. Ключевой индикатор производительности вычисляется на основе трех тестов в программе Excel



Полноценная метрика производительности базы данных в виде рабочей книги Excel доступна на сайте книги.

Периодическое тестирование производительности

При периодическом тестировании измеряется производительность базы данных в реальных условиях на основе сценария и базы данных, созданных на этапе тестирования. Время выполнения может регистрироваться в переменных T-SQL, а также в программе SQL Server Profiler.

Если архитектура использует уровень абстракции данных, то все вызовы тестов должны выполняться именно к нему. Если же база данных использует более необычный подход с помощью модели "клиент/сервер", то при тестировании придется смешивать различные запросы. Если база данных использует архитектуру, ориентированную на Web-службы, естественно, потребуются запросы именно к этим службам. Для балансировки операций извлечения и обновления данных для таблиц потребуется создать достаточно представительную выборку, учитывающую особенности клиентских приложений и пакетных процессов.

Изначальное выполнение повторяющегося теста производительности устанавливает базовую линию измерения. При каждом последующем выполнении теста исходная база данных восстанавливается и запускается исходный сценарий. Повторяющийся тест производительности наиболее полезен для оценки результатов оптимизации в "чистой" среде — т.е. неподверженной другим процессам, недоступной другим пользователям и изменениям данных.

На скорость отклика системы влияют многочисленные факторы. Чтобы оценить, какой из них влияет на конкретное изменение производительности, следует в каждом тесте изменять только один из них, после чего измерять время отклика некоторым единообразным методом. Каждый запуск теста производительности должен начинаться с одних и тех же данных, а сервер, используемый для тестирования, должен быть свободен от каких-либо других процессов, подключений и пользователей.

Вы можете запускать два теста: один для однопользовательской, а другой для многопользовательской среды при наличии конкуренции, при этом одновременно запуская несколько экземпляров отличных тестов. Эти тесты должны представлять собой комбинацию операции чтения и записи, генерации данных для отчетов и массовых обновлений, как можно ближе имитирующих реальную среду эксплуатации базы данных.

Для тестирования сценария многопользовательского режима лучше написать приложение .NET, создающее множество подключений и обращений к базе данных и таким образом имитирующее интенсивную загрузку. Чем точнее эта нагрузка (количественно и качественно) отражает запросы реальных пользователей, тем лучше тестирование. Не забывайте, что не все конкурентные пользователи загружают базу данных в каждый момент времени. Результатом этого теста должно стать количество транзакций, которые способна поддерживать база данных.

Чтобы оценить необходимое количество транзакций, я некоторое время посвящал наблюдениям и исследованиям (прогуливался по организации и записывал, каково количество пользователей и какой тип задач они выполняют, а также как часто эти задачи выполняются).

Сбор данных производительности

Измеряя реальное время отклика хранимых процедур уровня абстракции данных, тест производительности в производственной среде собирает информацию о деятельности пользователей.

Данные о производительности в производственной среде можно собирать несколькими способами.

- SQL Server Profiler может осуществлять трассировку на стороне сервера и собирать время выполнения всех хранимых процедур. Трассировки могут записываться в базу данных и анализироваться для тонкой настройки производительности или получения данных о реальной загрузке. Такую трассировку легко запустить и остановить, что предоставляет вам полную свободу в выборе времени проведения измерений.
- Программный код в конце каждой хранимой процедуры может вычислять продолжительность ее выполнения и записывать результат в журнал для дальнейшего анализа. В зависимости от конфигурации трассировки в программе Profiler, этот метод, скорее всего, даст вам меньше деталей, чем сам SQL Server Profiler. В зависимости от сложности кода протоколирования, возникает еще одна проблема, связанная с этим методом: такое протоколирование достаточно сложно отключить, если вы решили проводить измерения только в течение нескольких часов или дней.
- Программный код приложения может протоколировать информацию от времени щелчка пользователем на кнопке отправки формы до завершения обновления формы. Несмотря на то что этот метод дает полную картину реальной деятельности пользователей, он предоставляет наименьший объем информации, необходимой для настройки.

В то время как все представленные методы накладывают на базу данных дополнительную нагрузку и существует множество переменных величин, я считаю, что SQL Server Profiler обеспечит вас лучшей информацией с минимальными затратами.

Тестирование влияния масштабирования на производительность

Этот вид теста является всего лишь вариацией регулярного теста эффективности. Его целью является проверка пределов масштабирования и полезности базы данных в будущем. Возможности масштабируемости базы данных измеряются путем измерения времени отклика при полной ожидаемой загрузке и половинной загрузке. Для расширения изначальных рабочих таблиц генерируются новые данные, чтобы удвоить текущее рабочее количество строк. Количество конкурентных пользователей можно также удвоить в приложении .NET. После того как база данных была расширена, выполняется повторяющийся тест.

В SQL Server схема базы данных и запросы могут тестироваться на загрузку только тогда, когда объем данных превосходит емкость памяти сервера в несколько раз. Поведение индексов отличается, когда SQL Server имеет возможность загрузить все необходимые страницы данных и индексов в память. Когда все эти страницы находятся в памяти, логические чтения становятся единственным существенным фактором. Так как при этом не используются физические обращения к диску, производительность самих индексов становится практически несущественной.



СУБД SQL Server оптимизирована для интеллектуального кэширования данных в памяти, что влияет на последующие тесты. Память может быть пополнена с помощью остановки и перезапуска сервера, а также с помощью команд `DBCC DropCleanBuffers` и `DBCC FreeProcCache`.



В пакет обновлений SP1 были добавлены новые динамические представления управления, выполняющие мониторинг выделения памяти для запросов, — `sys.dm_exec_query_memory_grants` и `sys.dm_exec_query_resource_semaphores`.



В примерах создания ключевых индикаторов производительности базы данных вы можете использовать базу `PerfTest`, доступную на сайте книги.

Резюме

Измерение производительности является главным аспектом настройки и оптимизации базы данных. Для измерения общей производительности базы данных ваша организация может использовать ключевые индикаторы производительности (KPI), а для ее оптимизации потребуются детальные показатели эффективности, способные помочь идентифицировать проблему и устранить ее. Программы SQL Server Profiler и System Monitor, вместе с новыми динамическими представлениями управления, предоставляет вам детальную информацию о производительности базы данных.

Освоив методы наблюдения за эффективностью оптимизации, в следующей главе мы изучим настройку ключевых средств SQL Server — запросов и индексов. Благодаря средствам измерения и умению настраивать запросы и индексы вы сможете создавать высокопроизводительные базы данных.

Анализ запросов и настройка индексов

ГЛАВА

50

В этой главе...

Индексация с целью повышения производительности

Интерпретация плана выполнения запроса

Стратегии баз данных, повышающие производительность

Основные принципы информационной архитектуры, изложенные в главе 1, гласят, что главная предпосылка к планированию любых хранилищ данных — это ознакомление с требованиями к производительности.

Теория оптимизации проложила путь к достижению этой цели и заложила фундамент для проектирования высокопроизводительных баз данных, выявив зависимости между различными уровнями оптимизации.

Индексам теория оптимизации отводит средний уровень. Они зависят от качества проектирования нормализованной схемы и запросов. Без этих двух составляющих сложно ожидать от индексации хороших результатов.

Нет никаких сомнений, что индексация жизненно важна для высокой производительности базы данных. На индексы следует смотреть с перспективы общей модели базы. Они зависят от уровней схемы и запросов и в то же время обеспечивают настройку масштабируемости и конкуренции сервера.

Анализ запросов и настройка индексов являются областью, в которой СУБД SQL Server 2005 преуспела. Сервер предоставляет обширную информацию о плане выполнения запроса, его индексы быстры, строятся по алгоритму сбалансированного дерева, их легко настраивать, и даже без всякой настройки они прекрасно работают.

Глобальный подход к настройке индексов

SQL представляет собой описательный, декларативный язык. Это значит, что запросы SQL как бы задают вопросы, а SQL Server самостоятельно решает, как на них ответить. Исходя из этого, львиная доля оптимизации выполняется на уровне SQL Server, а не самих запросов.

Рассматривая примерную стоимость каждой логической операции на основе распределения данных, доступных индексов и возможностей оборудования, оптимизатор запросов (Query Optimizer) строит дерево логических операций, обеспечивающих наискорейшее выполнение поставленной задачи, т.е. общий план выполнения запроса.

Таким образом, оптимизация запросов большей частью является вопросом создания правильных индексов, таких, чтобы оптимизатор запросов смог отобрать данные наиболее быстрым способом. Только в исключительных случаях изменение структуры запроса может повлиять на производительность. В то же время большинство производственных запросов, написанных тремя различными способами, вернут идентичные (а то и полностью совпадающие) планы выполнения.

Настройка индексов является навыком, требующим глубокого понимания нескольких независимых факторов. На самом деле настройку запросов усложняет то, что сам оптимизатор запросов является чем-то вроде черного ящика. И перед тем как вы сможете предсказать его поведение, вы должны понять взаимосвязь следующих факторов.

- Страничная архитектура SQL Server, модель физической схемы и логические операторы плана выполнения запроса.
- Распределение данных, статистика индексов, выбор индексов оптимизатором запросов и обеспечение обслуживания индексов.
- Кластеризованная структура индексов, коэффициент заполнения, слияние страниц и обслуживание индексов.
- Запросы, индексы, планы выполнения запросов и оптимизатор запросов.
- Повторное использование плана запроса и применение параметров.

Как разработчик или администратор баз данных вы должны понимать, какие индексы являются лучшими среди массы производственных запросов, а не только одного запроса.

Самое лучшее, что я могу порекомендовать, — это изучить все факторы, а затем провести собственные эксперименты, чтобы понять все сложности корреляции между данными, схемой, индексами и запросами. Корректировка индексов не оказывает никакого влияния на логический результат любого запроса. Администраторам баз данных не следует бояться экспериментировать с индексами.

Индексация

Предметный указатель книги помогает читателям быстро найти ответ на интересующий их вопрос. Лучшим предметным указателем считается тот, который открывает читателю кратчайший путь от постановки им вопроса до нахождения соответствующего текста в книге. Индексы базы данных играют такую же роль.

Несмотря на то что индексы могут быть сложными, цель индексации проста: сокращение числа читаемых физических страниц.

Основы индексации

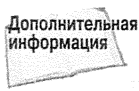
Существуют два основных типа индексов. Предметный указатель, который физически сортирует текстовые страницы, подобно телефонной книге, называется *кластеризованным*, так как сами данные выстроены в специфическом порядке. Этот тип предметного указателя физически находится в тексте книги. Естественно, текст может быть отсортирован в данном случае только в одном порядке. Аналогично, таблицы SQL Server могут иметь только один физический порядок сортировки, и именно он называется *кластеризованным индексом* таблицы.

Некластеризованный индекс подобен предметному указателю в конце книги — он отправляет читателя на определенные страницы. Любой вопрос может быть без труда найден в тексте, если вначале найти его в предметном указателе, а затем отправиться по ссылке на конкретную страницу. SQL Server также располагает некластеризованными индексами, которые выполняют сортировку, используя столбцы, отличные от кластеризованного индекса.

Столбцы, используемые в индексах, называют *ключевыми*.

Так как первичные ключи реализуют уникальный метод идентификации любой строки, индексы и первичные ключи взаимосвязаны — первичный ключ должен быть индексирован, но в то же время он может быть кластеризованным или некластеризованным.

При планировании индексов существует некоторое напряжение в отношениях между обслуживанием запросов отбора и запросов обновления. Несмотря на то что индексы повышают производительность запросов, за это приходится расплачиваться дополнительными затратами на обновление индексов при вставке и обновлении строк. Однако некоторые индексы необходимы при операциях записи. Операции обновления и удаления перед осуществлением физической записи должны отобрать строки, а хорошие индексы могут ускорить этот отбор, одновременно повышая эффективность самой записи.



Индексы таблиц нельзя путать с индексированными представлениями. Редакция Enterprise Edition содержит функцию денормализации, которая строит кластеризованный индекс, распространяющийся на множество таблиц. Неправильное использование индексированных представлений может существенно понизить производительность операционных баз данных. В главе 53 мы более подробно поговорим об индексированных представлениях.

Кластеризованные индексы

Кластеризованный индекс представляет собой инструмент слияния сбалансированого дерева со страницами данных таблиц, поддерживающий данные в том же физическом порядке, что и индекс (рис. 50.1). Листовые узлы сбалансированного дерева индекса фактически являются данными страниц данных.

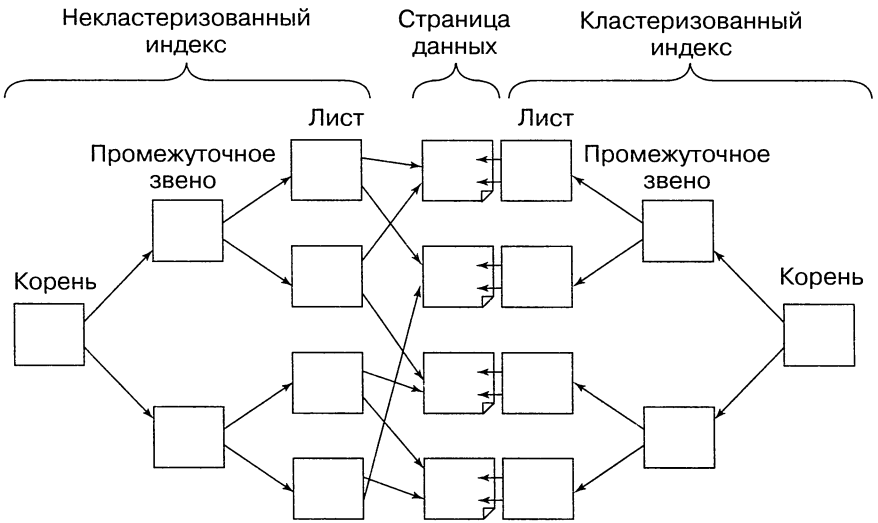
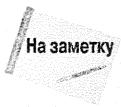


Рис. 50.1. Кластеризованный индекс связывает листовые узлы страниц индексов со страницами данных, обеспечивая тот же порядок данных, что и в индексе

Кластеризованный индекс может повлиять на производительность одним из описанных ниже способов.

- Когда операция поиска находит строку, используя кластеризованный индекс, нужные данные находятся именно в ней. Это делает столбец, используемый для идентификации строк, возможным кандидатом для первичного ключа и идеальным — для кластеризованного индекса.
- Кластеризованные индексы группируют строки с одинаковыми или сходными значениями в наименьшее возможное количество страниц, сокращая тем самым количество строк данных, необходимых для извлечения набора строк. Таким образом, кластеризованные индексы идеально подходят для столбцов, которые наиболее часто используют для отбора диапазонов строк (в качестве примера можно привести столбец `OrderDetail.OrderID`).



Любая таблица имеет некоторый физический порядок. Если таблица не имеет кластеризованного индекса, то она находится в виде *кучи*, т.е. в неупорядоченном виде. В этом случае строки, при невозможности идентификации по ключевому столбцу кластеризованного индекса, идентифицируются с помощью внутреннего идентификатора строки (`RowID`) кучи. Этот идентификатор не используется в запросах.

Некластеризованные индексы

Некластеризованным называется индекс в виде сбалансированного дерева, начинающегося с корневого узла и растущего через промежуточные узлы к листовым узлам. Листовые узлы указывают непосредственно на строки страниц данных (см. рис. 50.1). Таблица SQL Server 2005 может иметь до 249 некластеризованных индексов, но мне на практике не встречались таблицы, требующие больше десяти хорошо продуманных индексов.

Некластеризованный индекс может быть создан по вычисляемому столбцу. Для включения возможности создания индекса или индексированного представления по вычисляемому столбцу следует установить для параметра `quoted_identifier` значение `on`.

Создание индексов

В окне Object Explorer утилиты Management Studio существующие индексы каждой таблицы перечислены под узлом Databases⇨Tables⇨Indexes. Свойства каждого нового или существующего индекса можно корректировать в диалоговом окне Index Properties (рис. 50.2). Это окно открывается для существующего индекса щелчком правой кнопки мыши на его имени и выбором в контекстном меню пункта Properties. Новые индексы создаются с помощью контекстного меню узла Indexes конкретной таблицы.

В утилите Management Studio индексы отображаются как узлы на панели Object Explorer. С помощью выбора в контекстном меню узла Indexes пункта New Index можно создать новый индекс. В открывающейся при этом форме содержатся четыре вкладки.

- General. Содержит имя индекса, его тип, свойство уникальности, а также ключевые столбцы.
- Options. Управляет режимом работы индекса. Здесь же любой индекс может быть отключен и снова включен.
- Included Columns. Содержит неключевые столбцы, служащие оболочкой индекса.
- Storage. Позволяет поместить индекс в выбранную файловую группу.

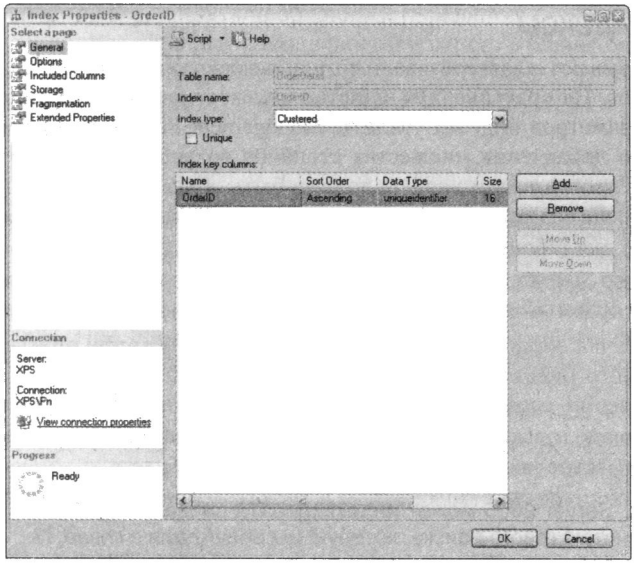


Рис. 50.2. Параметры любого индекса можно установить в диалоговом окне Index Properties утилиты Management Studio

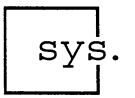
При открытии окна параметров существующего индекса оно содержит также и две дополнительные вкладки.

- Fragmentation. Отображает детальную информацию о состоянии индекса.
- Extended Properties. Содержит дополнительные параметры, определяемые пользователем.

Изменения, которые вносят в окне параметров индекса, можно применить немедленно с помощью щелчка на кнопке OK; их применение можно отложить на заданное время, а также воплотить все изменения в сценарий. Для этого можно воспользоваться значками в верхней части окна.

В программном коде индексы создаются с помощью инструкции `CREATE INDEX`. В следующем примере создается кластеризованный индекс `IxOrderId`, основанный на внешнем ключе `OrderID` таблицы `OrderDetail`:

```
CREATE CLUSTERED INDEX IxOrderID
  ON dbo.OrderDetail (OrderID);
```

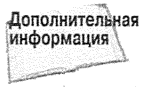


Для извлечения исчерпывающей информации об индексах с помощью программного кода используют следующие функции и представления каталогов: `sysindexes`, `sysindex_columns`, `sysstats`, `sysstats_columns`, `sysdm_db_index_physical_stats`, `sysdm_index_operational_stats`, `sysindexkey_property` и `sysindex_col`.

Кластеризованный индекс создается автоматически при определении первичного ключа.

Для удаления индекса используется инструкция `DROP INDEX`, в которой указывается имя таблицы и имя индекса, например:

```
DROP INDEX OrderDetail.IxOrderID
```



Созданные индексы не поддерживают эффективное состояние автоматически. Операции обновления могут фрагментировать индексы и изменять коэффициент заполнения их страниц. В этой главе мы лишь упоминаем о необходимости обслуживания индексов, тогдп как в главе 37 рассказывалось о требованиях, необходимых для обеспечения производительности индексов.

Составные индексы

Составным называют кластеризованный или некластеризованный индекс, включающий множество столбцов. На практике большинство индексов являются составными. Если вы используете окно параметров индекса утилиты Management Studio, то составные индексы создаются с помощью добавления множества столбцов во вкладке General. Для создания составного индекса в программном коде он должен быть объявлен с помощью инструкции DDL `CREATE INDEX` после создания таблицы. В следующем примере создается составной кластеризованный индекс для таблицы `GUIDE` базы данных CHA2:

```
CREATE CLUSTERED INDEX IxGuideName
  ON dbo.Guide (LastName, FirstName);
```

Порядок столбцов в составном индексе очень важен. Чтобы при поиске получить все преимущества составного индекса, последний должен содержать наиболее часто используемые столбцы в направлении слева направо. Если составной индекс выглядит как `lastname, firstname`, то поиск только по `firstname` не будет использовать индекс, а поиск по `lastname` или совместно по `lastname` и `firstname` — будет.



SQL Server 2005 может индексировать слова в столбцах с помощью функции полнотекстового поиска, которую мы обсуждали в главе 13.

Первичные ключи

Первичный ключ может быть изначально определен как кластеризованный или некластеризованный индекс. Однако, чтобы изменить тип индекса, ограничение первичного ключа должно быть удалено или снова создано. Это достаточно трудная задача, если присутствует множество внешних ключей или если таблица реплицируется.



О создании первичных ключей см. в главе 17.

Покрывающие индексы

Покрывающим называют любой индекс, который полностью удовлетворяет потребностям конкретного запроса `SELECT`. Так как СУБД SQL Server сама выбирает, какой индекс использовать для поиска данных, существует вероятность, что в некластеризованном индексе запросу понадобятся все столбцы. В данном случае реляционное ядро будет извлекать данные с индексных страниц и никогда не будет осуществлять чтение со страниц данных. Это существенно сокращает операции ввода-вывода, так как чтение осуществляется из более узких таблиц, содержащих на одной странице больше данных, а дополнительное чтение со страниц данных вообще не выполняется.

При проектировании покрывающего индекса очень важно осознавать, как кластеризованный индекс влияет на некластеризованный. Поскольку некластеризованный индекс должен иметь способность обращаться к страницам данных, на своих листовых узлах он должен содержать ключевой столбец кластеризованного индекса (если в таблице таковой существует). При этом столбцы кластеризованного индекса включаются в конец каждого некластеризованного индекса (даже если вы их не видите явно в диалоговом окне свойств индекса). Например, если некоторая таблица имеет кластеризованный индекс по столбцу `ContactID` и некластеризованный по столбцам `LastName` и `FirstName`, то некластеризованный индекс содержит данные из столбцов `LastName`, `FirstName` (отсортированные), а также `ContactID` (неотсортированные). Знание этого факта очень важно при проектировании покрывающих индексов.

Покрывающий индекс может иметь необходимость включать некоторые столбцы, которые не требуют реляционное ядро для идентификации строк, отсеиваемых предложением WHERE. Эти дополнительные, неключевые столбцы не испытывают потребность в сортировке в сбалансированном дереве индекса — они включаются исключительно с целью возвращения столбцов запросом SELECT.



Недостатком покрывающих индексов в предыдущих версиях SQL Server было то, что при обновлениях должны были сортироваться все столбцы — даже те, которые не использовались при отборе строк, а были включены исключительно для возвращения данных. Способность версии SQL Server 2005 отделять неключевые столбцы повысила производительность операций обновления с использованием покрывающих индексов, в которых не сортируются неключевые столбцы. При этом за счет уменьшения размера сбалансированного дерева индекса улучшилась производительность извлечения данных запросом.

Для определения неключевых столбцов в некластеризованных индексах используется параметр INCLUDE. Эти неключевые столбцы не сортируются как часть структуры сбалансированного дерева индекса и включаются только в его листовые узлы. В следующем примере создается индекс, сортирующий данные по столбцу OrderNumber и включающий данные из столбца OrderDate:

```
CREATE NONCLUSTERED INDEX IxOrderNumber
  ON dbo.[Order] (OrderNumber)
  INCLUDE (OrderDate);
```

Благодаря включаемым столбцам как узкой копии широкой таблицы, которые SQL Server постоянно синхронизирует с исходной таблицей, для извлечения данных не требуется чтение дополнительных страниц.

Включаемые столбцы не учтены в ограничениях некластеризованного индекса — 16 ключевых столбцов и 900 байтов. На самом деле в покрывающий индекс можно включить до 1023 неключевых столбцов. В качестве включаемых могут быть также и столбцы с особо крупными типами данных — XML, varchar(max), nvarchar(max) и varbinary(max), — даже несмотря на то, что они не могут выступать в роли ключевых столбцов.

Столбец таблицы нельзя разделить, если он участвует в роли включаемого столбца в покрывающем индексе. Перед тем как разделить такой столбец, следует удалить покрывающий индекс.

Местонахождение файловой группы

Если база данных использует множество именованных файловых групп, то индекс может быть создан в определенной из них. Для этого в определении индекса используется параметр ON *имя_файловой_группы*:

```
CREATE NONCLUSTERED INDEX имя_индекса
  ON Table (столбцы)
  ON имя_файловой_группы;
```

Этот параметр может оказаться полезным для распределения потоков ввода-вывода в особо сильно загруженных базах данных. Например, если Web-сайт запрашивается миллионном пользователей в минуту, его главная страница использует запрос, включающий две таблицы и три индекса, и при этом доступно несколько дисковых подсистем, то помещение каждой таблицы и ее индексов в собственную дисковую подсистему значительно повысит производительность. Следует заметить, что кластеризованный индекс должен находиться в одной дисковой подсистеме со связанной с ним таблицей, так как их страницы объединены.



Физическое размещение таблиц и индексов может быть сконфигурировано и более глубоко, с использованием файловых групп и разделов. Более подробно эти темы мы обсудим в главе 53.

Параметры индексов

Индексы SQL Server 2005 имеют несколько параметров, в том числе уникальность, выделение пространства, а также параметры производительности.

Уникальные индексы

Параметр `UNIQUE INDEX` является кое-чем бо́льшим, нежели просто индексом с ограничением на уникальность, — уникальным индексам доступна оптимизация. Первичный ключ или ограничение на уникальность автоматически создают уникальный индекс.

В Management Studio уникальный индекс создается с помощью установки флажка Unique во вкладке General диалогового окна параметров индекса.

В программном коде уникальность индекса указывается с помощью добавления в определение ключевого слова `UNIQUE`:

```
CREATE UNIQUE INDEX OrderNumber
  ON [Order] (OrderNumber);
```

Коэффициент заполнения индекса

Любому индексу нужно немного свободного пространства в дереве, чтобы вставка новых записей не приводила к реструктуризации индекса. Когда серверу нужно вставить новую запись в заполненную страницу, он разбивает эту страницу на две, после чего записывает две наполовину полные страницы на диск. Такой ход вещей вызывает три потенциальные проблемы: разбивается сама страница, новые страницы больше не являются последовательными, и на каждой странице содержится меньший объем информации. В результате для чтения того же объема данных потребуется просмотреть большее количество страниц.

Поскольку индекс представляет собой сбалансированное дерево, каждая страница должна содержать как минимум две строки. Коэффициент заполнения и параметр `pad index` влияют как на промежуточные страницы, так и на листовые узлы, как показано в табл. 50.1.

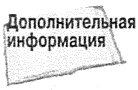
Таблица 50.1. Коэффициент заполнения и параметр `pad index`

Коэффициент заполнения	Промежуточные страницы	Листовой узел
0	Одна свободная запись	100%-ное заполнение
1–99	Одна свободная запись или объем, меньший коэффициента заполнения, если установлен параметр `pad index`	Объем, меньший коэффициента заполнения
100	Одна свободная запись	100%-ное заполнение

Коэффициент заполнения применяется только к листовым узлам индекса, если к нему не применен параметр `pad index`. Этот параметр указывает серверу применять слабость коэффициента заполнения также и к промежуточным уровням сбалансированного дерева.



Наилучший коэффициент заполнения зависит от назначения базы данных и типа кластеризованного индекса. Если база данных в основном предназначена для извлечения данных или первичный ключ является последовательным, то высокий коэффициент заполнения позволит максимально упаковать индексные страницы. Если кластеризованый индекс не последовательный (например, естественный первичный ключ), то таблица потенциально восприимчива к разделению страниц, — в этом случае используйте низкий коэффициент заполнения и часто выполняйте дефрагментацию страниц.



Коэффициент заполнения индекса по мере разделения страниц постепенно утрачивает свою роль. Для восстановления коэффициента заполнения план обслуживания должен включать в себя периодическую реиндексацию. Информация о порядке поддержки индексов содержится в главе 37.

В Management Studio коэффициент заполнения устанавливается во вкладке Options диалогового окна параметров индекса. В программном коде T-SQL параметры коэффициента заполнения и `index pad` указываются после команды `CREATE INDEX`. В следующем примере создается индекс `OrderNumber` с 15% свободного пространства как на листовых узлах, так и на промежуточных страницах:

```
CREATE NONCLUSTERED INDEX IxOrderNumber
  ON dbo.[Order] (OrderNumber)
  WITH FILLFACTOR = 85, PAD_INDEX = ON;
```



Управление блокировкой индексов, их создание в реальном времени и ограничение параллелизма индексов являются новыми параметрами, появившимися в версии SQL Server 2005. Для параметров `pad_index`, `fillfactor`, `sort_in_db`, `ignore_dup_key`, `statistics_norecompute` и `drop_existing` был изменен синтаксис. Новый синтаксис требует обязательного включения выражения `=on`. Из соображений обратной совместимости продолжает поддерживаться и старый синтаксис, однако в будущих версиях этого уже не будет.

Ограничение блокировок и параллелизма

Режим работы блокировок в запросах, использующих индексы, может управляться с помощью параметров `allow_row_locks` и `allow_page_locks`. Обычно эти блокировки разрешены.

Порядок сортировки индексов

СУБД SQL Server способна создавать нисходящие индексы. Любой запрос, использующий предложение `ORDER BY`, приведет к сортировке по возрастанию, если в этом предложении не будет явно указано ключевое слово `DESC`.

В инструкции DDL `CREATE INDEX` ключевые слова `ASC` и `DESC` следуют непосредственно за именем столбца.

Параметр игнорирования дублирующихся ключей

Параметр `IGNORE_DUP_KEY` не оказывает влияния на сам индекс, а только на то, как впоследствии индекс будет влиять на изменения данных.

Обычно транзакции являются атомарными. Это значит, что транзакция либо выполняется, либо нет, как логическая единица. В то же время параметр игнорирования дублирующихся ключей направляет транзакции вставки на подтверждение операции для всех строк, удовлетворяющих условию уникальности индекса, и игнорирование всех строк, нарушающих требование уникальности индекса.

Этот параметр не нарушает уникальность индекса. Дублирующиеся записи остаются в таблице, таким образом, целостность базы данных остается нетронутой, в то время как атомарность транзакций нарушается. Несмотря на то что этот параметр значительно облегчает импорт триллионов сомнительных строк, мне не нравятся никакие параметры, ослабляющие характеристики ACID (т.е. атомарности, целостности, изолированности и живучести) базы данных.

В следующем примере повторяется предыдущая инструкция создания уникального индекса, но в ней использован параметр игнорирования дублирующихся ключей:

```
CREATE UNIQUE INDEX OrderNumber
  ON [Order] (OrderNumber)
  WITH IGNORE_DUP_KEY = ON
```

Параметр удаления существующего индекса

Параметр DROP_EXISTING указывает серверу удалить текущий индекс и воссоздать его с нуля. Это может привести к некоторому выигрышу по сравнению с перестройкой существующего индекса в случае, если он является кластеризованным, а таблица содержит также и некластеризованные индексы. Это связано с тем, что перестройка кластеризованного индекса приводит к автоматической перестройке всех некластеризованных индексов.

Параметр запрета пересчета статистики индекса

Оптимизатор запросов SQL Server зависит от статистики распределения данных в вопросе определения, какой из индексов является более существенным в конкретном критерии поиска в таблице. Обычно SQL Server обновляет эту статистику автоматически. В то же время непосредственно перед запросом некоторые таблицы могут получить большой объем данных, и в этом случае статистика может оказаться устаревшей. Специально для ситуаций, в которых статистика требует обновления вручную, предназначен параметр statistics_norecompute, отменяющий автоматическое обновление статистики. Практически для всех индексов этот режим работы не рекомендуется.

Сортировка в базе tempdb

Параметр sort_in_tempdb=on изменяет метод создания индекса, форсируя использование базы tempdb, а не памяти. Если индекс постоянно удаляется и воссоздается, то этот параметр способен сократить время создания индекса. В большинстве индексов этот параметр не имеет большого значения, к тому же он не обязательный.

Отключение индекса

Любой индекс может быть временно отключен. Для этого достаточно снять флажок Use Index во вкладке Option диалогового окна параметров индекса. В программном коде T-SQL этот эффект достигается с помощью включения параметра DISABLE в инструкцию DDL ALTER INDEX:

```
ALTER INDEX [IxContact] ON [dbo].[Contact] DISABLE
```

В некоторых особо интенсивных операциях импорта оказывается быстрее удалить индекс и воссоздать его, чем обновлять его при вставке каждой строки. Преимущество отключения индекса заключается в том, что метаданные индекса обслуживаются в самой базе данных, а не зависят от программного кода, воссоздающего корректный индекс.



Отключение кластеризованного индекса фактически приводит к отключению всей таблицы.

Чтобы снова включить индекс, используется команда ALTER INDEX...REBUILD WITH:

```
ALTER INDEX [PK__Contact__0BC6C43E]
  ON [dbo].[Contact]
  REBUILD WITH
  ( PAD_INDEX = OFF,
    STATISTICS_NORECOMPUTE = OFF,
     ALLOW_ROW_LOCKS = ON,
```

```
ALLOW_PAGE_LOCKS = ON,
SORT_IN_TEMPDB = OFF,
ONLINE = OFF )
```

Создание базовых индексов

Даже перед настройкой местонахождение нескольких индексов легко определить. Эти базовые индексы являются первым шагом в создании цельного набора индексов. Ниже приведено несколько рекомендаций, которыми стоит воспользоваться перед созданием этих базовых индексов.

1. Создайте кластеризованный индекс для каждой из таблиц. В первичных таблицах такой индекс лучше организовать на столбцах, наиболее вероятно используемых для выбора строк, — отличным кандидатом является первичный ключ. Во вторичных таблицах, в которых чаще всего извлекается множество связанных строк, создавайте кластеризованный ключ для наиболее важного внешнего ключа, группирующего вместе эти связанные столбцы.
2. Создайте некластеризованные индексы для столбцов каждого внешнего ключа, за исключением тех, которые были проиндексированы в п. 1. В качестве ключей индекса используйте только значения внешнего ключа.
3. Создайте одностолбцовый индекс для всех столбцов, которые наиболее вероятно будут появляться в предложениях `WHERE`, `ORDER BY` или `GROUP BY`.

В то время как предложенный план индексации далек от совершенства, он предлагает изначальный компромисс между отсутствием индексов и настроенными индексами. В принципе его можно рассматривать как базовый уровень производительности по отношению к последующей настройке индексов.

Анализ запросов

После ознакомления с основами индексов и страниц SQL Server вторым этапом в освоении настройки индексов является рассмотрение плана выполнения запроса. Это предполагает умение читать план выполнения и идентифицировать действия, которые оптимизатор решил предпринять для выполнения запроса.

Просмотр плана выполнения запроса

Еще одной моей любимой составной частью SQL Server является редактор запросов (Query Editor), отображающий *план выполнения запроса*. Вы можете просмотреть либо предварительный, либо точный план — обе эти функции доступны на панели инструментов, а также в меню Query.

И в оценочном, и в точном плане выполнения запроса содержатся одни и те же логические операции. Единственным отличием является то, что последний можно просмотреть только после выполнения запроса.

Оценочный план использует статистику для оценки количества строк, вовлеченных в каждую из логических операций. И это может быть очень важно, поскольку оценочное количество строк, произведенное каждой логической операцией, играет большую роль в выборе операций.

План запроса следует читать слева направо, как показано на рис. 50.3. Каждая логическая операция представляется в виде пиктограммы. Однако редактор запросов представляет собой не только статическое средство просмотра.

- Помещение указателя мыши над логической операцией вызывает открытие диалогового окна, содержащего детальную информацию о ней, включая логическую стоимость и часть запроса, обслуживаемую операцией.
- Помещение указателя мыши над соединительной линией приводит к отображению информации о способе перемещения данных.

Планы можно сохранять в файлах с расширением `.sqlplan` для последующего исследования.

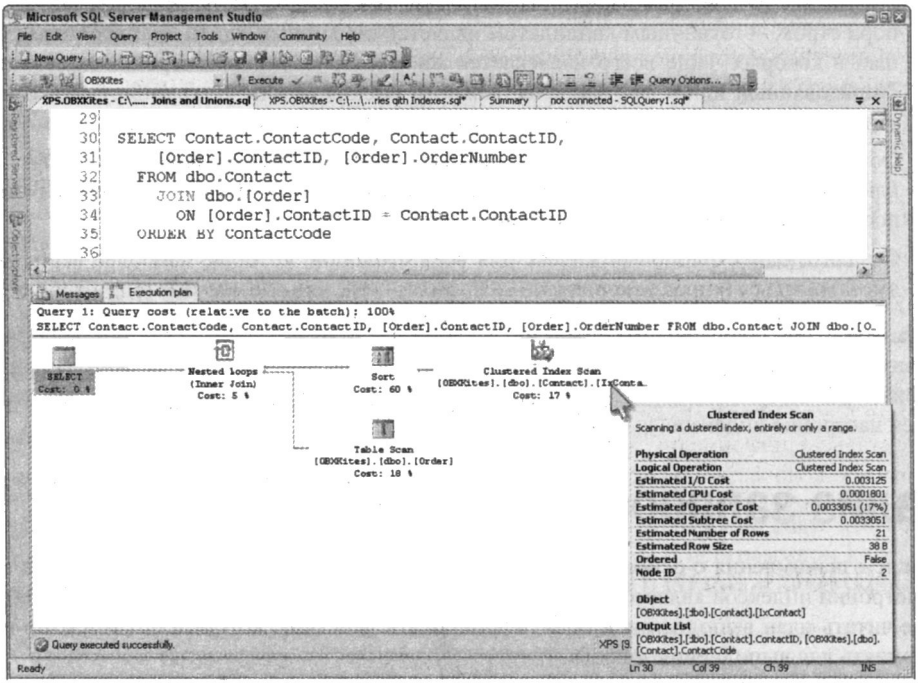


Рис. 50.3. План выполнения запроса отображает логические операции, которые SQL Server использует для выполнения запроса

Использование параметра `Showplan`

В дополнение к графическому отображению плана выполнения параметр `Showplan` раскрывает его дополнительные детали. Он доступен в трех формах: `all`, `text` и XML.

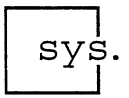
Когда параметр `showplan` включен, SQL Server возвращает план выполнения запроса, но не выполняет инструкцию, как при использовании параметра `show estimated plan`. Единственной инструкцией в пакете должна быть `Set showplan`.

Подобно параметру `show estimated plan`, SQL Server вернет план выполнения запроса, но не выполнит инструкцию. В версии XML параметр `show actual plan` должен находиться в отключенном состоянии. К тому же, если установлено табличное отображение

результатов запроса, таблица позволит открыть по ссылке файл XML, используя браузер. Работа параметра showplan описана ниже.

- Showplan_all отображает операции в виде результирующего набора данных. Эта информация эквивалентна той, которая отображается в графическом представлении. Выполняемая инструкция возвращается в первой строке, а все операторы — в последующих.
- Showplan_text очень похож на showplan_all, за исключением того, что выполняемая инструкция и операции находятся в разных результирующих наборах данных: при этом отображается только первый столбец stmt text.
- Showplan_xml отображает больше деталей, чем остальные методы просмотра плана выполнения, и предлагает преимущества хранения и отображения неструктурированных данных, позволяя, таким образом, отображать дополнительную информацию, не относящуюся ко всем планам выполнения. Например, в элементе <Statement> отображается причина того, почему оптимизатор запросов вернул этот план выполнения.

Параметр showplan_text, наряду с параметрами set statistics, может быть включен и в графическом интерфейсе редактора запросов. С помощью команды Query Options контекстного меню можно открыть диалоговое окно параметров запроса, после чего найти параметры showplan, выбрав в меню пункт Execution⇨Advanced.



Динамическое представление управления также позволяет отобразить тот же план выполнения запроса в формате XML, однако этот метод потребует некоторого изучения необходимых параметров.

Еще одним способом получения плана выполнения запроса является использование команды set statistics profile on. При этом будет выполнен запрос и предоставлена та же детальная информация, что и при использовании параметра showplan_all, включая фактическое количество строк и статистику выполнения. Это является эквивалентом плана выполнения show actual в виде результирующего набора данных.

Интерпретация плана выполнения запроса

Чтение плана выполнения запроса на первый взгляд может показаться сложным: графика — непривычна, а объем информации — ошеломляет. SQL Server использует около шестидесяти операторов. Некоторые из них представляют конкретные физические задачи, в то время как большинство представляет собой набор скрытых задач. В табл. 50.2 перечислены ключевые операторы, относящиеся к запросам отбора и индексации.


Проверено

Общим заблуждением является то, что поиск всегда хорош, в то время как сканирование всегда плохо. На самом деле это не всегда так. В связи с тем что СУБД SQL Server оптимизирована для последовательного извлечения данных, сканирование является отличным способом извлечения больших объемов данных, однако плохо подходит для извлечения нескольких изолированных строк. Поиск по индексу хорош для извлечения из разных мест по несколько строк, в то же время извлечение большей части таблицы с использованием того же метода показывает намного худшую производительность по сравнению со сканированием.

Таблица 50.2. Операторы плана выполнения запроса

Значок	Определение	Описание
	Сканирование кластеризованного индекса	При сканировании кластеризованного индекса SQL Server последовательно читает весь индекс или диапазон в нем. Сервер выбирает эту операцию, когда набор строк, запрошенный предложением `WHERE`, можно определить в виде диапазона в порядке кластеризованного индекса или когда запрашивается большой процент строк из таблицы
	Поиск в кластеризованном индексе	В этом поиске SQL Server быстро переходит к сбалансированному дереву кластеризованного индекса для извлечения конкретных строк. Преимущество такого поиска состоит в том, что когда запрашиваются строки, все столбцы становятся немедленно доступными
	Фильтр	В некоторых ситуациях SQL Server извлекает из таблицы все данные, а затем использует операции фильтрования для отбора корректных строк. Иногда оптимизатор запросов использует фильтр из соображений производительности, но чаще всего из-за отсутствия полезного индекса
	Сравнение в куче	Это неупорядоченный метод объединения, в котором создается временная таблица и выполняются последовательные сравнения с другой таблицей. Этот метод более эффективен, если одна таблица существенно больше другой. Этот метод объединения используется в крайнем случае, когда не доступны подходящие индексы
	Объединение слияния	Это самый быстрый метод сравнения двух предварительно отсортированных таблиц
	Вложенный цикл	Вложенный цикл итеративно проходит по двум таблицам и ищет соответствия. Лучше всего этот метод себя проявляет, когда большой индекс объединен с маленькой таблицей
	Сканирование некластеризованного индекса	В этом случае сервер в поиске данных последовательно читает весь индекс или его часть
	Поиск в некластеризованном индексе	Выполняется прохождение по сбалансированному дереву индекса, начиная с корневого узла, через промежуточные, к листовым, а затем к самим строкам. Для отбора нескольких отдельных строк эта операция может оказаться очень быстрой. Его преимущество в том, что он уменьшает количество столбцов в таблице, таким образом, на странице может поместиться большее количество строк. Как только корректная строка найдена, если все затребованные столбцы найдены в индексе, поиск завершается, так как индекс полностью удовлетворил потребностям запроса. Если же требуются дополнительные столбцы, они будут затребованы из страниц данных
	RID-поиск	Строки ищутся на страницах данных. Обычно этот метод работает с вложенным циклом для нахождения страниц данных, следуя за сканированием кластеризованного индекса или поиска в нем
	Сортировка	В некоторых ситуациях SQL Server извлекает из таблицы все данные, а затем сортирует их, подготавливая к предложению `ORDER BY`. Фильтрация и сортировка — медленные операции и являются индикатором отсутствия подходящих индексов

Значок	Определение	Описание
	Катушка	В этой операции SQL Server вынужден сохранить временный набор данных
	Сканирование таблицы	SQL Server последовательно читает всю таблицу для отбора строк, удовлетворяющих критерию. В зависимости от количества необходимых строк этот метод извлечения данных может оказаться самым быстрым. Например, если серверу нужно отобрать 80% таблицы, то это гораздо быстрее, чем последовательный поиск отдельных строк. Однако, если нужно всего несколько строк, а оптимизатор все равно выбирает сканирование таблицы, это значит, что либо отсутствует подходящий индекс, либо статистика индекса ошибочна, либо таблица настолько мала, что выбор метода не имеет значения



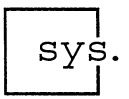
Логическая операция поиска по закладкам в SQL Server 2005 была удалена и заменена поиском в кластеризованном индексе и поиском в RID. В предыдущих версиях SQL Server отображался значок этой операции, но на самом деле выполнялись операции поиска в кластеризванном индексе и в RID.

Настройка индексов

Вооружившись знаниями об индексах и планах выполнения запроса, можно выявить некоторые проблемы производительности и решить их, используя индексы.

Отсутствие индексов

Наиболее очевидной возможностью повышения производительности с помощью индексов является создание индекса там, где он отсутствует. Однако вся хитрость состоит в выявлении нужного индекса.



SQL Server 2005 может отобразить статистику использования индексов с помощью динамических представлений управления. В частности, `sysdm_db_index_operational_stats` и `sysdm_index_usage_stats` раскрывают информацию о том, как используются индексы. Дополнительно существуют четыре динамических представления управления, которые вскрывают информацию о том, какие индексы искал оптимизатор запросов, но не смог найти. Это `sysdm_missing_index_groups`, `sysdm_missing_index_group_stats`, `sysdm_missing_index_columns` и `sysdm_missing_index_details`.

В первом приводимом примере используется простой запрос в базе данных OBXKites. Следующий простой код извлекает данные и упорядочивает их по `StartDate`:

```
SET STATISTICS TIME ON;
USE Adventureworks;
SELECT WorkOrderID
  FROM Production.WorkOrder
  ORDER BY StartDate;
```

Будет получен следующий результат (время из статистики):

```
SQL Server Execution Times:
  CPU time = 126 ms, elapsed time = 679 ms.
```

Этот запрос использует сканирование кластеризованного индекса, за чем следует операция сортировки для упорядочения данных по StartDate. В данном примере операция сортировки является признаком того, что отсутствует индекс.

Создадим индекс по StartDate, чтобы скорректировать план выполнения:

```
CREATE INDEX WOStartDate ON Production.WorkOrder (StartDate);
```

Повторный запуск запроса показывает значительное уменьшение времени работы процессора — со 126 до 63 миллисекунд. При этом новый план выполнения запроса покажет только сканирование некластеризованного индекса по WOStartDate.

Следующий, более сложный план запроса также вскрывает потребность в индексе:

```
USE OBXKites;
SELECT LastName, FirstName, ProductName
  FROM dbo.Contact
    JOIN dbo.[Order]
      ON Contact.ContactID = [Order].ContactID
    JOIN dbo.OrderDetail
      ON [Order].OrderID = OrderDetail.OrderID
    JOIN dbo.Product
      ON OrderDetail.ProductID = Product.ProductID
    JOIN dbo.ProductCategory
      ON Product.ProductCategoryID = ProductCategory.ProductCategoryID
  WHERE ProductCategoryName = 'Kite'
  ORDER BY LastName, FirstName;
```

Будут получены следующие результаты (время из статистики):

```
SQL Server Execution Times:
  CPU time = 0 ms, elapsed time = 146 ms.
```

Общее время выполнения мало ввиду небольших размеров базы данных. Тем не менее план выполнения запроса выявляет некоторые проблемы (рис. 50.4). Одна из этих проблем проявляется в том, что выполняется сканирование таблицы, объединенной с использованием операции сравнения кучи. Сравнение кучи объединяет таблицы OrderDetail и Product. Единственный индекс таблицы OrderDetail в этой копии базы OBXKites является некластеризованным первичным ключом, включающим в себя столбец OrderDetailID.

Мы начнем решение этой проблемы с создания некластеризованного индекса по полю ProductID. При наличии этого индекса сканирование таблицы и сравнение кучи заменяются сканированием индекса и вложенным циклом. Однако некластеризованный индекс не покрывает потребности запроса, поэтому генерируется дополнительная страница данных (рис. 50.5).

Поиск закладки

Несмотря на то что это не так затратно, как отсутствие индекса, самой распространенной проблемой является поиск сервером корректных строк с помощью некластеризованного индекса с последующим присоединением дополнительных строк со страниц данных. Эта ситуация известна как *поиск закладки*.

Поиск закладки может негативно сказаться на производительности ввиду того, что в то время как некластеризованный индекс может сгруппировать данные в несколько индексных страниц, поиск закладки приведет к извлечению дополнительных данных с множества страниц данных.

В качестве иллюстрации можно использовать следующий пример. Предположим, что в предметном указателе книги вы обнаружили нужную тему, но она ссылается на 200 разрозненных страниц книги. Обращение к 200 страницам в тексте книги является затратной по времени операцией (см. рис. 50.5).

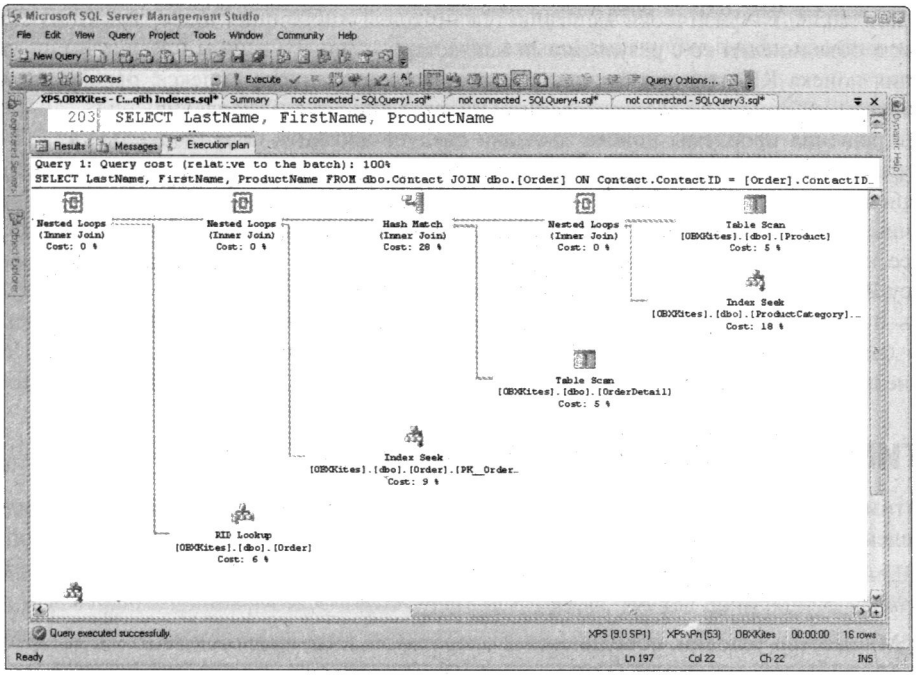


Рис. 50.4. Операции сканирования таблицы и сравнения кучи являются четким признаком отсутствия индекса

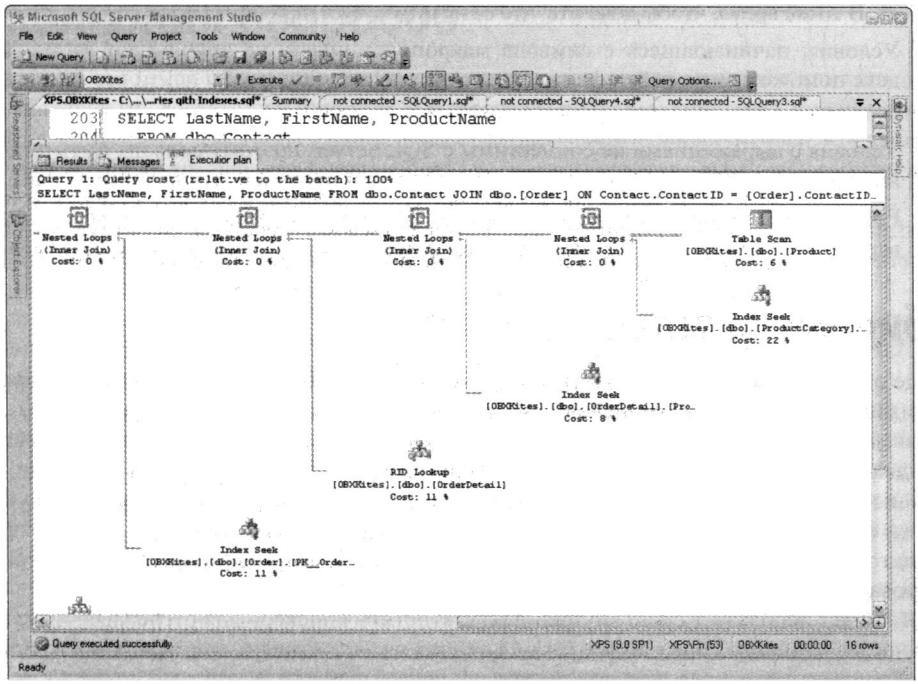


Рис. 50.5. Операция поиска в RID и вложенный цикл выявляют проблему поиска закладки

Поиск закладки физически выполняется методом извлечения информации из страниц данных и объединения ее с результатами некластеризованного индекса. В данном случае это операция поиска RID, так как не существует кластеризованного индекса, однако это может быть операцией поиска в кластеризованном индексе или его сканирования.

Для решения проблемы поиска закладки следует добавить поле `OrderDetailID` к некластеризованному индексу `ProductID`, чтобы объединяемые данные непосредственно извлекались из некластеризованного индекса. В качестве альтернативы можно преобразовать первичный ключ в кластеризованный индекс, который заменит RID в некластеризованном индексе ключами кластеризованного индекса. Оба решения позволят некластеризованному индексу `ProductID` полностью покрыть требования запроса к таблице `OrderDetail` и избежать поиска закладки. Так как данные `OrderDetail` часто извлекаются по внешнему ключу `OrderID`, он является наилучшим кандидатом на кластеризованный индекс. По этой причине лучшим решением является добавление поля `OrderDetailID` в индекс `ProductID`.

Оптимизируемый аргумент поиска

Оптимизатор запросов SQL Server проверяет условия в предложении `WHERE` запроса для определения того, какие индексы ему окажутся полезными. Если SQL Server может оптимизировать предложение `WHERE` с помощью индекса, то данное условие называется *аргументом поиска* (или SARG). Однако не любое условие является таковым.

- Множество условий, объединенных оператором `AND`, являются SARG, в то время как объединенные оператором `OR` — нет.
- Условия, содержащие отрицания (`<>`, `!>`, `!<`, `Not Exists`, `Not In` и `Not Like`), не являются оптимизируемыми. Очень легко доказать, что определенная строка существует. В то же время, чтобы доказать, что ее не существует, придется проверить все строки.
- Условия, начинающиеся с символа макроподстановки, не используют индексы. Индекс поможет легко найти фамилию `Smith`, но для того чтобы найти все фамилии, содержащие `ith`, придется просканировать все строки.
- Условия с выражениями не совместимы с SQL Server, поэтому должны быть разбиты с использованием алгебры для облегчения проверки правильности вводимых данных.
- Если предложение `WHERE` включает в себя функцию, например строковую, то для проверки всех строк с применением к данным функции потребуется сканирование таблицы.

Избирательность индексов

Еще одним аспектом настройки индексов является их избирательность. Более избирательный индекс учитывает больше значений и отбирает меньшее количество данных одним своим значением. Первичный ключ или уникальный индекс имеет наибольшую избирательность.

Индекс, имеющий всего несколько значений, распределенных по крупной таблице, является менее избирательным. Малоизбирательные индексы могут оказаться совершенно бесполезными в запросах. Столбец, имеющий в тысяче строк всего три значения, является плохим кандидатом на создание индекса. Битовый столбец имеет малую избирательность, и его индексирование бесполезно.

SQL Server использует свою внутреннюю статистику индексов для отслеживания их избирательности. Команда `DBCC Show_Statistics` выдает отчет о том, когда последний раз была обновлена статистика, и выдает основную информацию о статистике индексов, а также об их полезности. Низкая плотность является индикатором того, что индекс очень избира-

тельный. Высокая плотность указывает на то, что конкретный узел индекса указывает на несколько строк таблицы и что он может оказаться мало полезным. Это продемонстрировано в следующем примере:

```
Use CHA2
  DBCC Show_Statistics (Customer, IxCustomerName)
```

Будут получены следующие результаты (отформатированные и усеченные; полный листинг содержит детали о каждом значении индекса):

```
Statistics for INDEX 'IxCustomerName'.
Rows                                          Average
Updated   Rows  Sampled  Steps  Density  key length
--------- ----- -------- ------ -------- -----------
May 1,02  42    42       33     0.0      11.547619

All density    Average Length  Columns
-------------- --------------- ----------------------------
3.0303031E-2   6.6904764       LastName
2.3809524E-2   11.547619       LastName, FirstName

DBCC execution completed. If DBCC printed error messages,
contact your system administrator.
```

Иногда изменение порядка ключевых столбцов способно усилить избирательность индекса и улучшить его производительность. Однако будьте осторожны, так как производительность других запросов может сильно зависеть от текущего порядка ключевых столбцов в индексе.

Повторное использование планов выполнения запросов

Как было продемонстрировано в `statistics time`, время разбора и компиляции запроса может быть достаточно большим. Сохранение планов выполнения запросов может оказаться критичным для поддержания высокой производительности базы. Когда запрос определен, во время его первого выполнения SQL Server старается сохранить его план в процедурном кэше.

В то время как оптимизация плана выполнения запроса может быть достаточно затратной задачей и повторное его использование приносит немалую выгоду, в определенных ситуациях лучше всего перекомпилировать план.

Оптимизатор запросов выбирает наилучший индекс и корректный метод объединения на основании прогноза количества строк, возвращаемого каждой операцией. Этот прогноз зависит от двух факторов: параметров запроса и статистики индексов.

Чтобы проиллюстрировать этот тезис, предположим, что большинство клиентов вашей компании расположены локально — скажем, в Москве. Естественно, у вас есть заказчики и из других регионов страны, и даже несколько зарубежных. Рассматриваемый запрос ищет всех заказчиков из Москвы. SQL Server создает отличный план выполнения запроса, использующий сканирование кластеризованного индекса, так как из таблицы отбирается бо́льшая часть строк. После этого план сохраняется в памяти.

Теперь предположим, что через пару минут компания сливается с компанией в Санкт-Петербурге, которая в десять раз больше по размерам. Данные при этом также объединяются, и следующий запрос ищет заказчиков из Урюпинска. Исходный план запроса был основан на предположении о том, что будет отбираться бо́льшая часть строк таблицы. Теперь этот план оказался непригодным, поскольку в Урюпинске насчитывается всего несколько клиентов.

К счастью, SQL Server обладает интеллектуальной способностью определять, какой план следует перекомпилировать, основываясь на изменении данных и параметров. SQL Server хранит статистику распределения данных по таблицам, так что всегда можно сказать, как строки распределены между Москвой, Петербургом и Урюпинском.

Формирование статистики обычно установлено в автоматический режим, и SQL Server обновляет ее, когда в таблицу вставляется объем данных, способный ее поколебать. Однако временами пользователям из соображений быстродействия требуется отключать автоматический режим, например, когда в таблицу вставляется или удаляется из нее миллион строк.



С помощью команды DBCC вы можете управлять статистикой вручную (см. главу 37).

Чтобы запрос был сохранен в памяти, он должен иметь параметры — не только константы, но и использующие синтаксис *параметр=значение*. К счастью, SQL Server автоматически параметризирует запрос, заменяя литералы и константы запроса параметрами, что позволяет сохранить запрос.

Более строгое условие требует, чтобы все ссылки на таблицы были правильно формализованы (т.е. имели по крайней мере двухкомпонентную форму *схема*.*таблица*), и тогда план запроса может быть сохранен.

Вы можете взглянуть на кэш планов запросов, чтобы убедиться, что запрос был действительно кэширован. Процедурный кэш может быть достаточно большим. Хотя это и не рекомендуется выполнять в производственной среде, чтобы облегчить проверку конкретного запроса, этот кэш можно очистить:

```
DBCC FREEPROCCACHE
```

Для проверки процедурного кэша используется таблица `syscacheobjects`:

```
SELECT cast(C.sql as Char(35)) as StoredProcedure,
    cacheobjtype, usecounts as Count
  FROM Master.dbo.syscacheobjects C
  JOIN Master.dbo.sysdatabases D
    ON C.dbid = C.dbid
  WHERE D.Name = DB_Name()
    AND ObjType = 'Adhoc'
  ORDER BY StoredProcedure
```

Результат (усеченный):

```
cacheobjtype       Count       StoredProcedure
------------------ ----------- ------------------------------
Compiled Plan      1           INSERT [Lumigent_Profiler]([Pre
Executable Plan    1           SELECT LastName + ' ' + FirstNa
Compiled Plan      1           SELECT LastName + ' ' + FirstNa
Compiled Plan      1           UPDATE msdb.dbo.sysjobschedules
```

Производительность зависит от комбинации запроса, индексов и данных. Сохраненный план запроса будет полезным только до тех пор, пока статистика данных, индексы и параметры остаются единообразными. Когда структура таблицы или индексов изменяется, или обновляется статистика данных, или обновляется значительный объем данных, SQL Server помечает запрос как бесполезный и создает новый план во время следующего выполнения этого запроса.

Для эффективного управления памятью планы запросов также имеют свой срок хранения в памяти; при этом наиболее сложные запросы удаляются дольше всего.

Углубленная стратегия индексирования

Стратегия индексирования имеет дело с крупными задачами, а не с изолированными ситуациями, способными нанести ущерб общему процессу. При выполнении аудита фактической производительности оказывается, что наиболее частыми проблемами индексирования являются (в порядке важности) отсутствие индексов и лишние, ненужные индексы. Именно поэтому лучше тратить время на индексирование на основе таблиц, а не отдельных запросов.



Разработка стратегии индексирования без наличия полноценного уровня абстракции данных является сложным занятием. Вам нужно при этом на долгое время запустить Profiler и исследовать каждый запрос. Это является еще одной причиной того, почему ни одна промышленная база данных не должна проектироваться без надлежащего уровня абстракции данных.

В первую очередь выявите процедуры и запросы, которые осуществляют доступ к таблице и формируют схему данных, используя матрицу CRUD (`create`, `retrieve`, `update`, `delete`) (рис. 50.3). В следующем примере мы проанализируем таблицу `OrderDetail` и для упрощения задачи проанализируем только три фиктивные процедуры.

В следующей таблице будут использованы следующие аббревиатуры: `S` — для отбираемых столбцов; `O` — для упорядочения по столбцу; `W` — для ссылок на столбец в предложении `WHERE`, `G` — для группировки по функции.

Таблица 50.3. Анализ использования таблицы операциями CRUD

Столбец	pGetOrder	pCheckQuantity	pShipOrder
OrderDetailID	S		W
OrderID	W	W	
ProductID	S	S	
NonStockProduct	S		
Quantity	S	S	
UnitPrice	S		
ExtendedPrice	S		
ShipRequestDate	S	W	
ShipDate	S		U
ShipComment	S		U

Следующим шагом является проектирование наименьшего количества индексов, которые удовлетворяют требованиям таблицы. Этот процесс требует в первую очередь определения кластеризованного индекса, а затем создания индексов для каждой процедуры и запроса, осуществляющих доступ к таблице (табл. 50.4). Числа на диаграмме указывают на порядковую позицию столбца в индексе. Включаемые столбцы помечены символом `I`.

Так как таблица `OrderDetail` часто опрашивается с использованием столбца `OrderID` и этот столбец может также быть использован для объединения множества строк на одной странице данных, он является лучшим кандидатом на кластеризованный индекс (`CI`). Данный кластеризованный индекс состоит из одного столбца `OrderID`, на что указывает единица в первом столбце.

Данный кластеризованный индекс удовлетворяет процедуре `pGetOrder`.

Таблица 50.4. План стратегии индексирования таблицы

Столбец	Cl	Ix1	Ix2
OrderDetailID			1
OrderID	1	(cl)	(cl)
ProductID		I	
NonStockProduct			
Quantity		I	
UnitPrice			
ExtendedPrice			
ShipRequestDate		1	
ShipDate			
ShipComment			

Процедура `pShipQuantity` проверяет количество товаров в наличии перед доставкой и фильтрует строки по столбцам `ShipRequestDate` и `OrderID`. Создание некластеризованного индекса по `ShipRequestDate` приведет к использованию в этом индексе обоих столбцов: `ShipRequestDate` и `OrderID`. Это связано с тем, что кластеризованный индекс всегда присутствует в листовых узлах некластеризованного индекса. Поскольку данной процедуре нужны всего четыре столбца, добавление в индекс в качестве включаемых столбцов `ProductID` и `Quantity` позволит индексу `Ix1` полностью покрыть потребности запроса и существенно повысить производительность.

Третья процедура может удовлетвориться добавлением некластеризованного индекса `Ix2` на основе столбца `OrderDetailID`.

Хотя в данном примере мы рассматривали всего три процедуры, в производственной таблице их будет больше, и некоторые индексы смогут покрыть потребности одновременно нескольких процедур.

Использование Database Engine Tuning Advisor

SQL Server 2005 располагает прекрасным инструментом, который позволяет анализировать один запрос или множество запросов и генерирует рекомендации по созданию индексов и разделов, повышающих производительность (рис. 50.6).



Database Engine Tuning Advisor является модернизацией мастера настройки индексов (Index Tuning Wizard) версии SQL Server 2000. Новая утилита также рекомендует модификации файловой структуры.



Используйте утилиту Database Tuning Advisor для проверки и точной настройки стратегии индексации. Если эта утилита предложит вам добавить некоторый индекс, постарайтесь понять, почему. Хорошее понимание данных, запросов и индексации также имеет важное значение для разработчика, чтобы позволить этой утилите выполнить свою работу.

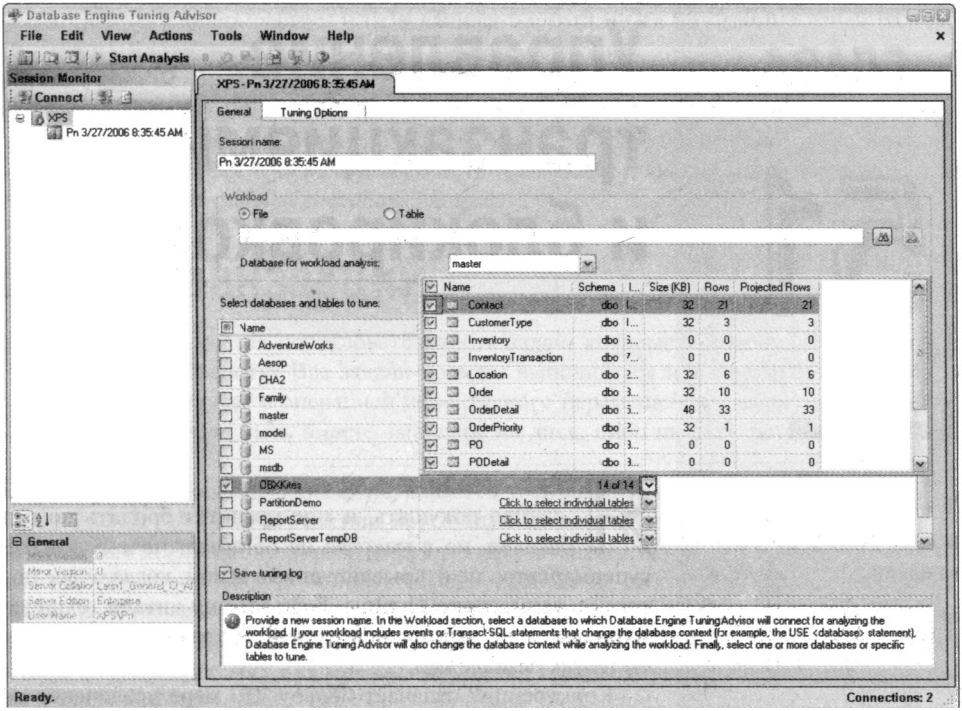


Рис. 50.6. Утилита Database Engine Tuning Advisor способна идентифицировать недостатки индексации и предложить решения проблемы

Резюме

На среднем уровне теории оптимизации лежит индексация. Для осознанного создания индексов нужно понимать не только технологии — оптимизатор запросов, индексные страницы и параметры индексации, — но также схему и уровень абстракции данных. Это одна из проблемных областей, где теоретические знания и практические навыки имеют первостепенное значение. В следующей главе мы продолжим разговор о производительности, затронув вопрос повышения доступности базы данных.

ГЛАВА

51

Управление транзакциями и блокировкой

В этой главе...

Теория целостности транзакций

Журнал транзакций и его важность

Блокировки и производительность

Обработка и предупреждение взаимоблокировок

Реализация оптимистической и пессимистической блокировок

Уровень изоляции мгновенного снимка

Снимки базы данных

Я люблю покушать, и мне нравится бродить по отделу бакалеи, но я никогда не понимал, почему владельцы супермаркетов не открывают новые кассы, когда очереди достигают неимоверной длины. Если бы они читали технические журналы, а не занимались пустой болтовней, все выглядело бы не так безнадежно.

Конкуренция означает борьбу. По мере увеличения количества пользователей, работающих с одними и теми же ресурсами, производительность запросов падает.

Базы данных всегда во главу угла ставили вопрос целостности транзакций, а главным врагом целостности транзакций является конкуренция между пользователями, которые одновременно пытаются извлечь и модифицировать данные. В то время как вопрос изоляции транзакций практически не поднимается в небольших базах данных, в производственной среде с тысячами пользователей конкуренция предполагает постоянную борьбу за целостность транзакций.

И вот почему. По умолчанию, чтобы защитить одну транзакцию от остальных, транзакция, осуществляющая чтение, будет блокировать транзакцию, выполняющую запись. Аналогично, транзакция, выполняющая запись, блокирует транзакции чтения и записи. Чем больше транзакций происходит одновременно (особенно это касается длительных транзакций), тем больше происходит блокировок, и проблемы растут экспоненциально по мере того, как одни ожидающие транзакции блокируют другие на все более продолжительный промежуток времени.

Именно поэтому на четвертом уровне теории оптимизации находится конкуренция.

Хорошая схема, пакетные запросы и приличная стратегия индексирования совместно работают в интересах базы данных, снижая длительность транзакций. Естественно, при этом уменьшается конкуренция (т.е. количество пользователей,

совместно использующих ресурс) и база данных настраивается на использование расширенных функций масштабирования, таких как разделение таблиц.

Мы не будем останавливаться на том, что если у вас существуют проблемы блокировки транзакций, то лучшим решением будет прежде всего разрешить проблемы схемы, запросов и индексации. В этой главе мы поговорим о том, как одни транзакции взаимодействуют с другими, как SQL Server обеспечивает целостность транзакций и как добиться наилучшей производительности, обеспечивая целостность данных.

Основы транзакций

Транзакцией называется последовательность задач, которые в совокупности составляют логическую единицу работы. Все задачи должны выполняться или не выполняться как одно целое. Например, в случае с операцией инвентарного перемещения товара операции вычитания и сложения должны быть вместе записаны на диск, либо на диск не должна попасть ни одна из них.

В SQL Server любая операция DML является транзакцией, независимо от того, имеет она оператор `BEGIN TRANSACTION` или нет. Например, инструкция `INSERT`, вставляющая 25 строк, является логической единицей работы. Все и каждая из этих 25 строк должны быть обновлены. Даже обновление одной строки представляет собой транзакцию, включающую операции, работающие с данными и индексами, которые либо должны быть выполнены в целом, либо в целом не должны быть выполнены.

Чтобы создать оболочку для множества команд одной транзакцией, требуется совсем немного кода: один компонент, служащий *началом* транзакции, другой — в конце. В конце транзакции она может либо *подтверждаться* после отправки на диск, либо *откатываться*, если в коде встретилась ошибка. Следующие три инструкции выглядят простыми, однако за ними стоит довольно сложный механизм:

- `BEGIN TRANSACTION`
- `COMMIT TRANSACTION`
- `ROLLBACK TRANSACTION`

В следующем примере продемонстрирована типичная транзакция. Последовательность задач облачена в инструкции `BEGIN TRANSACTION` и `COMMIT TRANSACTION`. За каждой задачей в последовательности следует код обработки ошибок, содержащий инструкцию `ROLLBACK TRANSACTION`, выполняющую откат транзакции. Если первая задача (вычитание из складских запасов) выполнилась успешно, а вторая задача (добавление товара на склад) — нет, то инструкция `ROLLBACK TRANSACTION` отменяет как вторую, так и первую задачу. SQL Server откатит транзакцию даже в том случае, если в середине транзакции произойдет сбой. Вот как описанная система обрабатывает складское перемещение:

```
BEGIN TRANSACTION;
  INSERT InventoryTransaction (InventoryID, Location, Quantity)
    VALUES (101, 'Камера A', -2000);
  IF @@Error <> 0
    BEGIN
      ROLLBACK TRANSACTION;
      RAISERROR('Ошибка складской транзакции', 16, 1);
      RETURN;
    END;
  INSERT InventoryTransaction (InventoryID, Location, Quantity)
     VALUES (101, 'Камера 12', 2000);
  IF @@Error <> 0
```

```
    BEGIN
      ROLLBACK TRANSACTION;
      RAISERROR('Ошибка складской транзакции', 16, 1);
      RETURN;
    END;
COMMIT TRANSACTION;
```



О командах управления потоком операций (`if`, `begin`, `end` и `return`) и обработки ошибок (`@@Error` и `raiserror`) см. в главе 18.

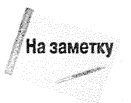
Транзакции могут быть вложенными, и если откатывается вложенная транзакция, то все ожидающие ее завершения (т.е. внешние) также откатываются.

Попытка откатить или подтвердить транзакцию, которая не была открыта явно, приведет к ошибке.

В то время как SQL Server требует явной инициации транзакции с помощью инструкции `BEGIN TRANSACTION`, такой режим работы можно изменить так, чтобы каждый пакет трактовался как транзакция. Следующий код самостоятельно не обновит столбец `Nickname` в базе данных CHA2:

```
USE CHA2;
SET Implicit_Transactions ON;
UPDATE CUSTOMER
  SET Nickname = 'Nicky'
  WHERE CustomerID = 10;
```

Добавление инструкции `COMMIT TRANSACTION` в конец пакета подтвердит транзакцию, и обновление произойдет.



В СУБД Oracle по умолчанию приняты неявные транзакции, и разработчики Oracle при переходе на SQL Server должны изменить свой подход к транзакциям.

В последовательности задач транзакции можно также объявить *точку сохранения* и затем выполнить откат к ней. Однако лично мне кажется, что это смешивает программный поток управления с обработкой транзакций. Если ошибка вызывает необходимость повторного выполнения некоторой задачи в транзакции, то будет более естественно организовать стандартную обработку ошибок, чем нарушать естественную обработку транзакций.

Целостность транзакций

Изучение влияния транзакций на производительность не может обойтись без рассмотрения вопроса о *целостности транзакций*, затрагивающей их качество. Существуют три типа проблем, которые нарушают целостность транзакций: "грязное" чтение, неповторяющееся чтение и призрачные строки. Решение этих трех проблем включает в себя выбор одного из уровней *изоляции* транзакций.

Свойства ACID

Качество базы данных измеряется соблюдением ее транзакций требованиям свойств ACID (т.е. атомарности, целостности, изолированности и живучести). Архитектура большинства современных реляционных баз данных базируется на этих свойствах. Понимание этих свойств является предпосылкой понимания SQL Server.

Атомарность

Транзакция должна быть *атомарной*. Это значит: или все, или ничего. В конце транзакции все ее содержимое либо регистрируется, либо отменяется. Если транзакция была только частично записана на диск, то ее свойство атомарности нарушается.

Целостность

Транзакция должна не нарушать *целостность* базы данных. Это значит, что транзакция начинается в целостном состоянии базы данных и должна вернуть ее в целостное состояние по завершении, независимо от результата операций. С точки зрения ACID целостность означает то, что все строки и значения должны соответствовать моделируемой реальности, а все ограничения — соблюдаться. Если все заказы были записаны на диск, а его строки остались не записанными, то целостность между таблицами `Order` и `OrderDetails` будет нарушена.

Изоляция

Каждая транзакция должна быть *изолирована*, т.е. отделена от эффекта других транзакций. Независимо от действий, осуществляемых другими транзакциями, транзакция должна быть способна продолжать работать именно с тем же набором данных, с которым начала. Изоляция является своеобразным барьером между двумя транзакциями. Доказательством существования изоляции является способность воспроизведения последовательного набора транзакций на исходном множестве данных и постоянного получения одного и того же результата.

Для примера предположим, что некий Саша обновил 100 строк. В то время как транзакция Саши выполнялась, Таня прочитала одну из строк, над которыми в это время работал Саша. Если выполняется чтение, то эти транзакции нельзя назвать полностью изолированными друг от друга. Это свойство менее критично в базах данных, доступных только для чтения, а также в базах данных, доступных только узкому кругу лиц.

Живучесть

Под *живучестью* транзакции понимается ее постоянство, независимо от системных сбоев. Как только транзакция подтверждена, она остается подтвержденной. Ядро базы данных должно быть спроектировано так, чтобы даже в случае выхода из строя устройства данных базу данных можно было восстановить до состояния последней подтвержденной транзакции перед сбоем устройства.

Сбои транзакций

Изоляция между транзакциями может быть далеко не совершенной, и это может проявляться несколькими способами: "грязное" чтение, неповторяющееся чтение и призрачные строки. Все эти нарушения могут потенциально повлиять на целостность транзакций.

"Грязное" чтение

Самым опасным изъяном в работе транзакции является видимость ее работы для других транзакций до подтверждения. Если некоторая транзакция может прочитать неподтвержденные обновления другой транзакции, то этот эффект называют *"грязным" чтением* (рис. 51.1).

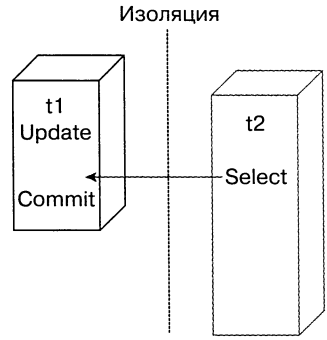


Рис. 51.1. Эффект "грязного" чтения возникает, когда транзакция 2 может прочитать неподтвержденные обновления, выполненные транзакцией 1

Для иллюстрации эффекта "грязного" чтения ниже приведен код. Транзакция 1 выполняет обновление, в это время транзакция 2 видит эти обновления, хотя они еще не подтверждены.

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL
  READ COMMITTED;
USE CHA2;
-- Транзакция 1
USE CHA2
BEGIN TRANSACTION\
  UPDATE Customer
    SET Nickname = 'Transaction Fault'
    WHERE CustomerID = 1;
```

В отдельном окне редактора запросов (рис. 51.2) выполните другую транзакцию. В этой транзакции установлен уровень изоляции, допускающий "грязное" чтение. (Этот уровень изоляции должен быть установлен здесь с целью демонстрации "грязного" чтения. Команды установки уровня изоляции будут описаны в следующем разделе.)

```
-- Транзакция 2
SET TRANSACTION ISOLATION LEVEL
  READ UNCOMMITTED
USE CHA2
SELECT Nickname
  FROM Customer
  WHERE CustomerID = 1;
```

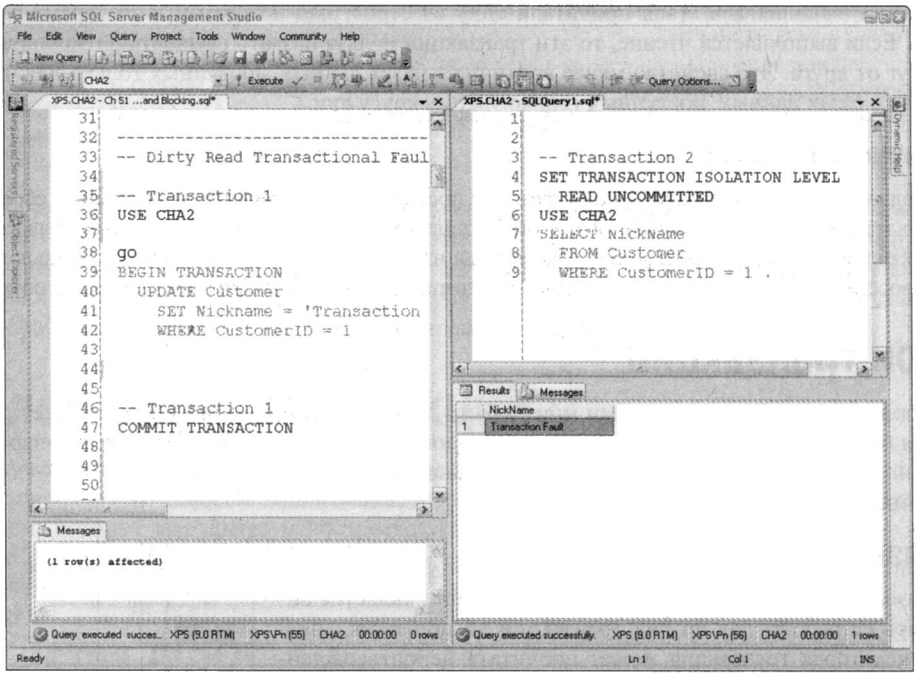


Рис. 51.2. Открытие нескольких окон редактора запросов является лучшим способом экспериментирования с транзакциями. В данном примере транзакция в левом окне обновляет псевдоним на "Transaction Fault", но не подтверждает изменения. Транзакция в правом окне извлекает значение столбца псевдонима и читает в нем "Transaction Fault"

Будет получен следующий результат:

```
NickName
----------------------
Transaction Fault
```

Несмотря на то что транзакция 1 не завершила свою работу с набором данных, транзакция 2 смогла прочитать строку "Transaction Fault". Таким образом, целостность транзакции была нарушена.

Для завершения примера в первом окне следует подтвердить транзакцию:

```
-- Транзакция 1
COMMIT TRANSACTION
```

Неповторяющееся чтение

Неповторяющееся чтение аналогично "грязному" чтению, однако оно возникает, когда транзакция может видеть подтвержденные обновления другой транзакции (рис. 51.3). *Истинная изоляция* предполагает, что одна транзакция никогда не влияет на другую. Если изоляция полная, то транзакция не может увидеть внешние изменения, происходящие с данными во время ее работы. Чтение некоторой строки в транзакции должно всегда приводить к одним и тем же результатам. Если два последовательных чтения одной и той же строки дают разные результаты, то такой тип ошибки транзакции называют неповторяющимся чтением.

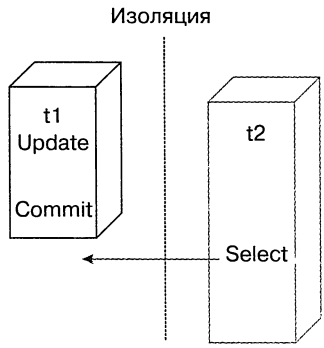


Рис. 51.3. Когда подтвержденные изменения транзакции 1 видны в транзакции 2, этот эффект называют неповторяющимся чтением

В следующем примере приведены две конкурентные транзакции. Транзакция 1 открывается и выполняет чтение данных. Изначальный псевдоним заказчика с кодом 1 является "Transaction Fault".

```
-- Транзакция 1
SET TRANSACTION ISOLATION LEVEL
  READ COMMITTED
BEGIN TRANSACTION
  USE CHA2
  SELECT NickName
    FROM Customer
    WHERE CustomerID = 1;
```

Будет получен следующий результат:

```
Nickname
-----------------
Transaction Fault
```

В то время как первая транзакция остается открытой, вторая изменяет значение поля псевдонима на "Non-Repeatable Read" и подтверждает изменение:

```
-- Транзакция 2
USE CHA2
BEGIN TRANSACTION
  UPDATE Customer
SET Nickname = 'Non-Repeatable Read'
WHERE CustomerID = 1;
COMMIT TRANSACTION;
```

После обновления псевдонима транзакцией 2 транзакция 1 извлекает ту же строку и пытается получить тот же псевдоним. Если она видит обновления, выполненные транзакцией 2, имеет место ошибка неповторяющегося чтения:

```
-- Transaction 2
USE CHA2
SELECT Nickname
  FROM Customer
  WHERE CustomerID = 1;
```

Будет получен следующий результат:.

```
Nickname
--------------------
Non-Repeatable Read
```

Мы видим, что транзакция 2 столкнулась с ошибкой неповторяющегося чтения. На вторую инструкцию SELECT транзакции 1 повлияла инструкция UPDATE второй. Для завершения работы подтвердим транзакцию 1:

```
COMMIT TRANSACTION
```

Призрачные строки

Наименее серьезной ошибкой транзакций являются *призрачные строки*. Подобно неповторяющемуся чтению, призрачные строки возникают, когда обновления, выполненные одной транзакцией, приводят к изменению набора данных, возвращаемого инструкцией SELECT другой транзакции (рис. 51.4).

В следующем примере транзакция 1 изменяет значение поля псевдонима на 'Missy', в то время как транзакция 2 отбирает строки со значениями псевдонимов:

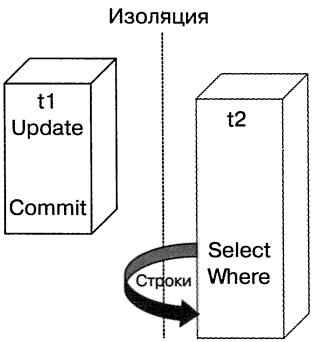


Рис. 51.4. Когда состав строк, возвращаемый инструкцией SELECT, изменяется в результате работы другой транзакции, этот эффект называют призрачными строками

```
-- Transaction 2
BEGIN TRANSACTION
USE CHA2
SELECT CustomerID, LastName
FROM Customer
WHERE NickName = 'Missy';
```

Результат следующий:

```
CustomerID LastName
-- ----------------
2          Anderson
```

```
-- Transaction 1
USE CHA2
BEGIN TRANSACTION
  UPDATE Customer
    SET Nickname = 'Missy'
    WHERE CustomerID = 1;
COMMIT TRANSACTION;
```

Если изоляция между транзакциями полная, то результирующий набор транзакции 2 будет содержать тот же набор строк, что и в предыдущем случае:

```
-- Transaction 2
USE CHA2
SELECT CustomerID, LastName
  FROM Customer
  WHERE Nickname = 'Missy';
```

Результат следующий:

```
CustomerID LastName
---------- --------
1          Adams
2          Anderson
```

Строка с фамилией Adams является призрачной, так как она появилась только во втором результирующем наборе данных в результате работы транзакции 2.

Завершим транзакцию 2 инструкцией `COMMIT TRANSACTION`.

Из всех ошибок транзакций "грязное" чтение является самым опасным, неповторяющееся чтение стоит на втором месте, а призрачные строки можно считать наименее опасными.

Уровни изоляции

Базы данных справляются с тремя описанными выше ошибками транзакций, изолируя транзакции друг от друга. Уровни изоляции можно сравнить с высотой ограды между транзакциями — они позволяют управлять тем, какие ошибки считаются допустимыми. В спецификации ANSI SQL-92 определены четыре уровня изоляции (табл. 51.1).

Таблица 51.1. Уровни изоляции ANSI-92

Уровень изоляции	"Грязное" чтение	Неповторяющееся чтение	Призрачные строки	Блокировка записи
	Возможность видеть неподтвержденные изменения другой транзакции	Способность видеть подтвержденные изменения другой транзакции	Отбор предложением `WHERE` дополнительных строк, внесенных другой транзакцией	Первая операция записи блокируется второй
Read Uncommited (наименее жесткий)	Допустимо	Допустимо	Допустимо	Нет
Read Commited (принят в SQL Server по умолчанию)	Запрещено	Допустимо	Допустимо	Нет
Repeatable Read	Запрещено	Запрещено	Допустимо	Нет
Serializable (наиболее жесткий)	Запрещено	Запрещено	Запрещено	Нет
Snapshot	Запрещено	Запрещено	Допустимо	Да
Read Commited Snapshot	Запрещено	Допустимо	Допустимо	Да

SQL Server реализует уровни изоляции с помощью блокировок. Так как блокировки влияют на производительность, то существует компромисс между уровнями изоляции и производительностью. Принятый в SQL Server уровень изоляции `Read Commited` является своеобразным балансом, подходящим большинству проектов OLTP.

Уровень изоляции можно устанавливать для подключения и пакета. В качестве альтернативы можно устанавливать уровень изоляции для отдельной инструкции DML, используя параметры блокировки таблицы в предложении FROM.

Уровень 1 — Read Uncommited

Этот, наименее строгий, уровень изоляции не предотвращает никаких ошибок транзакций. Он подобен отсутствию забора, так как не обеспечивает никакой изоляции между транзакциями. Установка этого уровня изоляции является аналогом указания серверу баз данных не выполнять блокировок. Этот режим лучше всего подходит для отчетов и приложений чтения данных. Так как этот режим имеет достаточно блокировок, чтобы предотвратить повреждение данных, но недостаточно, чтобы отслеживать конфликты строк, он не очень подходит базам данных, в которых регулярно выполняется обновление данных.

Уровень 2 — Read Commited

Этот уровень изоляции позволяет избежать самой опасной ошибки транзакций, но не нагружает систему излишними блокировками. По этой причине уровень изоляции Read Commited принят в SQL Server по умолчанию и является идеальным выбором для большинства проектов OLTP.

Уровень 3 — Repeatable Read

Предотвращая "грязное" и неповторяющееся чтение, этот уровень обеспечивает повышенную изоляцию транзакций без чрезмерных блокировок, характерных для уровня изоляции Serializable.

Уровень 4 — Serializable

Это наиболее строгий уровень изоляции, позволяющий избежать всех ошибок транзакций. Он больше всего подходит для баз данных, в которых абсолютная целостность транзакций важнее производительности. Обычно этот уровень изоляции используется в банковских, бухгалтерских базах, а также в очень загруженных приложениях продаж.

Использование уровня изоляции Serializable равноценно установке и удержанию блокировки на всем протяжении транзакции. Несмотря на то что этот уровень обеспечивает абсолютную изоляцию транзакций, он способен привести к серьезному снижению производительности.



Выйдя за пределы стандарта ANSI, разработчики SQL Server добавили еще один уровень изоляции — Snapshot. Этот уровень создает копию обновляемых данных в собственном физическом пространстве, которая полностью изолирована от других транзакций.

Уровень изоляции Snapshot

Этот уровень реализует оптимистическую блокировку на уровне базы данных. Как правило, конфликт конкуренции происходит между процессами чтения и записи. Это ни в коей мере не относится к изоляции Snapshot. В данном случае создается мгновенный снимок данных для обновления, и во время этого обновления операции чтения продолжают работать с исходными данными. Когда обновление будет подтверждено, измененная копия будет записана поверх исходных данных.

Изоляция Snapshot позволяет предотвратить конфликт между процессами чтения и записи, но, к сожалению, не в полной мере. Если второй процесс записи попытается обновить ресурс, который в данный момент уже обновляется, второй ресурс будет заблокирован.



Уровень изоляции Snapshot использует версионность строк, записывая их копии во временную базу данных TempDB, что создает невероятную ее загрузку. Если вы используете изоляцию Snapshot, то лучше разместить журнал транзакций и данные базы TempDB в отдельной дисковой подсистеме.

Уровень изоляции Read Commited Snapshot

Этот вариант уровня изоляции Snapshot подобен Read Commited, но устраняет конфликт между процессами чтения и записи.

Архитектура журнала транзакций

Структура SQL Server удовлетворяет требованиям ACID в основном за счет использования последовательного журнала транзакций, гарантирующего живучесть всех транзакций.

Последовательность работы с журналом транзакций

Любая операция модификации данных проходит одну и ту же последовательность действий: вначале осуществляется запись в журнал транзакций, а затем в файл данных. В следующих разделах будут описаны все двенадцать этапов, которые проходит любая транзакция.

Начальное состояние базы данных

Перед началом транзакции база данных находится в целостном состоянии. Все индексы заполнены и указывают на корректные строки, а данные удовлетворяют всем правилам целостности. Каждый внешний ключ указывает на корректный первичный ключ.

Некоторые страницы данных, вероятнее всего, уже кэшированы в память. Дополнительные страницы данных и индексов копируются в память по мере необходимости. Итак, приступаем к описанию последовательности действий в транзакции.

1. База данных находится в целостном состоянии.

Команда модификации данных

Транзакция инициируется отправленным запросом, пакетом или хранимой процедурой, как показано на рис. 51.5.

2. Программный код выполняет команду BEGIN TRANSACTION. Даже если инструкция DML является обособленной, т.е. не вложена в оболочку инструкций, BEGIN TRANSACTION и COMMIT TRANSACTION, она все равно обрабатывается как транзакция.
3. В программном коде выполняется одна инструкция DML (INSERT, UPDATE или DELETE) или их последовательность.

   Чтобы проиллюстрировать работу журнала транзакций, в следующем коде инициируется транзакция, после чего выполняются две команды обновления:

```
USE OBXKites;
BEGIN TRANSACTION;
UPDATE Product
SET ProductDescription = 'Transaction Log Test A',
DiscontinueDate = '12/31/2003'
WHERE Code = '1001';
```

```
UPDATE Product
SET ProductDescription = 'Transaction Log Test B',
DiscontinueDate = '4/1/2003'
WHERE Code = '1002';
```

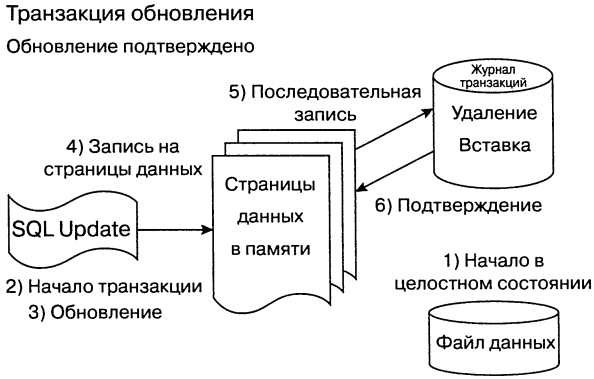


Рис. 51.5. Инструкции SQL DML обрабатываются в памяти как часть транзакции

Обратите внимание на то, что транзакция не была подтверждена.

4. План оптимизации запроса либо генерируется, либо извлекается из памяти. Выполняются все необходимые блокировки, после чего в памяти выполняются все операции модификации данных, в том числе обновления индексов, разбиения страниц и остальные системные операции.

В следующем разделе мы продолжим нашу хронологическую экскурсию по транзакции.

Выполнение записи в журнал транзакций

Наиболее важным аспектом журнала транзакций является то, что все изменения данных записываются в него и подтверждаются перед записью в файл данных (рис. 51.6).

5. Изменения в данных записываются в журнал транзакций.
6. Все записи DML подтверждаются в журнале транзакций. Это является фактической гарантией того, что эти записи в журнале транзакций действительно существуют.

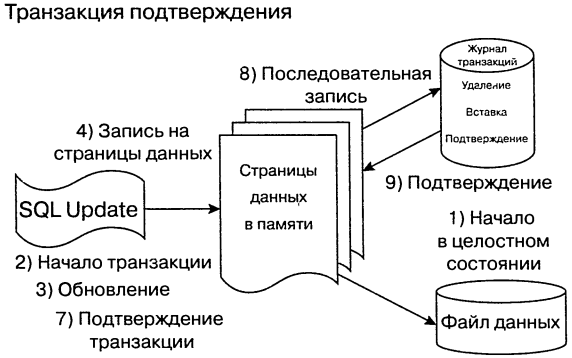


Рис. 51.6. Команда `COMMIT TRANSACTION` запускает следующую вставку в журнал транзакций



Последовательный характер записи в журнал транзакций является причиной того, почему его следует помещать в дисковую подсистему, отличную от той, в которой хранится файл данных. Если эти файлы хранятся раздельно, то выход из строя любой из этих дисковых подсистем не приведет к потере информации базы данных. В любом случае вы сможете восстановить ее в состоянии, предшествовавшем сбою. В то же время, если эти файлы хранить на одном диске, то восстановление базы данных возможно только из последней созданной резервной копии.

Подтверждение транзакции

После того как последовательность задач будет выполнена, инструкция `COMMIT TRANSACTION` закрывает транзакцию, и следующая транзакция получает возможность записи в журнал (см. рис. 51.6).

7. Следующий код закрывает транзакцию:

```
COMMIT TRANSACTION
```



Чтобы увидеть процесс отправки транзакции в журнал, обратитесь к ролику Transaction на сайте книги.

8. Запись `COMMIT` вставляется в журнал транзакций.
9. Запись `COMMIT` подтверждается в журнале транзакций (рис. 51.7).

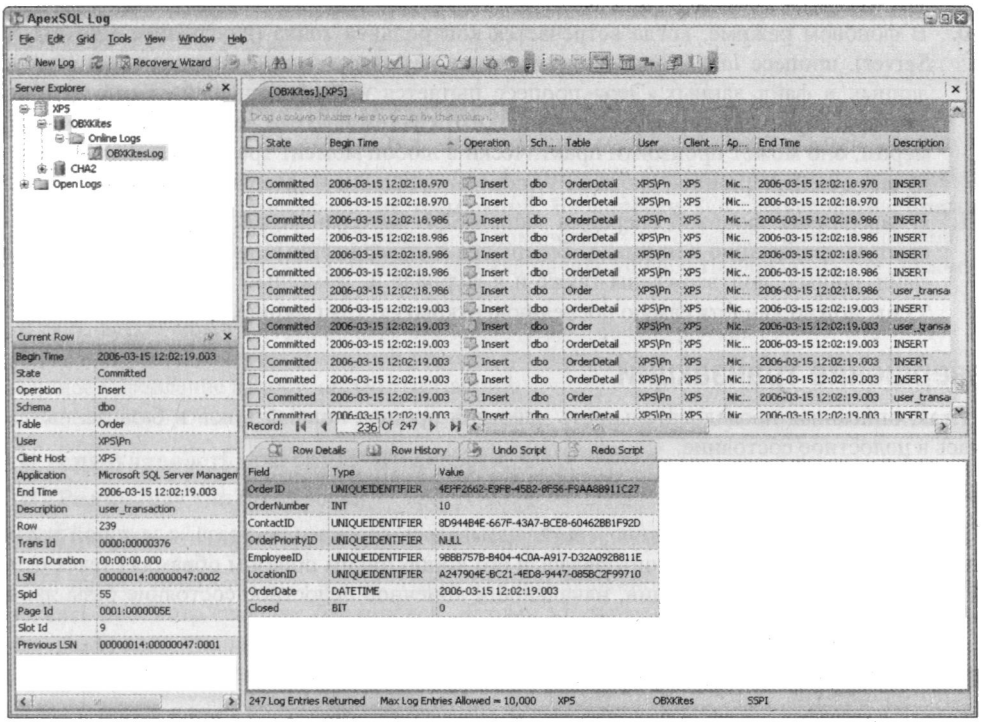


Рис. 51.7. Просмотр подтвержденных транзакций в журнале с помощью сторонней программы ApexSQL Log

Обновление файла данных

После успешного сохранения транзакции в журнале транзакций последняя дисковая операция записывает изменения в файл данных (рис. 51.8).

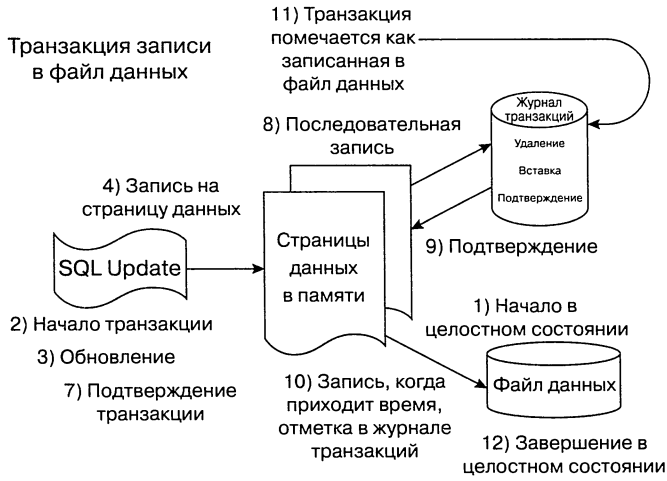


Рис. 51.8. Одним из последних действий является запись изменений в файл данных

10. В фоновом режиме, когда встречается контрольная точка (внутреннее событие SQL Server), процесс *lazy writer* записывает все черновые (модифицированные) страницы данных в файл данных. Этот процесс пытается найти последовательные страницы, чтобы ускорить запись. Несмотря на то что я упоминаю это действие под десятым номером, оно может произойти практически в любой момент транзакции. SQL Server получает от операционной системы Windows сообщение о завершении записи.
11. В завершение фоновой операции записи SQL Server помечает самую последнюю открытую транзакцию в журнале. Теперь все подтвержденные транзакции, записанные в файл данных, подтверждены также и в журнале. С помощью команды `DBCC OpenTran` можно увидеть самую последнюю открытую транзакцию.

Завершение транзакции

Итак, описанная последовательность действий прошла полный цикл, и база данных вернулась в целостное состояние.

12. База данных вернулась в целостное состояние.

В древнееврейской поэзии *включением* называлась строка или фраза, с которой начиналось стихотворение и которая повторялась в его конце, таким образом обрамляя текст основной темой. Так же и в транзакции, начальное и конечное целостное состояние базы данных является своеобразной устойчивой оболочкой.

Откат в журнале транзакций

Если транзакция откатывается, то инструкции DML обращаются к памяти, а в журнале делается запись об отмене транзакции.

Восстановление журнала транзакций

Основным достоинством журнала транзакций является то, что он поддерживает свойство атомарности транзакций в случае системного сбоя.

Если SQL Server перестает функционировать, журнал транзакций автоматически проверяется, как только сервер снова включается в работу. Все происходит так, как описано ниже.

- Если в журнале содержатся какие-либо операции DML, которые не были подтверждены, они откатываются.

  Для тестирования этого свойства нужно набраться храбрости. Начните транзакцию и до ее окончания выключите питание сервера. Обычное закрытие анализатора запросов не заменит этого, поскольку он запросит разрешение на завершение ожидающих транзакций и в случае получения отрицательного ответа от пользователя откатит их. Если остановить SQL Sever обычным образом, то перед фактическим остановом он завершит все ожидающие задачи. Поэтому, чтобы увидеть воочию процесс восстановления журнала транзакций, вам придется отключить компьютер от сети.

  Если следовать описанной выше последовательности действий, то сервер следует отключить непосредственно перед п. 7. В этом случае записи в журнале транзакций будут идентичны показанным позднее на рис. 51.10. SQL Server плавно восстановится после сбоя и откатит все незавершенные транзакции.

- Если в журнале транзакций существуют записи об операциях DML, которые подтверждены, но не отмечены как записанные в файл данных, они будут записаны в файл данных при восстановлении. Эту функцию практически невозможно продемонстрировать.

Концепция блокировок в SQL Server

SQL Server реализует свойство изоляции с помощью блокировок, которые защищают строки транзакций от влияния других транзакций. Блокировки SQL Server нельзя представлять себе только как состояния "страница заблокирована" и "страница разблокирована". На самом деле все гораздо сложнее. И перед тем как начать управлять блокировками, нужно понять схему их действия.

В SQL Server можно неформально определить два процесса: обработчик запросов и диспетчер блокировок. Основной задачей диспетчера блокировок является как можно более эффективное управление целостностью транзакций.

Каждая блокировка обладает тремя свойствами.

- **Гранулярность**, или размер блокировки.
- **Режим**, или тип блокировки.
- **Продолжительность**, или режим изоляции блокировки.

Нет ничего невозможного в том, чтобы увидеть блокировки, и некоторые приемы могут облегчить просмотр текущих блокировок. К тому же соревнование блокировок, т.е. способность различных типов блокировок допускать другие блокировки, может неблагоприятно сказаться на производительности, если этого не понимать и не управлять этим.

Гранулярность блокировок

Фрагмент данных, контролируемый блокировкой, может варьироваться от одной строки до всей базы данных (табл. 51.2). Некоторые комбинации блокировок, в зависимости от их гранулярности, могут удовлетворять требованиям сосуществования.

Таблица 51.2. Гранулярность блокировок

Размер блокировки	Описание
Блокировка строки	Блокировка одной строки. Наименьшая из возможных блокировок, поскольку SQL Server не блокирует столбцы
Блокировка страницы	Блокировка одной страницы размером в 8 Кбайт. На блокируемой странице может находиться одна или несколько строк
Блокировка экстента	Блокировка восьми страниц общим объемом в 64 Кбайт
Блокировка таблицы	Блокировка всей таблицы
Блокировка базы данных	Блокировка всей базы данных. Эта блокировка, как правило, используется при изменениях схемы
Блокировка ключа	Блокировка узлов индекса

Для увеличения производительности диспетчер подключений SQL Server пытается сбалансировать количество блокировок и их размер. Основная борьба идет между конкуренцией (малые блокировки позволяют организовать доступ к данным большему числу транзакций) и производительностью (меньшее число блокировок ускоряет работу сервера). Для достижения этого баланса диспетчер блокировок динамически замещает один набор блокировок другим.

1. Блокировка 25 строк может быть расширена на блокировку целой страницы.
2. Если 25 блокируемых строк распределены по четырем страницам одного и того же экстента, отдельные блокировки страниц и этих 25 строк могут быть заменены блокировкой всего экстента, так как в блокировках задействовано больше 50% страниц экстента.
3. Если в блокировке задействовано множество экстентов, то весь этот набор блокировок может быть заменен блокировкой всей таблицы.

Динамическая блокировка дает разработчикам баз данных SQL Server массу преимуществ.

- Автоматически обеспечивается оптимальный баланс между производительностью и конкуренцией, не требующий дополнительного программирования.
- Производительность базы данных сохраняется даже при увеличении ее размеров за счет автоматического повышения диспетчером гранулярности блокировок.
- Динамические блокировки упрощают администрирование.

Режимы блокировок

Блокировки имеют не только гранулярность (или размер), но также режим, определяющий их назначение. SQL Server имеет множество режимов блокировки (таких как общая, эксклюзивная и блокировка обновления). Недостаточное понимание режимов блокировки приведет к разработке базы данных с плохой производительностью.

Соревнование блокировок

Взаимодействие и совместимость блокировок играют жизненно важную роль в обеспечении целостности транзакций в SQL Server и их производительности. Некоторые режимы блокировки делают невозможными другие; ниже приведена полная таблица совместимости блокировок (табл. 51.3).

Таблица 51.3. Совместимость блокировок

	Блокировка 2 запрашивает					
Блокировка 1	**IS**	**S**	**U**	**IX**	**SIX**	**X**
Намеренная общая (Intent Shared, или IS)	Да	Да	Да	Да	Да	Да
Общая (Shared, или S)	Да	Да	Да	Нет	Нет	Нет
Обновления (Update, или U)	Да	Да	Нет	Нет	Нет	Нет
Намеренная эксклюзивная (Intent Exclusive, или IX)	Да	Нет	Нет	Да	Нет	Нет
Общая с намеренно эксклюзивной (SIX)	Да	Нет	Нет	Нет	Нет	Нет
Эксклюзивная (X)	Нет	Нет	Нет	Нет	Нет	Нет

Общая блокировка (S)

Самой распространенной и часто неправильно используемой блокировкой является общая, которую часто называют блокировкой чтения. Если некоторая транзакция устанавливает общую блокировку, она тем самым как бы сообщает: "Я просматриваю эти данные". Множество разных транзакций могут одновременно просматривать одни и те же данные, в зависимости от режима изоляции.



Будьте особо осторожны с использованием общей блокировки. Я уверен, что неправильно используемые общие блокировки являются самой распространенной причиной проблем обновления и производительности. Приложения должны захватывать данные таким образом, чтобы долго не удерживать общую блокировку. Это является одной из основных причин, почему для извлечения данных лучше использовать хранимые процедуры.

Эксклюзивная блокировка (X)

Под эксклюзивностью блокировки понимается то, что транзакция выполняет запись данных. Как и следует из названия, только одна транзакция может одновременно удерживать эксклюзивную блокировку; при этом ни одна другая транзакция не может просматривать эти данные.

Блокировка обновления (U)

Название "блокировка обновления" может ввести в некоторое заблуждение. Она не применяется, когда транзакция выполняет обновление, — в этом процессе используется эксклюзивная блокировка. На самом деле блокировка обновления означает подготовку транзакции к установке эксклюзивной блокировки и выполнение сканирования данных для выявления строк, которые необходимо блокировать. Блокировку обновления можно трактовать как общую, которая готова превратиться в эксклюзивную.

Во избежание возникновения взаимоблокировок (о них мы поговорим позже в этой главе) в конкретный момент времени только одна транзакция может удерживать блокировку обновления.

Преднамеренные блокировки

Преднамеренную блокировку можно представить себе как желтый сигнал светофора, т.е. предупреждение других транзакций о том, что должно что-то произойти. Главной целью преднамеренной блокировки является повышение производительности. Так как преднамеренность используется во всех типах блокировок и во всех гранулярностях, SQL Server имеет несколько типов преднамеренных блокировок:

- преднамеренная общая блокировка (IS);
- преднамеренная эксклюзивная блокировка (IX);
- общая с преднамеренной эксклюзивной блокировкой (SIX).

Преднамеренные блокировки служат для того, чтобы застолбить место для общей или эксклюзивной блокировки, на самом деле не устанавливая их. Таким образом они позволяют решить две проблемы производительности: иерархическую и постоянную блокировку.

Если бы не существовало преднамеренных блокировок и первая транзакция удерживала общую блокировку, а вторая собиралась установить эксклюзивную блокировку таблицы, то последней пришлось бы проверять наличие блокировок таблицы, страниц, экстентов, строк и ключей.

Вместо этого SQL Server использует преднамеренные блокировки для распространения блокировок на высший уровень иерархии данных. Например, когда некоторая транзакция устанавливает блокировку строки, она одновременно устанавливает преднамеренную блокировку на страницу и таблицу, содержащие эту строку.

Преднамеренные блокировки снимают дополнительную нагрузку с базы данных за счет исключения операций проверки наличия отдельных блокировок на более низком уровне иерархии данных — серверу достаточно проверить наличие преднамеренной блокировки на более высоком уровне. Одновременная установка трех блокировок сохраняет впоследствии сотни поисков блокировок другими транзакциями.

Преднамеренные блокировки также исключают возникновение проблем, связанных с соревнованием общих блокировок, — то, что я называю "постоянной блокировкой". Пока одна транзакция поддерживает общую блокировку, другая транзакция не может установить эксклюзивную. Что случится, если некто будет устанавливать общую блокировку каждые пять секунд и удерживать ее на десять секунд, пока транзакция ожидает эксклюзивной блокировки? В этом случае транзакция `UPDATE` теоретически может ожидать вечность. В то же время, если транзакция установила преднамеренную эксклюзивную блокировку (IX), никакая другая транзакция не сможет установить общую блокировку. Преднамеренная эксклюзивная блокировка не является в полной мере эксклюзивной — это подготовка места для установки эксклюзивной блокировки.

Блокировка схемы (Sch-M, Sch-S)

Блокировки схемы защищают схему базы данных. SQL Server применяет блокировку устойчивости схемы (Sch-S) во время любого запроса, чтобы избежать применения инструкций языка определения данных DDL.

Блокировка модификации схемы (Sch-M) применяется только тогда, когда SQL Server изменяет физическую схему базы данных. Если SQL Server выполнил операцию добавления столбца таблицы только наполовину, блокировка схемы предотвратит просмотр и модификацию данных другими транзакциями до завершения операции изменения схемы.

Продолжительность блокировки

Третье свойство блокировки — ее продолжительность — определяется уровнем изоляции. Чем строже изоляция, тем дольше удерживается блокировка. SQL Server реализует все четыре перечисленные выше уровня изоляции транзакций. Абсолютный уровень изоляции (`serializable`) создает самые строгие блокировки. С другой стороны, самый слабый уровень изоляции (`read uncommited`) эффективно отключает блокировку (табл. 51.4).

Таблица 51.4. Уровни изоляции и продолжительность блокировки

Уровень изоляции	Продолжительность общей блокировки	Продолжительность эксклюзивной блокировки
Read Uncommited	Отсутствует	Удерживается достаточно долго, чтобы избежать физического искажения; в противном случае эксклюзивная блокировка ни применяется, ни обеспечивается
Read Commited	Удерживается во время чтения данных	Удерживается до подтверждения транзакции
Repeatable Read	Удерживается до подтверждения транзакции	Удерживается до подтверждения транзакции
Serializable	Удерживается до подтверждения транзакции	Удерживается до подтверждения транзакции. Эксклюзивная блокировка использует и блокировку ключа (также называемую блокировкой диапазона) во избежание вставок
Snapshot	Нет данных	Нет данных

Мониторинг блокировок

За неимением возможности просмотреть блокировки может показаться, что различные типы блокировок — чистая теория. К счастью, SQL Server является относительно открытой средой, и в ней легко инспектировать текущие блокировки.



Утилита Activity Monitor является прекрасным графическим инструментом просмотра и отслеживания блокировок.

Использование Management Studio

В Management Studio можно просмотреть транзакции выбранной базы данных, используя страницу Summary, на которую помещаются данные из динамических представлений управления. Среди отчетов, связанных с транзакциями, следующие: All Transactions, All Blocking Transactions (рис. 51.9), Top Transactions by Age, Top Transactions by Blocked Transaction Count, Top Transactions by Lock Count, Resource Locking by Object и User Statistics.

Использование Activity Monitor

Утилита Activity Monitor (рис. 51.10) представляет собой инструмент отслеживания блокировок в разрезе процессов и объектов. Ее окно можно обновлять вручную или автоматически каждые пять секунд. Activity Monitor можно открыть из окна Object Explorer, выбрав в контекстном меню сервера пункт Management⇨SQL Server Logs.

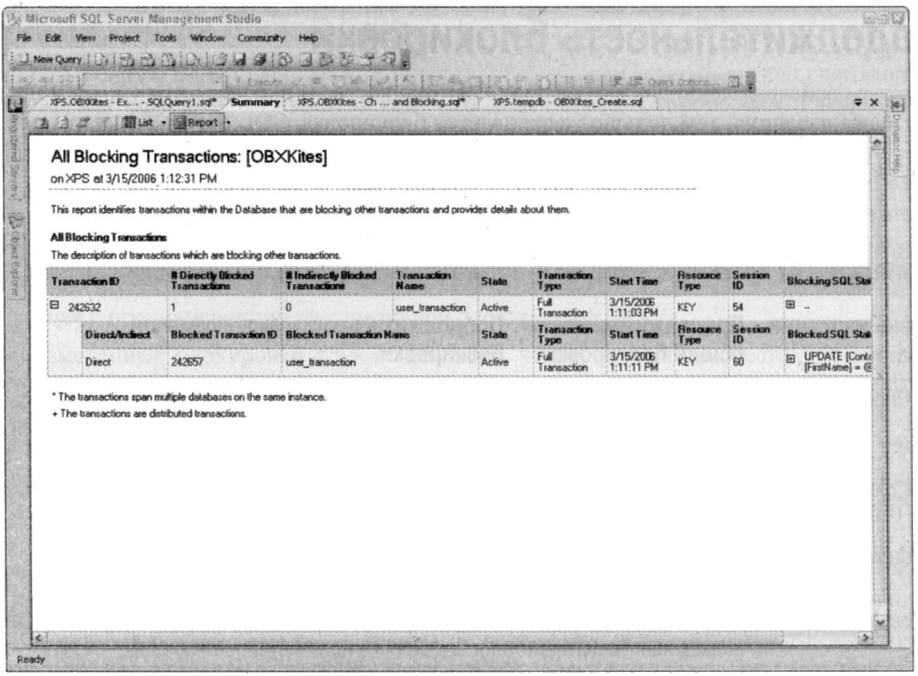


Рис. 51.9. Страница Summary *утилиты Management Studio открывает простой способ просмотра ключевой информации о блокировках транзакций*

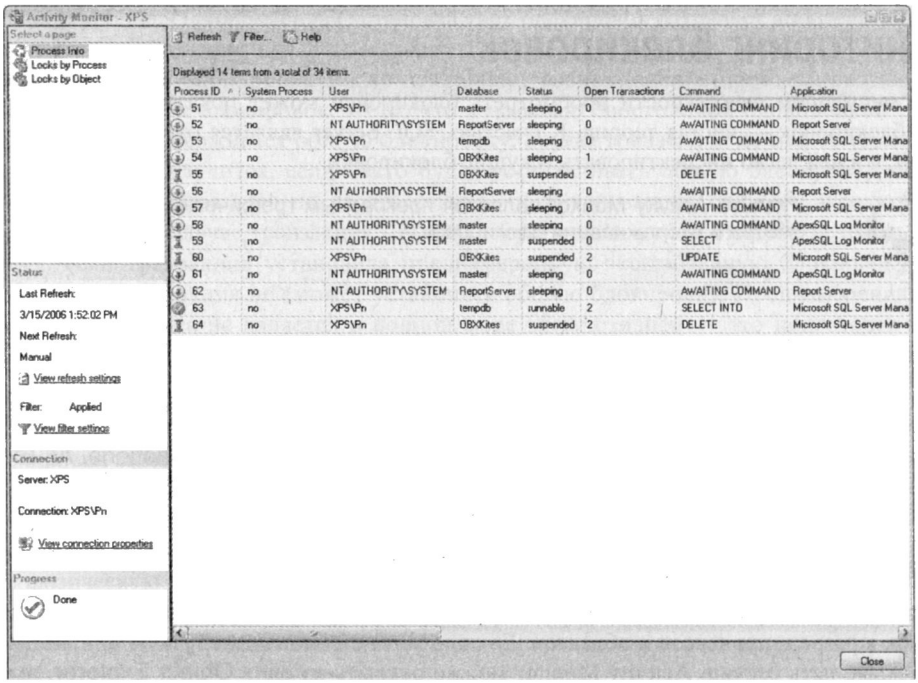


Рис. 51.10. Утилита Activity Monitor отображает массу информации о текущих блокировках

Использование утилиты Profiler

SQL Server Profiler можно использовать для просмотра блокированных процессов, используя событие Error and Warnings:Blocked Process Report (рис. 51.11).

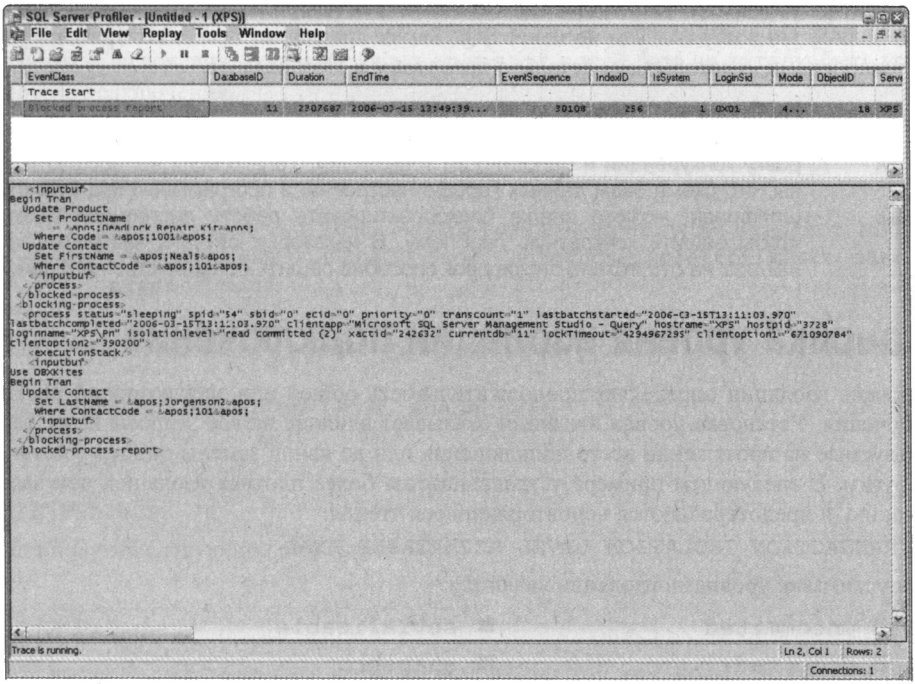


Рис. 51.11. SQL Server Profiler позволяет осуществлять мониторинг блокировок и вызвавшего их кода в формате XML



Из множества доступных методов мониторинга блокировок Activity Monitor и страница Summary утилиты Management Studio являются лучшим способом определения, когда блокировки являются источником проблем. Для локализации источника проблемы Profiler предлагает просмотр фактического кода транзакций.

Тонкость заключается в том, что по умолчанию это событие в Profiler отключено. Для его включения необходимо сконфигурировать параметр `the blocked process threshold`. К тому же это дополнительный параметр, который следует предварительно включить. В следующем фрагменте кода устанавливается продолжительность блокировок в одну секунду:

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'blocked process threshold', 1;
GO
RECONFIGURE;
```

Результатом является полное описание всех блокированных и блокирующихся процессов в формате XML (рис. 51.11). Сохранение этой трассировки в файле и ее последующий анализ является прекрасной методикой отладки блокировок.

Управление блокировками в SQL Server

Если вы уже когда-либо писали схемы блокировок на других языках баз данных для преодоления недостатка блокировок (как я), то у вас могло остаться чувство, что обязательно нужно самому заниматься блокировками. Позвольте вас заверить, что диспетчеру блокировок можно полностью доверять. Тем не менее SQL Server предлагает несколько методов управления блокировками, о которых мы детально поговорим в этом разделе.



Не применяйте параметры блокировки и не изменяйте уровни изоляции случайным образом — доверьте менеджеру блокировок SQL Server выполнять балансировку конкуренции и целостности транзакций. Только если вы абсолютно уверены, что схема базы данных хорошо настроена, а программный код буквально отшлифован, можете слегка подкорректировать работу диспетчера блокировок, чтобы решить конкретную проблему. В некоторых случаях настройка запросов `SELECT` на отсутствие блокировок способна решить большинство проблем.

Установка уровня изоляции подключения

Уровень изоляции определяет продолжительность общей или эксклюзивной блокировки подключения. Установка уровня изоляции оказывает влияние на все запросы и все таблицы, используемые на протяжении всего подключения или до явной замены одного уровня изоляции другим. В следующем примере устанавливается более плотная изоляция, чем задана по умолчанию, и предотвращаются неповторяющиеся чтения:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
```

Допустимыми уровнями изоляции являются:

- `read uncommited`
- `read commited`
- `repeatable read`
- `serializable`
- `snapshot`

Текущий уровень изоляции можно проверить с помощью команды проверки целостности базы данных (DBCC):

```
DBCC USEROPTIONS
```

Результаты будут следующими (сокращенно):

```
Set Option          Value
------------------- ---------------
isolation level     repeatable read
```

Уровни изоляции могут быть также установлены на уровне запроса или таблицы с помощью параметров блокировки.

Использование изоляции уровня снимков базы данных

Существуют два варианта уровня изоляции снимков базы данных: `snapshot` и `read commited snapshot`. Изоляция `snapshot` работает подобно `repeatable read`, не занимаясь вопросами блокировки. Изоляция `read commited snapshot` имитирует установленный по умолчанию в SQL Server уровень `read commited`, так же снимая вопросы блокировки.

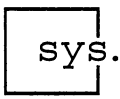
В то время как изоляцию транзакций, как правило, устанавливают на уровне подключений, изоляция `snapshot` должна конфигурироваться на уровне базы данных, поскольку она

эффективно отслеживает версионность строк в базе. *Версионность строк* — это технология, которая создает для обновления копии строк в базе данных TempDB. Кроме основной загрузки базы TempDB, версионность строк также добавляет 14-байтовый идентификатор строки.

Использование изоляции Snapshot

В следующем фрагменте включается уровень изоляции snapshot. Для корректировки базы данных и включения уровня изоляции snapshot к этой базе не должны быть установлены другие подключения.

```
USE Aesop;
ALTER DATABASE Aesop
SET ALLOW_SNAPSHOT_ISOLATION ON
```



Для проверки того, включена ли в базе данных изоляция snapshot, выполните следующий запрос: SELECT name, snapshot_isolation_state_desc FROM sysdatabases.

Теперь первая транзакция начинает чтение и остается открытой (т.е. не подтвержденной):

```
USE Aesop
SET TRANSACTION ISOLATION LEVEL Snapshot;
BEGIN TRAN
SELECT Title
  FROM FABLE
  WHERE FableID = 2
```

Будет получен следующий результат:

```
Title
---------------
The Bald Knight
```

В это время вторая транзакция начинает обновление той же строки, которая открыта первой транзакцией:

```
USE Aesop;
SET TRANSACTION ISOLATION LEVEL Snapshot;
BEGIN TRAN
UPDATE Fable
  SET Title = 'Rocking with Snapshots'
  WHERE FableID = 2;
SELECT * FROM FABLE WHERE FableID = 2
```

Результат следующий:

```
Title
----------------------
Rocking with Snapshots
```

Не правда ли, удивительно? Вторая транзакция смогла обновить строку, даже несмотря на то, что первая транзакция осталась открытой. Вернемся к первой транзакции и увидим исходные данные:

```
SELECT Title
  FROM FABLE
  WHERE FableID = 2
```

Результат следующий:

```
Title
---------------
The Bald Knight
```

Если открыть третью и четвертую транзакции, то они увидят все то же исходное значение `The Bald Knight`:

Даже после того как вторая транзакция подтвердит изменения, первая будет по-прежнему видеть исходное значение, а все следующие транзакции — новое, `Rocking with Snapshots`.

Использование изоляции `Read Commited Snapshot`

Изоляция `Read Commited Snapshot` включается с помощью аналогичного синтаксиса:

```
ALTER DATABASE Aesop
  SET READ_COMMITTED_SNAPSHOT ON
```

Подобно изоляции `Snapshot`, данный уровень для снятия вопросов блокировок также использует версионность строк. Если взять за основу пример, описанный в предыдущем разделе, то в данном случае первая транзакция увидит изменения, выполненные второй, как только они будут подтверждены.

Так как `Read Commited` является уровнем изоляции, принятым в SQL Server по умолчанию, требуется только установка параметров базы данных.

Разрешение конфликтов записи

Транзакции, записывающие данные при установленном уровне изоляции `Snapshot`, могут быть заблокированы предыдущими неподтвержденными транзакциями записи. Такая блокировка не заставит новую транзакцию ожидать — просто будет сгенерирована ошибка. Для обработки подобных ситуаций используйте выражение `try...catch`, выждите пару секунд и попробуйте повторить транзакцию снова.

Использование параметров блокировки

Параметры блокировки позволяют вносить временную коррекцию в статегию блокировки. В то время как уровень изоляции оказывает влияние на подключение в целом, параметры блокировки специфичны для каждой таблицы в конкретном запросе (табл. 51.5). Параметр `WITH (`*`параметр_блокировки`*`)` помещается после имени таблицы в предложении `FROM` запроса. Для каждой таблицы можно задать несколько параметров, разделяя их запятыми.

Таблица 51.5. Параметры блокировки

Параметр блокировки	Описание
`ReadUnCommited`	Уровень изоляции. Не устанавливает и не удерживает блокировку. Равносилен отсутствию блокировок
`ReadCommited`	Уровень изоляции, установленный для транзакций по умолчанию
`RepeatableRead`	Уровень изоляции. Удерживает общую и эксклюзивную блокировки до момента подтверждения транзакции
`Serializable`	Уровень изоляции. Удерживает общую блокировку до завершения транзакции
`ReadPast`	Пропуск заблокированных строк вместо ожидания
`RowLock`	Включение блокировки на уровне строк вместо уровня страницы, экстента или таблицы
`PagLock`	Включение блокировки на уровне страниц вместо уровня таблицы
`TabLock`	Автоматическая эскалация блокировок уровня строк, страниц и экстента до гранулярности уровня таблицы

Окончание табл. 51.5

Параметр блокировки	Описание
NoLock	Неприменение и неудержание блокировок. То же, что и ReadUnCommited
TablockX	Включение эксклюзивной блокировки таблицы. Запрет другим транзакциям работать с таблицей
HoldLock	Удержание общей блокировки до подтверждения транзакции (аналогично Serializable)
Updlock	Использование блокировки обновления вместо общей и ее удержание. Блокировка других записей в данные между изначальными операциями чтения и записи
Xlock	Удержание эксклюзивной блокировки данных до подтверждения транзакции

В следующем примере в предложении FROM инструкции UPDATE использован параметр блокировки, запрещающий диспетчеру эскалировать гранулярность блокировки:

```
USE OBXKites
UPDATE Product
  FROM Product WITH (RowLock)
  SET ProductName = ProductName + ' Updated'
```

Если запрос содержит подзапросы, не забывайте, что обращение к таблице каждого из запросов генерирует блокировку, которой можно управлять с помощью параметров.

Ограничения блокировок уровня индексов

Уровни изоляции и параметры блокировки применяются на уровне подключений и запросов. Единственным способом управления блокировками на уровне таблицы является ограничение гранулярности блокировок на основе конкретных индексов. С помощью системной хранимой процедуры sp_indexoption блокировки строк и/или страниц можно отключить для конкретного индекса, используя следующий синтаксис:

```
sp_indexoption
  'имя_индекса',
  AllowRowlocks или AllowPagelocks,
  1 или 0
```

Это может пригодиться в ряде особых случаев. Если таблица часто вызывает ожидания по причине блокировок страниц, то установка для параметра allowpagelocks значения off установит блокировку на уровне строк. Уменьшенная гранулярность блокировок положительно скажется на конкуренции. К тому же, если таблица редко обновляется, но часто считывается, блокировки на уровне строк и страниц нежелательны; в этом случае оптимальным является уровень блокировки на уровне таблиц. Если обновления выполняются нечасто, то эксклюзивная блокировка таблиц не приведет к большим проблемам.

Хранимая процедура Sp_indexoption предназначена для тонкой настройки схемы данных; именно поэтому в ней используется блокировка на уровне индексов. Для ограничения блокировок по первичному ключу таблицы используйте sp_help *имя_таблицы*, чтобы найти имя индекса первичного ключа.

Следующая команда конфигурирует таблицу ProductCategory как редко обновляемый классификатор. Вначале команда sp_help выводит имя индекса первичного ключа таблицы:

```
sp_help ProductCategory
```

Результат (усеченный) таков:

```
index                          index          index
name                           description    keys
------------------------------ -------------- ----------------
PK__ProductCategory__79A81403 nonclustered,  ProductCategoryID
                               unique,
                               primary key
                               located
                               on PRIMARY
```

Имея в наличии реальное имя первичного ключа, системная хранимая процедура может установить параметры блокировки индекса:

```
EXEC sp_indexoption
  'ProductCategory.PK__ProductCategory__79A81403',
  'AllowRowlocks', FALSE
EXEC sp_indexoption
  'ProductCategory.PK__ProductCategory__79A81403',
  'AllowPagelocks', FALSE
```

Управление временем ожидания блокировок

Если транзакция ожидает блокировку, то это ожидание будет продолжаться до тех пор, пока блокировка не станет возможной. По умолчанию не существует предела времени ожидания — чисто теоретически оно может длиться вечно.

К счастью, вы можете установить время ожидания блокировки с помощью параметра подключения `set lock_timeout`. Установите для этого параметра количество миллисекунд или, если хотите не ограничивать время, установите для него значение `-1` (оно принято по умолчанию). Если для этого параметра установлено значение `0`, то транзакция будет немедленно отклонена при наличии какой-либо блокировки. В этом случае приложение будет исключительно быстродействующим, но малоэффективным.

В следующем запросе время ожидания блокировки устанавливается в две секунды (2000 миллисекунд):

```
SET Lock_Timeout 2000
```

Если транзакция выходит за пределы установленного предельного времени ожидания, то генерируется ошибка с номером 1222.



Настоятельно рекомендую устанавливать предельное время ожидания блокировки на уровне подключения. Эта величина выбирается в зависимости от обычной производительности базы данных. Я предпочитаю устанавливать пятисекундное время ожидания.

Оценка производительности конкуренции в базе данных

Очень легко создать базу данных, в которой не решаются вопросы соревнования блокировок и конкуренции при тестировании на группе пользователей. Реальный тест — это когда несколько сотен пользователей одновременно обновляют заказы.

Тестирование конкуренции нужно правильно организовать. На одном уровне он должен содержать одновременное использование множеством пользователей одной и той же конечной формы. Может оказаться полезной и программа .NET, которая постоянно имитирует

просмотр пользователем данных и их обновление. Хороший тест должен запускать 20 экземпляров сценария, который постоянно загружает базу данных, а затем дать возможность команде тестировщиков использовать приложение. Количество блокировок поможет увидеть монитор производительности, о котором говорилось в главе 49.



Многопользовательскую конкуренцию лучше тестировать в процессе разработки несколько раз. Как говорится в экзаменационном руководстве MCSE, "не допускайте, чтобы тест в реальных условиях был первым".

Блокировки приложения

SQL Server использует очень сложную схему блокировок. Иногда процесс или ресурс, отличный от данных, требует блокировки. Например, может быть необходимо запускать процедуру, которая наносит вред, если другой пользователь запустил еще один экземпляр той же процедуры.

Несколько лет назад я написал программу разводки кабелей в проектах атомных электростанций. Когда геометрия предприятия была спроектирована и протестирована, инженеры вводили состав кабельного оборудования, его местоположение и типы используемых кабелей. После того как вводилось несколько кабелей, программа формировала маршрут их прокладки так, чтобы он оказался как можно более коротким. Процедура также учитывала вопросы безопасности кабельной проводки и разделяла несовместимые кабели. В то же время, если множество процедур прокладки запускалось одновременно, каждый экземпляр пытался маршрутизировать одни и те же кабели, в результате чего результат оказывался некорректным. Блокировка приложений стала отличным решением этой проблемы.

Блокировка приложений открывает целый мир блокировок SQL, предназначенных для использования в приложениях. Вместо использования данных в качестве блокируемого ресурса, блокировки приложений блокируют использование всех пользовательских ресурсов, объявленных в хранимой процедуре `sp_GetAppLock`.

Блокировка приложений может применяться в транзакциях; при этом может быть объявлен режим блокировки `Shared`, `Update`, `Exclusive`, `IntentExclusice` или `IntentShared`. Возвращаемое процедурой значение указывает, успешным ли было применение блокировки.

- `0`. Блокировка установлена успешно.
- `1`. Блокировка была установлена, когда другая процедура сняла свою блокировку.
- `-1`. Запрос на блокировку не был реализован из-за превышения времени ожидания.
- `-2`. Запрос на блокировку не был реализован по причине отмены.
- `-3`. Запрос на блокировку не был реализован из-за взаимоблокировки.
- `-999`. Блокировка не была установлена по другой причине.

Хранимая процедура `sp_ReleaseApLock` снимает блокировку. В следующем примере продемонстрировано, как блокировка приложения может использоваться в пакете или процедуре:

```
DECLARE @ShareOK INT
EXEC @ShareOK = sp_GetAppLock
                @Resource = 'CableWorm',
                @LockMode = 'Exclusive'
IF @ShareOK < 0
...Код обработки ошибки

... Программный код ...
EXEC sp_ReleaseAppLock @Resource = 'CableWorm'
Go
```

Когда блокировки приложения просматриваются с помощью Management Studio или процедуры sp_Lock, они отображаются с типом APP. В следующем листинге приведен сокращенный вывод процедуры sp_Lock, запущенной одновременно с приведенным выше кодом:

```
spid  dbid  ObjId  IndId  Type Resource        Mode  Status
----- ----- ------ ------ ---- --------------- ----- ------
57    8     0      0      APP  Cab11f94c136    X     GRANT
```

Следует обратить внимание на два небольших отличия в том, как блокировки приложения обрабатываются в SQL Server:

- взаимоблокировки не выявляются автоматически;
- если некоторая транзакция устанавливает блокировку несколько раз, она должна снять ее ровно такое же количество раз.

Взаимоблокировки

Взаимоблокировкой называют особую ситуацию, которая возникает только тогда, когда транзакции с множеством задач соревнуются за ресурсы друг друга. Например, первая транзакция установила блокировку ресурса А, и ей необходимо заблокировать ресурс Б, а в это же время вторая транзакция, заблокировавшая ресурс Б, нуждается в блокировке ресурса А.

Каждая из этих транзакций ожидает, пока другая снимет свою блокировку, и ни одна из них не может завершиться, пока этого не произойдет. Если не произойдет внешнего воздействия или одна из транзакций завершится по определенной причине (например, по времени ожидания), то эта ситуация может продолжаться до конца света.

Раньше взаимоблокировки представляли собой серьезную проблему, но теперь SQL Server позволяет успешно разрешить ее.

Создание взаимоблокировки

Проще всего создать ситуацию взаимоблокировки в SQL Server с помощью двух подключений в редакторе запросов утилиты Management Studio (рис. 51.12). Первая и вторая транзакции пытаются обновить одни и те же строки, однако в противоположном порядке. Используя третье окно для запуска процедуры pGetLocks, можно выполнять мониторинг блокировок.

1. Создайте в редакторе запросов второе окно.
2. Поместите код блока Шаг 2 во второе окно.
3. В первое окно поместите код блока Шаг 1 и нажмите клавишу <F5>.
4. Во втором окне аналогично выполните код Шаг 2.
5. Вернитесь в первое окно и выполните код блока Шаг 3.
6. Через короткий промежуток времени SQL Server обнаружит взаимоблокировку и автоматически устранит ее.

Ниже приведен программный код примера.

```
-- Транзакция 1
-- Шаг 1
USE OBXKites
BEGIN TRANSACTION
UPDATE Contact
  SET LastName = 'Jorgenson'
  WHERE ContactCode = '101'
```

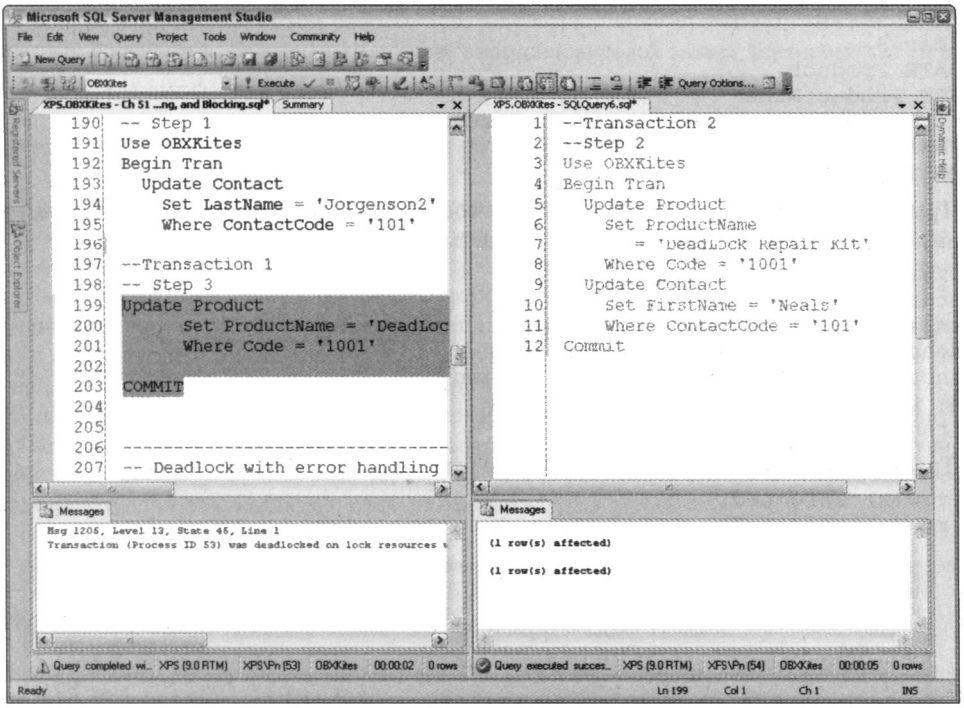


Рис. 51.12. Создание ситуации взаимоблокировки в Management Studio с помощью двух подключений (их окна расположены вверху)

Теперь первая транзакция установила эксклюзивную блокировку на запись со значением `101` в поле `ContactCode`. Вторая транзакция установит эксклюзивную блокировку строки со значением `1001` в поле `ProductCode`, а затем попытается эксклюзивно заблокировать запись, уже заблокированную первой транзакцией (`ContactCode=101`).

```
-- Транзакция 2
-- Шаг 2
USE OBXKites
BEGIN TRANSACTION
  UPDATE Product
    SET ProductName
      = 'DeadLock Repair Kit'
    WHERE ProductCode = '1001'
  UPDATE Contact
    SET FirstName = 'Neals'
    WHERE ContactCode = '101'
COMMIT TRANSACTION
```

Пока еще взаимоблокировки не существует, поскольку транзакция 2 ожидает завершения транзакции 1, но транзакция 1 пока еще не ожидает завершения транзакции 2. В этой ситуации, если транзакция 1 завершит свою работу и выполнит инструкцию `COMMIT TRANSACTION`, ресурс данных будет освобожден, и транзакция 2 благополучно получит возможность необходимой ей блокировки и продолжит свои действия.

Проблема возникает, когда транзакция 1 попытается обновить строку с `ProductCode=1`. Однако необходимую для этого эксклюзивную блокировку она не получит, поскольку эта запись заблокирована транзакцией 2:

```
-- Транзакция 1
-- Шаг 3
UPDATE Product
  SET ProductName
     = 'DeadLock Identification Tester'
  WHERE ProductCode = '1001'
COMMIT TRANSACTION
```

Транзакция 1 вернет следующее текстовое сообщение об ошибке спустя пару секунд. Возникшую взаимоблокировку можно также увидеть в SQL Server Profiler (рис. 51.13):

```
Server: Msg 1205, Level 13,
  State 50, Line 1
Transaction (Process ID 51) was
  deadlocked on lock resources with
  another process and has been chosen
  as the deadlock victim. Rerun the
  transaction.
```

Транзакция 2 завершит свою работу, будто бы проблемы и не существовало:

```
(1 row(s) affected)
(1 row(s) affected)
```

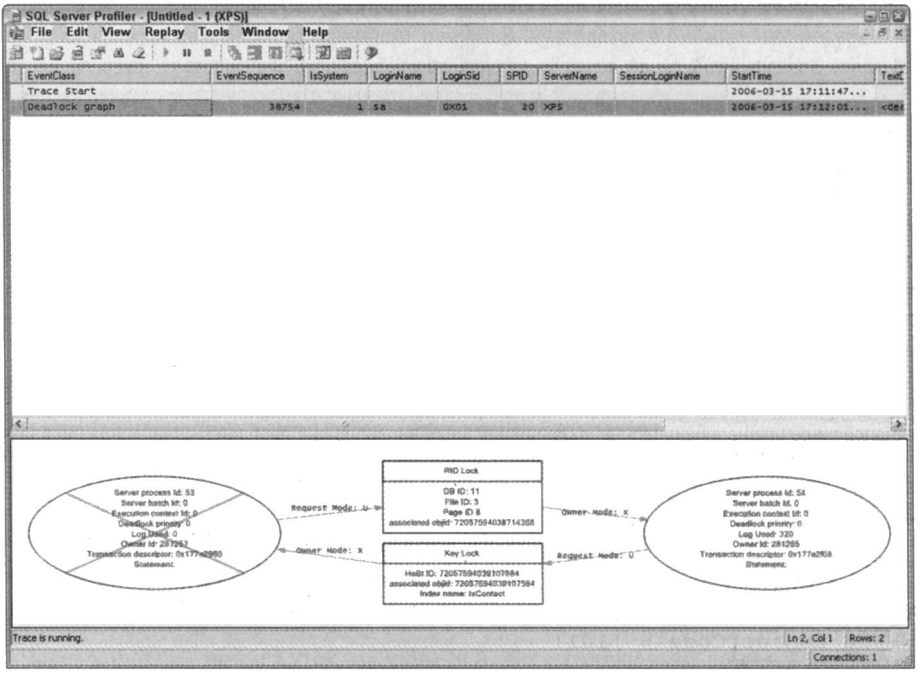


Рис. 51.13. SQL Server Profiler позволяет выполнять мониторинг взаимоблокировок с помощью события `Locks:Deadlock Graph` и выявлять ресурс, вызвавший взаимоблокировку

Автоматическое выявление взаимоблокировок

Как было продемонстрировано в приведенном выше коде, SQL Server автоматически выявляет ситуацию взаимоблокировки, проверяя блокирующие процессы и откатывая транзакции,

выполнившие наименьший объем работы. SQL Server постоянно проверяет существование перекрестных блокировок. Задержка выявления взаимоблокировок может варьироваться от нуля до двух секунд (на практике дольше всего мне приходилось ожидать этого пять секунд).

Обработка взаимоблокировок

Когда происходит взаимоблокировка, подключение, выбранное в качестве "жертвы" взаимоблокировки, должно повторить свою транзакцию. Так как работа должна быть переделана, хорошо, что откатывается именно та транзакция, которая успела выполнить наименьший объем работы, — именно она будет повторена сначала.

Ошибка с кодом `1205` должна перехватываться клиентским приложением, которое и должно перезапускать транзакцию. Если все происходит как нужно, пользователь даже не заподозрит о том, что произошла взаимоблокировка.

Вместо того чтобы позволять самому серверу решать, какую из транзакций выбирать на роль "жертвы", самой транзакции можно "сыграть в поддавки". Следующий код, будучи помещенным в транзакцию, информирует SQL Server о том, что в случае возникновения взаимоблокировки данную транзакцию следует откатить:

```
SET DEADLOCK_PRIORITY LOW
```

Минимизация взаимоблокировок

Даже несмотря на то, что взаимоблокировки легко выявить и обработать, лучше все-таки их избегать. Приведенные ниже рекомендации помогут вам избежать взаимоблокировок.

- Старайтесь делать транзакции короткими и не содержащими излишнего кода. Если некоторый код не обязательно должен присутствовать в транзакции, он должен быть выведен из нее.
- Никогда не ставьте код транзакции в зависимость от ввода пользователя.
- Старайтесь создавать пакеты и процедуры, устанавливающие блокировки, в одном и том же порядке. Например, вначале обрабатывается таблица `A`, затем таблицы `Б`, `B` и т.д. Таким образом, одна процедура будет ожидать второй и взаимоблокировки не смогут возникнуть по определению.
- Планируйте физическую схему так, чтобы хранить одновременно отбираемые данные как можно ближе на страницах данных. Для этого используйте нормализацию и продуманно выбирайте кластеризованные индексы. Уменьшение разброса блокировок поможет избежать их эскалации. Небольшие блокировки помогут избежать их конкуренции.
- Не увеличивайте уровень изоляции, если в этом нет необходимости. Более строгий уровень изоляции увеличивает продолжительность блокировок.

Проектирование блокировок в приложениях

Отдельно от блокировок SQL Server следует рассмотреть еще один связанный с этим вопрос. Способ обработки блокировок в приложениях и решение вопросов конкуренции в многопользовательской среде исключительно важны как с точки зрения работы пользователя, так и с точки зрения целостности данных.

Реализация оптимистической блокировки

В работе с многопользовательским доступом существуют два подхода: *оптимистическая* и *пессимистическая блокировки*. Выбор любой из них определяет конкретный метод программирования приложения.

Под оптимистической блокировкой понимается то, что пока пользователь работает с данными в форме, другие пользователи имеют к ним доступ. Таким образом, в данном случае, пока какой-либо пользователь работает с данными в клиентском приложении, никакие блокировки не применяются. Блокировка устанавливается только на время записи обновлений в базу. Недостатком оптимистической блокировки является возможность появления потерянных обновлений.

Пессимистическая блокировка использует другой подход, подобный известному закону Мерфи: "Если неприятность может произойти, она случается". Когда пользователь работает с какими-либо данными, в пессимистической схеме эти данные блокируются.

В то время как пессимистическая блокировка неплохо проявляет себя в небольших рабочих группах в настольных базах данных, большие приложения с архитектурой "клиент/сервер" требуют более высокого уровня конкуренции. Если блокировка SQL Server поддерживается, пока пользователь работает с данными в форме VB или Access, работа приложения будет необоснованно медленной.

Лучше всего использовать схему оптимистической блокировки, используя минимальную гранулярность блокировок как метод предотвращения потерянных обновлений.

Потерянные обновления

Потерянные обновления имеют место в том случае, когда два пользователя редактируют одну и ту же строку, сохраняют ее, и обновления второго пользователя замещают собой обновления первого. Приведем пример.

1. Некий Ваня открыл в клиентском приложении Visual Basic строку товара с кодом 1001. На долю секунды SQL Server применяет общую блокировку для извлечения данных в форму VB.
2. Таня также открывает строку товара с кодом 1001, используя клиентское приложение.
3. Оба пользователя — Ваня и Таня — вносят свои изменения в данные о товаре 1001; Ваня исправляет описание товара, а Таня — его категорию.
4. Ваня сохраняет строку на SQL Server. Инструкция `UPDATE` заменяет старое описание товара новым, введенным Ваней.
5. Таня также щелкает на кнопке сохранения, и ее данные также отправляются на сервер в инструкции `UPDATE`. Категория товара изменяется, но так как в форме Тани было старое описание товара, оно замещает собой новое, сохраненное Ваней.
6. Ваня обнаруживает ошибку и на очередном совещании жалуется вице-президенту компании на ненадежность базы данных SQL Server.

Так как потерянные обновления возникают только в том случае, когда два пользователя одновременно редактируют одну и ту же строку, эта проблема может не возникать месяцами. Однако это все же пробой в системе обеспечения целостности транзакций базы данных, и его необходимо устранить.

Минимизация потерянных обновлений

Если приложение будет использовать схему оптимистических блокировок, попытайтесь минимизировать шанс возникновения потерянных обновлений, а также эффект, производимый ими. Для этого можно воспользоваться несколькими методами.

- Нормализуйте базу данных так, чтобы она содержала длинные и узкие таблицы. Уменьшая число столбцов в строке, вы снижаете вероятность потерянных обновлений. Например, база данных OBXKites имеет отдельную таблицу для цен. Пользователь может работать с прайс-листом и при этом никак не пресекаться с пользователем, изменяющим другие данные о товарах.
- Если инструкция UPDATE создается в клиентском приложении, постарайтесь проверять изменения и обновлять только те столбцы, в которых они были фактически выполнены пользователем. Использование этого метода позволило бы избежать потерянных обновлений в приведенном выше примере с Ваней и Таней, а также в большинстве других практических ситуаций. В качестве дополнительного преимущества сокращается трафик между клиентом и сервером и загрузка сервера баз данных.
- Если схема оптимистической блокировки не позволяет избежать потерянных обновлений, следует использовать принцип "кто последний записал, тот и прав". Несмотря на то что в этом случае могут возникать потерянные обновления, журнал аудита поможет снизить их эффект. Просматривая обновления, выполненные с одной и той же строкой в течение минуты, вы сможете легко отследить все изменения в данных.

Предотвращение возможности потерянных обновлений

Более строгое решение проблемы потерянных обновлений, чем просто минимизация их эффекта, заключается в использовании метода отслеживания версий. Тип данных `rowversion`, в ранних версиях SQL Server известный как `timestamp`, автоматически присваивает новое значение полю при каждом обновлении строки. Сравнивая значение `rowversion` во время отбора данных для обновления со значением в момент сохранения данных, можно предсказать потерю обновлений с помощью простейшего программного кода.

Метод отслеживания версий может использоваться в инструкциях SELECT и UPDATE. Для этого в предложение WHERE нужно добавить значение `rowversion`.

В следующем примере продемонстрировано использование методики отслеживания версий с использованием двух пользовательских обновлений. Оба пользователя в клиентском приложении открывают строку с одним и тем же товаром. Обе инструкции SELECT извлекают столбцы RowVersion и ProductName:

```
SELECT RowVersion, ProductName
  FROM Product
  WHERE ProductCode = '1001'
```

Будет получен следующий результат:

```
RowVersion          ProductName
------------------- --------------------------
0x0000000000000077 Basic Box Kite 21 inch
```

Оба клиентских приложения заполнили формы извлеченными данными. Один из пользователей отредактировал название товара на "Обновление Вани", и, когда он щелкнул на кнопке сохранения, клиентское приложение выполнило следующую инструкцию SQL:

```
UPDATE Product
SET ProductName = 'Обновление Вани'
```

```
WHERE ProductCode = '1001'
AND RowVersion = 0x0000000000000077
```

Как только сервер обработал обновление Вани, он автоматически изменил значение RowVersion. Проверяя снова строку, Ваня может увидеть свои обновления:

```
SELECT RowVersion, ProductName
FROM Product
WHERE ProductCode = '1001'
```

Результат следующий:

```
RowVersion          ProductName
------------------- -------------------------
0x00000000000000B9 Обновление Вани
```

Если процедура обновления проверяет, не повлияли ли на строку чьи-то обновления, она может увидеть, что изменения Вани были приняты:

```
SELECT @@ROWCOUNT
```

В результате будет получено значение 1.

Несмотря на то что значение столбца RowVersion было изменено, клиентское приложение Тани не знает об этом. Когда Таня попытается сохранить свои изменения, ее инструкция UPDATE не найдет строк, удовлетворяющих критерию:

```
UPDATE Product
  SET ProductName = 'Обновление Тани'
  WHERE ProductCode = '1001'
    AND RowVersion = 0x0000000000000077
```

Если процедура обновления проверяет, были ли изменены какие-либо строки, то она увидит, что редактирование, выполненное Таней, было проигнорировано:

```
SELECT @@ROWCOUNT
```

Результатом будет нуль.

Этот метод может также быть внедрен в приложения, управляемые хранимыми процедурами. Хранимые процедуры fetch и get возвращают значение rowversion наряду с остальными данными строки. Когда приложение VB готово к обновлению и вызывает хранимую процедуру, оно включает rowversion в качестве одного из обязательных параметров. Хранимая процедура update может проверить значение rowversion и выдать ошибку, если два существующих значения не совпадают. Если усложнить этот метод, то хранимая процедура или клиентское приложение может проверить в журнале аудита, не существует ли обновлений столбцов, которые могут привести к потере обновлений, и если это так, сообщить об этом последнему пользователю в сообщении об ошибке.

Стратегии производительности транзакций

Теория целостности транзакций на первый взгляд может показаться пугающей, и SQL Server имеет множество инструментов управления изоляцией транзакций. Если база данных редко используется или предназначена только для чтения, то блокировка транзакций вообще не составляет проблемы. В то же время в интенсивно используемых базах данных OLTP следует применять теорию и знания, полученные в этой главе, а также использовать некоторые стратегии. Так как блокировки представляют собой четвертую стратегию оптимизации, перед ее использованием убедитесь, что были учтены первые три стратегии.

1. Начнем с теории оптимизации. Используйте в качестве основы прозрачную, упрощенную схему, чтобы уменьшить число ненужных объединений, а также объем программного кода, используемого для перемещения данных из одного набора в другой.
2. Используйте эффективный пакетный код, а не болезненно медленные курсоры и циклы.
3. Используйте целостную стратегию индексирования, чтобы исключить излишние сканирования таблиц и ускорить транзакции.

Для идентификации проблем блокировки используйте Activity Monitor или SQL Profiler. Чтобы уменьшить частоту возникновения проблем блокировки, выполните следующее.

- Выполните оценку и тестирование с использованием уровня изоляции `Read Commited Snapshot`. В зависимости от способа обработки ошибок и мощности оборудования этот уровень изоляции может существенно понизить конкуренцию.
- Проверьте уровень изоляции транзакций и убедитесь, что он не выше требуемого уровня.
- Убедитесь, что транзакции начинаются и подтверждаются быстро. Измените структуру тех транзакций, которые содержат курсор. Переместите весь программный код, который не является необходимым в транзакции, за ее пределы.
- Если две процедуры потенциально могут вызвать взаимоблокировку, убедитесь, что они блокируют ресурсы в одном и том же порядке.
- Убедитесь, что клиентское приложение осуществляет доступ к базе с помощью уровня абстракции данных.
- Форсируйте блокировки страниц с помощью параметра `(rowlock)`, чтобы избежать эскалации блокировок.

Резюме

Транзакция представляет собой логическую единицу работы. Несмотря на то что уровень изоляции, установленный в SQL Server по умолчанию, отлично подходит большинству приложений, существуют некоторые механизмы манипулирования и управления блокировками. Чтобы создать серьезное приложение SQL Server, требуется понимание принципов ACID, порядка работы журнала транзакций SQL Server, а также механизма блокировок. Все это отразится на качестве, производительности и надежности базы данных.

В следующей главе мы дополним методы оптимизации стратегией высокой доступности.

ГЛАВА

52

Обеспечение высокой доступности

В этой главе...

Конфигурирование "горячего" резервного сервера с помощью Enterprise Manager или SQL Server Agent

Концепции бессбойных кластеров и доступности

Понятие *доступности* базы данных связано с общей надежностью системы. В принципах информационной архитектуры (см. главу 1) говорилось, что "сервер должен быть всегда доступен и готов к работе". Смысл, вкладываемый в понятие постоянной доступности, зависит от конкретной организации и данных. В базе данных, обладающей высокой доступностью, сбои должны возникать очень редко. Для некоторых баз данных отключение на час не составляет проблему, для других же недоступность в течение 30 секунд является настоящей катастрофой. Правильное решение проблемы доступности зависит от организационных требований, а также от ограничений бюджета и прочих ресурсов.

План поддержки доступности выполняется на трех каскадных уровнях.

1. Поддержка доступности первичной базы данных.
2. Обеспечение практически мгновенного доступа к резервной, или вторичной, базе данных.
3. Восстановление поврежденной базы данных.

Некоторые методы повышения доступности непосредственно не связаны с SQL Server. Это качество и избыточность оборудования, качество электропитания, превентивное обслуживание компьютеров и своевременная замена жестких дисков, а также безопасность серверной комнаты. Все эти факторы вносят свой вклад в повышение доступности первичной базы данных. Первой линией обороны доступности является качество оборудования.

Если база данных была повреждена ввиду аппаратного сбоя, то должен быть выполнен п. 3, т.е. восстановление базы данных из резервной копии. Методы восстановления были подробно описаны в главе 36. Даже при самых благоприятных обстоятельствах восстановление базы данных может занять от нескольких минут до нескольких часов. На практике наиболее

вероятным является сценарий, в котором на перестройку сервера и восстановление данных затрачивается полдня. К тому же в зависимости от типа плана восстановления некоторые данные могут быть утерянными.

Повышенная доступность, заключающаяся в способности обрабатывать ошибки и переключаться на другой сервер, неразрывно связана с качеством оборудования и операцией восстановления. Именно это часто определяет различие между несколькими часами и несколькими секундами недоступности базы данных.



Перед реализацией повышенной доступности убедитесь в том, что уровень первичного сервера хорошо продуман, а план восстановления при сбоях является полноценным. Если не реализована избыточность устройств первичного компьютера, то все средства, затраченные на программные решения, можно считать выброшенными на ветер.

СУБД SQL Server 2005, оснащенная пакетами обновлений, обеспечивает три средства повышенной доступности: доставку протокола, кластеризацию и зеркальное отображение базы данных.



Некоторые решения для синхронизации основного и запасного серверов используют репликацию транзакций. Преимущество этого метода заключается в том, что каждая транзакция самостоятельно перемещается на резервный сервер. Недостатком является то, что любое программное обеспечение, разработанное для поддержки репликации как средства обеспечения избыточности, налагает на базу данных множество ограничений (о репликации см. в главе 39).

Тестирование доступности

Недоступную базу данных нельзя назвать полезной. Тестирование доступности представляет собой имитацию процесса восстановления базы данных в самом пессимистическом сценарии. Мерой доступности является время, необходимое для восстановления самых свежих производственных данных на тестовом сервере, и доказательство работоспособности клиентских приложений.

"Горячая" замена

Понятие *"горячей" замены* относится к базе данных, копия которой поддерживается на отдельном компьютере. Поддержка "горячей" замены реализуется с помощью метода, называемого *доставкой журнала*. Доставка журнала — это периодическое восстановление резервной копии журнала транзакций с первичного сервера на сервере "горячей" замены. Таким образом, поддерживается постоянная готовность к работе резервного сервера. В случае сбоя к работе мгновенно подключаются сервер "горячей" замены и самая последняя резервная копия журнала транзакций. Эта простая схема позволяет реализовать доставку журнала без покупки дорогостоящего оборудования и при малых дополнительных затратах. Однако доставка журнала имеет и ряд недостатков.

- При сбое первичного сервера все транзакции, выполненные с момента последней доставки журнала на резервный сервер, окажутся потерянными. По этой причине доставку журнала обычно организуют каждые несколько минут.
- Переключение происходит не совсем незаметно для пользователей. На сервере "горячей" замены должен быть выполнен некоторый программный код, а клиентское приложение должно переопределить источник данных и подключиться к резервному серверу. Далее в этой главе будет приведен пример такого программного кода.

- Должны быть созданы и периодически выполняться задания службы интеграции, перемещающие регистрационные записи пользователей с первичного сервера на резервный.
- Когда первичный сервер будет восстановлен, возвращение к исходной конфигурации потребует ручного вмешательства администратора базы данных.

Если все эти вопросы не противоречат организационным требованиям, то доставка журнала на сервер "горячей" замены может оказаться отличной альтернативой недоступности базы данных.

В идеальном случае первичный и резервный серверы должны находиться в разных географических местах, чтобы катаклизмы в одном месте не оказывали воздействие на работу другого сервера. К тому же доставка журнала накладывает дополнительную нагрузку на сеть, по которой каждые несколько минут будет перемещаться журнал транзакций. Если два сервера могут быть объединены в одну высокоскоростную частную сеть, то доставка журнала может не сказываться на работе других пользователей и необходимой для них полосе пропускания сети.

Доставка журнала

Корпоративная редакция SQL Server включает в себя мастер доставки журнала (Log Shipping Wizard), интегрированный в мастер обслуживания базы данных, который создает план обслуживания, резервирующий, копирующий и восстанавливающий журнал транзакций с первичного сервера на резервный каждый несколько минут.

Серверы

Доставка журнала обычно охватывает три экземпляра SQL Server: первичный, сервер "горячей" замены и сервер мониторинга.

- *Первичным* является главный сервер, к которому обычно подключаются клиенты. Этот сервер должен быть очень качественным и содержать избыточную дисковую подсистему.
- *Сервером "горячей" замены* является резервный сервер, также называемый *вторичным*. Если главный сервер дает сбой, то он становится первичным. Этот сервер должен удовлетворять минимальным требованиям, достаточным для поддержания работы пользователей во время восстановления главного сервера.
- *Сервер мониторинга* отслеживает работу первичного и резервного серверов. Он знает, когда и какие файлы были перемещены, и генерирует предупреждения, когда синхронизация этих серверов нарушается. Один сервер мониторинга может контролировать множество конфигураций доставки журнала.



Сервер мониторинга может представлять собой экземпляр сервера назначения. В то же время размещение сервера мониторинга на сервере источника является явной реализацией плана самоуничтожения. Если сервер источника физически выйдет из строя, то сервер мониторинга также станет неработоспособным и сервер назначения не получит сигнал на включение в работу. Самым лучшим решением является помещение сервера мониторинга на отдельный компьютер.

Каждая база данных первичного сервера может иметь только один план доставки журнала, а каждый план может доставлять только одну базу данных. В то же время план может доставлять журнал на множество серверов назначения.

Конфигурирование доставки журнала в Management Studio

Доставку журнала можно сконфигурировать с использованием двух методов: с помощью Management Studio и с помощью системных хранимых процедур. При использовании любого из этих методов должно быть создано и сконфигурировано для совместного использования

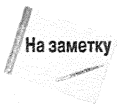
дисковое пространство. Как первичный, так и все вторичные серверы должны иметь возможность считывать и записывать в этот каталог. В Management Studio конфигурирование доставки журнала осуществляется следующим образом.

1. В окне Object Explorer утилиты Management Studio первичного сервера щелкните правой кнопкой мыши на базе данных, журнал которой должен доставляться, и выберите в контекстном меню пункт Properties.
2. На странице Transaction Log Shipping открывшегося диалогового окна установите флажок Enable transaction log shipping.
3. Сконфигурируйте резервирование, щелкнув на кнопке Backup Setting. Введите имя общего каталога на локальном или сетевом сервере. Именно в этот каталог будет помещаться журнал транзакций и храниться до момента копирования на вторичный сервер.



Сетевой совместно используемый ресурс, не размещенный на первичном сервере, лучше защитит базы данных в случае сбоя оборудования главного сервера.

4. Введите такое значение времени, по истечении которого файлы журнала транзакций должны удаляться. Например, если файлы должны удаляться через день, введите в поле Delete Files Older Than интервал в один день.
5. Введите такое значение времени в поле If No Backup Occurs Within, по истечении которого в случае отсутствия нового журнала транзакций сервер должен отправлять предупреждение.



Чем больше промежуток времени до отправки предупреждения, тем большему риску подвергается база данных, так как в случае сбоя первичного сервера потенциально может быть потеряно большее количество транзакций.

6. Создайте расписание заданий, которые будут резервировать журнал транзакций. Для этого введите имя задания, время и частоту. Непродолжительное время между копированием журнала транзакций минимизирует объем данных, утраченных в случае сбоя.



Убедитесь, что на странице Transaction Log Shipping создано только одно расписание резервирования журнала транзакций. В противном случае не все изменения в данных будут распространены на вторичные серверы.

7. Добавьте вторичные серверы в конфигурацию журнала транзакций, щелкнув на кнопке Add в окне второго экземпляра. Повторяя действия пп. 7–14, вы можете создать множество вторичных экземпляров.
8. На экране вторичной базы данных подключитесь к серверу, который будет выступать в роли резервного, и введите имя вторичной базы данных. Если этой базы данных пока не существует, ее следует создать.
9. Инициализируйте вторичную базу данных. Для этого можете установить создание ее из резервной копии, создаваемой сразу после завершения операции конфигурирования доставки журнала, или использовать для этого уже существующую резервную копию. В последнем случае вам следует также указать имя каталога, в котором находится резервная копия базы данных. При желании вы можете не инициализировать вторичную базу данных.
10. Во вкладке Copy Files установите, куда должны копироваться файлы резервной копии и журнала транзакций. В этой же вкладке можно определить, через какое время файлы резервных копий должны удаляться.



Неплохая идея заключается в том, чтобы копировать файлы на резервный сторонний сервер. Это гарантирует то, что в случае сбоя оборудования первичного сервера файлы останутся доступными.

11. Во вкладке Copy Files укажите, когда файлы должны копироваться заданием.
12. Во вкладке Restore Transaction Logs выберите режим ожидания (standby) либо отмену восстановления. Режим ожидания обеспечит доступ к вторичному серверу только для операций чтения. Если вы выберете именно этот вариант, то подключения пользователей будут удаляться во время доступности копии журнала транзакций. Режим отмены восстановления не откроет доступ к вторичной базе данных никакой другой базе данных.
13. Во вкладке Restore Transaction Log также доступны варианты отложенного восстановления и отправки предупреждения. Эти варианты позволяют сохранить все журналы транзакций до окончания рабочего дня либо применить их немедленно по получении.
14. Для задания во вкладке Restore Transaction Logs доступны также параметры повышения гранулярности и времени их применения.
15. Щелкните на кнопке OK, чтобы завершить настройку вторичной базы данных и вернуться на страницу параметров базы данных.
16. Когда настройка вторичной базы данных будет завершена, сервер мониторинга может быть сконфигурирован во вкладке Properties базы данных. Здесь же доступен вариант создания сценария установки.



Если доставка журнала имеет множество параметров, не предусмотренных по умолчанию, то создание сценария будет гарантировать, что все выполненные вами изменения конфигурации будут каждый раз оставаться теми же.

17. После щелчка на кнопке OK на странице параметров базы данных откроется диалоговое окно сохранения конфигурации, после чего настройку доставки журнала можно считать завершенной.

Чтобы удалить доставку журнала, откройте страницу параметров базы данных, перейдите к вкладке Transaction Log Shipping, снимите флажок Log Shipping и щелкните на кнопке OK.

Мониторинг доставки журнала

Для осуществления мониторинга доставки журнала можно использовать несколько таблиц и хранимых процедур. Информация, извлекаемая из этих источников, содержит имя базы данных, характеристики последнего резервирования и восстановления, время, прошедшее с момента последнего восстановления, а также сведения об активности предупреждений.

Ниже приведен список таблиц, которые можно использовать для мониторинга доставки журнала.

- `log_shipping_monitor_alert`
- `log_shipping_monitor_error_detail`
- `log_shipping_monitor_history_detail`
- `log_shipping_monitor_primary`
- `log_shipping_monitor_secondary`

Все эти таблицы находятся в базе данных `msdb` (так как доставка журнала по своей сути представляет набор заданий) всех серверов, вовлеченных в конфигурацию доставки журнала.

Ниже приведен список хранимых процедур, которые можно использовать для мониторинга доставки журнала. Они существуют на всех серверах, вовлеченных в конфигурацию доставки журнала в базе данных `master`.

- `sp_help_log_shipping_monitor_primary`
- `sp_help_log_shipping_monitor_secondary`
- `sp_help_log_shipping_alert_job`
- `sp_help_log_shipping_primary_database`
- `sp_help_log_shipping_primary_secondary`
- `sp_help_log_shipping_secondary_database`
- `sp_help_log_shipping_secondary_primary`

Для мониторинга доставки журнала также может пригодиться отчет `Transaction log shipping`, который находится в узле Reports страницы Summary сервера.

Переключение ролей

Доставка журнала позволяет вручную переключать роли. Это действие можно осуществлять для обслуживания баз данных, а также в случае сбоя сервера. В зависимости от причины сбоев и их частоты лучше вначале выполнить резервирование журнала транзакций на первичном сервере. Если сбой привел к повреждению данных, то вы можете не выполнять это действие, чтобы поврежденные данные не попали на резервный сервер.

Для того чтобы переключиться на резервный сервер, на него следует либо вручную скопировать файлы журнала транзакций, либо с помощью задания SQL Server Agent. После копирования файлы журнала транзакций должны быть восстановлены в правильном порядке на вторичном сервере. Последний журнал транзакций восстанавливается с параметром `WITH RECOVERY`, позволяющим закрыть все открытые транзакции и привести SQL Server в обновляемое состояние. В идеальном случае все эти действия лучше выполнить и задокументировать прежде, чем переключиться на вторичный сервер.

Конфигурирование ожидающего сервера запросов, доступного только для чтения

В SQL Server 2005 изменить конфигурацию доставки журнала гораздо легче, чем в более ранних версиях SQL Server. Изменить вторичную базу данных, доступную только для чтения, чтобы разрешить чтение, достаточно просто. База данных доставки журнала способна уменьшить нагрузку на первичный сервер, перенося операции чтения на вторичный сервер. Для этого выполните следующие действия.

1. В SQL Server Management Studio щелкните правой кнопкой мыши на базе данных, чтобы получить доступ к ее параметрам.
2. Щелкните на странице Transaction Log Shipping, чтобы получить доступ к информации о доставке журнала транзакций.
3. Отредактируйте свойства второго узла доставки журнала, щелкнув на эллипсе с тремя точками.
4. Измените параметры восстановления во вкладке Restore журнала транзакций и щелкните на кнопке OK.
5. Щелкните на кнопке OK на странице Transaction Log, и изменения конфигурации доставки журнала будут применены.

С данного момента все операции доступа к данным, которым нужно чтение, но не нужна запись, получат доступ ко второму узлу, а первичный сервер баз данных будет освобожден от операций чтения.

Доставка учетных записей пользователей

Ни доставка журналов, организованная с помощью Management Studio, ни хранимые процедуры не синхронизируют регистрационные записи первичного сервера на сервере "горячей" замены. Если резервный сервер становится текущим, то он должен иметь те же регистрационные записи, что и основной; в противном случае пользователи не смогут получить к нему доступ. Таким образом, одной из важных задач становится перемещение регистрационных записей пользователей с первичного сервера на сервер "горячей" замены. Лучше всего для выполнения этой задачи использовать задание службы интеграции, которое выполняет подключение к каждому из серверов и перемещает учетные записи пользователей. Если вы постоянно добавляете учетные записи пользователей сервера, то задание их перемещения также нужно выполнять несколько раз в день; в противном случае одного раза в день будет вполне достаточно.

Возвращение к исходному первичному серверу

После того как первичный сервер был восстановлен и приведен в полную готовность, следующие действия позволят реинициализировать его в период, когда пользователи не подключены.

1. Используйте задание службы интеграции для перемещения всех регистрационных записей пользователей с сервера горячей замены на первичный.
2. Переместите базу данных со вторичного сервера на первичный, используя для этого либо полное резервное копирование и восстановление, либо отключение и подключение базы данных.

Резервные серверы и кластеризация

Лучший метод обеспечить быстрое восстановление и высокую доступность сервера — это внедрить резервные серверы путем кластеризации. *Кластер* позволяет множеству серверов (их количество зависит от конкретной операционной системы) выступать в роли единого виртуального экземпляра SQL Server. Пользователи подключаются к виртуальному серверу баз данных и даже не знают, какой из физических серверов обрабатывает их запрос.

Кластер SQL Server не является сетевым кластером балансировки загрузки, который обеспечивает масштабируемость Web-серверов. Кластер SQL Server реализует избыточность, а не масштабируемость. Он не балансирует загрузку сервера баз данных.

Использование этого метода требует наличия кластеров Windows 2003 и общей дисковой подсистемы. Каждый сервер требует наличия подключения (оптического или SCSI) к общей дисковой подсистеме. Так как оба сервера используют общую дисковую подсистему, они совместно используют журнал транзакций и файл данных; однако в каждый конкретный момент времени файловым ресурсом владеет только один из серверов. Кластер также использует высокоскоростную сеть, выделенную для серверов кластера (обычно оптическую). Производители аппаратного обеспечения для кластеризации создают специальные модели и конфигурации, с которыми поставляется OEM-версия Windows 2000 Server. Использование резервных серверов и кластеризации требует в несколько раз больших инвестиций, чем использование серверов "горячей" замены.

Каждый сервер в кластере может быть активным или пассивным. Кластер с активным и пассивным серверами называют *одиночным*. Кластер может содержать несколько активных серверов (на каждом из которых находится по крайней мере одна база данных). В этом случае он называется *множественным*. При отказе активного сервера назначенный пассивный сервер автоматически становится активным и принимает управление общим журналом транзакций и файлом данных. С точки зрения пользователя, переключение выглядит как сбой подключения. Многие приложения способны увидеть это и через несколько секунд выполняют переподключение. При использовании таких приложений пользователи практически не ощущают переключение серверов.


Проверено

Если вы планируете реализовать отказоустойчивый кластер, рекомендую обратиться к материалам SQL Server 2005 Resource Kit, содержащим пять глав, посвященных повышенной доступности. Также неплохо прочитать как можно больше документации, предоставляемой поставщиком оборудования, или посетить компанию, где установлено предлагаемое поставщиком кластерное оборудование. Если ваш поставщик оборудования не предоставил список таких ресурсов, выберите другого поставщика.

Установка резервного сервера баз данных

Во многих отношениях установка отказоустойчивого кластера не отличается от установки обычного экземпляра SQL Server. Одной из причин этого является то, что кластеризация является функцией операционной системы. Для конфигурирования кластера SQL Server операционная система должна быть настроена как кластерная. Как только эта настройка будет завершена, кластерной системе будет присвоено имя виртуального сервера. SQL Server устанавливается в операционной системе виртуального сервера, имея другое имя и IP-адрес. На рис. 52.1 показана комбинация физических и виртуальных серверов.

Конфигурирование

После того как операционная система создаст кластер, можно начинать установку SQL Server. Следует заметить, что кластеризация SQL Server доступна только в редакции Enterprise Edition. Перед началом проверьте, имеет ли учетная запись, под которой вы вошли, права добавления пользователей NT в группы NT. Следующие действия помогут вам установить кластер SQL Server.

1. Запустите установку SQL Server 2005 Enterprise Edition. Программа установки распознает, что операционная система создала кластер. Продолжайте установку как обычно, пока не дойдете до диалогового окна Microsoft SQL Server Setup, запрашивающего состав устанавливаемых служб.
2. Установите флажок Create a SQL Server Failover Claster, а также другие необходимые параметры; например, выберите установку службы интеграции или уведомлений. Щелкните на кнопке Next.
3. Поскольку варианты установки экземпляра по умолчанию и именованного экземпляра не доступны, выберите нужный вариант и щелкните на кнопке Next.
4. В следующем окне введите имя виртуального сервера — это имя будет использоваться для подключения к кластеру. Щелкните на кнопке Next.



Рис. 52.1. Пользователи подключаются к виртуальному экземпляру SQL Server, который затем передает запросы активным физическим экземплярам SQL Server в кластере

5. В следующем окне от вас потребуют ввода IP-адреса виртуального сервера. Введите этот адрес, затем щелкните на кнопках Add и OK.
6. В открывшемся окне вас попросят назначить SQL Server допустимому дисковому кластеру; выберите его и щелкните на кнопке OK.



Кластер уровня операционной системы используется для хранения конфигурации всех узлов кластера и текущего владельца ресурсов (файлов). Считается правильным не помещать ресурсы SQL на это устройство.

7. В открывшемся окне Cluster Node Configuration вас попросят добавить в кластер узлы. Добавьте вторичный узел и щелкните на кнопке Next.
8. Введите имя учетной записи и пароль, под которыми будут запускаться SQL Server и связанные с ним службы.

9. Назначьте группы и учетные записи, которые будут управлять ресурсами. Учетная запись, под которой вы вошли в систему перед установкой сервера баз данных, должна иметь права назначения пользователей группам. Щелкните на кнопке Next. Оставшаяся часть установки сервера ничем не отличается от обычной.



Когда кластеризация будет установлена, позвольте операционной системе управлять остановом и запуском всех служб. Я бы не рекомендовал использовать Management Studio для запуска и остановки каких-либо служб SQL Server.



При установке пакетов обновлений обязательно ознакомьтесь в документации с особенностями их установки в кластерных системах. Многие из них позволяют устанавливать пакет на вторичном узле, прежде чем переключать вторичный узел на роль владельца ресурса. Это даст вам возможность установить пакет обновлений на первичном узле, в то время как вторичный узел будет отвечать на все запросы.

По завершении установки потратьте время на переключение между узлами кластера, чтобы убедиться, что кластер был установлен корректно. Деятельность базы данных теперь может продолжаться в обычном режиме с подключением к виртуальному имени, присвоенному в п. 4. Термин "кластеризация" звучит более серьезно, чем сама кластеризация является на самом деле. С выходом последних улучшений серверных систем Windows 2000 и Windows 2003 установка и управление кластером стали еще более простыми.

Зеркальное отображение баз данных

Это решение занимает нишу между кластеризацией и доставкой журнала. Оно предполагает зеркальное отображение базы данных с первичного сервера на вторичный и незаметное переключение между ними. Третий сервер, называемый *свидетелем*, ведет наблюдение за обоими серверами и активизирует вторичный при сбое первичного.

Зеркальное отображение баз данных делает таким прекрасным механизмом то, что оно не зависит от аппаратного обеспечения и расстояния между узлами. Его можно рассматривать как комбинацию кластеризации без дорогостоящего оборудования и доставки журнала без вмешательства вручную. Для дополнительной избыточности зеркальное отображение базы данных можно использовать в комбинации с доставкой журнала и кластеризацией.



Одной из самых привлекательных функций SQL Server 2005 является зеркальное отображение баз данных. Однако эта функция требует установки пакета обновлений SQL Server 2005 Service Pack 1. Она недоступна в исходной версии продукта без активизации флага трассировки 14000.

Предварительные требования

Подобно доставке журнала, зеркальное отображение баз данных в поддержании синхронизации полностью полагается на журнал транзакций. По этой причине база данных, которую вы собираетесь отображать, должна находиться в *полном режиме восстановления*.

Любая база данных, которая будет участвовать в зеркальном отображении, должна иметь сконфигурированные конечные точки. Это выполняется вручную с помощью T-SQL или специального мастера на странице зеркального отображения. Раздел конфигурирования продемонстрирует работу с этим мастером. В то же время команда, используемая для создания концевой точки вручную, использует синтаксис `CREATE ENDPOINT`.

Перед конфигурированием зеркального отображения на каждом из серверов должна быть создана соответствующая база данных. Лучше всего для этого выполнить резервирование базы данных ведущего сервера и восстановить ее на зеркальном. При восстановлении базы данных следует использовать режим NO RECOVERY. Это гарантирует корректную установку концевых точек и готовность базы данных к зеркальному отображению сразу же после выполнения нескольких простых действий.

Конфигурирование

Чтобы сконфигурировать зеркальное отображение базы данных, выполните следующие действия.

1. В Object Explorer щелкните правой кнопкой мыши на ведущей базе данных и выберите в контекстном меню пункт Properties.
2. Щелкните на странице Mirroring.
3. Запустите мастер системы безопасности (Security Wizard). Он сконфигурирует концевые точки базы данных после щелчка на кнопке Configure Security.
4. На экране приветствия щелкните на Next.
5. Укажите, хотите ли вы использовать сервер-свидетель (в данном примере мы будем его использовать). При использовании сервера-свидетеля концевые точки системы безопасности сохраняются во всех системах, отличных от этого сервера.
6. Потребуйте сохранение системы безопасности также и на сервере-свидетеле. Щелкните на кнопке Next.
7. Введите номер порта и имя концевой точки. Установите флажок Encrypt, чтобы все данные хранились в шифрованном виде. Щелкните на кнопке Next.
8. Выберите из раскрывающегося списка имя зеркального сервера. Если нужного имени в списке нет, щелкните на кнопке Connect и подключитесь к этому серверу. Также вы можете изменить номер порта и имя концевой точки зеркального сервера.
9. Введите ту же информацию для сервера-свидетеля. Если сервер-свидетель является вторым экземпляром ведущего или зеркального сервера, их порты должны отличаться. Щелкните на кнопке Next.
10. Если по определенным соображениям на разных серверах должны использоваться разные учетные записи, введите их и щелкните на кнопке Next.
11. Откроется окно проверки и создания. После того как процесс будет завершен, откроется диалоговое окно, запрашивающее, хотите ли вы запустить зеркальное отображение немедленно. В данном случае щелкните на кнопке Start Mirroring.



Для удаления зеркального отображения вернитесь на страницу Mirroring окна свойств базы данных в Object Explorer и щелкните на кнопке Remove Mirroring.

Архитектура среды

Решения, описанные в этой главе, могут подойти не каждому; в то же время многие из этих решений можно скомбинировать для создания среды, удовлетворяющей практически любым производственным потребностям. В комбинации с системами высокой доступности

SQL Server 2005 допускает лучшее горизонтальное масштабирование, чем какие-либо его предыдущие версии.

На рис. 52.2 продемонстрировано, как зеркальное отображение, доставка журнала и кластеризация могут быть скомбинированы для создания избыточной отказоустойчивой системы и сервера для интерактивной витрины данных (в этой области данные сохраняются перед помещением в хранилище данных). Сервер в Чикаго может выступать свидетелем для кластерных серверов в Лос-Анджелесе и Нью-Йорке. Это позволяет всегда поддерживать промышленные серверы в работоспособном состоянии и обслуживать запросы. Доставка журнала транзакций поддерживает в актуальном состоянии витрину данных для обновления хранилища данных.

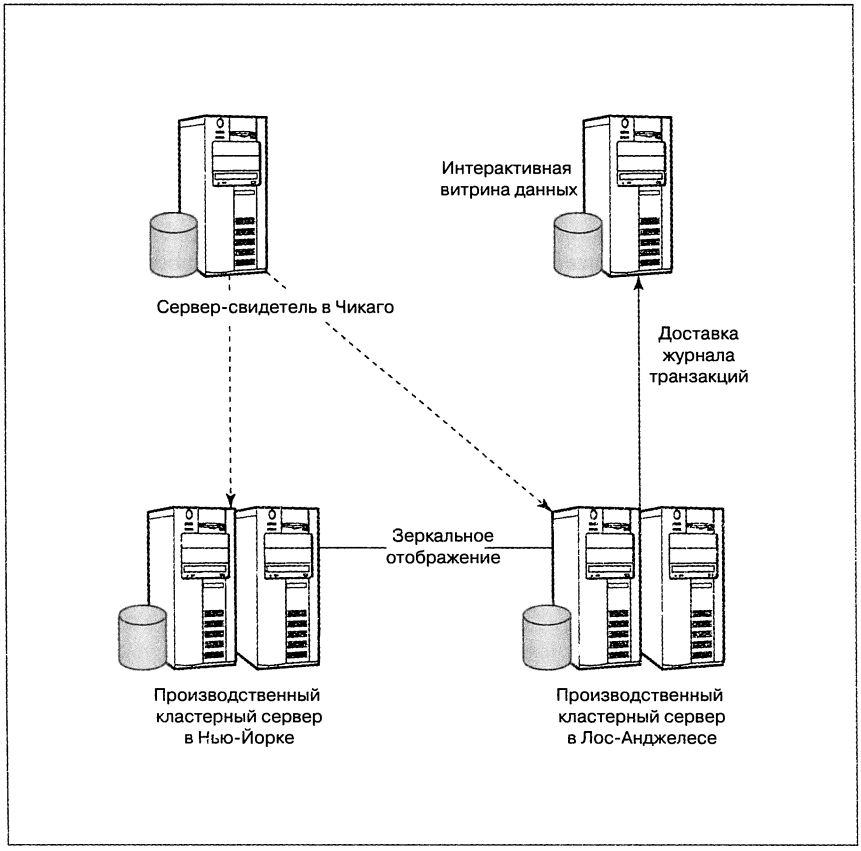


Рис. 52.2. Зеркальное отображение базы данных использует сервер-свидетель для проверки синхронизации зеркально отображаемых баз данных

Резюме

Доступность является ключом к успешности большинства проектов баз данных и становится еще более важной с точки зрения производственных требований. Доставка журнала, отказоустойчивые кластеры, зеркальное отображение баз данных являются высокотехнологичными методами, обеспечивающими устойчивую среду базы данных для пользователей.

Доставка журнала каждые несколько минут позволяет выполнить резервирование журнала транзакций и восстановить его на сервере "горячей" замены с параметром NO RESTORE. В результате сервер "горячей" замены постоянно готов включиться в работу. Однако доставка журнала требует ручного вмешательства. Таким образом, доставка журнала лучше всего себя проявляет для разгрузки процессов с производственного сервера и когда абсолютно свежие данные не требуются.

Отказоустойчивые кластеры конфигурируют множество серверов, совместно использующих одну дисковую подсистему, как единый виртуальный сервер, выполняющий незаметное переключение между физическими серверами в случае выхода из строя одного из них. Единственным недостатком отказоустойчивых кластеров (конфигураций активный/активный или активный/пассивный) является потеря выполняющихся в данный момент запросов. Однако грамотно спроектированное приложение подождет несколько секунд и повторит попытку подключения, прежде чем выдать сообщение об ошибке. Это создает у конечного пользователя впечатление, будто сервер просто несколько замедлил свою работу, а не вышел из строя.

Зеркальное отображение баз данных является еще одним прекрасным инструментом для поддержания географического баланса и планирования действий при катастрофах. Предположим, что сервер в Сан-Франциско был поврежден при землетрясении. При этом его зеркальный образ в Омахе может принять на себя его нагрузку, без каких-либо последствий для производственного процесса.

Спутником доступности является масштабируемость — способность сервера поддерживать большее количество пользователей и больший объем данных. В следующей главе мы рассмотрим вопросы масштабируемости и предложим практические рекомендации по улучшению масштабируемости проектов.

Масштабирование особо крупных баз данных

ГЛАВА

53

В этой главе...

Концепции масштабирования

Разделение таблиц

Индексированные представления

Несколько лет назад я занимался огородничеством. В то время у меня было четыре грядки кукурузы, десятифутовая квадратная горка гороха и десяток других овощей. Виноградная лоза обвивала забор заднего дворика. Процесс был прост: я проводил одну субботу за высадкой, а затем отдыхал по полчаса в день, пропалывая и поливая растения, пока они росли. Вот и все! В результате каждый день на столе у нас были прекрасные свежие овощи.

Когда мне пришлось переехать через полстраны из Северной Каролины в Колорадо, я проезжал десятки миль и не видел ничего, кроме кукурузы.

Контраст между моим огородиком размером в несколько соток и мегафермами в Канзасе прекрасно продемонстрировали понятие масштабируемости. Масштабирование любой задачи нуждается в собственном наборе решений.

Переход от маленького размера к большому можно решить путем масштабирования. *Масштабирование* подразумевает применение для большего объема данных того же решения, что и для меньшего. Проводя аналогию с сельским хозяйством, переход от моего маленького огорода к сельскохозяйственному комплексу подразумевает применение большего количества ручных технологий, используемых мной, т.е. сотен наемных рабочих, которые занимаются посевом, поливом и сбором урожая.

Масштабирование может привнести в решение дополнительную сложность. Например, использование комбайна является средством масштабирования в сельском хозяйстве. Переход от культиватора к трактору является масштабированием.

С другой точки зрения, для каждого масштаба требуется выбирать адекватное решение. Например, комбайн в домашнем огороде будет просто кучей металлолома.

Теперь противопоставим доступность и масштабируемость. Доступность можно себе представить как соглашение с соседом

о том, что, если комбайн одного из вас поломается, вы поможете друг другу. Представьте себе это как план обеспечения отказоустойчивости комбайна.

В Web-серверах масштабирование подразумевает добавление большего числа серверов в Web-ферму. SQL Server работает несколько по-другому — обычно добавляется дополнительный сервер баз данных (это не распространяется на топологию с репликацией). Таким образом, в рамках настоящей главы я определяю масштабирование SQL Server как добавление дополнительного оборудования для обслуживания повышенной загрузки без изменения базы данных.

Теория оптимизации и масштабируемость

Проектирование базы данных, которая будет масштабироваться, требует большего, чем просто подключения новых технологий или добавления сети хранения (Storage Area Network, или SAN). Теория оптимизации описывает зависимость между технологиями оптимизации, и эти расширенные технологии зависят от низлежащих слоев.

Наиболее масштабируемой будет база данных с упрощенной схемой, грамотно спроектированным пакетным кодом, прекрасной индексацией и минимальными конфликтами конкуренции. Если все эти четыре уровня оптимизированы, вы можете сфокусировать внимание на расширенных технологиях масштабирования. Именно поэтому дополнительная настройка масштабирования находится на верхнем уровне теории оптимизации.

Если ваш начальник думает, будто он сможет повысить производительность базы данных, купив SAN и включив функцию масштабирования редакции Enterprise Edition, не обращая внимания на фундаментальные уровни теории оптимизации, он просто выбрасывает деньги на ветер. Мне встречались базы данных, "задыхавшиеся" на восьми серверах с SAN, а ведь они могли бы, в принципе, отлично работать на четырех серверах с локальной дисковой подсистемой, если бы начальство вложило средства в уровни теории оптимизации. Приложение будет работать быстрее сегодня и будет более масштабируемым в будущем, если вложить средства в очистку схемы и переработку итеративного программного кода, а не в запуск медленной базы на более быстром компьютере и написание новой оболочки вокруг старой схемы базы данных.

Масштабирование платформы

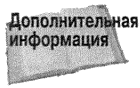
Предполагая, что база данных была оптимизирована с учетом уровней 1–4 теории оптимизации, разрешите предложить мои рекомендации относительно проектирования масштабируемой аппаратной платформы.

- Выделите для SQL Server отдельный компьютер. Не используйте этот сервер как файловый, сервер печати, почтовый или IIS.
- Сбалансируйте производительность процессоров, памяти, дисковых подсистем; добавьте больше памяти, чем вам кажется было бы достаточно, так как SQL Server может использовать память, чтобы уменьшить нагрузку на процессор и дисковую подсистему. Я рекомендую по максимуму использовать возможности использования памяти сервером и редакцию Windows Server, которая поддерживает такой объем памяти. Всегда приобретайте самую быструю память из доступных.
- Узким местом масштабируемости обычно является пропускная способность дисковой подсистемы. Если в качестве дисковой подсистемы вы можете использовать SAN, поступайте именно так. Хорошо сконфигурированная SAN позволяет масштабировать больше, чем локальная дисковая подсистема, и дает четыре существенных преимущества. Во-первых, она распределяет файлы по нескольким дисковым приводам; во-вторых, использует оптоволоконное подключение. В-третьих, SAN обычно содержит

довольно большой буфер памяти, чтобы компенсировать внезапные скачки трафика. И кроме того, данная подсистема обычно выполняет резервирование и восстановление снимков на аппаратном уровне.

Недостатком SAN является то, что ее дисковое пространство стоит в 40–50 раз больше пространства локального диска, и ее довольно сложно конфигурировать и настраивать. Поэтому помогите администратору SAN сконцентрироваться на требованиях базы данных и тщательно сконфигурировать базы данных, чтобы они не потерялась в массе файловых потоков организации. Эта задача может оказаться довольно тяжелой, особенно если потоки базы данных и файлового сервера скомбинированы в одной и той же SAN.

- Используйте доставку журнала или репликацию для создания доступной только для чтения базы данных отчетности.



О репликации см. в главе 39, а о доставке журнала и зеркальном отображении базы данных — в главе 52.

- Подумайте об использовании отдельного резидентного диска для базы `tempdb` или файла подкачки Windows.

Масштабируемость и бизнес-процессы

Вопросы масштабирования связаны и с другими областями работы организации. Первыми приходят на ум обработка платежей и заказов. Как профессионалам в области обработки данных нам нужно обращать внимание на то, как данные перемещаются внутри организации.

В небольшой организации голосового общения может оказаться достаточно. По мере роста организации электронная почта или SharePoint становится стандартным инструментом общения Однако и эти два средства однажды станут малопригодными, когда организация буквально утонет в сотнях электронных сообщений.

Программисты обычно думают о пользователе как некоем обобщенном работнике. Он вводит данные, база данных их сохраняет и возвращает при получении запроса. Приложения с единообразным, учитывающим стандартизированные требования сотрудников, графическим интерфейсом демонстрируют этот подход.

По мере роста организации задачей становится уже не обработка потоков между сотрудниками и базой данных — реальной задачей становится обработка потоков данных между отдельными сотрудниками. Разработка приложений, которые учитывают эти потоки и помогают компании в организации их обработки, является реальным решением масштабируемости.

Если вы не используете SAN, прислушайтесь к моим рекомендациям относительно конфигурирования локального непосредственно подключенного хранилища (DAS). Каждая дисковая подсистема DAS имеет собственный дисковый контроллер.

- Использование большого дискового массива RAID 5 и размещение на нем всех файлов можно легко сконфигурировать, но расплачиваться придется производительностью. Цель дисковой подсистемы не ограничивается избыточностью. Нужно распределить разные файлы по выделенным дискам, предназначенным для разных целей.
- При выборе приводов в первую очередь обращайте внимание на пропускную способность. В настоящее время устройства Ultra 320 SCSI предлагают самую большую пропускную способность, на пятки им наступают устройства SATA 2, и уже существуют некоторые интересные контроллеры SATA RAID. Скорость вращения привода представляет собой еще один ключевой фактор пропускной способности. Существует прямая зависимость между скоростью вращения привода и объемом данных, которые мо-

гут быть считаны или записаны за одну миллисекунду. В настоящее время на вершине находятся устройства со скоростью вращения 15000 об/мин.

- SQL Server оптимизирован для последовательного чтения и записи в дисковой подсистеме как файлов данных, так и журнала транзакций. Поэтому я использую массив RAID 1 (зеркальный), который также оптимизирован для последовательных операций, а не RAID 5, который лучше себя проявляет в случайном доступе.
- Основной целью дисковой подсистемы является не использование как можно более емкого диска, а использование как можно большего количества приводов. Использование двух устройств емкостью 73 Гбайт гораздо лучше, чем одного, емкостью 146 Гбайт.
- Журнал транзакций любой базы данных, подверженной массированным обновлениям, должен находиться в выделенной DAS, чтобы головки постоянно оставались в конце журнала транзакций, не перемещаясь к другим файлам. К тому же выделите дисковому контроллеру достаточно памяти, чтобы буферизировать всплески потоков транзакций.
- SQL Server для обслуживания дополнительных файлов данных создает дополнительные потоки. Исходя из этого, гораздо лучше использовать три файла данных в трех подсистемах DAS, чем один большой файл. Использование множества файлов для распределения нагрузки по нескольким устройствам гораздо лучше, чем использование созданных вручную для разделения таблиц файловых групп.
- Оптимизатор запросов SQL Server 2005 оказывает сильную нагрузку на базу `tempdb`. Лучшая дисковая оптимизация заключается в выделении для `tempdb` отдельного DAS, и еще одного — для журнала транзакций этой базы данных. Еще одним хорошим решением является помещение `tempdb` в разные файлы, размещенные в разных дисковых подсистемах DAS.
- Системе Windows нужен быстрый файл подкачки. Независимо от объема физической памяти в сервере, сконфигурируйте большой файл подкачки и поместите его в отдельную дисковую подсистему.
- Чтобы обобщить рекомендации относительно масштабирования дисковой подсистемы, отличной от SAN, в табл. 53.1 описана одна из возможных конфигураций.

Таблица 53.1. Масштабирование дисковой подсистемы, отличной от SAN

Логический диск	Назначение
C:	Система Windows и исполняемые файлы SQL Server
D:	Файл подкачки Windows
E:	Файл данных `TempDB`
F:	Журнал транзакций `TempDB`
G:	Файл данных
H:	Журнал транзакций
I:, ...	Дополнительные файлы данных

Дополнительная информация

Практика показывает, что производительность SAN примерно в четыре раза выше, чем у дисковой подсистемы DAS. В то же время, воспользовавшись исследованиями доктора Джима Грея (Jim Gray), служба Microsoft Consulting Services создала хранилище данных емкостью 24 Тбайт, используя 48 приводов SATA, и достигла общей пропускной способности, равной SAN (2,2 Гбайт при

последовательном чтении и 2,0 Гбайт при последовательной записи), примерно за сороковую часть стоимости SAN. Более подробно об этом вы можете прочитать по адресу:

```
www.sqlmag.com/Article/ArticleID/49557/sql_server_49557.html
```

Естественно, еще одним вариантом масштабирования является переход с редакции Standard Edition на редакцию Enterprise Edition, чтобы использовать больше процессоров, или перейти на 64-разрядную редакцию SQL Server 2005.

Масштабирование решений

В вершине пирамиды теории оптимизации находится настройка расширенного масштабирования. SQL Server 2005. Она предлагает несколько технологий и средств масштабирования уровня базы данных для повышения производительности.

- Брокер служб может буферизировать запросы, повышая масштабируемость во время пиковых нагрузок. (О брокере служб см. в главе 28.)
- Уровень изоляции `Read Commited Snapshot` позволяет выполнять высокопроизводительные операции чтения без конфликтов конкуренции с операциями записи. (Об изоляции `Snapshot` см. в главе 51.)

Редакция Enterprise Edition добавляет следующие потенциальные ресурсы масштабирования.

- Разделение таблиц и индексов позволяет интеллектуально распределять загрузку данных между множеством файлов, упрощая поддержку и одновременно повышая производительность отбора и обновления.
- Индексированные представления позволяют создавать кластеризованные индексы в представлениях, которые эффективно денормализуют материализованные представления.
- Операции параллельной и интерактивной индексации могут повысить производительность базы данных во время обслуживания.

В этой главе мы сосредоточим внимание на функциях масштабирования редакции Enterprise Edition.

Разделение таблиц и индексов

Разделить таблицу — это значит разбить ее на два или более небольших сегмента, основываясь на диапазонах данных. Разделение наиболее эффективно, когда его ключом является столбец, чаще всего используемый для отбора диапазона данных. По этой причине велика вероятность того, что запросы будут обращаться только к одному из сегментов данных. Приведем примеры.

- Организация имеет представителей в пяти различных регионах; разделение таблицы по регионам позволяет запросам каждого из представителей обращаться только к собственному разделу.
- Крупная производственная компания имеет большую таблицу регистрации действий. Ее лучше разбить на несколько разделов, по одному для каждого из подразделений. При этом каждое специализированное приложение будет иметь тенденцию обращаться только к собственным данным, находящимся в одном разделе.
- Крупная финансовая компания имеет несколько терабайтов исторических данных; при этом следует обеспечить максимальную эффективность запросов как к текущим, так и

к историческим данным. Сегментирование данных по предыдущим и текущему периодам позволит наиболее часто выполняемым запросам к текущим данным обращаться к меньшей по размеру таблице.



СУБД SQL Server 2000 имела разделенные представления разделов и федеративные базы данных, однако их было довольно тяжело конфигурировать и использовать. Несмотря на то что саму теорию нельзя назвать новой, ее практическая реализация в версии SQL Server 2005 совершенно новая. Разделением таблиц в SQL Server 2005 значительно проще управлять. Представления распределенных разделов в SQL Server 2005 все еще поддерживаются из соображений обратной совместимости, и все же их стоит преобразовать в разделенные таблицы.

Разделение таблиц уменьшает полный размер кластеризованных и некластеризованных индексов сбалансированного дерева, что предоставляет ряд преимуществ в области масштабируемости.

- Операции вставки и обновления должны также вставлять и обновлять страницы индексов. Когда таблица разделена, обновляются только страницы индексов тех разделов, которые были обновлены.
- Обслуживание индексов может быть довольно затратной операцией. Индексы раздела могут быть значительно меньше, что существенно сократит влияние реиндексации и дефрагментации индекса на производительность.
- Резервирование части таблицы с использованием файловых групп облегчает архивирование.
- Малый размер сбалансированных деревьев индексов ускоряет поиск индексов.
- Возможно, наиболее существенным преимуществом масштабирования разделенных таблиц является то, что SQL Server может использовать параллельные процессорные операции при работе с разделенными таблицами.



Преимущества разделения таблиц в части производительности практически невозможно почувствовать, пока таблица не станет чрезвычайно большой (например, с миллиардом строк в базе данных размером в несколько терабайтов). На самом деле, по результатам тестирования, разделение таблиц негативно сказывается на производительности, если таблица имеет меньше миллиона строк, так что зарезервируйте эту технологию до действительно крупных задач. Возможно, именно по этой причине функция разделения таблиц не была включена в редакцию Standard.

Создание разделения таблицы в SQL Server 2005 является простейшим четырехшаговым процессом.

1. Создайте функцию разделения, определяющую характер сегментирования данных.
2. Создайте схему разделения, назначающую разделы файловым группам.
3. Создайте таблицу с некластеризованным первичным ключом.
4. Создайте кластеризованный индекс для таблицы, используя схему и функцию разделения.

Функции и схемы разделения работают совместно для сегментирования данных (рис. 53.1).



В приведенном примере использована база данных `PartitionDemo`, которая имеет три файловые группы: `Primary`, `Part1` и `Part2`. Соответствующий сценарий вы можете загрузить с сайта данной книги.

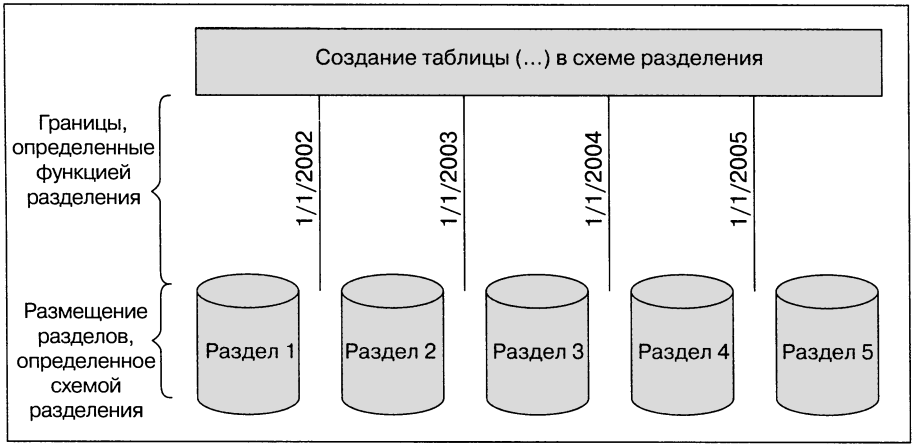


Рис. 53.1. Функция разделения используется схемой разделения для помещения данных в отдельные файловые группы

Создание функции разделения

Под *функцией разделения* понимается механизм определения границ разделов.

При разделении таблиц данные могут сегментироваться на основании одного столбца. Даже если таблица еще не определена, функция должна знать тип данных столбца сегментирования. Таким образом, параметром функции разделения является тип данных, который использовался для сегментирования данных. В следующем примере функция `fnYears` принимает тип данных `datatime`.

Важным аспектом разделов является то, что они функционируют только как барьеры, т.е. не определяют верхний и нижний пределы диапазона данных таблицы. В качестве барьера граница раздела обычно определяет верхнюю границу диапазона; данные, которые находятся за ней, рассматриваются как *следующий* раздел.

Границы, или диапазоны, определяются как левый или правый. Левая граница подразумевает, что данные, равные ей, включаются в раздел, находящийся слева от нее. Например, граница `'31/12/2004'` создает два раздела. Нижний раздел будет включать данные, относящиеся к данным, меньшим или равным `'31/12/2004'`, а верхний раздел — данные, относящиеся к более поздним датам.

Под *правой границей* подразумевается то, что граница относится к верхнему разделу. Например, для отделения нового, 2005 года, правый диапазон должен содержать границу `'1/1/2005'`. Все значения, меньшие этой даты, переходят в левый, или нижний, раздел. Все данные с датами, равными или большими заданной, переходят в следующий раздел. Следующие две функции используют левую и правую границу, формируя один и тот же результат:

```
CREATE PARTITION FUNCTION fnyears(DateTime)
AS RANGE LEFT FOR VALUES
('31/12/2001','31/12/2002','31/12/2003','31/12/2004');
```

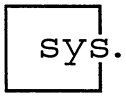
и

```
CREATE PARTITION FUNCTION fnYearsRT(DateTime)
AS RANGE RIGHT FOR VALUES
('1/1/2002', '1/1/2003', '1/1/2004', '1/1/2005');
```

Обе эти функции создают четыре определенные границы и пятую, неопределенную.



Разделение таблиц в SQL Server 2005 является декларативным — это значит, что таблица сегментируется по значениям данных. Хеш-разделы сегментируют данные случайным образом. Для имитации хеш-разделов и случайного распределения данных по множеству дисковых подсистем DAS определите таблицу, используя файловые группы, а затем добавьте в каждую файловую группу множество файлов.



Существуют три представления каталогов, которые отображают информацию о функции разделения: `syspartition_functions`, `syspartition_function_range_values` и `syspartition_parameters`.

Создание схем разделения

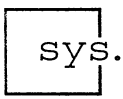
Схема разделения строится на функции разделения и определяет физическое размещение разделов. Физические разделы таблиц могут размещаться в одной и той же файловой группе или распределяться среди нескольких. В первом примере схема разделения `psYearAll` использует функцию разделения `pfYearsRT` и помещает все разделы в файловую группу `Primary`:

```
CREATE PARTITION SCHEME psYearsAll
AS PARTITION pfYearsRT
  ALL TO ([Primary]);
```

Для помещения разделов таблицы в собственную файловую группу опустите ключевое слово `all` и перечислите файловые группы в индивидуальном порядке:

```
CREATE PARTITION SCHEME psYearsFiles
AS PARTITION pfYearsRT
  TO (Part01, Part02, Part03, Part04, Part05);
```

Функции и схемы разделения должны создаваться с помощью программного кода T-SQL, однако после создания их можно увидеть в окне Object Explorer утилиты Management Studio под узлом базы данных `Storage` (рис. 53.2).



Для просмотра информации о схеме разделения программным путем выполните запрос к базе `syspartition_schemes`.

Создание разделенной таблицы

После того как созданы функция и схема разделения, создание таблицы не составляет труда. Разделенная таблица должна создаваться с некластеризованным первичным ключом. После этого добавление в таблицу кластеризованного индекса фактически сегментирует таблицу на основании схемы разделения (рис. 53.3). Обратите внимание на то, что на панели свойств таблицы также отображается схема разделения, используемая таблицей.

Функции и схемы разделения не имеют владельцев, поэтому при обращении к ним не нужно использовать четырехкомпонентное имя или упоминать в имени владельца схемы.

Следующая таблица аналогична таблице рабочих заказов базы данных Adventureworks, но реализована в производственной среде:

```
CREATE TABLE dboWorkOrder (
  WorkOrderID INT NOT NULL PRIMARY KEY NONCLUSTERED,
  ProductID INT NOT NULL,
  OrderQty INT NOT NULL,
  StockerQty INT NOT NULL,
  ScappedQty INT NOT NULL,
```

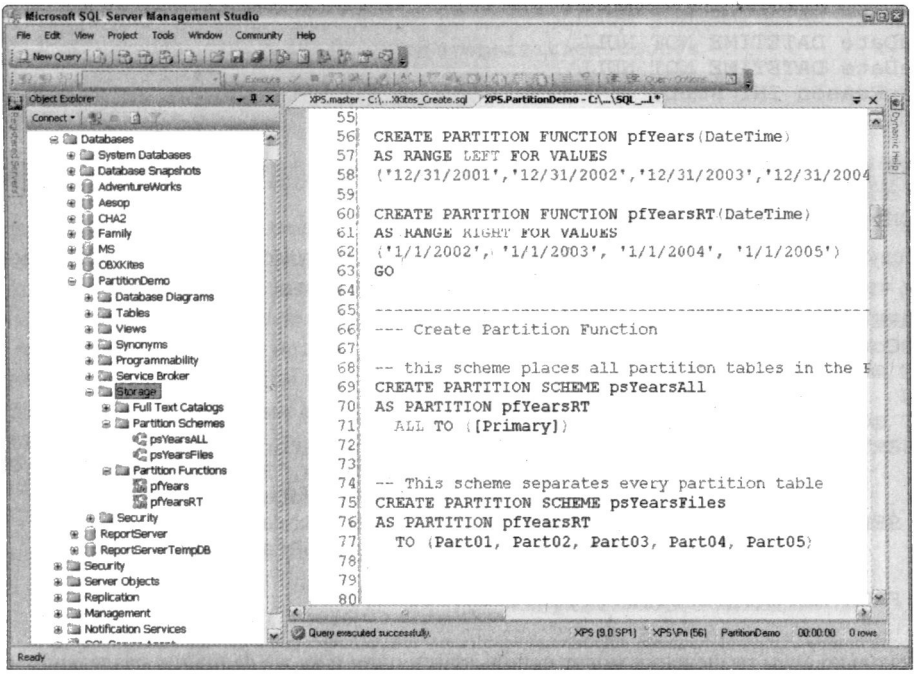


Рис. 53.2. Конфигурацию разделов можно увидеть в Object Explorer

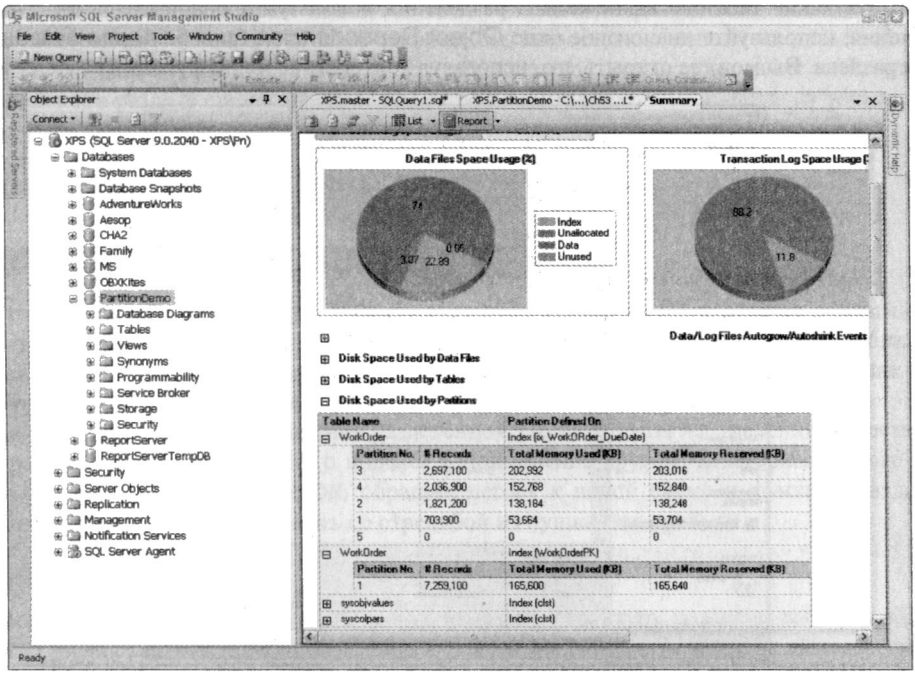


Рис. 53.3. Отчет `Disk Usage` *на странице* Summary *утилиты Management Studio для базы данных* `PartitionDemo` *отображает информацию о каждом из разделов*

```
StartDate DATETIME NOT NULL,
  EndDate DATETIME NOT NULL,
  DueDate DATETIME NOT NULL,
  ScapReason INT NULL,
  ModifiedDate DATETIME NOT NULL
  );
CREATE CLUSTERED INDEX ix_WorkORder_DueDate
  ON dbo.WorkOrder (DueDate)
    ON psYearsAll(DueDate);
```

Следующий сценарий вставляет 7259100 строк в таблицу `WorkOrder` за 2 минуты 42 секунды, что подтверждается на странице Summary базы данных:

```
DECLARE @Counter INT;
SET @Counter = 0;
WHILE @Counter < 100
BEGIN
  SET @Counter = @Counter + 1;
  INSERT dbo.WorkOrder (ProductID, OrderQty, StockedQty,
                        ScrappedQty, StartDate, EndDate, DueDate,
                        ScrapReasonID, ModifiedDate)
     SELECT ProductID, OrderQty, StockedQty, ScrappedQty,
            StartDate, EndDate, DueDate, ScrapReasonID,
            ModifiedDate
     FROM AdventureWorks.Production.WorkOrder;
END;
```

Несколько схем разделения могут использовать одну и ту же функцию разделения. С точки зрения архитектуры это может иметь смысл, если несколько таблиц должны быть разделены с использованием одних и тех же границ, что повышает единообразие разделов. Для проверки того, какие таблицы какие схемы разделения используют, основываясь на функциях разделения, используйте диалоговое окно Object Dependencies (рис. 53.4) для функции или схемы раздела. Вы можете открыть его, используя контекстное меню функции разделения.

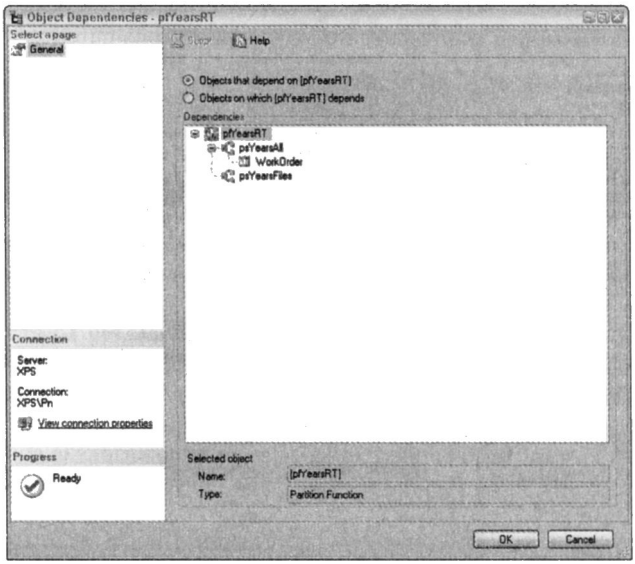


Рис. 53.4. В окне Object Dependencies видно, что таблица `WorkOrder` использует схему разделения `psYearAll`, которая в свою очередь использует функцию `pfYerarsRT`



Для просмотра информации об использовании разделов взгляните на представления `syspartitions` и `syspartition_counts`.

Выполнение запросов к разделенным таблицам

В разделенных таблицах следует отметить одну прекрасную особенность: для выполнения к ним запросов не требуется написания какого-либо особого программного кода. Оптимизатор запросов для извлечения данных автоматически выбирает правильные таблицы.

Оператор `$partition` может вернуть целочисленный идентификатор разделенной таблицы при использовании с функцией разделения. В следующем фрагменте кода подсчитывается количество строк в каждом из разделов:

```
SELECT $PARTITION.pfYearsRT(DueDate) AS Partition,
     COUNT(*) AS Count
  FROM WorkOrder
  GROUP BY $PARTITION.pfYearsRT(DueDate)
  ORDER BY Partition;
```

Будет получен следующий результат:

```
Partition   Count
----------- -----------
1           703900
2           1821200
3           2697100
4           2036900
```

Следующий запрос отбирает данные за один год, таким образом, все эти данные находятся в одном разделе. Анализируя план выполнения запроса, показанный на рис. 53.5, мы видим, что оптимизатор запросов использует высокоскоростное сканирование кластеризованного индекса в разделе с идентификатором `PtnIds1005`:

```
SELECT WorkOrderID,ProductID, OrderQty, StockedQty, ScrappedQty
  FROM dbo.WorkOrder
  WHERE year(DueDate) = '2002';
```

Изменение разделенных таблиц

Чтобы обновить разделенную таблицу в соответствии с изменившимися данными и выполнить тестирование разных схем разделения, ее можно без труда модифицировать. Несмотря на видимую простоту команд, изменение структуры разделенных таблиц, как вы понимаете, не может произойти быстро.

Простейшим способом изменения разделенной таблицы является удаление существующего ограничения кластеризованного индекса и применение другого кластеризованного индекса, включающего предложение `ON`, обращающегося к новой схеме и функции разделения. Однако этот метод можно сравнить со стрельбой из пушки по воробьям.

Слияние разделов

Модификаторы `MERGE` и `SPLIT` могут изменить структуру разделов таблицы. Команда `ALTER PARTITION... MERGE RANGE` позволяет эффективно удалить одну из границ из функции разделения и объединить два раздела. Например, удаление границы между 2003 и 2004 гг. в функции разделения `pfYearsRT` и объединение данных из соответствующих разделов можно выполнить с помощью следующей инструкции:

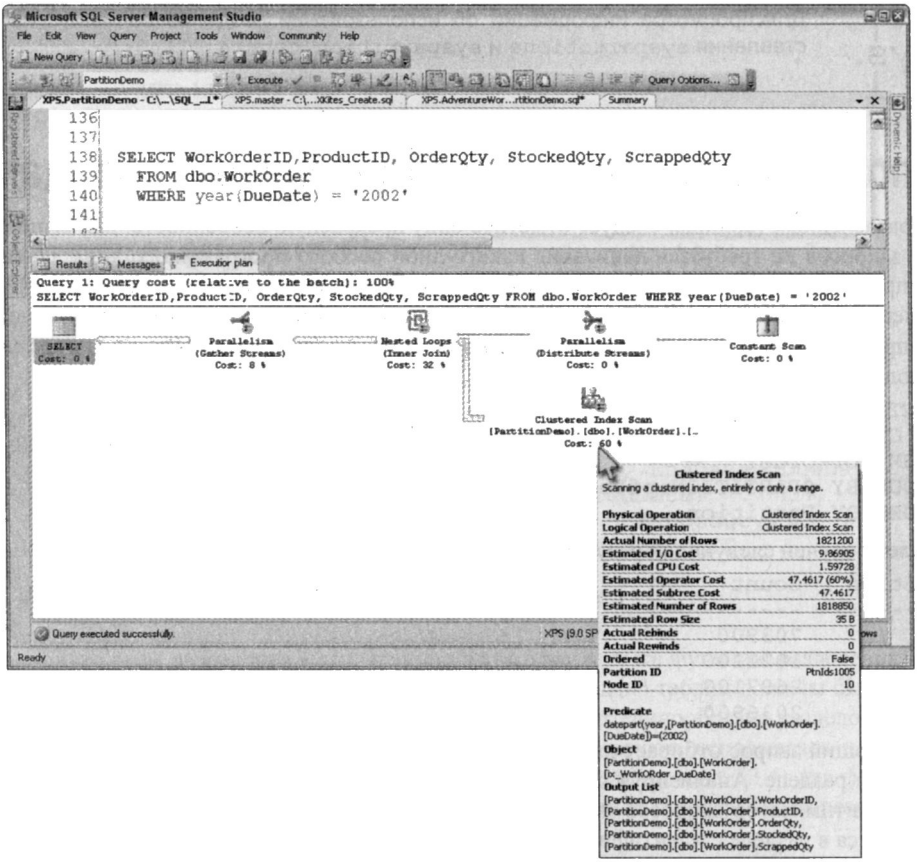


Рис. 53.5. Логическая операция сканирования кластеризованного индекса содержит новый параметр Partition ID

```
ALTER PARTITION FUNCTION pfYearsRT()
MERGE RANGE ('1/1/2004');
```

Совершенно очевидно, что если выполнить описанный выше подсчет строк в разделах, то запрос вернет три раздела, а при просмотре функции разделения в Object Explorer отобразится сценарий с тремя границами.



Если несколько таблиц совместно используют одну и ту же модифицируемую схему и функцию разделения, эти изменения повлияют на все таблицы.

Разбиение разделов

Для разбиения существующего раздела вначале нужно назначить файловую группу новому разделу в схеме, используя команду `ALTER PARTITION...NEXT USED`. После этого может быть модифицирована функция разделения. Команда `ALTER PARTITION...SPLIT RANGE` вставляет новую границу в функцию разделения. Именно команда изменения функции разделения фактически выполняет всю работу.

В следующем примере сегмент данных о рабочих заказах за 2003–2004 годы разбивается на два раздела. Новый раздел будет содержать данные только за июль 2004 года — последний месяц с данными в учебной базе данных AdventureWorks:

```
ALTER PARTITION SCHEME psYearsFiles
NEXT USED [Primary];
ALTER PARTITION FUNCTION pfYearsRT()
SPLIT RANGE ('1/7/2004');
```

Переключение таблиц

Переключение таблиц представляет собой прекрасную возможность перемещать всю таблицу в пространство раздела разделенной таблицы или удалять некоторый раздел, превращая его в обособленную таблицу. Это очень полезно при импорте новых данных, однако следует обратить внимание на некоторые ограничения.

- Каждый индекс в разделенной таблице также должен быть разделенным.
- Новая таблица должна иметь те же столбцы (за исключением столбцов идентичности), индексы и ограничения (включая внешние ключи), что и разделенная таблица, за исключением того, что новая таблица не может быть разделенной.
- Исходная разделенная таблица не может быть назначением внешнего ключа.
- Таблица не может быть опубликована с помощью репликации и не может иметь привязанные к схеме представления.
- Новая таблица должна иметь ограничение проверки, ограничивающее диапазон данных нового раздела, таким образом, СУБД SQL Server не должна перепроверять диапазон данных.
- Как обособленная таблица, так и раздел, который ее получает, должны находиться в одной файловой группе.
- Раздел назначения должен быть пустым.

По существу, преобразование таблицы в раздел является реорганизацией метаданных с целью переназначения существующей таблицы в качестве раздела (рис. 53.6). Никакие данные фактически не перемещаются, так что переключение таблицы происходит практически мгновенно, независимо от размеров таблицы.

Выделение новой таблицы

Таблица `WorkOrderNEW` удовлетворяет вышеописанным критериям и будет хранить данные за август 2004 года из базы данных Adventureworks:

```
CREATE TABLE dbo.WorkOrderNEW (
  WorkOrderID INT IDENTITY NOT NULL,
  ProductID INT NOT NULL,
  OrderQty INT NOT NULL,
  StockedQty INT NOT NULL,
  ScrappedQty INT NOT NULL,
  StartDate DATETIME NOT NULL,
  EndDate DATETIME NOT NULL,
  DueDate DATETIME NOT NULL,
  ScrapReasonID INT NULL,
  ModifiedDate DATETIME NOT NULL
  )
  ON Part05;
```

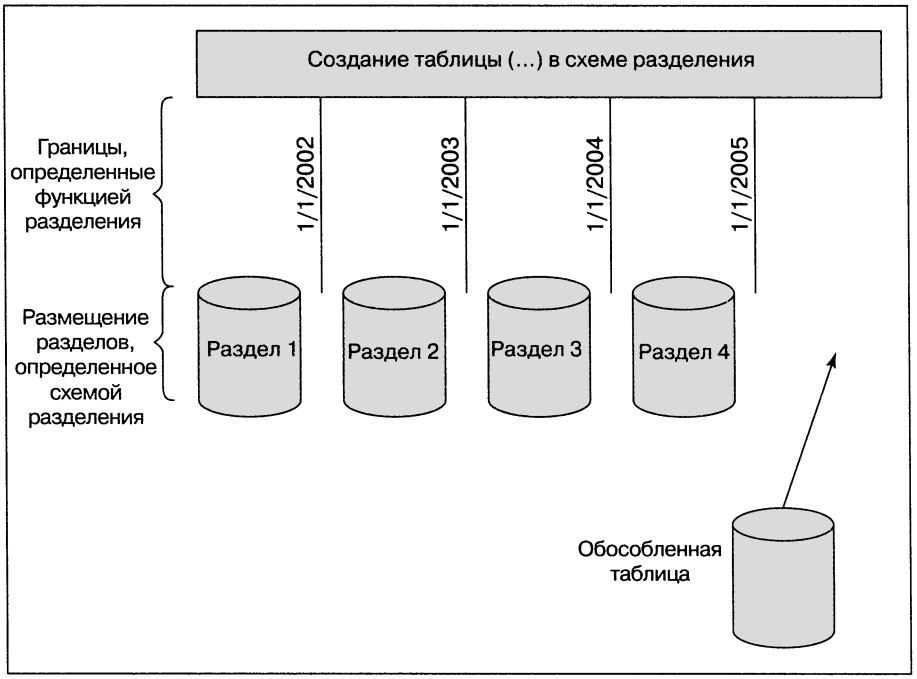


Рис. 53.6. Переключение таблицы фактически является изменением внутренних метаданных базы данных, преобразующим обособленную таблицу в один из разделов

Индексы, идентичные предыдущей таблице, будут созданы в разделенной таблице:

```
ALTER TABLE dbo.WorkOrderNEW
  ADD CONSTRAINT WorkOrderNEWPK
  PRIMARY KEY NONCLUSTERED (WorkOrderID, DueDate)
go
CREATE CLUSTERED INDEX ix_WorkOrderNEW_DueDate
  ON dbo.WorkOrderNEW (DueDate)
```

Добавляем обязательное ограничение:

```
ALTER TABLE dbo.WorkOrderNEW
  ADD CONSTRAINT WONewPT
    CHECK (DueDate BETWEEN '1/8/2004' AND '31/8/2004');
```

Теперь импортируем новые данные из Adventureworks, используя данные за январь 2004 года:

```
INSERT dbo.WorkOrderNEW (ProductID, OrderQty, StockedQty,
        ScrappedQty, StartDate, EndDate, DueDate, ScrapReasonID,
        ModifiedDate)
SELECT
    ProductID, OrderQty, StockedQty, ScrappedQty,
    DATEADD(mm,7,StartDate), DATEADD(mm,7,EndDate),
    DATEADD(mm,7,DueDate), ScrapReasonID,
    DATEADD(mm,7,ModifiedDate)
  FROM AdventureWorks.Production.WorkOrder
  WHERE DueDate BETWEEN '1/1/2004' and '31/1/2004';
```

Новая таблица теперь имеет 3158 строк.

Переключение в разделенную таблицу

Исходная разделенная таблица, созданная ранее в этой главе, имела неразделенный некластеризованный первичный ключ. Так как одно из правил переключения в разделенную таблицу гласит, что все индексы должны быть разделенными, первой задачей в следующем примере является удаление и перестройка первичного ключа таблицы `WorkOrder`, чтобы он стал разделенным:

```
ALTER TABLE dbo.WorkOrder
  DROP CONSTRAINT WorkOrderPK
ALTER TABLE dbo.WorkOrder
  ADD CONSTRAINT WorkOrderPK
  PRIMARY KEY NONCLUSTERED (WorkORderID,DueDate)
    ON psYearsAll(DueDate);
```

Далее разделенной таблице нужен свободный раздел.

```
ALTER PARTITION SCHEME psYearsFiles
  NEXT USED [Primary]
ALTER PARTITION FUNCTION pfYearsRT()
  SPLIT RANGE ('1/8/2004')
```

Осуществление переключения

Инструкция `ALTER TABLE...SWITCH` переместит новую таблицу в заданный раздел. Чтобы определить пустой раздел назначения, выберите на странице Summary базы данных отчет Disk Usage.

```
ALTER TABLE WorkOrderNEW
  SWITCH TO WorkOrder PARTITION 5
```

Обратное переключение

Та же технология может использоваться и для переключения раздела таблицы в обособленную таблицу. Поскольку никакого слияния на этот раз не происходит, выполнять обратное переключение проще, чем прямое. Следующий программный код берет только что созданный раздел таблицы `WorkOrder` и переконфигурирует метаданные базы так, чтобы этот раздел стал обособленной таблицей:

```
ALTER TABLE
  SWITCH PARITION 1 to WorkOrderArchive
```

Подвижные разделы

Если проявить немного воображения, то технологию создания и слияния существующих разделов можно применить для создания подвижных разделов. Подвижные разделы полезны для функций разделения, основанных на времени, например для создания разделов для каждого месяца. Каждый месяц подвижный раздел расширяется на новый месяц. Для создания 13-месячного подвижного раздела выполняйте каждый месяц следующие действия.

1. Добавьте новую границу.
2. Укажите границе на следующую используемую файловую группу.
3. Объедините два последних раздела, чтобы объединить все данные.

Включение таблиц в разделы и их обратное исключение могут улучшить структуру подвижных разделов за счет включения полностью заполненных текущих таблиц и обратного перемещения таблиц в архив.

Индексация разделенных таблиц

Большие таблицы подразумевают существование больших индексов, поэтому некластеризованные индексы при желании также можно разделять.

Создание разделенных индексов

Разделенные некластеризованные индексы должны включать столбец, используемый функцией разделения, и создаются с помощью того же предложения ON, что и разделенные кластеризованные индексы.

```
CREATE INDEX WorkOrder_ProductID
  ON WorkOrder (ProductID, DueDate)
    ON psYearsFiles(DueDate);
```

Обслуживание разделенных индексов

Одним из преимуществ разделенных индексов является то, что их можно обслуживать по разделам. В следующем примере перестраивается только что добавленный пятый раздел:

```
ALTER INDEX WorkOrder_ProductID
  ON dbo.WorkOrder
  REBUILD
  PARTITION = 5
```

Так как разделение доступно только в редакции Enterprise Edition, в этой же редакции доступна и интерактивная перестройка индексов.

Удаление разделения

Для удаления разделения любой таблицы нужно удалить кластеризованный индекс и создать новый, но уже без использования предложения ON. При удалении кластеризованного индекса следует добавить параметр MOVE TO, чтобы консолидировать данные в заданную файловую группу, ликвидируя таким образом разделение таблицы:

```
DROP INDEX ix_WorkOrder_DueDate
  ON dbo.Workorder
  WITH (MOVE TO [Primary]);
```

Работа с индексированными представлениями

Популярной методикой повышения производительности базы данных является создание *денормализованной* копии некоторого подмножества данных и сохранение ее в любом месте для быстрого чтения. Например, данные, хранимые в пяти больших таблицах, можно извлечь и сохранить в одной широкой таблице. Мне однажды пришлось выполнять экстремальную денормализацию в проекте, заменяя запрос, содержавший добрый десяток объединений, одной таблицей. сокращая таким образом время поиска от нескольких минут до одной секунды. В этом проекте все прошло гладко, так как данные были доступны только для чтения. Когда же запрос выполняется относительно рабочих данных, денормализация может стать источником проблем целостности данных.

Компания Microsoft предложила альтернативу денормализации фактических данных. *Индексированные представления* SQL Server на самом деле являются кластеризованными индексами, хранящими денормализованное множество данных (рис. 53.7).

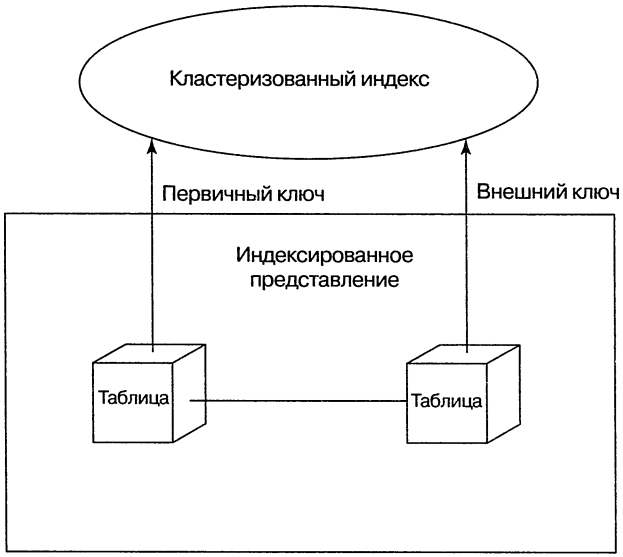


Рис. 53.7. Индексированные представления перекидывают мост между двумя таблицами, которые на самом деле должны содержать десяток объединений

Вместо создания таблиц для денормализации объединений может быть создано представление, отбирающее два первичных ключа из объединяемых таблиц. Кластеризованный индекс, созданный на представлении, хранит допустимые данные из пар первичного и внешнего ключей.

В то время как обычное представление хранит инструкцию SQL `SELECT` и данные не материализуются до тех пор, пока представление не будет вызвано, индексированное представление хранит копию данных в кластеризованном индексе. Кластеризованные индексы объединяют страницы данных и листья индекса двоичного дерева для хранения фактических данных в физическом порядке индекса. Кластеризованный индекс использует представление как среду определения хранимых столбцов.

В индексированных представлениях существует множество ограничений, в частности, описанные ниже.

- При создании представлений и когда какие-либо подключения пытаются модифицировать данные в базовых таблицах, должны быть включены пустые значения ANSI и заключенные в кавычки идентификаторы.
- Индекс должен быть уникальным и кластеризованным. Таким образом, представление должно создать уникальный набор строк без использования ключевого слова `DISTINCT`. Это может стать проблемой, поскольку среди ситуаций, в которых необходима денормализация, многие содержат отношения "многие ко многим", имеющие тенденцию генерировать дублирующиеся строки в результирующем наборе данных. По этой причине большинство индексированных представлений покрывают только отношения "один ко многим".

- Таблицы в представлении должны быть именно таблицами (а не вложенными представлениями) локальной базы данных, и ссылки на них должны выполняться с помощью двухкомпонентного имени (*владелец*.*таблица*).
- Представление должно создаваться с параметром WITH SCHEMA BINDING.



Поскольку индексированные представления требуют связывания со схемой, их нельзя смешивать с переключением разделенных таблиц и подвижными разделами. Для переключения к разделенной таблице следует удалить индексированное представление, выполнить переключение, после чего снова воссоздать индексированное представление.

Приведем пример индексированного представления, используемого для денормализации больших запросов. Следующее представление отбирает данные из таблиц Contact и Product базы данных OBXKites:

```
USE OBXKites;

SET ANSI_Nulls ON;
SET ANSI_Padding ON;
SET ANSI_Warnings ON;
SET ArithAbort ON;
SET Concat_Null_Yields_Null ON;
SET Quoted_Identifier ON;
SET Numeric_RoundAbort OFF;
GO

CREATE VIEW vContactOrder
WITH SCHEMABINDING
AS
SELECT c.ContactID, o.OrderID
  FROM dbo.Contact as c
    JOIN dbo.[Order] as o
      ON c.ContactID = o.ContactID;
GO

CREATE UNIQUE CLUSTERED INDEX ivContactOrder ON vContactOrder
(ContactID, OrderID);
```

Индексированные представления и запросы

Когда оптимизатор запросов SQL Server создает план выполнения запроса, он включает кластеризованный индекс индексированного представления в качестве одного из индексов, используемого запросом, даже если запрос явно не ссылается на это представление.

Это означает, что кластеризованный индекс индексированного представления может выступать в роли покрывающего индекса, ускоряющего запросы. Когда оптимизатор запросов отбирает кластеризованный индекс индексированного представления, план выполнения запроса отбирает его для сканирования (рис. 53.8). Следующий запрос отбирает те же данные, что и индексированное представление:

```
SELECT Contact.ContactID, OrderID
  FROM dbo.Contact
    JOIN dbo.[Order]
      ON Contact.ContactID = [Order].ContactID
```

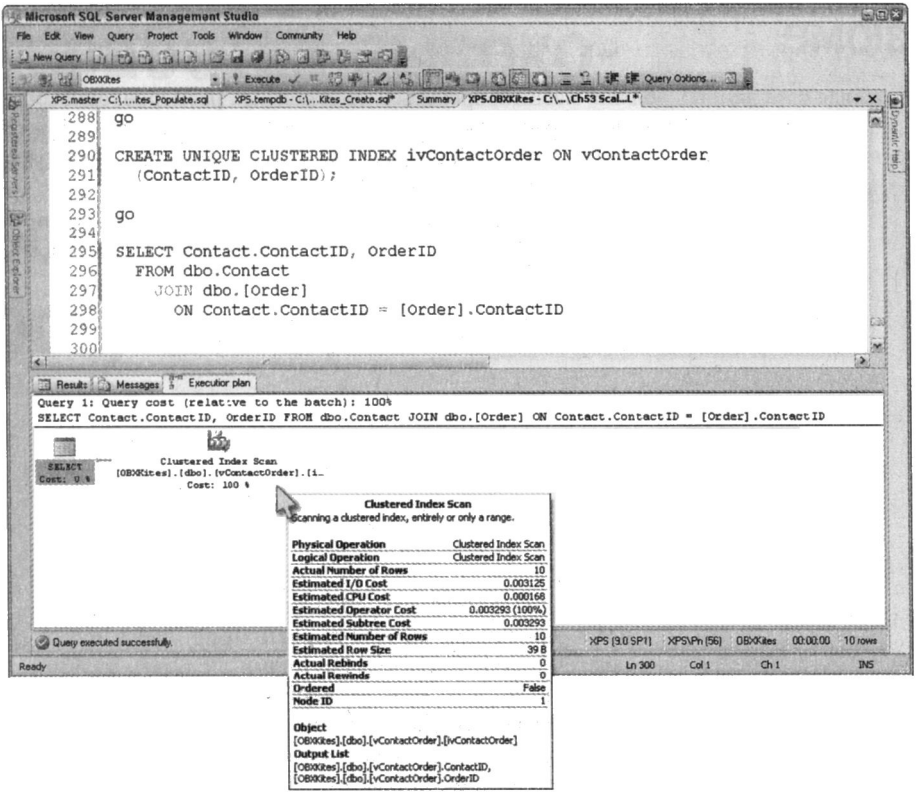


Рис. 53.8. План выполнения запроса осуществляет сканирование кластеризованного индекса для непосредственного извлечения данных вместо доступа к таблицам базы



В то время как индексированные представления в сущности остались такими же, как в SQL Server 2000, оптимизатор запросов теперь может использовать индексированные представления в большем числе типов запросов.



Обычное создание индексированных представлений без полного анализа их использования запросами скорее навредит производительности, чем улучшит ее. Обновленные индексированные представления скрывают в себе серьезную угрозу производительности, так что избегайте их в транзакционных базах данных. Обдуманно добавляйте их в базы данных, используемые преимущественно для отчетов, аналитической обработки данных и запросов, идентифицируя специфичные объединения, препятствующие часто выполняемым запросам, и ювелирно точно вставляя кластеризованные индексы индексированных представлений.

Обновление индексированных представлений

Как и в любой денормализованной копии данных, основная сложность заключается в поддержании данных в постоянно обновленном состоянии. То же относится и к индексированным представлениям. Когда данные базовых таблиц обновляются. индексированное представление должно находиться с ними в синхронизации. Этот процесс полностью прозрачен для пользователя и больше относится к области производительности, чем программирования.

Резюме

Не каждая база данных должна масштабироваться до огромных размеров, однако для проектов, объем которых стремится к терабайтам, SQL Server 2005 предлагает ряд эффективных технологий, позволяющих справиться с подобным ростом. Тем не менее даже эти расширенные технологии не могут заменить теорию оптимизации.

Разработка высокопроизводительных поставщиков доступа к данным

ГЛАВА

54

В этой главе...

Концепции доступа к данным

Работа с объектами доступа к данным (DAO)

Производства

Поставщики данных

Сегодня почти вся информация хранится в базах данных и в основном накапливается ежедневно. Эти массивы информации не имеют практически никакого значения, если их невозможно просмотреть с определенной точки зрения. Именно в этом аспекте информация приобретает реальную ценность. Если вы предоставляете пользователю некоторый образ данных, то можете придать ему смысловое значение и даже предсказать будущие модели, направления и тенденции. Когда информация представляется в некотором смысловом аспекте, пользователи и приложения имеют возможность принимать решения, влияющие на реальные события.

В качестве примера рассмотрим отчеты о кредитах. Если в реальной жизни покупки совершаются с использованием дебетовых или кредитных карточек, то вся эта информация накапливается в базе данных. Каждая покупка сама по себе не несет в себе много информации, однако когда составляется кредитный отчет за длительный период времени, это позволяет составить определенное представление о покупателе. При грамотном подходе это может принести только пользу.

В этой главе мы рассмотрим некоторые концепции доступа к данным. В частности, рассмотрим модели, использующие объекты доступа к данным VB.NET (DAO), фабрики и поставщиков. В среде разработки Microsoft модель DAO представляет собой множество классов, реализующих промежуточный слой между базой данных и клиентским приложением. Эта технология не полагается целиком на методы самой базы данных.



Когда возможно, старайтесь использовать программные шаблоны. Под *шаблоном* подразумевается технология, обычно применяемая для решения определенной задачи. Такое решение обычно включает в себя классы и программный код, необходимый для их взаимодействия.



В примерах программного кода этой главы обработка исключений проиллюстрирована с использованием общего класса `Exception`, содержащегося в блоке `Catch`. На практике эти исключения будут привязаны к типам ошибок, которые вы хотите обработать.

Концепции доступа к данным

Одни факторы определяют потребности доступа к данным, другие — влияют на конкретную реализацию доступа. Главное — удобный доступ к данным, и могут быть созданы различные эвристические решения и правила, способствующие выбору подходящей модели доступа к данным.

В чем ценность хорошего доступа к данным

В идеальном случае реализация удобного доступа к данным обеспечивает высокопроизводительный, гибкий и целостный "слой", с помощью которого любое количество клиентов могут сохранять информацию, изменять ее и извлекать по мере необходимости. Этот слой может изолировать клиента от каких-либо знаний о базе данных и быть настолько прозрачным, насколько это возможно. Также он может выступать в качестве связующего звена между функциональностью базы данных и клиентским приложением. Грамотно спроектированный уровень данных может многократно использоваться, модернизироваться и обслуживаться с минимальными усилиями.

Определение требований

Существует множество моделей доступа к данным, и каждая из них имеет свои сильные и слабые стороны применительно к решению реальных задач.

Перед тем как выбрать модель, следует задать себе несколько вопросов.

- Насколько часто должны происходить изменения в схеме? Как часто требуется применение другой схемы? В расширяемой, легко обслуживаемой и дружественной для пользователя модели должны поддерживаться незаметные для пользователя преобразования схемы.
- Насколько часто будут добавляться новые базы данных? Для поддержки этих изменений выбирайте шаблон, который будет просто расширять и развертывать.
- Насколько "большим" будет уровень данных? Если существует потребность в модели, которая допускает существование большего количества объектов, выбирайте модель, которую будет не сложно поддерживать.
- Должны ли данные использоваться совместно группой пользователей? Выбирайте ту модель, которая позволяет прозрачно для пользователя внедрять объекты, улучшающие развертывание, расширяемость и обслуживание.
- Следует ли поддерживать обновление транзакций? Если это так, учтите изменения модели в проекте. Взаимодействие транзакций в базе данных должно находиться на среднем уровне. Этот вопрос выходит за рамки рассмотрения настоящей книги, и его нужно рассматривать как дополнительное соглашение.

Прозрачность данных должны обеспечивать и другие вопросы. Главное в данном вопросе — это потратить некоторое время на задачи уровня данных и учесть их в проекте.

В табл. 54.1 приведены некоторые критерии, которые следует принять во внимание при выборе модели проектирования уровня доступа к данным. Далее в этой главе будут продемонстрированы все три эти модели.

Таблица 54.1. Характеристики шаблонов доступа к данным

Проект	Сложность	Характер обслуживания	Повторное использование	Расширяемость	Дружественность развертывания
Объекты доступа к данным (DAO)	Низкая	Нейтральная	Низкая	Низкая	Высокая
Фабрики (Factories)	Нейтральная	Низкая	Нейтральная	Нейтральная	Высокая
Поставщики данных (Data Providers)	Высокая	Низкая	Высокая	Высокая	Высокая

Принимая во внимание эти характеристики, давайте более подробно познакомимся с тремя вышеперечисленными шаблонами.

Объекты доступа к данным



Все примеры программного кода, приведенные в настоящей главе, были созданы на основе VB.NET 2005. Их можно найти на сайте книги.

В шаблоне доступа к данным DAO наследование и полиморфизм используются для построения иерархии объектов данных. Общая модель продемонстрирована на рис. 54.1.

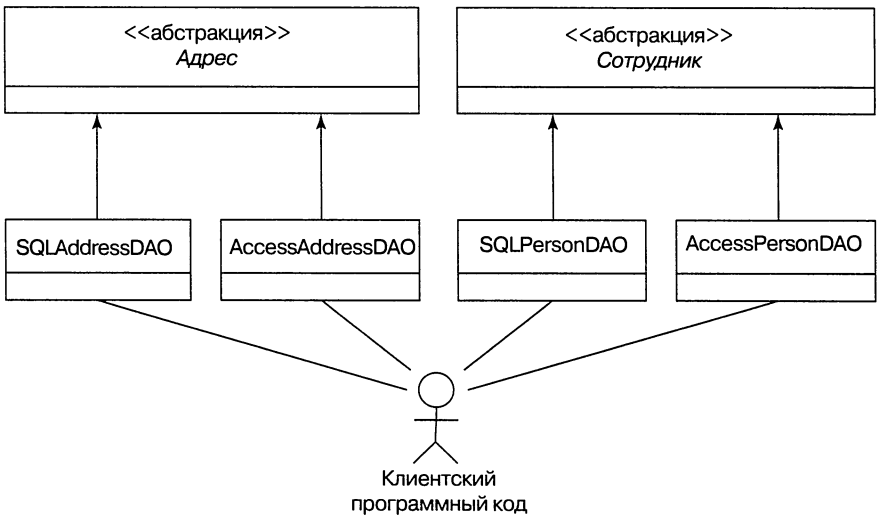
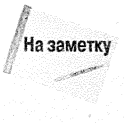


Рис. 54.1. В этом примере модели DAO показано, как управляется отдельная реализация и как клиент может ссылаться на каждую из них



В объектно-ориентированной среде *полиморфизмом* называют способность классов реализовывать единообразный общий интерфейс. На практике это значит, что два или более класса представляют общий интерфейс, который в клиентском программном коде может трактоваться как однотипный.



Модель DAO для обеспечения функциональности низлежащих классов полагается на технологии наследования. В этой модели всегда существуют базовые абстрактные классы, специфичные для объектов каждого типа данных, которые должен поддерживать разработчик. Специализированный класс создается для каждого объекта доступа к базе данных, который необходимо поддерживать, и содержит весь необходимый код SQL для своей базы данных. Обычно имя класса включает в себя имя базы данных, например `AddressSQLDAO` или `AddressAccessDAO`. Любой программный код, который обеспечивает использование этих объектов, должен изначально знать, какой тип DAO создавать.

Как работают сценарии DAO

Рассмотрим один из сценариев DAO. Основная задача заключаются в том, чтобы поддерживались базы данных Microsoft SQL Server 2005 и Microsoft Access. Определение типа объектов данных, который должен поддерживаться, основывается на внешнем флаге, содержащемся в файле конфигурации.

Базовый класс DAO `Address`, приведенный в следующем фрагменте программного кода, обеспечивает функциональность всех потомков, поддерживающих конкретную базу данных. Различные свойства членов будут общими. Единственным фактическим отличием является способ загрузки данных из источника.

Базовый класс `DAO Address`

```
Public MustInherit Class Address
' Определены различные закрытые переменные-члены,
' такие как City, State, Street1, Street2, и их общие члены.
'''-------------------------------------------
''' В каждом классе-наследнике будет реализован метод Load
''' для уникального источника данных, к которому осуществляется доступ.
'''-------------------------------------------
Public MustOverride Sub Load(ByVal strId As String)
End Class
```

Ключевое слово `MustOveride` требует, чтобы в каждом классе-наследнике был реализован собственный метод `Load`. В примере, приведенном ниже, продемонстрирован класс, который является наследником базового класса `Address`.

`AddressSQLDAO.vb`

```
Public Class SqlAddressDAO
  Inherits Address
' Определение значения ключа файла конфигурации.
  Private Const SQL_BINDING_INFO As String = "Sql_BindingInfo"
' Переменные класса унаследованы от базового класса Address.
'''-------------------------------------------------------
''' Замещение метода Load, определенного в классе Address.
'''-------------------------------------------------------
Public Overrides Sub Load(ByVal strId As String)
  Dim cmd As SqlCommand = Nothing
```

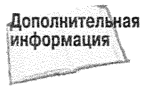
```
  Dim conn As SqlConnection = Nothing
  Dim dr As SqlDataReader = Nothing
Try
' Определение подключения к базе данных.
  conn = New SqlConnection(PropertyLoader.Instance _
.GetProperty(SQL_BINDING_INFO))
  conn.Open()
' Настройка и выполнение команды SQL.
  cmd = New SqlCommand(PropertyLoader.Instance
.GetProperty("Sql_Address_Select").Replace("%1", strId), conn)
  dr = cmd.ExecuteReader(CommandBehavior.CloseConnection)
  If dr.Read Then
' Загрузка переменных-членов чтения.
  End If
  Catch ex As Exception
' Пример обработки ошибок.
  Throw (ex)
  Finally
' Закрытие переменных
  If cmd IsNot Nothing Then cmd.Dispose()
  If conn IsNot Nothing Then conn.Close()
  If dr IsNot Nothing Then dr.Close()
 End Try
End Sub
End Class
```



В версии VB.NET 2005 было введено ключевое слово `IsNot`, реализующее более понятную форму условных логических проверок. Функционально это ключевое слово идентично предыдущему синтаксису — `Not <переменная> Is <значение>`.



Объекты данных, когда это возможно, осуществляют доступ к базам данных с помощью хранимых процедур. Это обеспечивает наилучшую изоляцию от изменений реализации базы данных. В крайнем случае обрамляйте инструкции SQL и по мере необходимости параметризуйте их.



Практические советы относительно абстрагирования доступа к базе данных в программном коде см. в главе 25.

Когда переменные-члены определены в абстрактном родительском классе `Address`, класс `AddressSQLDAO` должен инициализировать только собственные переменные-члены. Класс `AddressAccessDAO` обеспечивает ту же функциональность, но содержит специфичный для программы Microsoft Access код подключения и инструкции SQL.

В примере, приведенном ниже, продемонстрирован клиентский программный код, использующий классы `AddressSqlDAO` и `AddressAccessDAO` для сбора информации из источников данных SQL Server и Access.

TestMain.vb

```
Public Class TestMain
'''-------------------------------------------------
''' Создание объектов SQL Server и Access.
'''-------------------------------------------------
```

```
Sub Main()
  Try
' Локальная переменная, хранящая ссылку на общий адрес.
    Dim objAddr As Address
' Получение данных из базы данных SQL Server и их отображение.
    objAddr = New SqlAddressDAO
    objAddr.Load("2")
    Call PrintAddress( objAddr )
' Получение данных из базы данных Access и их отображение.
    objAddr = New AccessAddressDAO
    objAddr.Load("1")
    Call PrintAddress( objAddr )
    Catch ex As Exception
    Console.WriteLine(ex.Message)
  End Try
End Sub
-----------------------------------------------
''' Вывод информации об адресе.
'''-----------------------------------------
  Private Sub PrintAddress( ByVal objAddr As Address )
  Console.WriteLine(objAddr.Street1)
  Console.WriteLine(objAddr.City)
' Остальные атрибуты выводятся аналогичным образом.
 End Sub
End Class
```

Посмотрев на фактическое использование объектов DAO, становится совершенно очевидным, что доступ к данным SQL Server и Access реализован единообразно, поскольку управляется одним и тем же классом — `Address`. В то время как полиморфизм является гарантом доступа клиента к объектам данных SQL Server и Access, сохраняется необходимость знать, с каким конкретно источником данных ведется работа. Как было продемонстрировано в программном коде `TestMain`, соответствующие объекты данных создавались явным образом, и эта зависимость могла вызвать проблемы в случае поддержки другой базы данных или внесения изменений в существующую.

Концепция DAO довольно простая. В реализации и поддержке проекта не требуются какие-либо специализированные знания. Эта концепция больше всего подходит для небольшой группы объектов и источников данных, и ее можно реализовать в программном коде достаточно быстро.

Достоинства модели DAO

Клиент должен определить, какой источник данных использовать. Это довольно тесно связывает клиента с уровнем доступа к данным. Одним из подходов к решению этой проблемы является обеспечение соответствующих методов родительскими объектами, и их наследование потомками. Однако следует заметить, что это далеко не лучший способ абстрагирования логики источника данных.

```
Public MustInherit Class Address
'''-------------------------------------------------------
''' В этом классе реализуется та же функциональность,
''' что и в классе Address, описанном выше,
''' однако добавлен метод GetInstance(),
''' определяющий источник данных
'''-------------------------------------------------------
Public Shared Function GetInstance() As Address
If PropertyLoader.Instance.GetProperty("UseSql"). _
```

```
Equals("True") Then
  Return New AddressSqlDAO
Else
  Return New AddressAccessDAO
End If
End Function
' Вся остальная функциональность описана выше.
End Class
```

В то время как описанная схема работоспособна, все ошибки проектирования теперь перенесены на уровень данных. Еще одна проблема заключается в том, что родительский класс теперь должен явно знать обо всех своих потомках. Еще один недостаток связан с поддержкой. Если условная логика должна быть изменена, то следует обновлять каждый отдельный родительский объект данных, реализующий решение. Также следует обновлять все родительские объекты данных, если поддерживаемая база данных расширяется или добавляются какие-либо дополнительные объекты данных. Таким образом, исходя из вышесказанного, лучше стараться избегать использования этого подхода.

Возникает вопрос, как изолировать клиента от решения этих вопросов. Модель фабрики позволяет вынести бизнес-логику `GetInstance()` в отдельный класс и скрыть различия баз данных от клиентского программного кода.

Фабрики



Примеры программного кода, описываемые в настоящем разделе, созданы на языке VB.NET 2005; их можно найти на сайте книги на странице `files.htm`.

Фабрика является централизованной точкой создания объектов. Когда клиентский программный код нуждается в объектах данных, он выполняет запрос к фабрике, которая реализует их создание и инициализацию. К тому же фабрика может содержать бизнес-логику, определяющую, какие именно типы объектов следует создать. В примере, показанном на рис. 54.2, эта логика определяет, какая база данных используется: SQL Server или Access.



Фабрики представляют собой довольно простую конструкцию, изолирующую процесс создания экземпляров класса от клиентского приложения. Фабрики могут также выступать в качестве промежуточного уровня доступа к другим фабрикам. На рис. 54.2 оба способа доступа к данным были реализованы с помощью первичной фабрики — `DataObjectFactory`. Именно в ней реализована бизнес-логика определения конкретной базы данных и передача запросов на создание объектов к соответствующей фабрике баз данных.

В данном случае модель фабрики помогла отделить бизнес-логику от объектов доступа к данным и предоставила клиентскому программному коду возможность формирования единого подхода к созданию новых объектов данных. Преимущество этого подхода заключается в том, что при изменении логики источника данных вся модификация выполняется в одном месте, а не в каждом базовом классе DAO. Это упрощает поддержку и приводит программный код в соответствие с общими принципами объектно-ориентированного проектирования.

Создание классов DAO в специфичных для конкретных баз данных фабриках и использование их в качестве подчиненных фабрик позволяет добиться большей инкапсуляции и единообразия. К тому же это упрощает добавление новых баз данных — создается подходящая новая подчиненная фабрика и вносятся необходимые изменения в главную фабрику.

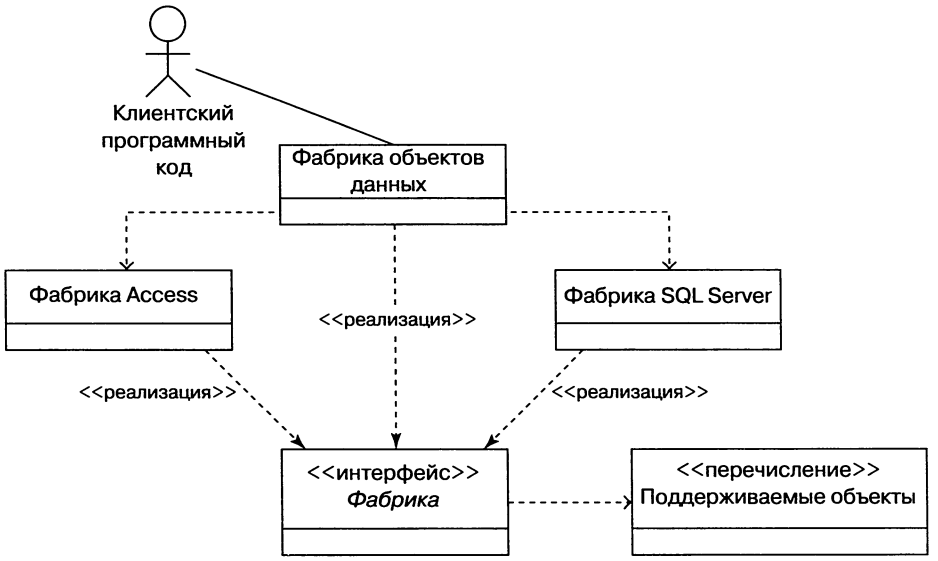


Рис. 54.2. В этом примере фабрики продемонстрировано взаимодействие клиентов и зависимости фабрик

Представленный выше программный код, использующий класс `Address`, остается пригодным. Фабрика просто консолидирует ответственность за создание объектов данных в едином месте. Это продемонстрировано в следующем примере.

Главная фабрика

```
Public Class DataObjectFactory
Inherits Factory
'''-----------------------------------------------
''' Делегирование запроса к объекту подчиненной фабрики.
'''-----------------------------------------------
Public Shared Shadows Function GetDAO(ByVal typeRequested _
As Factory.SupportedDAOs) As Object
  Try
    If CBool( PropertyLoader.Instance.GetProperty("UseSql") ) Then
      Return SqlServerFactory.GetDAO(typeRequested)
    Else
      Return AccessFactory.GetDAO(typeRequested)
    End If
  Catch ex As Exception
' Здесь обрабатывается ситуация, когда флаг или значение
'"UseSql" недоступны. Данное значение должно определяться
'в файле конфигурации приложения.
  Throw New ApplicationException( "Invalid UseSql value." )
  End Try
End Function
End Class
```

В приведенном выше фрагменте программного кода главная фабрика определяет источник данных и делегирует запрос специализированной фабрике.

Следующий фрагмент кода демонстрирует обработку этого делегирования фабрикой SqlServerFactory. Фактическая работа выполняется в приспособленной к конкретному источнику данных фабрике. Фабрика Access выполняет работу аналогичным образом.

Фабрика SqlServerFactory

```
Public Class SqlServerFactory
Inherits Factory
'''-------------------------------------------------------
''' Создание запрошенного объекта DAO.
'''-------------------------------------------------------
Public Shared Shadows Function GetDAO(ByVal typeRequested _
As Factory.SupportedDAOs) As Object
  Select Case typeRequested
    Case Factory.SupportedDAOs.Address
      Return New AddressSqlDAO
    Case Else
      Return Nothing
   End Select
 End Function
End Class
```

Класс SqlServerFactory знает, какие классы следует инициализировать, и не должен беспокоиться о других источниках данных. Он самодостаточен для объектов SQL.



В VB.NET общие методы не могут замещаться в подчиненных классах. Для достижения нужного эффекта они должны объявляться с использованием ключевого слова Shadows.

Теперь займемся вопросом использования фабрик в программном коде клиентских приложений. Рассмотрим пример.

MainTester.vb

```
Public Class MainTester
'''-------------------------------------------------------
''' Главный метод запрашивает объект данных у фабрики.
''' Так как фабрика возвращает тип данных Objects,
''' тип возвращаемого значения должен быть соответствующим
''' образом преобразован.
'''-------------------------------------------------------
Sub Main()
  Try
    Dim objAddr As Address
    objAddr = CType( SmarterDaoFactory.GetDAO( _
    Factory.SupportedDAOs.Address), Address)
    objAddr.Load("1")
    Call PrintAddress(objAddr)
    Catch ex As Exception
    Console.WriteLine(ex.Message)
  End Try
End Sub
'''-------------------------------------------------------
''' Вывод информации об адресе.
'''-------------------------------------------------------
Private Sub PrintAddress( ByVal objAddr As Address )
```

```
  Console.WriteLine(objAddr.Street1)
  Console.WriteLine(objAddr.City)
' Остальные атрибуты можно вывести тем же способом.
End Sub
End Class
```

Обратите внимание на то, что создание специфичного для источника данных объекта в клиентском коде было опущено — вместо этого вызывался метод `Factory.GetDAO(...)`. Этот метод содержит логику, определяющую, какая база данных должна использоваться, и направляющую запрос на создание объекта соответствующей подчиненной фабрике. Это существенный шаг в направлении изолирования клиента от тонкостей создания источника и объекта данных.

В следующем разделе мы обсудим все достоинства и недостатки использования фабрик. После этого будет показано, как сделать доступ к данным в клиентском программном коде полностью прозрачным.

Достоинства фабричной модели

Как уже говорилось ранее, одним из преимуществ использования фабрик является централизация бизнес-логики определения источника данных (класс `SmarterDAOFactory`). Главная фабрика обрабатывает запрос с помощью привязанных к конкретным источникам данных фабрик. Таким образом, фабрика `SmarterDAOFactory` выступает в роли посредника между клиентским кодом и специфичными для баз данных фабриками. Клиент, всегда работающий с одним и тем же источником данных, может направлять запрос к специфичным фабрикам напрямую, в обход главной фабрики. Дополнительные достоинства заключены в упрощении поддержки, облегченной концептуализации проекта, повышении удобочитаемости программного кода. К тому же разработчики еще на этапе проектирования знают, какие объекты поддерживаются, поскольку функция среды .NET IntelliSense отображает списки значений объектов фабрики. Более того, эта конструкция способна работать с бо́льшим числом объектов и баз данных, чем позволяет DAO.

Недостатки фабричной модели

Все недостатки, связанные с использованием фабричной модели, касаются реализации. Рассмотренный в настоящей главе пример очень прост, и использование подчиненных фабрик на самом деле не требовалось. В то же время на практике приходится иметь дело с бо́льшим количеством объектов и потенциальных источников данных, и использование подчиненных фабрик превращается в естественную потребность. В данном случае использование главной фабрики в качестве посредника требует дополнительной координации работы с подчиненными фабриками при добавлении новых баз и объектов данных. Все подчиненные фабрики должны программироваться таким образом, чтобы обслуживать запросы к неизвестным источникам данных. Еще одним недостатком является то, что главная фабрика должна предоставлять все множество доступных объектов; и это может вызвать проблемы, если подчиненные фабрики не поддерживают эти объекты.



Проект `Factory` в полном объеме можно найти на странице `files.htm` сайта книги. В него включен также и класс `BasicDaoFactory.vb`, требующий от клиента указания типа источника данных запрашиваемого объекта. Этот подход не рекомендуется использовать по той же причине, что и модель DAO. Он включен в проект только для наглядности.

Поставщики данных



Программный код, использованный в следующем примере, можно найти на странице `files.htm` сайта книги.

Возникает вопрос: что нужно делать, если возникает потребность в гибкой конструкции, допускающей обновления и даже расширения режимов работы программы без перекомпиляции? Что делать, если источником данных клиента является не традиционная база данных, а XML, или когда вычисленные значения должны быть полностью обособлены? Что делать, если доступ к данным должен быть полностью прозрачным для клиентского кода? Модель использования поставщиков данных полностью решает эти и многие другие вопросы, в том числе выделение логики источника данных из модели фабрики и способность динамически корректировать режим работы уровня данных во время выполнения клиентской программы.



Термин *поставщик данных* также используется компанией Microsoft в среде ADO.NET. В данном случае он обозначает группу классов, осуществляющих доступ к конкретному источнику данных. В настоящем разделе под поставщиком данных понимается отдельный класс, реализующий прозрачный доступ к информации, обычно хранимой в базе данных.



Обычно файлы конфигурации загружаются только один раз, либо при запуске приложения, либо в момент первого доступа к данным. Однако иногда возникает необходимость коррекции режима работы приложения уже в процессе его работы, без перезапуска. В такой ситуации проблему может решить загрузчик параметров, который осуществляет мониторинг файла конфигурации. Класс `PropertyLoader.vb` может стать отличной стартовой точкой, и его можно найти на сайте книги. Для примера рассмотрим Web-приложение, работающее под управлением службы IIS, режим работы которого необходимо динамически корректировать. Данное приложение работает в среде высокой доступности, и его перезапуск для перезагрузки файла конфигурации недопустимо. Реализуя загрузчик параметров, регистрирующий изменения файла конфигурации и выполняющий в нужный момент его перезагрузку, полностью отпадает необходимость в использовании перезагрузки приложения. Все измененные значения заново загружаются и становятся доступными приложению в процессе его выполнения.

Поставщик данных представляет собой задействуемый клиентом класс, который абстрагирует реализацию источника данных. Поставщики данных в своей реализации используют шаблон моста и полиморфизм. На рис. 54.3 показана модель поставщика данных с точки зрения абстрактного класса. На рис. 54.4 приведен контекстный пример использования поставщика данных `Address` (`DpAddress`) и необходимых классов поддержки.



Шаблон моста отделяет интерфейс от изменяющейся реализации. На практике это означает, что задействуемый клиентом класс не изменяет свой интерфейс. Вместо этого реализация, стоящая за интерфейсом, варьируется соответствующим образом для поддержки конкретного источника данных. Это также является примером полиморфизма, так как множество классов поддержки обеспечивают различную реализацию при поддержке одного и того же интерфейса.

Поставщики данных

Конкретный класс `DpDeterminer`, SqlOrAccessDeterminer, реализует бизнес-логику для определения того, с какой базой данных следует взаимодействовать. Так как эта логика инкапсулирована с параметрами конфигурации, ее идентифицирующими, ее можно вставить

в любой класс, наследуемый от `DpDeterminer` и изменяющий бизнес-логику без перекомпиляции.

Абстрактный базовый класс `DataProvider` представляет собой ядро динамической идентификации и создания специфичной для источника данных реализации.

Абстрактный базовый класс `DpImplementation` реализует первый уровень иерархии реализации. Он содержит некоторые общие переменные и методы. От этого класса наследуются конкретные реализации.

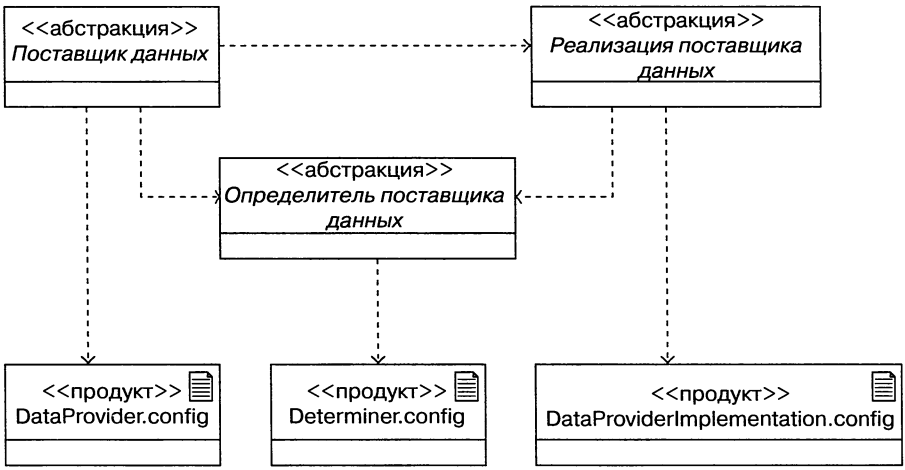


Рис. 54.3. Этот пример поставщика данных демонстрирует базовые классы и требуемые файлы конфигурации

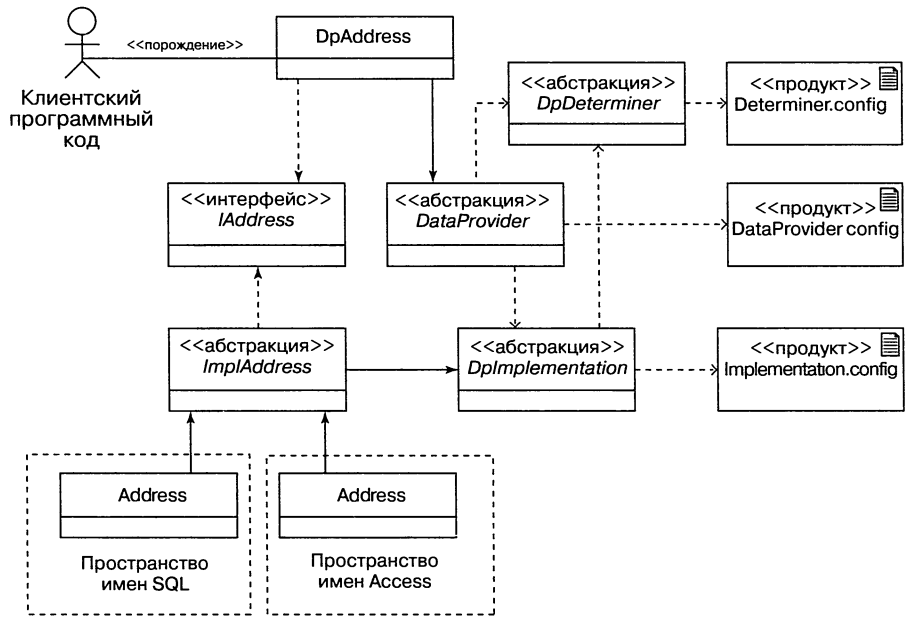


Рис. 54.4. В примере поставщика данных `Address` клиентский код ссылается только на класс `DpAddress`

Как работает поставщик данных

Поставщик данных формирует запрос к ассоциированному определителю и динамически создает в процессе работы специфичную для источника данных реализацию. Различные файлы конфигурации (см. рис. 54.3) обеспечивают гибкость поставщика и позволяют во время работы изменять информацию, необходимую для создания экземпляра класса, определяющего конкретную реализацию.

Класс `Determiner` играет критичную роль в модели поставщика данных, так как он отделяет бизнес-логику, определяющую, какой источник данных должен поддерживаться и как к нему подключаться. Это является основным отличием от моделей фабрики и DAO. Определитель существует сам по себе и может заменяться другим по мере необходимости. Схема, показанная на рис. 54.5, иллюстрирует фактический поток задач. Обратите внимание на то, что объекты `SqlOrAccessDeterminer` и `Address` создаются на основе базового класса `DataProvider`. Информация, необходимая этим объектам, содержится в файлах конфигурации. Также заметим, что при вызове функции `Load(id)` специфичный для базы данных объект `Address` запрашивает информацию о подключении к базе данных у объекта `SqlOrAccessDeterminer`. И, что более важно, клиентский код взаимодействует только с объектом `DpAddress` для загрузки и доступа к свойствам адреса. Таким образом, с точки

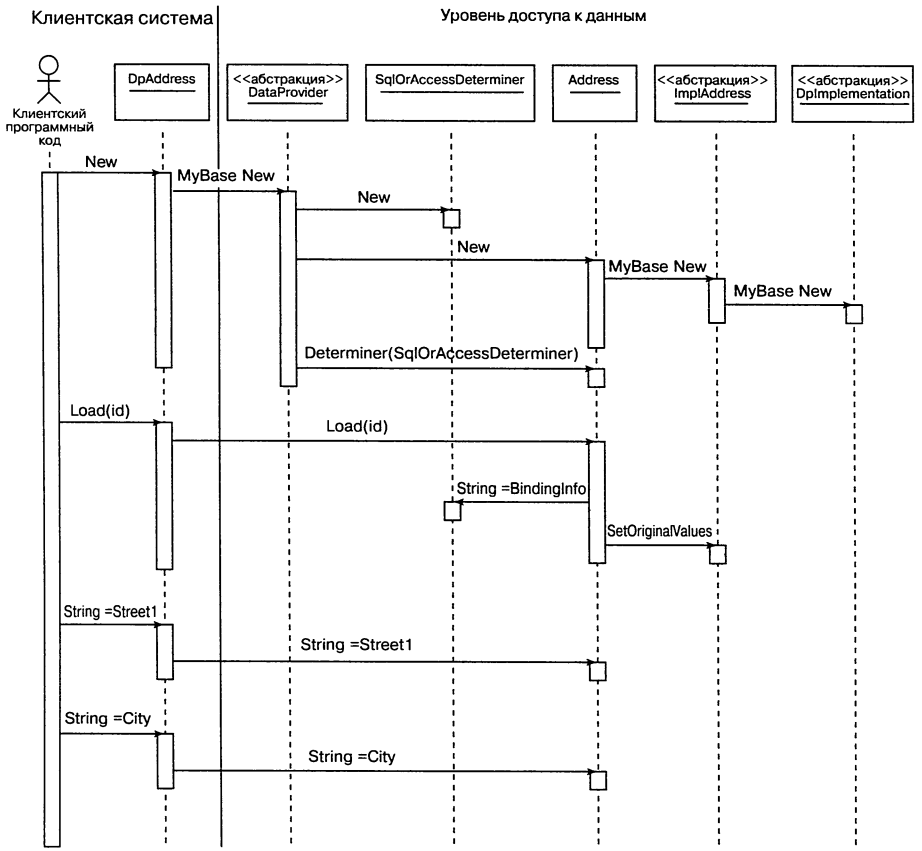


Рис. 54.5. Эта схема иллюстрирует создание поставщика данных Address, за чем следует загрузка соответствующих данных адреса и доступ к конкретным полям

зрения пользователя, DpAddress является не более чем обычным классом — с методами и свойствами. Сам доступ к базе данных при этом выполняется "за кулисами".

В следующем программном коде и приведенном за ним примере реализации рассмотрим базовый класс DpDeterminer.

Базовый класс определителя

```
Public MustInherit Class DpDeterminer
Private Const BINDING_INFO_TAG As String = "_BindingInfo"
Private Const DESCRIPTION_TAG As String = "_Description"
'''-----------------------------------------------------
''' Конструктор.
'''-----------------------------------------------------
Public Sub New()
' Часть инициализации опущена. Важен конфигурационный файл,
' загружаемый для доступа к свойству...

' Конфигурационный файл существует -- загружаем его.
PropertyLoader.Instance.LoadProperties(strPropertyFile)
End Sub
'''-----------------------------------------------------
''' Информация о подключении и связывании для
''' текущего источника данных.
'''-----------------------------------------------------
Public ReadOnly Property BindingInfo() As String
Get
Return PropertyLoader.Instance.GetProperty( _
Me.GetType.Name & "_" & Me.DataSource &
BINDING_INFO_TAG)
End Get
End Property
-------------------------------------------------------
''' Источник данных для этого определителя.
'''-------------------------------------------------------
Public MustOverride ReadOnly Property DataSource() As String
'''-------------------------------------------------------
''' Описание текущего источника данных.
'''-------------------------------------------------------
Public ReadOnly Property Description() As String
Get
Return PropertyLoader.Instance.GetProperty( _
Me.GetType.Name & "_" & Me.DataSource &
DESCRIPTION_TAG)
End Get
End Property
```

Основная часть этого базового класса возвращает различные значения файла конфигурации в зависимости от типа определителя. Наибольший интерес представляет свойство MustOverride DataSource(). Этот метод, определенный в классе-наследнике, содержит бизнес-логику, определяющую тип источника данных, к которому осуществляется доступ. Рассмотрим пример:

SqlOrAccessDeterminer.vb

```
Public Class SqlOrAccessDeterminer
  Inherits DpDeterminer
'''-------------------------------------------------------
```

```
''' Внутренняя бизнес-логика, определяющая, какой тип
''' источника данных будет использоваться.
'''-------------------------------------------------------
Public Overrides ReadOnly Property DataSource() As String
Get
  With PropertyLoader.Instance
If CBool(.GetProperty(Me.GetType.Name & "_UseSQL")) Then
  Return .GetProperty(Me.GetType.Name & "_SqlTag")
Else
  Return .GetProperty(Me.GetType.Name & _
      "_AccessTag")
End If
End With
End Get
End Property
End Class
```

Просматривая бизнес-логику, заложенную в классе `SqlOrAccessDetermoner`, обратите внимание на то, что если `"_UseSQL"` имеет значение `True`, то определитель возвращает строку `"SqlTag"`, указывающую на использование базы данных SQL Server. В противном случае возвращается строка `"AccessTag"`. Ключевая концепция заключается в том, что вся бизнес-логика была полностью отделена от создания объекта данных и теперь существует сама по себе. Подобная слабая связность повышает гибкость модели.



Именами файлов конфигурации, в соответствии с соглашением корпорации Microsoft, являются *<имя_сборки>*`.dll.config`. Поэтому при просмотре программного кода имейте в виду, что файлы конфигурации должны носить именно такие имена, например `Providers.dll.config` или `ProviderImplementations.dll.config`.

Файл конфигурации, на который ссылается `SqlOrAccessDeterminer`, выглядит следующим образом:

ProviderDeterminers.dll.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
<!-- Настройки определителя. -->
<add key="SqlOrAccessDeterminer_UseSQL" value="True" />
<!-- The 'tag' values below are included in building a -->
<!-- class instance namespace. i.e -->
<!-- Providers.Implementations.Sql.Address or -->
<!-- Providers.Implementations.Access.Address -->
<add key="SqlOrAccessDeterminer_SqlTag" value="Sql" />
<add key="SqlOrAccessDeterminer_AccessTag" value="Access" />
<!-- Остальные настройки опущены... -->
</appSettings>
</configuration>
```

При изменении значения `True` ключа `SqlOrAccessDeterminer_UseSQL` на `False` изменяется база данных и все поставщики данных, использующие данный определитель, без необходимости перекомпиляции приложения. Это является существенным преимуществом, особенно в среде, в которой простои недопустимы.



Когда возникает необходимость создания загрузчика параметров для управления настройками приложения, включайте в него функции мониторинга изменений файла конфигурации. Это позволит автоматически перезагружать любые обновления и использовать их в приложении без необходимости перезагрузки. Среда .NET предлагает поддержку мониторинга файлов с помощью класса `FileSystemWatcher`. Используя этот класс, можно определить, какие файловые события следует обрабатывать и какие методы для этого использовать.

Ниже приведен пример файла конфигурации, в котором продемонстрировано, почему поставщикам данных так важен внутренний взгляд на них и почему они имеют такое существенное влияние на доступ к данным.

Файл конфигурации `Providers.dll.config`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
<!-- Настройки реализации поставщика данных. -->
<!-- Поставщик данных Address -->
<add key="DpAddress_Determiner"
  value="Providers.Determiners.SqlOrAccessDeterminer" />
<add key="DpAddress_Determiner_Assembly"
  Value="ProviderDeterminers" />
<add key="DpAddress_Instance" value="Address" />
<add key="DpAddress_Instance_Assembly"
  value="ProviderImplementations" />
<add key="DpAddress_Instance_RootNameSpace"
  value="Providers.Implementations" />
</appSettings>
</configuration>
```

Строка `DpAddress_Determiner` указывает, какой определитель будет использовать данный поставщик данных при создании реализации источника данных. По этой причине любые изменения, которые влияют на логику определителя, будут воздействовать на все поставщики данных, зависящие от него. Также в раздел конфигурации включены значения, определяющие, в каких сборках следует искать определитель, а также значения, необходимые для создания экземпляра поставщика данных методом отражения. Теперь, когда вы увидели, как конфигурируется поставщик данных, взгляните на программный код. Начнем мы с базового класса `DataProvider`.

Класс `DataProvider.vb`

```
Public MustInherit Class DataProvider
' Константы и локальные определения опущены
'''-----------------------------------------------------------
''' Конструктор.
'''-----------------------------------------------------------
Public Sub New ()
' Код для загрузки файла конфигурации...
' Создание определителя для данного поставщика данных
' на основе данных файла конфигурации.
With PropertyLoader.Instance
Dim strAssembly As String = .GetProperty( _ Me.GetType.Name &
DETERMINER_ASSEMBLY_TAG)
Dim strClass As String = .GetProperty(Me.GetType.Name _ & DETERMINER_TAG)
' Создание определителя для поставщика данных.
Determiner = CType(Activate(strAssembly, strClass), _ DpDeterminer)
```

```
' Используя новый определитель, получаем реализацию.
strAssembly = .GetProperty(Me.GetType.Name & _ INSTANCE_ASSEMBLY_TAG)
strClass = .GetProperty(Me.GetType.Name & _ INSTANCE_ROOT_NAME_SPACE) &
"." & _ Determiner.DataSource & "." & _ .GetProperty(Me.GetType.Name &
INSTANCE_TAG)
' Установка реализации для поставщика данных.
Implementation = CType(Activate(strAssembly,strClass),_ DpImplementation)
Implementation.Determiner = Determiner
End With
End Sub
'''-----------------------------------------------------------
''' Создание экземпляра затребованного класса
''' из заданной сборки.
'''-----------------------------------------------------------
Private Function Activate(ByVal strAssembly As String, _
  ByVal strClassName As String) As Object
' код проверки...
' создание экземпляра с помощью отражения и его возвращение.
Dim instanceType As Type = _
  System.Reflection.Assembly.Load(strAssembly).GetType( _      strClassName)
Return Activator.CreateInstance(instanceType)
End Function
' Остальные общие методы поддержки опущены.
' Найти их можно на сайте книги.
End Class
```

Конструктор пытается загрузить файл конфигурации, а затем начинает динамически создавать экземпляры класса. Вначале создается определитель, связанный с поставщиком данных (определенный в файле конфигурации). Затем создается конкретная реализация источника данных (Access, SQL и т.п.), идентифицированная данным определителем. Код автоматически выполняется, когда поставщик данных идентифицируется. На этапе проектирования это освобождает разработчика от создания классов поддержки, и он может сосредоточить свое внимание на деталях поставщика данных. Поставщик данных `Address` приведен ниже.

Поставщик данных `DpAddress.vb`

```
Public Class DpAddress
  Inherits DataProvider
  Implements Iaddress
' Этот класс только представляет реализацию свойств,
' определенных в интерфейсе IAddress. Помните, что этот
' интерфейс также реализован в классах, специфичных
' для источника данных.
'''-----------------------------------------------------------
''' Вспомогательный метод преобразует реализацию в
''' подходящий тип. Implementation -- это свойство,
''' определенное в базовом классе DataProvider.
'''-----------------------------------------------------------
Protected Function MyImplementation() As Iaddress
Return CType(Implementation, IAddress)
End Function
' Вот пример свойства.
'''-----------------------------------------------------------
''' Значение City.
'''-----------------------------------------------------------
Public Property City() As String Implements   Interfaces.IAddress.City
Get
  Return MyImplementation.City
```

```
End Get
Set(ByVal Value As String)
  MyImplementation.City = Value
End Set
End Property
' Все остальные свойства интерфейса представляются аналогичным
' образом.
End Class
```

Являясь наследником класса `DataProvider`, класс `DpAddress` принимает всю функциональность создания. Реализуя интерфейс `IAddress`, класс `DpAddress` соглашается с общим контрактом, определяющим его возможности. Единственным дополнительным методом этого класса является `MyImplementation()`, и он был введен для удобства разработчика и активизации функции `IntelliSense` программы Visual Studio. Теперь рассмотрим средства, обеспечивающие реализацию источника данных. Перед рассмотрением специфических особенностей реализации взглянем на базовый класс `DpImplementation.vb`.

Базовый класс `DpImplementation.vb`

```
'''-------------------------------------------------------
''' Определитель
'''-------------------------------------------------------
Public Property Determiner() As DpDeterminer
' Реализация крайне проста, поэтому она опускается.
End Property
'''-------------------------------------------------------
''' Конструктор.
'''-------------------------------------------------------
Public Sub New()
' Загрузка соответствующего файла конфигурации.
Dim strFileName As String
strFileName = My.Application.CurrentDirectory & "\" & _
Me.GetType.Module.ScopeName & ".config"
' Загрузка файла свойств.
PropertyLoader.Instance.LoadProperties(strFileName)
End Sub
End Class
```

Этот класс реализации содержит свойство `Determiner`, локальное хранилище и загружает соответствующий файл конфигурации. Фактическая работа по созданию, чтению, обновлению и удалению содержится в специфичной для источника данных реализации.



В языке VB.NET 2005 было введено ключевое слово `My`, обеспечивающее удобный доступ к общим значениям запущенной программы и платформы, в которой она установлена. Благодаря использованию ключевых слов `My`, `Application`, `User` и `WebServices` доступ к значениям становится намного легче.



В предыдущих версиях VB.NET, когда разработчику нужно было определить текущий каталог, использовался метод `AppDomain.CurrentDomain.BaseDirectory`. Этот метод возвращал разделитель `Path.DirectorySeparatorChar()`, добавленный в конец строки. Новое свойство `My.Application.CurrentDirectory()` возвращает текущий каталог без добавления разделителя. Это позволяет сократить программный код, который ранее предполагал существование символа каталога в конце строки.

Ниже приведен базовый класс реализации, служащий основой всех классов, специфичных для баз данных.

Базовый класс ImplAddress.vb

```
Public MustInherit Class ImplAddress
  Inherits DpImplementation
  Implements Iaddress
  Private m_CurrentValues As doAddress
  Private m_OriginalValues As doAddress
' Этот класс реализует интерфейс Iaddress,
' общий для всех реализаций, специфичных для
' источников данных.
' Отдельно от объявлений объектов данных, приведенных выше,
' в данном классе нет ничего особенного. При желании вы можете
' просмотреть полный программный код на сайте книги.
'''-------------------------------------------------------------
''' Загрузка определения интерфейса. Она должна применяться в
''' классах-наследниках.
'''-------------------------------------------------------------
Public MustOverride Sub Load(ByVal strId As String)
End Class
```

Класс `ImplAddress` обеспечивает общую функциональность всех реализаций, оставляя специфичные для источников данных функции конкретным реализациям. Все это можно увидеть в объявлении метода `Load(...)`. Также отметьте для себя, что метод `Load(...)` был объявлен с атрибутом `MustOverride`. Таким образом, все дочерние классы должны реализовать собственную интерпретацию этого метода. Еще следует обратить внимание на объект данных `doAddres`. Этот объект хранит состояние класса `Address`, обращаясь к нему как к *напоминанию* (memento).



Поддержка *напоминаний* обеспечивает захват и выделение всех внутренних состояний экземпляра. Такой режим работы обеспечивает массу преимуществ, включая сериализацию значений, отслеживание состояния объектов, отмену и повторное выполнение функций, простую проверку на равенство и т.д.



Выделяйте состояние поставщиков данных, используя объекты напоминаний, которые можно использовать для сохранения и загрузки состояния объекта, тестирования равенства и текущих изменений значений, в частности функциональности `IsDirty()`.



Абстрактный класс `DataObject.vb` обеспечивает функциональность родительских объектов при создании объектов данных `memento`. Этот абстрактный класс содержит методы отражения, служащие для клонирования объектов, переустановки их значений и их сравнения. Полный программный код этого базового класса приведен на сайте книги.

Рассмотрим специфичный для источника данных класс `Address`.

Address.vb — реализация SQL Server

```
'''-------------------------------------------------------------
''' Это реализация SQL Server.
'''-------------------------------------------------------------
```

```
Public Class Address
  Inherits ImplAddress
'''-------------------------------------------------------------
''' Фактическая функция загрузки реализации SQL Server
'''-------------------------------------------------------------
  Public Overrides Sub Load(ByVal strId As String)
  Dim cmd As SqlCommand = Nothing
  Dim conn As SqlConnection = Nothing
  Dim dataReader As SqlDataReader = Nothing
  Try
    ' установка подключения к базе данных с помощью
    ' информации определителя.
    conn = New SqlConnection(Me.Determiner.BindingInfo)
    conn.Open()
    ' Настройка команды.
    cmd = New SqlCommand( PropertyLoader.Instance.GetProperty( _
    "Sql_Address_Select").Replace("%1", _ strId.ToString), conn)
    dataReader = cmd.ExecuteReader( _
         CommandBehavior.CloseConnection)
    If dataReader.Read Then
       Me.Street1 = MySqlHelper.GetStringValue( _
         dataReader,   "Street1", "<no value>")
       Me.Street2 = MySqlHelper.GetStringValue( _
         dataReader, "Street2", "<no value>")
' Аналогичный программный код используется для инициализации
' города, штата, индекса и т.д.
    End If
      Me.SetOriginalValues()
      Catch ex As Exception
' Здесь выполняется обработка.
      Throw(ex)
   Finally
' Очистка
   If cmd IsNot Nothing Then cmd.Dispose
   If conn IsNot Nothing Then conn.Close()
   If dataReader IsNot Nothing Then dataReader.Close()
  End Try
End Sub
End Class
```

Поскольку класс `Address` является дочерним для класса `ImplAddress`, он наследует полный интерфейс и не обязан беспокоиться об определении методов установки и проверки свойств отдельных полей. Вместо этого данный класс фокусирует свое внимание на специфике взаимодействия с базой данных SQL Server и наполнении соответствующего объекта данных. То же справедливо и для класса `Address`, предназначенного для доступа к базам данным Access.



При определении общих контрактов (интерфейсов программирования и методов) поставщиков данных используйте интерфейсы. Это позволяет обеспечить согласованность используемых клиентом объектов данных и специфичных для конкретных баз данных реализаций этих объектов. Это также позволяет клиентам создавать собственные классы, реализующие определенные интерфейсы, и вставлять их в приложения.

Клиентский программный код в следующем примере использует именно такой шаблон. Сравните его с программным кодом фабрики и DAO, представленными ранее.

Поставщик данных `MainTester`

```
Public Class MainTester
'''-------------------------------------------------------
''' Этот метод тестирует поставщика данных.
'''-------------------------------------------------------
Sub Main()
  Try
    Dim objAddr As New Address
    objAddr.Load("1")
    Call PrintAddress(objAddr)
  Catch ex As Exception
    Console.WriteLine(ex.Message)
  End Try
End Sub
End Class
```

Этот клиентский тестовый программный код даже меньше по размеру, чем используемый для работы с фабрикой, к тому же в данном случае нет никакой необходимости в преобразовании объекта. На самом деле данному клиентскому коду даже не нужно знать, что для доступа к источникам данных используются разные классы. Если в решении имеет значение компактность клиентского программного кода, то следует использовать подход с применением поставщика данных. Если сравнить его с подходом DAO, то в данном клиентском коде существуют три строки вместо семи, к тому же в нем более интуитивно понятна работа с объектом данных `Address`, поскольку в нем нет отличий от стандартного кода, в частности, используется команда `Dim strSomeString As String`.

Достоинства шаблона поставщика данных

Шаблон поставщика данных может быть чрезвычайно гибким, позволяя клиенту или разработчику быстро замещать классы доступа к данным и указывать, какие поставщики данных используются и к каким источникам данных можно подключаться. Так как реализации поставщика данных могут быть выведены из приложения, становится довольно просто вносить изменения в файл конфигурации, поскольку при развертывании новой сборки нет необходимости вносить изменения в файл конфигурации. К тому же способность изменения режима работы программы после развертывания является дополнительным достоинством приложения как с точки зрения пользователя, так и производителя программы.

Недостатки шаблона поставщика данных

При использовании шаблона поставщика данных требуется более внимательно устанавливать внешние значения, содержащиеся в конфигурационных файлах, создавать пространства имен для классов и организовывать программный код. Кроме того, эта модель более сложная, чем две остальные, о которых мы говорили ранее. Она требует большего времени обучения и больших теоретических знаний в области объектно-ориентированных технологий. Эта дополнительная сложность может подойти не всем проектам и командам программистов.



Полный проект `Data Providers` вы можете найти на странице `files.htm` сайта книги. В него включены все шаблоны, которые могут стать отправной точкой в создании поставщиков данных.

Резюме

Если вы определили, что в приложении требуется создать уровень доступа к данным, то должны задать себе ряд вопросов относительно модели, которую хотите реализовать для доступа пользователей к данным, и решить, какой шаблон больше всего соответствует вашим потребностям. Выбор наиболее подходящего шаблона может ускорить процесс программирования и повысить расширяемость на этапе разработки, одновременно снижая затраты на сопровождение уровня доступа к данным.

Выделяя запросы SQL, зависимости базы данных и бизнес-логику в файлы конфигурации, как это реализовано в шаблоне поставщика данных, вы обеспечиваете гибкость модернизации приложений на этапе их эксплуатации. Вы можете даже поддерживать новые базы данных, добавляя новые элементы в файлы конфигурации и создавая специфичные для баз данных сборки .NET без перекомпиляции приложений и их перезапуска. Эта возможность динамической корректировки уровня доступа к данным без перекомпиляции и перезагрузки способна обеспечить минимально возможное время отсутствия доступа к данным, т.е. их повышенную доступность.

В этой главе были описаны всего три шаблона доступа к данным, в то время как существует множество других. Не бойтесь опробовать другие решения обеспечения доступа к данным и основательно подходите к вопросам их выбора.

Имея гибкий, хорошо проработанный уровень доступа к данным, вы обеспечите себе множество преимуществ на этапе проектирования приложений, их реализации и развертывания.

Приложения

ЧАСТЬ



В данной части приведена информация, которую можно применить к каждой главе этой книги, в частности к спецификациям SQL Server и деталям использования учебных баз данных. Также не забывайте просматривать Web-сайт книги по адресу `www.SQLServerBible.com`.

В этой части...

Приложение А
Спецификации SQL Server 2005

Приложение Б
Учебные базы данных

ПРИЛОЖЕНИЕ A

Спецификации SQL Server 2005

В этом приложении...

Спецификации SQL Server

Сравнение спецификаций SQL Server 2005 и SQL Server 6.5 – 2000

В табл. A.1 приведены различные спецификации SQL Server, а в табл. A.2 перечислены средства редактирования.

Таблица A.1. Спецификации SQL Server

Функция	SQL Server 6.5	SQL Server 7.0	SQL Server 2000	SQL Server 2005
Функции сервера				
Автоматическое конфигурирование	Нет	Да	Да	Да
Размер страниц	2 Кбайт	8 Кбайт	8 Кбайт	8 Кбайт и более
Максимальный размер строки	1962 байт	8060 байт	8060 байт	8060 байт
Блокировка на уровне страниц	Да	Да	Да	Да
Блокировка на уровне строк	Только при вставке	Да	Да	Да
Обнаружение файлов	Устройства	Файлы и файловые группы	Файлы и файловые группы	Файлы и файловые группы
Kerberos и делегирование защиты	Нет	Нет	Да	Да
Сертификация системы безопасности C2	Нет	Нет	Да	Да
Количество байтов символов на столбец	255	8000	8000	8000
Автоматическая доставка журнала	Нет	Нет	Да	Да
Вычисляемые по индексу столбцы	Нет	Нет	Да	Да
Максимальный размер пакета	128 Кбайт	65536 байт × размер сетевого пакета	65536 байт × размер сетевого пакета	65536 байт × размер сетевого пакета
Количество байтов в тексте/изображении	2 Гбайт	2 Гбайт	2 Гбайт	2 Гбайт
Количество объектов в базе данных	2 миллиарда	2147483647	2147483647	2147483647
Количество параметров в хранимой процедуре	255	1024	1024	1024
Количество ссылок в таблице	31	253	253	253
Количество строк в таблице	Ограничено доступным дисковым объемом	Ограничено доступным дисковым объемом	Ограничено доступным дисковым объемом	Ограничено доступным дисковым объемом
Количество таблиц в базе данных	2 миллиарда	Ограничено доступным дисковым объемом	Ограничено доступным дисковым объемом	Ограничено доступным дисковым объемом
Количество таблиц в инструкции `SELECT`	16	256	256	256
Количество триггеров в таблице	3	Ограничено доступным дисковым объемом	Ограничено доступным дисковым объемом	Ограничено доступным дисковым объемом
Количество байтов в ключе (первичном, внешнем или индексном)	900	900	900	900
Количество байтов в предложении `GROUP BY` или `ORDER BY`	900	8060	8060	8060

Функция	SQL Server 6.5	SQL Server 7.0	SQL Server 2000	SQL Server 2005
Количество байтов в строке	900	8060	8060	8060
Количество байтов программного текста в хранимой процедуре	65025	Размер пакета или 250 Мбайт (выбирается меньшее)	Размер пакета или 250 Мбайт (выбирается меньшее)	Размер пакета или 250 Мбайт (выбирается меньшее)
Количество столбцов в ключе (первичных внешних или индексных)	16	16	16	16
Количество столбцов в предложениях `GROUP BY` или `ORDER BY`	16	Ограничено количеством байтов	Не определено	Не определено
Количество столбцов в таблице	255	1024	1024	1024
Количество столбцов в инструкции `SELECT`	4096	4096	4096	4096
Количество столбцов в инструкции `INSERT`	250	1024	1024	1024
Размер базы данных	1 Тбайт	1048516 Тбайт	1048516 Тбайт	1048516 Тбайт
Количество баз данных на сервере	32767	32767	32767 (в одном экземпляре)	32767 (в одном экземпляре)
Количество файловых групп в базе данных	–	256	256	256
Количество файлов в базе данных	32	32767	32767	32767
Размер файла данных	32 Гбайт	32 Тбайт	32 Тбайт	16 Тбайт
Размер файла журнала	32 Гбайт	4 Тбайт	32 Тбайт	2 Тбайт
Количество внешних ключей в таблице	16	253	253	Не ограничено
Длина идентификатора (таблиц, названий столбцов и т.п.)	30	128	128	128
Индексы XML	–	–	–	249
Количество экземпляров на компьютере	1	1	16	16
Количество блокировок в экземпляре	2147483647	2147483647 или 40% памяти SQL Server	2147483647 или 40% памяти SQL Server	2147483647 или 40% памяти SQL Server (в 64-разрядной версии ограничено только объемом памяти)
Параллельное выполнение запросов	Нет	Да	Да	Да
Поддержка федеративных баз данных	Нет	Нет	Да	Да
Количество индексов в таблице, используемое при выполнении запросов	1	Множество	Множество	Множество

Функция	SQL Server 6.5	SQL Server 7.0	SQL Server 2000	SQL Server 2005
Административные функции				
Автоматическое увеличение размеров файлов данных и журнала	Нет	Да	Да	Да
Автоматическая статистика индексов	Нет	Да	Да	Да
Связь Profiler с событиями оптимизатора	Нет	Да	Да	Да
Предупреждения по условиям производительности	Нет	Да	Да	Да
Условные многошаговые задания агента	Нет	Да	Да	Да
Функции программирования				
Рекурсивные триггеры	Нет	Да	Да	Да
Множественные триггеры в расчете на события таблицы	Нет	Да	Да	Да
Триггеры `INSTEAD OF`	Нет	Нет	Да	Да
Поддержка символов `UNICODE`	Нет	Да	Да	Да
Пользовательские функции	Нет	Нет	Да	Да
Индексированные представления	Нет	Нет	Да	Да
Каскадные удаления и обновления DRI	Нет	Нет	Да	Да
Уровень сличения	Сервер	Сервер	Сервер, база данных, таблица, запрос	Сервер, база данных, таблица, запрос
Количество вложенных уровней хранимых процедур	16	32	32	32
Количество вложенных подзапросов	16	32	32	32
Количество уровней вложенных триггеров	16	32	32	32
Поддержка XML	Нет	Нет	Да	Да
Функции репликации				
Репликация мгновенных снимков	Да	Да	Да	Да
Репликация транзакций	Да	Да	Да	Да
Репликация слияния с разрешением конфликтов	Нет	Да	Да	Да
Функции Enterprise Manager и Management Studio				
Диаграмма базы данных	Нет	Да	Да	Да
Создание таблиц в графическом интерфейсе	Да	Да	Да	Да

Функция	SQL Server 6.5	SQL Server 7.0	SQL Server 2000	SQL Server 2005
Конструктор баз данных	Нет	Да	Да	Да
Конструктор запросов	Нет	Да	Да	Да

Таблица А.2. Функции редактирования

Функция	Mobile	Express	Workgroup	Standard	Enterprise
Целевые приложения	Устройства Windows Mobile Smart Client	Внедренные в приложения	Небольшие рабочие группы	Подразделения среднего размера, базы данных рабочих групп	Крупные корпоративные базы данных
Ограничения в масштабировании					
Максимальный размер базы данных	2 Гбайт	4 Гбайт	Нет ограничений	Нет ограничений	Нет ограничений
Количество поддерживаемых процессоров (может зависеть от версии Windows)	1	1	2	4	Нет ограничений
Поддерживаемая память (может зависеть от версии Windows)	1 Гбайт	1 Гбайт	3 Гбайт	Максимально поддерживаемый Windows объем	Максимально поддерживаемый Windows размер
Поддержка 64-разрядных процессоров	Нет данных	WOW	WOW	Да	Да
Функции ядра					
Множество экземпляров	Нет	Да	Да	Да	Да
Бессбойная кластеризация	Нет	Нет	Нет	Да	Да
Доставка журнала	Нет	Нет	Нет	Да	Да
Расширенный параллелизм	Нет	Нет	Нет	Нет	Да
Индексированные представления	Нет	Нет	Нет	Да	Да
Федеративные базы данных	Нет	Нет	Нет	Да	Да
Средства разработки					
Служба уведомлений	Нет	Нет	Нет	Да	Да
Брокер служб	Нет	Да, только для подписчиков	Да	Да	Да
Web-службы	Нет	Нет	Нет	Да	Да
Импорт/экспорт	Нет	Нет	Да	Да	Да

Функция	Mobile	Express	Workgroup	Standard	Enterprise
Репликация					
Репликация слияния	Да	Только для подписчиков	Да, до 25 подписчиков	Да	Да
Репликация транзакций		Только для подписчиков	Да, до 5 подписчиков	Да	Да
Репликация Oracle	Нет	Нет	Нет	Нет	Да (публикация Oracle)
Управляемость					
Profiler	Нет	Нет	Нет	Да	Да
Management Studio	?	Express Edition, только для загрузки	Да	Да	Да
Database Tuning Advisor				Да	Да
Полнотекстовый поиск		Нет	Да	Да	Да
SQL Agent		Нет	Да	Да	Да
Функции бизнес-аналитики					
Служба отчетности		Скоро будет	Да	Да	Да
Построитель отчетов			Да	Да	Да
Источники данных отчетов		Скоро будет, только локально	Только локально	Да	Да

ПРИЛОЖЕНИЕ Б

Учебные базы данных

В этом приложении...

Список файлов, требования, диаграммы и описания пяти учебных баз данных

В дополнение к предложенной компанией Microsoft учебной базе данных `Adventureworks` в этой книге были использованы примеры из пяти других баз данных. Каждая из них создавалась для иллюстрирования определенной концепции проектирования или стиля разработки.

- *Cape Hatteras Adventures.* Это фактически две базы данных, демонстрирующие переход к реляционной СУБД SQL Server. Первая версия состоит из простой базы данных Access и рабочего листа Excel. Ни одно из этих представлений данных нельзя назвать сложным. Вторая версия — это небольшого размера база данных SQL Server, содержащая столбцы идентичности и представления. Она использует проект Access в качестве клиента и публикует данные в Web, используя мастер публикаций SQL Server, а также хранимые процедуры.
- База данных *OBXKites* предназначена для отслеживания складских запасов, клиентов и продаж фиктивной торговой компании, занимающейся продажами воздушных змеев и аксессуаров к ним. Компания имеет четыре магазина, расположенных в Северной Каролине. Эта база данных спроектирована так, чтобы поддерживать устойчивое масштабирование. Она использует глобальные универсальные идентификаторы для репликаций и кодовую таблицу Unicode для поддержки международных контрактов. В различных главах этой книги в базу данных OBXKites добавлялись разделенные представления, полноценные средства аудита, а также кубы службы анализа.
- База данных *Family* хранит родословные. Несмотря на то что эта база состоит всего из двух таблиц — `person` и `marriage`, — она иллюстрирует сложности самообъединений "многие ко многим" и способна извлекать иерархические данные.

- Двадцать пять басен Эзопа, содержащихся в базе *Aesop's Fables*, позволили в главе 13 продемонстрировать технологию полнотекстового поиска.

В настоящем приложении документируются необходимые файлы (табл. Б.1) и описываются схемы учебных баз данных.



Все файлы, необходимые для создания учебных баз данных, а также программный код, приведенный в книге, можно загрузить с сайта `www.SQLServerBible.com`.

Таблица Б.1. Файлы учебных баз данных

Cape Hatteras Adventures версии 2

`C:\SQLServerBible\Sample Databases\CapeHatterasAdventures`	
`CHA2_Create.sql`	Сценарий, генерирующий базу данных Cape Hatteras Adventures версии 2, включая таблицы, ограничения, индексы, представления, хранимые процедуры и систему защиты
`CHA_Convert.sql`	Распределенные запросы, преобразующие данные Access и Excel в базу данных Cape Hatteras Adventures версии 2. Этот сценарий является отражением пакета DTS и предполагает нахождение исходных файлов Access и Excel в каталоге `C:\SQLServerBible`
`CHA1_Customers.mdb`	База данных Access со списком заказчиков, используемая в преобразовании в базу SQL Server. Данные из этого файла импортируются в базу данных `CHA1`
`CHA1_Schdule.xls`	Рабочий лист Excel с событиями, турами и экскурсоводами, используемый в преобразовании в базу данных SQL Server `CHA1`
`CHA2_Events.xml`	Пример файла XML
`CHA2_Events.dtd`	Пример файла определения типов данных XML
`CHA2.adp`	Пример клиентского приложения Access, работающего с базой данных `CHA2`

OBXKites

`C:\SQLServerBible\Sample Databases\OBXKites`	
`OBXKites_Create.sql`	Сценарий, генерирующий базу данных `OBXKites`, включая таблицы, представления, хранимые процедуры и функции
`OBXKites_Populate.sql`	Сценарий, наполняющий базу `OBXKites` данными с помощью вызова хранимых процедур
`OBXKites_Query.sql`	Набор запросов, тестирующих заполнение базы данных `OBXKites`

Family

`C:\SQLServerBible\Sample Databases\Family`	
`Family_Create.sql`	Сценарий, создающий таблицы и хранимые процедуры базы данных `Family`, а также заполняющий ее данными
`Family_Queries.sql`	Набор запросов к базе данных `Family`

Aesop's Fables	
`C:\SQLServerBible\Sample Databases\Aesop`	
`Aesop_Create.sql`	Сценарий, создающий таблицу `Fable` и заполняющий базу данных двадцатью пятью баснями Эзопа. Эта учебная база данных используется для полнотекстового поиска
`Aesop.adp`	Клиентское приложение Access, предназначенное для просмотра басен

Файлы учебных баз данных

Файлы учебных баз данных лучше установить в каталог `C:\SQLServerBible`. Все примеры Web-приложений запрограммированы с расчетом на определенную структуру каталогов. Пакеты DTS и распределенные запросы также предполагают нахождение файлов Access и Excel именно в этом каталоге.

Для создания любой из учебных баз данных нужно запустить сценарий `Create` в анализаторе запросов. Если база данных уже существует, то этот сценарий удалит ее. Этот сценарий облегчает перестройку базы данных, так что не бойтесь экспериментировать. Поскольку этот сценарий удаляет базу данных, во время его запуска не может существовать подключений. Managementy Studio часто сохраняет подключение, даже если выбирается другая база данных. Если вы столкнетесь с ошибкой, наиболее вероятно, что она будет связана с тем, что в анализаторе запросов или в Management Studio поддерживается открытое подключение

Cape Hatteras Adventures версии 2

Фиктивная компания Cape Hatteras Adventures (CHA) предлагает туристические услуги. Она находится на востоке Северной Каролины — в районе, известном своими просторными пустыми пляжами на побережье Атлантики.

Компания Cape Hatteras Adventures предлагает экзотические и иногда экстремальные туры. Эта туристическая компания обеспечивает доставку своих клиентов к месту назначения и обратно.

Персонал компании CHA состоит из увлеченных людей. Список заказчиков и предложений поддерживается в базе данных Access, состоящей из одной таблицы. Она используется в основном для отправки почтовых сообщений. Основная нагрузка ложится на рабочий лист Excel, на котором поддерживаются списки событий, туров и гидов в формате "плоского" файла. На той же странице во втором списке отслеживаются заказчики, принимающие участие в каждом из событий. Несмотря на то что данный рабочий лист не является нормализованной базой данных, он содержит всю информацию, необходимую для ведения бизнеса.

Приложение QuickBooks обслуживает всю финансовую деятельность компании, и руководство компании вполне удовлетворено таким положением дел. По этой причине нет никакой необходимости в совершенствовании финансовых приложений.

Требования приложения

Компания CHA выросла до таких размеров, когда возникла потребность в лучшем управлении расписанием. По этой причине компания заключила контракт на разработку и поддержку базы данных.

Составление расписаний и покупка туров осуществляется в центральном офисе компании, находящемся в Северной Каролине. Туры компании начинаются в многочисленных базовых лагерях, разбросанных по всему миру. Эти базовые лагеря, как правило, не оснащены компьютерами и иногда даже не электрифицированы. Гиды отправляются в базовые лагеря с распечатанным списком туристов. Если в будущем планируется обеспечить базовые лагеря доступом к базе данных через Интернет, то следует создать Web-страницу.

Каждый из базовых лагерей может отвечать за несколько туров. Все туры тщательно спланированы и рассматриваются как повторяющиеся события. Каждая экскурсия имеет одного ведущего гида, отвечающего за безопасность и удовлетворенность туристов. В случае необходимости в туре могут быть задействованы и другие гиды.

Так как в компании CHA работает множество гидов с разными навыками, база данных должна отслеживать, какие гиды могут проводить каждый из туров.

Модель базы данных

Модель базы данных использует типичные отношения "один ко многим" между типом заказчиков и их списком, а также между таблицами заказчиков и событий, гидов и туров, гидов и событий (рис. Б.1).

С точки зрения стиля разработки нет никакой потребности в поддержке нескольких баз данных. Таким образом, для упрощения модели в базе данных используются столбцы идентичности. Доступ к данным осуществляется с помощью представлений и прямых инструкций `SELECT`.

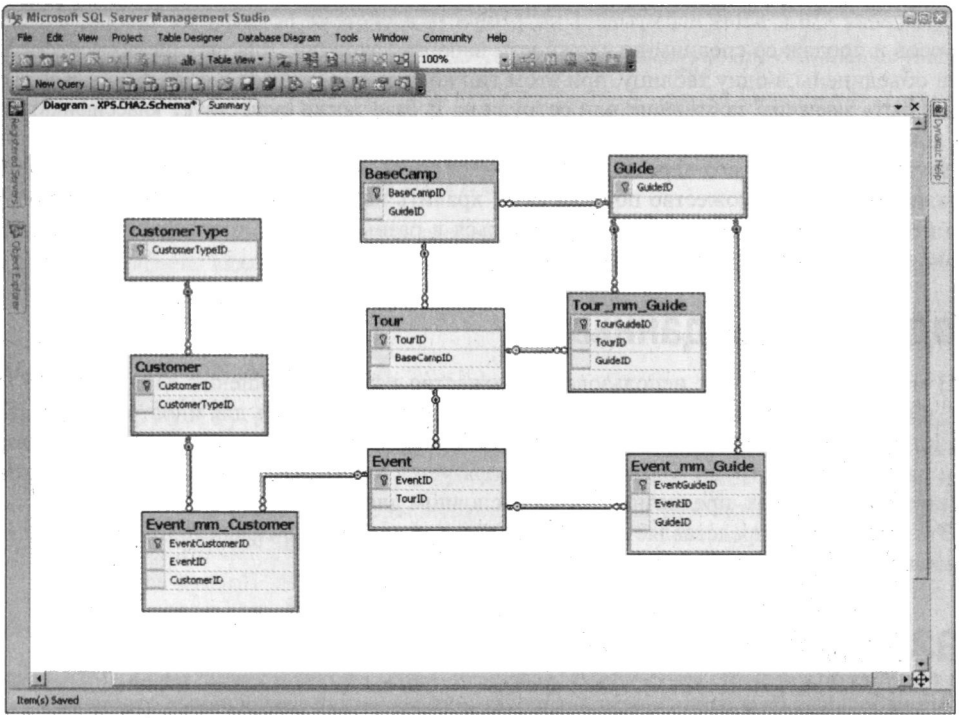


Рис. Б.1. Схема базы данных Cape Hatteras Adventures

Преобразование данных

Сценарий `CHA2_Create.sql` создает пустую базу данных. Сама информация находится в базе данных Access и на рабочем листе Excel. Как пакет DTS `CHA_Conversion`, так и сценарий `CHA_Convert.sql` извлекает данные из этих первоисточников и помещает их в базу данных SQL Server.

Клиентское приложение `CHA2.adp`

Поскольку персонал компании Cape Hatteras Adventures чувствует себя комфортно с формами Access и не нуждается в надежности приложений Visual Basic или .NET, клиентская часть была создана с помощью проекта Access.

OBX Kites

Компания OBX Kites занимается продажами воздушных змеев и аксессуаров к ним. Ее магазины находятся в районе Килл-Девил в Северной Каролине, известном своими постоянными ветрами. Именно в этом районе братья Райт совершили свой исторический полет. Компания OBX Kites имеет главный склад и четыре удаленные торговые точки и планирует открыть интерактивный Web-магазин.

Требования к приложению

Компания OBX Kites испытывает потребность в системе поддержки заказов, складских ресурсов и продаж со средним по сложности набором функций. Для упрощения все контакты были объединены в одну таблицу, при этом тип контакта выступает в роли флага. Контактом может быть заказчик, поставщик или сотрудник. В базе также существует классификатор типов контактов, используемый для вычисления скидок. Для каждого из заказчиков в таблице контактов содержатся его адреса. Система учета товаров на складе позволяет поддерживать для каждого товара множество поставщиков, хранить историю цен. Товар может включать в себя несколько компонентов, может находиться в разных местах; при этом складские транзакции отслеживают перемещения товаров.

Модель базы данных

В модели базы данных использовано множество разных отношений типа "один ко многим". Она должна поддерживать репликации и кодировку Unicode для иностранных заказчиков. Из соображений производительности и гибкости база данных реализована с использованием двух файловых групп. Одна из них используется для транзакций, а другая — для хранения статичных данных, предназначенных в основном для чтения.

Эта база данных представляет собой стандартное хранилище складской информации и отслеживания заказов (рис. Б.2).

База данных Family

Эта небольшая база данных демонстрирует иерархические рекурсивные отношения. Навигация в ней осуществляется с помощью как курсоров, так и запросов.

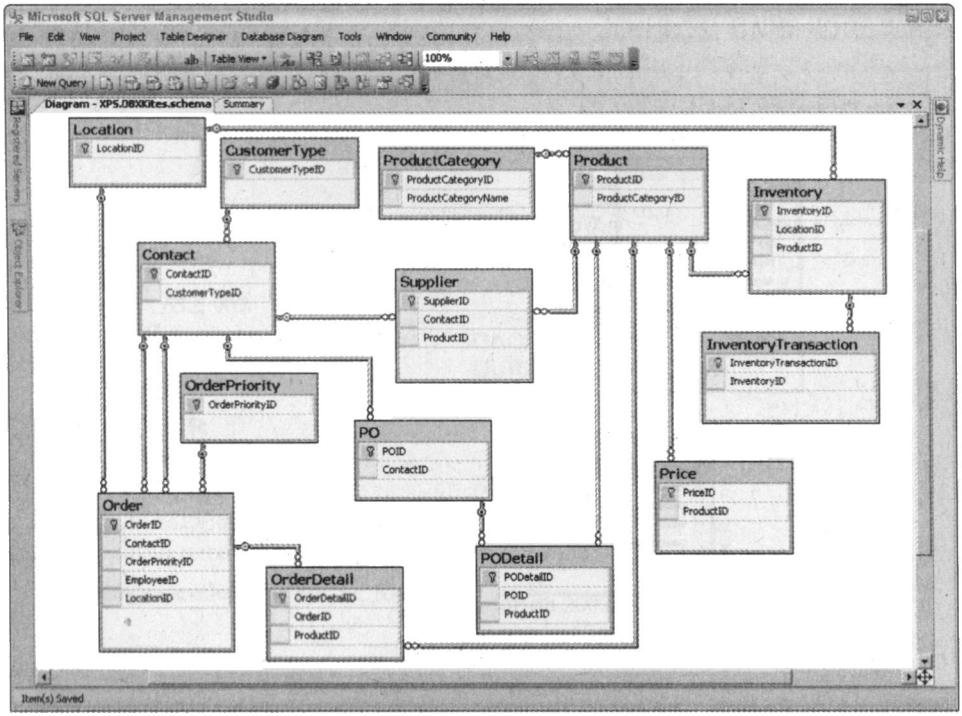


Рис. Б.2. Схема учебной базы OBXKites, отображенная в конструкторе баз данных утилиты SQL Server Management Studio

Требования к приложению

База данных `Family` должна хранить сведения обо всех членах семей наряду с генеалогической информацией, включая родственные и брачные отношения. Эта база наполнена информацией о пяти поколениях фиктивной семьи.

Модель базы данных

База данных `Family` состоит из двух таблиц и трех отношений. Ее схема показана на рис. Б.3. Каждый член семьи имеет необязательные внешние ключи идентификаторов матери и отца, ссылающиеся на идентификаторы других людей. Таблица `marriage` имеет внешний ключ, ссылающийся на идентификаторы жены или мужа. Для упрощения в роли первичного ключа использованы целые числа.

База данных Aesop's Fables

Коллекция басен Эзопа позволяет протестировать поиск строк, а также полнотекстовый поиск. Басни относительно короткие и расположены в общедоступном домене.

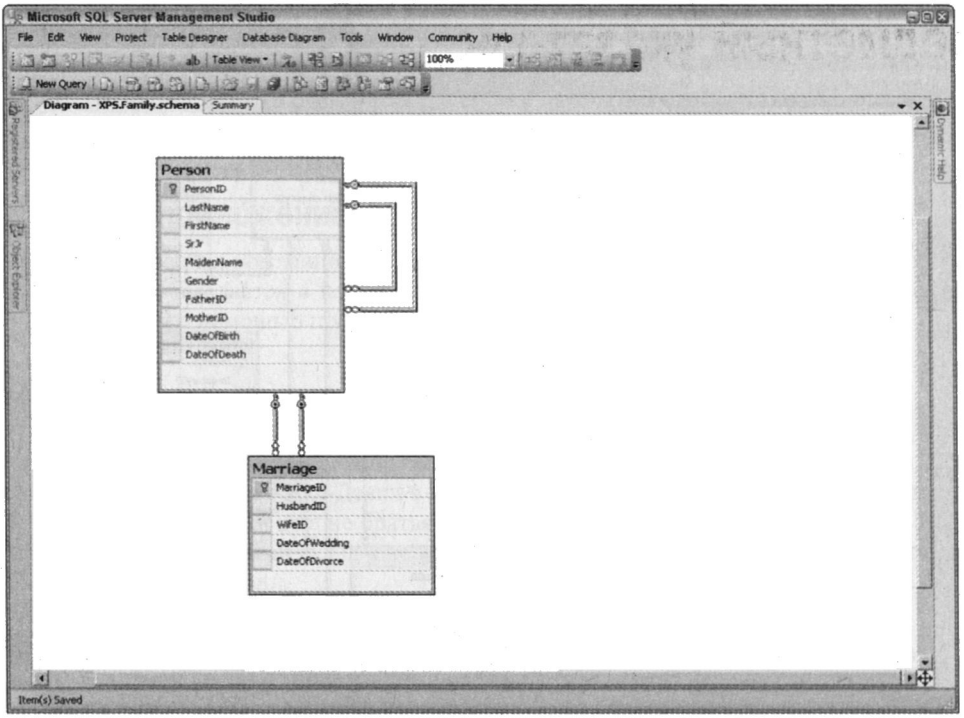


Рис. Б.3. Схема учебной базы данных `Family`*, отображенная в окне конструктора баз данных утилиты Management Studio*

Требования к приложению

База данных предназначена для экспериментирования с полнотекстовым поиском в SQL Server. Исходя из этого, база данных должна содержать несколько символьных столбцов, столбец с особо крупными объектами (BLOB) или изображениями и столбец типа BLOC.

Модель базы данных

Модель базы данных крайне проста. Она содержит всего одну таблицу, в каждой строке которой находится одна басня.

Предметный указатель

A

ACID, 1116
Activity Monitor, 1131
ADO, 639; 640; 907
ADO.NET, 134; 639; 652; 907
AdventureWorks, 107
AWE, 115; 711

B

BCP, 447
BIDS, 897
Books Online, 105
BulkCopy, 106
Business Intelligence Development Studio, 104

C

CAS, 593
CLR, 589
  объекты, 595
CLS, 592
CoType, 642
CSP, 605
CTS, 592

D

DAO, 1184
DAS, 1163
Data Analyzer, 1069
Database Migration Assistant, 107
Database Tuning Advisor, 106
DB, 639
DBCC, 767
DCL, 95
DCOM, 662
DDL, 95
DTC, 97
DTS, 896

E

ETL, 47; 896
Everywhere Server Agent, 558
Expression Builder, 900

F

FLWOR, 676

G

GAC, 591
GUID, 365

H

HTTP, 684

I

InfoPath 2003, 689
IP-адрес, 624

J

JIT, 591

K

KPI, 967; 1087

L

Lazy Writer, 94; 1126

M

Managenent Studio, 103
MARS, 136; 653; 662
MDAC, 134; 639; 644
MDX, 997; 1010
Microsoft Query, 1064
Migration Assistant, 124
MOLAP, 942
MSFTESQL, 288
MSIL, 591

O

Object Explorer, 144
ODBC, 646; 655; 907
OLE DB, 640; 642; 647

P

Performance Monitor, 106
PIA, 643
Profiler, 665; 1079; 1133

Q

Query Analyzer 3.0, 563

R

RAID-массив, 117
RDA, 577
RDL, 1014
RMO, 827

S

Server Configuration Manager, 133
Server Explorer, 664
SID, 864
SMO, 97
SNAC, 639; 644
SOA, 590; 683
SOAP, 943
Solution Explorer, 155
Soundex, 200
SQL Everywhere, 560
  администрирование, 583
  восстановление, 586
  обслуживание, 584
  установка, 560
SQL Mail, 98
SQL Native Client, 658
SQL Profiler, 106
SQL Server
  Developer Edition, 101
  Enterprise Edition, 101
  Everywhere Edition, 103
  Express Edition, 102
  Standard Edition, 101
  Workgroup Edition, 102
  журналы, 440
  обновление версии, 121
  установка, 118
SQL Server 2005 Upgrade Advisor, 121
SQL Server Agent, 97; 787
SQL Server Configuration Manager, 104
SQL Server Upgrade Advisor, 107
SQLCmd, 106
SQLNCLI, 639
Surface Area Configuration, 104; 127; 855

T

Transact-SQL, 94

U

UDM, 942; 962

W

Web-синхронизация, 822
Web-служба, 579
  Reporting Services, 1047
WSDL, 686

X

XML, 136
XPath, 676
XQuery, 675
XSD, 674

A

Автоматическое сжатие, 724
Агент
  распространения, 815; 817
  слияния, 819
  снимков, 814; 816
  чтения журнала, 816
  чтения очереди, 815
Агрегация, 942
Адаптер данных, 667
Аргумент поиска, 1108
Архивирование, 542
Архитектура
  клиент/сервер, 88
  многоуровневая, 91
  ориентированная на службы, 92
  программы, 49
Асинхронное выполнение, 669
Ассоциация, 979
Атомарность, 52; 1117
Атрибут, 209; 595
  SqlUserDefinedType, 627
  StructLayout, 627
Аутентификатор, 881
Аутентификация Windows, 859

Б

База данных
  миграция из SQL CE 2.0, 575
  объектно-ориентированная, 48
  отключение, 739
  по умолчанию, 862
  подключение, 739
  распространения, 811
  реляционная, 47
  сжатие, 777
  системная, 108
    Master, 108
    MSDB, 108
  файлы, 384
  федеративная, 309
Безопасность, 60
Бизнес-аналитика, 895
Бизнес-логика, 510
Блокировка, 63; 711
  обновления, 1129
  общая, 1129
  оптимистическая, 1144
  преднамеренная, 1130
  схемы, 1130
  эксклюзивная, 1129
Брокер служб, 614

В

Варианса, 256
Вертикальная фильтрация, 830
Вертикальное разделение, 830
Взаимоблокировка, 54; 1140
Виртуальная частная сеть, 819
Виртуальный сервер, 1155
Включение, 1126
Владелец объекта, 857
Внешний ключ, 399
Воссоздание базы данных, 761
Восстановление, 808
  системных баз данных, 764
Временной ряд, 994
Выделение памяти, 708
Выражение, 182; 900
Вычисляемый член, 1007

Г

Глобальный кэш сборок, 591
Горизонтальная фильтрация, 831
Горизонтальное разделение, 831
Горячая замена, 1149
Гранулярность, 1128
Группа взаимодействия, 615
Грязное чтение, 53; 1117

Д

Декларативная ссылочная целостность, 51; 400
Делегирование полномочий, 863
Денормализация, 381; 462
Дерево решений, 989
Диаграмма
  базы данных, 148
  Венна, 210
Диапазон данных, 1068
Динамический SQL, 431
Динамическое представление управления, 108; 1085
Дискретизация, 962
Диспетчер
  блокировок, 94
  буфера, 94
  отчетов, 1043
  подключений, 898; 906
Домен, 51
  приложения, 592
Доставка журнала, 805; 808; 1150
Доступность, 57; 1148

Ж

Живучесть, 53; 743; 1117
Журнал аудита, 61; 524
Журнал транзакций, 754
  восстановление, 1127
  сжатие, 756

З

Задание, 791; 796
Задержка репликации, 816
Закладка, 153
Запрос, 418
  кубический, 265
  пакетный, 1073
  перекрестный, 267
  перекрестный динамический, 271
  распределенный, 322
    гибридный метод, 337
    локальный, 334
    сквозной, 337
  сведения, 264

Защита C2, 716
Защита управления доступом, 878
Зеркальное отображение, 135; 805; 1157
  SQL Server, 809
  аппаратное, 809
  программное, 809

И

Идентификатор пользователя сервера, 864
Иерархия
  естественная, 955
Иерархия атрибутов, 951
Избыточность, 58
Издатель, 578; 810
Измерение, 318; 937; 999
  временное, 958
Изоляция, 53; 1117
  полная, 53
Индекс, 56; 1092
  кластеризованный, 1092
  некластеризованный, 1093
  обслуживание, 772
  отключение, 1100
  первичный, 675
  покрывающий, 1096
  составной, 1096
  уникальный, 1098
Индексация, 63
Индексированное представление, 1176
Инкапсуляция, 683
Инструкция
  ALTER, 466
  CREATE, 382; 466
  DELETE, 358
  DENY, 856
  DROP, 466
  DROP DATABASE, 389
  GRANT, 856
  INSERT, 345
    DEFAULT, 350
    EXEC, 348
    INTO, 350
    VALUES, 345
  REVOKE, 856
  SELECT, 159; 422
  SET, 422
  UPDATE, 353
Интерфейс, 642
  SQLXML, 659
Инъекция SQL, 884
Источник, 917
Исчезающие строки, 314

К

Каскадное удаление, 359; 402
Класс, 50; 622
  Command, 654
  Connection, 654
  DataAdapter, 654
  DataReader, 654
  DataSet, 653; 654; 660
  Error, 654
  Exception, 654
  IUnknown, 652
  Parameter, 654
  ProviderFactory, 654
  Recordset, 660
  TableAdapter, 655
  Transaction, 655
Классификация, 979
Кластер, 1154
Кластеризация, 805; 809; 990
Ключ
  альтернативный, 957
  временной, 994
  первичный, 1096
  суррогатный, 957
Ключевой индикатор производительности, 967; 1087
Кодовая страница, 901
Команда
  APPLY, 491
  BREAK, 426
  BULK INSERT, 445
  CONTINUE, 426
  EXEC, 474
  execute, 432
  FETCH, 450
  Goto, 427
  Imports, 627
  Raiserror, 438
  RECEIVE, 616
  RETURN, 472
  START, 888
  Try...Catch, 435
  WAITFOR, 616
Комментарий, 420
Конкуренция, 53; 670
Консолидация, 253; 603
  вложенная, 261
Конструктор
  баз данных, 390

запросов, 149
таблиц, 390
Контейнер, 910
Контракт, 615; 684
Контрольная точка, 905
Конфигурирование
автозапуска, 705
базы данных, 701
индексов, 722
памяти, 706
подключения, 702
процессора, 712
сервера, 699; 705; 720
системы безопасности, 715
состояния базы данных, 729
триггеров, 728
удаленного доступа, 718
Концевая точка, 684
неявная, 685
явная, 686
Координатор распределенных транзакций, 97; 340
Кортеж, 209; 1000
Коэффициент заполнения индекса, 775; 1098
Криптография, 879
алгоритм, 882
Куб, 962
Куб данных, 949
Курсор, 271; 449
обновления, 458
прямого доступа, 457
рекурсивный, 280
Куча, 598; 1094
Кэширование, 57

Л

Линейная регрессия, 990
Логистическая регрессия, 992
Логическая запись, 824
Логическое удаление, 539
каскадное, 541

М

Манифест метаданных, 592
Маркер трассировки, 825; 851
Массовая операция, 444
Мастер
Analysis Wizard, 122
Web-синхронизации, 845
восстановления, 759
доставки журнала, 1150
запросов, 1064
конфигурирования адаптера таблицы, 668
конфигурирования каталога полнотекстового поиска, 289
конфигурирования распространения, 829
копирования баз данных, 733
новой публикации, 820
плана обслуживания, 781
проектирования агрегаций, 970
раскрытия данных, 981
Мастер-ключ
базы данных, 880
служебный, 880
Масштабирование, 1161
Масштабируемость, 59
Матрица, 1035
Медиана, 256
Мера, 1001
Метаданные, 316
Метка, 427
Метод
EXIST, 677
MODIFY, 677
NODES, 677
QUERY, 677
VALUE, 677
выражений CASE, 269
коррелированных подзапросов, 268
поворота, 269
Миграция данных, 125
Множество, 998; 1006
Модель восстановления, 731; 743
Модель с собственным объединением, 275
Монитор ресурсов, 94
Монитор системы, 1076

Н

Набор данных, 653
типизированный, 661
Набор элементов, 993
Напоминание, 1199
Наследование, 623
Нейронная сеть, 992
Неповторяющееся чтение, 54; 1119
Нечеткий поиск, 303
Нормализация, 78
Нормальная форма
Бойса-Кодда, 83
вторая, 81
первая, 80

пятая, 84
третья, 82
четвертая, 84

О

Обновление, 549
Обобщение, 59
Обработка ошибок, 434; 905
Общее табличное выражение, 236
рекурсивное, 285
Общий интерфейс, 683
Объединение, 209
ANSI SQL-89, 213
внешнее, 216
внутреннее, 211
естественное, 211
неключевое, 226
перекрестное, 224
полное, 219
собственное, 222
тета, 225
Объект
DataAdapter, 661
DataReader, 601
Recordset, 668
SqlConnection, 608
SqlContext, 601
SQLPipe, 608
защищаемый, 855
Объекты управления сервером, 97
Обязательность отношений, 72
Ограничение, 408
проверки, 410
Окно
плавающее, 141
прикрепленное, 141
Оперативное представление, 162
Оператор, 184; 791
BEGIN...END, 425
CASE, 186
булева форма, 188
простая форма, 187
IF, 425
IS, 189
WHILE, 426
бинарный, 184
булев, 185
Оптимизатор запросов, 93
Оптимистическая конкуренция, 578
Осиротевший пользователь, 862
Отношение, 69
возвратное, 77
многие ко многим, 75
один к одному, 74
один ко многим, 73
Отчет, 1014
связанный, 1052
Очередь сообщений, 615
Ошибка доступа, 643

П

Пакет, 418
Пакетная обработка, 56
Палиндром, 750
Папка снимков, 814
Параллелизм, 715
Параметр
identity_insert, 365
INCLUDE, 1097
ON DELETE CASCADE, 359; 403
showplan, 1102
VIEW_METADATA, 316
WITH, 679
WITH CHECK OPTION, 314; 374
WITH ENCRYPTION, 316; 468
WITH RECOMPILE, 432
WITH SCHEMABINDING, 316
WITH TIES, 180
входной, 469
выходной, 471
Парафраза, 880
Первичный ключ, 68; 395
естественный, 396
суррогатный, 396
Переиздание, 811
Переменная, 421; 899
глобальная, 428
@@cursor_rows, 451
@@Error, 436
@@fetch_status, 451
@@RowCount, 437
с множественным присвоением, 424
табличная, 431
План выполнения запроса, 161; 1101
Плоский файл, 908
Плотность индекса, 775
Подзапрос, 85; 233
коррелированный, 244
простой, 234
скалярный, 237
Подкуб, 1005
Подписка, 578

Web-синхронизации, 844
по запросу, 813
принудительная, 813
управляемая данными, 1053
Подписчик, 578; 810
Позднее связывание, 652
Поиск закладки, 1106
Полиморфизм, 623; 1184
Полнотекстовый поиск, 288
Пользовательская консолидация CLR, 620
Пользовательский тип CLR, 619
гетерогенный, 632
развертывание, 633
тестирование, 631
Популяция, 256
Поставщик, 642
службы криптографии, 605
Поставщик данных, 647; 653; 1191
управляемый, 655
Построитель отчетов, 1048
Потерянные обновления, 54; 1144
Поток, 669; 712
данных, 904
управления, 901
Потребитель, 642
Правило, 412
Предикат
ALL, 176
DISTINCT, 176
TOP, 178
Предложение
COMPUTE, 265
GROUP BY, 257
HAVING, 263
ORDER BY, 172; 312
OUTPUT, 362
RETURN, 677
WHERE, 164; 170
WITH. 1005
Представление, 108; 307; 1070
вложенное, 318
горизонтально позиционированное, 315
индексированное, 309
разделенное, 318
распределенное, 309; 334
Предупреждение, 791
Преобразование, 922
типов данных, 203
Приемник, 919
Призрачные строки, 54; 1120
Принципал, 855
Принципы информационной архитектуры, 44; 45
Приоритет процесса, 714
Проблема исчезающих столбцов, 374
Прогнозирование, 980
Проекция, 962; 971
Протокол, 133
Kerberos, 863
Named Pipes, 133
TCP/IP, 133
Протокол счетчика, 1079
Протоколирование, 929
Профиль агента, 848
Процессор запросов, 555
Псевдоним, 163
Публикация, 578
Oracle, 822; 837
двусторонней репликации транзакций, 836
одноранговой репликации, 838
репликации слияния, 840
репликации снимков базы данных, 830
репликации транзакций, 832
Пустое значение, 55; 188; 408

P

Рабочее пространство, 127
Развертывание, 581
Раздел, 57: 968
Разделение, 1165
Разделитель пакета, 418
Разность множеств, 227
Разрешение, 870; 872
Раскрытие данных, 980
развертывание, 986
Распространитель, 811
локальный, 829
удаленный, 829
Регистрационная запись, 858; 863
Редактор запросов, 152
Редактор потоков данных. 904
Резервирование, 808
Резервное копирование, 743
в Management Studio, 750
журнала транзакций, 755
Резервный набор, 744
Реляционная алгебра, 85; 208
Реляционное деление, 248
с остатком, 249
точное, 251
Реляционное ядро, 92
Реляционный оператор, 85
Репликация, 804

инструкций DDL, 823
однoранговая, 817; 823
полнотекстовых индексов, 823
с централизованным издателем, 811
с централизованным подписчиком, 811
слияния, 578; 819
снимков баз данных, 814
транзакций, 816
Роль
public, 856; 873
sysadmin, 856
пользовательская, 856
приложения, 870
серверная, 865
стандартная, 873
фиксированная, 868

C

Сборка, 591; 592; 602
interop, 624; 640; 643
Свидетель, 1157
Сводная диаграмма, 1067
Сводная таблица, 1060
Свойство, 50
Связанные серверы, 322; 808
Сегментация, 979
Сервер
горячей замены, 58
кластеризованный, 58
Сериализация, 627; 630
Сертификат, 880
Синоним, 321
Синхронизация, 576
Система безопасности
Windows, 857
для мобильных решений, 582
службы отчетности, 1049
уровень C2, 877
уровня базы данных, 856; 866
уровня объекта, 871
уровня сервера, 855
Скалярное значение, 80
Слияние, 230
пересечения, 231
разности, 231
Служба
анализа, 100; 129
интеграции, 98; 808; 896
отчетности, 99; 129
репликаций, 96
уведомлений, 97
Смешанный режим, 113
Снимок, 579
Совместимость с SQL ANSI, 725
Соглашение об именах, 393
Среда быстрой разработки приложений, 594
Ссылочная целостность, 359; 511
Статистика, 256
Статья, 578; 812
Столбец
вычисляемый, 373
идентичности, 365; 396
ключевой, 1093
проецируемый, 308
Строгость отношения, 71
Строка инициализации, 335
Строка подключения, 1015
Структура, 622
Суффикс
FOR SOAP, 686
FOR XML, 680
Сущность, 209
видимая, 67
категорий, 76
поддерживаемая, 67
Схема, 392
звезда, 937; 950
Счетчик, 1076

T

Таблица, 1035
`Deleted`, 501; 503
`Inserted`, 501; 503
временная, 429
классификаторов, 76
стыковочная, 75
управляемая, 242
Тезаурус, 300
Теория оптимизации, 61; 1073
Тип данных, 404; 648; 659
`varbinary`, 502
XML, 660; 673
даты-времени, 406
пользовательский, 603; 619
символьный, 404
среды .NET, 599
управляемый, 592
числовой, 405
Точка трассировки, 666
Транзакция, 52; 1115
вложенная, 1116
порядок выполнения, 496
распределенная, 339; 807

Транзитивная зависимость, 82
Трассировка, 665
Триггер, 413; 495; 808
  AFTER, 372; 496; 498
  INSTEAD OF, 371; 496; 499
  вложенный, 506
  интеграции CLR, 610
  логического удаления, 539
  последовательность выполнения, 508
  рекурсивный, 506

У

Уведомление, 973
  опроса, 973
Удаленность, 593
Удаленный доступ к данным, 554
Узел кластера, 805
Универсальное соглашение об именах, 749; 908
Унифицированная модель измерений, 942
Упаковка, 581; 598
Упреждающее кэширование, 943; 972
Уровень абстракции, 59; 544
  хранилища данных, 59
Уровень изоляции, 54; 1121
  Read Comited Snapshot, 1123
  Read Commited, 1122
  Read Commited Snapshot, 1136
  Read Uncommited, 1122
  Repeatable Read, 1122
  Seriakizable, 1122
  SNAPSHOT, 137; 1122; 1135
Условие
  BETWEEN, 165
  IN, 167
  LIKE, 168
Условное форматирование, 691
Устройство резервирования, 749
Утечка памяти, 643
Учетная запись guest, 866

Ф

Фабрика, 1187
Фабричная модель, 657
Файл данных
  вторичный, 386
  первичный, 386
Файл шума, 294
Файловая группа, 388; 394
Флаг активности, 542
Форма, 689
  доверительная, 691
  шаблон, 692
Функция
  App_name, 194
  Cast, 204
  CharIndex, 198
  Coalesce, 191
  contains, 295
  ContainsTable, 295
  Convert, 205
  DateAdd, 196
  DateDiff, 196
  DateName, 195
  DatePart, 196
  Db_name, 206
  Difference, 203
  EventData, 415
  Exists, 425
  GetDate, 195
  GetUTCDate, 195; 206
  Host_name, 194
  ident_current, 366
  IsDate, 406
  IsNull, 190
  Left, 198
  Len, 198
  Lower, 198
  Ltrim, 198
  newid, 367; 398
  NewsequentionalID, 399
  Nullif, 192
  objectproperty, 500
  OpenDataSource, 335
  OpenQuery, 338; 474
  OpenRowSet, 338
  OpenXML, 678
  PatIndex, 198
  pTitleCase, 199
  Replace, 199; 354
  Right, 198
  Rtrim, 198
  scope_identity, 366
  ServerProperty, 207
  Soundex, 201
  Str, 206
  Stuff, 197
  SubString, 197
  suser_name, 522
  Suser_sname, 194
  Trigger_NestLevel, 506
  updated, 501

Upper, 198
UserName, 194
детерминированная, 486
интеграции CLR, 609
итоговая, 254
коррелированная, 491
пользовательская, 484
разделения, 1167
скалярная, 193; 485
со связанной схемой, 488
табличная, 488
потоковая, 610

Х

Хеш-раздел, 1168
Хранилище данных, 937; 971
главное, 46
глобальное, 47
для кэширования, 46
объектно-реляционное, 48
поддержки объектов, 48
ссылочное, 47
Хранимая процедура, 465
sp_executeSQL, 432
sp_help, 427
интеграции CLR, 607
системная, 468
удаленная, 475

Ц

Целостность, 52; 70
данных, 771
домена, 51
ссылочная, 51
сущности, 51
транзакций, 52; 1116
Цепочка принадлежности, 876
Цифровая подпись, 634

Ш

Шаблон, 155; 1182
материализованного пути, 279
смежного списка, 275; 279
формы, 690
Шифрование, 582
асимметричное, 879
симметричное, 879

Я

Ядро, 129
базы данных, 92
хранилища данных, 555
Язык
манипулирования данными, 95
определения данных, 95
управления данными, 95
Ячейка, 998

Научно-популярное издание

Пол Нильсен

Microsoft SQL Server 2005.
Библия пользователя

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректоры *Л.А. Гордиенко, Л.В. Чернокозинская*

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 30.10.2007. Формат 70x100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 99,33. Уч.-изд. л. 76,84
Тираж 3000 экз. Заказ № 5236

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15

Создавайте базы данных SQL Server 2005 и профессионально управляйте ими

Опытные разработчики баз данных всегда знают, какие процессы происходят внутри СУБД. Предлагаемое руководство поможет вам узнать о том, как работает самая современная и мощная СУБД — SQL Server 2005. Каждая из семи частей книги логически последовательно фокусирует внимание на ключевых элементах сервера, и вам будет нетрудно найти именно то, что нужно, — как описание функциональности сервера, так и практические примеры. Если вы разрабатываете или администрируете базы данных SQL Server 2005, то эта книга поможет вам добиться успеха!

- Основы технологии SQL Server
- Логика запросов и оптимизация
- Защита промышленных баз данных и обеспечение высокой доступности
- Интеграция с .NET CLR и XML
- Бизнес-логика и инструменты анализа данных
- Оценка производительности, блокировка транзакций и масштабирование сверхкрупных баз данных
- Новинки пакетов обновлений SP1 и SP2

Уровень:
от начинающих до опытных пользователей

Категория:
базы данных/SQL Server 2005

Об авторе

Пол Нильсен — известный разработчик баз данных, автор и преподаватель, специализирующийся на архитектуре баз данных и технологиях Microsoft SQL Server. Пол входит в совет директоров PASS (Professional Association for SQL Server), курирует ряд форумов, посвященных SQL Server, и организовывает специализированные семинары.

Web-сайт книги

Посетите сайт книги по адресу www.sqlserverbible.com, где вы найдете демонстрационные ролики, которые помогут вам увидеть пошаговое решение различных задач, описываемых в книге. На этом сайте вы также найдете учебные базы данных, примеры программного кода, используемые в книге, а также ссылки на различные источники в Интернете, посвященные SQL Server 2005.



ISBN 978-5-8459-1314-2

ДИАЛЕКТИКА
www.dialektika.com

WILEY
wiley.com

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL