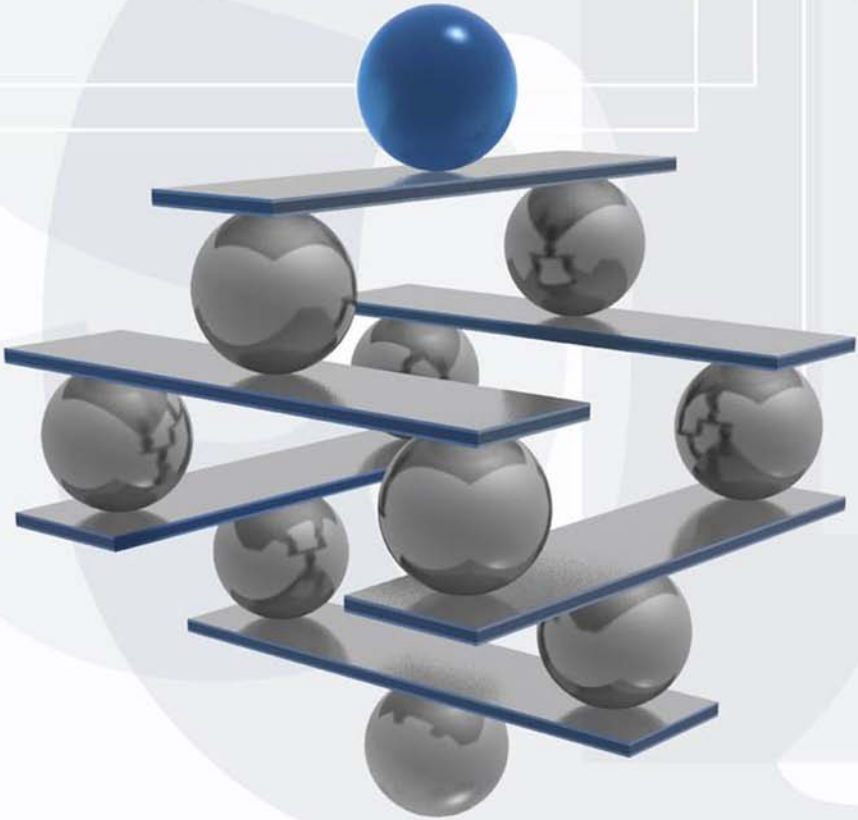
Научитесь быстро и легко
использовать все возможности SQL Server 2012

Microsoft® SQL Server™ 2012

Руководство для начинающих



ДУШАН ПЕТКОВИЧ

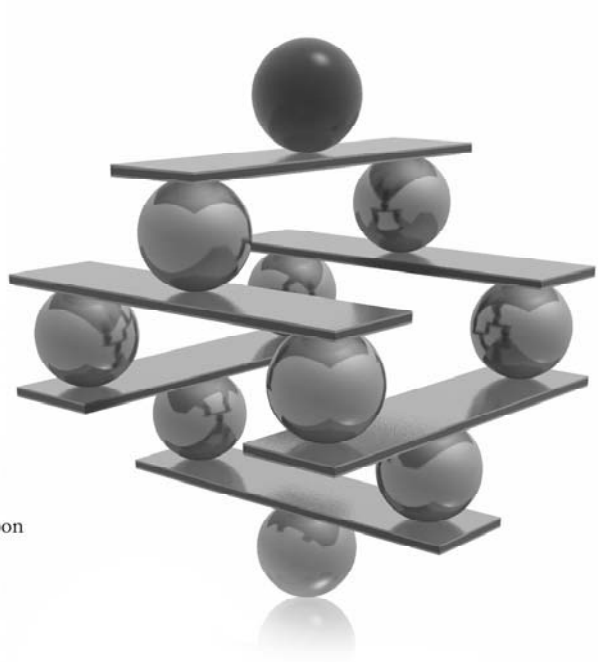
bhv
McGraw Hill

Microsoft® SQL Server™ 2012

A BEGINNER'S GUIDE

Fifth Edition

Dušan Petković



McGraw Hill

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

Душан Петкович

Microsoft® SQL Server™ 2012

Руководство для начинающих

Санкт-Петербург
«БХВ-Петербург»
2013

УДК 004.65
ББК 32.973.26-018.2
П29

Петкович Д.

П29 Microsoft® SQL Server™ 2012. Руководство для начинающих: Пер. с англ. — СПб.: БХВ-Петербург, 2013. — 816 с.: ил.

ISBN 978-5-9775-0854-4

Просто и доступно рассмотрены теоретические основы СУБД SQL Server 2012. Показана установка, конфигурирование и поддержка MS SQL Server 2012. Описан язык манипулирования данными Transact-SQL. Рассмотрены создание базы данных, изменение таблиц и их содержимого, запросы, индексы, представления, триггеры, хранимые процедуры и функции, определенные пользователем. Показана реализация безопасности с использованием аутентификации, шифрования и авторизации. Уделено внимание автоматизации задач администрирования СУБД. Рассмотрено создание резервных копий данных и выполнение восстановления системы. Описаны службы Microsoft Analysis Services, Microsoft Reporting Services и другие инструменты для бизнес-анализа. Рассмотрены технология работы с документами XML, управление пространственными данными, полнотекстовый поиск и многое другое.

Для начинающих программистов

УДК 004.65
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Екатерина Капалыгина*
Перевод с английского	*Сергея Таранушенко*
Редактор	*Юрий Рожко*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление обложки	*Марины Дамбиевой*

Original edition copyright © 2012 by the McGraw-Hill Companies. All rights reserved.
Russian edition copyright © 2012 year by BHV — St.Petersburg. All rights reserved.
No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Оригинальное издание выпущено McGraw-Hill Companies в 2012 году. Все права защищены.
Русская редакция издания выпущена издательством "БХВ-Петербург" в 2012 году. Все права защищены.
Никакая часть настоящей книги не может быть воспроизведена или передана 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 то нет письменного разрешения издательства.

Подписано в печать 30.11.12.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 65,79.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-0-07-176160-4 (англ.)
ISBN 978-5-9775-0854-4 (рус.)

© 2012 by The McGraw-Hill Companies
© Перевод на русский язык "БХВ-Петербург", 2013

Оглавление

Об авторе 3

О техническом редакторе 3

Благодарности 4

Введение 5
Цели книги 5
Новые возможности SQL Server 2012, рассматриваемые в этой книге 6
Организация этой книги 7
Изменения по сравнению с предыдущими изданиями 12
Различия между синтаксисами языка SQL и Transact-SQL 14
Работа с образцовыми базами данных 14

ЧАСТЬ I. ОСНОВНЫЕ ПОНЯТИЯ И УСТАНОВКА 17

Глава 1. Введение в системы реляционных баз данных 19
Обзор систем баз данных 19
- Разнообразные пользовательские интерфейсы 20
- Физическая независимость данных 21
- Логическая независимость данных 21
- Оптимизация запросов 21
- Целостность данных 21
- Управление параллелизмом 22
- Резервное копирование и восстановление 22
- Безопасность баз данных 22

Системы реляционных баз данных 23
- Работа с демонстрационной базой данных книги 23
- SQL — язык реляционной базы данных 26

Проектирование базы данных 27
- Нормальные формы 28
  - Первая нормальная форма 28
  - Вторая нормальная форма 29
  - Третья нормальная форма 30
- Модель "сущность — отношение" 30

Соглашения о синтаксисе 32
Резюме 33
Упражнения 33

Глава 2. Планирование установки и установка SQL Server 37
Версии SQL Server 37
Этап планирования 39
Общие рекомендации 39
Какие компоненты SQL Server следует установить? 39
Где расположить корневой каталог? 41
Следует ли использовать множественные экземпляры компонента Database Engine? 41
Какой режим проверки подлинности использовать для компонента Database Engine? 42
Планирование установки 42
Требования к оборудованию и программному обеспечению 44
Требования к оборудованию 44
Требования к сети 44
Документация по безопасности 45
Заметки о версии в сети 45
Электронная справка по установке 46
Средство проверки конфигурации 46
Установка SQL Server 46
Резюме 54

Глава 3. Среда управления SQL Server Management Studio 55
Введение в среду управления SQL Server Management Studio 55
Подключение к серверу 56
Компонент Registered Servers 58
Компонент Object Explorer 58
Организация панелей среды SQL Server Management Studio и перемещение по ним 59
Использование среды SQL Server Management Studio с компонентом Database Engine 60
Администрирование серверов баз данных 60
Регистрация серверов 61
Подключение к серверу 62
Создание новой группы серверов 62
Управление множественными серверами 62
Запуск и останов серверов 63
Управление базами данных посредством обозревателя объектов 64
Создание баз данных, не прибегая к использованию языка Transact-SQL 64
Модифицирование баз данных, не прибегая к использованию языка Transact-SQL 67
Управление таблицами, не прибегая к использованию языка Transact-SQL 68
Разработка запросов, используя среду SQL Server Management Studio 73
Редактор запросов 73
Обозреватель решений 77
Отладка SQL Server 77
Резюме 80
Упражнения 81

ЧАСТЬ II. ЯЗЫК TRANSACT-SQL 83

Глава 4. Компоненты SQL 85
Основные объекты SQL 85
Литералы 86
Ограничители 87

Комментарии ... 87
Идентификаторы ... 88
Зарезервированные ключевые слова ... 88
Типы данных ... 88
Числовые типы данных ... 89
Символьные типы данных ... 89
Временные типы данных ... 90
Прочие типы данных ... 92
Двоичные и битовые типы данных ... 92
Тип данных больших объектов ... 93
Тип данных *UNIQUEIDENTIFIER* ... 94
Тип данных *SQL_VARIANT* ... 94
Тип данных *HIERARCHYID* ... 94
Тип данных *TIMESTAMP* ... 95
Варианты хранения ... 95
Хранение данных типа *FILESTREAM* ... 95
Разреженные столбцы ... 96
Функции языка SQL ... 96
Агрегатные функции ... 96
Скалярные функции ... 97
Числовые функции ... 98
Функции даты ... 100
Строковые функции ... 100
Системные функции ... 103
Функции метаданных ... 105
Скалярные операторы ... 106
Глобальные переменные ... 107
Значение *NULL* ... 107
Резюме ... 109
Упражнения ... 109

Глава 5. Язык описания данных ... 111
Создание объектов баз данных ... 111
Создание базы данных ... 112
Создание моментального снимка базы данных ... 115
Присоединение и отсоединение баз данных ... 116
Инструкция *CREATE TABLE*: базовая форма ... 116
Инструкция *CREATE TABLE* и ограничения декларативной целостности ... 119
Предложение *UNIQUE* ... 120
Предложение *PRIMARY KEY* ... 121
Предложение *CHECK* ... 122
Предложение *FOREIGN KEY* ... 123
Ссылочная целостность ... 124
Возможные проблемы со ссылочной целостностью ... 125
Опции *ON DELETE* и *ON UPDATE* ... 127
Создание других объектов баз данных ... 128
Ограничения для обеспечения целостности и домены ... 130
Псевдонимы типов данных ... 130
Типы данных CLR ... 132

Модифицирование объектов баз данных 132
Изменение базы данных 132
Добавление и удаление файлов базы данных, файлов журналов и файловых групп 133
Изменение свойств файлов и файловых групп 134
Установка опций базы данных 134
Хранение данных типа *FILESTREAM* 135
Автономные базы данных 138
Изменение таблиц 140
Добавление и удаление столбцов 140
Изменение свойств столбцов 141
Добавление и удаления ограничений для обеспечения целостности 141
Разрешение и запрещение ограничений 142
Переименование таблиц и других объектов баз данных 143
Удаление объектов баз данных 144
Резюме 144
Упражнения 145

Глава 6. Запросы 149
Инструкция *SELECT*. Ее предложения и функции 149
Предложение *WHERE* 151
Логические операторы 153
Операторы *IN* и *BETWEEN* 157
Запросы, связанные со значением *NULL* 159
Оператор *LIKE* 161
Предложение *GROUP BY* 164
Агрегатные функции 165
Обычные агрегатные функции 165
Статистические агрегатные функции 170
Агрегатные функции, определяемые пользователем 170
Предложение *HAVING* 171
Предложение *ORDER BY* 172
Использование предложения *ORDER BY* для разбиения результатов на страницы 173
Инструкция *SELECT* и свойство *IDENTITY* 174
Оператор *CREATE SEQUENCE* 176
Операторы работы с наборами 178
Оператор *UNION* 179
Операторы *INTERSECT* и *EXCEPT* 182
Выражения *CASE* 183
Подзапросы 185
Подзапросы и операторы сравнения 186
Подзапросы и оператор *IN* 186
Подзапросы и операторы *ANY* и *ALL* 188
Временные таблицы 189
Оператор соединения *JOIN* 190
Две синтаксические формы реализации соединений 191
Естественное соединение 192
Соединение более чем двух таблиц 196

Декартово произведение 197
Внешнее соединение 198
Другие формы операций соединения 200
Тета-соединение 201
Самосоединение, или соединение таблицы самой с собой 201
Полусоединение 202
Связанные подзапросы 203
Подзапросы и функция *EXISTS* 204
Что использовать, соединения или подзапросы? 205
Преимущества подзапросов 205
Преимущества соединений 206
Табличные выражения 206
Производные таблицы 206
Обобщенные табличные выражения 208
ОТВ и нерекурсивные запросы 208
ОТВ и рекурсивные запросы 210
Резюме 214
Упражнения 214

Глава 7. Модифицирование содержимого таблиц 219
Инструкция *INSERT* 219
Вставка одной строки 220
Вставка нескольких строк 222
Конструкторы значений таблицы и инструкция *INSERT* 223
Инструкция *UPDATE* 224
Инструкция *DELETE* 226
Другие инструкции и предложения Transact-SQL для модификации таблиц 228
Инструкция *TRUNCATE TABLE* 228
Инструкция *MERGE* 229
Предложение *OUTPUT* 230
Резюме 233
Упражнения 234

Глава 8. Хранимые процедуры и определяемые пользователем функции 237
Процедурные расширения 237
Блок инструкций 238
Инструкция *IF* 238
Инструкция *WHILE* 239
Локальные переменные 240
Прочие процедурные инструкции 241
Обработка исключений с помощью инструкций *TRY*, *CATCH* и *THROW* 242
Хранимые процедуры 245
Создание и исполнение хранимых процедур 246
Предложение *WITH RESULTS SETS* инструкции *EXECUTE* 250
Изменение структуры хранимых процедур 251
Хранимые процедуры и среда CLR 251
Определяемые пользователем функции 256
Создание и выполнение определяемых пользователем функций 257
Вызов определяемой пользователем функции 259
Возвращающие табличное значение функции 259

Возвращающие табличное значение функции и инструкция *APPLY* 260
Возвращающие табличное значение параметры 262
Изменение структуры определяемых пользователями инструкций 263
Определяемые пользователем функции и среда CLR 264
Резюме 265
Упражнения 266

Глава 9. Системный каталог 267
Введение в системный каталог 267
Общие интерфейсы 269
Представления каталога 269
Запросы к представлениям каталога 271
Динамические административные представления и функции 272
Информационная схема 274
Представление *information_schema.tables* 275
Представление *information_schema.columns* 275
Специализированные интерфейсы 275
Системные хранимые процедуры 276
Системные функции 277
Функции свойств 277
Резюме 278
Упражнения 278

Глава 10. Индексы 281
Общие сведения 281
Кластеризованные индексы 283
Некластеризованные индексы 284
Язык Transact-SQL и индексы 286
Создание индексов 286
Получение информации о фрагментации индекса 290
Редактирование информации индекса 291
Изменение индексов 292
Пересоздание индекса 292
Реорганизация страниц листьев индекса 293
Отключение индекса 293
Удаление и переименование индексов 294
Рекомендации по созданию и использованию индексов 294
Индексы и условия предложения *WHERE* 295
Индексы и оператор соединения 296
Покрывающий индекс 296
Специальные типы индексов 297
Виртуальные вычисляемые столбцы 298
Постоянные вычисляемые столбцы 298
Резюме 299
Упражнения 299

Глава 11. Представления 301
Инструкции DDL и представления 301
Создание представления 302

Изменение и удаление представлений 305
Редактирование информации о представлениях 307
Инструкции DML и представления 307
Выборка информации из представления 307
Инструкция *INSERT* и представление 308
Инструкция *UPDATE* и представление 311
Инструкция *DELETE* и представление 313
Индексированные представления 314
Создание индексированного представления 314
Модифицирование структуры индексированного представления 316
Редактирование информации, связанной с индексированными представлениями 317
Преимущества индексированных представлений 318
Резюме 319
Упражнения 320

Глава 12. Система безопасности Database Engine 323
Аутентификация 325
Реализация режима аутентификации 326
Шифрование данных 326
Симметричные ключи 327
Асимметричные ключи 328
Сертификаты 329
Редактирование пользовательских ключей 330
Расширенное управление ключами SQL Server 331
Способы шифрования данных 331
Настройка безопасности компонента Database Engine 332
Управление безопасностью с помощью среды Management Studio 332
Управление безопасностью посредством инструкций Transact-SQL 333
Схемы 335
Разделение пользователей и схем 335
Инструкции языка DDL для работы со схемами 336
Инструкция *CREATE SCHEMA* 336
Инструкция *ALTER SCHEMA* 337
Инструкция *DROP SCHEMA* 338
Безопасность базы данных 338
Управление безопасностью базы данных с помощью среды Management Studio 339
Управление безопасностью базы данных посредством инструкций языка Transact-SQL 339
Схемы базы данных по умолчанию 341
Роли 341
Фиксированные серверные роли 342
Управление фиксированными серверными ролями 342
Регистрационное имя *sa* 344
Фиксированные роли базы данных 344
Фиксированная роль базы данных *public* 345
Присвоения пользователю членства в фиксированной роли базы данных 345
Роли приложений 346
Управление ролями приложений посредством среды Management Studio 346
Управление ролями приложений посредством инструкций Transact-SQL 346
Активация ролей приложений 347

Определяемые пользователем роли сервера 348
Определяемые пользователем роли баз данных 348
Управление определяемыми пользователем ролями базы данных с помощью среды Management Studio 348
Управление определяемыми пользователем ролями базы данных с помощью инструкций Transact-SQL 348
Авторизация 350
Инструкция *GRANT* 350
Инструкция *DENY* 355
Инструкция *REVOKE* 356
Управление разрешениями с помощью среды Management Studio 357
Управление авторизацией и аутентификацией для автономных баз данных 360
Отслеживание изменений 361
Безопасность данных и представления 364
Резюме 365
Упражнения 366

Глава 13. Управление параллельной работой 369
Модели одновременного конкурентного доступа 370
Транзакции 371
Свойства транзакций 372
Инструкции Transact-SQL и транзакции 373
Журнал транзакций 376
Блокировка 377
Режимы блокировки 378
Гранулярность блокировки 380
Укрупнение блокировок 381
Настройка блокировок 382
Подсказки блокировок 382
Параметр *LOCK_TIMEOUT* 383
Отображение информации о блокировках 383
Взаимоблокировки 384
Уровни изоляции 385
Проблемы одновременного конкурентного доступа 385
Компонент Database Engine и уровни изоляции 386
Уровень изоляции *READ UNCOMMITTED* 386
Уровень изоляции *READ COMMITTED* 387
Уровень изоляции *REPEATABLE READ* 387
Уровень изоляции *SERIALIZABLE* 388
Установка и редактирование уровня изоляции 388
Управление версиями строк 389
Уровень изоляции *READ COMMITTED SNAPSHOT* 389
Уровень изоляции *SNAPSHOT* 390
Разница между уровнями изоляции *READ COMMITTED* SNAPSHOT и *SNAPSHOT* 391
Резюме 391
Упражнения 392

Глава 14. Триггеры 393
Введение 393
Создание триггера DML 394

Изменение структуры триггера 395
Использование виртуальных таблиц *deleted* и *inserted*....395
Области применения DML-триггеров....396
Триггеры *AFTER*....396
Создание журнала аудита....396
Реализация бизнес-правил....398
Принудительное обеспечение ограничений целостности 399
Триггеры *INSTEAD OF*....400
Триггеры *first* и *last*....401
Триггеры DDL и области их применения....402
Триггеры DDL уровня базы данных 403
Триггеры DDL уровня сервера 404
Триггеры и среда CLR....405
Резюме 409
Упражнения....409

ЧАСТЬ III. SQL SERVER: СИСТЕМНОЕ АДМИНИСТРИРОВАНИЕ....411

Глава 15. Системная среда компонента Database Engine....413
Системные базы данных 413
База данных *master* 414
База данных *model* 414
База данных *tempdb* 414
База данных *msdb*....415
Хранение данных на диске....416
Свойства страниц данных 417
Заголовок страницы....417
Пространство для данных 418
Таблица смещения строк....418
Типы страниц данных....419
Страницы данных последовательных строк....419
Страницы данных переполнения строк 420
Параллельное выполнение задач....421
Утилиты и команда DBCC....422
Утилита *bcp* 422
Утилита *sqlcmd*....423
Утилита *sqlservr* 426
Команды DBCC....427
Команды проверки....427
Управление на основе политик....428
Ключевые термины и концепты управления на основе политик....429
Применение управления на основе политик 429
Резюме 432
Упражнения....433

Глава 16. Резервное копирование, восстановление и доступность системы 435
Причины потери данных....436
Введение в методы резервного копирования 437
Полное резервное копирование базы данных 437

Разностное резервное копирование 438
Резервное копирование журнала транзакций 438
Резервное копирование файлов или файловых групп 440
Выполнение резервного копирования базы данных 440
Резервное копирование с помощью инструкций Transact-SQL 440
Типы устройств резервного копирования 441
Инструкция *BACKUP DATABASE* 442
Инструкция *BACKUP LOG* 444
Резервное копирование с помощью интегрированной среды Management Studio 444
Создание графика резервного копирования в среде SQL Server Management Studio 448
Для каких баз данных создавать резервную копию? 448
Резервное копирование базы данных *master* 448
Резервное копирование производственных баз данных 449
Восстановление базы данных 449
Автоматическое восстановление 449
Ручное восстановление 450
Проверка резервного набора на пригодность для восстановления 450
Восстановления баз данных и журналов транзакций с помощью инструкций Transact-SQL 452
Восстановление баз данных и журналов транзакций с помощью среды Management Studio 455
Восстановление до метки 458
Восстановление базы данных *master* 458
Восстановление других системных баз данных 459
Модели восстановления 459
Модель полного восстановления 460
Модель восстановления с неполным протоколированием 460
Простая модель восстановления 461
Изменение и редактирование модели восстановления 462
Доступность системы 463
Использование резервного сервера 463
Использование технологии RAID 464
Расслоение дисков. RAID 0 465
Зеркалирование. RAID 1 465
Контроль по четности. RAID 5 466
Зеркальное отображение базы данных 466
Отказоустойчивая кластеризация 467
Доставка журналов транзакций 467
Высокий уровень доступности и восстановления в аварийных ситуациях (HARD) 468
Группы, реплики и режимы обеспечения доступности 468
Конфигурация технологии HADR 469
Мастер плана обслуживания 470
Резюме 473
Упражнения 475

Глава 17. Система автоматизации задач администрирования 477
Запуск службы SQL Server Agent 478
Создание заданий и операторов 479
Создание задания и его шагов 479

Создание расписания задания 483
Операторы извещения о состоянии задания 484
Просмотр журнала истории задания 486
Предупреждающие сообщения 486
Сообщения об ошибках 487
Журнал ошибок службы SQL Server Agent 488
Журнал событий приложений Windows 489
Определение предупреждающих сообщений для обработки ошибок 489
Создание предупреждающих сообщений для системных ошибок 489
Создание предупреждающих сообщений для ошибок определенного уровня 491
Создание предупреждающих сообщений для определяемых пользователем ошибок 492
Резюме 494
Упражнения 494

Глава 18. Репликация данных 497
Распределенные данные и методы распределения 498
Общие сведения о репликации в SQL Server 499
Издатели, распространители и подписчики 500
Публикации и статьи 501
База данных *distribution* 502
Агенты 503
Агент Snapshot Agent 503
Агент Log Reader 503
Агент Distribution Agent 503
Агент Merge Agent 503
Типы репликации 504
Репликации транзакций 504
Одноранговая репликация транзакций 505
Репликация моментальных снимков 506
Репликация слиянием 507
Модели репликации 508
Центральный издатель с распространителем 508
Центральный издатель с удаленным распространителем 509
Центральный подписчик с множественными издателями 510
Множественные издатели с множественными подписчиками 510
Управление репликацией 511
Настройка серверов распространения и публикации 511
Настройка публикаций 513
Настройка серверов подписки 514
Резюме 515
Упражнения 515

Глава 19. Оптимизатор запросов 517
Этапы обработки запроса 518
Как работает оптимизация запроса 519
Анализ запроса 519
Выбор индексов 520
Селективность выражения с индексированным столбцом 520

Статистические данные индекса 522
Статистические данные столбца....523
Выбор порядка соединения....524
Выбор метода выполнения соединения 524
Вложенный цикл 524
Соединение слиянием....525
Соединение хешированием....525
Кэширование планов 526
Редактирование планов выполнения....526
Отображение информации о кэше планов....527
Инструменты для редактирования стратегии оптимизатора 527
Инструкция *SET* 528
Текстовый план выполнения 528
План выполнения XML 530
Другие опции инструкции *SET* 531
Среда Management Studio и графические планы выполнения....532
Пример планов выполнения....533
Динамические административные представления и оптимизатор запросов 537
Представление *sys.dm_exec_query_optimizer_info*....537
Представление *sys.dm_exec_query_plan*....538
Представление *sys.dm_exec_query_stats*....539
Представления *sys.dm_exec_sql_text* и *sys.dm_exec_text_query_plan*....540
Представление *sys.dm_exec_procedure_stats* 540
Представление *sys.dm_exec_cached_plans* 540
Подсказки оптимизации....540
Зачем надо использовать подсказки оптимизации 541
Типы подсказок оптимизации....541
Табличные подсказки 542
Подсказки соединения....544
Подсказки запросов 547
Структуры планов....548
Резюме 549

Глава 20. Настройка производительности....551
Факторы, влияющие на производительность....552
Приложения базы данных и производительность....552
Эффективность кода приложения 552
Проектирование на физическом уровне 553
Компонент Database Engine и производительность 554
Оптимизатор запросов....555
Блокировки....555
Системные ресурсы и производительность....555
Дисковые операции ввода/вывода....557
Операции с памятью....559
Мониторинг производительности 560
Обзор монитора Performance Monitor 561
Мониторинг центрального процессора....562
Мониторинг центрального процессора посредством счетчиков 562
Мониторинг центрального процессора посредством представлений 563

Мониторинг памяти 564
Мониторинг памяти посредством счетчиков 565
Мониторинг памяти с использованием динамических административных представлений 565
Мониторинг памяти с использованием команды *DBCC MEMORYSTATUS* 566
Мониторинг дисковой системы 567
Мониторинг дисковой системы с использованием счетчиков 567
Мониторинг дисковой системы с использованием динамических административных представлений 568
Мониторинг сетевого интерфейса 568
Мониторинг сетевого интерфейса с помощью счетчиков 569
Мониторинг сетевого интерфейса с помощью динамического административного представления 569
Мониторинг сетевого интерфейса с помощью системной процедуры *sp_monitoring* 570
Выбор подходящего инструмента для мониторинга 570
Приложение SQL Server Profiler 571
Помощник Database Engine Tuning Advisor 571
Предоставление информации помощнику Database Engine Tuning Advisor 572
Работа с помощником Database Engine Tuning Advisor 574
Другие средства SQL Server для настройки производительности 579
Сборщик Performance Data Collector 579
Создание хранилища MDW 580
Организация сбора данных 581
Просмотр отчетов 581
Регулятор ресурсов Resource Governor 582
Создание групп нагрузок и пулов ресурсов 584
Мониторинг конфигурации регулятора Resource Governor 586
Резюме 586
Упражнения 587

ЧАСТЬ IV. SQL SERVER И БИЗНЕС-АНАЛИТИКА 589

Глава 21. Введение в бизнес-аналитику 591
Оперативная обработка транзакций в сравнении с бизнес-аналитикой 591
Оперативная обработка транзакций 592
Системы бизнес-аналитики 593
Хранилища данных и киоски данных 594
Проектирование хранилищ данных 596
Кубы и их архитектура 600
Агрегирование 601
Уровень агрегирования 602
Физическое хранение куба 603
Доступ к данным 604
Резюме 605
Упражнения 605

Глава 22. Службы SQL Server Analysis Services 607
Терминология служб SSAS 608
Разработка многомерного куба, используя средство BIDS 609
Создание проекта бизнес-аналитики 610

Определение источников данных 611
Создание представлений источников данных 613
Создание куба 616
Проектирование агрегирования для хранилища 617
Обработка куба 619
Просмотр куба 620
Извлечение и доставка данных 622
Обращение к данным посредством PowerPivot for Excel 624
Работа с PowerPivot for Excel 624
Обращение к данным посредством многомерных выражений 630
Безопасность служб Analysis Services SQL Server 632
Резюме 633
Упражнения 633

Глава 23. Бизнес-аналитика и Transact-SQL 635
Конструкция окна 636
Секционирование 638
Упорядочение и кадрирование 639
Расширения предложения *GROUP BY* 642
Оператор *CUBE* 643
Оператор *ROLLUP* 645
Функции группирования 646
Функция *GROUPING* 646
Функция *GROUPING_ID* 647
Наборы группирования 648
Функции запросов OLAP 649
Ранжирующие функции 649
Статистические агрегатные функции 652
Стандартные и нестандартные аналитические функции 653
Предложение *TOP* 653
Комбинация предложений *OFFSET* и *FETCH* 655
Функция *NTILE* 657
Сведение данных 658
Оператор *PIVOT* 658
Оператор *UNPIVOT* 661
Резюме 662
Упражнения 662

Глава 24. Службы отчетности SQL Server Reporting Services 665
Введение в информационные отчеты 665
Архитектура служб отчетности SQL Server Reporting Services 667
Служба Windows Reporting Services 667
Каталог отчетов 668
Диспетчер отчетов Report Manager 669
Настройка служб отчетности SQL Server Reporting Services 670
Создание отчетов 671
Создание отчетов с помощью мастера Report Server Project Wizard 673
Организация источников данных и наборов данных 673
Выбор источника данных 674

Проектирование запроса 675
Выбор типа отчета 676
оздание таблицы 677
Выбор макета отчета 678
Выбор формата отчета 679
Выбор места развертывания и завершение работы мастера 679
Просмотр результирующего набора 679
Развертывание отчета 681
Создание параметризованных отчетов 681
Управление отчетами 683
Отчеты по требованию 683
Отчеты по подписке 684
Стандартные подписки 684
Подписки, управляемые данными 685
Параметры доставки отчетов 685
Кэширование отчетов 685
Выполнение моментальных снимков 686
Резюме 686
Упражнения 687

Глава 25. Методы оптимизации для реляционной оперативной аналитической обработки 689
Секционирование данных 690
Как компонент Database Engine секционирует данные 690
Шаги для создания секционированных таблиц 691
Установление целей секционирования 691
Определение ключа секционирования и количества секций 692
Создание файловой группы для каждой секции 692
Создание функции секционирования и схемы секционирования 694
Создание секционированных индексов 696
Методы секционирования для повышения производительности системы 697
Совместное размещение таблиц 697
Использование операций поиска, поддерживающих секционирование 698
Параллельное выполнение запросов 698
Руководство по секционированию таблиц и индексов 699
Оптимизация запроса схемы типа "звезда" 700
Колончатые индексы 702
Работа с колоночными индексами 703
Создание колоночных индексов посредством Transact-SQL 703
Создание колоночных индексов посредством среды SQL Server Management Studio 704
Преимущества и недостатки колоночных индексов 704
Достоинства колоночных индексов 705
Недостатки колоночных индексов 705
Резюме 706

ЧАСТЬ V. ЗА ПРЕДЕЛАМИ РЕЛЯЦИОННЫХ ДАННЫХ 709

Глава 26. SQL Server и XML 711
Основные концепции XML 711
Требования к правильно сформированному документу XML 712

Элементы языка XML 713
Атрибуты XML 714
Пространства имен XML....715
XML и всемирная паутина 716
Родственные XML языки 716
Языки схем 717
Язык DTD 717
Язык XML Schema 719
Хранение XML-документов в SQL Server 720
Хранение XML-документов, используя тип данных *XML* 722
Хранение XML-документов с использованием декомпозиции 728
Представление данных 729
Представление XML-документов в качестве реляционных данных 729
Представление реляционных данных в качестве XML-документов 730
Режим *RAW*....730
Режим *AUTO* 731
Режим *EXPLICIT*....732
Режим *PATH*....733
Директивы 735
Запрашивание данных из XML-документов....736
Резюме 738

Глава 27. Пространственные данные....739
Введение....739
Модели для представления пространственных данных 740
Тип данных *GEOMETRY* 741
Тип данных *GEOGRAPHY* 743
Различия между типами данных *GEOMETRY* и *GEOGRAPHY*....743
Внешние форматы данных....743
Работа с данными пространственного типа 744
Работа с типом данных *GEOMETRY* 744
Работа с типом данных *GEOGRAPHY* 748
Работа с пространственными индексами....749
Отображение информации о пространственных данных....751
Новые возможности SQL Server 2012 для работы с пространственными данными....753
Новые подтипы дуг окружностей....753
Тип *CircularString*....754
Тип *CompoundCurve* 754
Тип *CurvePolygon* 755
Новые пространственные индексы 755
Индекс *auto_grid_index* для типа данных *GEOMETRY* 755
Новые системные хранимые процедуры, касающиеся пространственных данных....756
Резюме 756

Глава 28. Полнотекстовый поиск в SQL Server....759
Введение....760
Лексемы, делители текста на слова и списки стоп-слов....760
Делители текста на слова и фильтры IFilter 761
Списки стоп-слов 761

Операции с лексемами 762
Расширенные операции над словами 762
Операции задания параметров соответствия 763
Операции определения близости 763
Коэффициент релевантности 764
Как работает компонент FTS сервера SQL Server 764
Индексирование полнотекстовых данных 765
Полнотекстовое индексирование посредством Transact-SQL 765
Создание однозначного индекса 766
Разрешение полнотекстового индексирования для базы данных 766
Создание полнотекстового каталога 767
Создание полнотекстового индекса 768
Полнотекстовое индексирование с помощью среды SQL Server Management Studio ... 769
Полнотекстовые запросы 773
Предикат *FREETEXT* 773
Предикат *CONTAINS* 774
Функция *FREETEXTTABLE* 776
Функция *CONTAINSTABLE* 777
Поиск и устранение проблем с полнотекстовыми данными 779
Новые возможности SQL Server 2012 по полнотекстовому поиску 780
Настройка поиска с учетом близости 781
Свойства расширенного поиска 782
Резюме 783

Предметный указатель 784

Посвящаю моим сыновьям,
Илье и Игорю

Об авторе

Душан Петкович — профессор факультета компьютерных наук в Университете прикладных наук в городе Розенхайм (Rosenheim) в Германии. Его бестселлер *SQL Server. Руководство для начинающих* выходит уже в четвертом издании; его перу также принадлежат многочисленные статьи для журнала *SQL Server Magazine* и технические статьи для компании Embarcadero Technologies.

О техническом редакторе

Тодд Майстер (Todd Meister) работает в ИТ-индустрии свыше 15 лет. Он отредактировал свыше 75 работ, охватывающих диапазон технических областей от SQL Server до .NET Framework. Кроме технического редактирования книг, он также занимает должность главного ИТ-инженера-разработчика в университете Болла штата Индиана (Ball State University), расположенного в г. Манси (Muncie). Тодд живет в центральной Индиане со своей женой Кимберли (Kimberly) и их четырьмя смышлеными ребятами.

Благодарности

Прежде всего, я бы хотел поблагодарить моего финансирующего редактора, Венди Ринальди (Wendy Rinaldi). Венди руководила изданием всех пяти моих книг, выпущенных издательством McGraw-Hill с 1998 г. Я высоко ценю ее замечательную поддержку в течение всех этих лет. Я также хотел бы отметить важный вклад моего технического редактора, Тодда Майстера, и моего литературного редактора Билла МакМануса (Bill McManus).

Введение

Система SQL Server, в состав которой входит компонент Database Engine, службы анализа Analysis Services, службы отчетов Reporting Services, интеграционные службы Integration Services и расширение SQLXML, — является наилучшим выбором для широкого диапазона конечных пользователей и программистов баз данных, работающих над созданием бизнес-приложений, по двум причинам.

- SQL Server — несомненно, наилучшая система для операционных систем Windows, вследствие ее тесной интеграции с ними (а также вследствие низкой стоимости). Благодаря огромному и все возрастающему количеству установленных систем Windows, SQL Server является широко применяемой системой управления базами данных.
- Будучи составляющей системы реляционной базы данных, компонент Database Engine является самой легкой в использовании системой баз данных. Кроме хорошо знакомого пользовательского интерфейса, разработчики Microsoft предоставляют несколько разных инструментов для создания объектов баз данных, настраивания приложений баз данных и управления задачами системного администрирования.

В целом, SQL Server является больше, чем просто системой управления реляционными базами данных. Это платформа, которая не только позволяет управлять структурированными, полуструктурированными и неструктурированными данными, но также предоставляет комплексное, интегрированное системное программное обеспечение и программное обеспечение для аналитических исследований, которые позволяют организациям надежно управлять критически важными данными.

Цели книги

Издание *Microsoft SQL Server 2012. Руководство для начинающих* следует за четырьмя предыдущими изданиями, в которых рассматривался SQL Server 7, 2000, 2005 и 2008.

По большому счету, все пользователи SQL Server, которые хотят получить хорошее понимание этой системы управления базами данных и успешно работать с ней, найдут эту книгу очень полезной. Если вы новичок в SQL Server, но понимаете язык SQL, прочитайте *разд. "Различия между синтаксисами языка SQL и Transact-SQL"* далее в этом введении.

Эта книга направлена на пользователей всех компонентов системы SQL Server. По этой причине она разделена на несколько частей. Первые три части будут наиболее полезными для пользователей, которые хотят расширить свои знания ядра системы управления реляционными базами данных, называющегося Database Engine. Четвертая часть книги направлена на пользователей, работающих в области бизнес-аналитики (БА) (business intelligence), которые используют службы Analysis Services или реляционные расширения, касающиеся БА. В последней части книги предоставляется информация для пользователей, которые хотят изучить возможности, выходящие за рамки работы с реляционными данными, такие как технологии XML, пространственные данные и поиск данных в документах.

Новые возможности SQL Server 2012, рассматриваемые в этой книге

В SQL Server 2012 добавлено много новых возможностей, почти все из которых рассматриваются в этой книге. Для каждой возможности приводится, по крайней мере, один рабочий пример, чтобы дать лучшее понимание этой возможности. В табл. В1 дается перечень всех глав, в которых рассматриваются новые возможности, с кратким их описанием. (В таблице также приводятся возможности второго выпуска SQL Server 2008.)

Таблица В1. *Описание новых возможностей SQL Server*

Глава	Новые возможности
Глава 2	Процесс установки SQL Server 2012 в общем и использование Помощника по обновлению (Upgrade Advisor) в частности. Помощник по обновлению анализирует все компоненты, установленные в предыдущих выпусках сервера, и определяет проблемы, которые нужно устранить, прежде чем выполнять обновление до SQL Server 2012
Глава 3	В SQL Server 2012 усовершенствован отладчик среды Management Studio. В этой главе рассматриваются такие новые возможности отладчика, как указание условия для точки прерывания, счетчик срабатываний точек прерывания, фильтр точек прерываний и действие в точке прерывания, а также использование окна быстрого просмотра QuickWatch
Глава 5	В этой главе рассматривается такая новая возможность SQL Server 2012, как автономные базы данных в общем и частично автономные базы данных в частности. (Пример создания таких баз данных приводится в *примере 5.20*)
Глава 6	В этой главе вводятся два новых предложения инструкции `SELECT`: `OFFSET` и `FETCH`. Также в *разд. "Инструкция CREATE SEQUENCE"* вводятся последовательности и метод их создания
Глава 8	В SQL Server 2012 улучшена обработка исключений компонента Database Engine с помощью новой инструкции `THROW` (см. *пример 8.4*). А в *примере 8.5* показано использование предложений `OFFSET` и `FETCH` для разбиения данных на страницы на серверной стороне. Кроме этого, в *примере 8.11* показывается расширение инструкции `EXECUTE` посредством опции `RESULT SETS`

Таблица В1 *(окончание)*

Глава	Новые возможности
Глава 9	В *разд. "Представления и функции динамического управления"* этой главы рассматриваются два новых представления: `sys.dm_exec_describe_first_result_set` и `sys.dm_db_uncontained_entites` (см. *пример 9.4*)
Глава 12	В этой главе вводится инструкция `CREATE SERVER ROLE`, которая используется для создания определяемых пользователем ролей сервера. Кроме этого, в ней описывается управление авторизацией и аутентификацией для автономных баз данных (см. *главу 5*)
Глава 16	В этой главе описывается одна из самых важных новых возможностей SQL Server 2012: возможность HADR[1]. Эта возможность позволяет преодолеть недостатки зеркального отображения баз данных, а также максимизировать доступность баз данных
Глава 22	В этой главе рассматривается новый и мощный инструмент для работы с запросами аналитических данных: PowerPivot для Excel. Этот инструмент позволяет анализировать данные, используя наиболее популярное средство для этой цели — Microsoft Excel. Инструмент PowerPivot для Excel был впервые представлен в SQL Server 2008 R2
Глава 23	В этой главе описываются новые оконные функции. Сначала объясняется, с использованием примера, оператор рамки окна и его предложения (`CURRENT ROW`, `UNBOUNDED PRECEDING` и `UNBOUNDED FOLLOWING`). Потом перечисляются различия между предложениями `ROWS` и `RANGE`. Также рассматриваются новые функции `LEAD` и `LAG`
Глава 24	В этой главе рассматриваются общие наборы данных, которые были впервые введены в SQL Server 2008 R2
Глава 25	В конечной части этой главы подается полностью новый материал, в котором описываются индексы `columnstore`
Глава 27	В последнем *разд. "Новые возможности пространственных данных в SQL Server 2012"* этой главы рассматриваются три новых подтипа дуговых сегментов (составные строки, составные кривые и полигоны-кривые), новый пространственный индекс, а также две новые системные хранимые процедуры для работы с пространственными данными
Глава 28	В последнем *разд. "Новые возможности в FTS SQL Server 2012"* этой главы представлены два усовершенствования полнотекстового поиска: настройка поиска с учетом расположения и поиск по расширенным свойствам

Организация этой книги

Книга состоит из 28 глав, разделенных на пять частей.

- *Часть I. Основные понятия и установка*. Здесь рассматривается понятие систем управления базами данных и даются объяснения по установке SQL Server 2012 и его компонентов.

[1] HADR (High Availability and Disaster Recovery) — высокий уровень доступности и аварийного восстановления. — *Пер.*

Эта часть состоит из следующих глав:

- *Глава 1. Введение в системы управления реляционными базами данных*. В этой главе рассматриваются базы данных в общем и компонент Database Engine в частности. Здесь представляется понятие нормальных форм и образцовая база данных, используемая в книге. Также в этой главе дается ознакомление с соглашениями о синтаксисе, используемыми в книге.
- *Глава 2. Планирование установки и установка SQL Server*. В этой главе рассматривается первый шаг в администрировании системы: установка всей системы. Хотя установка SQL Server является прямолинейной задачей, она содержит некоторые моменты, требующие более подробного объяснения.
- *Глава 3. Компонент Management Studio сервера SQL Server*. Здесь описывается одноименный компонент сервера. Этот компонент представлен в начале книге, в случае если вы захотите создавать объекты баз данных и запрашивать данные, не зная языка SQL.

♦ *Часть II. Язык Transact-SQL.* Предназначена для конечных пользователей и прикладных программистов компонента Database Engine. Эта часть состоит из следующих глав:

- *Глава 4. Компоненты SQL*. Описываются основы наиболее важной составляющей системы управления реляционными базами данных: язык базы данных. Для всех таких систем только один язык представляет важность — язык SQL. В этой главе рассматриваются все компоненты собственного языка базы данных для SQL Server, который называется Transact-SQL. Также в этой главе описываются основные принципы и типы данных этого языка. Кроме этого, исследуются системные функции и операторы языка Transact-SQL.
- *Глава 5. Язык определения данных*. Здесь рассматриваются инструкции DDL[1] языка Transact-SQL. В зависимости от их назначения, инструкции DDL представлены в трех группах. Первая группа содержит все формы инструкции `CREATE`, которая используется для создания объектов баз данных. Во второй группе представлены все формы инструкции `ALTER`, которая применяется для модифицирования некоторых объектов баз данных. А третья группа содержит все формы инструкции `DROP`, которая используется для удаления разных объектов баз данных.
- *Глава 6. Запросы*. Глава полностью посвящена инструкции `SELECT`, наиболее важной инструкции языка Transact-SQL. В этой главе дается введение в поиск и извлечение данных из баз данных и описывается использование простых и сложных запросов. Для каждого предложения инструкции `SELECT` дается отдельное определение и объяснение со ссылкой на образцовую базу данных `sample`.
- *Глава 7. Модифицирование содержимого таблиц*. В этой главе рассматриваются четыре инструкции языка Transact-SQL, используемые для обновления

[1] DDL (Data Definition Language) — язык определения данных. — *Пер.*

данных: `INSERT`, `UPDATE`, `DELETE` и `MERGE`. Каждая из этих инструкций объясняется с использованием множественных примеров.

- *Глава 8. Хранимые процедуры и пользовательские функции.* Описываются процедурные расширения, посредством которых можно создавать мощные программы, называющиеся *хранимыми процедурами* и *пользовательскими функциями*, которые можно сохранять на сервере для многократного использования. Так как Transact-SQL — это полный вычислительный язык, все процедурные расширения являются его неотъемлемыми частями. Одни хранимые процедуры создаются пользователями, другие же предоставляются разработчиками Microsoft и называются системными хранимыми процедурами. Реализация хранимых процедур и пользовательские процедуры, использующие среду CLR (Common Language Runtime), также обсуждаются в этой главе.

- *Глава 9. Системный каталог*. Рассматривается одна из наиболее важных составляющих системы управления базами данных: системные таблицы и представления. Системный каталог содержит таблицы, в которых хранится информация об объектах баз данных и их связях. Основной особенностью системных таблиц компонента Database Engine является то, что к ним нельзя обращаться напрямую. Компонент Database Engine поддерживает несколько интерфейсов, которые можно использовать для запросов данных из системного каталога.

- *Глава 10. Индексы*. Здесь рассматривается первостепенный и наиболее мощный способ настройки приложений баз данных для улучшения времени отклика и, соответственно, производительности системы. В главе описывается роль индексов и даются рекомендации по их созданию и использованию. В конце главы представлены специальные типы индексов, поддерживаемые компонентом Database Engine.

- *Глава 11. Представления.* В этой главе объясняется создание представлений, обсуждается, с предоставлением многочисленных примеров, практическое их использование, а также рассматриваются специальные типы представлений, называющиеся *индексированными представлениями* (indexed view).

- *Глава 12. Системы обеспечения безопасности компонента Database Engine*. Предоставляются ответы на все вопросы касательно обеспечения безопасности данных, хранящихся в базе данных. Рассматриваются вопросы авторизации (предоставление санкционированного доступа к системе баз данных) и аутентификации (предоставление определенных привилегий доступа определенным пользователям). В главе обсуждаются три инструкции языка Transact-SQL — `GRANT`, `DENY` и `REVOKE`, которые предоставляют привилегии доступа к объектам базы данных, предотвращая несанкционированный доступ. В конце главы дается объяснение, каким образом можно отслеживать изменения в данных, используя компонент Database Engine.

- *Глава 13. Управление параллелизмом.* Дается исчерпывающее описание управления параллелизмом. В начале главы обсуждаются две разные модели

параллелизма, поддерживаемые компонентом Database Engine. Также дается объяснение всех инструкций языка Transact-SQL, связанных с транзакциями. Далее рассматривается использование блокировки для решения проблем управления параллелизмом. В конце главы мы узнаем, какие существуют уровни изоляции и версии строк.

- *Глава 14. Триггеры*. В этой главе описывается реализация бизнес-логики с помощью триггеров. В каждом из примеров в этой главе рассматривается проблема, с которой программисту приложений баз данных, возможно, придется столкнуться в действительности. В главе также обсуждается реализация управляемого кода для триггеров с помощью среды CLR.

♦ *Часть III. SQL Server: Администрирование системы.* В этой части рассматриваются все цели системного администрирования компонента Database Engine. Эта часть состоит из следующих глав:

- *Глава 15. Системная среда компонента Database Engine.* Обсуждаются некоторые внутренние вопросы касательно компонента Database Engine. Также предоставляется подробное описание элементов систем дискового хранения данных, системные базы данных и служебные программы компонента Database Engine.
- *Глава 16. Резервное копирование, восстановление и доступность системы*. Предоставляется обзор методов обеспечения отказоустойчивости, применяемых для реализации стратегии резервного копирования, используя среду Management Studio сервера SQL Server или соответствующие инструкции языка Transact-SQL. В первой части главы рассматриваются различные методы реализации стратегии резервного копирования. Во второй части обсуждается восстановление баз данных. А в последней подробно описываются такие варианты обеспечения доступности системы, как *отказоустойчивая кластеризация* (failover clustering), *зеркальное отображение базы данных*, *доставка журналов* (log shipping) и *возможность HADR*.
- *Глава 17. Автоматизация задач администрирования системы*. В этой главе описывается компонент Агент SQL Server, который позволяет автоматизировать определенные задачи администрирования системы, такие как резервное копирование данных и использование возможностей планирования и предупреждения для отправки извещений операторам. Также объясняется, как создавать задачи, операторы и извещения.
- *Глава 18. Репликация данных*. Здесь дается введение в предмет репликации данных, включая такие понятия, как издатель и подписчик. В главе представляются разные модели репликации; она служит руководством для настройки публикаций и подписок, используя имеющихся мастеров.
- *Глава 19. Оптимизатор запросов*. Описывается роль и принципы работы оптимизатора запросов. В главе подробно рассматривается инструментарий компонента Database Engine (инструкция `SET`, среда Management Studio сервера SQL Server и различные представления динамического управления), с по-

мощью которого можно редактировать стратегию оптимизатора. В конце главы предоставляются советы по оптимизации.

- *Глава 20. Настройка производительности*. В этой главе обсуждаются вопросы производительности и инструменты для настройки компонента Database Engine, релевантные для повседневного администрирования системы. Во вступительном материале рассматриваются вопросы оценки производительности, после чего описываются факторы, влияющие на производительность, и представляются инструменты для мониторинга SQL Server.

♦ *Часть IV. SQL Server и бизнес-аналитика*. Обсуждается бизнес-аналитика (БА) и все связанные темы. В главах этой части представляются такие службы Microsoft, как службы аналитики Analysis Services и службы отчетов Reporting Services. Кроме этого, подробно описываются возможность OLAP[1] и существующие методы оптимизации, связанные с хранением реляционных данных. Эта часть состоит из следующих глав:

- *Глава 21. Введение в бизнес-аналитику*. Дается введение в принципы *хранения данных* (data warehousing). В первой части главы объясняется разница между оперативной обработкой транзакций и хранением данных. Сохраняемые данные могут помещаться либо в *хранилище данных* (data warehouse), либо в *киоск данных* (data mart). Обсуждаются оба типа хранилищ, а их различия перечисляются во второй части главы. В конце главы рассматривается проектирование хранилища данных.
- *Глава 22. Службы SQL Server Analysis Services*. Обсуждается архитектура служб Analysis Services и основной компонент этих служб, средство разработки Business Intelligence Development Studio (BIDS). На двух примерах показывается разработка набора данных в виде многомерной структуры (куба) с помощью средства BIDS. В конце главы рассматривается несколько способов извлечения данных и доставки их пользователям.
- *Глава 23. Бизнес-аналитика и язык Transact-SQL.* В этой главе объясняется применение языка Transact-SQL для решения задач бизнес-аналитики. Рассматривается конструкция окна, включая его разделение, упорядочивание и обрамление, операторы `CUBE` и `ROLLUP`, функции ранжирования, предложение `TOP n`, а также реляционный оператор `PIVOT`.
- *Глава 24. Службы SQL Server Reporting Services.* Описывается решение Microsoft для создания бизнес-отчетности. Этот компонент используется для проектирования и использования отчетов. В главе обсуждается среда разработки для проектирования и создания отчетов, а также показываются различные способы доставки установленных отчетов.
- *Глава 25. Методы оптимизации реляционной оперативной аналитической обработки.* Здесь описываются три конкретных метода оптимизации, которые можно применить особенно в области бизнес-аналитики: разделение

[1] OLAP (Online Analytical Processing) — оперативная аналитическая обработка. — *Пер.*

данных, оптимизация соединения типа "звезда" (star join) и индексы `columnstore`. Также рассматривается метод разделения данных, называющийся *секционирование по диапазонам* (range partitioning). Объясняется роль растровых фильтров в оптимизации соединений типа "звезда". В конце главы обсуждается использование индексов `columstore`. Рассматривается создание таких индексов и их использование для улучшения производительности конкретной группы аналитических запросов.

- *Часть V. За пределами реляционных данных.* Эта часть содержит следующие главы:
  - *Глава 26. SQL Server и XML*. Обсуждается SQLXML, набор типов данных и функций от Microsoft, поддерживающий XML в SQL Server, заполняя пробел между XML и реляционными данными. В начале главы представляется стандартный тип данных, называющийся XML, и объясняется извлечение сохраненных XML-документов. Дальше подробно рассматривается представление реляционных данных в виде документов XML. В конце главы дается описание методов для запроса XML-данных.
  - *Глава 27. Пространственные данные.* В этой главе обсуждаются пространственные данные и два разных типа данных (`GEOMETRY` и `GEOGRAPHY`), с помощью которых можно создавать пространственные данные. Также рассматривается использование нескольких стандартных функций для работы с пространственными данными.
  - *Глава 28. Полнотекстовый поиск в SQL Server*. В начале этой главы обсуждаются общие понятия предмета полнотекстового поиска. Далее описываются общие шаги, требуемые для создания полнотекстового индекса, и демонстрируется применение этих шагов, сначала используя язык Transact-SQL, а потом среду SQL Server Management Studio. В конце главы рассматриваются полнотекстовые запросы. Дается описание двух предикатов и двух строчных функций, которые можно использовать для полнотекстового поиска. Для этих предикатов и функций дается несколько примеров, чтобы показать, как с их помощью можно решать конкретные проблемы, связанные с расширенными операциями на документах.

В конце почти всех глав даются упражнения, которые можно использовать, чтобы улучшить свои знания содержимого соответствующей главы. Ответы на все упражнения даются как на веб-сайте издательства McGraw-Hill Professional (**www.mhprofessional.com/computingdownload**), так и на моей домашней странице (**www.fh-rosenheim.de/~petkovic**).

Изменения по сравнению с предыдущими изданиями

Если вы знакомы с предыдущим изданием этой книги, *Microsoft SQL Server 2008. Руководство для начинающих*, вы должны увидеть, что текущее издание подверглось значительным изменениям по сравнению с предыдущим. Чтобы облегчить ис-

пользование книги, я разделил некоторые темы на отдельные, новые главы. Например, полнотекстовый поиск всесторонне рассматривается в полностью новой *главе 28*. В табл. B2 дается краткое изложение *значительных* структурных изменений в книге (незначительные изменения в этой таблице не указываются).

Таблица B2. *Значимые изменения в книге*

Глава	**Изменения**
Глава 4	В полностью новом *разд. "Опции хранения"* описываются две разные опции хранения, доступные, начиная с SQL Server 2008: `FILESTREAM` и *разреженные столбцы* (sparse columns). Опция хранения `FILESTREAM` поддерживает управление большими объектами, которые сохраняются в файловой системе NTFS, в то время как разреженные столбцы помогают минимизировать объем требуемого хранилища. (Эти столбцы позволяют оптимизировать хранение столбцов, большинство значений которых равны `NULL`)
Глава 7	Инструкции Transact-SQL для модифицирования данных `TRUNCATETABLE` и `MERGE` теперь описываются вместе, в конечном *разд. "Прочие инструкции и предложения T-SQL для модифицирования данных"* этой главы
Глава 10	Все существующие специальные типы индексов перечислены в последнем *разд. "Специальные типы индексов"* этой главы. Некоторые типы рассматриваются в этой главе, а для других даются перекрестные ссылки на главы, в которых они описываются
Глава 15	Инфраструктура Declarative Management Framework, которая в предыдущем издании книги рассматривалась в *главе 16*, теперь называется Система управления на основе политик (Policy-Based Management) и рассматривается в конце *главы 15*. (**Примечание.** *Глава 16 "Управление экземплярами и содержание баз данных"* предыдущего издания была убрана из данного издания, и ее материал, имеющий отношение к SQL Server 2012, был распределен среди других глав. Главы с 17 по 26 включительно предыдущего издания теперь перенумерованы в главы 16 по 25 соответственно. Главы в левом столбце этой таблицы перечислены в порядке новой нумерации)
Глава 16	Рассмотрение *Мастера планов обслуживания* из *главы 16* предыдущего издания перенесено в этом издании в *главу 16* (которая в предыдущем издании была *главой 17*)
Глава 18	Структура этой главы подверглась значительным изменениям. Методы для распределения данных теперь упрощены и рассматриваются в начале главы
Глава 19	Эта глава теперь содержит новый раздел, называющийся *Кэширование планов*. Этот раздел был улучшен новым примером, иллюстрирующим, как можно влиять на исполнение запросов
Глава 20	Во все разделы, касающиеся мониторинга системных ресурсов (центральный процессор, ввод/вывод и сеть), было добавлено по несколько примеров по представлениям динамического управления
Глава 22	Эта глава (которая в предыдущем издании была *главой 23*) претерпела значительные изменения. Добавлен новый главный *разд. "Извлечение и доставка данных"*, в котором представлен инструмент PowerPivot для Excel и описывается язык *MDX*[1]. Кроме этого, добавлен раздел по обеспечению безопасности служб Analysis Services сервера SQL Server

[1] MDX (Multidimensional Expressions) — язык запросов к многомерным базам данных. — *Пер.*

Таблица В2 *(окончание)*

Глава	Изменения
Глава 23	Прежний *разд. "Упорядочивание"* заменен новым *"Упорядочивание и кадрирование"*
Глава 24	Добавлен новый *разд. "Управление отчетами"*, в котором описывается доставка отчетов
Глава 25	Кроме новой темы, *Индексы columnstore*, *разд. "Оптимизация соединения типа "звезда""* усовершенствован добавлением нескольких примеров
Глава 26	*Глава 27 "Обзор XML"* и *глава 28 "SQL Server и XML"* из предыдущего издания были упрощены и объединены в одну, под заглавием *"SQL Server и XML"*. Были добавлены два основных раздела, в которых описываются все возможности, касающиеся представления и извлечения данных
Глава 27	Эта глава, которая в предыдущем издании была *главой 29*, была переписана практически заново, чтобы предоставить более обширное рассмотрение пространственных данных
Глава 28	Это новая глава, добавленная в этом издании, в которой рассматривается полностью новая тема: полнотекстовый поиск в SQL Server

Различия между синтаксисами языка SQL и Transact-SQL

Язык реляционной базы данных Transact-SQL системы SQL Server обладает несколькими нестандартными свойствами, которые неизвестны пользователям, знакомых только с языком SQL.

- В то время как в языке SQL точка с запятой (;) применяется для разделения инструкций SQL в группе инструкций (отсутствие точки с запятой в таких случаях обычно вызывает сообщение об ошибке), использование точки с запятой в языке Transact-SQL не обязательно.
- В языке Transact-SQL используется инструкция `GO`. Эта нестандартная инструкция обычно используется для разделения групп инструкций одна от другой, в то время как некоторые инструкции языка Transact-SQL (такие как `CREATE TABLE`, `CREATE INDEX` и т. п.) должны быть единственной инструкцией в группе.
- Инструкция `USE`, которая в этой книге используется очень часто, меняет контекст базы данных на указанный. Например, инструкция `USE sample` означает, что последующие инструкции будут исполняться в контексте базы данных `sample`.

Работа с образцовыми базами данных

В этом издании книги используется три образцовых базы данных:

- база данных этой книги — `sample`;
- база данных корпорации Microsoft `Adventure Works`;
- база данных корпорации Microsoft `AdventureWorksDW`.

Для книги, предназначенной для начинающих, требуется простая база данных, которую могут с легкостью понимать все читатели. По этой причине я применил очень простую концепцию для моей базы данных `sample`: она состоит всего лишь из четырех таблиц, каждая из которых содержит несколько строк. Но в то же самое время она достаточно сложная для демонстрации сотен примеров, представленных в материале книги. База данных `sample` представляет компанию посредством отделов (department) и сотрудников (employee). Каждый сотрудник состоит только в одном отделе, а отдел может содержать одного или нескольких сотрудников. Сотрудники работают над проектами (project): в любое время каждый сотрудник занят одновременно в одном или нескольких проектах, а над каждым проектом может работать один или несколько сотрудников.

Далее приводятся таблицы базы данных `sample` (табл. В3—В6).

Таблица В3. *Таблица отделов* `department` *(отдел)*

dept_no	dept_name	location
d1	Research	Dallas
d2	Accounting	Seattle
d3	Marketing	Dallas

Таблица В4. *Таблица сотрудников* `employee` *(сотрудники)*

emp_no	emp_fname	emp_lname	dept_no
25348	Matthew	Smith	d3
10102	Ann	Jones	d3
18316	John	Barrimore	dl
29346	James	James	d2
9031	Elsa	Bertoni	d2
2581	Elke	Hansel	d2
28559	Sybill	Moser	d1

Таблица В5. *Таблица проектов* `project` *(проекты)*

project_no	project_name	budget
p1	Apollo	120000
p2	Gemini	95000
p3	Mercury	185600

Таблица В6. *Таблица привязки сотрудников к проектам* `works_on`

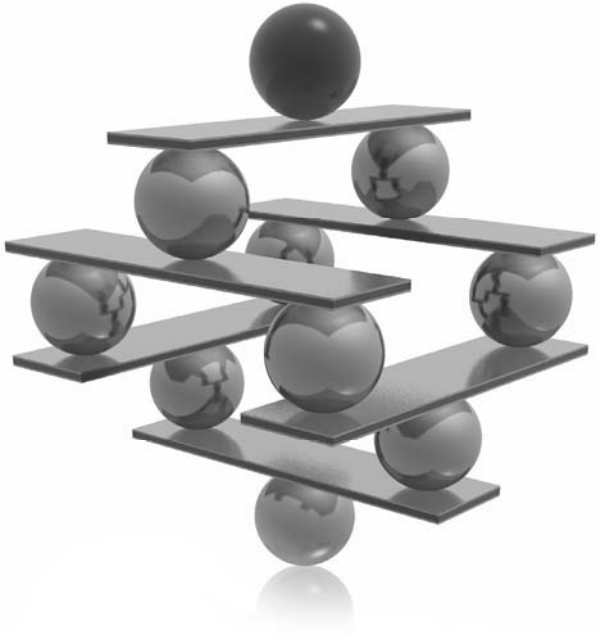
emp_no	project_no	Job	enter_date
10102	p1	Analyst	2006.10.1
10102	p3	Manager	2008.1.1
25348	p2	Cork	2007.2.15
18316	p2	NULL	2007.6.1
29346	p2	NULL	2006.12.15
2581	p3	Analyst	2007.10.15
9031	p1	Manager	2007.4.15
28559	p1	NULL	2007.8.1
28559	p2	Clerk	2008.2.1
9031	p3	Clerk	2006.11.15
29346	p1	Clerk	2007.1.4

Базу данных `sample` можно загрузить с веб-сайта издательства McGraw-Hill Professional (**www.mhprofessional.com/computingdownload**) или с моей домашней страницы (**www.fh-rosenheim.de/~petkovic**). С моей страницы также можно загрузить все примеры и решения для упражнений.

Хотя базу данных `sample` можно использовать и для многих примеров в книге, для некоторых примеров требуются таблицы с большим количеством строк (например, для демонстрации возможностей оптимизации). Для этих целей используются две базы данных корпорации Microsoft: `Adventure Works` и `AdventureWorksDW`. Обе эти базы данных можно загрузить с веб-сайта CodePlex корпорации Microsoft (**www.codeplex.com/MSFTDBProdSamples**).

Часть I

Основные понятия и установка



Глава 1

Введение в системы реляционных баз данных

- **Обзор систем баз данных**
- **Системы реляционных баз данных**
- **Проектирование баз данных**
- **Соглашения о синтаксисе**

В этой главе приводится общее описание систем баз данных. Вначале рассматривается, что собой представляет система баз данных и составляющие ее компоненты. Дается краткое описание каждого компонента со ссылкой на главу, в которой предоставлено подробное описание соответствующего компонента. Второй основной раздел этой главы отводится системам реляционных баз данных. В нем рассматриваются свойства систем реляционных баз данных и используемый в этих системах язык SQL (Structured Query Language — язык структурированных запросов).

Обычно, прежде чем приступать к реализации базы данных, нужно ее спроектировать, включая все ее объекты. В третьем основном разделе этой главы рассматривается усовершенствование баз данных с помощью нормальных форм, а также представляется модель типа "сущность — связь", посредством которой можно концептуализировать все сущности и связи между ними. А в последнем разделе рассматриваются соглашения о синтаксисе, используемые в книге.

Обзор систем баз данных

Система баз данных — это общий набор различных программных компонентов баз данных и собственно баз данных, содержащий следующие составляющие:

- прикладные программы баз данных;
- клиентские компоненты;

- сервер(ы) баз данных;
- собственно базы данных.

Прикладная программа баз данных представляет собой программное обеспечение специального назначения, разработанное и реализованное пользователями или сторонними компаниями-разработчиками ПО. В противоположность, *клиентские компоненты* — это программное обеспечение баз данных общего назначения, разработанное и реализованное компанией-разработчиком базы данных. С помощью клиентских компонентов пользователи могут получить доступ к данным, хранящимся на локальном или удаленном компьютере.

Сервер баз данных выполняет задачу управления данными, хранящимися в базе данных. Клиенты взаимодействуют с сервером баз данных, отправляя ему запросы. Сервер обрабатывает каждый полученный запрос и отправляет результаты соответствующему клиенту.

В общих чертах, *базу данных* можно рассматривать с двух точек зрения — пользователя и системы базы данных. Пользователи видят базу данных как набор логически связанных данных, а для системы баз данных это просто последовательность байтов, которые обычно хранятся на диске. Хотя это два полностью разных взгляда, между ними есть что-то общее: система баз данных должна предоставлять не только интерфейс, позволяющий пользователям создавать базы данных и извлекать или модифицировать данные, но также системные компоненты для управления хранимыми данными. Поэтому система баз данных должна предоставлять следующие возможности:

- разнообразные пользовательские интерфейсы;
- физическую независимость данных;
- логическую независимость данных;
- оптимизацию запросов;
- целостность данных;
- управление параллелизмом;
- резервное копирование и восстановление;
- безопасность баз данных.

Все эти возможности вкратце описываются в следующих далее разделах.

Разнообразные пользовательские интерфейсы

Большинство баз данных проектируются и реализовываются для работы с ними разных типов пользователей, имеющих разные уровни знаний. По этой причине система баз данных должна предоставлять несколько отдельных пользовательских интерфейсов. Пользовательский интерфейс может быть графическим или текстовым. В графических интерфейсах ввод осуществляется посредством клавиатуры или мыши, а вывод реализуется в графическом виде на монитор. Разновидностью текстового интерфейса, часто используемого в системах баз данных, является ин-

терфейс командной строки, с помощью которого пользователь осуществляет ввод посредством набора команд на клавиатуре, а система отображает вывод в текстовом формате на мониторе.

Физическая независимость данных

Физическая независимость данных означает, что прикладные программы базы данных не зависят от физической структуры данных, хранимых в базе данных. Эта важная особенность позволяет изменять хранимые данные без необходимости вносить какие-либо изменения в прикладные программы баз данных. Например, если данные были сначала упорядочены по одному критерию, а потом этот порядок был изменен по другому критерию, изменение физических данных не должно влиять на существующие приложения баз данных или ее *схему* (описание базы данных, созданное языком определения данных системы базы данных).

Логическая независимость данных

При обработке файлов, используя традиционные языки программирования, файлы объявляются прикладными программами, поэтому любые изменения в структуре файла обычно требуют внесения соответствующих изменений во все использующие его программы. Системы баз данных предоставляют логическую независимость файлов, т. е., иными словами, логическую структуру базы данных можно изменять без необходимости внесения каких-либо изменений в прикладные программы базы данных. Например, добавление атрибута к уже существующей в системе баз данных структуре объекта с именем `PERSON` (например, адрес) вызывает необходимость модифицировать только логическую структуру базы данных, а не существующие прикладные программы. (Однако приложения потребуют модифицирования для использования нового столбца.)

Оптимизация запросов

Большинство систем баз данных содержат подкомпонент, называющийся *оптимизатором*, который рассматривает несколько возможных стратегий исполнения запроса данных и выбирает из них наиболее эффективную. Выбранная стратегия называется *планом исполнения запроса*. Оптимизатор принимает решение, принимая во внимание такие факторы, как размер таблиц, к которым направлен запрос, существующие индексы и логические операторы (`AND`, `OR` или `NOT`), используемые в предложении `WHERE`. Тема оптимизатора рассматривается подробно в *главе 19*.

Целостность данных

Одной из стоящих перед системой баз данных задач является идентифицировать логически противоречивые данные и не допустить их помещения в базу данных. (Примером таких данных будет дата "30 февраля" или время "5:77:00".) Кроме этого, для большинства реальных задач, которые реализовываются с помощью систем баз данных, существуют *ограничения для обеспечения целостности* (integrity

constraints), которые должны выполняться для данных. (В качестве примера ограничения для обеспечения целостности можно назвать требование, чтобы табельный номер сотрудника был пятизначным целым числом.) Обеспечение целостности данных может осуществляться пользователем в прикладной программе или же системой управления базами данных. До максимально возможной степени эта задача должна осуществляться посредством *СУБД* (системы управления базами данных). Подробно тема целостности данных рассматривается в двух главах: декларативная целостность в *главе 5*, а процедурная целостность — в *главе 14*.

Управление параллелизмом

Система баз данных представляет собой многопользовательскую систему программного обеспечения, что означает одновременное обращение к базе данных множественных пользовательских приложений. Поэтому каждая система баз данных должна обладать каким-либо типом механизма, обеспечивающим управление попытками модифицировать данные несколькими приложениями одновременно. Далее приводится пример проблемы, которая может возникнуть, если система баз данных не оснащена таким механизмом управления:

1. На общем банковском счете № 4711 в банке *X* имеется $2000.
2. Владельцы этого счета, госпожа *А* и господин *Б*, идут в разные отделения банка и *одновременно* снимают со счета по $1000 каждый.
3. Сумма, оставшаяся на счету № 4711 после этих транзакций, должна быть $0, и ни в коем случае не $1000.

Все системы баз данных должны иметь необходимые механизмы для обработки подобных ситуаций. Управление параллелизмом рассматривается подробно в *главе 13*.

Резервное копирование и восстановление

Система баз данных должна быть оснащена подсистемой для восстановления после ошибок в программном и аппаратном обеспечении. Например, если в процессе обновления 100 строк таблицы базы данных происходит сбой, то подсистема восстановления должна выполнить откат всех выполненных обновлений, чтобы обеспечить непротиворечивость данных. Тема резервного копирования и восстановления подробно рассматривается в *главе 16*.

Безопасность баз данных

Наиболее важными понятиями безопасности баз данных являются аутентификация и авторизация. *Аутентификация* — это процесс проверки подлинности учетных данных пользователя, чтобы не допустить использования системы несанкционированными пользователями. Аутентификация наиболее часто реализуется, требуя, чтобы пользователь вводил свое имя пользователя и пароль. Система проверяет достоверность этой информации, чтобы решить, имеет ли данный пользователь

право входа в систему или нет. Этот процесс можно усилить применением шифрования.

Авторизация — это процесс, применяемый к пользователям, уже получившим доступ к системе, чтобы определить их права на использование определенных ресурсов. Например, доступ к информации о структуре базы данных и системному каталогу определенной сущности могут получить только пользователи, имеющие на это право. Тема аутентификации и авторизации рассматривается подробно в *главе 12*.

Системы реляционных баз данных

Компонент Database Engine сервера Microsoft SQL Server является системой реляционных баз данных. Понятие систем реляционных баз данных было впервые введено в 1970 г. Эдгаром Ф. Коддом в статье *"Реляционная модель данных для больших банков данных совместного использования"* (Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks). В отличие от предшествующих систем баз данных (сетевых и иерархических), *реляционные системы баз данных* основаны на реляционной модели данных, обладающей мощной математической теорией.

ПРИМЕЧАНИЕ

Модель данных — это набор концепций, взаимосвязей между ними и их ограничений, которые используются для представления данных в реальной задаче.

Центральным понятием реляционной модели данных является таблица. Поэтому, с точки зрения пользователя, реляционная база данных содержит только таблицы и ничего больше. Таблицы состоят из столбцов (одного или нескольких) и строк (ни одной или нескольких). Каждое пресечение строки и столбца таблицы всегда содержит ровно одно значение данных.

Работа с демонстрационной базой данных книги

Используемая в этой книге база данных `sample` представляет некую компанию, состоящую из отделов (department) и сотрудников (employee). Каждый сотрудник принадлежит только одному отделу, а отдел может содержать одного или нескольких сотрудников. Сотрудники работают над проектами (project): в любое время каждый сотрудник занят одновременно в одном или нескольких проектах, а над каждым проектом может работать один или несколько сотрудников.

Эти информация представлена в базе данных `sample` посредством четырех таблиц:

- `department`;
- `employee`;
- `project`;
- `works_on`.

Организация этих таблиц показана в табл. 1.1—1.4.

Таблица `department` представляет все отделы компании. Каждый отдел обладает следующими атрибутами (столбцами):

`department (dept_no, dept_name, location)`

Атрибут `dept_no` представляет однозначный номер каждого отдела, атрибут `dept_name` — его название, а атрибут `location` — расположение.

Таблица 1.1. *Таблица отделов* `department`

dept_no	dept_name	location
d1	Research	Dallas
d2	Accounting	Seattle
d3	Marketing	Dallas

Таблица `employee` представляет всех работающих в компании сотрудников. Каждый сотрудник обладает следующими атрибутами (столбцами):

`employee (emp_no, emp_fname, emp_lname, dept_no)`

Атрибут `emp_no` представляет однозначный табельный номер каждого сотрудника, атрибуты `emp_fname` и `emp_lname` — имя и фамилию сотрудника соответственно, а атрибут `dept_no` — номер отдела, в котором работает сотрудник.

Таблица 1.2. *Таблица сотрудников* `employee`

emp_no	emp_fname	emp_lname	dept_no
25348	Matthew	Smith	d3
10102	Ann	Jones	d3
18316	John	Barrimore	d1
29346	James	James	d2
9031	Elke	Hansel	d2
2581	Elsa	Bertoni	d2
28559	Sybill	Moser	d1

Все проекты компании представлены в таблице проектов `project`, состоящей из следующих столбцов (атрибутов):

`project (project_no, project_name, budget)`

В столбце `project_no` указывается однозначный номер проекта, а в столбцах `project_name` и `budget` — название и бюджет проекта соответственно.

Таблица 1.3. *Таблица проектов* `project`

project_no	project_name	budget
p1	Apollo	120000
p2	Gemini	95000
p3	Mercury	186500

В таблице `works_on` указывается связь между сотрудниками и проектами. Она состоит из следующих столбцов:

`works_on (emp_no, project_no, job, enter_date)`

В столбце `emp_no` указывается табельный номер сотрудника, а в столбце `project_no` — номер проекта, в котором он принимает участие. Комбинация значений этих двух столбцов всегда однозначна. В столбцах `job` и `enter_date` указывается должность и начало работы сотрудника в данном проекте соответственно.

Таблица 1.4. *Таблица привязки сотрудников к проектам* `works_on`

emp_no	project_no	job	enter_date
10102	p1	Analyst	2006.10.1
10102	p3	Manager	2008.1.1
25348	p2	Clerk	2007.2.15
18316	p2	NULL	2007.6.1
29346	p2	NULL	2006.12.15
2581	p3	Analyst	2007.10.15
9031	p1	Manager	2007.4.15
28559	p1	NULL	2007.8.1
28559	p2	Clerk	2008.2.1
9031	p3	Clerk	2006.11.15
29346	p1	Clerk	2007.1.4

На примере базы данных `sample` можно описать некоторые основные свойства реляционных систем баз данных.

- Строки таблицы не организованы в каком-либо определенном порядке.
- Также не организованы в каком-либо определенном порядке столбцы таблицы.
- Каждый столбец таблицы должен иметь однозначное имя в любой данной таблице. Но разные таблицы могут содержать столбцы с одним и тем же именем. Например, таблица `department` содержит столбец `dept_name` и столбец с таким же именем имеется в таблице `employee`.

- Каждый элемент данных таблицы должен содержать одно значение. Это означает, что любая ячейка на пресечении строк и столбцов таблицы никогда не содержит какого-либо набора значений.
- Каждая таблица содержит, по крайней мере, один столбец, значения которого определяют такое свойство, что никакие две строки не содержат одинаковой комбинации значений для всех столбцов таблицы. В реляционной модели данных такой столбец называться *потенциальным ключом* (candidate key). Если таблица содержит несколько потенциальных ключей, разработчик указывает один из них, как *первичный ключ* (primary key) данной таблицы. Например, первичным ключом таблицы `department` будет столбец `dept_no`, а первичными ключами таблиц `employee` и `project` будут столбцы `emp_no` и `project_no` соответственно. Наконец, первичным ключом таблицы `works_on` будет комбинация столбцов `emp_no` и `project_no`.
- Таблица никогда не содержит одинаковых строк. Но это свойство существует только в теории, т. к. компонент Database Engine и все другие реляционные системы баз данных допускают существование в таблице одинаковых строк.

SQL — язык реляционной базы данных

Язык реляционной базы данных в системе SQL Server называется Transact-SQL. Это разновидность самого значимого на сегодняшний день языка базы данных — языка *SQL* (Structured Query Language — язык структурированных запросов). Происхождение языка SQL тесно связано с проектом, называемым System R, разработанным и реализованным компанией IBM еще в начале 80-х годов прошлого столетия. Посредством этого проекта было продемонстрировано, что, используя теоретические основы работы Эдгара Ф. Кодда, возможно создание системы реляционных баз данных.

В отличие от традиционных языков программирования, таких как C, C++ и Java, язык SQL является *множество-ориентированным* (set-oriented). Разработчики языка также называют его *запись-ориентированным* (record-oriented). Это означает, что в языке SQL можно запрашивать данные из нескольких строк одной или нескольких таблиц, используя всего лишь одну инструкцию. Это одно из наиболее важных преимуществ языка SQL, позволяющее использовать этот язык на логически более высоком уровне, чем традиционные языки программирования.

Другим важным свойством языка SQL является его непроцедурность. Любая программа, написанная на процедурном языке (C, C++, Java), пошагово описывает, *как* выполнять определенную задачу. В противоположность этому, язык SQL, как и любой другой непроцедурный язык, описывает, *что* хочет пользователь. Таким образом, ответственность за нахождение подходящего способа для удовлетворения запроса пользователя лежит на системе.

Язык SQL содержит два подъязыка: язык описания данных *DDL* (Data Definition Language) и язык обработки данных *DML* (Data Manipulation Language). Инструкции языка DDL также применяются для описания схем таблиц баз данных. Язык DDL содержит три общие инструкции SQL: `CREATE`, `ALTER` и `DROP`. Эти инструкции

используются для создания, изменения и удаления, соответственно, объектов баз данных, таких как базы данных, таблицы, столбцы и индексы. Более подробно эти инструкции рассматриваются в *главе 5*.

В отличие от языка DDL, язык DML охватывает все операции по манипулированию данными. Для манипулирования базами данных всегда применяются четыре общие операции: извлечение, вставка, удаление и модифицирование данных. Инструкция `SELECT` для извлечения данных подробно рассматривается в *главе 6*, а инструкции `INSERT`, `DELETE` и `UPDATE` — в *главе 7*.

Проектирование базы данных

Проектирование базы данных является очень важным этапом в жизненном цикле базы данных, который предшествует всем другим этапам, за исключением этапа сбора и анализа требований. Если создать проект базы данных, руководствуясь единственно интуицией и без какого-либо плана, то получившаяся база данных, скорей всего, не будет отвечать требованиям пользователя в плане производительности.

Другим последствием плохого проекта базы данных будет чрезмерная *избыточность данных* (data redundancy), которая сама по себе имеет два недостатка: наличие аномалий в данных и повышенные требования дискового пространства.

Нормализацией данных (data normalization) называется процесс, в котором таблицы базы данных проверяются на наличие определенных зависимостей между столбцами таблицы. Если таблица содержит такие зависимости, она разбивается на несколько таблиц (обычно две), что позволяет избавиться от зависимостей между столбцами. Если одна из этих таблиц все еще содержит зависимости, процесс нормализации повторяется до тех пор, пока все зависимости не будут разрешены.

Процесс удаления избыточных данных в таблицах основан на теории функциональных зависимостей. *Функциональная зависимость* означает, что по значению одного столбца можно всегда однозначно определить соответствующее значение другого столбца. (То же самое действительно и для групп столбцов.) Функциональная зависимость между столбцами `A` и `B` обозначается как `A` → `B`, что означает, что значение в столбце `B` можно всегда определить по соответствующему значению в столбце `A`. (Данная формула читается как "B функционально зависит от A".)

В примере 1.1 демонстрируется функциональная зависимость между двумя атрибутами таблицы `employee` базы данных `sample`.

Пример 1.1

```
emp_no → emp_lname
```

Имея однозначное значение табельного номера сотрудника, можно определить его фамилию (и все прочие соответствующие атрибуты). Такой тип функциональной

зависимости, когда столбец зависит от первичного ключа таблицы, называется *тривиальной* функциональной зависимостью.

Другой тип функциональной зависимости называется *многозначной зависимостью*. В отличие от только что описанной функциональной зависимости, многозначная зависимость задается для многозначных атрибутов. Это означает, что, используя известное значение одного атрибута (столбца), можно однозначно определить *набор значений* другого многозначного атрибута. Многозначная зависимость обозначается как →→.

В примере 1.2 показана многозначная зависимость, действующая между двумя атрибутами объекта `BOOK`.

Пример 1.2

```
ISBN →→ Authors
```

Значение ISBN книги всегда определяет всех ее авторов. Поэтому атрибут `Authors` многозначно зависит от атрибута `ISBN`.

Нормальные формы

Нормальные формы применяются в процессе нормализации данных и, следственно, при проектировании баз данных. Теоретически, существует, по крайней мере, пять разных нормальных форм, из которых наиболее важными для практического применения являются первые три. Третью нормальную форму таблицы можно получить, выполнив проверку на первую и вторую нормальные формы в качестве промежуточных этапов. Цель получения хорошего проекта базы данных обычно достигается, если все таблицы базы данных находятся в третьей нормальной форме.

ПРИМЕЧАНИЕ

Многозначная зависимость применяется для проверки таблиц на четвертую нормальную форму. Поэтому данный тип зависимости в этой книге не будет использоваться.

Первая нормальная форма

Первая нормальная форма (1NF) означает, что таблица не содержит многозначных или составных атрибутов. (Составной атрибут содержит другие атрибуты, поэтому его можно разделить на меньшие части.) По определению, все реляционные таблицы находятся в первой нормальной форме, т. к. значение любой ячейки должно быть *атомарным*, т. е. однозначным.

В табл. 1.5 демонстрируется первая нормальная форма (1NF) на примере таблицы `works_on` базы данных `sample`.

Строки этой таблицы можно сгруппировать, используя табельный номер сотрудника.

Таблица 1.5. *Часть таблицы* `works_on`

emp_no	project_no	.....
10102	p1	.....
10102	p3	.....
.....	.....	.....

Получившаяся табл. 1.6 уже не будет в первой нормальной форме, поскольку столбец `project_no` содержит набор значений `(p1, p3)`.

Таблица 1.6. *Эта "таблица" уже не в первой нормальной форме*

emp_no	project_no	.....
10102	(p1, p3)	.....
.....	.....	.....

Вторая нормальная форма

Таблица находится во второй нормальной форме (2NF), если она в первой нормальной форме (1NF) и не содержит ключевых столбцов, зависимых от части столбцов первичного ключа этой таблицы. Это означает, что если `(A,B)` является комбинацией двух столбцов таблицы, составляющих первичный ключ, тогда таблица не содержит столбцов, зависящих только от `A` или только от `B`.

Например, в табл. 1.7 показана таблица `works_on1`, которая идентична таблице `works_on`, за исключением наличия дополнительного столбца `dept_no`.

Таблица 1.7. *Таблица* `works_on1`

emp_no	project_no	job	enter_date	dept_no
10102	Pi	Analyst	2006.10.1	d3
10102	p3	Manager	2008.1.1	d3
25348	p2	Clerk	2007.2.15	d3
18316	p2	NULL	2007.6.1	dl

Первичным ключом этой таблицы является комбинация столбцов `emp_no` и `project_no`. Столбец `dept_no` этой таблицы зависит от части первичного ключа — столбца `emp_on` — и не зависит от второй его части, т. е. столбца `project_no`. Поэтому эта таблица не находится во второй нормальной форме. (Исходная же таблица, `works_on`, находится во второй нормальной форме.)

ПРИМЕЧАНИЕ

Любая таблица, первичный ключ которой состоит из одного столбца, всегда находится во второй нормальной форме.

Третья нормальная форма

Таблица находится в третьей нормальной форме (3NF), если она имеется во второй нормальной форме (2NF) и отсутствуют функциональные зависимости между неключевыми столбцами. Например, таблица `employee1` (табл. 1.8), которая точно такая же, как и таблица `employee`, за исключением дополнительного столбца `dept_name`, не находится в третьей нормальной форме, т. к. для каждого известного значения столбца `dept_no` можно точно определить соответствующее значение столбца `dept_name`. (Исходная таблица `employee` находится в третьей нормальной форме, как и все остальные таблицы базы данных `sample`.)

Таблица 1.8. *Таблица* `employee1`

emp_no	emp_fname	emp_lname	dept_no	dept_name
25348	Matthew	Smith	d3	Marketing
10102	Ann	Jones	d3	Marketing
18316	John	Barrimore	d1	Research
29346	James	James	d2	Accounting

Модель "сущность — отношение"

Можно с легкостью спроектировать базу данных с единственной таблицей, содержащей все данные. Основным недостатком такой базы данных будет высокий уровень избыточности данных. Например, единственная таблица базы данных, содержащая все данные о сотрудниках и проектах, в которых они участвуют (полагая, что каждый сотрудник может одновременно работать в одном или нескольких проектах и что в каждом проекте может работать один или несколько сотрудников), будет состоять из большого числа столбцов и строк. Основным недостатком такой таблицы является трудность удержания согласованности данных по причине их повторяемости.

Модель *"сущность — отношение"* (entity-relationship (ER)) используется при проектировании реляционных баз данных с целью удаления любой избыточности данных. Основным объектом модели "сущность — отношение" является *сущность*, т. е. реальный объект. Каждая сущность обладает несколькими *атрибутами*, которые являются свойствами сущности и, таким образом, описывают ее. Атрибут может быть одного из следующих типов:

- *атомарным* (или *однозначным*). Атомарный атрибут конкретной сущности всегда представляется одним значением. Например, статус состояния лица в браке

всегда будет атомарным атрибутом. Большинство атрибутов являются атомарными;

- *многозначным*. Для конкретной сущности многозначный атрибут может иметь одно или несколько значений. Например, атрибут `Location` сущности `ENTERPRISE` является многозначным, т. к. предприятие может располагаться в нескольких местах;
- *составным*. Составные атрибуты не являются атомарными, т. к. они состоят из нескольких атомарных атрибутов. Типичным примером составного атрибута будет почтовый адрес, который состоит из таких атомарных атрибутов, как город, индекс, улица и т. п.

Сущность `PERSON` в примере 1.3 обладает несколькими атомарными атрибутами, одним составным атрибутом `Address` и многозначным атрибутом `College_degree`.

Пример 1.3

```
PERSON (Personal_no, F_name, L_name, Address(City,Zip,Street),
                                                {College_degree})
```

Каждая сущность имеет один или несколько ключевых атрибутов, которые являются атрибутами (или комбинацией двух или больше атрибутов) с однозначными значениями для каждой конкретной сущности. В примере 1.3 ключевым атрибутом сущности `PERSON` является атрибут `Personal_no`.

Помимо сущности и атрибута, еще одним базовым понятием модели "сущность — отношение" является *отношение*. Отношение существует, когда одна сущность ссылается на другую (или несколько других). Количество вовлеченных сущностей определяет степень отношения. Например, между сущностями `EMPLOYEE` и `PROJECTS` существует отношение второй степени `works_on`.

Каждое существующее отношение между двумя сущностями должно быть одним из следующих трех типов: 1:1, 1:*N* или *M*:*N*. Это свойство отношения называется *мощностью отношения* (cardinality ratio) или кардинальным числом. Например, между сущностями `DEPARTMENT` и `EMPLOYEE` существует отношение типа 1:*N*, т. к. каждый сотрудник принадлежит в точности одному отделу, который, в свою очередь, содержит одного или нескольких сотрудников. А между сущностями `PROJECT` и `EMPLOYEE` существует отношение типа *M*:*N*, поскольку в каждом проекте участвует один или больше сотрудников и каждый сотрудник одновременно участвует в одном или больше проектов.

Отношение также может иметь свои атрибуты. На рис. 1.1 показан пример диаграммы модели "модель — отношение" (ER-диаграммы), т. е. графическое описание этой модели.

В этой диаграмме сущности представлены прямоугольниками, внутри которых указывается имя сущности. Атрибуты представлены в виде овалов, и каждый атрибут прикреплен к конкретной сущности (или отношению) посредством прямой линии.

Наконец, отношения представлены в виде ромбов, а участвующие в отношении сущности соединены с ним прямыми линиями. Мощность каждого отношения указывается на соответствующей соединяющей линии.

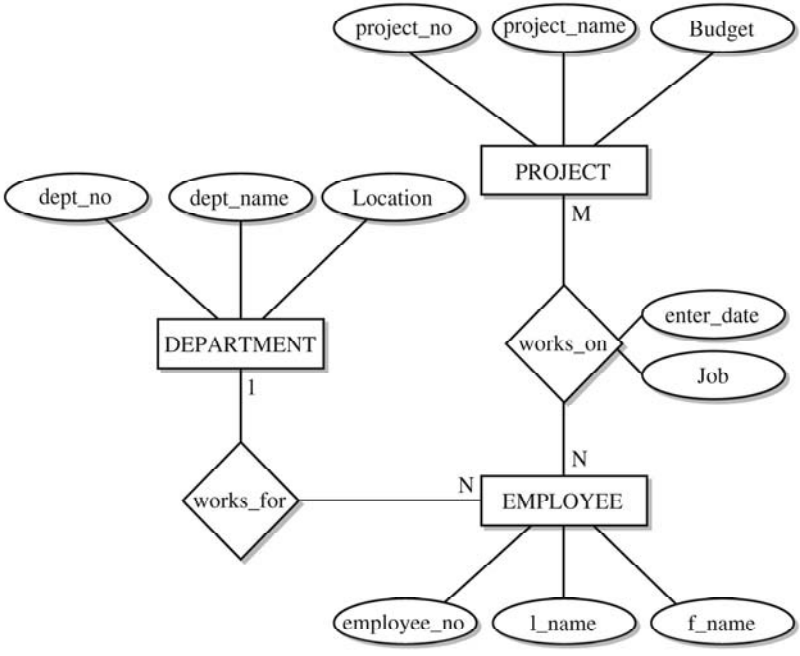


Рис. 1.1. Пример ER-диаграммы

Соглашения о синтаксисе

В этой книге для описания синтаксиса языка Transact-SQL и для выделения фрагментов текста используются соглашения, показанные в табл. 1.9.

ПРИМЕЧАНИЕ

В отличие от квадратных и фигурных скобок, которые указывают соглашения о синтаксисе, круглые скобки относятся к синтаксису инструкций и должны вводиться в обязательном порядке.

Таблица 1.9. *Соглашения о синтаксисе*

Обозначение	**Значение**
Курсив	Новые термины или выделяемые элементы
ПРОПИСНЫЕ СИМВОЛЫ	Ключевые слова Transact-SQL, например `CREATE TABLE`. Дополнительная информация о ключевых словах языка Transact-SQL представлена в *главе 5*

Таблица 1.9 *(окончание)*

Обозначение	Значение
строчные символы	Переменные в инструкциях Transact-SQL, например, CREATE TABLE *имя_таблицы* (пользователь должен заменить "имя_таблицы" собственно названием таблицы)
Var1 | var2	Выбор между элементами var1 и var2. (Пользователь может выбрать только один из элементов, разделенных вертикальной чертой)
{ }	Выбор из нескольких элементов. Пример: {*выражение* | USER | NULL}
[ ]	Необязательные элементы. Пример: [FOR LOAD]
{}...	Элементы, которые можно повторять любое количество раз. Пример: {, @param1 typ1}...
courier	Имя объекта базы данных (собственно база данных, таблицы, столбцы) в тексте
Полужирный	Наименования элементов интерфейса
По умолчанию	Значение по умолчанию всегда подчеркнуто. Пример: ALL | DISTINCT

Резюме

Все системы баз данных предоставляют следующие возможности:

- разнообразные пользовательские интерфейсы;
- физическую независимость данных;
- логическую независимость данных;
- оптимизацию запросов;
- целостность данных;
- управление параллелизмом;
- резервное копирование и восстановление;
- безопасность баз данных.

В следующей главе мы рассмотрим установку SQL Server 2012.

Упражнения

Упражнение 1.1

Что означает выражение "независимость данных" и какие существуют два типа независимости данных?

Упражнение 1.2

Что является основным концептом реляционной модели?

Упражнение 1.3

Какую часть реального мира представляет таблица `employee`? И что представляет строка в этой таблице с данными для Ann Jones?

Упражнение 1.4

Какой элемент реального мира (и по отношению к другим таблицам базы данных `sample`) представляет таблица `works_on`?

Упражнение 1.5

Пусть имеется таблица `book`, состоящая из двух столбцов — `isbn` и `title`. Полагая, что значения `isbn` являются однозначными и что все названия `title` разные, дайте ответы на следующие вопросы:

а. Является ли столбец `title` ключом таблицы?

б. Зависит ли функционально столбец `isbn` от столбца `title`?

в. Находится ли таблица `book` в третьей нормальной форме (3NF)?

Упражнение 1.6

Пусть имеется таблица `order`, содержащая столбцы `order_no`, `customer_no`, `discount`. Если столбец `customer_no` функционально зависит от столбца `order_no`, а столбец `discount` функционально зависит от столбца `customer_no`, ответьте на следующие вопросы с предоставлением подробного объяснения:

а. Является ли столбец `order_no` ключом таблицы?

б. Является ли столбец `customer_no` ключом таблицы?

Упражнение 1.7

Пусть имеется таблица `company`, содержащая столбцы `company_no` и `location`. Все компании расположены в одном или нескольких местах. В какой нормальной форме находится таблица `company`?

Упражнение 1.8

Пусть имеется таблица `supplier`, содержащая столбцы `supplier_no`, `article` и `city`. Ключом таблицы является комбинация первых двух столбцов. Каждый поставщик (supplier) поставляет несколько названий товара (article), а каждое название товара поставляется несколькими поставщиками. В каждом городе имеется только один поставщик. Дайте ответы на следующие вопросы:

а. В какой нормальной форме находится таблица `supplier`?

б. Как можно устранить существующие функциональные зависимости?

Упражнение 1.9

Пусть имеется отношение $R(\underline{A}, \underline{B}, C)$ с функциональной зависимостью $B \rightarrow C$. (Подчеркнутые атрибуты A и B создают составной ключ, а атрибут C функцио-

нально зависит от атрибута B.) В какой нормальной форме находится отношение R?

Упражнение 1.10

Пусть имеется отношение $R(\underline{A}, \underline{B}, C)$ с функциональной зависимостью $C \rightarrow B$. (Подчеркнутые атрибуты A и B создают составной ключ, а атрибут B функционально зависит от атрибута C.) В какой нормальной форме находится отношение R?

Глава 2

Планирование установки и установка SQL Server

- **Версии SQL Server**
- **Этап планирования**
- **Установка SQL Server**

В начале этой главы дается краткий обзор версий SQL Server, чтобы позволить вам определить версию, наиболее подходящую для вашей среды. Прежде чем приступать к установке этой системы базы данных, вам нужно разработать план установки. Поэтому вторая часть этой главы отведена под рассмотрение этапа планирования установки. Вначале предоставляются несколько общих рекомендаций, а потом рассматриваются шаги установки посредством центра установки SQL Server, являющегося компонентом программного обеспечения SQL Server. Собственно установка сервера базы данных SQL Server рассматривается в конечном материале главы. Установка выполняется с помощью упомянутого ранее центра установки SQL Server.

ПРИМЕЧАНИЕ

В этой главе рассматривается базовая установка SQL Server.

Версии SQL Server

При планировании установки SQL Server нужно знать, какие существуют версии этой системы, чтобы можно было выбрать наиболее подходящую для ваших условий. Корпорация Microsoft предлагает следующие версии SQL Server 2012.

- *Express Edition*. Облегченная версия SQL Server, предназначенная для разработчиков приложений. По этой причине продукт содержит базовую программу Express Manager (XM) и поддерживает интеграцию общеязыковой среды выпол-

нения CLR и собственный язык XML. Управление базой данных можно упростить с помощью компонента SQL Server Management Express для SQL Server Express, который можно бесплатно загрузить по адресу **http://msdn.microsoft.com/express**.

- *Workgroup Edition*. Эта версия предназначена для малого бизнеса и для использования на уровне отделов предприятия. В ней предоставляется поддержка реляционных баз данных, но отсутствуют средства бизнес-аналитики (БА) и возможности обеспечения высокого уровня доступности. Эта версия поддерживает системы с двумя процессорами и максимум 2 Гбайтами оперативной памяти.
- *Standard Edition*. Версия предназначена для малого и среднего бизнеса. Поддерживает системы с четырьмя процессорами и 2 Тбайтами оперативной памяти, а также содержит весь диапазон возможностей бизнес-аналитики, включая службы Analysis Services, Reporting Services и Integration Services. Эта версия не содержит многих возможностей из версии Enterprise Edition (таких как, например, отказоустойчивая кластеризация).
- *Web Edition*. Версия предназначена для поставщиков веб-хостинга. Кроме компонента Database Engine этот выпуск содержит службы отчетности Reporting Services. Поддерживается до четырех процессоров и 2 Тбайта оперативной памяти.
- *Enterprise Edition*. Специальная версия системы SQL Server, предназначенная для приложений, критичных по времени и с большим количеством пользователей. В отличие от версии Standard Edition, эта версия содержит дополнительные возможности, которые могут быть полезными для установок очень высокого уровня на оборудовании с симметричными мультипроцессорами или кластерами. Наиболее важными дополнительными возможностями версии Enterprise Edition является секционирование данных, возможность получения мгновенных снимков состояния базы данных и онлайновая поддержка баз данных.
- *Developer Edition*. Эта версия позволяет разработчикам создавать и тестировать приложения любого типа для 32- и 64-разрядных платформ SQL Server. Содержит всю функциональность версии Enterprise Edition, но лицензия разрешает использование только для разработки, тестирования и демонстрации. Каждая лицензия для версии Developer Edition дает разработчику право использовать это программное обеспечение на необходимом количестве систем. Для использования его другими разработчиками необходимо приобрести дополнительные лицензии. Для быстрого перехода к использованию в качестве производственной системы версию Developer Edition можно с легкостью обновить до версии Enterprise Edition.
- *Datacenter Edition*. Новая версия SQL Server 2009 R2, предназначенная для поддержки масштабирования наивысшего уровня. Для этой версии нет ограничений по памяти, что позволяет создавать до 25 экземпляров. Также обеспечивается поддержка до 256 логических процессоров.
- *Parallel Data Warehouse Edition*. Эта версия специализирована для хранилищ данных и поддерживает базы данных хранилищ данных размером от 10 Тбайт до

1 Пбайт (петабайт, причем 1 Пбайт = 1024 Тбайт). Для управления такими громадными базами данных в ней используется архитектура массово-параллельной обработки (МПО), представленная корпорацией Microsoft в ее операционных системах Windows с возможностями высокопроизводительных вычислений HPC (High Performance Computing).

Этап планирования

Описание этапа планирования установки состоит из двух частей. В первой части даются общие рекомендации, а во второй рассматривается использование центра установки SQL Server для планирования установки системы.

Общие рекомендации

В процессе установки вам придется принимать множество решений. Общей рекомендацией будет ознакомиться со следствиями этих решений, прежде чем приступать к установке системы. Вначале нужно ответить на приведенные далее вопросы.

- Какие компоненты SQL Server следует установить?
- Где расположить корневой каталог?
- Следует ли использовать множественные экземпляры компонента Database Engine?
- Какой режим аутентификации использовать для компонента Database Engine?

Эти вопросы рассматриваются в следующих подразделах.

Какие компоненты SQL Server следует установить?

Прежде чем приступать к установке системы, нужно точно знать, какие компоненты SQL Server вы хотите установить. Частичный список имеющихся компонентов показан на рис. 2.1.

Страницу выбора компонентов **Feature Selection** мы увидим снова при установке SQL Server далее в этой главе, но зная наперед, какие компоненты выбрать, вам не нужно будет прерывать процесс установки, чтобы заниматься исследованиями по этому вопросу. Компоненты на странице **Feature Selection** сгруппированы в две категории: **Instance Features** (компоненты экземпляра) и **Shared Features** (общие функции).

В этом разделе рассматриваются только основные компоненты. Для описания общих компонентов обратитесь к онлайновой документации по SQL Server 2012.

Первым в списке основных возможностей идет узел Database Engine Services. Компонент Database Engine является реляционной системой баз данных сервера SQL Server. Разные аспекты компонента Database Engine рассматриваются в *части II* и *III* этой книги. Первый подкомпонент узла Database Engine Services, подузел SQL Server Replication, позволяет дублировать данные с одной системы на другую.

Иными словами, репликация данных позволяет получить среду распределенных данных. Подробно предмет репликации данных рассматривается в *главе 18*.

Вторым подузлом является Full-Text Search (полнотекстовый и семантический поиск). Компонент Database Engine позволяет сохранять структурированные данные в столбцах реляционных таблиц. В противоположность неструктурированные данные, как правило, сохраняются в файловой системе в виде текста. По этой причине нам будут нужны различные способы извлечения информации из неструктурированных данных. Компонент Full-Text Search системы SQL Server позволяет сохранять и опрашивать неструктурированные данные. Этот компонент рассматривается подробно в *главе 28*.

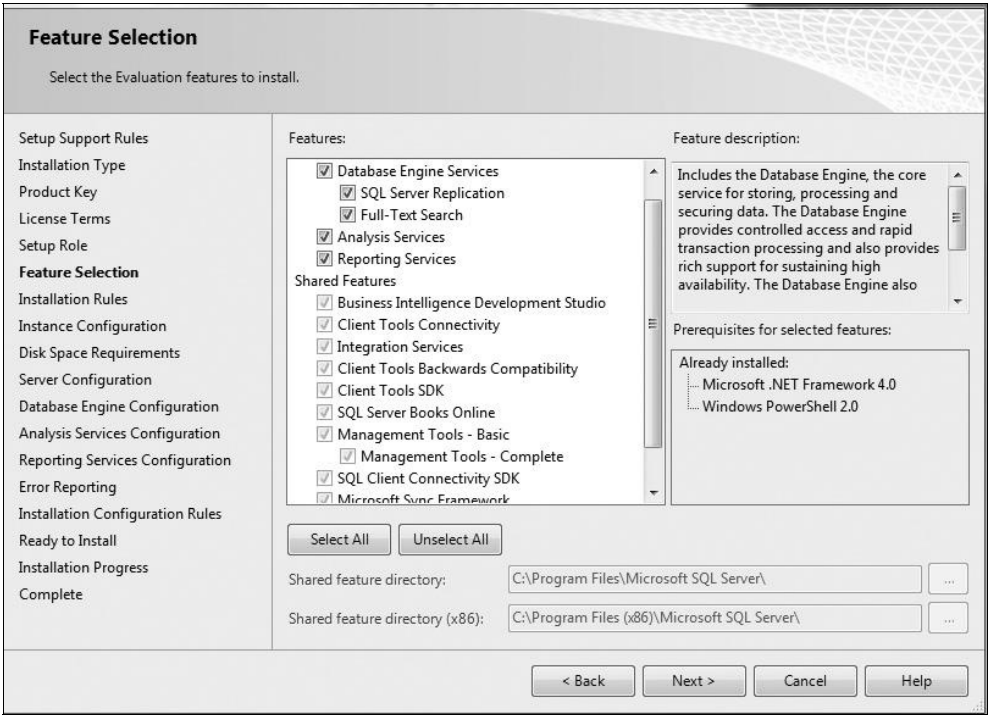


Рис. 2.1. Страница выбора компонентов **Feature Selection**

Кроме компонента Database Engine система SQL Server содержит компоненты Analysis Services и Reporting Services для обеспечения функций бизнес-аналитики и отчетности. Службы Analysis Services используются для управления и запроса данных, хранящихся в хранилище данных. (*Хранилище данных* представляет собой базу данных, содержащую все данные корпорации, к которым пользователи могут иметь единообразный доступ.) Общее рассмотрение предмета выполнения бизнес-аналитики в SQL Server дается в *части IV* книги, а службы Analysis Services в частности рассматриваются в *главе 22*.

Служба отчетности Reporting Services используется для создания и управления отчетами. Этот компонент системы SQL Server подробно рассматривается в *главе 24*.

Где расположить корневой каталог?

Корневой каталог — это каталог, в который программа установки помещает все файлы программы и те файлы, неизменяемые при использовании системы SQL Server. По умолчанию программа установки помещает все программные файлы в подкаталог Microsoft SQL Server каталога Program Files системного диска. Рекомендуется использовать название каталога по умолчанию, т. к. оно однозначно определяет версию системы.

Следует ли использовать множественные экземпляры компонента Database Engine?

Является возможным установка и использование нескольких экземпляров компонента Database Engine. *Экземпляром* называется сервер базы данных, чья система и пользовательские базы данных не доступны для совместного использования другим серверам (экземплярам), выполняющимся на том же компьютере.

Существует два типа экземпляров:

- по умолчанию (default);
- именованные (named).

Экземпляр по умолчанию работает таким же образом, как и серверы баз данных в предыдущих версиях SQL Server, в которых был возможен только один сервер баз данных, а дополнительные экземпляры не поддерживались. Имя компьютера, на котором выполняется экземпляр сервера, указывает только имя экземпляра по умолчанию. Любой экземпляр сервера базы данных, иной, чем экземпляр по умолчанию, называется *именованным экземпляром*. Для идентификации именованного экземпляра нужно указать его имя, а также имя компьютера, на котором он выполняется, например, NTB11901A INSTANCE1. На одном компьютере, кроме экземпляра по умолчанию, может выполняться несколько именованных экземпляров сервера. Кроме этого, можно создать именованные экземпляры на компьютере без экземпляра по умолчанию.

Хотя большинство системных ресурсов (службы SQL Server и Агент SQL Server, системные и пользовательские базы данных и ключи реестра) не используются совместно всеми выполняющимися на компьютере экземплярами сервера, некоторые ресурсы находятся в совместном использовании. А именно:

- программная группа SQL Server;
- сервер служб Analysis Services;
- библиотеки разработки.

Факт наличия на компьютере только одной программной группы SQL Server означает наличие также только одной копии каждой служебной программы, которая представляется значком в программной группе. (Это также включает электронную документацию по SQL Server.) Поэтому каждая служебная программа используется всеми экземплярами сервера, сконфигурированными на компьютере.

Использование множественных экземпляров следует рассмотреть в том случае, если удовлетворяются следующие два условия:

- на компьютере установлены разные типы баз данных;
- компьютер обладает достаточной производительностью для работы с несколькими экземплярами.

Множественные экземпляры в основном используются с целью разбиения существующих баз данных организации на различные группы. Например, если система управляет базами данных, которые используются разными типами пользователей (производственные базы данных, тестовые базы данные, образцовые базы данных и т. п.), следует разнести их по разным экземплярам. Таким образом, можно изолировать производственные базы данных от баз данных, которыми пользуются нечастые или неопытные пользователями. Однопроцессорная машина не будет подходящей платформой для выполнения множественных экземпляров компонента Database Engine по причине недостаточности ресурсов. Поэтому использование множественных экземпляров следует рассматривать только для многопроцессорных компьютеров.

Какой режим проверки подлинности использовать для компонента Database Engine?

Для компонента Database Engine возможны два разных режима проверки подлинности или иначе *режима аутентификации* (authentication mode).

- *Режим Windows* (Windows mode). Определяет безопасность исключительно на уровне операционной системы, т. е. определяет способ подключения пользователей к операционной системе Windows посредством своих учетных записей и членства в группе.
- *Смешанный режим* (mixed mode). Позволяет пользователям подключаться к компоненту Database Engine посредством проверки подлинности Windows или проверки подлинности SQL Server. Это означает, что некоторых пользователей можно настроить для использования подсистемы безопасности Windows, а другие вдобавок к этому могут применять подсистему безопасности SQL Server.

Корпорация Microsoft рекомендует использовать режим Windows. (Подробная информация по этому вопросу в *главе 12*.)

Планирование установки

SQL Server оснащен инструментом SQL Server Installation Center (центр установки) (рис. 2.2), который выводится при запуске программы установки этого программного обеспечения.

Этот инструмент предоставляет поддержку пользователя на этапах планирования установки, установки и обслуживания системы базы данных.

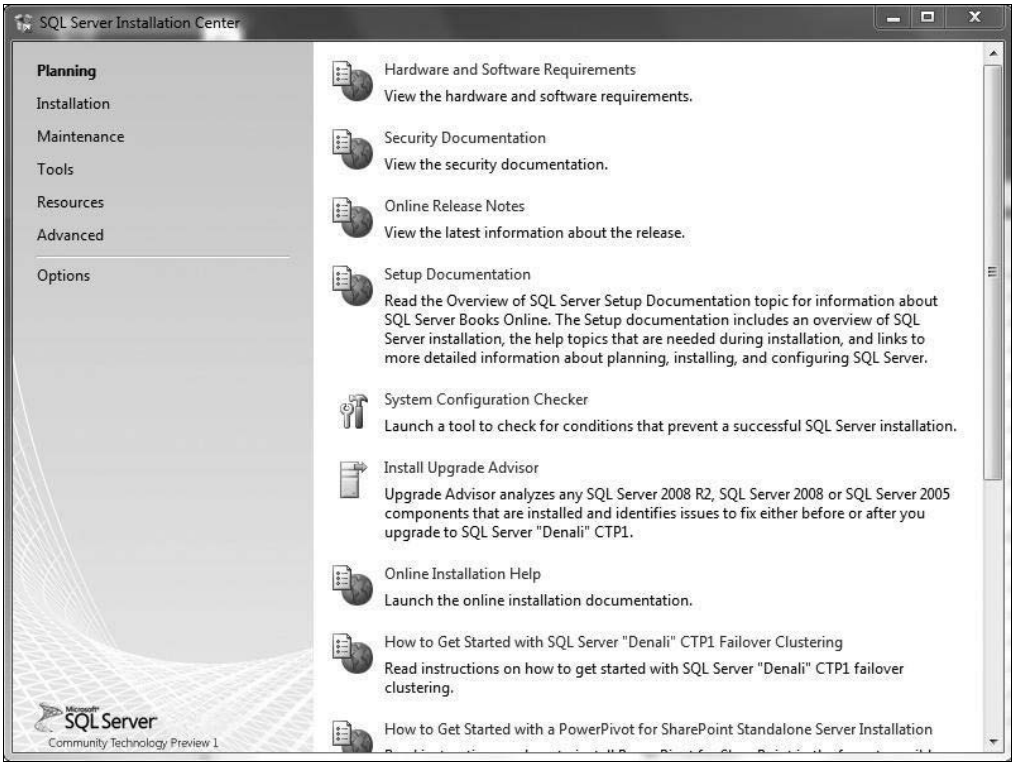


Рис. 2.2. Центр установки SQL Server Installation Center

Чтобы начать этап планирования установки, вставьте установочный DVD-диск в привод. (Данное программное обеспечение также распространяется в виде файла ISO.) Откроется мастер Install Shield и запросит указать каталог на диске, в который поместить извлеченные файлы. Укажите требуемый каталог и нажмите кнопку **Next** (Далее); мастер Install Shield распакует из DVD-диска все необходимые файлы в указанный каталог и завершит свою работу.

Первым пунктом в центре установки является пункт планирования установки **Planning**. Выбор этого пункта предоставляет возможность выполнения следующих основных задач (см. рис. 2.2):

- **Hardware and Software Requirements** (Требования к оборудованию и программному обеспечению);
- **Security Documentation** (Документация по безопасности);
- **Online Release Notes** (Заметки о версии в сети);
- **System Configuration Checker** (Средство проверки конфигурации);
- **Install Upgrade Advisor** (Установка помощника по обновлению);
- **Online Installation Help** (Электронная справка по установке).

Помощник по обновлению анализирует все компоненты установленных предыдущих выпусков сервера и определяет проблемы, которые нужно устранить, прежде

чем выполнять обновление до SQL Server 2012. Поддерживаются предыдущие выпуски SQL Server 2005 и 2008 (включая версию Release 2).

Эти задачи рассматриваются более подробно в следующих подразделах.

Требования к оборудованию и программному обеспечению

То обстоятельство, что система SQL Server может исполняться только на платформах с операционной системой Windows, упрощает решения относительно требований к аппаратному и программному обеспечению. Поскольку вопрос программного обеспечения не стоит, системный администратор должен принять во внимание только требования к оборудованию и сети.

Требования к оборудованию

Операционные системы Windows поддерживаются аппаратными платформами Intel и AMD (Opteron и Athlon 64). Рабочая частота процессора должна быть минимум 1,4 ГГц.

ПРИМЕЧАНИЕ

Как правило, существует две группы версий SQL Server: 32-разрядная и 64-разрядная. Требования для каждой группы разные, поэтому перечисленные в этом разделе значения являются только общими.

Официально требуется минимум 512 Мбайт оперативной памяти. Но практически любому очевидно, что такого минимального объема памяти будет недостаточно для эффективной работы, поэтому общей рекомендацией будет иметь, по крайней мере, 2 Гбайт оперативной памяти и желательно больше.

Требуемый объем жесткого диска зависит от конфигурации системы и приложений, которые планируется устанавливать. Чем больше компонентов SQL Server вы хотите установить, тем больше дискового пространства вам потребуется.

Требования к сети

Для подключения к любому компоненту SQL Server требуется наличие сетевого протокола. Система SQL Server может обслуживать запросы одновременно по нескольким протоколам. Клиенты же подключаются к системе по одному определенному протоколу. Если клиенту неизвестен протокол, который прослушивает система, следует настроить клиент для попытки подключиться, последовательно перебирая несколько протоколов.

Будучи системой типа "клиент-сервер", SQL Server позволяет клиентским программам использовать разные сетевые протоколы для взаимодействия с сервером, и наоборот. При установке средств обеспечения связи системный администратор должен решить, какие сетевые протоколы (библиотеки) предоставить для обеспе-

чения клиентам доступа к системе. На стороне сервера можно выбрать следующие сетевые протоколы:

- *Разделяемая память* (Shared memory). Используется подключениями к системе от клиента, исполняющегося на том же компьютере, что и SQL Server. Разделяемая память не имеет никаких настраиваемых свойств, поэтому этот протокол всегда пробуется первым.
- *Именованные каналы* (Named pipes). Альтернативный сетевой протокол на платформах Windows. После установки можно удалить поддержку для именованных каналов и пользоваться другим сетевым протоколом для взаимодействия между сервером и клиентом.
- *Протокол TCP/IP* (Transmission Control Protocol/Internet Protocol). Позволяет обмениваться информацией, используя стандартные сокеты Windows в качестве метода взаимодействия процессов (IPC — inter-process communication) по протоколу TCP/IP.
- *Протокол адаптера виртуального интерфейса* (Virtual Interface Adapter — VIA). Работает с аппаратным обеспечением адаптера VIA. Для информации по использованию адаптера VIA свяжитесь с поставщиком вашего оборудования. (Протокол VIA устарел и будет изъят из будущих версий SQL Server.)

ПРИМЕЧАНИЕ

Разделяемая память (Shared memory) не поддерживается для отказоустойчивых кластеров (см. *главу 16*).

Документация по безопасности

Выбор ссылки **Security Documentation** (Документация по безопасности) в правой панели пункта **Planning** (Планирование) запускает браузер по умолчанию, в котором открывается страница сайта Microsoft, на которой рассматриваются общие соображения безопасности. Одной из наиболее важных мер безопасности является изолирование служб друг от друга. Для этого отдельные службы SQL Server выполняются с разными учетными записями Windows. (Учетные записи Windows и другие аспекты безопасности рассматриваются в *главе 12*.) Информацию обо всех других аспектах безопасности можно найти в электронной документации по SQL Server 2012.

Заметки о версии в сети

Двумя основными источниками информации обо всех возможностях системы SQL Server является *электронная документация* (Books Online) и *заметки о версии в сети* (Online Release Notes). Электронная документация поставляется со всеми компонентами SQL Server, а заметки о версии в сети содержат только самую последнюю информацию, которая может отсутствовать в электронной документации. (Причиной этому является то обстоятельство, что ошибки и определенные пробле-

мы с работой системы иногда обнаруживаются уже после подготовки и выпуска электронной документации.) Настоятельно рекомендуется внимательно ознакомиться с содержанием заметок о версии в сети, чтобы получить представление о возможностях, которые были модифицированы незадолго до выпуска окончательной версии.

Электронная справка по установке

Электронная документация по установке содержит обзор установки SQL Server, все темы справки релевантные к установке, а также ссылки на информацию о планировании установки, а также конфигурированию SQL Server. Если в процессе установки вам придется столкнуться с проблемой, не рассмотренной в этой главе, обратитесь за информацией по ее возможному решению к соответствующей теме электронной справки.

Средство проверки конфигурации

Одной из наиболее важных задач планирования установки является проверка удовлетворения всех условий, требуемых для успешной установки системы. Выбор ссылки **System Configuration Checker** (Средство проверки конфигурации) пункта **Planning** (Планирование) запускает инструмент Setup Support Rules (правила поддержки установки). Этот же инструмент запускается в начале этапа установки, которая рассматривается в следующем разделе. Инструмент Setup Support Rules определяет проблемы, которые могут возникнуть при установке вспомогательных файлов SQL Server. По завершению выполнения этой задачи система выводит отчет о выполненных проверках и их результатах. Все обнаруженные проблемы необходимо устранить, прежде чем приступать к установке.

Установка SQL Server

Если вам уже когда-либо ранее приходилось устанавливать сложное программное обеспечение, то вам, наверное, будет знакомо чувство неопределенности, которое сопровождает первый запуск программы установки. Это чувство порождается сложностью устанавливаемого продукта и великим разнообразием вопросов, которые нужно ответить в процессе установки. Так как вы можете не совсем понимать продукт и быть не совсем уверены в том, что сможете дать правильные ответы на все вопросы, задаваемые программой установки с целью выполнения своей задачи. Материал этого раздела поможет вам держаться правильного пути в процессе установки, предоставив ответы на большинство вопросов, с которыми вам, скорей всего, придется иметь дело.

Как можно судить по его названию, кроме планирования установки программного обеспечения, центр установки также обеспечивает собственно его установку. Этот инструмент показывает несколько опций по отношению к установке системы базы данных и ее компонентов. Чтобы начать установку, выберите в левой панели центра установки пункт **Installation** (Установка), а потом в правой панели щелкните

ссылку **New SQL Server stand-alone installation or add features to and existing installation** (Новая установка изолированного экземпляра SQL Server или добавление экземпляров к существующей установке). Будет запущен мастер установки SQL Server 2012.

На первой странице мастера **Setup Support Rules** (Правила поддержки установки) (рис. 2.3) определяются проблемы, которые могут возникнуть при установке вспомогательных файлов программы установки SQL Server. (Это тот же самый инструмент, который используется при запуске средства проверки конфигурации на этапе планирования.)

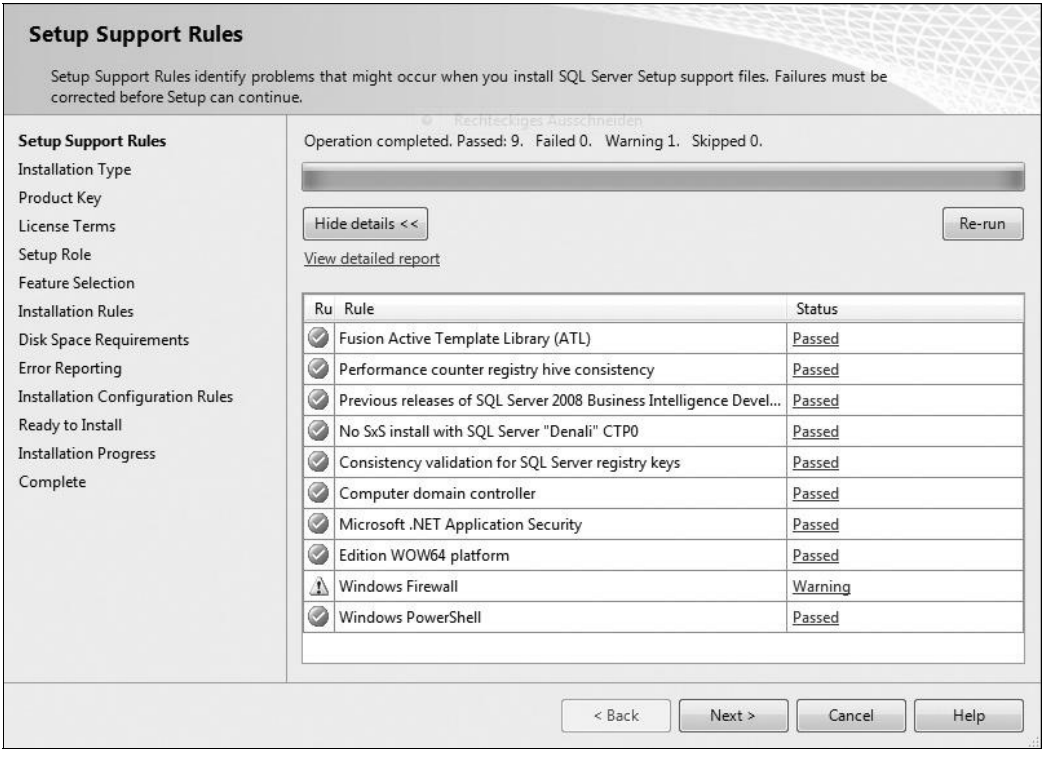


Рис. 2.3. Страница **Setup Support Rules**

Любые обнаруженные этим инструментом ошибки необходимо устранить, прежде чем процесс установки может продолжиться. При отсутствии ошибок нажмите кнопку **Next** (Далее).

На странице **Installation Type** (Тип установки) выберите один из следующих двух переключателей:

- **Perform a new installation of SQL Server 2012** (Выполнить новую установку SQL Server 2012);
- **Add features to an existing instance of SQL Server 2012** (Добавить компоненты в существующий экземпляр SQL Server 2012).

Если выбран второй вариант, то в раскрывающемся списке выберите экземпляр SQL Server, который нужно обновить. Выбрав требуемый вариант установки, нажмите кнопку **Next** (Далее).

На следующей странице мастера **Product Key** (Ключ продукта) введите 25-значный код, который находится на упаковке продукта. (Альтернативно можно указать бесплатную версию программы, например, SQL Server Express.) Чтобы продолжить установку, нажмите кнопку **Next** (Далее). На странице **License Terms** (Условия лицензии) установите флажок **I accept the license terms** (Я принимаю условия лицензии).

На следующей странице **Setup Role** (Роль установки) предоставляется возможность выбора между установкой только основных компонентов SQL Server 2012 (службы компонента Database Engine, службы анализа Analysis Services и службы отчетности Reporting Services) и установкой дополнительных вспомогательных компонентов PowerPivot для SharePoint. Выберите вариант установки компонентов SQL Server и нажмите кнопку **Next** (Далее).

На странице **Feature Selection** (Выбор компонентов) (рис. 2.4) установите флажки для компонентов, которые требуется установить. На этой же странице внизу можно указать каталог для размещения компонентов общего использования. Указав все необходимые опции, нажмите кнопку **Next** (Далее).

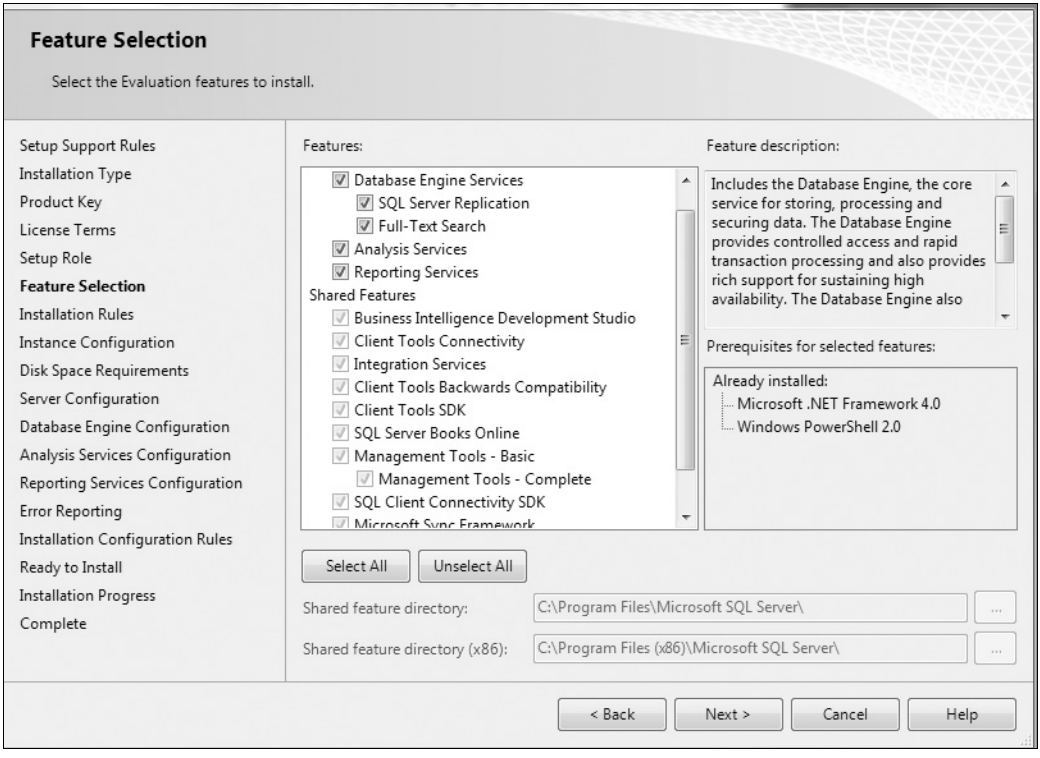


Рис. 2.4. Страница **Feature Selection**

ПРИМЕЧАНИЕ

На рис. 2.4 названия всех совместно используемых компонентов указаны затененным (тусклым) шрифтом, что означает, что в данном случае эти возможности не выбираются. Но при установке системы на своей машине вам следует решить, какие из этих функциональностей нужно установить и отметить соответствующие флажки.

ПРИМЕЧАНИЕ

Выбранные компоненты SQL Server будут установлены один за другим в последовательности, в которой они указаны на странице **Feature Selection** (Выбор компонентов). Процесс установки начинается с установки компонента Database Engine, после чего следует установка служб Analysis Services и т. д. Установлены будут только выбранные компоненты.

На следующей странице **Installation Rules** (Правила установки) программа установки выполняет проверки, чтобы определить удовлетворение всех правил для успешной установки. В случае удовлетворения всех правил (или пометки **Not applicable** (Неприменимо)), для продолжения установки нажмите кнопку **Next** (Далее).

На странице **Instance Configuration** (Настройка экземпляра) (рис. 2.5) предоставляется возможность выбора установки экземпляра по умолчанию или именованного экземпляра.

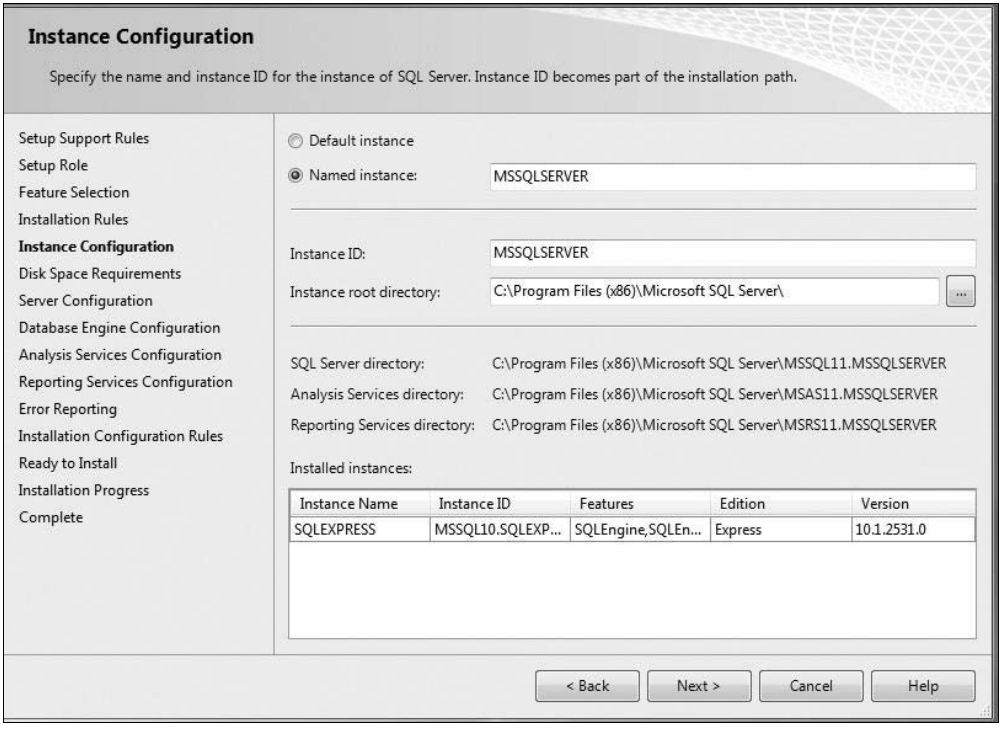


Рис. 2.5. Страница **Instance Configuration**

(Эти два типа экземпляров подробно рассматривались ранее в *разд. "Следует ли использовать множественные экземпляры компонента Database Engine?"* этой главы.) Для установки экземпляра по умолчанию установите переключатель **Default instance** (Экземпляр по умолчанию). Если выбрать эту опцию, когда экземпляр по умолчанию уже установлен, то программа установки выполнит обновление этого экземпляра и предоставит возможность установить дополнительные компоненты. Таким образом, есть возможность установить компоненты, которые были пропущены в предыдущей установке.

Для установки именованного экземпляра установите соответствующий переключатель и введите для него имя в текстовое поле справа. В нижней части страницы **Instance Configuration** (Настройка экземпляра) в поле **Installed instances** (Установленные экземпляры) указываются экземпляры сервера, уже установленные на данном компьютере. (Как можно видеть на рис. 2.5, на компьютере, на котором выполняется установка, уже установлен экземпляр по умолчанию, называющийся MSSQLSERVER.) Чтобы продолжить установку, нажмите кнопку **Next** (Далее).

Откроется страница **Disk Space Requirements** (Требования к свободному месту на диске), на которой отображается сводка по требованиям дискового пространства для установки системы базы данных и доступное свободное пространство. Если требования дискового пространства удовлетворяются, то, чтобы продолжить установку, нажмите кнопку **Next** (Далее).

На следующей странице **Server Configuration** (Конфигурация сервера) (рис. 2.6) можно указать имена пользователей и соответствующие пароли для служб всех компонентов, которые будут установлены. (Можно использовать одну учетную запись для всех служб, но такой подход не рекомендуется по причинам безопасности.)

На вкладке **Collation** (Параметры сортировки) этой страницы можно задать требуемый порядок сортировки для устанавливаемых компонентов экземпляра. Можно или выбрать порядок сортировки по умолчанию, или же нажать кнопку **Customize** (Настройка) для требуемого компонента и выбрать какой-либо другой порядок сортировки, поддерживаемый системой. Завершив работу на этой странице, чтобы продолжить установку, нажмите кнопку **Next** (Далее).

Откроется страница **Database Engine Configuration** (Настройка компонента Database Engine) (рис. 2.7), на которой нужно выбрать режим проверки подлинности для системы Database Engine.

Как уже рассматривалось ранее, компонент Database Engine поддерживает режим аутентификации Windows и смешанный режим аутентификации. Если выбрать режим проверки подлинности Windows, то установив переключатель **Windows authentication mode** (Режим проверки подлинности Windows), программа установки создаст учетную запись администратора **sa** (system administrator) системы SQL Server, которая по умолчанию будет отключена. (Учетные записи подробно рассматриваются в *главе 12*.) Если же выбрать смешанный режим проверки подлинности, установив переключатель **Mixed Mode (SQL Server authentication and Windows authentication)**) (Смешанный режим (проверка подлинности SQL Server и Windows)), то нужно будет ввести и подтвердить пароль для учетной записи адми-

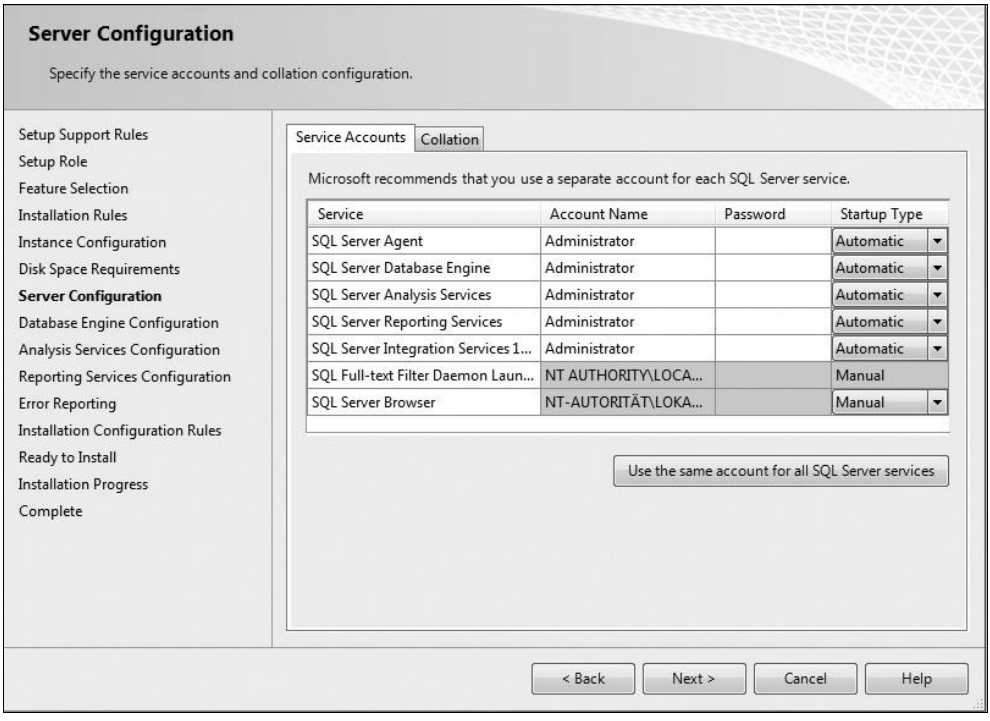


Рис. 2.6. Страница **Server Configuration**, вкладка **Service Accounts**

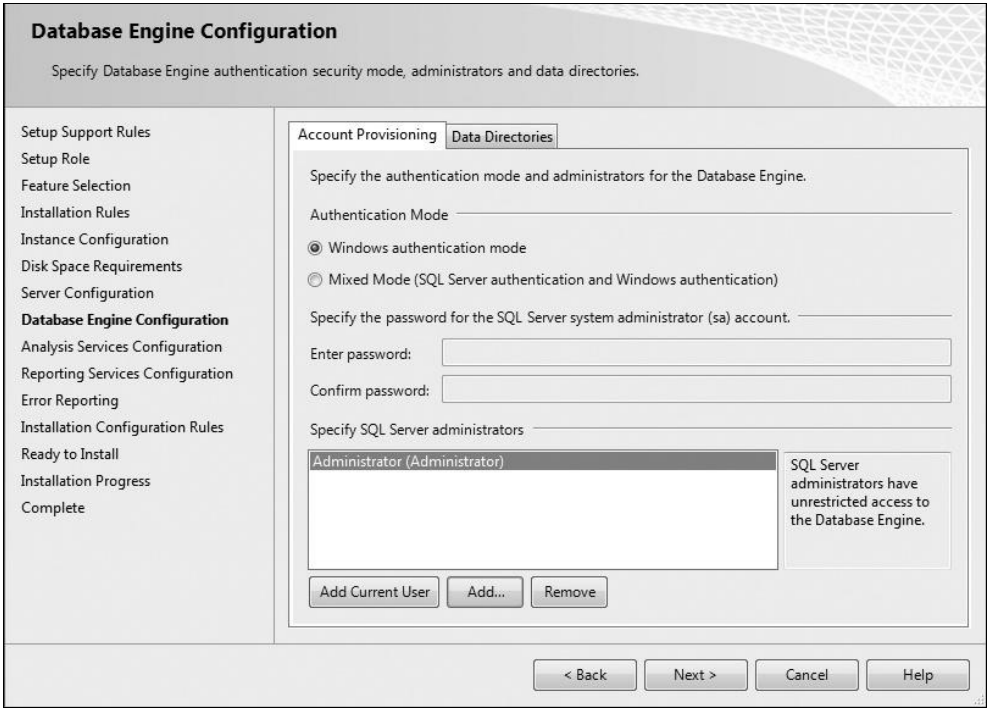


Рис. 2.7. Страница **Database Engine Configuration**, вкладка **Account Provisioning**

нистратора. Чтобы добавить текущего пользователя в список пользователей с неограниченным доступом к данному экземпляру Database Engine, нажмите кнопку **Add Current User** (Добавить текущего пользователя). Для добавления других пользователей нажмите кнопку **Add** (Добавить).

ПРИМЕЧАНИЕ

Информацию об учетных записях можно будет изменить после установки. В таком случае нужно будет перезапустить службу MSSQLSERVER компонента Database Engine.

На вкладке **Data Directories** (Каталоги данных) (рис. 2.8) страницы настройки Database Server можно указать расположение для всех каталогов, в которых хранятся файлы, связанные с Database Engine. Выполнив все необходимые настройки, нажмите кнопку **Next** (Далее).

Последующие развития событий будут зависеть от того, были ли выбраны для установки службы Analysis Services или нет. (Для каждого устанавливаемого компонента SQL Server выводится страница конфигурации.) Если была выбрана установка этого компонента, то для его настройки откроется страница наподобие приведенной на рис. 2.7. Укажите на ней пользователей, которым будет разрешен доступ к службам Analysis Services, и нажмите кнопку **Next** (Далее).

Подобным образом последующие развития событий будут зависеть от того, был ли выбран для установки компонент службы Reporting Services. Если этот компонент

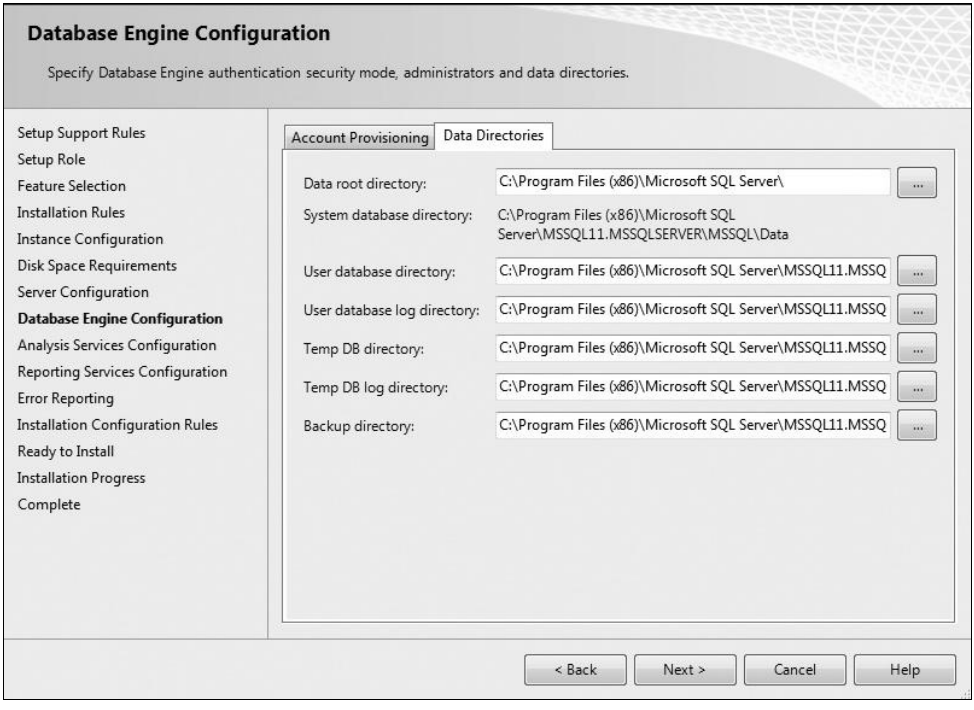


Рис. 2.8. Вкладка **Data Directories** страницы настройки Database Engine

Рис. 2.9. Страница **Reporting Services Configuration**

был выбран, то для установки откроется страница **Reporting Services Configuration** (Настройка служб Reporting Services) (рис. 2.9).

На этой странице можно выбрать или просто установить сервер отчетов, не выполняя его настройку, или же установить и настроить его. Третьей опцией будет интегрировать сервер отчетов с сервером SharePoint пакета Microsoft Office. Сервер SharePoint можно использовать для облегчения совместной работы, предоставления возможности управления содержимым и реализации бизнес-процессов. Выполнив все необходимые настройки на этой странице, нажмите кнопку **Next** (Далее).

Откроется страница **Error Reporting** (Отчет об ошибках), на которой можно указать информацию об ошибках сервера и которую вы хотите автоматически отправлять компании Microsoft. Сбросьте флажок внизу страницы, если вы не хотите принимать участие в этом автоматическом предоставлении отчетов об ошибках сервера. Нажмите кнопку **Next** (Далее).

Следующая страница **Installation Configuration Rules** (Правила конфигурации установки) похожа на страницу **Installation Rules** (Правила установки). После завершения выполнения этого шага эта страница будет отображать сведения обо всех правилах конфигурации и их удовлетворении.

Последней страницей перед началом процесса установки является страница **Ready to Install** (Все готово для установки). На этой странице можно просмотреть сводную информацию об устанавливаемых компонентах SQL Server. Чтобы начать процесс установки, на этой странице нажмите кнопку **Install** (Установить). Будет запущен процесс установки SQL Server, ход которой будет отображаться в от-

крывшемся окне **Installation Progress** (Ход выполнения установки). В случае успешной установки, нажмите кнопку **Next** (Далее).

По завершению установки открывается страница **Complete** (Завершено) (рис. 2.10), на которой предоставляется информация о месте размещения файла журнала установки. Чтобы выйти из программы установки, нажмите кнопку **Close** (Закрыть). Теперь можно использовать все установленные компоненты.

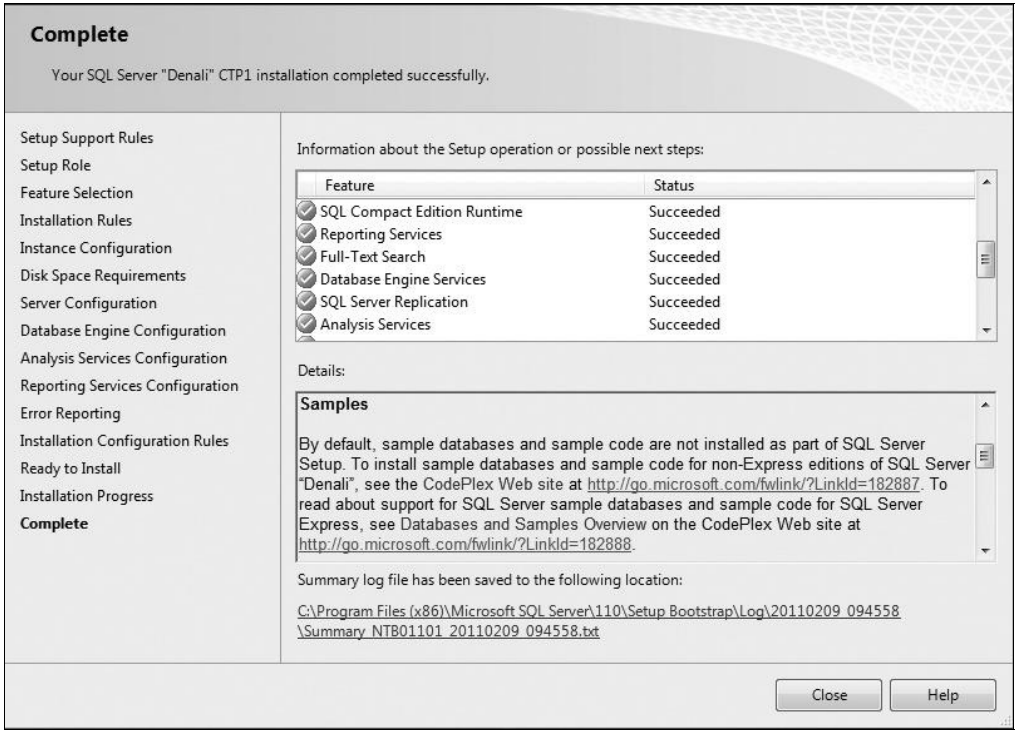


Рис. 2.10. Страница **Complete**

Резюме

Инструмент Центр установки SQL Server позволяет как планировать установку сервера, так и осуществлять ее. Самым важным шагом этапа планирования установки является применение средства проверки конфигурации системы для определения проблем, которые могут возникнуть при установке файлов SQL Server.

Установка SQL Server является простым последовательным процессом. Самым важным решением, которое нужно принять на этом этапе, является выбор компонентов для установки. Это решение подготавливается на этапе планирования установки.

В следующей главе рассматривается среда Management Studio сервера SQL Server. Этот компонент SQL Server используется как администраторами базы данных, так и пользователями для взаимодействия с системой.

Глава 3

Среда управления SQL Server Management Studio

- **Введение в среду управления SQL Server Management Studio**
- **Использование среды SQL Server Management Studio с компонентом Database Engine**
- **Разработка запросов, используя среду SQL Server Management Studio**

В начале этой главе рассматривается среда управления SQL Server Management Studio, включая информацию о ее подключении к серверу, ее компонентах Registered Servers (Зарегистрированные серверы) и Object Explorer (Обозреватель объектов) и ее различных панелях пользовательского интерфейса. Далее подробно рассматриваются функциональности среды SQL Server Management Studio, связанные с компонентом Database Engine, включая возможности администрирования и управления базами данных, которые необходимо понимать, чтобы быть в состоянии создавать и выполнять инструкции языка Transact-SQL. В заключительном материале главы обсуждается использование компонентов Query Editor (Редактор запросов) и Solution Explorer (Обозреватель решений) и средства отладки для разработки запросов в среде SQL Server Management Studio.

Введение в среду управления SQL Server Management Studio

Система SQL Server 2012 предоставляет различные инструменты для выполнения всевозможных задач, таких как установка, конфигурирование, контрольная проверка системы (аудит) и настройка ее производительности. (Инструменты для выполнения всех этих задач рассматриваются в разных главах книги.) Основным инструментом администратора для взаимодействия с системой является среда управления SQL Server Management Studio. Как администраторы, так и конечные пользователи

могут использовать этот инструмент для администрирования множественных серверов, разработки баз данных и репликации данных.

ПРИМЕЧАНИЕ

Эта глава полностью посвящена рассмотрению действий конечного пользователя. Поэтому в ней подробно рассматривается только функциональность среды SQL Server Management Studio применительно создания объектов баз данных посредством компонента Database Engine. Все задачи администрирования и задачи, связанные со службами анализа Analysis Services, а также другие, поддерживаемые этим инструментом компоненты рассматриваются в *части III* этой книги.

Для запуска среды SQL Server Management Studio выполните последовательность команд **Пуск | Все программы | Microsoft SQL Server 2012 | SQL Server Management Studio**.

Среда управления SQL Server Management Studio состоит из нескольких разных компонентов, которые используются для администрирования и управления всей системой. Основные из этих компонентов перечислены в следующем списке:

- Registered Servers (Зарегистрированные серверы);
- Object Explorer (Обозреватель объектов);
- Query Editor (Редактор запросов);
- Solution Explorer (Обозреватель решений).

В этом разделе рассматриваются первые два компонента списка. Редактор запросов и обозреватель решений рассматриваются далее в *разд. "Разработка запросов, используя среду SQL Server Management Studio"* этой главы.

Чтобы открыть главный интерфейс среды SQL Server Management Studio, нужно сначала подключиться к серверу, как это описывается в следующем разделе.

Подключение к серверу

При запуске среды SQL Server Management Studio открывается диалоговое окно **Connect to Server** (Соединение с сервером) (рис. 3.1), в котором нужно задать необходимые параметры для подключения к серверу.

- **Server type** (Тип сервера). Для целей этой главы из раскрывающегося списка выберите опцию **Database Engine** (Компонент Database Engine).

ПРИМЕЧАНИЕ

С помощью среды SQL Server Management Studio, среди прочего, можно управлять объектами компонента Database Engine и служб Analysis Services. В этой главе рассматривается использование среды SQL Server Management Studio только для управления объектами компонента Database Engine.

- **Server name** (Имя сервера). Выберите из раскрывающегося списка или введите с клавиатуры имя сервера, к которому нужно подключиться. (Обычно, среду SQL

Server Management Studio можно подключить к любому установленному продукту на конкретном сервере.)

- **Authentication** (Проверка подлинности). Выберите один из следующих двух типов проверки подлинности:
  - **Windows Authentication** (Проверка подлинности Windows). Подключиться к SQL Server по своей учетной записи Windows. Это наиболее легкий вариант подключения и рекомендуется компанией Microsoft;
  - **SQL Server Authentication** (Проверка подлинности SQL Server). Используется проверка подлинности компонента Database Engine.

ПРИМЕЧАНИЕ

Дополнительную информацию, связанную с проверкой подлинности SQL Server, см. в *главе 12*.

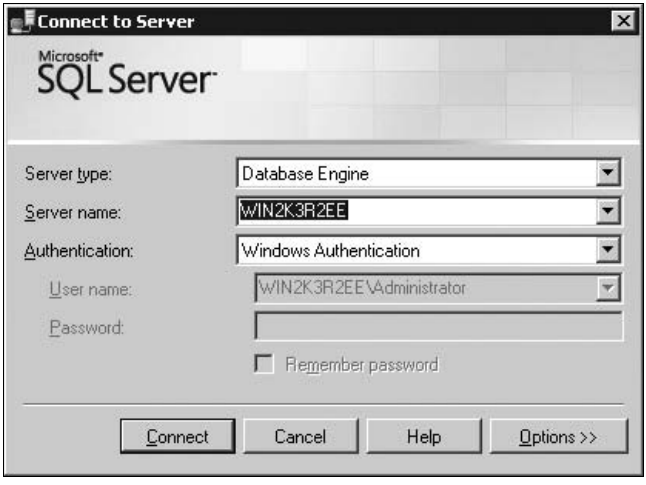


Рис. 3.1. Диалоговое окно **Connect to Server**

Указав все необходимые параметры, нажмите кнопку **Connect** (Соединить) и Database Engine подключится к указанному серверу. После подключения к серверу базы данных открывается главное окно среды SQL Server Management Studio. Своим внешним видом это окно похоже на главное окно среды разработки Visual Studio, поэтому пользователи могут применить свой опыт работы в Visual Studio в данной среде. На рис. 3.2 показано главное окно среды SQL Server Management Studio с несколькими панелями.

ПРИМЕЧАНИЕ

Среда SQL Server Management Studio предоставляет единый интерфейс для управления серверами и создания запросов для всех компонентов SQL Server. Иными словами, для компонентов Database Engine, служб Analysis Services, служб Integration Services и служб Reporting Services применяется один и тот же интерфейс.

Компонент Registered Servers

Компонент Registered Servers (Зарегистрированные серверы) представлен в виде панели, позволяющей работать с уже использованными серверами (см. рис. 3.2). Если панель **Registered Servers** (Зарегистрированные серверы) отсутствует, то ее можно открыть, выбрав ее имя в меню **View** (Вид). С помощью этих подключений можно проверять состояние сервера или управлять его объектами. Для каждого пользователя применяется отдельный список зарегистрированных серверов, который хранится локально.

В список можно добавлять новые серверы или же удалять из него находящиеся в нем. Серверы можно упорядочивать по группам. В каждую такую группу следует помещать серверы, между которыми существует логическая связь. Серверы также можно группировать по типу, например серверы для компонента Database Engine, служб Analysis Services, Reporting Services и Integration Services.

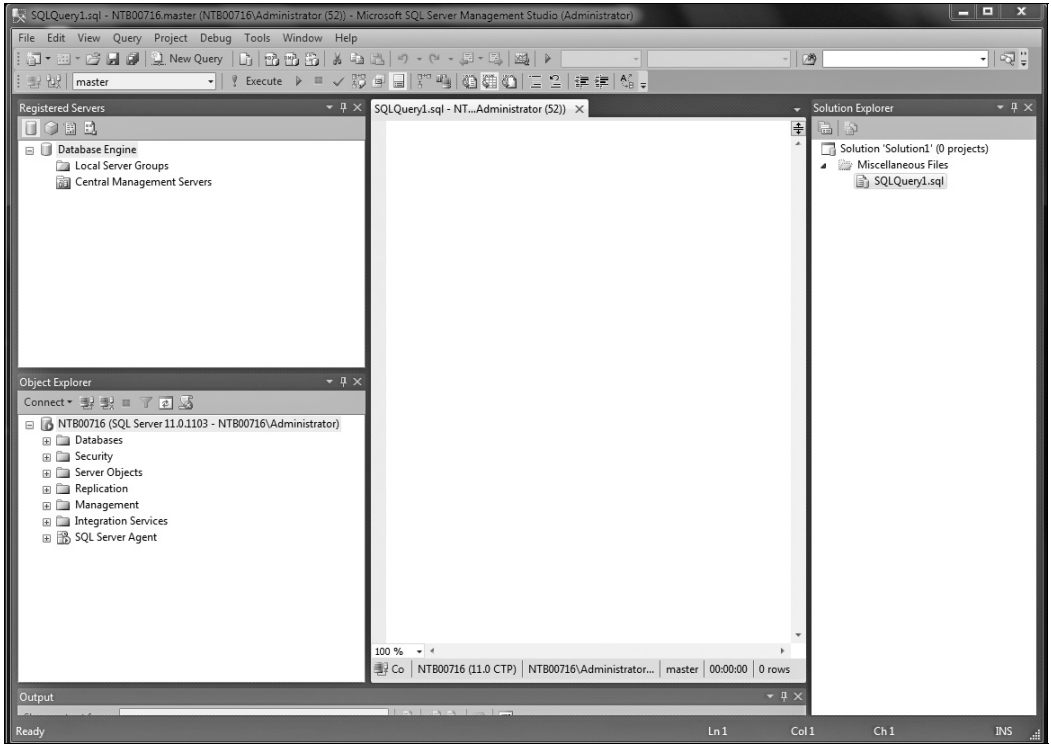


Рис. 3.2. Среда управления SQL Server Management Studio

Компонент Object Explorer

Панель **Object Explorer** (Обозреватель объектов) содержит в виде дерева представление всех объектов баз данных сервера. Если панель **Object Explorer** (Обозреватель объектов) не отображена, то ее можно открыть, выбрав последовательность

команд из меню **View | Object Explorer** (Вид | Инспектор объектов). Данное древовидное представление отображает иерархию объектов на сервере. Таким образом, если ее развернуть, будет показана логическая структура соответствующего сервера.

Обозреватель объектов позволяет подключаться в одной панели к нескольким серверам. Это могут быть любые из имеющихся серверов для компонента Database Engine, служб Analysis Services, Reporting Services или Integration Services. Данная возможность облегчает работу пользователя, поскольку она позволяет управлять всеми серверами одного или разных типов с одного места.

ПРИМЕЧАНИЕ

Обозреватель объектов также обладает другими функциональностями, которые рассматриваются далее в этой главе.

Организация панелей среды SQL Server Management Studio и перемещение по ним

Все панели среды SQL Server Management Studio можно закреплять в главном окне или скрывать из вида. Щелчок правой кнопкой мыши по строке заголовка панели предоставляет выбор из следующих вариантов отображения данной панели:

- **Floating** (Плавающая область) — панель становится свободно перемещающейся поверх остальных панелей среды SQL Server Management Studio, и ее можно поместить в любом месте на экране;
- **Dockable** (Закрепить) — панель можно переместить и закрепить в требуемом месте. Чтобы переместить панель в нужное место, щелкните ее строку заголовка и, не отпуская кнопки мыши, перетащите панель, куда следует;
- **Tabbed Document** (Закрепить как вкладки) — панели можно организовать в виде вкладок документа, когда состояние панели изменяется из закрепляемой на документ с вкладкой;
- **Hide** (Скрыть) — панель можно скрыть. Альтернативно панель можно скрыть, щелкнув крестик в ее правом верхнем углу. Чтобы снова отобразить закрытую панель, выберите ее имя в меню **View** (Вид);
- **Auto Hide** (Автоматически скрывать) — панель сворачивается и прикрепляется в виде вкладки на левой стороне экрана. Чтобы открыть (развернуть) такую панель, наведите указатель мыши на вкладки на левой стороне экрана, а чтобы удерживать панель открытой, нажмите значок канцелярской кнопки в правом верхнем углу панели.

ПРИМЕЧАНИЕ

Различие между режимами **Hide** (Скрыть) и **Auto Hide** (Автоматически скрывать) состоит в том, что в первом случае панель полностью убирается из представления в среде SQL Server Management Studio, а во втором она сворачивается во вкладку.

Для того чтобы восстановить конфигурацию по умолчанию, выберите последовательность команд из меню **Window | Reset Window Layout** (Окно | Сброс макета окон). После сброса настроек с левой стороны среды SQL Server Management Studio располагается панель обозревателя объектов, а с правой — вкладка **Object Explorer Details** (Подробности обозревателя объектов). На вкладке **Object Explorer Details** (Подробности обозревателя объектов) отображается информация о текущем узле, выбранном в обозревателя объектов.

ПРИМЕЧАНИЕ

Среда SQL Server Management Studio позволяет выполнять одну и ту же задачу несколькими способами. В этой главе рассматривается несколько способов выполнения одной задачи, но в последующих главах будет разбираться только один способ. Разные люди отдают предпочтение различным методам — некоторым более по душе двойной щелчок, другие щелкают значки "+"/"–", третьи пользуются правой кнопкой, четвертые обращаются к раскрывающимся меню, пятым нравятся ярлыки и т. п. Чтобы определить наиболее удобный для вас способ перемещения по среде, экспериментируйте с разными способами, пока не выберите самый подходящий.

Подобъект отображается в панелях **Object Explorer** (Обозреватель объектов) и **Registered Servers** (Зарегистрированные серверы) только в том случае, если щелкнуть значок плюс "+" его немедленного корневого узла в дереве иерархии. Чтобы просмотреть свойства объекта, щелкните по нему правой кнопкой мыши и в появившемся контекстном меню выберите пункт **Properties** (Свойства). Знак минус (–) слева от имени объекта означает, что иерархия данного объекта развернута. Чтобы свернуть иерархию подобъектов объекта, нужно опять щелкнуть этот значок. (Другим подходом к сворачиванию иерархии объекта будет выполнение двойного щелчка по его папке или выбор его папки и нажатие клавиши <←> "стрелка влево".)

Использование среды SQL Server Management Studio с компонентом Database Engine

Среда SQL Server Management Studio имеет два основных назначения:

- администрирование серверов баз данных;
- управление объектами баз данных.

Эти функции рассматриваются в следующих разделах.

Администрирование серверов баз данных

Задачи администрирования, которые можно выполнять с помощью среды SQL Server Management Studio, включают, среди прочих, следующие:

- регистрация серверов;
- подключение к серверу;

- создание новых групп серверов;
- управление множественными серверами;
- пуск и остановка серверов.

Эти задачи администрирования описываются в следующих подразделах.

Регистрация серверов

Среда SQL Server Management Studio отделяет деятельность по регистрации серверов от деятельности по исследованию баз данных и их объектов. (Действия этих обоих типов можно выполнять посредством обозревателя объектов.) Прежде чем можно использовать базы данных и объекты любого сервера, будь то локального или удаленного, его нужно зарегистрировать. Сервер можно зарегистрировать при первом запуске среды SQL Server Management Studio или позже. Чтобы зарегистрировать сервер базы данных, щелкните правой кнопкой требуемый сервер в обозревателе объектов и в контекстном меню выберите пункт **Register** (Зарегистрировать). Если панель обозревателя объектов скрыта, то откройте ее, выбрав последовательность команд из меню **View | Object Explorer** (Вид | Обозреватель объектов). Откроется диалоговое окно **New Server Registration** (Регистрация нового сервера), как это показано на рис. 3.3.

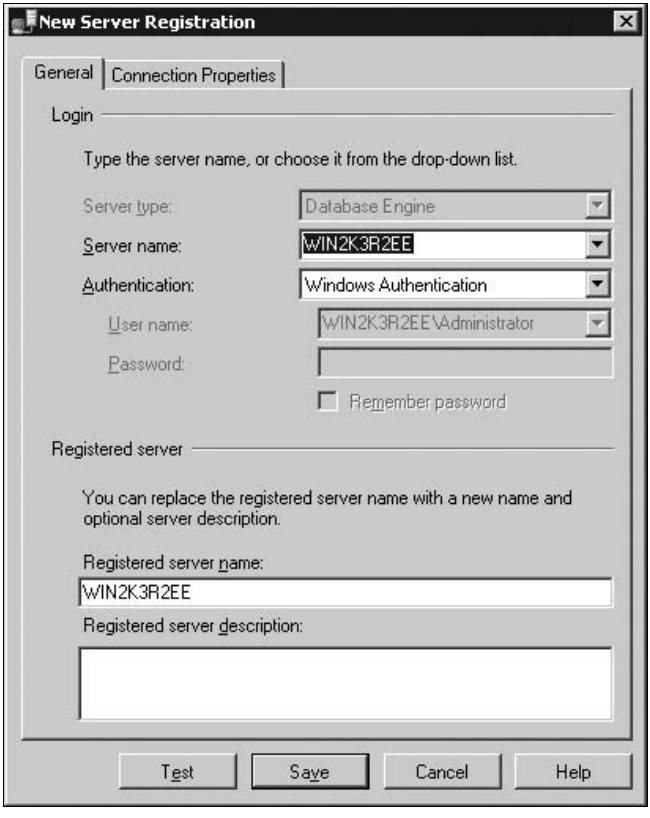


Рис. 3.3. Диалоговое окно **New Server Registration**

Выберите имя сервера, который нужно зарегистрировать, и тип проверки подлинности для этого сервера (т. е. проверка подлинности Windows или проверка подлинности SQL Server), после чего нажмите кнопку **Save** (Сохранить).

Подключение к серверу

Среда SQL Server Management Studio также разделяет задачи регистрации сервера и подключения к серверу. Это означает, что при регистрации сервера автоматического подключения этого сервера не происходит. Чтобы подключиться к зарегистрированному серверу, нужно щелкнуть правой кнопкой требуемый сервер в окне инспектора объектов и в появившемся контекстном меню выбрать пункт **Connect** (Подключиться).

Создание новой группы серверов

Чтобы создать новую группу серверов в панели зарегистрированных серверов, щелкните правой кнопкой узел **Local Server Groups** (Группы локальных серверов) и в контекстном меню выберите пункт **New Server Group** (Создание группы серверов). В открывшемся диалоговом окне **New Server Group Properties** (Свойства новой группа серверов) введите однозначное имя группы и, по выбору, ее описание.

Управление множественными серверами

Посредством обозревателя объектов среда SQL Server Management Studio позволяет администрировать множественные серверы баз данных (называемые *экземплярами*) на одном компьютере. Каждый экземпляр компонента Database Server имеет свой собственный набор объектов баз данных (системные и пользовательские базы данных), который не разделяется между экземплярами.

Для управления сервером и его конфигурацией щелкните правой кнопкой имя сервера в обозревателе объектов и в появившемся контекстном меню выберите пункт **Properties** (Свойства). Откроется диалоговое окно **Server Properties** (Свойства сервера), содержащее несколько страниц, таких как **General** (Общие), **Security** (Безопасность), **Permissions** (Разрешения) и т. п.

На странице **General** (Общие) (рис. 3.4) отображаются общие свойства сервера.

Страница **Security** (Безопасность) содержит информацию о режиме аутентификации сервера и методе аудита входа. На странице **Permissions** (Разрешения) воспроизводятся все учетные записи и роли, которые имеют доступ к серверу. В нижней части страницы отображаются все разрешения, которые можно предоставлять этим учетным записям и ролям.

Можно изменить имя сервера, присвоив ему новое имя. Для этого щелкните правой кнопкой требуемый сервер в окне обозревателя объектов и в контекстном меню выберите пункт **Register** (Зарегистрировать). Теперь можно присвоить серверу новое имя и изменить его описание.

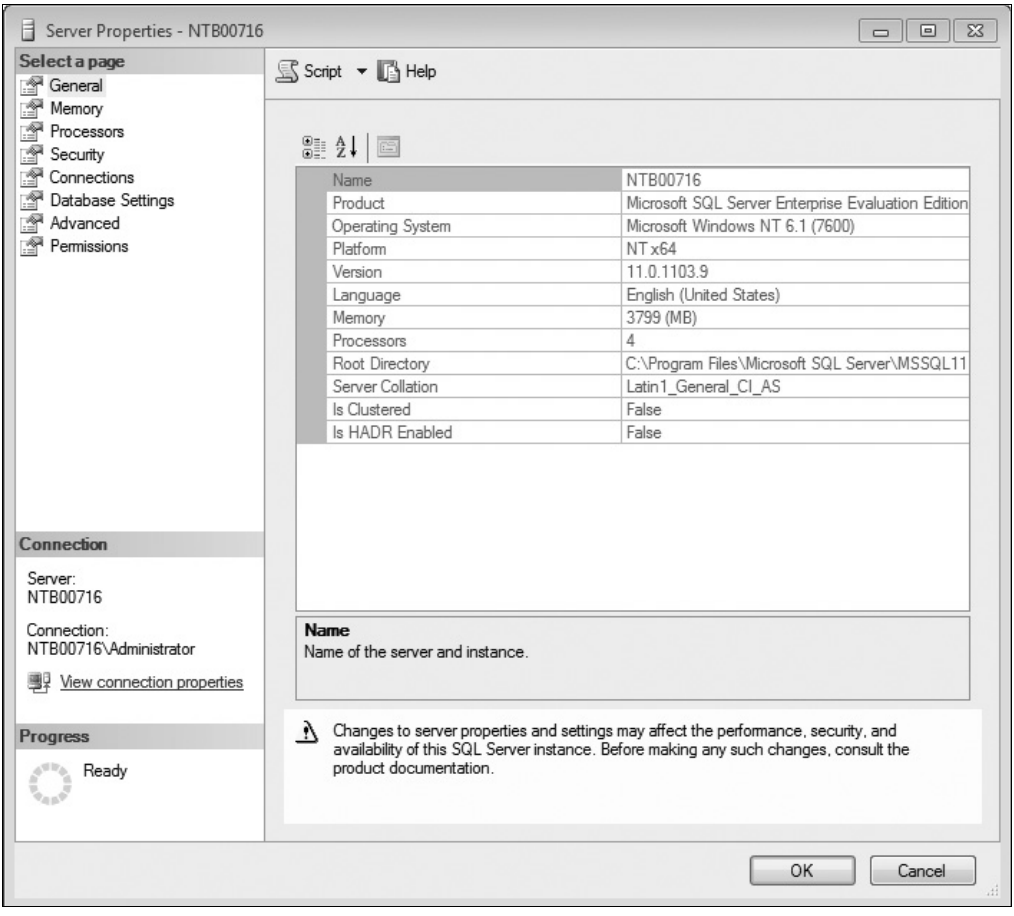


Рис. 3.4. Страница **General** диалогового окна **Server Properties**

ПРИМЕЧАНИЕ

Серверы не следует переименовывать без особой на это надобности, поскольку это может повлиять на другие серверы, которые ссылаются на них.

Запуск и останов серверов

Сервер Database Engine по умолчанию запускается автоматически при запуске операционной системы Windows. Чтобы запустить сервер с помощью среды SQL Server Management Studio, щелкните правой кнопкой требуемый сервер в инспекторе объектов и в контекстном меню выберите пункт **Start** (Запустить). Это меню также содержит пункты **Stop** (Остановить) и **Pause** (Приостановить) для выполнения соответствующих действий с сервером.

Управление базами данных посредством обозревателя объектов

Задачи администрирования, которые можно выполнять с помощью среды SQL Server Management Studio, включают, среди прочих, следующие:

- создание баз данных, не прибегая к использованию языка Transact-SQL;
- модифицирование баз данных, не прибегая к использованию языка Transact-SQL;
- управление таблицами, не прибегая к использованию языка Transact-SQL;
- создание и исполнение инструкций SQL (описывается далее в этой главе в *разд. "Редактор запросов".*)

Создание баз данных, не прибегая к использованию языка Transact-SQL

Новую базу данных можно создать посредством обозревателя объектов Object Explorer или языка Transact-SQL. (Создание баз данные с помощью языка Transact-SQL рассматривается в *главе 5*.) Как можно судить по его названию, обозреватель объектов также можно использовать для исследования объектов сервера. С панели этого инструмента можно просматривать все объекты сервера и управлять сервером и базами данных. Дерево иерархии объектов сервера содержит, среди прочих папок, папку **Databases** (Базы данных). Эта папка, в свою очередь, содержит несколько подпапок, включая папку для системных баз данных, и по папке для каждой базы данных, созданной пользователем. (Системные и пользовательские базы данных подробно рассматриваются в *главе 15*.)

Чтобы создать базу данных посредством обозревателя объектов, щелкните правой кнопкой узел **Databases** (Базы данных) и выберите пункт меню **New Database** (Создать базу данных). В открывшемся диалоговом окне **New Database** (Создание базы данных) (рис. 3.5) в поле **Database name** (Имя базы данных) введите имя новой базы данных, после чего нажмите кнопку **OK**.

Каждая база данных обладает несколькими свойствами, такими как тип файла, начальный размер и т. п. Список страниц свойств базы данных расположен в левой панели диалогового окна **New Database** (Создание базы данных). Существует несколько разных страниц (групп) свойств:

- **General** (Общие);
- **Files** (Файлы);
- **Filegroups** (Файловые группы);
- **Options** (Параметры);
- **Change Tracking** (Отслеживание изменений);
- **Permissions** (Разрешения);
- **Extended Properties** (Расширенные свойства);

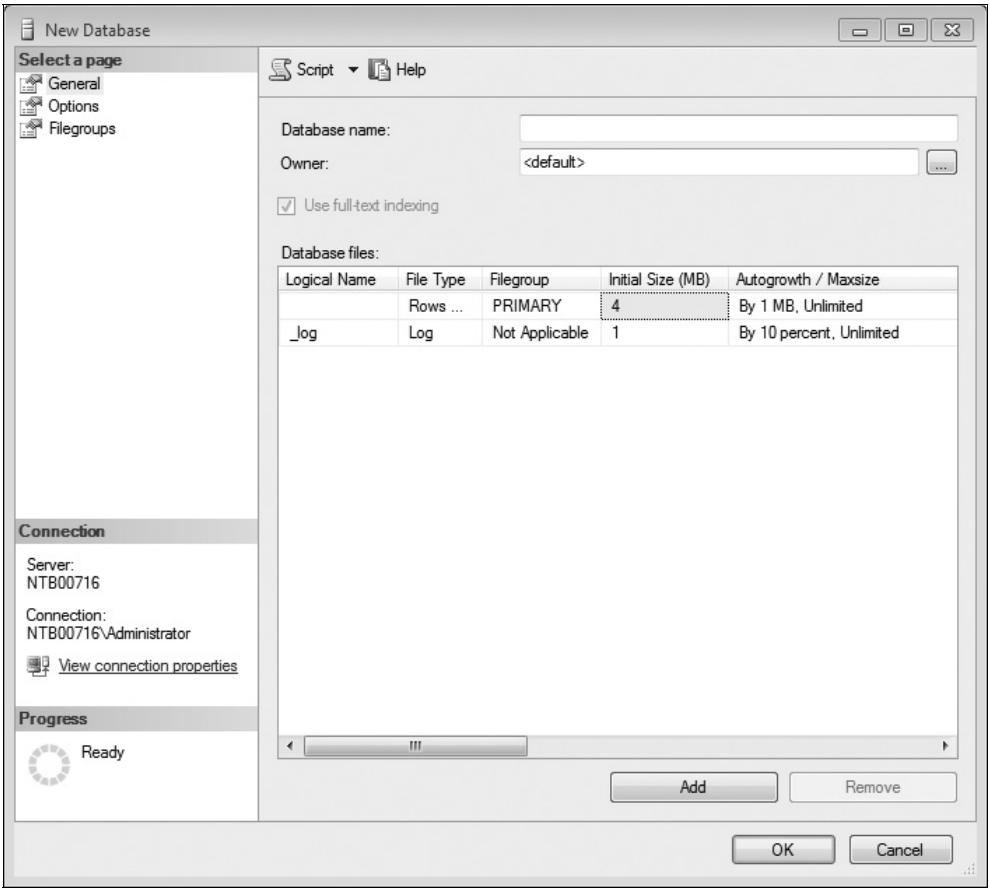


Рис. 3.5. Диалоговое окно **New Database**

- **Mirroring** (Зеркальное отображение);
- **Transaction Log Shipping** (Доставка журналов транзакций).

ПРИМЕЧАНИЕ

Для уже существующей базы данных отображаются все группы свойств в ранее перечисленном списке. Для создаваемой базы данных существует только три страницы свойств: **General** (Общие), **Options** (Параметры) и **Filegroups** (Файловые группы) (см. рис. 3.5).

Страница **General** (Общие) диалогового окна **Database Properties** (Свойства базы данных) (рис. 3.6) содержит, среди прочего, такую информацию, как имя, владелец и параметры сортировки базы данных.

Свойства файлов данных определенной базы данных перечисляются на странице **Files** (Файлы) и содержат такую информацию, как имя и начальный размер файла, расположение базы данных, а также тип файла (например, `PRIMARY`). База данных может храниться в нескольких файлах.

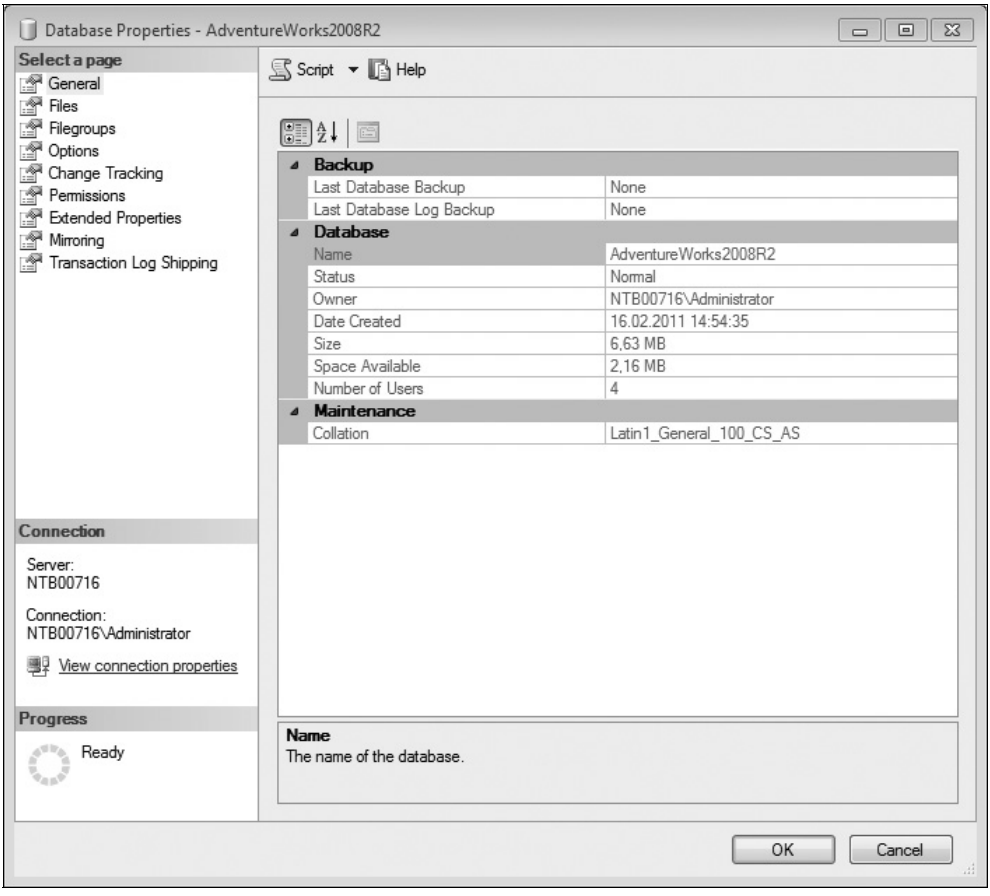


Рис. 3.6. Страница **General** диалогового окна **Database Properties**

ПРИМЕЧАНИЕ

В SQL Server применяется динамическое управление дисковым пространством. Это означает, что можно сконфигурировать размер базы данных для автоматического увеличения и уменьшения по мере надобности. Чтобы изменить свойство **Autogrowth** (Автоувеличение) на странице **Files** (Файлы), в столбце **Autogrowth** (Автоувеличение) нажмите значок троеточия (**...**) и внесите соответствующие изменения в диалоговом окне **Change Autogrowth** (Изменить авторасширение). Чтобы позволить автоматическое увеличение размера базы данных, нужно установить флажок **Enable Autogrowth** (Разрешить авторасширение). Каждый раз, когда существующий размер файла недостаточен для хранения добавляемых данных, сервер автоматически запрашивает систему выделить файлу дополнительное дисковое пространство. Объем дополнительного дискового пространства (в процентах или мегабайтах) указывается в поле **File Growth** (Увеличение размера файла) в том же диалоговом окне. А в разделе **Maximum File Size** (Максимальный размер файла) можно или ограничить максимальный размер файла, установив переключатель **Limited to (MB)** (Ограниченное (Мбайт)), или снять ограничения на размер, установив переключатель **Unlimited** (Без ограничений) (это настройка по умолчанию). При ограниченном размере файла нужно указать его допустимый максимальный размер.

На странице **Filegroups** (Файловые группы) диалогового окна **Database Properties** (Свойства базы данных) отображаются имена файловых групп, к которым принадлежит файл базы данных, раздел файловой группы (по умолчанию или заданный явно), а также операции, разрешенные для выполнения с файловой группой (чтение и запись или только чтение).

На странице **Options** (Параметры) диалогового окна **Database Properties** (Свойства базы данных) можно просмотреть и модифицировать все параметры уровня базы данных. Существуют следующие группы параметров: **Automatic** (Автоматически), **Containment** (Включение), **Cursor** (Курсор), **Miscellaneous** (Вспомогательные), **Recovery** (Восстановление), **Service Broker** (Компонент Service Broker) **и State** (Состояние).

Группа **State** (Состояние) содержит, например, следующие четыре параметра.

- **Database Read-Only** (База данных доступна только для чтения). Позволяет установить доступ к базе данных полный доступ или доступ только для чтения. В последнем случае пользователи не могут модифицировать данные. Значение по умолчанию этого параметра — `False`.
- **Restrict Access** (Ограничение доступа). Устанавливает количество пользователей, которые могут одновременно использовать базу данных. Значение по умолчанию — `MULTI_USER`.
- **Database State** (Состояние базы данных). Описывает состояние базы данных. Значение по умолчанию этого параметра — `Normal`.
- **Encryption Enabled** (Шифрование включено). Определяет режим шифрования базы данных. Значение по умолчанию этого параметра — `False`.

На странице **Extended Properties** (Расширенные свойства) отображаются дополнительные свойства текущей базы данных. На этой странице можно удалять существующие свойства и добавлять новые.

На странице **Permissions** (Разрешения) отображаются все пользователи, роли и соответствующие разрешения. (Тема разрешений подробно рассматривается в *главе 12*.)

Остальные страницы **Change Tracking** (Отслеживание изменений), **Mirroring** (Зеркальное отображение) и **Transaction Log Shipping** (Доставка журналов транзакций) описывают возможности, связанные с доступностью данных, и поэтому рассматриваются в *главе 16*.

Модифицирование баз данных, не прибегая к использованию языка Transact-SQL

С помощью обозревателя объектов можно модифицировать существующие базы данных, изменяя файлы и файловые группы базы данных. Чтобы добавить новые файлы в базу данных, щелкните правой кнопкой требуемую базу данных и в контекстном меню выберите пункт **Properties** (Свойства). В открывшемся диалоговом окне **Database Properties** (Свойства базы данных) выберите страницу **Files** (Файлы) и нажмите кнопку **Add** (Добавить), расположенную внизу раздела **Database files**

(Файлы базы данных). В раздел будет добавлена новая строка, в поле **Logical Name** (Логическое имя) которой следует ввести имя добавляемого файла базы данных, а в других полях задать необходимые свойства этого файла. Также можно добавить и вторичную файловую группу для базы данных, выбрав страницу **Filegroups** (Файловые группы) и нажав кнопку **Add** (Добавить).

ПРИМЕЧАНИЕ

Упомянутые ранее свойства базы данных может модифицировать только системный администратор или владелец базы данных.

Чтобы удалить базы данных с помощью обозревателя объектов, щелкните правой кнопкой имя требуемой базы данных и в открывшемся контекстном меню выберите пункт **Delete** (Удалить).

Управление таблицами, не прибегая к использованию языка Transact-SQL

Следующей задачей после создания базы данных является создание всех необходимых таблиц. Подобно созданию базы данных, таблицы в ней также можно создать либо с помощью языка Transact-SQL, либо посредством обозревателя объектов. Как и в случае с созданием базы данных, здесь мы рассмотрим создание таблиц только с помощью обозревателя объектов. (Создание таблиц и всех других объектов баз данных посредством языка Transact-SQL подробно рассматривается в *главе 5*.)

Для практики создания таблиц, в базе данных `sample` создадим таблицу `department`. Чтобы создать таблицу базы данных с помощью обозревателя объектов, разверните в нем узел **Databases** (Базы данных), а потом узел требуемой базы данных, щелкните правой кнопкой папку **Tables** (Таблицы) и в открывшемся контекстном меню выберите пункт **New Table** (Создать таблицу). В верхней части с правой стороны окна средства Management Studio откроется окно для создания столбцов новой таблицы. Введите имена столбцов таблицы, их типы данных и разрешение значений `NULL` для каждого столбца, как это показано в правой верхней панели на рис. 3.7.

Чтобы выбрать для столбца один из поддерживаемых системой типов данных, в столбце **Data Type** (Тип данных) выберите, а затем нажмите направленный вниз треугольник у правого края поля (этот треугольник появляется после того, как будет выбрана ячейка). В результате в открывшемся раскрывающемся списке выберите требуемый тип данных для столбца.

Тип данных существующего столбца можно изменить на вкладке **Column Properties** (Свойства столбца) (нижняя панель справа на рис. 3.7). Для одних типов данных, таких как `char`, требуется указать длину в строке **Length** (Длина), а для других, таких как `decimal`, на вкладке **Column Properties** (Свойства столбца) требуется указать масштаб и точность в соответствующих строках **Scale** (Масштаб) и **Precision** (Точность). Для некоторых других, таких как `int`, не требуется указывать

ни одно из этих свойств. (Недействительные значения для конкретного типа данных выделены затененным шрифтом в списке всех возможных свойств столбца.)

Чтобы разрешить значения `NULL` для данного столбца, следует установить для него соответствующий флажок поля. Также, если для столбца требуется значение по умолчанию, его следует ввести в строку **Default Value or Binding** (Значение по умолчанию или привязка) панели **Column Properties** (Свойства столбца). Значение по умолчанию присваивается ячейке столбца автоматически, если для нее явно не введено значение.

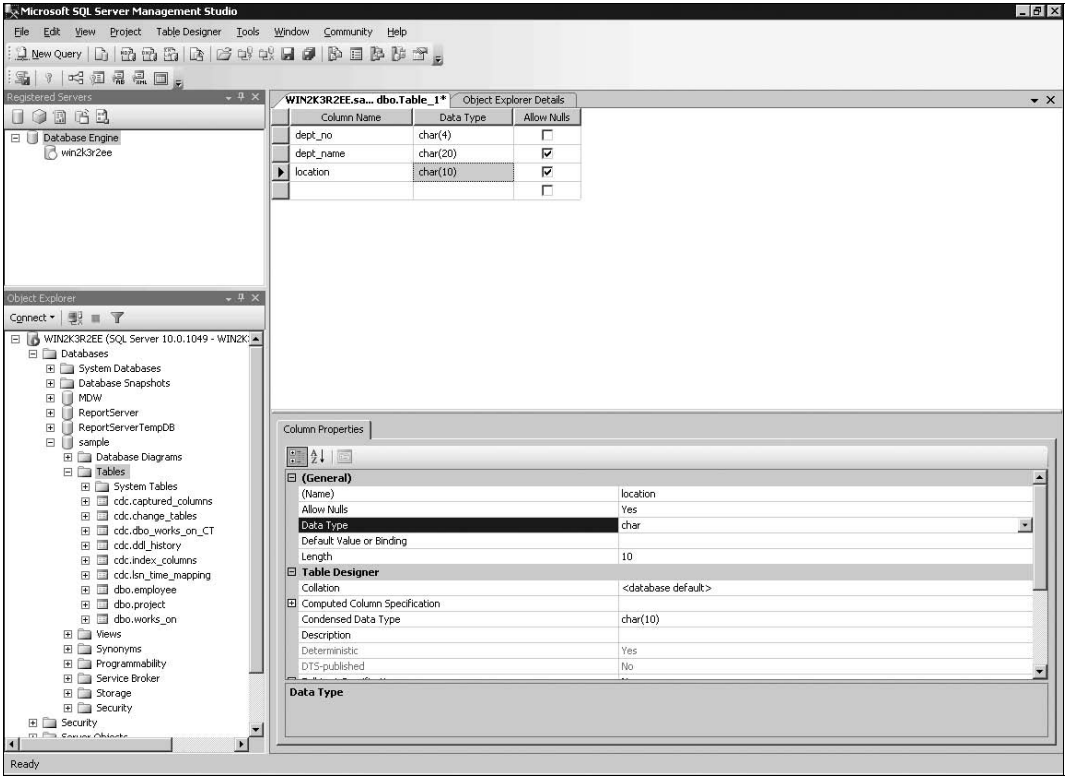


Рис. 3.7. Создание таблицы `department` базы данных `sample` посредством обозревателя объектов

Столбец `dept_no` является первичным ключом таблицы `department`. (Подробное обсуждение темы первичных ключей базы данных `sample` см. в *главе 1*). Чтобы сделать столбец первичным ключом таблицы, щелкните его правой кнопкой и в контекстном меню выберите пункт **Set Primary Key** (Задать первичный ключ). Завершив все работы по созданию таблицы, щелкните крестик вкладки конструктора таблиц. Откроется диалоговое окно с запросом, сохранить ли сделанные изменения. Нажмите кнопку **Yes** (Да), после чего откроется диалоговое окно **Choose Name** (Выбор имени) с запросом ввести имя таблицы. Введите требуемое имя таблицы и нажмите кнопку **OK**. Таблица будет сохранена под указанным именем. Чтобы ото-

бразить новую таблицу в иерархии базы данных, в панели инструментов обозревателя объектов щелкните значок **Renew** (Обновить).

Для просмотра и изменения свойств существующей таблицы разверните узел базы данных, содержащей требуемую таблицу, разверните узел **Tables** (Таблицы) в этой базе данных и щелкните правой кнопкой требуемую таблицу, а затем в контекстном меню выберите пункт **Properties** (Свойства). В результате для данной таблицы откроется диалоговое окно **Table Properties** (Свойства таблицы). Для примера, на рис. 3.8 показано диалоговое окно **Table Properties** (Свойства таблицы) на вкладке **General** (Общие) для таблицы `employee` базы данных `sample`.

Чтобы переименовать таблицу, в папке **Tables** (Таблицы) щелкните ее правой кнопкой в списке таблиц и в контекстном меню выберите пункт **Rename** (Переименовать). А чтобы удалить таблицу, щелкните ее правой кнопкой и выберите пункт **Delete** (Удалить).

ПРИМЕЧАНИЕ

На данном этапе следует создать три остальные таблицы базы данных `sample`.

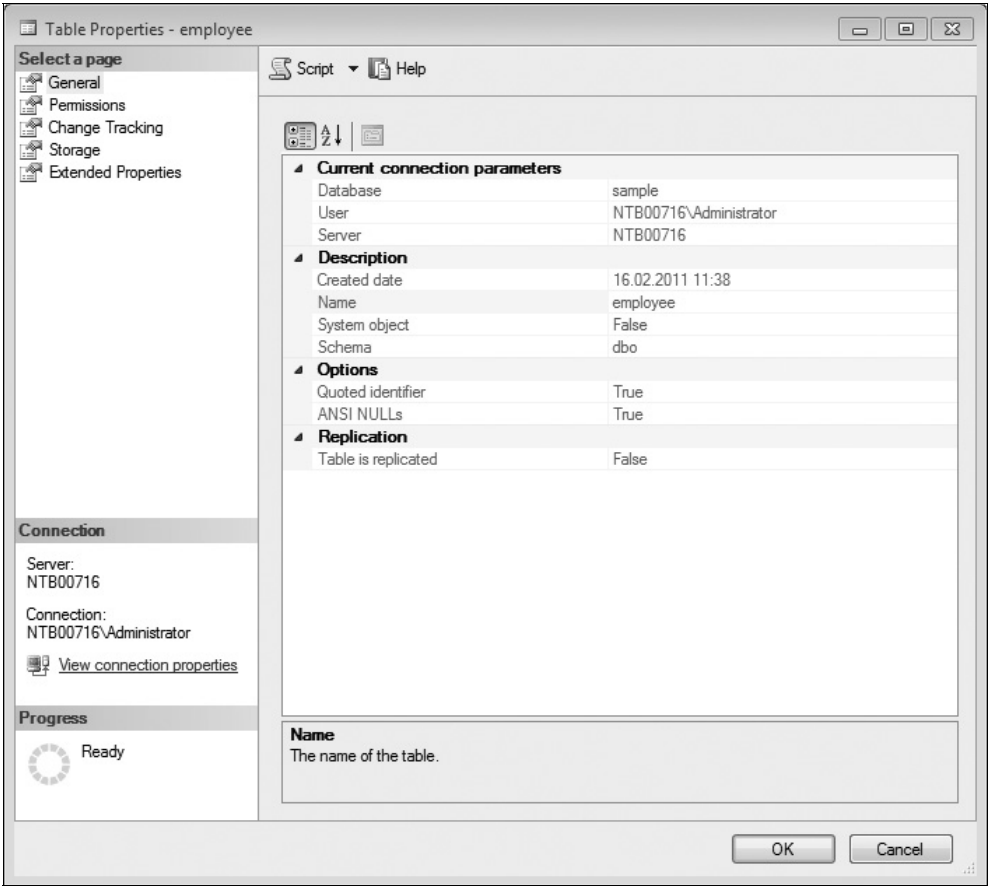


Рис. 3.8. Диалоговое окно **Table Properties**, вкладка **General** для таблицы `employee`

Создав все четыре таблицы базы данных `sample` (`employee`, `department`, `project` и `works_on`), можно использовать еще одну возможность среды SQL Server Management Studio, чтобы отобразить диаграмму типа "сущность — отошение" — диаграмму (ER) (entity-relationship) этой базы данных. (Процесс преобразования таблиц базы данных в диаграмму "сущность — отношение" (ER) называется *обратным проектированием*.)

Чтобы создать диаграмму "сущность — отношение" (ER) для базы данных `sample`, щелкните правой кнопкой ее подпапку **Database Diagrams** (Диаграммы баз данных) и в контекстном меню выберите пункт **New Database Diagram** (Создать диаграмму базы данных).

ПРИМЕЧАНИЕ

Если откроется диалоговое окно, в котором спрашивается, создавать ли вспомогательные объекты, выберите ответ **Yes** (Да).

Откроется диалоговое окно **Add Table** (Добавление таблицы), в котором нужно выбрать таблицы для добавления в диаграмму. Добавив все необходимые таблицы (в данном случае все четыре), нажмите кнопку **Close** (Закрыть), и мастер создаст диаграмму, подобную показанной на рис. 3.9.

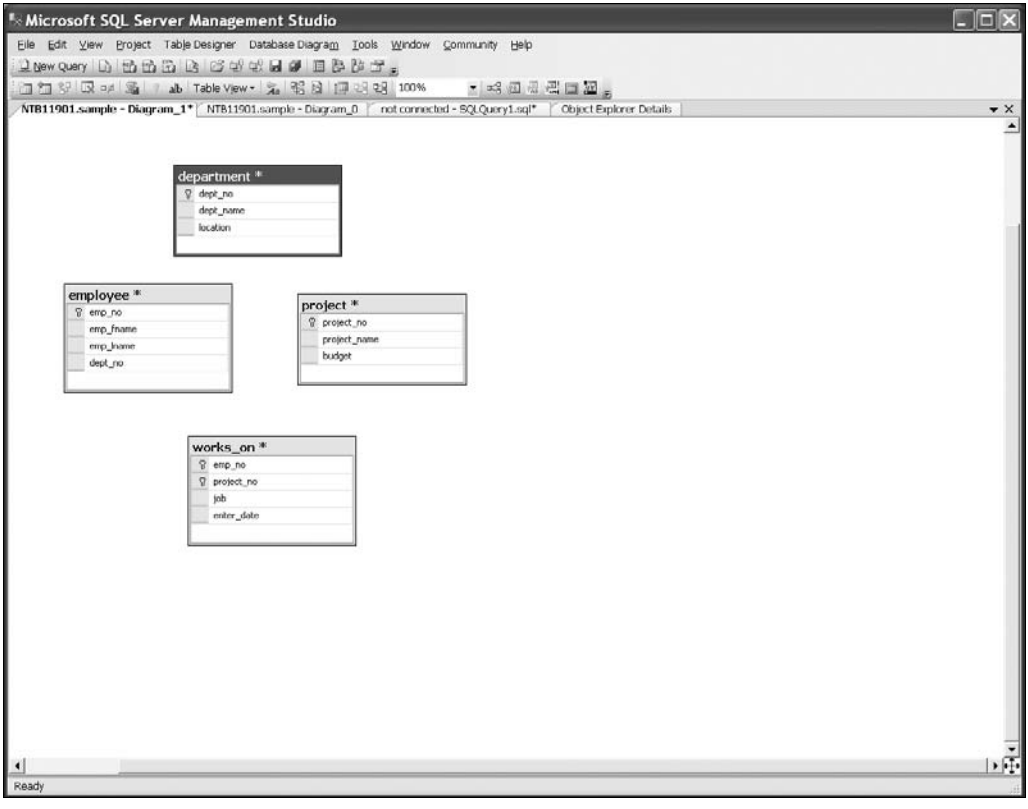


Рис. 3.9. Начальная диаграмма "сущность — отношение" (ER) базы данных `sample`

На рис. 3.9 показана только промежуточная, а не конечная, диаграмма "сущность — отношение" (ER) базы данных sample, поскольку, хотя на ней и показаны все четыре таблицы с их столбцами (и соответствующими первичными ключами), на ней все же отсутствуют отношения между таблицами. Отношение между двумя таблицами основывается на первичном ключе одной из таблиц и возможным соответствующим столбцом (или столбцами) другой таблицы. (Подробное обсуждение темы этих отношений и целостности ссылочных данных см. в *главе 5*.)

Между таблицами базы данных sample существует три отношения. Таблица department имеет отношение типа 1:N с таблицей employee, поскольку каждому значению первичного ключа таблицы department (столбец dept_no) соответствует одно или более значений столбца dept_no таблицы employee. Аналогично существует отношение между таблицами employee и works_on, поскольку только значения, которые присутствуют в столбце первичного ключа таблицы employee (emp_no) также имеются в столбце emp_no таблицы works_on. Третье отношение существует между таблицами project и works_on, т. к. только значения, которые присутствуют в первичном ключе таблицы project (project_no) также присутствуют в столбце project_no таблицы works_on.

Чтобы создать эти три отношения, диаграмму "сущность — отношение" (ER) нужно реконструировать, указав для каждой таблицы столбцы, которые соответствуют ключевым столбцам других таблиц. Такой столбец называется *внешним ключом* (foreign key). Чтобы увидеть, как это делается, определим столбец dept_no таблицы employee, как внешний ключ таблицы department. Для этого выполним следующие действия:

1. В созданной диаграмме щелкните правой кнопкой графическое представление таблицы employee и в контекстном меню выберите пункт **Relationships** (Отношения). В открывшемся диалоговом окне **Foreign Key Relationships** (Связи по внешнему ключу) нажмите кнопку **Add** (Добавить).
2. В правой панели диалогового окна расширьте первый столбец, выберите в нем строку **Table and Columns Specification** (Спецификация таблиц и столбцов) и нажмите кнопку с троеточием во втором столбце этой строки.
3. В открывшемся диалоговом окне **Tables and Columns** (Таблицы и столбцы) в раскрывающемся списке **Primary key table** (Таблица первичного ключа) выберите таблицу с соответствующим первичным ключом. В данном случае это будет таблица department.
4. Выберите для этой таблицы столбец dept_no в качестве первичного ключа и этот же столбец для таблицы employee в качестве внешнего ключа, после чего нажмите кнопку **OK**, чтобы закрыть окно **Tables and Columns** (Таблицы и столбцы). Нажмите кнопку **Close** (**Закрыть**), чтобы закрыть окно **Foreign Key Relationships** (Связи по внешнему ключу).

Подобным образом создаются и другие два отношения. На рис. 3.10 показана диаграмма "сущность — отношение" (ER), отображающая все три отношения между таблицами базы данных sample.

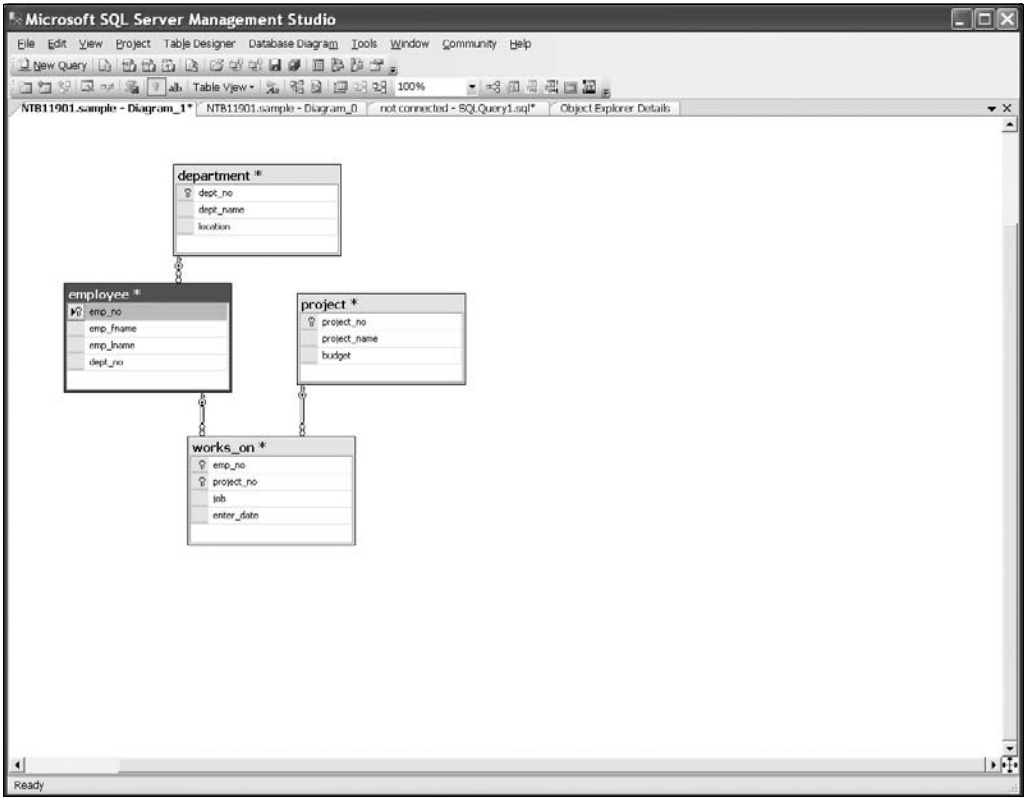


Рис. 3.10. Конечная диаграмма "сущность — отношение" (ER) базы данных `sample`

Разработка запросов, используя среду SQL Server Management Studio

Среда SQL Server Management Studio предоставляет завершенное средство для создания всех типов запросов. С ее помощью можно создавать, сохранять, загружать и редактировать запросы. Кроме этого, над запросами можно работать без подключения к какому-либо серверу. Этот инструмент также предоставляет возможность разрабатывать запросы для разных проектов.

Предоставляется возможность работать с запросами как посредством редактора запросов, так и с помощью обозревателя решений. В этом разделе рассматриваются оба эти инструмента. Кроме этих двух компонентов среды SQL Server Management Studio мы рассмотрим отладку SQL-кода, используя встроенный отладчик.

Редактор запросов

Чтобы открыть панель редактора запросов **Query Editor** (Редактор запросов), на панели инструментов среды SQL Server Management Studio нажмите кнопку **New Query** (Создать запрос). Эту панель можно расширить, чтобы отображать кнопки

создания всех возможных запросов, а не только запросов компонента Database Engine. По умолчанию создается новый запрос компонента Database Engine, но, нажав соответствующую кнопку на панели инструментов, можно также создавать запросы MDX, XMLA и др.

Строка состояния внизу панели редактора запросов указывает статус подключения редактора к серверу. Если подключение к серверу не выполнено автоматически, при запуске редактора запросов выводится диалоговое окно подключения к серверу (см. рис. 3.1), в котором можно выбрать сервер для подключения и режим проверки подлинности.

ПРИМЕЧАНИЕ

Редактирование запросов в автономном режиме предоставляет больше гибкости, чем при подключении к серверу. Для редактирования запросов не обязательно подключаться к серверу, и окно редактора запросов можно отключить от одного сервера (выбрав последовательность команд из меню **Query | Connection | Disconnect** (Запрос | Соединение | Отключить) и подключить к другому, не открывая другого окна редактора. Чтобы выбрать автономный режим редактирования, в диалоговом окне подключения к серверу, открывающемуся при запуске редактора конкретного вида запросов, просто нажмите кнопку **Cancel** (Отмена).

Редактор запросов можно использовать для выполнения следующих задач:

- создания и выполнения инструкций языка Transact-SQL;
- сохранения созданных инструкций языка Transact-SQL в файл;
- создания и анализирования планов выполнения общих запросов;
- графического иллюстрирования плана выполнения выбранного запроса.

Редактор запросов содержит встроенный текстовый редактор и панель инструментов с набором кнопок для разных действий. Главное окно редактора запросов разделено по горизонтали на панель запросов (вверху) и панель результатов (внизу). Инструкции Transact-SQL (т. е. запросы) для исполнения вводятся в верхнюю панель, а результаты обработки системой этих запросов отображаются в нижней панели. На рис. 3.11 показан пример ввода запроса в редактор запросов и результатов выполнения этого запроса.

В первой инструкции запроса `USE` указывается использовать базу данных `sample` в качестве текущей базы данных. Вторая инструкция — `SELECT` — извлекает все строки таблицы `works_on`. Чтобы выполнить этот запрос и вывести результаты, в панели инструментов редактора запросов нажмите кнопку **Execute** (Выполнить) или клавишу <F5>.

ПРИМЕЧАНИЕ

Можно открыть несколько окон редактора запросов, т. е. выполнить несколько подключений к одному или нескольким экземплярам компонента Database Engine. Новое подключение создается нажатием кнопки **New Query** (Создать запрос) в панели инструментов среды SQL Server Management Studio.

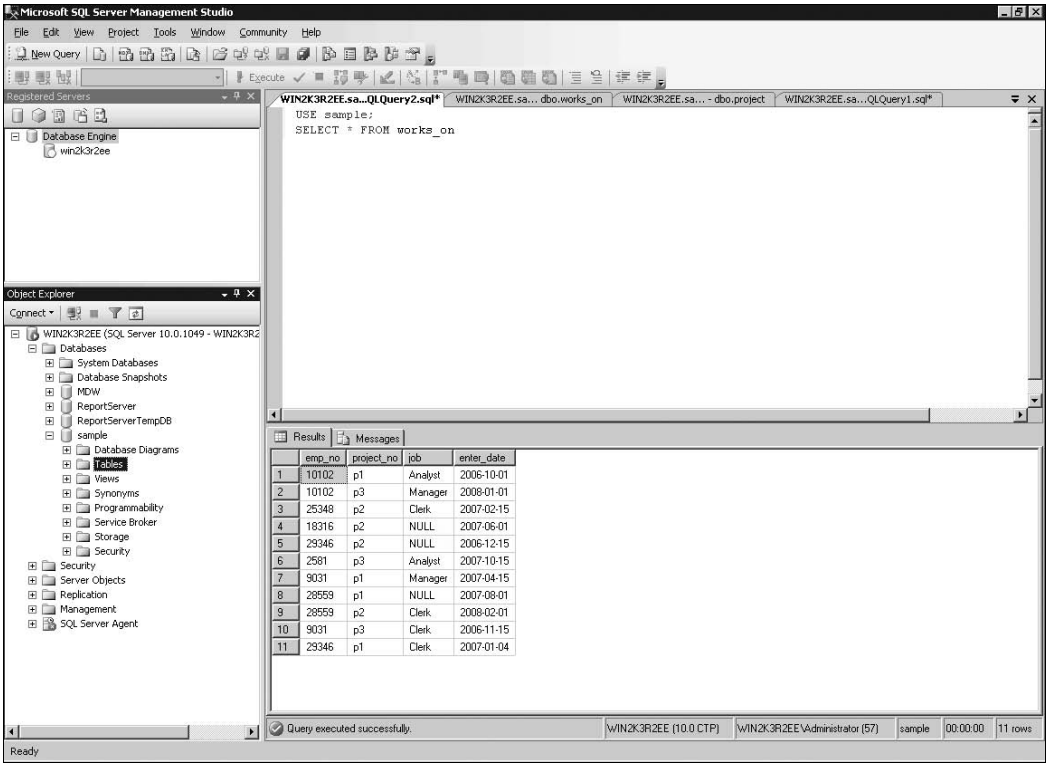


Рис. 3.11. Редактор запросов с запросом (верхняя панель) и результатами его выполнения (нижняя панель)

В строке состояния внизу окна редактора запросов отображается следующая информация, связанная с выполнением инструкций запроса:

- состояние текущей операции (например, "Запрос успешно выполнен");
- имя сервера базы данных;
- имя текущего пользователя и идентификатор серверного процесса;
- имя текущей базы данных;
- время, затраченное на выполнение последнего запроса;
- количество найденных строк.

Одним из основных достоинств среды SQL Server Management Studio является легкость ее использования, что также относится и к редактору запросов Query Editor. Редактор запросов поддерживает множество возможностей, облегчающих задачу кодирования инструкций языка Transact-SQL. В частности, в нем используется подсветка синтаксиса, чтобы улучшить читаемость инструкций языка Transact-SQL. Все зарезервированные слова отображаются синим цветом, переменные — черным, строки — красным, а комментарии — зеленым. (Зарезервированные слова рассматриваются в *главе 4*.)

Кроме этого, редактор запросов оснащен контекстно-зависимой справкой, называющейся *Dynamic Help*, посредством которой можно получить сведения о кон-

кретной инструкции. Если вы не знаете синтаксиса инструкции, выделите ее в редакторе, а потом нажмите клавишу <F1>. Также можно выделить параметры различных инструкций Transact-SQL, чтобы получить справку по ним из электронной документации.

ПРИМЕЧАНИЕ

В SQL Server 2012 поддерживается инструмент *SQL Intellisense*, который является видом средства автозавершения. Иными словами, этот модуль предлагает наиболее вероятное завершение частично введенных элементов инструкций Transact-SQL.

С редактированием запросов может также помочь обозреватель объектов. Например, если вы хотите узнать, как создать инструкцию `CREATE TABLE` для таблицы `employee`, щелкните правой кнопкой эту таблицу в обозревателе объектов и в появившемся контекстном меню выберите пункты **Script Table As | CREATE to | New Query Editor Window** (Создать скрипт для таблицы | Используя CREATE | Новое окно редактора запросов). Окно редактора запросов, содержащее созданную таким образом инструкцию `CREATE TABLE`, показано на рис. 3.12. Эта возможность также применима и с другими объектами, такими как хранимые процедуры и функции.

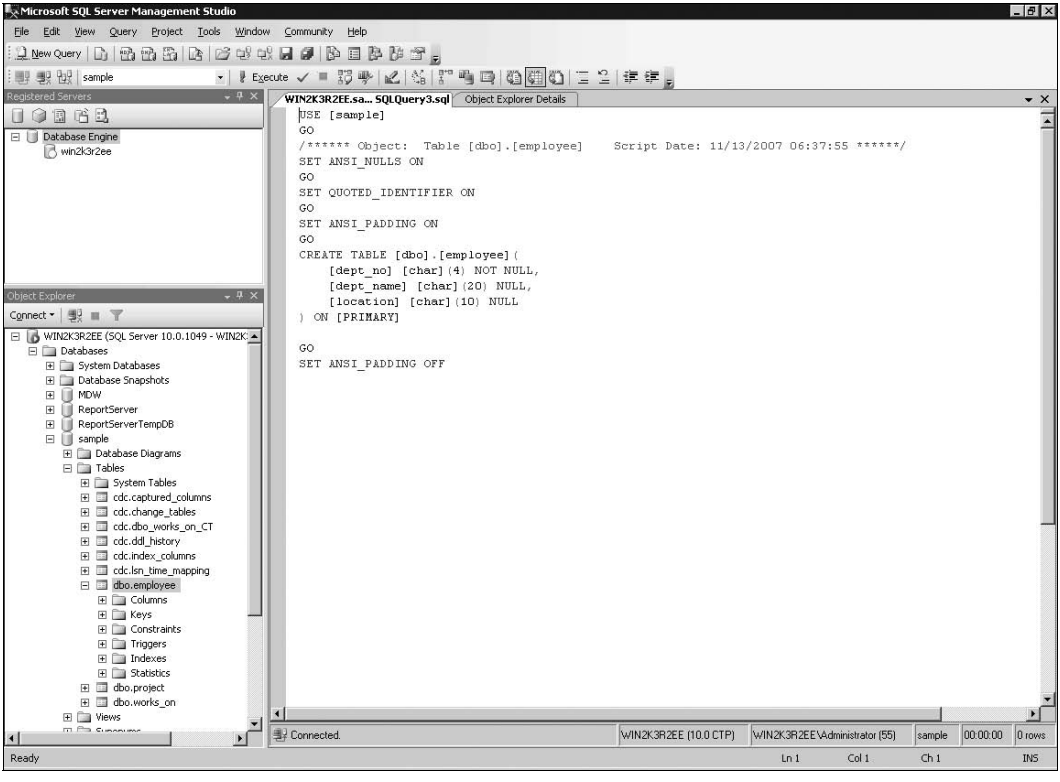


Рис. 3.12. Окно редактора запросов, содержащее созданную в нем инструкцию `CREATE TABLE`

Обозреватель объектов очень полезен для графического отображения плана исполнения конкретного запроса. *Планом выполнения запроса* называется вариант выполнения, выбранный оптимизатором запроса среди нескольких возможных вариантов выполнения конкретного запроса. Введите в верхнюю панель редактора требуемый запрос, выберите последовательность команд из меню **Query | Deispaly Estimated Execution Plan** (Запрос | Показать предполагаемый план выполнения) и в нижней панели окна редактора будет показан план выполнения данного запроса. Эта тема подробно рассматривается в *главе 19*.

Обозреватель решений

Редактирование запросов в среде SQL Server Management Studio основано на методе решений. Если создать пустой запрос с помощью кнопки **New Query** (Создать запрос), то он будет основан на пустом решении. Это можно увидеть, выполнив последовательность команд из меню **View | Solution Explorer** (Вид | Обозреватель решений) сразу же после открытия пустого запроса.

Решение может быть связано ни с одним, с одним или с несколькими проектами. Пустое решение, не связано ни с каким проектом. Чтобы связать проект с решением, закройте пустое решение, обозреватель решений и редактор запросов и создайте новый проект, выполнив последовательность команд из меню **File | New | Project** (Файл | Создать | Проект). В открывшемся окне **New Project** (Создать проект) выберите в средней панели опцию **SQL Server Scripts** (Скрипты SQL Server). *Проект* — это способ организации файлов в определенном месте. Проекту можно присвоить имя и выбрать место для его расположения на диске. При создании нового проекта автоматически запускается новое решение. Проект можно добавить к существующему решению с помощью обозревателя решений.

Для каждого созданного проекта в обозревателе решений отображаются папки **Connections** (Соединения), **Queries** (Запросы) и **Miscellaneous** (Разное). Чтобы открыть новое окно редактора запросов для данного проекта, щелкните правой кнопкой его папку **Queries** (Запросы) и в контекстном меню выберите пункт **New Query** (Создать запрос).

Отладка SQL Server

SQL Server, начиная с версии SQL Server 2008, оснащен встроенным отладчиком кода. Чтобы начать сеанс отладки, выберите в главном меню среды SQL Server Management Studio следующую последовательность команды **Debug | Start Debugging** (Отладка | Начать отладку). Мы рассмотрим работу отладчика на примере с использованием пакета из *главы 8* (см. *пример 8.1*). *Пакетом* называется последовательность инструкций SQL и процедурных расширений, составляющих логическое целое, отправляемая компоненту Database Engine для выполнения всех содержащихся в ней инструкций. На рис. 3.13 показан пакет, который подсчитывает количество сотрудников, работающих над проектом `p1`. Если это количество равно 4 или больше, то выводится соответствующее сообщение. В противном случае выводятся имена и фамилии сотрудников.

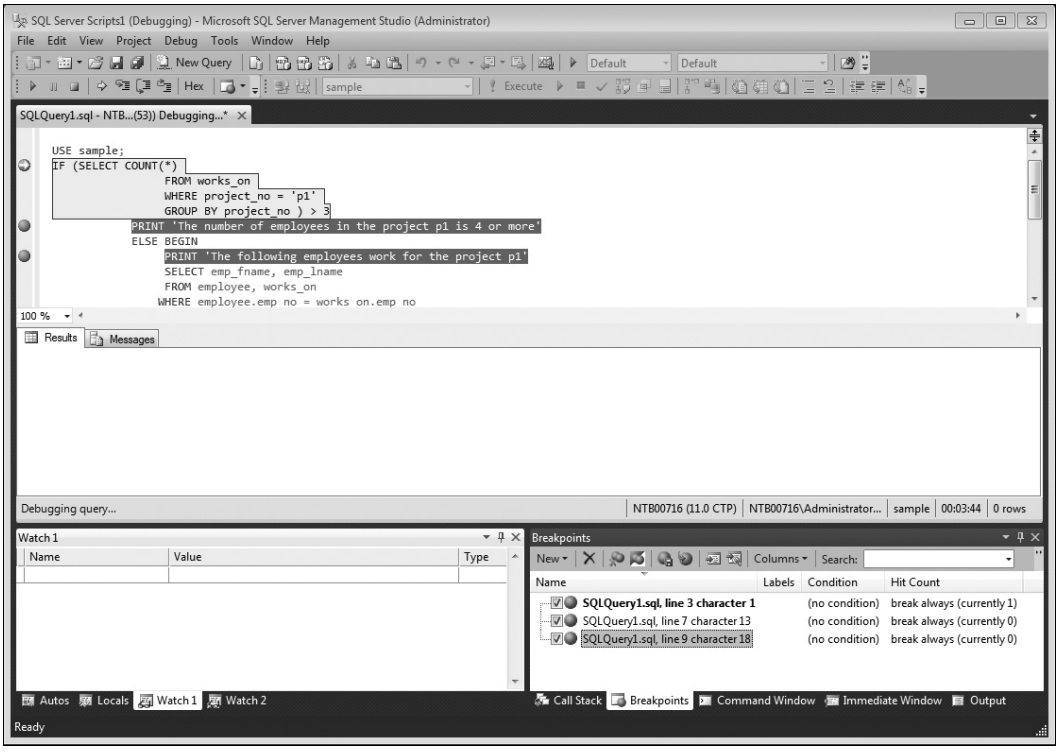


Рис. 3.13. Отладка кода SQL

Чтобы остановить выполнение пакета на определенной инструкции, можно установить точки останова, как это показано на рис. 3.13. Для этого нужно щелкнуть слева от строки, на которой нужно остановиться. В начале отладки выполнение останавливается на первой линии кода, которая отмечается желтой стрелкой. Чтобы продолжить выполнение и отладку, выберите следующую последовательность команд из меню **Debug | Continue** (Отладка | Продолжить). Выполнение инструкций пакета продолжится до первой точки останова, и желтая стрелка остановится на этой точке.

Информация, связанная с процессом отладки, отображается в двух панелях внизу окна редактора запросов. Информация о разных типах информации об отладке сгруппирована в этих панелях на нескольких вкладках. Левая панель содержит вкладку **Autos** (Автоматические), **Locals** (Локальные) и до пяти вкладок **Watch** (Видимые). Правая панель содержит вкладки **Call Stack** (Стек вызовов), **Threads** (Потоки), **Breakpoints** (Точки останова), **Command Window** (Окно команд), **Immediate Window** (Окно интерпретации) и **Output** (Вывод). На вкладке **Locals** (Локальные) отображаются значения переменных, на вкладке **Call Stack** (Стек вызовов) — значения стека вызовов, а на вкладки **Breakpoints** (Точки останова) — информация о точках останова.

Чтобы завершить процесс отладки, выполните последовательность команд из главного меню **Debug | Stop Debugging** (Отладка | Остановить отладку) или нажмите синюю кнопку на панели инструментов отладчика.

В SQL Server 2012 функциональность встроенного в SQL Server Management Studio отладчика расширена несколькими новыми возможностями. Теперь в нем можно выполнять ряд следующих операций.

- *Указывать условие точки останова. Условие точки останова* — это SQL-выражение, вычисленное значение которого определяет, будет ли выполнение кода остановлено в данной точке или нет. Чтобы указать условие точки останова, щелкните правой кнопкой красный значок требуемой точки и в контекстном меню выберите пункт **Condition** (Условие). Откроется диалоговое окно **Breakpoint Condition** (Условие для точки останова), в котором нужно ввести необходимое логическое выражение. Кроме этого, если нужно остановить выполнение, в случае если выражение верно, то следует установить переключатель **Is True** (Верно). Если же выполнение нужно остановить, если выражение изменилось, то нужно установить переключатель **Has Changed** (Изменилось).
- *Указать число попаданий в точку останова. Число попаданий* — это условие останова выполнения в данной точке в зависимости от количества раз, когда была достигнута эта точка останова в процессе выполнения. При достижении указанного числа прохождений и любого другого условия, указанного для данной точки останова, отладчик выполняет указанное действие. Условие прерывания выполнения на основе числа попаданий может быть одним из следующих:
  - безусловное (действие по умолчанию) (**Break always**);
  - если число попаданий равно указанному значению (**Break when the his count equals a specified value**);
  - если число попаданий кратно указанному значению (**Break when the hit count equals a multiple of a specified value**);
  - если число попаданий равно или больше указанного значения (**Break when the his count is greater or equal to a specified value**).

  Чтобы задать число попаданий в процессе отладки, щелкните правой кнопкой значок требуемой точки останова на вкладке **Breakpoints** (Точки останова), в контекстном меню выберите пункт **Hit Count** (Число попаданий), затем в открывшемся диалоговом окне **Breakpoint Hit Count** (Число попаданий в точку останова) выберите одно из условий из приведенного ранее списка. Для опций, требующих значение, введите его в текстовое поле справа от раскрывающегося списка условий. Чтобы сохранить указанные условия, нажмите кнопку **OK**.
- *Указывать фильтр точки останова.* Фильтр точки останова ограничивает работу останова только на указанных компьютерах, процессах или потоках. Чтобы установить фильтр точки останова, щелкните правой кнопкой требуемую точку и в контекстном меню выберите пункт **Filter** (Фильтр). Затем в открывшемся диалоговом окне **Breakpoint Filters** (Фильтр точки останова) укажите ресурсы, которыми нужно ограничить выполнение данной точки останова. Чтобы сохранить указанные условия, нажмите кнопку **OK**.
- *Указывать действие в точке останова.* Условие **When Hit** (При попадании) (в точку останова) указывает действие, которое нужно выполнить, когда выполнение пакета попадает в данную точку останова. По умолчанию, когда удовле-

творяются как условие количества попаданий, так и условие останова, тогда выполнение прерывается. Альтернативно можно вывести заранее указанное сообщение.

Чтобы указать действие при попадании в точку останова, щелкните правой кнопкой красный значок требуемой точки и выберите в контекстном меню пункт **When Hit** (При попадании). В открывшемся диалоговом окне **When Breakpoint is Hit** (При попадании в точку останова) выберите требуемое действие. Чтобы сохранить указанные условия, нажмите кнопку **OK**.

- *Использовать окно быстрой проверки Quick Watch*. В окне **QuickWatch** (Быстрая проверка) можно просмотреть значение выражения Transact-SQL, а потом сохранить это выражение в окне просмотра значений **Watch** (Просмотр значений). Чтобы открыть окно **Quick Watch** (Быстрая проверка), в меню **Debug** (Отладка) выберите пункт **Quick Watch** (Быстрая проверка). Выражение в этом окне можно или выбрать из раскрывающегося списка **Expression** (Выражение), или ввести его в это поле.
- *Использовать всплывающую подсказку Quick Info*. При наведении указателя мыши на идентификатор кода средство **Quick Info** (Краткие сведения) отображает его объявление во всплывающем окне.

Резюме

В этой главе мы рассмотрели самый важный инструмент сервера SQL Server: среду управления SQL Server Management Studio. Среда SQL Server Management Studio очень полезна как для конечных пользователей, так и для администраторов. С ее помощью можно осуществлять многие функции администрирования. Эти функции были только слегка затронуты в этой главе, но рассматриваются более подробно далее в книге. В этой главе мы познакомились с наиболее важными для конечного пользователя функциями среды SQL Server Management Studio, такими как создание баз данных и таблиц.

Среда управления SQL Server Management Studio содержит, среди прочих, следующие инструменты:

- *Зарегистрированные серверы* (Registered Servers) — используется для регистрации экземпляров SQL Server и подключения к ним;
- *Обозреватель объектов* (Object Explorer) — содержит в виде дерева представление всех объектов баз данных сервера;
- *Редактор запросов* (Query Editory) — позволяет конечным пользователям создавать, выполнять и сохранять инструкции Transact-SQL. Кроме этого, данный инструмент предоставляет возможность для анализа запросов, отображая план выполнения;
- *Обозреватель решений* (Solution Explorer) — используется для создания решений; причем решение может быть связано ни с одним, с одним или с несколькими проектами;
- *Отладчик* (Debugger) — используется для отладки SQL-кода.

В следующей главе дается введение в язык Transact-SQL и рассматриваются его основные компоненты. После этого обсуждаются системные функции, поддерживаемые языком Transact-SQL.

Упражнения

Упражнение 3.1

Используя среду SQL Server Management Studio, создайте базу данных под названием `test`. Сохраните эту базу данных в файле `testdate_a` в папку C:\tmp и выделите для этого файла 10 Мбайт дискового пространства. Установите параметры файла базы данных для автоматического увеличения размера с шагом по 2 Мбайта до максимального размера 20 Мбайт.

Упражнение 3.2

С помощью среды SQL Server Management Studio измените параметры журнала транзакций для базы данных `test`. Разрешите авторасширение файла журнала транзакций и установите его начальный размер в 3 Мбайта и увеличение размера с шагом по 20%.

Упражнение 3.3

Используя среду SQL Server Management Studio, установите разрешения на использование базы данных `test` только ее владельцем и системным администратором. Могут ли оба эти пользователя обращаться к этой базе данных одновременно?

Упражнение 3.4

Используя среду SQL Server Management Studio, создайте все четыре таблицы базы данных `sample` со всеми их столбцами (см. *главу 1*).

Упражнение 3.5

Используя среду SQL Server Management Studio, просмотрите, какие таблицы содержит база данных `AdventureWorks`. Затем выберите в этой базе данных таблицу `Person.Address` и просмотрите ее свойства.

Упражнение 3.6

Введите и выполните в редакторе запросов следующую инструкцию Transact-SQL:

```
CREATE DATABASE test
```

Объясните причину сообщения об ошибке, выводимого в панели результатов.

Упражнение 3.7

Сохраните инструкцию Transact-SQL из упражнения 3.6 в файл C:\tmp\createdb.sql.

Упражнение 3.8

Как можно в редакторе запросов сделать базу данных `test` текущей базой данных?

Упражнение 3.9

Используя редактор запросов, сделайте базу данных `AdventureWorks` текущей и выполните следующую инструкцию Transact-SQL:

```
SELECT * FROM Sales.Customer
```

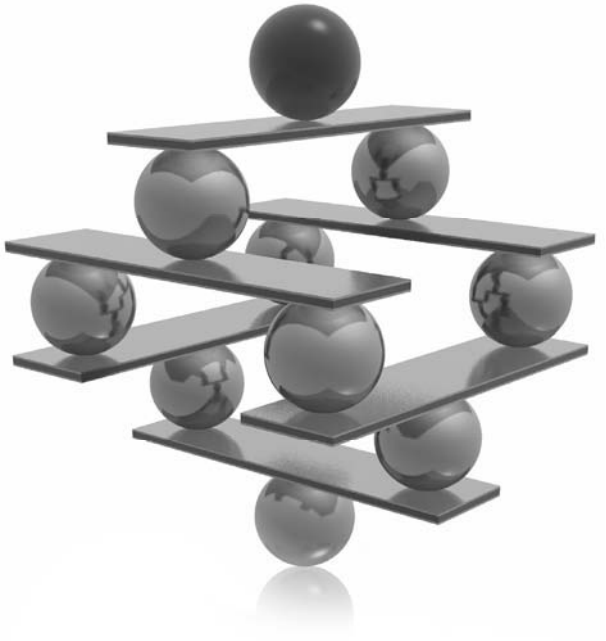
Как можно остановить исполнение этой инструкции?

Упражнение 3.10

Используя редактор запросов, измените вывод инструкции `SELECT` из упражнения 3.9 так, чтобы результаты выводились в виде текста, а не в виде таблицы.

Часть II

Язык Transact-SQL



Глава 4

Компоненты SQL

- **Основные объекты SQL**
- **Типы данных**
- **Функции языка SQL**
- **Скалярные операторы**
- **Значения *NULL***

В этой главе представлено введение в основные объекты и базовые операторы языка Transact-SQL. Сначала рассматриваются базовые элементы языка, включая константы, идентификаторы и ограничители. Далее, поскольку каждый простой объект имеет соответствующий тип данных, подробно рассматриваются типы данных. Кроме этого, объясняются все операторы и функции. В конце главы представлены значения `NULL`.

Основные объекты SQL

Языком компонента Database Engine является Transact-SQL, который обладает основными свойствами любого другого распространенного языка программирования. Эта такие свойства, как:

- литералы (или константы);
- ограничители;
- комментарии;
- идентификаторы;
- зарезервированные ключевые слова.

Все эти особенности описываются в следующих разделах.

Литералы

Литерал — это буквенно-цифровая (строковая), шестнадцатеричная или числовая константа. Строковая константа содержит один или больше символов определенного набора символов, заключенных между одинарными (`' '`) или двойными (`" "`) прямыми кавычками. (Константы предпочтительно заключать в одинарные кавычки, т. к. двойные кавычки используются во многих других случаях, как мы увидим вскоре.) Чтобы вставить одинарную кавычку в строку, заключенную в одинарные кавычки, нужно использовать две одинарные кавычки последовательно. Шестнадцатеричные константы используются для представления непечатаемых символов и прочих двоичных данных. Они начинаются символами `0x`, за которыми следует четное число буквенных или цифровых символов. В примерах 4.1 и 4.2 представлена иллюстрация некоторых допустимых и недопустимых строковых и шестнадцатеричных констант.

Пример 4.1. Допустимые строковые и шестнадцатеричные константы

```
'Philadelphia'
"Berkeley, CA 94710"
'9876'
0x53514C0D
'Апостроф представляется таким образом: can''t'
```

(Обратите внимание в последней строке примера на использование двух последовательных одинарных кавычек для представления апострофа в строковых константах.)

Пример 4.2. Недопустимые строковые константы

`'AB'C'` — нечетное число одинарных кавычек;

`'New York"` — строка должна быть заключена в одинаковые кавычки, т. е. одинарные или двойные с каждого конца строки.

К числовым константам относятся все целочисленные значения и значения с фиксированной и плавающей запятой (точкой). В примере 4.3 иллюстрируется несколько числовых констант.

Пример 4.3. Числовые константы

```
130
-130.00
```

`-0.357E5` — экспоненциальное представление, где `nEm` означает `n` умножено на `10` в степени `m`

```
22.3E-3
```

Константы всегда обладают такими свойствами, как тип данных и длина, которые оба зависят от формата константы. Кроме этого, числовые константы имеют дополнительные свойства — точность и коэффициент масштабирования (scale factor). (Типы данных разных видов литералов рассматриваются в *этой главе далее*.)

Ограничители

В языке Transact-SQL двойные кавычки имеют два разных применения. Кроме применения для заключения строк, они также могут использоваться в качестве ограничителей для так называемых *идентификаторов с ограничителями* (delimited identifier). Идентификатор с ограничителями — это особый вид идентификатора, который обычно применяется для того, чтобы разрешить использование ключевых слов в качестве идентификаторов, а также разрешить использование пробелов в именах объектов баз данных.

ПРИМЕЧАНИЕ

Различие между одинарными и двойными кавычками было впервые введено в стандарте SQL92. Применительно к идентификаторам, этот стандарт различает между обычными идентификаторами и идентификаторами с ограничителями. Два ключевых отличия состоят в том, что идентификаторы с ограничителями заключаются в двойные кавычки и чувствительны к регистру. (В языке Transact-SQL также поддерживается применение квадратных скобок вместо двойных кавычек.) Двойные кавычки применяются только в качестве ограничителя строк. По большому счету, идентификаторы с ограничителями были введены для того, чтобы позволить определять идентификаторы, которые иначе идентичны зарезервированным ключевым словам. В частности, идентификаторы с ограничителями предотвращают использование имен (идентификаторов и имен переменных), которые могут быть введены в качестве зарезервированных ключевых слов в будущих стандартах SQL. Кроме этого, идентификаторы с ограничителями могут содержать символы, которые, как правило, не разрешаются в именах обычных идентификаторов, например, пробелы.

Применение двойных кавычек в языке Transact-SQL определяется с помощью параметра `QUOTED_IDENTIFIER` инструкции `SET`. Если этому параметру присвоено значение `ON` (значение по умолчанию), идентификаторы, заключенные в двойные кавычки, будут определяться как идентификаторы с ограничителями. В таком случае двойные кавычки нельзя будет применять для ограничения строк.

Комментарии

В языке Transact-SQL существует два способа определения комментариев. В первом, текст, заключенный между парами символов `/*` и `*/`, является комментарием. Такой комментарий может занимать несколько строк. В другом способе два символа дефиса (`--`) указывают, что следующий за ними до конца текущей строки текст является комментарием. (Способ обозначения комментариев двумя дефисами отвечает стандарту ANSI SQL, а способ с помощью символов `/*` и `*/` является расширением языка Transact-SQL.)

Идентификаторы

В языке Transact-SQL идентификаторы применяются для обозначения объектов баз данных, таких как собственно базы данных, их таблицы и индексы. Идентификатор состоит из строки длиной до 128 символов, которая может содержать буквы, цифры и символы _, @, # и &. Первым символом идентификатора должна быть буква или символ _, @ или #. Символ # в начале имени таблицы или хранимой процедуры обозначает временный объект, а символ @ обозначает переменную. Как упоминалось ранее, эти правила не относятся к идентификаторам с ограничителями (также называемыми идентификаторами в скобках), которые могут содержать или начинаться с любого символа, иного, чем сам ограничитель.

Зарезервированные ключевые слова

Каждый язык программирования имеет набор зарезервированных имен, которые требуется писать и применять в определенном формате. Такие имена называются *зарезервированными ключевыми словами*. В языке Transact-SQL используются разные виды таких имен, которые, как и во многих других языках программирования, нельзя применять в качестве имен объектов, за исключением, когда они используются для этой цели, как идентификаторы с ограничителями (или идентификаторы в скобках).

ПРИМЕЧАНИЕ

В языке Transact-SQL имена всех типов данных и системных функций, такие как CHARACTER и INTEGER, не являются зарезервированными ключевыми словами. Поэтому такие слова можно использовать для обозначения объектов. Тем не менее настоятельно рекомендуется не делать этого, т. к. такая практика влечет за собой трудности в чтении и понимании инструкций Transact-SQL.

Типы данных

Все значения в столбце должны быть одного типа данных. (Единственным исключением из этого правила являются значения типа данных SQL_VARIANT.) Используемые в Transact-SQL типы данных можно разбить на следующие категории:

- числовые типы;
- символьные типы;
- временные типы (даты и/или времени);
- прочие типы данных.

Все эти категории данных рассматриваются далее в последующих разделах.

Числовые типы данных

Как и следовало ожидать по их названию, числовые типы данных применяются для представления чисел. Эти типы и их краткое описание приводятся в табл. 4.1.

Таблица 4.1. *Числовые типы данных*

Тип данных	Описание
INTEGER	Представляет целочисленные значения длиной в 4 байта в диапазоне от –2 147 483 648 до 2 147 483 647. INT — сокращенная форма от INTEGER
SMALLINT	Представляет целочисленные значения длиной в 2 байта в диапазоне от –32 768 до 32 767
TINYINT	Представляет целочисленные значения длиной в 1 байт в диапазоне от 0 до 255
BIGINT	Представляет целочисленные значения длиной в 8 байт в диапазоне от -2^{63} до $2^{63} - 1$
DECIMAL(p,[s])	Представляет значения с фиксированной запятой (точкой). Аргумент p (precision — точность) указывает общее количество разрядов, а аргумент s (scale — степень) — количество разрядов справа от полагаемой десятичной точки. В зависимости от значения аргумента p, значения DECIMAL сохраняются в 5 до 17 байтах. DEC — сокращенная форма от DECIMAL
NUMERIC(p,[s])	Синоним DECIMAL
REAL	Применяется для представления значений с плавающей точкой. Диапазон положительных значений простирается приблизительно от 2,23E –308 до 1,79E +308, а отрицательных приблизительно от –1,18E –38 до –1,18E +38. Также может быть представлено и нулевое значение
FLOAT[(p)]	Подобно типу REAL, представляет значения с плавающей точкой. Аргумент p определяет точность. При значении p < 25 представляемые значения имеют одинарную точность (требуют 4 байта для хранения), а при значении p >= 25 — двойную точность (требуют 8 байтов для хранения)
MONEY	Используется для представления денежных значений. Значения типа MONEY соответствуют 8-байтовым значениям типа DECIMAL, округленным до четырех разрядов после десятичной точки
SMALLMONEY	Представляет такие же значения, что и тип MONEY, но длиной в 4 байта

Символьные типы данных

Существует два общих вида символьных типов данных. Строки могут представляться однобайтовыми символами или же символами в кодировке Unicode. (В кодировке Unicode для представления одного символа применяется несколько байтов.) Кроме этого, строки могут быть разной длины. В табл. 4.2 перечислены категории символьных типов данных с их кратким описанием.

Таблица 4.2. *Символьные типы данных*

Тип данных	Описание
CHAR[(n)]	Применяется для представления строк фиксированной длины, состоящих из n однобайтовых символов. Максимальное значение n равно 8000. CHARACTER(n) — альтернативная эквивалентная форма CHAR(n). Если n явно не указано, то его значение полагается равным 1
VARCHAR[(n)]	Используется для представления строки однобайтовых символов переменной длины (0 < n < 8 000). В отличие от типа данных CHAR, количество байтов для хранения значений типа данных VARCHAR равно их действительной длине. Этот тип данных имеет два синонима: CHAR VARYING и CHARACTER VARYING
NCHAR[(n)]	Используется для хранения строк фиксированной длины, состоящих из символов в кодировке Unicode. Основная разница между типами данных CHAR и NCHAR состоит в том, что для хранения каждого символа строки типа NCHAR требуется 2 байта, а строки типа CHAR — 1 байт. Поэтому строка типа данных NCHAR может содержать самое большее 4000 символов
NVARCHAR[(n)]	Используется для хранения строк переменной длины, состоящих из символов в кодировке Unicode. Основная разница между типами данных VARCHAR и NVARCHAR состоит в том, что для хранения каждого символа строки типа NVARCHAR требуется 2 байта, а строки типа VARCHAR — 1 байт. Поэтому строка типа данных NVARCHAR может содержать самое большее 4000 символов

ПРИМЕЧАНИЕ

Тип данных VARCHAR идентичен типу данных CHAR, за исключением одного различия: если содержимое строки CHAR(n) короче, чем n символов, остаток строки заполняется пробелами. А количество байтов, занимаемых строкой типа VARCHAR, всегда равно количеству символов в ней.

Временные типы данных

В языке Transact-SQL поддерживаются следующие временные типы данных:

- DATETIME;
- SMALLDATETIME;
- DATE;
- TIME;
- DATETIME2;
- DATETIMEOFFSET.

Типы данных DATETIME и SMALLDATETIME применяются для хранения даты и времени в виде целочисленных значений длиной в 4 и 2 байта соответственно. Значения типа DATETIME и SMALLDATETIME сохраняются внутренне как два отдельных числовых значения. Составляющая даты значений типа DATETIME хранится в диапазоне от 01/01/1753 до 31/12/9999, а соответствующая составляющая значений типа

DATETIME — в диапазоне от 01/01/1900 до 06/06/2079. Составляющая времени хранится во втором 4-байтовом (2-байтовом для значений типа SMALLDATETIME) поле в виде числа трехсотых долей секунды (для DATETIME) или числа минут (для SMALLDATETIME), истекших после полуночи.

Если нужно сохранить только составляющую даты или времени, использование значений типа DATETIME или SMALLDATETIME несколько неудобно. По этой причине в SQL Server были введены типы данных DATE и TIME, в которых хранятся только составляющие даты и времени значений типа DATETIME, соответственно. Значения типа DATE занимают 3 байта, представляя диапазон дат от 01/01/0001 до 31/12/9999. Значения типа TIME занимают 3—5 байт и представляют время с точностью до 100 нс.

Тип данных DATETIME2 используется для представления значений дат и времени с высокой точностью. В зависимости от требований, значения этого типа можно определять разной длины, и занимают они от 6 до 8 байтов. Составляющая времени представляет время с точностью до 100 нс. Этот тип данных не поддерживает переход на летнее время.

Все рассмотренные на данный момент временные типы данных не поддерживают часовые пояса. Тип данных DATETIMEOFFSET имеет составляющую для хранения смещения часового пояса. По этой причине значения этого типа занимают от 6 до 8 байтов. Все другие свойства этого типа данных аналогичны соответствующим свойствам типа данных DATETIME2.

Значения дат в Transact-SQL по умолчанию определены в виде строки формата *'ммм дд гггг'* (например, 'Jan 10 1993'), заключенной в одинарные или двойные кавычки. (Но относительный порядок составляющих месяца, дня и года можно изменять с помощью инструкции SET DATEFORMAT. Кроме этого, система поддерживает числовые значения для составляющей месяца и разделители / и –.) Подобным образом, значение времени указывается в 24-часовом формате в виде *'чч:мм'* (например, '22:24').

ПРИМЕЧАНИЕ

Язык Transact-SQL поддерживает различные форматы ввода значений типа DATETIME. Как уже упоминалось, каждая составляющая определяется отдельно, поэтому значения дат и времени можно указать в любом порядке или отдельно. Если одна из составляющих не указывается, система использует для него значение по умолчанию. (Значение по умолчанию для времени — 12:00 AM (до полудня).)

В примерах 4.4 и 4.5 показаны разные способы представления значений даты и времени в разных форматах.

Пример 4.4. Действительные представления даты

'28/5/1959' (с использованием инструкции SET DATEFORMAT dmy)

'May 28, 1959'

'1959 MAY 28'

Пример 4.5. Действительные представления времени

```
'8:45 AM'
'4 pm'
```

Прочие типы данных

Язык Transact-SQL поддерживает несколько типов данных, которые не принадлежат не к одной из вышеперечисленных групп типов данных. В частности:

- двоичные типы данных;
- битовый тип данных `BIT`;
- тип данных больших объектов;
- тип данных `CURSOR` (рассматривается в *главе 8*);
- тип данных `UNIQUEIDENTIFIER`;
- тип данных `SQL_VARIANT`;
- тип данных `TABLE` (рассматривается в *главах 5* и *8*);
- тип данных `XML` (рассматривается в *главе 26*);
- пространственные типы данных (рассматриваются в *главе 27*);
- тип данных `HIERARCHYID`;
- тип данных `TIMESTAMP`;
- типы данных, определяемые пользователем (рассматриваются в *главе 5*).

Все эти типы данных рассматриваются в последующих разделах этой главы (за исключением тех, для которых указано, что они обсуждаются в других главах).

Двоичные и битовые типы данных

К двоичным типам данным принадлежат два типа: `BINARY` и `VARBINARY`. Эти типы данных описывают объекты данных во внутреннем формате системы и используются для хранения битовых строк. По этой причине значения этих типов вводятся, используя шестнадцатеричные числа.

Значения битового типа `BIT` содержат лишь один бит, вследствие чего в одном байте можно сохранить до восьми значений этого типа. Краткое описание свойств двоичных и битовых типов данных приводится в табл. 4.3.

Таблица 4.3. *Двоичные и битовые типы данных*

Тип данных	Описание
`BINARY[(n)]`	Определяет строку битов фиксированной длины, содержащую ровно `n` байтов (0 < n < 8000)
`VARBINARY[(n)]`	Определяет строку битов переменной длины, содержащую до `n` байтов (0 < `n` < 8000)

Таблица 4.3 *(окончание)*

Тип данных	Описание
BIT	Применяется для хранения логических значений, которые могут иметь три возможных состояния: FALSE, TRUE и NULL

Тип данных больших объектов

Тип данных LOB (Large OBject — большой объект) используется для хранения объектов данных размером до 2 Гбайт. Такие объекты обычно применяются для хранения больших объемов текстовых данных и для загрузки подключаемых модулей и аудио- и видеофайлов. В языке Transact-SQL поддерживаются следующие типы данных LOB:

- VARCHAR(max);
- NVARCHAR(max);
- VARBINARY(max).

Начиная с версии SQL Server 2005, для обращения к значениям стандартных типов данных и к значениям типов данных LOB применяется одна и та же модель программирования. Иными словами, для работы с объектами LOB можно использовать удобные системные функции и строковые операторы.

В компоненте Database Engine параметр max применяется с типами данных VARCHAR, NVARCHAR и VARBINARY для определения значений столбцов переменной длины. Когда вместо явного указания длины значения используется значение длины по умолчанию max, система анализирует длину конкретной строки и принимает решение, сохранять ли эту строку как обычное значение или как значение LOB. Параметр max указывает, что размер значений столбца может достигать максимального размера LOB данной системы.

Хотя решение о способе хранения объектов LOB принимается системой, настройки по умолчанию можно переопределить, используя системную процедуру sp_tableoption с аргументом LARGE_VALUE_TYPES_OUT_OF_ROW. Если значение этого аргумента равно 1, то данные в столбцах, объявленных с использованием параметра max, будут сохраняться отдельно от остальных данных. Если же значение аргумента равно 0, то компонент Database Engine сохраняет все значения размером до 8060 байт в строке таблицы, как обычные данные, а значения большего размера хранятся вне строки в области хранения объектов LOB.

Начиная с версии SQL Server 2008, для столбцов типа VARBINARY(max) можно применять атрибут FILESTREAM, чтобы сохранять данные BLOB (Binary Large OBject — большой двоичный объект) непосредственно в файловой системе NTFS. Основным достоинством этого атрибута является то, что размер соответствующего объекта LOB ограничивается только размером тома файловой системы. (Этот атрибут более подробно рассматривается далее в *разд. "Варианты хранения" этой главы*.)

Тип данных *UNIQUEIDENTIFIER*

Как можно судить по его названию, тип данных UNIQUEIDENTIFIER является однозначным идентификационным номером, который сохраняется в виде 16-байтовой двоичной строки. Этот тип данных тесно связан с идентификатором GUID (Globally Unique Identifier — глобально уникальный идентификатор), который гарантирует однозначность в мировом масштабе. Таким образом, этот тип данных позволяет однозначно идентифицировать данные и объекты в распределенных системах.

Инициализировать столбец или переменную типа UNIQUEIDENTIFIER можно посредством функции NEWID или NEWSEQUENTIALID, а также с помощью строковой константы особого формата, состоящей из шестнадцатеричных цифр и дефисов. (Функции NEWID и NEWSEQUENTIALID рассматриваются далее в *разд. "Системные функции" этой главы*.)

К столбцу со значениями типа данных UNIQUEIDENTIFIER можно обращаться, используя в запросе ключевое слово ROWGUIDCOL, чтобы указать, что столбец содержит значения идентификаторов. (Это ключевое слово не генерирует никаких значений.) Таблица может содержать несколько столбцов типа UNIQUEIDENTIFIER, но только один из них может иметь ключевое слово ROWGUIDCOL.

Тип данных *SQL_VARIANT*

Тип данных SQL_VARIANT можно использовать для хранения значений разных типов одновременно, таких как числовые значения, строки и даты. (Исключением являются значения типа TIMESTAMP.) Каждое значение столбца типа SQL_VARIANT состоит из двух частей: собственно значения и информации, описывающей это значение. Эта информация содержит все свойства действительного типа данных значения, такие как длина, масштаб и точность.

Для доступа и отображения информации о значениях столбца типа SQL_VARIANT применяется функция Transact-SQL SQL_VARIANT_PROPERTY. Использование типа данных SQL_VARIANT рассматривается в примере 5.5 в *главе 5*.

ПРИМЕЧАНИЕ

Объявлять тип столбца как SQL_VARIANT следует только в том случае, если это действительно необходимо. Например, если столбец предназначается для хранения значений разных типов данных или если при создании таблицы тип данных, которые будут храниться в данном столбце, неизвестен.

Тип данных *HIERARCHYID*

Тип данных HIERARCHYID используется для хранения полной иерархии. Например, в значении этого типа можно сохранить иерархию всех сотрудников или иерархию папок. Этот тип реализован в виде определяемого пользователем типа CLR (Common Language Runtime — общеязыковая среда исполнения), который охватывает несколько системных функций для создания узлов иерархии и работы с ними.

Следующие функции, среди прочих, принадлежат к методам этого типа данных: `GetLevel()`, `GetAncestor()`, `GetDescendant()`, `Read()` и `Write()`. (Подробное рассмотрение этого типа находится вне рамок этой книги.)

Тип данных *TIMESTAMP*

Тип данных `TIMESTAMP` указывает столбец, определяемый как `VARBINARY(8)` или `BINARY(8)`, в зависимости от свойства столбца принимать значения `NULL`. Для каждой базы данных система содержит счетчик, значение которого увеличивается всякий раз, когда вставляется или обновляется любая строка, содержащая ячейку типа `TIMESTAMP`, и присваивает этой ячейке данное значение. Таким образом, с помощью ячеек типа `TIMESTAMP` можно определить относительное время последнего изменения соответствующих строк таблицы. (`ROWVERSION` является синонимом `TIMESTAMP`.)

ПРИМЕЧАНИЕ

Само по себе значение, сохраняемое в столбце типа `TIMESTAMP`, не представляет никакой важности. Этот столбец обычно используется для определения, изменилась ли определенная строка таблицы со времени последнего обращения к ней.

Варианты хранения

Начиная с версии SQL Server 2008, существует два разных варианта хранения, каждый из которых позволяет сохранять объекты LOB и экономить дисковое пространство. Это следующие варианты:

- хранение данных типа `FILESTREAM`;
- хранение с использованием разреженных столбцов (sparse columns).

Эти варианты хранения рассматриваются в следующих подразделах.

Хранение данных типа *FILESTREAM*

Как уже упоминалось ранее, SQL Server поддерживает хранение больших объектов (LOB) посредством типа данных `VARBINARY(max)`. Свойство этого типа данных таково, что большие двоичные объекты (BLOB) сохраняются в базе данных. Это обстоятельство может вызвать проблемы с производительностью в случае хранения очень больших файлов, таких как аудио- или видеофайлов. В таких случаях эти данные сохраняются вне базы данных во внешних файлах.

Хранение данных типа `FILESTREAM` поддерживает управление объектами LOB, которые сохраняются в файловой системе NTFS. Основным преимуществом этого типа хранения является то, что хотя данные хранятся вне базы данных, управляются они базой данных. Таким образом, этот тип хранения имеет следующие свойства:

- данные типа `FILESTREAM` можно сохранять с помощью инструкции `CREATE TABLE`, а для работы с этими данными можно использовать инструкции для модифицирования данных (`SELECT`, `INSERT`, `UPDATE` и `DELETE`);

- система управления базой данных обеспечивает такой же самый уровень безопасности для данных типа FILESTREAM, как и для данных, хранящихся внутри базы данных.

Подробно создание данных типа FILESTREAM рассматривается в *главе 5*.

Разреженные столбцы

Цель варианта хранения, предоставляемого разреженными столбцами, значительно отличается от цели хранения типа FILESTREAM. Тогда как целью хранения типа FILESTREAM является хранение объектов LOB вне базы данных, целью разреженных столбцов является минимизировать дисковое пространство, занимаемое базой данных. Столбцы этого типа позволяют оптимизировать хранение столбцов, большинство значений которых равны NULL. (Значения NULL рассматриваются в конце этой главы.) При использовании разреженных столбцов для хранения значений NULL дисковое пространство не требуется, но, с другой стороны, для хранения значений, отличных от NULL, требуется дополнительно от 2 до 4 байтов, в зависимости от их типа. По этой причине разработчики Microsoft рекомендуют использовать разреженные столбцы только в тех случаях, когда ожидается, по крайней мере, 20% общей экономии дискового пространства.

Разреженные столбцы определяются таким же образом, как и прочие столбцы таблицы; аналогично осуществляется и обращение к ним. Это означает, что для обращения к разреженным столбцам можно использовать инструкции SELECT, INSERT, UPDATE и DELETE таким же образом, как и при обращении к обычным столбцам. (Эти инструкции SQL подробно рассматриваются в *главах 6* и *7*.) Единственная разница касается создания разреженных столбцов: для определения конкретного столбца разреженным применяется аргумент SPARSE после названия столбца, как это показано в данном примере:

```
имя_столбца тип_данных SPARSE
```

Несколько разреженных столбцов таблицы можно сгруппировать в набор столбцов. Такой набор будет альтернативным способом сохранять значения во всех разреженных столбцах таблицы и обращаться к ним. Дополнительную информацию о наборах столбцов см. в *электронной документации*.

Функции языка SQL

Функции языка Transact-SQL могут быть агрегатными или скалярными. Эти типы функций рассматриваются в последующих разделах.

Агрегатные функции

Агрегатные функции выполняют вычисления над группой значений столбца и всегда возвращают одно значение результата этих вычислений.

Язык Transact-SQL поддерживает несколько групп агрегатных функций. В частности, следующие:

- обычные агрегатные функции;
- статистические агрегатные функции;
- агрегатные функции, определяемые пользователем;
- аналитические агрегатные функции.

Статистические и аналитические агрегатные функции рассматриваются в *главе 23*. Агрегатные функции, определяемые пользователем, выходят за рамки этой книги. Далее мы будем рассматривать только следующие обычные агрегатные функции:

- функция `AVG` — вычисляет среднее арифметическое значение данных, содержащихся в столбце. Значения, над которыми выполняется вычисление, должны быть числовыми;
- функции `MAX` и `MIN` — определяют максимальное и минимальное значение из всех значений данных, содержащихся в столбце. Значения могут быть числовыми, строковыми или временными (дата/время);
- функция `SUM` — вычисляет общую сумму значений в столбце. Значения, над которыми выполняется вычисление, должны быть числовыми;
- функция `COUNT` — подсчитывает количество значений, отличных от `NULL` в столбце. Функция `COUNT(*)` является единственной агрегатной функцией, которая не выполняет вычисления над столбцами. Эта функция возвращает количество строк (независимо от того, содержат ли отдельные столбцы значения `NULL`);
- функция `COUNT_BIG` — аналогична функции `COUNT`, с той разницей, что возвращает значение данных типа `BIGINT`.

Использование обычных агрегатных функций в инструкции `SELECT` рассматривается в *главе 6*.

Скалярные функции

Скалярные функции Transact-SQL используются в создании скалярных выражений. (Скалярная функция выполняет вычисления над одним значением или списком значений, тогда как агрегатная функция выполняет вычисления над группой значений из нескольких строк.) Скалярные функции можно разбить на следующие категории:

- числовые функции;
- функции даты;
- строковые функции;
- системные функции;
- функции метаданных.

Эти типы функций рассматриваются в последующих разделах.

Числовые функции

Числовые функции языка Transact-SQL — это математические функции для модифицирования числовых значений. Список числовых функций и их краткое описание приводится в табл. 4.4.

Таблица 4.4. *Числовые функции Transact-SQL*

Функция	Описание
ABS(*n*)	Возвращает абсолютное значение (т. е. отрицательные значения возвращаются, как положительные) числового выражения *n*. Примеры: SELECT ABS(-5.767) = 5.767 SELECT ABS(6.384) = 6.384
ACOS(*n*)	Вычисляет арккосинус значения *n*. Исходное значение *n* и результат имеют тип данных FLOAT
ASIN(*n*)	Вычисляет арксинус значения *n*. Исходное значение *n* и результат имеют тип данных FLOAT
ATAN(*n*)	Вычисляет арктангенс значения *n*. Исходное значение *n* и результат имеют тип данных FLOAT
ATN2(*n*,*m*)	Вычисляет арктангенс значения *n*/*m*. Исходные значения *n* и *m* и результат имеют тип данных FLOAT
CEILING(*n*)	Возвращает наименьшее целое значение, большее или равное указанному параметру. Примеры: SELECT CEILING(4.88) = 5 SELECT CEILING(-4.88) = -4
COS(*n*)	Вычисляет косинус значения *n*. Исходное значение *n* и результат имеют тип данных FLOAT
COT(*n*)	Вычисляет котангенс значения *n*. Исходное значение *n* и результат имеют тип данных FLOAT
DEGREES(*n*)	Преобразовывает радианы в градусы. Примеры: SELECT DEGREES(PI()/2) = 90 SELECT DEGREES(0.75) = 42
EXP(*n*)	Вычисляет значение e^n. Пример: SELECT EXP(1) = 2.7183
FLOOR(*n*)	Возвращает наибольшее целое значение, меньшее или равное указанному параметру. Пример: SELECT FLOOR(4.88) = 4
LOG(*n*)	Вычисляет натуральный логарифм (т. е. с основанием e) числа *n*. Примеры: SELECT LOG(4.67) = 1.54 SELECT LOG(0.12) =-2.12

Таблица 4.4 *(окончание)*

Функция	Описание
LOG10(*n*)	Вычисляет десятичный (с основанием 10) логарифм числа *n*. Примеры: SELECT LOG10(4.67) = 0.67 SELECT LOG10(0.12) =-0.92
PI()	Возвращает значение π (3,14)
POWER(*x*,*y*)	Вычисляет значение *x* в степени *y*. Примеры: SELECT POWER(3.12,5) = 295.65 SELECT POWER(81,0.5) = 9
RADIANS(*n*)	Преобразовывает градусы в радианы. Примеры: SELECT RADIANS(90.0) = 1.57 SELECT RADIANS(42.97) = 0.75
RAND()	Возвращает произвольное число типа FLOAT в диапазоне значений между 0 и 1
ROUND(*n*,*p*,[*t*])	Округляет значение *n* с точностью до *p*. Когда аргумент *p* положительное число, округляется дробная часть числа *n*, а когда отрицательное — целая часть. При использовании необязательного аргумента $t \neq 0$, число *n* не округляется, а усекается. Примеры: SELECT ROUND(5.4567,3) = 5.4570 SELECT ROUND(345.4567-1) = 350.0000 SELECT ROUND(345.4567-1,1) = 340.0000
ROWCOUNT_BIG	Возвращает количество строк таблицы, которые были обработаны последней инструкцией Transact-SQL, исполненной системой. Возвращаемое значение имеет тип BIGINT
SIGN(*n*)	Возвращает знак значения *n* в виде числа: +1, если положительное, –1, если отрицательное. Пример: SELECT SIGN(0.88) = 1.00
SIN(*n*)	Вычисляет синус значения *n*. Исходное значение *n* и результат имеют тип данных FLOAT
SQRT(*n*)	Вычисляет квадратный корень числа *n*. Пример: SELECT SQRT(9) = 3
SQUARE(*n*)	Возвращает квадрат аргумента *n*. Пример: SELECT SQUARE(9) = 81
TAN(*n*)	Вычисляет тангенс аргумента *n*. Исходное значение *n* и результат имеют тип данных FLOAT

Функции даты

Функции даты вычисляют соответствующие части даты или времени выражения или возвращают значение временного интервала. Поддерживаемые в Transact-SQL функции даты и их краткое описание приводятся в табл. 4.5.

Таблица 4.5. *Функции даты*

Функция	Описание
`GETDATE()`	Возвращает текущую системную дату и время. Пример: `SELECT GETDATE() = 2012-03-31 13:03:31.390`

`DATEPART(`*`item`*`, `*`date`*`)`	Возвращает указанную в параметре *`item`* часть даты *`date`* в виде целого числа.
	Примеры:
	`SELECT DATEPART(month, '01.01.2005') = 3`
	(1 = Январь)
	`SELECT DATEPART(weekday, '01.01.2005') = 2`
	(2 = Вторник)

`DATENAME(`*`item`*`, `*`date`*`)`	Возвращает указанную в параметре *`item`* часть даты *`date`* в виде строки символов. Пример: `SELECT DATENAME(weekday, '01.01.2005') = Sanday`

`DATEDIFF(`*`item`*`, `*`dat1`*`, `*`dat2`*`)`	Вычисляет разницу между двумя частями дат *`dat1`* и *`dat2`* и возвращает целочисленный результат в единицах, указанных в аргументе *`item`*.
	Пример (возвращает возраст каждого служащего):
	`SELECT DATEDIFF(year, BirthDate, GETDATE())`
	`AS age FROM employee`

`DATEADD(`*`i`*`, `*`n`*`, `*`d`*`)`	Прибавляет *`n`*-е количество единиц, указанных в аргументе *`i`* к указанной дате *`d`*. (Значение аргумента *`n`* также может быть отрицательным.) Пример (добавляет три дня к дате приема на работу каждого служащего; см. базу данных `sample`): `SELECT DATEADD(DAY,3,HireDate) AS age FROM employee`

Строковые функции

Строковые функции манипулируют значениями столбцов, которые обычно имеют символьный тип данных. Поддерживаемые в Transact-SQL строковые функции и их краткое описание приводятся в табл. 4.6.

Таблица 4.6. *Строковые функции*

Функция	**Описание**
ASCII(*символ*)	Преобразовывает указанный символ в соответствующее целое число кода ASCII. Пример: SELECT ASCII('A') = 65
CHAR(*целое число*)	Преобразовывает код ASCII в соответствующий символ. Пример: SELECT CHAR(65) = 'A'
CHARINDEX(*z1*,*z2*)	Возвращает начальную позицию вхождения подстроки *z1* в строку *z2*. Если строка *z2* не содержит подстроки *z1*, возвращается значение 0. Пример: SELECT CHARINDEX('bl', 'table') = 3
DIFFERENCE(*z1, z2*)	Возвращает целое число от 0 до 4, которое является разницей между значениями SOUNDEX двух строк *z1* и *z2*. Метод SOUNDEX возвращает число, которое характеризует звучание строки. С помощью этого метода можно определить подобно звучащие строки. Пример: SELECT DIFFERENCE('spelling', 'telling') = 2 (звучания несколько похожи, если же функция возвращает 0, то звучания не похожи)
LEFT(*z, length*)	Возвращает количество первых символов строки *z*, заданное параметром *length*
LEN(*z*)	Возвращает количество символов (не количество байт) строки *z*, указанной в аргументе, включая конечные пробелы
LOWER(*z1*)	Преобразовывает все прописные буквы строки *z1*, заданной в аргументе *z1*, в строчные. Входящие в строку строчные буквы и иные символы не затрагиваются. Пример: SELECT LOWER('BiG') = 'big'
LTRIM(*z*)	Удаляет начальные пробелы в строке *z*. Пример: SELECT LTRIM(' String') = 'String'
NCHAR(*i*)	Возвращает символ в кодировке Unicode, заданный целочисленным кодом, как определено в стандарте Unicode
QUOTENAME(*char_string*)	Возвращает строку в кодировке Unicode с добавленными ограничителями, чтобы преобразовать строку ввода в действительный идентификатор с ограничителями
PATINDEX(*%p%, expr*)	Возвращает начальную позицию первого вхождения шаблона *p* в заданное выражение *expr*, или ноль, если данный шаблон не обнаружен. Примеры (второй запрос возвращает все имена из столбца custormers): SELECT PATINDEX('%gs%', 'longstring') = 4 SELECT RIGHT(ContactName, LEN(ContactName)-PATINDEX('% %', ContactName)) AS First_ name FROM Customers

Таблица 4.6 *(продолжение)*

Функция	Описание
REPLACE(*str1*,*str2*,*str3*)	Заменяет все вхождения подстроки *str2* в строке *str1* подстрокой *str3*. Пример: SELECT REPLACE('shave', 's', 'be') = behave
REPLICATE(*z*,*i*)	Повторяет *i* раз строку *z*. Пример: SELECT REPLICATE('a',10) = 'aaaaaaaaaa'
REVERSE(*z*)	Выводит строку *z* в обратном порядке. Пример: SELECT REVERSE('calculate') = 'etaluclac'
RIGHT(*z*,*length*)	Возвращает последние *length*-символов строки *z*. Пример: SELECT RIGHT('Notebook',4) = 'book'
RTRIM(*z*)	Удаляет конечные пробелы в строке *z*. Пример: SELECT RTRIM('Notebook ') = 'Notebook'
SOUNDEX(*a*)	Возвращает четырехсимвольный код SOUNDEX, используемый для определения похожести двух строк. Пример: SELECT SOUNDEX('spelling') = S145
SPACE(*length*)	Возвращает строку пробелов длиной, указанной в параметре *length*. Пример: SELECT SPACE(4) = ' ' (Кавычки не являются частью возвращенной строки.)
STR(*f*,[*len*[,*d*]])	Преобразовывает заданное выражение с плавающей точкой *f* в строку, где *len* — длина строки, включая десятичную точку, знак, цифры и пробелы (по умолчанию равно 10), а *d* — число разрядов дробной части, которые нужно возвратить. Пример: SELECT STR(3.45678,4,2) ='3.46'
STUFF(*z1*,*a*,*length*,*z2*)	Удаляет из строки *z1* *length*-символов, начиная с позиции *a*, и вставляет на их место строку *z2*. Примеры: SELECT STUFF('Notebook',5,0,' in a') = 'Note in a book SELECT STUFF('Notebook',1,4, 'Hand') = 'Handbook'
SUBSTRING(*z*,*a*,*length*)	Извлекает из строки *z*, начиная с позиции a, подстроку длиной *length*. Пример: SELECT SUBSTRING('wardrobe',1,4) = 'ward'

Таблица 4.6 *(окончание)*

Функция	Описание
UNICODE(*z*)	Возвращает код Unicode первого символа строки *z*
UPPER(*z*)	Преобразовывает все строчные буквы строки *z* в прописные. Прописные буквы и цифры не затрагиваются. Пример: SELECT UPPER('loWer') = 'LOWER'

Системные функции

Системные функции языка Transact-SQL предоставляют обширную информацию об объектах базы данных. Большинство системных функций использует внутренний числовой идентификатор (ID), который присваивается каждому объекту базы данных при его создании. Посредством этого идентификатора система может однозначно идентифицировать каждый объект базы данных. В табл. 4.7 приводятся некоторые из наиболее важных системных функций вместе с их кратким описанием. (Полный список всех системных функций см. в *электронной документации*.)

Таблица 4.7. *Системные функции*

Функция	Описание
CAST(*a* AS *type* [(*length*)]	Преобразовывает выражение *a* в указанный тип данных *type* (если это возможно). Аргумент *a* может быть любым действительным выражением. Пример: SELECT CAST(3000000000 AS BIGINT) = 3000000000
COALESCE(*a1*,*a2*,...)	Возвращает первое значение выражения из списка выражений *a1*, *a2*,..., которое не является значением NULL
COL_LENGTH(*obj*,*col*)	Возвращает длину столбца *col* объекта базы данных (таблицы или представления) *obj*. Пример: SELECT COL_LENGTH('customers', 'custID') = 10
CONVERT(*type*[(*length*)],*a*)	Эквивалент функции CAST, но аргументы указываются по-иному. Может применяться с любым типом данных
CURRENT_TIMESTAMP	Возвращает текущие дату и время. Пример: SELECT CURRENT_TIMESTAMP = '2011-01-0117:22:55.670'
CURRENT_USER	Возвращает имя текущего пользователя
DATALENGTH(*z*)	Возвращает число байтов, которые занимает выражение *z*. Пример (возвращает длину каждого поля): SELECT DATALENGTH (ProductName) FROM products

Таблица 4.7 *(окончание)*

Функция	**Описание**
GETANSINULL('*dbname*')	Возвращает 1, если использование значений NULL в базе данных *dbname* отвечает требованиям стандарта ANSI SQL (см. также обсуждение значений NULL в конце этой главы). Пример: SELECT GETANSINULL('AdventureWorks') = 1
ISNULL(*expr*, *value*)	Возвращает значение выражения *expr*, если оно не равно нулю; в противном случае возвращается значение *value*
ISNUMERIC(*expression*)	Определяет, имеет ли выражение действительный числовой тип
NEWID()	Создает однозначный идентификационный номер ID, состоящий из 16-байтовой двоичной строки, предназначенной для хранения значений типа данных UNIQUEIDENTIFIER
NEWSEQUENTIALID()	Создает идентификатор GUID, больший, чем любой другой идентификатор GUID, созданный ранее этой функцией на указанном компьютере. (Эту функцию можно использовать только как значение по умолчанию для столбца.)
NULLIF(*expr1*,*expr2*)	Возвращает значение NULL, если значения выражений *expr1* и *expr2* одинаковые. Пример (возвращает значение NULL для проекта, для которого project_no='p1'): SELECT NULLIF(project_no, 'p1') FROM projects
SERVERPROPERTY(*propertyname*)	Возвращает информацию о свойствах сервера базы данных
SYSTEM_USER	Возвращает имя пользователя текущего пользователя. Пример: SELECT SYSTEM_USER = LTB13942\dusan
USER_ID([*user_name*])	Возвращает идентификатор пользователя user_name. Если пользователь не указан, то возвращается идентификатор текущего пользователя. Пример: SELECT USER_ID('guest') = 2
USER_NAME([*id*])	Возвращает имя пользователя с указанным идентификатором *id*. Если идентификатор не указан, то возвращается имя текущего пользователя. Пример: SELECT USER_NAME(1) = 'dbo'

Все строковые функции можно вкладывать друг в друга в любой порядке, например:

```
REVERSE(CURRENT_USER)
```

Функции метаданных

По большому счету, функции метаданных возвращают информацию об указанной базе данных и объектах базы данных. В табл. 4.8 приводятся некоторые из наиболее важных функций метаданных вместе с их кратким описанием. (Полный список всех функций метаданных см. в *электронной документации*.)

Таблица 4.8. *Функции метаданных*

Функция	Описание
COL_NAME(*tab_id*, *col_id*)	Возвращает имя столбца с указанным идентификатором *col_id* таблицы с идентификатором *tab_id*. Пример: SELECT COL_NAME(OBJECT_ID('employee'), 3) = 'emp_lname'
COLUMNPROPERTY(*id*, *col*, *property*)	Возвращает информацию об указанном столбце. Пример: SELECT COLUMNPROPERTY(object_id('project'), 'project_no', 'PRECISION') = 4
DATABASEPROPERTYEX (*database*, *property*)	Возвращает значение свойства *property* базы данных *database*. Пример (указывает, удовлетворяет ли база данных требованиям правил SQL-92 для разрешения значений NULL): SELECT DATABASEPROPERTYEX('sample', 'IsAnsiNullDefault') = 0
DB_ID([*db_name*])	Возвращает идентификатор базы данных *db_name*. Если имя базы данных не указано, то возвращается идентификатор текущей базы данных. Пример: SELECT DB_ID('AdventureWorks') = 6
DB_NAME([*db_id*])	Возвращает имя базы данных, имеющей идентификатор *db_id*. Если идентификатор не указан, то возвращается имя текущей базы данных. Пример: SELECT DB_NAME(6) = 'AdventureWorks'
INDEX_COL(*table*, *i*, *no*)	Возвращает имя индексированного столбца таблицы *table*. Столбец указывается идентификатором индекса *i* и позицией *no* столбца в этом индексе
INDEXPROPERTY(*obj_id*, *index_name*, *property*)	Возвращает свойства именованного индекса или статистики для указанного идентификационного номера таблицы, имя индекса или статистики, а также имя свойства
OBJECT_NAME(*obj_id*)	Возвращает имя объекта базы данных, имеющего идентификатор *obj_id*. Пример: SELECT OBJECT_NAME(453576654) = 'products'

Таблица 4.8 *(окончание)*

Функция	Описание
OBJECT_ID(*obj_name*)	Возвращает идентификатор объекта *obj_name* базы данных. Пример: SELECT OBJECT_ID('products') = 453576654
OBJECTPROPERTY(*obj_id*, *property*)	Возвращает информацию об объектах из текущей базы данных

Скалярные операторы

Скалярные операторы используются для работы со скалярными значениями. Язык Transact-SQL поддерживает числовые и логические операторы, а также конкатенацию.

Существуют унарные и бинарные арифметические операторы. Унарными операторами являются знаки + и –. К бинарным арифметическим операторам относятся операторы сложения (+), вычитания (–), умножения (*), деления (/) и деления по модулю (%).

Логические операторы обозначаются двумя разными способами, в зависимости от того, применяются они к битовым строкам или к другим типам данных. Операторы NOT, AND и OR применяются со всеми типами данных (за исключением данных типа BIT). Эти операторы подробно рассматриваются в *главе 6*.

Далее описаны побитовые (логические) операторы для обработки строк, а в примере 4.6 демонстрируется их использование.

- ~ — логическое отрицание — инверсия (т. е. оператор NOT);
- & — конъюнкция битовых строк (т. е. оператор AND);
- | — дизъюнкция битовых строк (т. е. оператор OR);
- ^ — исключающая дизъюнкция (т. е. оператор XOR или "Исключающее OR").

Пример 4.6. Применение побитовых операторов для манипулирования битовыми строками

```
-(1001001) = (0110110)
(11001001) | (10101101) = (11101101)
(11001001)&(10101101) = (10001001)
(11001001) ^ (10101101) = (01100100)
```

Оператор конкатенации + применяется для соединения символьных или битовых строк.

Глобальные переменные

Глобальные переменные — это специальные системные переменные, которые можно использовать, как будто бы они были скалярными константами. Язык Transact-SQL поддерживает большое число глобальных переменных, именам которых предшествует префикс `@@`. В табл. 4.9 приводится список некоторых наиболее важных глобальных переменных и их краткое описание. (Полный список глобальных переменных см. в *электронной документации*.)

Таблица 4.9. *Глобальные переменные*

Переменная	Описание
`@@CONNECTIONS`	Возвращает число попыток входа в систему со времени запуска системы
`@@CPU_BUSY`	Возвращает общее время занятости центрального процессора (в миллисекундах), прошедшее с момента старта системы
`@@ERROR`	Возвращает информацию о возвращенном значении последней исполненной инструкции Transact-SQL
`@@IDENTITY`	Возвращает последнее значение, добавленное в столбец, имеющий свойство `IDENTITY` (см. *главу 6*)
`@@LANGID`	Возвращает идентификатор языка, используемого в настоящий момент базой данных
`@@LANGUAGE`	Возвращает названия языка, используемого в настоящий момент базой данных
`@@MAX_CONNECTIONS`	Возвращает максимальное число фактических соединений с системой
`@@PROCID`	Возвращает идентификатор хранимой процедуры, исполняемой в настоящий момент
`@@ROWCOUNT`	Возвращает количество строк таблицы, которые были затронуты последней инструкцией Transact-SQL, исполненной системой
`@@SERVERNAME`	Возвращает информацию о локальном сервере базы данных. Эта информация содержит, среди прочего, имя сервера и имя экземпляра
`@@SPID`	Возвращает идентификатор серверного процесса
`@@VERSION`	Возвращает текущую версию программного обеспечения системы баз данных

Значение *NULL*

Значение `NULL` — это специальное значение, которое можно присвоить ячейке таблицы. Это значение обычно применяется, когда информация в ячейке неизвестна или неприменима. Например, если неизвестен номер домашнего телефона служащего компании, рекомендуется присвоить соответствующей ячейке столбца `home_telephone` значение `NULL`.

Если значение любого операнда любого арифметического выражения равно `NULL`, значение результата вычисления этого выражения также будет `NULL`. Поэтому в

унарных арифметических операциях, если значение выражения A равно NULL, тогда как +A, так и -A возвращает NULL. В бинарных выражениях, если значение одного или обоих операндов A и B равно NULL, тогда результат операции сложения (A+B), вычитания (A-B), умножения (A*B), деления (A/B) и деления по модулю (A%B) также будет NULL. (Операнды A и B должны быть числовыми выражениями.)

Если выражение содержит операцию сравнения и значение одного или обоих операндов этой операции равно NULL, результат этой операции также будет NULL. Следовательно, в таком случае все выражения: A=B, A<>B, A<B и A>B — возвратят значение NULL.

Для логических операций AND, OR и NOT поведение значений NULL описывается в следующих *таблицах истинности*, где И означает истина (true), Н — неизвестно (NULL), а Л — ложь (false). В этих таблицах, операнды логических операторов AND и OR представлены как заглавия столбцов и строк, а результат операции для конкретной пары операндов находится в ячейке на пресечении соответствующего столбца и строки. Для оператора NOT операнды представлены в столбце "Операнд", а результат в ячейке справа.

Операция AND			
	И	**Н**	**Л**
И	И	Н	Л
Н	Н	Н	Л
Л	Л	Л	Л

Операция OR			
	И	**Н**	**Л**
И	И	И	И
Н	И	Н	Н
Л	И	Н	Л

Операция NOT	
Операнд	
И	Л
Н	Н
Л	И

Перед вычислением агрегатных функций AVG, SUM, MAX, MIN и COUNT (за исключением функции COUNT(*)) в их аргументах удаляются все значения NULL. Если все ячейки столбца содержат только значения NULL, функция возвращает NULL. Агрегатная функция COUNT(*) обрабатывает все значения NULL таким же образом, как и значения, не являющиеся NULL. Если столбец содержит только значения NULL, функция COUNT(DISTINCT *column_name*) возвращает 0.

Значение NULL должно отличаться от всех других значений. Для числовых типов данных значение 0 и значение NULL не являются одинаковыми. То же самое относится и к пустой строке и значению NULL для символьных типов данных.

Значения NULL можно сохранять в столбце таблицы только в том случае, если это явно разрешено в определении данного столбца. С другой стороны, значения NULL не разрешаются для столбца, если в его определении явно указано NOT NULL. Если для столбца с типом данных (за исключением типа TIMESTAMP) не указано явно NULL или NOT NULL, то присваиваются следующие значения:

- NULL, если значение параметра ANSI_NULL_DFLT_ON инструкции SET равно ON.
- NOT NULL, если значение параметра ANSI_NULL_DFLT_OFF инструкции SET равно ON.

Если инструкцию SET не активировать, то столбец по умолчанию будет содержать значение NOT NULL. (Для столбцов типа TIMESTAMP значения NULL не разрешаются.)

Резюме

Основные возможности языка Transact-SQL предоставляются типами данных, предикатами и функциями. Типы данных отвечают требованиям к типам данных стандарта ANSI SQL92. Язык Transact-SQL поддерживает большое число различных полезных системных функций.

В следующей главе мы познакомимся с инструкциями языка Transact-SQL, в частности с подмножеством инструкций языка описания данных SQL. Это подмножество Transact-SQL содержит все инструкции, необходимые для создания, изменения и удаления объектов баз данных.

Упражнения

Упражнение 4.1

Какая разница между числовыми типами данных INT, SMALLINT и TINYINT?

Упражнение 4.2

Какая разница между типами данных CHAR и VARCHAR? Когда следует использовать первый, а не второй, и наоборот?

Упражнение 4.3

Как настроить столбец типа данных DATE для ввода значений в формате 'гггг/мм/дд'?

В следующих двух упражнениях используйте инструкцию SELECT в окне редактора запросов средства Management Studio для вывода результатов всех системных функций и глобальных переменных. (Например, инструкция SELECT host_id() выводит идентификационный номер текущего хоста.)

Упражнение 4.4

Используя системные функции, узнайте идентификационный номер базы данных test (см. упражнение 2.1).

Упражнение 4.5

Используя системные переменные, узнайте текущую версию программного обеспечения системы базы данных и используемый в программном обеспечении язык.

Упражнение 4.6

Используя битовые операторы &, | и ^, выполните следующие операции над битовыми строками:

```
(11100101) & (01010111)
(10011011) | (11001001)
(10110111) ^ (10110001)
```

Упражнение 4.7

Какими будут результаты следующих выражений? (Выражение `A` — числовое, а `B` — логическое.)

```
A + NULL
NULL = NULL
B OR NULL
B AND NULL
```

Упражнение 4.8

В каких случаях можно использовать как одинарные, так и двойные кавычки для определения строковых и временных констант?

Упражнение 4.9

Что такое идентификатор с ограничителями и когда требуется использовать идентификаторы этого типа?

Глава 5

Язык описания данных

- **Создание объектов баз данных**
- **Модифицирование объектов баз данных**
- **Удаление объектов баз данных**

В этой главе рассматриваются все инструкции Transact-SQL, связанные с языком описания данных DDL (Data Definition Language). Инструкции языка DDL разбиты на три группы, которые рассматриваются последовательно. В первую группу входят инструкции для создания объектов, вторая содержит инструкции для модифицирования структуры объектов, а третья состоит из инструкций для удаления объектов.

Создание объектов баз данных

В организации базы данных задействуется большое число различных объектов. Все объекты базы данных являются либо физическими, либо логическими. *Физические объекты* связаны с организацией данных на физических устройствах (дисках). Физическими объектами компонента Database Engine являются файлы и файловые группы. *Логические объекты* являются пользовательскими представлениями базы данных. В качестве примера логических объектов можно назвать таблицы, столбцы и представления (виртуальные таблицы).

Объектом базы данных, который требуется создать в первую очередь, является сама база данных. Компонент Database Engine управляет как системными, так и пользовательскими базами данных. Пользовательские базы данных могут создаваться авторизованными пользователями, тогда как системные базы данных создаются при установке системы базы данных.

В этой главе рассматривается создание, изменение и удаление пользовательских баз данных, а подробное описание всех системных баз данных приводится в *главе 15*.

Создание базы данных

Для создания базы данных используется два основных метода. В первом методе задействуется обозреватель объектов среды SQL Server Management Studio (см. *главу 3*), а во втором применяется инструкция языка Transact-SQL `CREATE DATABASE`. Далее приводится общая форма этой инструкции, а затем подробно рассматриваются ее составляющие:

```
CREATE DATABASE db_name
      [ON [PRIMARY] {file_spec1},.]
      [LOG ON {file_spec2},.]
            [COLLATE collation_name]
   [FOR {ATTACH ATTACH_REBUILD_LOG}]
```

ПРИМЕЧАНИЕ

Синтаксис инструкций языка Transact-SQL представляется соответственно соглашениям, рассмотренным в *разд. "Соглашения о синтаксисе"* в *главе 1*. По этим соглашениям необязательные элементы инструкций приводятся в квадратных скобках `[]`. В фигурных скобках `{}` с последующим троеточием представлены элементы, которые можно повторять любое количество раз.

Параметр `db_name` — это *имя базы данных*. Имя базы данных может содержать максимум 128 символов. (Правила для идентификаторов, рассмотренные в *главе 4*, применяются к именам баз данных.) Одна система может управлять до 32 767 базами данных.

Все базы данных хранятся в файлах, которые могут быть указаны явно администратором или предоставлены неявно системой. Если инструкция `CREATE DATABASE` содержит параметр `ON`, все файлы базы данных указываются явно.

ПРИМЕЧАНИЕ

Компонент Database Engine хранит файлы данных на диске. Каждый файл содержит данные одной базы данных. Эти файлы можно организовать в файловые группы. Файловые группы предоставляют возможность распределять данные по разным приводам дисков и выполнять резервное копирование и восстановление частей базы данных. Это полезная функциональность для очень больших баз данных.

Параметр `file_spec1` представляет спецификацию файла и сам может содержать дополнительные опции, такие как логическое имя файла, физическое имя и размер. Параметр `PRIMARY` указывает первый (и наиболее важный) файл, который содержит системные таблицы и другую важную внутреннюю информацию о базе данных. Если параметр `PRIMARY` отсутствует, то в качестве первичного файла используется первый файл, указанный в спецификации.

Учетная запись компонента Database Engine, применяемая для создания базы данных, называется *владельцем базы данных*. База данных может иметь только одного владельца, который всегда соответствует учетной записи. Учетная запись, принад-

лежащая владельцу базы данных, имеет специальное имя dbo. Это имя всегда используется в отношении базы данных, которой владеет пользователь.

Опция LOG ON параметра dbo определяет один или более файлов в качестве физического хранилища журнала транзакций базы данных. Если опция LOG ON отсутствует, то журнал транзакций базы данных все равно будет создан, поскольку каждая база данных должна иметь, по крайней мере, один журнал транзакций. (Компонент Database Engine ведет учет всем изменениям, которые он выполняет с базой данных. Система сохраняет все эти записи, в особенности значения до и после транзакции, в одном или более файлов, которые называются журналами транзакций. Для каждой базы данных системы ведется ее собственный журнал транзакций. Журналы транзакций подробно рассматриваются в *главе 13.*)

В опции COLLATE указывается порядок сортировки по умолчанию для базы данных. Если опция COLLATE не указана, базе данных присваивается порядок сортировки по умолчанию базы данных model, совершенно такой же, как и порядок сортировки по умолчанию системы баз данных.

В опции FOR ATTACH указывается, что база данных создается за счет подключения существующего набора файлов. При использовании этой опции требуется явно указать первый первичный файл. В опции FOR ATTACH_REBUILD_LOG указывается, что база данных создается методом присоединения существующего набора файлов операционной системы. (Предмет присоединения и отсоединения базы данных рассматривается далее в этой главе.)

Компонент Database Engine создает новую базу данных по шаблону образцовой базы данных model. Свойства базы данных model можно настраивать для удовлетворения персональных концепций системного администратора.

ПРИМЕЧАНИЕ

Если определенный объект базы данных должен присутствовать в каждой пользовательской базе данных, то этот объект следует сначала создать в базе данных model.

В примере 5.1 показан код для создания простой базы данных, без указания дополнительных подробностей. Чтобы исполнить этот код, введите его в редактор запросов среды Management Studio и нажмите клавишу <F5>.

Пример 5.1. Код для создания простой базы данных

```
USE  master;
CREATE DATABASE sample;
```

Код, приведенный в примере 5.1, создает базу данных, которая называется sample. Такая сокращенная форма инструкции CREATE DATABASE возможна благодаря тому, что почти все ее параметры имеют значения по умолчанию. По умолчанию система создает два файла. Файл данных имеет логическое имя sample и исходный размер

2 Мбайта. А файл журнала транзакций имеет логическое имя sample_log и исходный размер 1 Мбайт. (Значения размеров обоих файлов, а также другие свойства новой базы данных зависят от соответствующих спецификаций базы данных model.)

В примере 5.2 показано создание базы данных с явным указанием файлов базы данных и журнала транзакций.

Пример 5.2. Создание базы данных с явным указанием файлов

```
USE  master;
CREATE DATABASE projects
   ON (NAME=projects_dat,
      FILENAME = 'C:\projects.mdf',
      SIZE = 10,
      MAXSIZE = 100,
      FILEGROWTH = 5)
LOG ON
   (NAME=projects_log,
      FILENAME = 'C:\projects.ldf',
      SIZE = 40,
      MAXSIZE = 100,
      FILEGROWTH = 10);
```

Созданная в листинге 5.2 база данных называется projects. Поскольку опция PRIMARY не указана, то первичным файлом предполагается первый файл. Этот файл имеет логическое имя projects_dat и он сохраняется в дисковом файле projects.mdf. Исходный размер этого файла 10 Мбайт. При необходимости, система выделяет этому файлу дополнительное дисковое пространство в приращениях по 5 Мбайт. Если не указать опцию MAXSIZE или если этой опции присвоено значение UNLIMITED, то максимальный размер файла может увеличиваться и будет ограничиваться только размером всего дискового пространства. (Единицу размера файла можно указывать с помощью суффиксов KB, TB и MB, означающих килобайты, терабайты и мегабайты соответственно. По умолчанию используется единица размера MB, т. е. мегабайты.)

Кроме файла данных создается файл журнала транзакций, который имеет логическое имя projects_log и физическое имя projects.ldf. Все опции спецификации файла журнала транзакций имеют такие же имена и значения, как и соответствующие опции для спецификации файла данных.

В языке Transact-SQL можно указать конкретный контекст базы данных (т. е. какую базу данных использовать в качестве текущей) с помощью инструкции USE.

(Альтернативный способ — выбрать имя требуемой базы данных в раскрывающемся списке **Database** (Базы данных) в панели инструментов среды SQL Server Management Studio.)

Системный администратор может назначить пользователю текущую базу данных по умолчанию с помощью инструкции CREATE LOGIN или инструкции ALTER LOGIN (см. также *главу 12*). В таком случае пользователям не нужно выполнять инструкцию USE, если только они не хотят использовать другую базу данных.

Создание моментального снимка базы данных

Кроме создания новой базы данных, инструкцию CREATE DATABASE можно применить для получения моментального снимка существующей базы данных (база данных-источник). Моментальный снимок базы данных является согласованной с точки зрения завершенных транзакций копией исходной базы данных на момент создания моментального снимка. Далее показан синтаксис инструкции для создания моментального снимка базы данных:

```
CREATE DATABASE database_snapshot_name
   ON (NAME = logical_file_name,
      FILENAME = 'os_file_name') [,...n ]
      AS SNAPSHOT OF source_database_name
```

Таким образом, чтобы создать моментальный снимок базы данных, в инструкцию CREATE DATABASE нужно вставить предложение AS SNAPSHOT OF. В примере 5.3 иллюстрируется создание моментального снимка базы данных Adventure Works и сохранения его в папке C:\temp. (Прежде чем выполнять этот пример, нужно создать данный каталог. Кроме этого, нужно загрузить и установить базу данных Adventure Works, если вы еще этого не сделали. База данных Adventure Works — это образцовая база данных для SQL Server. Информацию по ее загрузке смотрите в конце *разд. "Работа с образцовыми базами данных" во Введении*.)

Пример 5.3. Создание моментального снимка базы данных Adventure Works

```
USE master;
CREATE DATABASE AdventurWorks_snapshot
   ON (NAME = 'AdventureWorks_Data' ,
      FILENAME = 'C:\temp\snapshot_DB.mdf')
      AS SNAPSHOT OF AdventureWorks;
```

Моментальный снимок существующей базы данных — это доступная только для чтения копия базы данных-источника, которая отражает состояние этой базы данных на момент копирования. (Таким образом, можно создавать множественные моментальные снимки существующей базы данных.) Файл моментального снимка (в примере 5.3 это файл C:\temp\snapshot_DB.mdf) содержит только измененные данные базы данных-источника. Поэтому в коде для создания моментального снимка необходимо указывать логическое имя каждого файла данных базы данных-источника, а также соответствующие физические имена (см. пример 5.4).

Поскольку моментальный снимок содержит только измененные данные, то для каждого снимка требуется лишь небольшая доля дискового пространства, требуемого для соответствующей базы данных-источника.

ПРИМЕЧАНИЕ

Моментальные снимки баз данных можно создавать только на дисках с файловой системой *NTFS* (New Technology File System — файловая система новой технологии), т. к. только эта файловая система поддерживает технологию разреженных файлов, применяемую для хранения моментальных снимков.

Моментальные снимки баз данных обычно применяются в качестве механизма предохранения данных от искажения.

Присоединение и отсоединение баз данных

Все данные базы данных можно отсоединить, а потом снова присоединить к этому же или другому серверу базы данных. Эта функциональность используется при перемещении базы данных.

Для отсоединения базы данных от сервера баз используется системная процедура `sp_detach_db`. (Отсоединяемая база данных должна находиться в однопользовательском режиме.)

Для присоединения базы данных используется инструкция `CREATE DATABASE` с предложением `FOR ATTACH`. Для присоединяемой базы данных должны быть доступными все требуемые файлы. Если какой-либо файл данных имеет путь, отличающийся от исходного пути, то для этого файла необходимо указать текущий путь.

Инструкция *CREATE TABLE*: базовая форма

Инструкция `CREATE TABLE` создает новую таблицу базы данных со всеми соответствующими столбцами требуемого типа данных. Далее приводится базовая форма инструкции `CREATE TABLE`:

```
CREATE TABLE table_name
   (col_name1 type1 [NOT NULL | NULL)
   [{, col_name2 type2 [NOT NULL | NULL]} ...])
```

ПРИМЕЧАНИЕ

Кроме основных таблиц, существуют также некоторые специальные виды таблиц, такие как временные таблицы и представления (см. *главу 6* и *11* соответственно).

Параметр *table_name* — имя создаваемой базовой таблицы. Максимальное количество таблиц, которое может содержать одна база данных, ограничивается количеством объектов базы данных, число которых не может быть более 2 миллиардов, включая таблицы, представления, хранимые процедуры, триггеры и ограничения. В параметрах *col_name1*, *col_name2*, ... указываются имена столбцов таблицы, а в параметрах *type1*, *type2*, ... — типы данных соответствующих столбцов (см. *главу 4*).

ПРИМЕЧАНИЕ

Имя объекта базы данных может обычно состоять из четырех частей, в форме: [*server_name*.[*db_name*.[*schema_name*.]]]*object_name*

Здесь *object_name* — это имя объекта базы данных, *schema_name* — имя схемы, к которой принадлежит объект, а *server_name* и *db_name* — имена сервера и базы данных, к которым принадлежит объект. Имена таблиц, сгруппированные с именем схемы, должны быть однозначными в рамках базы данных. Подобным образом имена столбцов должны быть однозначными в рамках таблицы.

Рассмотрим теперь ограничение, связанное с присутствием или отсутствием значений NULL в столбце. Если для столбца не указано, что значения NULL разрешены (NOT NULL), то данный столбец не может содержать значения NULL, и при попытке вставить такое значение система возвратит сообщение об ошибке.

Как уже упоминалось, объект базы данных (в данном случае таблица) всегда создается в схеме базы данных. Пользователь может создавать таблицы только в такой схеме, для которой у него есть полномочия на выполнение инструкции ALTER. Любой пользователь с ролью sysadmin, db_ddladmin или db_owner может создавать таблицы в любой схеме. (Полномочия для выполнения инструкции ALTER, а также роли базы данных и сервера подробно рассматриваются в *главе 12*.)

Создатель таблицы не обязательно должен быть ее владельцем. Это означает, что один пользователь может создавать таблицы, которые принадлежат другим пользователям. Подобным образом таблица, создаваемая с помощью инструкции CREATE TABLE, не обязательно должна принадлежать к текущей базе данных, если в префиксе имени таблицы указать другую (существующую) базу данных и имя схемы.

Схема, к которой принадлежит таблица, может иметь два возможных имени по умолчанию. Если таблица указывается без явного имени схемы, то система выполняет поиск имени таблицы в соответствующей схеме по умолчанию. Если имя объекта найти в схеме по умолчанию не удается, то система выполняет поиск в схеме dbo.

ПРИМЕЧАНИЕ

Имена таблиц всегда следует указывать вместе с именем соответствующей схемы. Это позволит избежать возможных неопределенностей.

Временные таблицы — это специальный тип базовой таблицы. Она сохраняются в базе данных tempdb и автоматически удаляются в конце сессии. В *главе 6* рассматриваются свойства временных таблиц и приводятся соответствующие им примеры.

В примере 5.4 показано создание всех таблиц базы данных sample. (База данных sample должна быть установлена в качестве текущей базы данных.)

Пример 5.4. Создание всех таблиц базы данных sample

```
USE sample;
CREATE TABLE employee  (emp_no INTEGER NOT NULL,
                        emp_fname CHAR(20) NOT NULL,
```

```
                        emp_lname CHAR(20) NOT NULL,
                        dept_no CHAR(4) NULL);
CREATE TABLE department(dept_no CHAR(4) NOT NULL,
                        dept_name CHAR(25) NOT NULL,
                        location CHAR(30) NULL);
CREATE TABLE project   (project_no CHAR(4) NOT NULL,
                        project_name CHAR(15) NOT NULL,
                        budget FLOAT NULL);
CREATE TABLE works_on  (emp_no INTEGER NOT NULL,
                        project_no CHAR(4) NOT NULL,
                        job CHAR (15) NULL,
                        enter_date DATE NULL);
```

Кроме типа данных и свойства содержать значения NULL, в спецификации столбца можно указать следующие параметры:

- предложение DEFAULT;
- свойство IDENTITY.

Предложение DEFAULT в спецификации столбца указывает значение столбца по умолчанию, т. е. когда в таблицу вставляется новая строка, ячейка этого столбца будет содержать указанное значение, которое останется в ячейке, если в нее не будет введено другое значение. В качестве значения по умолчанию можно использовать константу, например одну из системных функций, таких как, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_TIMESTAMP и NULL.

Столбец идентификаторов, создаваемый указанием свойства IDENTITY, может иметь только целочисленные значения, которые системой присваиваются обычно неявно. Каждое следующее значение, вставляемое в такой столбец, вычисляется, увеличивая последнее, вставленное в этот столбец, значение. Поэтому определение столбца со свойством IDENTITY содержит (явно или неявно) начальное значение и шаг инкремента. Это свойство подробно рассматривается в *главе 6* (см. пример 6.42).

В заключение этого раздела, в примере 5.5, показано создание таблицы, содержащей столбец типа SQL_VARIANT.

Пример 5.5. Создание таблицы, содержащей столбец типа SQL_VARIANT

```
USE sample;
CREATE TABLE Item_Attributes (
             item_id INT NOT NULL,
             attribute NVARCHAR(30) NOT NULL,
             value SQL_VARIANT NOT NULL,
             PRIMARY KEY (item_id, attribute) )
```

В примере 5.5 создается таблица, содержащая столбец value, который имеет тип SQL_VARIANT. Как рассматривалось в *главе 4*, тип данных SQL_VARIANT можно использовать для хранения значений разных типов одновременно, таких как числовые

значения, строки и даты. Обратите внимание на то, что в примере 5.5 столбцу присваивается тип данных `SQL_VARIANT` по той причине, что значения разных атрибутов могут быть разных типов данных. Например, для атрибута размера тип данных значения `attribute` будет целочисленным, а для атрибута имени — строковым.

Инструкция *CREATE TABLE* и ограничения декларативной целостности

Одной из самых важных особенностей, которую должна предоставлять СУБД, является способ обеспечения целостности данных. Ограничения, которые используются для проверки данных при их модификации или вставке, называются *ограничениями для обеспечения целостности* (integrity constraints). Обеспечение целостности данных может осуществляться пользователем в прикладной программе или же системой управления базами данных. Наиболее важными преимуществами предоставления ограничений целостности системой управления базами данных являются следующие:

- повышается надежность данных;
- сокращается время на программирование;
- упрощается техническое обслуживание.

Определение ограничений для обеспечения целостности посредством СУБД повышает надежность данных, поскольку устраняется возможность, что программист прикладного приложения может забыть реализовать их. Если ограничения целостности предоставляются прикладными программами, то *все* приложения, затрагиваемые этими ограничениями, должны содержать соответствующий код. Если код отсутствует хоть в одном приложении, то целостность данных будет поставлена под сомнение.

Если ограничения для обеспечения целостности не предоставляются системой управления базами данных, то их необходимо определить в каждой программе приложения, которая использует данные, включенные в это ограничение. В противоположность этому, если ограничения для обеспечения целостности предоставляются системой управления базами данных, то их требуется определить только один раз. Кроме этого, код для ограничений, предоставляемых приложениями, обычно более сложный, чем в случае таких же ограничений, предоставляемых СУБД.

Если ограничения для обеспечения целостности предоставляются СУБД, то в случае изменений ограничений, соответствующие изменения в коде необходимо реализовать только один раз — в системе управления базами данных. А если ограничения предоставляются приложениями, то модификацию для отражения изменений в ограничениях необходимо выполнить в каждом из этих приложений.

Системами управления базами данных предоставляются два типа ограничений для обеспечения целостности:

- декларативные ограничения для обеспечения целостности;
- процедурные ограничения для обеспечения целостности, реализуемые посредством триггеров (информацию о триггерах см. в *главе 13*).

Декларативные ограничения определяются с помощью инструкций языка DDL `CREATE TABLE` и `ALTER TABLE`. Эти ограничения могут быть уровня столбцов или уровня таблицы. Ограничения уровня столбцов определяются наряду с типом данных и другими свойствами столбца в объявлении столбца, тогда как ограничения уровня таблицы всегда определяются в конце инструкции `CREATE TABLE` или `ALTER TABLE` после определения всех столбцов.

ПРИМЕЧАНИЕ

Между ограничениями уровня столбцов и ограничениями уровня таблицы есть лишь одно различие: ограничения уровня столбцов можно применять только к одному столбцу, в то время как ограничения уровня таблицы могут охватывать больше, чем один столбец таблицы.

Каждому декларативному ограничению присваивается имя. Это имя может быть присвоено явно посредством использования опции `CONSTRAINT` в инструкции `CREATE TABLE` или `ALTER TABLE`. Если опция `CONSTRAINT` не указывается, то имя ограничению присваивается неявно компонентом Database Engine.

ПРИМЕЧАНИЕ

Настоятельно рекомендуется использовать явные имена ограничений, поскольку это может значительно улучшить поиск этих ограничений.

Декларативные ограничения можно сгруппировать в следующие категории:

- предложение `DEFAULT`;
- предложение `UNIQUE`;
- предложение `PRIMARY KEY`;
- предложение `CHECK`;
- ссылочная целостность и предложение `FOREIGN KEY`.

Использование предложения `DEFAULT` для определения ограничения по умолчанию было показано в этой главе ранее (см. также пример 5.6). Все другие ограничения рассматриваются в последующих разделах.

Предложение *UNIQUE*

Иногда несколько столбцов или группа столбцов таблицы имеет уникальные значения, что позволяет использовать их в качестве первичного ключа. Столбцы или группы столбцов, которые можно использовать в качестве первичного ключа, называются *потенциальными ключами* (candidate key). Каждый потенциальный ключ определяется, используя предложение `UNIQUE` в инструкции `CREATE TABLE` или `ALTER TABLE`. Синтаксис предложения `UNIQUE` следующий:

```
[CONSTRAINT c_name]
     UNIQUE [CLUSTERED NONCLUSTERED1 ({col_name1},...)
```

Опция CONSTRAINT в предложении UNIQUE присваивает явное имя потенциальному ключу. Опция CLUSTERED или NONCLUSTERED связана с тем обстоятельством, что компонент Database Engine создает индекс для каждого потенциального ключа таблицы. Этот индекс может быть кластеризованным, когда физический порядок строк определяется посредством индексированного порядка значений столбца. Если порядок строк не указывается, индекс является некластеризованным (см. *главу 10*). По умолчанию применяется опция NONCLUSTERED. Параметр *col_name1* обозначает имя столбца, который создает потенциальный ключ. (Потенциальный ключ может иметь до 16 столбцов.)

Применение предложения UNIQUE показано в примере 5.6. (Прежде чем выполнять этот пример, в базе данных sample нужно удалить таблицу projects, используя для этого инструкцию DROP TABLE projects.)

Пример 5.6. Применение предложения UNIQUE

```
USE sample;
CREATE TABLE projects (project_no CHAR(4) DEFAULT 'p1',
                       project_name CHAR(15) NOT NULL,
                       budget FLOAT NULL
                       CONSTRAINT unique_no UNIQUE (project_no));
```

Каждое значение столбца project_no таблицы projects является уникальным, включая значение NULL. (Точно так же, как и для любого другого значения с ограничением UNIQUE, если значения NULL разрешены для соответствующего столбца, этот столбец может содержать не более одной строки со значением NULL.) Попытка вставить в столбец project_no уже имеющееся в нем значение будет неуспешной, т. к. система не примет его. Явное имя ограничения, определяемого в примере 5.6, — unique_no.

Предложение *PRIMARY KEY*

Первичным ключом таблицы является столбец или группа столбцов, значения которого разные в каждой строке. Каждый первичный ключ определяется, используя предложение PRIMARY KEY в инструкции CREATE TABLE или ALTER TABLE. Синтаксис предложения PRIMARY KEY следующий:

```
[CONSTRAINT c_name]
   PRIMARY KEY [CLUSTERED | NONCLUSTERED] ({col_name1},...)
```

Все параметры предложения PRIMARY KEY имеют такие же значения, как и соответствующие одноименные параметры предложения UNIQUE. Но в отличие от столбца UNIQUE, столбец PRIMARY KEY не разрешает значений NULL и имеет значение по умолчанию CLUSTERED.

В примере 5.7 показано объявление первичного ключа для таблицы employee базы данных sample.

ПРИМЕЧАНИЕ

Прежде чем выполнять этот пример, в базе данных `sample` нужно удалить таблицу `employee`, используя для этого инструкцию `DROP TABLE employee`.

Пример 5.7. Определение первичного ключа

```
USE sample;
CREATE TABLE employee (emp_no INTEGER NOT NULL,
                       emp_fname CHAR(20) NOT NULL,
                       emp_lname CHAR(20) NOT NULL,
                       dept_no CHAR(4) NULL,
                       CONSTRAINT prim_empl PRIMARY KEY (emp_no));
```

В результате выполнения кода в примере 5.7 снова создается таблица `employee`, в которой определен первичный ключ. Первичный ключ таблицы определяется посредством декларативного ограничения для обеспечения целостности с именем `prim_empl`. Это ограничение для обеспечения целостности является ограничением уровня таблицы, поскольку оно указывается после определения всех столбцов таблицы `employee`.

Пример 5.8 эквивалентный примеру 5.7, за исключением того, что первичный ключ таблицы `employee` определяется как ограничение уровня столбца.

ПРИМЕЧАНИЕ

Опять же, прежде чем выполнять этот пример, в базе данных `sample` нужно удалить таблицу `employee`, используя для этого инструкцию `DROP TABLE employee`.

Пример 5.8. Определение ограничения уровня столбца

```
USE sample;
CREATE TABLE employee
       (emp_no INTEGER NOT NULL CONSTRAINT prim_empl PRIMARY KEY,
        emp_fname CHAR(20) NOT NULL,
        emp_lname CHAR(20) NOT NULL,
        dept_no CHAR(4) NULL);
```

В примере 5.8 предложение `PRIMARY KEY` принадлежит к объявлению соответствующего столбца, наряду с объявлением его типа данных и свойства содержать значения `NULL`. По этой причине это ограничение называется *ограничением на уровне столбца*.

Предложение *CHECK*

Проверочное ограничение (check constraint) определяет условия для вставляемых в столбец данных. Каждая вставляемая в таблицу строка или каждое значение, ко-

торым обновляется значение столбца, должно отвечать этим условиям. Проверочные ограничения устанавливаются посредством предложения CHECK, определяемого в инструкции CREATE TABLE или ALTER TABLE. Синтаксис предложения CHECK следующий:

```
[CONSTRAINT c_name]
     CHECK [NOT FOR REPLICATION] expression
```

Параметр *expression* должен иметь логическое значение (true или false) и может ссылаться на любые столбцы в текущей таблице (или только на текущий столбец, если определен как ограничение уровня столбца), но не на другие таблицы. Предложение CHECK не применяется принудительно при репликации данных, если присутствует параметр NOT FOR REPLICATION. (При репликации база данных, или ее часть, хранится в нескольких местах. С помощью репликации можно повысить уровень доступности данных. Репликация данных рассматривается в *главе 19*.)

В примере 5.9 показано применение предложения CHECK.

Пример 5.9. Применение предложения CHECK

```
USE sample;
CREATE TABLE customer
     (cust_no INTEGER NOT NULL,
      cust_group CHAR(3) NULL,
      CHECK (cust_group IN ('c1', ' c2 ', 'c10')));
```

Создаваемая в примере 5.9 таблица customer включает столбец cust_group, содержащий соответствующее проверочное ограничение. При вставке нового значения, отличающегося от значений в наборе ('c1', 'c2', 'c10), или при попытке изменения существующего значения на значение, отличающегося от этих значений, система управления базой данных возвращает сообщение об ошибке.

Предложение *FOREIGN KEY*

Внешним ключом (foreign key) называется столбец (или группа столбцов таблицы), содержащий значения, совпадающие со значениями первичного ключа в этой же или другой таблице. Внешний ключ определяется с помощью предложения FOREIGN KEY в комбинации с предложением REFERENCES. Синтаксис предложения FOREIGN KEY следующий:

```
[CONSTRAINT c_name]
   [[FOREIGN KEY] ({col_name1},.)]
   REFERENCES table_name ({col_name2},.)
      [ON DELETE (NO ACTION] CASCADE | SET NULL | SET DEFAULT}]
      [ON UPDATE (NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
```

Предложение FOREIGN KEY явно определяет все столбцы, входящие во внешний ключ. В предложении REFERENCES указывается имя таблицы, содержащей столбцы, создающие соответствующий первичный ключ. Количество столбцов и их тип дан-

ных в предложении FOREIGN KEY должны совпадать с количеством соответствующих столбцов и их типом данных в предложении REFERENCES (и, конечно же, они должны совпадать с количеством столбцов и типами данных в первичном ключе таблицы, на которую они ссылаются).

Таблица, содержащая внешний ключ, называется *ссылающейся (или дочерней) таблицей* (referencing table), а таблица, содержащая соответствующий первичный ключ, называется *ссылочной* (referenced table) или *родительской* (parent table) таблицей. В примере 5.10 показано объявление внешнего ключа для таблицы works_on базы данных sample.

ПРИМЕЧАНИЕ

Прежде чем выполнять этот пример, в базе данных sample нужно удалить таблицу works_on, используя для этого инструкцию DROP TABLE works_on.

Пример 5.10. Объявление внешнего ключа

```
USE sample;
CREATE TABLE works_on (emp_no INTEGER NOT NULL,
   project_no CHAR(4) NOT NULL,
   job CHAR (15) NULL,
   enter_date DATE NULL,
      CONSTRAINT prim_works PRIMARY KEY(emp_no, project_no),
      CONSTRAINT foreign_works FOREIGN KEY(emp_no)
         REFERENCES employee (emp_no));
```

Таблица works_on в примере 5.10 задается с двумя декларативными ограничениями для обеспечения целостности: prim_works и foreign_works. Оба ограничения являются уровня таблицы, где первое указывает первичный ключ, а второе — внешний ключ таблицы works_on. Кроме этого, ограничение foreign_works определяет таблицу employee, как ссылочную таблицу, а ее столбец emp_no, как соответствующий первичный ключ столбца с таким же именем в таблице works_on.

Предложение FOREIGN KEY можно пропустить, если внешний ключ определяется, как ограничение уровня таблицы, поскольку столбец, к которому применяется ограничение, является неявным "списком" столбцов внешнего ключа, и ключевого слова REFERENCES достаточно для указания того, какого типа является это ограничение. Таблица может содержать самое большее 63 ограничения FOREIGN KEY.

Определение внешних ключей в таблицах базы данных налагает определение другого важного ограничения для обеспечения целостности: ссылочной целостности, которая рассматривается следующей.

Ссылочная целостность

Ссылочная целостность (referential integrity) обеспечивает выполнение правил для вставок и обновлений таблиц, содержащих внешний ключ и соответствующее ог-

раничение первичного ключа. В примерах 5.7 и 5.10 задаются два таких ограничения: prim_empl и foreign_works. Предложение REFERENCES в примере 5.10 определяет таблицу employee в качестве ссылочной (родительской) таблицы.

Если для двух таблиц указана ссылочная целостность, модифицирование значений в первичном ключе и соответствующем внешнем ключе будет не всегда возможным. В последующих разделах рассматривается, когда это возможно, а когда нет.

Возможные проблемы со ссылочной целостностью

Модификация значений внешнего или первичного ключа может создавать проблемы в четырех случаях. Все эти случаи будут продемонстрированы с использованием базы данных sample. В первых двух случаях затрагиваются модификации ссылающейся таблицы, а в последних двух — родительской.

Случай 1

Вставка новой строки в таблицу works_on с номером сотрудника 11111.

Соответствующая инструкция Transact-SQL выглядит таким образом:

```
USE sample;
INSERT INTO works_on (emp_no,...)
   VALUES (11111,...);
```

При вставке новой строки в дочернюю таблицу works_on используется новый номер сотрудника emp_no, для которого нет совпадающего сотрудника (и номера) в родительской таблице employee. Если для обеих таблиц определена ссылочная целостность, как это сделано в примерах 5.7 и 5.10, то компонент Database Engine не допустит вставки новой строки с таким номером emp_no.

Случай 2

Изменение номера сотрудника 10102 во всех строка таблицы works_on на номер 11111.

Соответствующая инструкция Transact-SQL выглядит таким образом:

```
USE sample;
UPDATE works_on
   SET emp_no = 11111 WHERE emp_no = 10102;
```

В данном случае существующее значение внешнего ключа в ссылающейся таблице works_on заменяется новым значением, для которого нет совпадающего значения в родительской таблице employee. Если для обеих таблиц определена ссылочная целостность, как это сделано в примерах 5.7 и 5.10, то система управления базой данных не допустит модификацию строки с таким номером emp_no в таблице works_on.

Случай 3

Замена значения 10102 номера сотрудника emp_no на значение 22222 в таблице employee. Соответствующая инструкция Transact-SQL будет выглядеть таким образом:

```
USE sample;
UPDATE employee
   SET emp_no = 22222 WHERE emp_no = 10102;
```

В данном случае предпринимается попытка заменить существующее значение 10102 номера сотрудника emp_no значением 22222 только в родительской таблице employee, не меняя соответствующие значения emp_no в ссылающейся таблице works_on. Система не разрешает выполнения этой операции. Ссылочная целостность не допускает существования в ссылающейся таблице (таблице, для которой предложением FOREIGN KEY определен внешний ключ) таких значений, для которых в родительской таблице (таблице, для которой предложением PRIMARY KEY определен первичный ключ) не существует соответствующего значения. В противном случае такие строки в ссылающейся таблице были бы "сиротами". Если бы описанная выше модификация таблицы employee была разрешена, тогда строки в таблице works_on со значением emp_no равным 10102 были бы сиротами. Поэтому система и не разрешает выполнения такой модификации.

Случай 4

Удаление строки в таблице employee со значением emp_no равным 10102.

Этот случай похожий на *случай 3*. В случае выполнения этой операции, из таблицы employee была бы удалена строка со значением emp_no, для которого существуют совпадающие значения в ссылающейся (дочерней) таблице works_on. В примере 5.11 приводится код для определения всех ограничений первичного и внешнего ключей для таблиц базы данных sample. Для выполнения этого кода, если таблицы employee, department, project и works_on уже существуют, то их нужно удалить с помощью инструкции DROP TABLE *table_name* *(имя_таблицы)*.

Пример 5.11. Определение всех ограничений первичного и внешнего ключей для таблиц базы данных sample

```
USE sample;
CREATE TABLE department(dept_no CHAR(4) NOT NULL,
                        dept_name CHAR(25) NOT NULL,
                        location CHAR(30) NULL,
             CONSTRAINT prim_dept PRIMARY KEY (dept_no));
CREATE TABLE employee (emp_no INTEGER NOT NULL,
                       emp_fname CHAR(20) NOT NULL,
                       emp_lname CHAR(20) NOT NULL,
                       dept_no CHAR(4) NULL,
             CONSTRAINT prim_emp PRIMARY KEY (emp_no),
             CONSTRAINT foreign_emp FOREIGN KEY(dept_no) REFERENCES
                                          department(dept_no));
CREATE TABLE project (project_no CHAR(4) NOT NULL,
                      project_name CHAR(15) NOT NULL,
                      budget FLOAT NULL,
             CONSTRAINT prim_proj PRIMARY KEY (project_no));
```

```
CREATE TABLE works_on (emp_no INTEGER NOT NULL,
                       project_no CHAR(4) NOT NULL,
                       job CHAR (15) NULL,
                       enter_date DATE NULL,
            CONSTRAINT prim_works PRIMARY KEY(emp_no, project_no),
            CONSTRAINT foreign1_works FOREIGN KEY(emp_no) REFERENCES
                                          employee(emp_no),
            CONSTRAINT foreign2_works FOREIGN KEY(project_no)
                                  REFERENCES project(project_no));
```

Опции *ON DELETE* и *ON UPDATE*

Компонент Database Engine на попытку удаления и модифицирования первичного ключа может реагировать по-разному. Если попытаться обновить значения внешнего ключа, то все эти обновления будут несогласованы с соответствующим первичным ключом (см. *случай 1* и *2* в предыдущем разделе), база данных откажется выполнять эти обновления и выведет сообщение об ошибке, наподобие следующего:

```
Server: Msg 547, Level 16, State 1, Line 1 UPDATE statement conflicted with
COLUMN FOREIGN KEY constraint 'FKemployee'. The conflict occurred in database
'sample', table 'employee', column 'dept_no'. The statement has been
terminated.
```

(Сообщение 547, уровень 16, состояние 0, строка 1. Конфликт инструкции UPDATE с ограничением FOREIGN KEY "foreign1_works". Конфликт произошел в базе данных "sample", таблица "dbo.employee", столбец 'emp_no'. Выполнение данной инструкции было прервано.)

Но при попытке внести обновления в значения первичного ключа, вызывающие несогласованность в соответствующем внешнем ключе (см. *случай 3* и *4* в предыдущем разделе), система базы данных может реагировать достаточно гибко. В целом, существует четыре опции, определяющих то, как система базы данных может реагировать.

- `NO ACTION`. Модифицируются (обновляются или удаляются) только те значения в родительской таблице, для которых нет соответствующих значений во внешнем ключе дочерней (ссылающейся) таблицы.
- `CASCADE`. Разрешается модификация (обновление или удаление) любых значений в родительской таблице. При обновлении значения первичного ключа в родительской таблице или при удалении всей строки, содержащей данное значение, в дочерней (ссылающейся) таблице обновляются (т. е. удаляются) все строки с соответствующими значениями внешнего ключа.
- `SET NULL`. Разрешается модификация (обновление или удаление) любых значений в родительской таблице. Если обновление значения в родительской таблице вызывает несогласованность в дочерней таблице, система базы данных присваивает внешнему ключу всех соответствующих строк в дочерней таблице значение `NULL`. То же самое происходит и в случае удаления строки в родительской табли-

це, вызывающего несогласованность в дочерней таблице. Таким образом, все несогласованности данных пропускаются.

- SET DEFAULT. Аналогично опции SET NULL, но с одним исключением: всем внешним ключам, соответствующим модифицируемому первичному ключу, присваивается значение по умолчанию. Само собой разумеется, что после модификации первичный ключ родительской таблицы все равно должен содержать значение по умолчанию.

ПРИМЕЧАНИЕ

В языке Transact-SQL поддерживаются первые две из этих опций.

Использование опций ON DELETE и ON UPDATE показано в примере 5.12.

Пример 5.12. Применение опций ON DELETE и ON UPDATE

```
USE sample;
CREATE TABLE works_on1
   (emp_no INTEGER NOT NULL,
    project_no CHAR(4) NOT NULL,
    job CHAR (15) NULL,
    enter_date DATE NULL,
    CONSTRAINT prim_works1 PRIMARY KEY(emp_no, project_no),
    CONSTRAINT foreign1_works1 FOREIGN KEY(emp_no)
           REFERENCES employee(emp_no) ON DELETE CASCADE,
    CONSTRAINT foreign2_works1 FOREIGN KEY(project_no)
           REFERENCES projact(project_no) ON UPDATE CASCADE);
```

В примере 5.12 создается таблица works_on1 с использованием опций ON DELETE CASCADE и ON UPDATE CASCADE. Если таблицу works_on1 загрузить значениями из табл. 1.4, каждое удаление строки в таблице employee будет вызывать каскадное удаление всех строк в таблице works_on1, которые имеют значения внешнего ключа, соответствующие значениям первичного ключа строк, удаляемых в таблице employee. Подобным образом каждое обновление значения столбца project_no таблицы project будет вызывать такое же обновление всех соответствующих значений столбца project_no таблицы works_on1.

Создание других объектов баз данных

Кроме основных таблиц, которые существуют как самостоятельные сущности, реляционная база данных также содержит *представления* (view), которые являются виртуальными таблицами. Данные базовой таблицы существуют физически, т. е. сохранены на диске, тогда как представление извлекается из одной или нескольких базовых таблиц. Представление на основе одной или нескольких существующих таблиц баз данных (или представлений) создается с помощью инструкции CREATE VIEW и инструкции SELECT, которая является неотъемлемой частью инструкции

CREATE VIEW. Поскольку создание представления всегда содержит запрос, инструкция CREATE VIEW принадлежит к языку манипуляции данными (DML), а не к языку описания данных (DDL). По этой причине создание и удаление представлений рассматривается в *главе 11*, после представления инструкции Transact-SQL для модификации данных.

Инструкция CREATE INDEX создает новый *индекс* для указанной таблицы. Индексы в основном применяются для обеспечения эффективного доступа к данным, хранящимся на диске. Наличие индекса может значительно улучшить доступ к данным. Индексы совместно с инструкцией CREATE INDEX подробно рассматриваются в *главе 10*.

Еще одним дополнительным объектом базы являются *хранимые процедуры* (stored procedure), которые создаются посредством инструкции CREATE PROCEDURE. *Хранимая процедура* — это последовательность инструкций Transact-SQL, созданная посредством языка SQL и процедурных расширений. Хранимые процедуры подробно рассматриваются в *главе 8*.

Триггером (trigger) называется объект базы данных, который задает определенное действие в ответ на определенное событие. Это означает, что когда для предопределенной таблицы происходит определенное событие (модификации, вставка или удаление данных), компонент Database Engine автоматически запускает одно или несколько дополнительных действий. Триггеры создаются посредством инструкции CREATE TRIGGER и подробно рассматриваются в *главе 14*.

Синоним (synonym) — это локальный объект базы данных, который предоставляет связь между самим собой и другим объектом, управляемым одним и тем же или связанным сервером баз данных. Синонимы объектов создаются посредством инструкции CREATE SYNONYM, применение которой показано в примере 5.13.

Пример 5.13. Создание синонима объекта базы данных

```
USE AdventureWorks;
CREATE SYNONYM  prod
                FOR AdventureWorks.Production.Product;
```

В примере 5.13 создается синоним таблицы Product в схеме Production базы данных AdventureWorks. Этот синоним можно потом использовать в инструкциях языка DML, таких как SELECT, INSERT, UPDATE и DELETE.

ПРИМЕЧАНИЕ

Синонимы в основном используются во избежание необходимости применять длинные имена в инструкциях DML. Как уже упоминалось, имя объекта базы данных может состоять из четырех частей. Использование синонима, состоящего из одной части, для объекта с именем, состоящим из трех или четырех частей, позволяет сэкономить время на вводе имени такого объекта.

Схема (schema) — это объект базы данных, содержащий инструкции для создания таблиц, представлений и пользовательских разрешений. Схему можно рассматри-

вать, как конструкцию, в которой собраны вместе несколько таблиц, соответствующие представления и пользовательские разрешения.

ПРИМЕЧАНИЕ

В Database Engine применяется такое же понятие схемы, как и в стандарте ANSI SQL. В стандарте SQL схема определяется как коллекция объектов базы данных, имеющая одного владельца и формирующая одно пространство имен. *Пространство имен* (namespace) — это набор объектов с однозначными именами. Например, две таблицы могут иметь одно и то же имя только в том случае, если они находятся в разных схемах. Схема является очень важным концептом в модели безопасности компонента Database Engine. Подробно предмет схемы базы данных рассматривается в *главе 12*.

Ограничения для обеспечения целостности и домены

Домен (domain) — это набор всех возможных разрешенных значений, которые могут содержать столбцы таблицы. Почти во всех системах управления базами данных для определения таких возможных значений столбца используются такие типы данных, как `INT`, `CHAR` и `DATE`. Такого метода принудительного обеспечения "целостности домена" недостаточно, как можно увидеть в следующем примере.

В таблице `person` есть столбец `zip`, в котором указывается индекс города, в котором проживает данное лицо. Тип данных этого столбца можно определить как `SMALLINT` или `CHAR(5)`. Определение типа данных столбца как `SMALLINT` будет неточным, потому что этот тип данных содержит все положительные и отрицательные целые числа в диапазоне от $-2^{15} - 1$ до 2^{15}. Объявление с использованием типа данных `CHAR(5)` будет еще менее точным, поскольку в таком случае можно будет использовать все буквенно-цифровые и специальные символы. Поэтому для точного определения данных столбца индексов требуется диапазон положительных значений от `00601` и `99950`.

Более точно целостность домена можно принудительно обеспечить с помощью ограничений `CHECK` (определяемые в инструкции `CREATE TABLE` или `ALTER TABLE`), благодаря их гибкости и тому, что они всегда принудительно применяются при вставке или модифицировании данных столбца.

Язык Transact-SQL поддерживает домены посредством создания псевдонимов типов данных с помощью инструкции `CREATE TYPE`. Рассмотрению псевдонимов типов данных и типов данных среды CLR посвящены следующие два раздела.

Псевдонимы типов данных

Псевдоним типа данных (alias data type) — это специальный, который определяется пользователем при использовании существующих базовых типов данных. Такой тип данных можно использовать в инструкции `CREATE TABLE` для определения одного или большего количества столбцов таблицы.

Для создания псевдонимного типа данных обычно применяется инструкция CREATE TYPE. Далее показан синтаксис этой инструкции для определения псевдонимного типа данных:

```
CREATE TYPE [type_schema_name.] type_name
{[FROM base_type[(precision[ , scale ])] [NULL | NOT NULL]]
   | [EXTERNAL NAME assembly_name [.class_name]]}
```

Использование инструкции CREATE TYPE для создания псевдонимного типа данных показано в примере 5.14.

Пример 5.14. Создание псевдонимного типа данных с помощью инструкции CREATE TYPE

```
USE sample;
CREATE TYPE zip
   FROM SMALLINT NOT NULL;
```

В примере 5.14 создается псевдонимный тип данных zip на основе стандартного типа данных SMALLINT. Теперь этот определенный пользователем тип данных можно присвоить столбцу таблицы, как показано в примере 5.15.

ПРИМЕЧАНИЕ

Прежде чем выполнять этот пример, в базе данных AdventureWorks нужно удалить таблицу customer, используя для этого инструкцию DROP TABLE customer.

Пример 5.15. Определение столбца псевдонимным типом данных

```
USE sample;
CREATE TABLE customer
    (cust_no INT NOT NULL,
     cust_name CHAR(20) NOT NULL,
     city CHAR(20),
     zip_code ZIP,
CHECK (zip_code BETWEEN 601 AND 99950));
```

В примере 5.15 тип данных столбца zip_code таблицы customer определяется псевдонимным типом данных zip. Допустимые значения этого столбца требуется ограничить диапазоном целочисленных значений от 601 до 99950. Как можно видеть в примере 5.15, это ограничение можно наложить с помощью предложения CHECK.

ПРИМЕЧАНИЕ

Обычно компонент Database Engine неявно преобразовывает разные типы данных совместимых столбцов. Это также относится и к псевдонимным типам данных.

Начиная с версии SQL Server 2008, стали поддерживаться определяемые пользователем табличные типы. В примере 5.16 показано создание такого типа с помощью инструкции `CREATE TYPE`.

Пример 5.16. Создание табличного типа

```
USE sample;
CREATE TYPE person_table_t AS TABLE
   (name VARCHAR(30), salary DECIMAL(8,2));
```

Создаваемый в примере 5.16 определяемый пользователем табличный тип данных `person_table_t` имеет два столбца: `name` и `salary`. Основное синтаксическое отличие табличных типов от псевдонимных состоит в наличии предложения `AS TABLE`, как это можно видеть в примере 5.15. Определяемые пользователем табличные типы обычно применяются с возвращающими табличные значения параметрами (см. *главу 8*).

Типы данных CLR

Инструкцию `CREATE TYPE` можно также применить для создания определяемых пользователем типов данных с использованием .NET. В этом случае, реализация определяемого пользователем типа определяется в классе сборки в среде CLR. Это означает, что для реализации нового типа данных можно использовать один из языков .NET, такой как C# или Visual Basic. Более подробное рассмотрение определяемых пользователем типов данных выходит за рамки тематики этой книги.

Модифицирование объектов баз данных

Язык Transact-SQL поддерживает модифицирование структуры следующих объектов базы данных, среди прочих:

- базы данных;
- таблицы;
- хранимые процедуры;
- представления;
- схемы;
- триггеры.

В последующих двух разделах описывается изменение первых двух объектов из этого списка: баз данных и таблиц. Изменение структуры последних четырех объектов этого списка описывается в *главе 18*, *11*, *12* и *14* соответственно.

Изменение базы данных

Для изменения физической структуры базы данных используется инструкция `ALTER DATABASE`. Язык Transact-SQL позволяет выполнять следующие действия по изменению свойств базы данных:

- добавлять и удалять один или несколько файлов базы данных;
- добавлять и удалять один или несколько файлов журнала;

- добавлять и удалять файловые группы;
- изменять свойства файлов или файловых групп;
- устанавливать параметры базы данных;
- изменять имя базы данных с помощью хранимой процедуры `sp_rename` (рассматривается в *разд. "Изменение таблиц"* далее в этой главе).

Эти разные типы модификаций базы данных рассматриваются в последующих далее подразделах. В этом разделе мы также используем инструкцию `ALTER DATABASE`, чтобы показать, как можно сохранять данные типа `FILESTREAM` в файлах и файловых группах, а также для объяснения концепции автономной базы данных.

Добавление и удаление файлов базы данных, файлов журналов и файловых групп

Добавление или удаление файлов базы данных осуществляется посредством инструкции `ALTER DATABASE`. Операция добавления нового или удаления существующего файла указывается предложением `ADD FILE` и `REMOVE FILE` соответственно. Кроме этого, новый файл можно определить в существующую файловую группу посредством параметра `TO FILEGROUP`.

В примере 5.17 показано добавление нового файла базы данных в базу данных `projects`.

Пример 5.17. Добавление нового файла в базу данных

```
USE master;
GO
ALTER DATABASE projects
ADD FILE (NAME=projects_dat1,
   FILENAME = 'C:\projects1.mdf', SIZE = 10,
   MAXSIZE = 100, FILEGROWTH = 5);
```

В примере 5.17 инструкция `ALTER DATABASE` добавляет новый файл с логическим именем `projects_dat1`. Здесь же указан начальный размер файла 10 Мбайт и автоувеличение по 5 Мбайт до максимального размера 100 Мбайт. Файлы журналов добавляются так же, как и файлы баз данных. Единственным отличием является то, что вместо предложения `ADD FILE` используется предложение `ADD LOG FILE`.

Удаления файлов (как файлов базы данных, так и файлов журнала) из базы данных осуществляется посредством предложения `REMOVE FILE`. Удаляемый файл должен быть пустым.

Новая файловая группа создается посредством предложения `CREATE FILEGROUP`, а существующая удаляется с помощью предложения `DELETE FILEGROUP`. Как и удаляемый файл, удаляемая файловая группа также должна быть пустой.

Изменение свойств файлов и файловых групп

С помощью предложения MODIFY FILE можно выполнять следующие действия по изменению свойств файла:

- изменять логическое имя файла, используя параметр NEWNAME;
- увеличивать значение свойства SIZE;
- изменять значение свойств FILENAME, MAXSIZSE и FILEGROWTH;
- отмечать файл как OFFLINE.

Подобным образом с помощью предложения MODIFY FILEGROUP можно выполнять следующие действия по изменению свойств файловой группы:

- изменять логическое имя файловой группы, используя параметр NAME;
- помечать файловую группу, как файловую группу по умолчанию, используя для этого параметр DEFEAULT;
- помечать файловую группу как позволяющую осуществлять доступ только для чтения или для чтения и записи, используя для этого параметр READ_ONLY или READ_WRITE соответственно.

Установка опций базы данных

Для установки различных опций базы данных используется предложение SET инструкции ALTER DATABASE. Некоторым опциям можно присвоить только значения ON или OFF, но для большинства из них предоставляется выбор из списка возможных значений. Каждый параметр базы данных имеет значение по умолчанию, которое устанавливается в базе данных model. Поэтому значения определенных опций по умолчанию можно модифицировать, изменив соответствующим образом базу данных model.

Все опции, значения которых можно изменять, можно разбить на несколько групп, наиболее важными из которых являются следующие:

- опции состояния;
- опции автоматических действий;
- опции SQL.

Опции состояния управляют следующими возможностями:

- доступом пользователей к базе данным (это опции SINGLE_USER, RESTRICTED_USER и MULTI_USER);
- статусом базы данных (это опции ONLINE, OFFLINE и EMERGENCY);
- режимом чтения и записи (опции READ_ONLY и READ_WRITE).

Опции автоматических операций управляют, среди прочего, остановом базы данных (опция AUTO_CLOSE) и способом создания статистики индексов (опции AUTO_CREATE_STATISTICS и AUTO_UPDATE_STATISTICS).

Опции SQL управляют соответствием базы данных и ее объектов стандарту ANSI. Значения всех операторов SQL можно узнать посредством функции

DATABASEPROPERTYEX, а редактировать — с помощью инструкции ALTER DATABASE. Опции восстановления FULL, BULK-LUGGED и SIMPLE управляют процессом восстановления базы данных.

Хранение данных типа *FILESTREAM*

В предыдущей главе мы рассмотрели данные типа FILESTREAM и причины, по которым их используют. В этом разделе мы рассмотрим, как данные типа FILESTREAM можно сохранять в базе данных. Чтобы данные FILESTREAM можно было сохранять в базе данных, система должна быть должным образом инициирована. В следующем подразделе объясняется, как инициировать операционную систему и экземпляр базы данных для хранения данных типа FILESTREAM.

Инициирование хранилища *FILESTREAM*

Хранилище данных типа FILESTREAM требуется инициировать на двух уровнях:

- для операционной системы Windows;
- для конкретного экземпляра сервера базы данных.

Инициирование хранилища данных типа FILESTREAM на уровне системы осуществляется с помощью диспетчера конфигурации SQL Server. Чтобы запустить диспетчер конфигурации, выполните следующую последовательность команд по умолчанию **Пуск | Все программы | Microsoft SQL Server 2012 | Configuration Tools** (Средства настройки | Диспетчер конфигурации SQL Server Configuration Manager). В открывшемся окне **Sql Server Configuration Manager** (Sql Server Configuration Manger) щелкните правой кнопкой пункт **SQL Server Services** (Службы SQL Server) и в появившемся контекстном меню выберите команду **Open** (Открыть). В правой панели щелкните правой кнопкой экземпляр, для которого требуется разрешить хранилище FILESTREAM, и в контекстном меню выберите команду **Properties** (Свойства). В открывшемся диалоговом окне **SQL Server Properties** (Свойства | SQL Server) выберите вкладку **FILESTREAM** (FILESTREAM) (рис. 5.1).

Чтобы иметь возможность только читать данные типа FILESTREAM, установите флажок **Enable FILESTREAM for Transact-SQL access** (Разрешить FILESTREAM при доступе через Transact-SQL). Чтобы кроме чтения можно было также записывать данные, установите дополнительно флажок **Enable FILESTREAM for file I/O streaming access** (Разрешить использование FILESTREAM при доступе файлового ввода/вывода). Введите имя общей папки Windows в одноименное поле. Общая папка Windows используется для чтения и записи данных FILESTREAM, используя интерфейс API Win32. Если для возвращения пути для FILESTREAM BLOB использовать имя, то это будет имя общей папки Windows.

Диспетчер конфигурации SQL Server создаст на системе хоста новую общую папку с указанным именем. Чтобы применить изменения, нажмите кнопку **OK**.

ПРИМЕЧАНИЕ

Чтобы разрешить хранилище FILESTREAM, необходимо быть администратором Windows локальной системы и обладать правами администратора (syssdmin). Чтобы изменения вступили в силу, необходимо перезапустить экземпляр сервера базы данных.

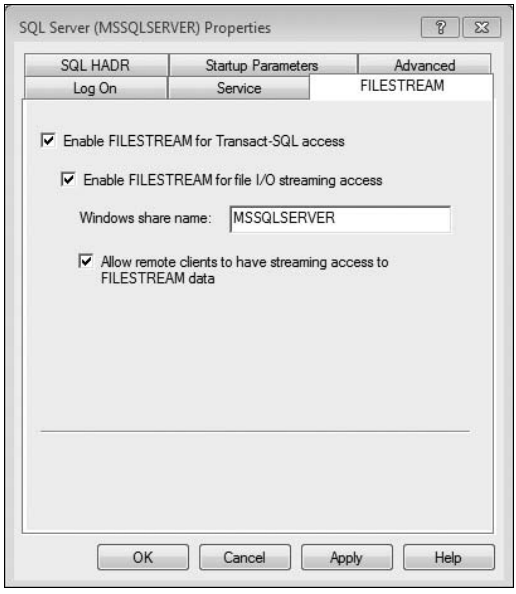


Рис. 5.1. Диалоговое окно **SQL Server Properties**, вкладка **FILESTREAM**

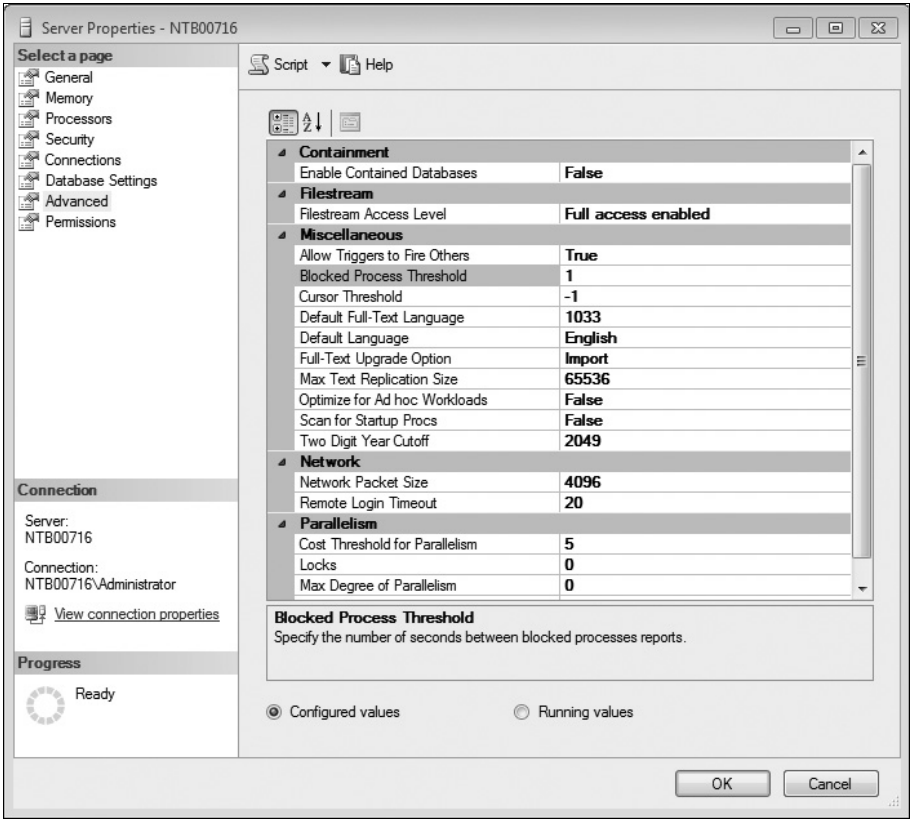


Рис. 5.2. Диалоговое окно **Server Properties** с уровнем доступа FILESTREAM, установленным в **Full Access Enabled**

Следующим шагом будет разрешить хранилище FILESTREAM для конкретного экземпляра. Мы рассмотрим, как выполнить эту задачу с помощью среды SQL Server Management Studio. (Для этого можно также воспользоваться хранимой системной процедурой sp_configure с параметром filestream access level.) Щелкните правой кнопкой требуемый экземпляр в обозревателе объектов и в появившемся контекстном меню выберите пункт **Properties** (Свойства), в левой панели открывшегося диалогового окна **Server Properties** (Свойства сервера) выберите пункт **Advanced** (Дополнительно) (рис. 5.2), после чего в правой панели из выпадающего списка выберите **Filestream Access Level** (Уровень доступа FILESTREAM) одну из следующих опций:

- **Disabled** (Отключено) — хранилище FILESTREAM не разрешено;
- **Transact-SQL Access Enabled** (Включен доступ с помощью Transact-SQL) — к данным FILESTREAM можно обращаться посредством инструкций T-SQL;
- **Full Access Enabled** (Включен полный доступ). К данным FILESTREAM можно обращаться как посредством инструкций T-SQL, так и через интерфейс API Win32.

Добавление файла в файловую группу

Разрешив хранилище FILESTREAM для требуемого экземпляра, можно сначала создать файловую группу для данных FILESTREAM (посредством инструкции ALTER DATABASE), а затем добавить файл в эту файловую группу, как это показано в примере 5.18. (Конечно же, эту задачу также можно было бы выполнить с помощью инструкции CREATE DATABASE.)

ПРИМЕЧАНИЕ

Прежде чем выполнять инструкции, приведенные в примере 5.18, измените имя файла в предложении FILENAME.

Пример 5.18. Добавление файла в файловую группу

```
USE sample;
ALTER DATABASE sample
   ADD FILEGROUP Employee_FSGroup CONTAINS FILESTREAM;
GO
ALTER DATABASE sample
   ADD FILE (NAME= employee_FS,
   FILENAME = 'C:\DUSAN\emp_FS')
   TO FILEGROUP Employee_FSGroup
```

Первая инструкция ALTER DATABASE в примере 5.18 добавляет в базу данных sample новую файловую группу Employee_FSGroup. Параметр CONTAINS FILESTREAM этой инструкции указывает системе, что данная файловая группа будет содержать только

данные FILESTREAM. Вторая инструкция ALTER DATABASE добавляет в созданную файловую группу новый файл.

Теперь можно создавать таблицы, содержащие столбцы с типом данных FILESTREAM. Создание такой таблицы показано в примере 5.19.

Пример 5.19. Создание таблицы, содержащей столбец FILESTREAM

```
CREATE TABLE employee_info
   (id UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE,
   filestream_data VARBINARY(MAX) FILESTREAM NULL)
```

В примере 5.19 таблица employee_info содержит столбец filestream_data, тип данных которого должен быть VARBINARY(max). Определение такого столбца включает атрибут FILESTREAM, указывающий, что данные столбца сохраняются в файловой группе FILESTREAM. Для всех таблиц, в которых хранятся данные типа FILESTREAM, требуется наличие свойств UNIQUE ROWGUILDCOL. Поэтому таблица employee_info содержит столбец id, определенный с использованием этих двух атрибутов.

Данные в столбец типа FILESTREAM вставляются посредством стандартной инструкции INSERT, которая рассматривается в *главе 7*. А для считывания данных используется стандартная инструкция SELECT, которая рассматривается *далее в этой главе*. Подробное рассмотрение операций записи и чтение данных типа FILESTREAM выходят за рамки тематики этой книги.

Автономные базы данных

Одна из значительных проблем с базами данных SQL Server состоит в том, что они трудно поддаются экспортированию и импортированию. Как рассматривалось ранее в этой главе, базы данных можно присоединять и отсоединять, но при этом утрачиваются важные части и свойства присоединенных баз данных. (Основной проблемой в таких случаях является безопасность базы данных, в общем, и учетные записи, в частности, в которых после перемещения обычно отсутствует часть информации или содержится неправильная информация.)

Разработчики Microsoft планируют решить эти проблемы посредством использования *автономных баз данных* (contained databases). Автономная база данных содержит все параметры и данные, необходимые для определения базы данных, и изолирована от экземпляра Database Engine, на котором она установлена. Иными словами, база данных данного типа не имеет конфигурационных зависимостей от экземпляра и ее можно с легкостью перемещать с одного экземпляра SQL Server на другой.

По большому счету, что касается автономности, существует три вида баз данных:

- полностью автономные базы данных;
- частично автономные базы данных;
- неавтономные базы данных.

Полностью автономными являются такие базы данных, объекты которые не могут перемещаться через границы приложения. (Граница приложения определяет область видимости приложения. Например, пользовательские функции находятся в границах приложения, в то время как функции, связанные с экземплярами сервера, находятся вне границ приложения.)

Частично автономные базы данных позволяют объектам пересекать границы приложения, в то время как неавтономные базы данных вообще не поддерживают концепции границы приложения.

ПРИМЕЧАНИЕ

В SQL Server 2012 поддерживаются частично автономные базы данных. В будущих версиях SQL Server также будет поддерживаться полная автономность. Базы данных предшествующих версий SQL Server являются неавтономными.

Рассмотрим, как создать частично автономную базу данных в SQL Server 2012. Если существующая база данных `my_sample` является неавтономной (созданная, например, посредством инструкции `CREATE DATABASE`), с помощью инструкции `ALTER DATABASE` ее можно преобразовать в частично автономную, как это показано в примере 5.20.

Пример 5.20. Преобразование неавтономной базы данных в частично автономную

```
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE WITH OVERRIDE;
EXEC sp_configure 'contained database authentication', 1;
RECONFIGURE WITH OVERRIDE;
ALTER DATABASE my_sample SET CONTAINMENT = PARTIAL;
EXEC sp_configure 'show advanced options', 0;
RECONFIGURE WITH OVERRIDE;
```

Инструкция `ALTER DATABASE` изменяет состояние автономности базы данных `my_sample` с неавтономного на частично автономное. Это означает, что теперь система базы данных позволяет создавать как автономные, так неавтономные объекты для базы данных `my_sample`. Все другие инструкции в примере 5.20 являются вспомогательными для инструкции `ALTER DATABASE`.

ПРИМЕЧАНИЕ

Функция `sp_configure` является системной процедурой, с помощью которой можно, среди прочего, изменить дополнительные параметры конфигурации, такие как `contained database authentication`. Чтобы изменить дополнительные параметры конфигурации, сначала нужно присвоить параметру `show advanced options` значение `1`, а потом переконфигурировать систему (инструкция `RECONFIGURE`). В конце кода примера 5.20 этому параметру опять присваивается его значение по умолчанию — `0`. Системная процедура `sp_configure` подробно рассматривается в *разд. "Системные хранимые процедуры" главы 9*.

Теперь для базы данных sp_smaple можно создать пользователя, не привязанного к учетной записи. Этот процесс рассматривается подробно в *разд. "Управление авторизацией и аутентификацией для автономных баз данных" главы 12*.

Изменение таблиц

Для модифицирования схемы таблицы применяется инструкция ALTER TABLE. Язык Transact-SQL позволяет осуществлять следующие виды изменений таблиц:

- добавлять и удалять столбцы;
- изменять свойства столбцов;
- добавлять и удалять ограничения для обеспечения целостности;
- разрешать или отключать ограничения;
- переименовывать таблицы и другие объекты базы данных.

Эти типы изменений рассматриваются в последующих далее разделах.

Добавление и удаление столбцов

Чтобы добавить новый столбец в существующую таблицу, в инструкции ALTER TABLE используется предложение ADD. В одной инструкции ALTER TABLE можно добавить только один столбец. Применение предложения ADD показано в примере 5.21.

Пример 5.21. Добавление нового столбца в таблицу

```
USE sample;
ALTER TABLE employee
   ADD telephone_no CHAR(12) NULL;
```

В примере 5.21 инструкция ALTER TABLE добавляет в таблицу employee столбец telephone_no. Компонент Database Engine заполняет новый столбец значениями NULL или IDENTITY или указанными значениями по умолчанию. По этой причине новый столбец должен или поддерживать свойство содержать значения NULL, или для него должно быть указано значение по умолчанию.

ПРИМЕЧАНИЕ

Новый столбец нельзя вставить в таблицу в какой-либо конкретной позиции. Столбец, добавляемый предложением ADD, всегда вставляется в конец таблицы.

Столбцы из таблицы удаляются посредством предложения DROP COLUMN. Применение этого предложения показано в примере 5.22.

Пример 5.22. Удаление столбца таблицы

```
USE sample;
ALTER TABLE employee
   DROP COLUMN telephone_no;
```

В примере 5.22 инструкция ALTER TABLE удаляет в таблице employee столбец telephone_no, который был добавлен в эту таблицу предложением ADD в примере 5.21.

Изменение свойств столбцов

Для изменения свойств существующего столбца применяется предложение ALTER COLUMN инструкции ALTER TABLE. Изменению поддаются следующие свойства столбца:

- тип данных;
- свойство столбца принимать значения NULL.

Применение предложения ALTER COLUMN показано в примере 5.23.

Пример 5.23. Изменение свойств столбца

```
USE sample;
ALTER TABLE department
   ALTER COLUMN location CHAR(25) NOT NULL;
```

В примере 5.23 инструкция ALTER TABLE изменяет начальные свойства (CHAR(30), значения NULL разрешены) столбца location таблицы department на новые (CHAR(25), значения NULL не разрешены — NOT NULL).

Добавление и удаления ограничений для обеспечения целостности

Для добавления в таблицу новых ограничений для обеспечения целостности используется параметр ADD CONSTRAINT инструкции ALTER TABLE. В примере 5.24 показано использование параметра ADD CONSTRAINT для добавления проверочного ограничения.

Пример 5.24. Добавление проверочного ограничения

```
USE sample;
CREATE TABLE sales
   (order_no INTEGER NOT NULL,
   order_date DATE NOT NULL,
   ship_date DATE NOT NULL);
ALTER TABLE sales
      ADD CONSTRAINT order_check CHECK(order_date <= ship_date);
```

В примере 5.24 инструкцией CREATE TABLE сначала создается таблица sales, содержащая два столбца с типом данных DATE: order_date и ship_date. Далее, инструкция ALTER TABLE определяет ограничение для обеспечения целостности order_check, которое сравнивает значения обоих этих столбцов и выводит сообщение об ошибке, если дата отправки ship_date более ранняя, чем дата заказа order_date.

В примере 5.25 показано использование инструкции ALTER TABLE для определения первичного ключа таблицы.

Пример 5.25. Определение первичного ключа таблицы

```
USE sample;
ALTER TABLE sales
ADD CONSTRAINT primaryk_sales PRIMARY KEY(order_no);
```

В примере 5.25 столбец order_no определен, как первичный ключ таблицы sales.

Ограничения для обеспечения целостности можно удалить посредством предложения DROP CONSTRAINT инструкции ALTER TABLE, как это показано в примере 5.26.

Пример 5.26. Удаление ограничений для обеспечения целостности

```
USE sample;
ALTER TABLE sales
   DROP CONSTRAINT order_check;
```

В примере 5.26 инструкция ALTER TABLE удаляет проверочное ограничение order_check, установленное в примере 5.24.

ПРИМЕЧАНИЕ

Определения существующих ограничений нельзя модифицировать. Чтобы изменить ограничение, его сначала нужно удалить, а потом создать новое, содержащее требуемые модификации.

Разрешение и запрещение ограничений

Как упоминалось ранее, ограничение для обеспечения целостности всегда имеет имя, которое может быть объявленным или явно посредством опции CONSTRAINT, или неявно посредством системы. Имена всех ограничений таблицы (объявленных как явно, так и неявно) можно просмотреть с помощью системной процедуры sp_helpconstraint.

В последующих операциях вставки или обновлений значений в соответствующий столбец ограничение по умолчанию обеспечивается принудительно. Кроме этого, при объявлении ограничения все существующие значения соответствующего столбца проверяются на удовлетворение условий ограничения. Начальная проверка не выполняется, если ограничение создается с параметром WITH NOCHECK. В таком случае ограничение будет проверяться только при последующих операциях вставки и обновлений значений соответствующего столбца. (Оба параметра — WITH CHECK и WITH NOCHECK — можно применять только с ограничениями проверки целостности CHECK и проверки внешнего ключа FOREIGN KEY.)

В примере 5.27 показано, как отключить все существующие ограничения таблицы.

Пример 5.27. Отключение ограничений таблицы

```
USE sample;
ALTER TABLE sales
NOCHECK CONSTRAINT ALL;
```

В примере 5.27 все ограничения таблицы `sales` отключаются посредством ключевого слова `ALL`.

ПРИМЕЧАНИЕ

Применять опцию `NOCHECK` не рекомендуется, поскольку любые подавленные нарушения условий ограничения могут вызвать ошибки при будущих обновлениях.

Переименование таблиц и других объектов баз данных

Для изменения имени существующей таблицы (и любых других объектов базы данных, таких как база данных, представление или хранимая процедура) применяется системная процедура `sp_rename`. В примерах 5.28—5.29 показано использование этой системной процедуры.

Пример 5.28. Переименование таблицы

```
USE sample;
EXEC sp_rename @objname = department, @newname = subdivision
```

В примере 5.28 таблице `department` присваивается новое имя `subdivision`.

Пример 5.29. Переименование столбца таблицы

```
USE sample;
EXEC sp_rename @objname = 'sales.order_no', Snewname = ordernumber
```

В примере 5.29 столбцу `order_no` таблицы `sales` присваивается новое имя `ordernumber`. При переименовании столбца таблицы имя этого столбца требуется указывать в виде: *`table_name.column_name`* (т. е. *`имя_таблицы.имя_столбца`*).

ПРИМЕЧАНИЕ

Использовать системную процедуру `sp_rename` настоятельно не рекомендуется, поскольку изменение имен объектов может повлиять на другие объекты базы данных, которые ссылаются на них. Вместо этого следует удалить объект и воссоздать его с новым именем.

Удаление объектов баз данных

Все инструкции Transact-SQL для удаления объектов базы данных имеют следующий общий вид:

```
DROP object_type object_name
```

Для каждой инструкции `CREATE object` для создания объекта имеется соответствующая инструкция `DROP object` для удаления. Инструкция для удаления одной или нескольких баз данных имеет следующий вид:

```
DROP DATABASE database1 {, ...}
```

Эта инструкция безвозвратно удаляет базу данных из системы баз данных.

Для удаления одной или нескольких таблиц применяется следующая инструкция:

```
DROP TABLE table_name1 {, ...}
```

При удалении таблицы удаляются все ее данные, индексы и триггеры. Но представления, созданные по удаленной таблице, не удаляются. Таблицу может удалить только пользователь, имеющий соответствующие разрешения.

Кроме объектов `DATABASE` и `TABLE`, в параметре `objects` инструкции `DROP` можно указывать, среди прочих, следующие объекты:

- `TYPE` (тип);
- `SYNONYM` (синоним);
- `PROCEDURE` (процедура);
- `INDEX` (индекс);
- `VIEW` (представление);
- `TRIGGER` (триггер);
- `SCHEMA` (схема).

Инструкции `DROP TYPE` и `DROP SYNONYM` удаляют тип и синоним соответственно. Остальные инструкции рассматриваются в других главах: `DROP PROCEDURE` — в *главе 8*, `DROP INDEX` — в *главе 10*, `DROP VIEW` — в *главе 11*, `DROP SCHEMA` — в *главе 12* и `DROP TRIGGER` — в *главе 14*.

Резюме

Язык Transact-SQL поддерживает большое число различных инструкций описания данных для создания, изменения и удаления объектов баз данных. Используя инструкции `CREATE object` и `DROP object` можно создавать и удалять, соответственно, следующие объекты баз данных:

- базы данных;
- таблицы;
- схемы;
- представления;
- триггеры;
- хранимые процедуры;
- индексы.

Структуру любого из перечисленных в предшествующем списке объектов баз данных можно изменить с помощью инструкции `ALTER object`. Обратите внимание на

то обстоятельство, что единой стандартной инструкцией в данном списке является инструкция ALTER TABLE. Все другие инструкции типа ALTER *object* являются расширениями стандарта SQL для языка Transact-SQL.

В следующей главе мы рассмотрим инструкцию SELECT, которая используется для манипулирования данными.

Упражнения

Упражнение 5.1

Используя инструкцию CREATE DATABASE, создайте новую базу данных test_db, задав явные спецификации для файлов базы данных и журнала транзакций. Файл базы данных с логическим именем test_db_dat сохраняется в физическом файле C:\tmp\test_db.mdf, его начальный размер — 5 Мбайт, автоувеличение по 8%, максимальный размер не ограничен. Файл журнала транзакций с логическим именем test_db_log сохраняется в физическом файле C:\tmp\test_db_log.ldf, его начальный размер — 2 Мбайта, автоувеличение по 500 Кбайт, максимальный размер 10 Мбайт.

Упражнение 5.2

Используя инструкцию ALTER DATABASE, добавьте новый файл журнала в базу данных test_db. Файл сохраняется в физическом файле C:\tmp\emp_log.ldf, его начальный размер — 2 Мбайта, автоувеличение по 2 Мбайта, максимальный размер не ограничен.

Упражнение 5.3

Используя инструкцию ALTER DATABASE, измените начальный размер файла базы данных test_db на 10 Мбайт.

Упражнение 5.4

В примере 5.4 для некоторых столбцов четырех созданных таблиц запрещены значения NULL. Для каких из этих столбцов это определение является обязательным, а для каких нет?

Упражнение 5.5

Почему в примере 5.4 тип данных для столбцов dept_no и project_no определен как CHAR, а не как один из целочисленных типов?

Упражнение 5.6

Создайте таблицы customers и orders, содержащие перечисленные в следующей таблице столбцы. Не объявляйте соответствующие первичный и внешние ключи.

customers	orders
customerid char(5) not null	orderid integer not null
companyname varchar(40) not null	customerid char(5) not null
contactname char(30) null	orderdate date null
address varchar(60) null	shippeddate date null
city char(15) null	freight money null
phone char(24) null	shipname varchar(40) null
fax char(24) null	shipaddress varchar(60) null
	quantity integer null

Упражнение 5.7

Используя инструкцию ALTER TABLE, добавьте в таблицу orders новый столбец shipregion. Столбец должен иметь целочисленный тип данных и разрешать значения NULL.

Упражнение 5.8

Используя инструкцию ALTER TABLE, измените тип данных столбца shipregion с целочисленного на буквенно-цифровой длиной 8 символов. Столбец может содержать значения NULL.

Упражнение 5.9

Удалите созданный ранее столбец shipregion.

Упражнение 5.10

Дайте точное описание происходящему при удалении таблицы с помощью инструкции DROP TABLE.

Упражнение 5.11

Создайте заново таблицы customers и orders, усовершенствовав их определение всеми ограничениями первичных и внешних ключей.

Упражнение 5.12

Используя среду SQL Server Management Studio, попробуйте вставить следующую новую строку в таблицу orders:

```
(10, 'ord01', getdate(), getdate(), 100.0, 'Windstar', 'Ocean', 1).
```

Почему система отказывается вставлять эту строку в таблицу?

Упражнение 5.13

Используя инструкцию ALTER TABLE, определите значение по умолчанию столбца orderdate таблицы orders в виде текущей даты и времени системы.

Упражнение 5.14

Используя инструкцию `ALTER TABLE`, создайте ограничение для обеспечения целостности, ограничивающее допустимые значения столбца `quantity` таблицы `orders` диапазоном значений от 1 до 30.

Упражнение 5.15

Отобразите все ограничения для обеспечения целостности таблицы `orders`.

Упражнение 5.16

Попытайтесь удалить первичный ключ таблицы `customers`. Почему это не удается?

Упражнение 5.17

Удалите ограничение для обеспечения целостности `prim_empl`, определенное в примере 5.7.

Упражнение 5.18

В таблице `customers` измените имя столбца `city` на `town`.

Глава 6

Запросы

- **Инструкция *SELECT*. Ее предложения и функции**
- **Подзапросы**
- **Временные таблицы**
- **Операторы соединения**
- **Связанные подзапросы**
- **Табличные выражения**

В первой части этой главы обсуждается, как использовать инструкцию `SELECT` для выборки данных из таблиц. В частности, рассматриваются все предложения этой инструкции и приводятся многочисленные примеры с использованием базы данных `sample` для демонстрации практического применения каждого предложения. Далее дается введение в агрегатные функции и операторы над множествами, вычисляемые столбцы и временные таблицы. Во второй части главы рассматриваются более сложные запросы. Представляется наиболее важный оператор реляционной системы баз данных — оператор соединения (join operator) и рассматриваются все его формы. Далее представляются подзапросы и функция `EXISTS`. В конце главы рассматриваются распространенные табличные выражения и оператор `APPLY`.

Инструкция *SELECT*. Ее предложения и функции

В языке Transact-SQL имеется одна основная инструкция для выборки информации из базы данных — инструкция `SELECT`. Эта инструкция позволяет извлекать информацию из одной или нескольких таблиц базы данных и даже из нескольких баз данных. Результаты выполнения инструкции `SELECT` помещаются в еще одну таблицу, называемую *результирующим набором* (result set).

Самая простая форма инструкции SELECT состоит из списка столбцов выборки и предложения FROM. (Все прочие предложения являются необязательными.) Эта форма инструкции SELECT имеет следующий синтаксис:

```
SELECT [ALL | DISTINCT] column_list
   FROM {table1 [tab_alias1]},...
```

В параметре *table1* указывается имя таблицы, из которой извлекается информация, а в параметре *tab_alias1* — псевдоним имени соответствующей таблицы. *Псевдоним* — это другое, сокращенное, имя таблицы, посредством которого можно обращаться к этой таблице или к двум логическим экземплярам одной физической таблицы. Если вы испытываете трудности с пониманием этого концепта, не волнуйтесь, поскольку все станет ясно в процессе рассмотрения примеров.

В параметре *column_list* указывается один или несколько из следующих спецификаторов:

- символ звездочка (*) указывает все столбцы таблиц, перечисленных в предложении FROM (или с одной таблицы, если задано квалификатором в виде *table2.**);
- явное указание имен столбцов, из которых нужно извлечь значения;
- спецификатор в виде *column_name* [AS] *column_heading*, что позволяет заменить имя столбца или присвоить новое имя выражению;
- выражение;
- системная или агрегатная функция.

ПРИМЕЧАНИЕ

Кроме только что перечисленных спецификаторов есть и другие опции, которые рассматриваются далее в этой главе.

Инструкция SELECT может извлекать из таблицы как отдельные столбцы, так и строки. Первая операция называется *списком выбора* (или *проекцией*), а вторая — *выборкой*. Инструкция SELECT также позволяет выполнять комбинацию обеих этих операций.

ПРИМЕЧАНИЕ

Перед тем как можно исполнять пример запросов в этой главе, необходимо заново создать базу данных sample.

В примере 6.1 показана самая простая форма выборки данных посредством инструкции SELECT.

Пример 6.1. Извлечение полной информации об отделах

```
USE sample;
SELECT dept_no, dept_name, location
   FROM department;
```

Результат выполнения этого запроса:

dept_no	dept_name	location
D1	Research	Dallas
D2	Accounting	Seattle
D3	Marketing	Dallas

В примере 6.1 инструкция SELECT извлекает все строки всех столбцов таблицы department. Если список выбора инструкции SELECT содержит все столбцы таблицы (как это показано в примере 6.1), их можно указать с помощью звездочки (*), но использовать этот способ не рекомендуется. Имена столбцов служат в качестве заголовков столбцов в результирующем выводе.

Только что рассмотренная простейшая форма инструкции SELECT не очень полезна для практических запросов. На практике в запросах с инструкцией SELECT приходится применять намного больше предложений, чем в запросе, приведенном в примере 6.1. Далее показан синтаксис инструкции SELECT, содержащий почти все возможные предложения:

```
SELECT select_list
   [INTO new_table]
   FROM table
   [WHERE search_condition]
   [GROUP BY group_by_expression]
   [HAVING search_condition]
   [ORDER BY order_expression [ASC | DESC]];
```

ПРИМЕЧАНИЕ

Порядок предложений в инструкции SELECT должен быть таким, как показано в приведенном синтаксисе. Например, предложение GROUP BY должно следовать за предложением WHERE и предшествовать предложению HAVING. Предложение INTO не является настолько важным, как другие предложения, и поэтому будет рассмотрено позже других.

В последующих подразделах мы рассмотрим эти предложения в том порядке, в каком они следуют в запросе, а также рассмотрим свойство IDENTITY, возможность упорядочивания результатов, операторы над множествами и выражение CASE. Но так как первое в списке предложение INTO менее важно, чем остальные, оно будет рассматриваться позже, после всех других предложений.

Предложение *WHERE*

Часто при выборке данных из таблицы нужны данные только из определенных строк, для чего в запросе определяется одно или несколько соответствующих условий. В предложении WHERE определяется логическое выражение (т. е. выражение, возвращающее одно из двух значений: TRUE или FALSE), которое проверяется для

каждой из строк, кандидатов на выборку. Если строка удовлетворяет условию выражения, т. е. выражение возвращает значение TRUE, она включается в выборку; в противном случае строка пропускается. Применение предложения WHERE показано в примере 6.2.

Пример 6.2. Выборка имен и номеров отделов, расположенных в городе Даллас

```
USE sample;
SELECT dept_name, dept_no
   FROM department
   WHERE location = 'Dallas';
```

Результат выполнения этого запроса:

Dept_name	dept_no
Research	d1
Marketing	d3

Кроме знака равенства, в предложении WHERE могут применяться другие операторы сравнения, включая следующие:

<> (или !=)	не равно
<	меньше чем
>	больше чем
>=	больше чем или равно
<=	меньше чем или равно
!>	не больше чем
!<	не меньше чем

В примере 6.3 показано использование в предложении WHERE одного из этих операторов сравнения: >= (больше чем или равно).

Пример 6.3. Выборка имен и фамилий всех сотрудников, чей табельный номер больше или равен 15 000

```
USE sample;
SELECT emp_lname, emp_fname
   FROM employee
   WHERE emp_no >= 15000;
```

Результат выполнения этого запроса:

emp_lname	emp_fname
Smith	Matthew
Barrimore	John
James	James
Moser	Sybill

Частью условия в предложении WHERE может быть выражение, показанное в примере 6.4.

Пример 6.4. Выборка проектов с бюджетом свыше 60 000 фунтов стерлингов. Валютный курс: 0.51 фунтов стерлингов за $1

```
USE sample;
SELECT project_name
   FROM project
   WHERE budget*0.51 > 60000;
```

Результат выполнения этого запроса будет таким:

project_name
Apollo
Mercury

Сравнение строк (т. е. значений с типами данных CHAR, VARCHAR, NCHAR и NVARCAR) выполняется в действующем порядке сортировки, а именно в порядке сортировки, указанном при установке компонента Database Engine. При сравнении строк в кодировке ASCI (или в любой другой кодировке) сравниваются соответствующие символы каждой строки (т. е. первый символ первой строки с первым символом второй строки, второй символ первой строки со вторым символом второй строки и т. д.). Старшинство символа определяется его позицией в кодовой таблице: символ, чей код стоит в таблице перед кодом другого, считается меньше этого символа. При сравнении строк разной длины, более короткая строка дополняется в конце пробелами до длины более длинной строки. Числа сравниваются алгебраически. Значения временных типов данных (таких как DATE, TIME и DATETIME) сравниваются в хронологическом порядке.

Логические операторы

Условия предложения WHERE могут быть простыми или составными, т. е. содержащими несколько простых условий. Множественные условия можно создавать посредством логических операторов AND, OR и NOT. Действия этих операторов изложены в таблицах истинности, приведенных в *главе 4*.

При соединении условий оператором AND возвращаются только те строки, которые удовлетворяют обоим условиям. При соединении двух условий оператором OR возвращаются все строки таблицы, которые удовлетворяют одному или обоим этим условиям, как показано в примере 6.5.

Пример 6.5. Выборка номеров сотрудников, которые работают над проектом p1 или p2 (или над обоими)

```
USE sample;
SELECT project_no, emp_no
```

```
   FROM works_on
   WHERE project_no = 'p1'
   OR project_no = 'p2';
```

Результат выполнения этого запроса:

project_no	emp_no
p1	10102
p2	25348
p2	18316
p2	29346
p1	9031
p1	28559
p2	28559
p1	29346

Результаты выполнения примера 6.5 содержат дубликаты значений столбца `emp_no`. Эту избыточную информацию можно устранить с помощью ключевого слова `DISTINCT`, как показано в следующем примере:

```
USE sample;
SELECT DISTINCT emp_no
   FROM works_on
   WHERE project_no = 'p1'
   OR project_no = 'p2';
```

Результат выполнения модифицированного кода:

emp_no
9031
10102
18316
25348
28559
29346

Обратите внимание, что опцию `DISTINCT` можно использовать только один раз в списке выбора, и она должна предшествовать всем именам столбцов. Поэтому, код в примере 6.6 ошибочен.

Пример 6.6. Неправильное положение опции `DISTINCT`

```
USE sample;
SELECT emp_fname, DISTINCT emp_no
   FROM employee
   WHERE emp_lname = 'Moser';
```

Результатом выполнения этого запроса будет сообщение об ошибке, выданное сервером:

```
Server: Msg 156, Level 15, State 1, Line 1
Incorrect syntax near the keyword 'DISTINCT'.
```

(Неправильный синтаксис при использовании ключевого слова `'DISTINCT'`).

ПРИМЕЧАНИЕ

Если список выбора содержит больше, чем один столбец, то предложение `DISTINCT` выводит все строки с разными комбинациями значений столбцов.

Предложение `WHERE` может содержать любое число одинаковых или разных логических операций. Следует помнить, что логические операторы имеют разный приоритет выполнения: оператор `NOT` имеет самый высший приоритет, далее идет оператор `AND`, а оператор `OR` имеет самый низший приоритет. Неправильное размещение логических операторов может дать непредвиденные результаты, как это показано в примере 6.7.

Пример 6.7. Неправильное размещение логических операторов

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE emp_no = 25348 AND emp_lname = 'Smith'
   OR emp_fname = 'Matthew' AND dept_no = 'd1';

SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE ((emp_no = 25348 AND emp_lname = 'Smith')
   OR emp_fname ='Matthew') AND dept_no = 'd1';
```

Результат выполнения этих запросов:

emp_no	emp_fname	emp_lname
25348	Matthew	Smith

emp_no	emp_fname	emp_lname

Как можно видеть, эти два кажущиеся одинаковыми запроса `SELECT` выдают два разных результирующих набора данных. В первой инструкции `SELECT` система сначала вычисляет оба оператора `AND` (слева направо), а потом вычисляет оператор `OR`. Во второй же инструкции `SELECT` порядок выполнения операторов изменен вследствие использования скобок, операторы в скобках выполняются первыми, в порядке слева направо. Как можно видеть, первая инструкция возвратила одну строку, тогда как вторая не возвратила ни одной.

Наличие логических операторов в предложении WHERE усложняет содержащую его инструкцию SELECT и способствует появлению в ней ошибок. В таких случаях настоятельно рекомендуется применять скобки, даже если они не являются необходимыми. Применение скобок значительно улучшает читаемость инструкции SELECT и уменьшает возможность появления в ней ошибок. Далее приводится первый вариант инструкции SELECT из примера 6.7, модифицированной в соответствии с этой рекомендацией:

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE (emp_no = 25348 AND emp_lname = 'Smith')
   OR (emp_fname = 'Matthew' AND dept_no = 'd1');
```

Логический оператор NOT изменяет логическое значение, к которому он применяется, на противоположное. В таблице истинности в *главе 4* показано, что отрицание истинного значения (TRUE) дает ложь (FALSE) и наоборот. Отрицание значения NULL также дает NULL.

В примере 6.8 демонстрируется использование оператора отрицания NOT.

Пример 6.8. Выборка табельных номеров и имен сотрудников, не принадлежащих к отделу d2

```
USE sample
SELECT emp_no, emp_lname
   FROM employee
   WHERE NOT dept_no = 'd2';
```

Результат выполнения этого запроса:

emp_no	emp_lname
25348	Smith
10102	Jones
18316	Barrimore
28559	Moser

В данном случае логический оператор NOT можно заменить логическим оператором сравнения <> (не равно).

ПРИМЕЧАНИЕ

В этой книге оператор "не равно" обозначается как <>, а не !=, с целью соответствия стандарту ANSI SQL.

Операторы *IN* и *BETWEEN*

Оператор IN позволяет указать одно или несколько выражений, по которым следует выполнять поиск в запросе. Результатом выражения будет истина (TRUE), если значение соответствующего столбца равно одному из условий, указанных в предикате IN. В примере 6.9 демонстрируется использование оператора IN.

Пример 6.9. Выборка всех столбцов сотрудников, чей табельный номер равен 29346, 28559 или 25348

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE emp_no IN (29346, 28559, 25348);
```

Результат выполнения этого запроса будет следующим:

emp_no	emp_fname	emp_lname
25348	Matthew	Smith
29346	James	James
28559	Sybil	Moser

Оператор IN равнозначен последовательности условий, соединенных операторами OR. (Число операторов OR на один меньше, чем количество выражений в списке оператора IN.)

Оператор IN можно использовать совместно с логическим оператором NOT, как показано в примере 6.10. В данном случае запрос выбирает все строки, не содержащие ни одного из указанных значений в соответствующих столбцах.

Пример 6.10. Выборка всех столбцов для сотрудников, чей табельный номер не равен ни 10102, ни 9031

```
USE sample;
SELECT emp_no, emp_fname, emp_lname, dept_no
   FROM employee
   WHERE emp_no NOT IN (10102, 9031);
```

Результат выполнения этого запроса:

emp_no	emp_fname	emp_lname	dept_no
25348	Matthew	Smith	d3
18316	John	Barrimore	d1
29346	James	James	d2
2581	Elke	Hansel	d2
28559	Sybill	Moser	d1

В отличие от оператора IN, для которого указываются отдельные значения, для оператора BETWEEN указывается диапазон значений, чьи границы определяются нижним и верхним значениями. Использование оператора BETWEEN показано в примере 6.11.

Пример 6.11. Выборка наименований проектов и бюджетов всех проектов с бюджетом, находящимся в диапазоне от $95 000 и до $120 000 включительно

```
USE sample;
SELECT project_name, budget
   FROM project
   WHERE budget BETWEEN 95000 AND 120000;
```

Результат выполнения этого запроса:

project_name	budget
Apollo	120000
Gemini	95000

Оператор BETWEEN возвращает все значения в указанном диапазоне, включая значения для границ; т. е. приемлемые значения могут быть между значениями указанных границ диапазона или быть *равными* значениям этих границ.

Оператор BETWEEN логически эквивалентен двум отдельным сравнениям, соединенным логическим оператором AND. Поэтому запрос, приведенный в примере 6.12, эквивалентен запросу примера 6.11.

Пример 6.12. Замена оператора BETWEEN сравнениями, соединенными оператором AND

```
USE   sample;
SELECT project_name, budget
   FROM project
   WHERE budget >= 95000 AND budget <= 120000;
```

Подобно оператору BETWEEN, оператор NOT BETWEEN можно использовать для выборки значений, находящихся за пределами указанного диапазона значений. Оператор BETWEEN также можно применять со значениями, которые имеют символьный или временной тип данных.

В примере 6.13 показаны две разные формы запроса SELECT, которые дают одинаковые результаты.

Пример 6.13. Выборка всех проектов с бюджетом меньшим, чем $100 000 и большим, чем $150 000

```
USE sample;
SELECT project_name
   FROM project
   WHERE budget NOT BETWEEN 100000 AND 150000;
```

Запрос с использованием операторов сравнения выглядит иначе:

```
USE sample;
SELECT project_name
   FROM project
   WHERE budget < 100000 OR budget > 150000;
```

Но, несмотря на разную форму запросов, результат они дают одинаковый:

project_name
Gemini
Mercury

ПРИМЕЧАНИЕ

Формулировка требования запроса: "Выбрать наименования всех проектов с бюджетом меньшим, чем $100 000 и имена всех проектов с бюджетом большим, чем $150 000" может навести на мысль, что во втором запросе SELECT в примере 6.13 требуется применить логический оператор AND. Но логический смысл запроса требует применения оператора OR, т. к. использование оператора AND не даст никаких результатов вообще. Это потому, что не может бюджет быть одновременно и меньшим, чем $100 000 и большим, чем $150 000. Но это и является ответом, почему используется оператор OR, а не AND, поскольку мы выбираем все проекты, бюджет которых меньше $100 000 *или* больше $150 000.

Запросы, связанные со значением *NULL*

Параметр NULL в инструкции CREATE TABLE указывает, что соответствующий столбец может содержать специальное значение NULL (которое обычно представляет неизвестное или неприменимое значение). Значения NULL отличаются от всех других значений базы данных. Предложение WHERE инструкции SELECT обычно возвращает строки, удовлетворяющие указанным в нем условиям сравнения. Но здесь возникает вопрос, как будут оцениваться в этих сравнениях значения NULL?

Все сравнения со значением NULL возвращают FALSE, даже если им предшествует оператор NOT. Для выборки строк, содержащих значения NULL, в языке Transact-SQL применяется оператор IS NULL. Указание в предложении WHERE строк, содержащих (или не содержащих) значение NULL, имеет следующую общую форму:

```
column IS [NOT] NULL
```

Использование оператора IS NULL демонстрируется в примере 6.14.

Пример 6.14. Выборка табельных номеров служащих и соответствующих номеров проектов для служащих, чья должность неизвестна и которые работают над проектом p2

```
USE sample;
SELECT emp_no, project_no
```

```
    FROM works_on
     WHERE project_no = 'p2'
     AND job IS NULL;
```

Результат выполнения этого запроса:

emp_no	project_no
18316	p2
29346	p2

А в примере 6.15 демонстрируется синтаксически правильное, но логически неправильное использование сравнения с NULL. Причиной ошибки является то обстоятельство, что сравнение любого значения, включая NULL, с NULL возвращает FALSE.

Пример 6.15. Неправильное построение проверки значения на NULL

```
USE sample;
SELECT project_no, job
   FROM works_on
   WHERE job <> NULL;
```

Выполнение этого запроса не возвращает никаких строк:

project_no	job

Условие "column IS NOT NULL" эквивалентно условию "NOT (column IS NULL)".

Системная функция ISNULL позволяет отображать указанное значение вместо значения NULL (пример 6.16).

Пример 6.16. Использование системной функции ISNULL

```
USE sample;
SELECT emp_no, ISNULL(job, 'Job unknown') AS task
FROM works_on
WHERE project_no = 'p1';
```

Результат выполнения этого запроса:

emp_no	task
10102	Analyst
9031	Manager
28559	Job unknown
29346	Clerk

В примере 6.16 для столбца должностей job в результате запроса используется заголовок task.

Оператор *LIKE*

Оператор LIKE используется для сопоставления с образцом, т. е. он сравнивает значения столбца с указанным шаблоном. Столбец может быть любого символьного типа данных или типа дата. Общая форма оператора LIKE выглядит таким образом:

```
column [NOT] LIKE 'pattern'
```

Параметр 'pattern' может быть строковой константой, или константой даты, или выражением (включая столбцы таблицы), и должен быть совместимым с типом данных соответствующего столбца. Для указанного столбца сравнение значения строки и шаблона возвращает TRUE, если значение совпадает с выражением шаблона.

Определенные применяемые в шаблоне символы, называющиеся *подстановочными символами* (wildcard characters), имеют специальное значение. Рассмотрим два из этих символов:

- % (знак процента) — обозначает последовательность любых символов любой длины;
- _ (символ подчеркивания) — обозначает любой один символ.

Использование подстановочных символов % и _ показано в примере 6.17.

Пример 6.17. Выборка имен, фамилий и табельных номеров сотрудников, у которых второй буквой имени является буква "a"

```
USE sample;
SELECT emp_fname, emp_lname, emp_no
   FROM employee
   WHERE emp_fname LIKE '_a%';
```

Результат выполнения этого запроса:

emp_fname	emp_lname	emp_no
Matthew	Smith	25348
James	James	29346

Кроме знака процентов и символа подчеркивания, поддерживает другие специальные символы, применяемые с оператором LIKE. Использование этих символов ([,], ^) демонстрируется в примерах 6.18 и 6.19.

Пример 6.18. Выборка всех столбцов отделов, для которых наименование места расположения (location) начинается с символа в диапазоне от "C" до "F"

```
USE sample;
SELECT dept_nt, dept_name, location
   FROM department
   WHERE location LIKE '[C-F]%';
```

Результат выполнения этого запроса:

dept_no	dept_name	location
d1	Research	Dallas
d3	Marketing	Dallas

Как можно видеть по результатам примера 6.18, квадратные скобки `[]` ограничивают диапазон или список символов. Порядок отображения символов диапазона определяется порядком сортировки, указанным при установке системы.

Символ `^` обозначает отрицание диапазона или списка символов. Но такое значение этот символ имеет только тогда, когда находится внутри квадратных скобок, как показано в примере 6.19. В данном примере запроса осуществляется выборка имен и фамилий тех сотрудников, чьи фамилии начинаются с буквы отличной от `J`, `K`, `L`, `M`, `N`, `O` и чьи имена начинаются не с буквы `E` или `Z`.

Пример 6.19. Использование символа отрицания

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE emp_lname LIKE '[^J-O]%1'
   AND emp_fname LIKE '[^EZ]%';
```

Результат выполнения этого запроса:

emp_no	emp_fname	emp_lname
25348	Matthew	Smith
18316	John	Barrimore

Условие "`column NOT LINE 'pattern'`" эквивалентно условию "`NOT (column LIKE 'pattern')`".

В примере 6.20 демонстрируется использование оператора `LIKE` совместно с отрицанием `NOT`.

Пример 6.20. Выборка всех столбцов сотрудников, чьи имена оканчиваются на букву, отличную от буквы "n"

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE emp_fname NOT LIKE '%n';
```

Результат выполнения этого запроса:

emp_no	emp_fname	emp_lname
25348	Matthew	Smith
29346	James	James
2581	Elke	Hansel
9031	Elsa	Bertoni
28559	Sybill	Moser

Любой подстановочный символ (%, _, [,] или ^), заключенный в квадратные скобки, остается обычным символом и представляет сам себя. Такая же возможность существует при использовании параметра ESCAPE. Поэтому оба варианта применения инструкции SELECT, показанные в примере 6.21, эквивалентны.

Пример 6.21. Представление подстановочных символов, как обычных символов

```
USE sample;
SELECT project_no, project_name
   FROM project
   WHERE project_name LIKE '%[_]%';

SELECT project_no, project_name
   FROM project
   WHERE project_name LIKE '%!_%' ESCAPE '!';
```

Результат выполнения этих двух инструкций SELECT будет одинаковым:

project_no	project_name

project_no	project_name

В примере 6.21 обе инструкции SELECT рассматривают символ подчеркивания в значениях столбца project_name, как таковой, а не как подстановочный. В первой инструкции SELECT это достигается заключением символа подчеркивания в квадратные скобки. А во второй инструкции SELECT этот же эффект достигается за счет применения символа перехода (escape character), каковым в данном случае является символ восклицательного знака. Символ перехода переопределяет значение символа подчеркивания, делая его из подстановочного символа обычным. (Результат выполнения этих инструкций содержит ноль строк, потому что ни одно имя проекта не содержит символов подчеркивания.)

ПРИМЕЧАНИЕ

Стандарт SQL поддерживает только подстановочные символы %, _ и оператор ESCAPE. Поэтому если требуется представить подстановочный символ как обычный символ, то вместо квадратных скобок рекомендуется применять оператор ESCAPE.

Предложение *GROUP BY*

Предложение GROUP BY группирует выбранный набор строк для получения набора сводных строк по значениям одного или нескольких столбцов или выражений. Простой случай применения предложения GROUP BY показано в примере 6.22.

Пример 6.22. Выборка и группирование должностей сотрудников

```
USE sample;
SELECT job
   FROM works_on
   GROUP BY job;
```

Результат выполнения этого запроса:

job
NULL
Analyst
Clerk
Manager

В примере 6.22 предложение GROUP BY создает отдельную группу для всех возможных значений (включая значение NULL) столбца job.

ПРИМЕЧАНИЕ

Использование столбцов в предложении GROUP BY должно отвечать определенным условиям. В частности, каждый столбец в списке выборки запроса также должен присутствовать в предложении GROUP BY. Это требование не распространяется на константы и столбцы, являющиеся частью агрегатной функции. (Агрегатные функции рассматриваются в следующем подразделе.) Это имеет смысл, т. к. только для столбцов в предложении GROUP BY гарантируется одно значение для каждой группы.

Таблицу можно сгруппировать по любой комбинации ее столбцов. В примере 6.23 демонстрируется группирование строк таблицы works_on по двум столбцам.

Пример 6.23. Группирование сотрудников по номеру проекта и должности

```
USE sample;
SELECT project_no, job
   FROM works_on
   GROUP BY project_no, job;
```

Результат выполнения этого запроса:

project_no	job
p1	Analyst
p1	Clerk

```
p1              Manager
p1              NULL
p2              NULL
p2              Clerk
p3              Analyst
p3              Clerk
p3              Manager
```

По результатам выполнения запроса примера 6.23 можно видеть, что существует девять групп с разными комбинациями номера проекта и должности. И только две группы, содержащие более одной строки, которыми являются:

```
p2   Clerk   25348, 28559
p2   NULL    18316, 29346
```

Последовательность имен столбцов в предложении `GROUP BY` не обязательно должна быть такой же, как и в списке столбцов выборки `SELECT`.

Агрегатные функции

Агрегатные функции используются для получения суммарных значений. Все агрегатные функции можно разделить на следующие категории:

- обычные агрегатные функции;
- статистические агрегатные функции;
- агрегатные функции, определяемые пользователем;
- аналитические агрегатные функции.

Первые три типа агрегатных функций рассматриваются в последующих разделах, а аналитические функции — в *главе 23*.

Обычные агрегатные функции

Язык Transact-SQL поддерживает следующие шесть агрегатных функций:

- `MIN`
- `MAX`
- `SUM`
- `AVG`
- `COUNT`
- `COUNT_BIG`

Все агрегатные функции выполняют вычисления над одним аргументом, который может быть или столбцом, или выражением. (Единственным исключением является вторая форма двух функций: `COUNT` и `COUNT_BIG`, а именно `COUNT(*)` и `COUNT_BIG(*)` соответственно.) Результатом вычислений любой агрегатной функции является константное значение, отображаемое в отдельном столбце результата.

Агрегатные функции указываются в списке столбцов инструкции `SELECT`, который также может содержать предложение `GROUP BY`. Если в инструкции `SELECT` отсутст-

вует предложение GROUP BY, а список столбцов выборки содержит, по крайней мере, одну агрегатную функцию, тогда он не должен содержать простых столбцов (кроме как столбцов, служащих аргументами агрегатной функции). Поэтому код в примере 6.24 неправильный.

Пример 6.24. Неправильный синтаксис использования агрегатной функции

```
USE sample;
SELECT emp_lname, MIN(emp_no)
   FROM employee;
```

В примере 6.24 столбец emp_lname таблицы employee не должен быть в списке выборки столбцов, поскольку он не является аргументом агрегатной функции. С другой стороны, список выборки столбцов может содержать имена столбцов, которые не являются аргументами агрегатной функции, если эти столбцы служат аргументами предложения GROUP BY.

Аргументу агрегатной функции может предшествовать одно из двух возможных ключевых слов:

- ALL — указывает, что вычисления выполняются над всеми значениями столбца. Значение по умолчанию;
- DISTINCT — указывает, что для вычислений применяются только уникальные значения столбца.

Агрегатные функции *MIN* и *MAX*

Агрегатные функции MIN и MAX вычисляют наименьшее и наибольшее значение столбца соответственно. Если запрос содержит предложение WHERE, функции MIN и MAX возвращают наименьшее и наибольшее значение строк, отвечающих указанным условиям. В примере 6.25 показано использование агрегатной функции MIN.

Пример 6.25. Определение наименьшего значения табельного номера

```
USE sample;
SELECT MIN(emp_no) AS min_employee_no
   FROM employee;
```

Результат выполнения запроса:

min_employee_no
2581

Возвращенный в примере 6.25 результат не очень информативный. Например, неизвестно имя сотрудника, которому принадлежит этот номер. Но получить это имя обычным способом невозможно, потому что, как упоминалось ранее, явно указать столбец emp_name не разрешается. Для того чтобы вместе с наименьшим табельным

номером сотрудника также получить и имя этого сотрудника, используется подзапрос. В примере 6.26 показано использование такого подзапроса, где вложенный запрос содержит инструкцию SELECT из предыдущего примера.

Пример 6.26. Выборка наименьшего табельного номера сотрудника и соответствующего имени сотрудника

```
USE sample;
SELECT emp_no, emp_lname
   FROM employee
   WHERE emp_no =
   SELECT MIN(emp_no)
      FROM employee);
```

Результат выполнения запроса:

emp_no	emp_lname
2581	Hansel

Использование агрегатной функции MAX показано в примере 6.27.

Пример 6.27. Выборка табельного номера менеджера, введенного последним в таблицу works_on

```
USE sample;
SELECT emp_no
   FROM works_on
   WHERE enter_date =
   (SELECT MAX(enter_date)
      FROM works_on
      WHERE job = 'Manager');
```

Результат выполнения запроса:

emp_no
10102

В качестве аргумента функции MIN и MAX также могут принимать строки и даты. В случае строкового аргумента значения сравниваются, используя фактический порядок сортировки. Для всех аргументов временных данных типа "дата" наименьшим значением столбца будет наиболее ранняя дата, а наибольшим — наиболее поздняя.

С функциями MIN и MAX можно применять ключевое слово DISTINCT. Перед применением агрегатных функций MIN и MAX из столбцов их аргументов исключаются все значения NULL.

Агрегатная функция *SUM*

Агрегатная функция SUM вычисляет общую сумму значений столбца. Аргумент этой агрегатной функции всегда должен иметь числовой тип данных. Использование агрегатной функции SUM показано в примере 6.28.

Пример 6.28. Вычисление общей суммы бюджетов всех проектов

```
USE sample;
SELECT SUM(budget) sum_of_budgets
   FROM project;
```

Результат выполнения запроса:

```
sum_of_budgets
--------------
401500
```

В примере 6.28 агрегатная функция группирует все значения бюджетов проектов и определяет их общую сумму. По этой причине запрос в примере 6.28 (как и все аналогичные запросы) содержит неявную функцию группирования. Неявную функцию группирования из примера 6.28 можно указать явно, как это показано в примере 6.29.

Пример 6.29. Явное указание группирующей функции в запросе с агрегатной функцией SUM

```
SELECT SUM(budget) sum_of_budgets
FROM project
   GROUP BY();
```

Рекомендуется использовать этот синтаксис в предложении GROUP BY, поскольку таким образом группирование определяется явно. (В *главе 23* рассматриваются несколько других вариантов предложения GROUP BY.)

Использование параметра DISTINCT устраняет все повторяющиеся значения в столбце перед применением функции SUM. Аналогично удаляются все значения NULL перед применением этой агрегатной функции.

Агрегатная функция *AVG*

Агрегатная функция AVG возвращает среднее значение для всех значений столбца. Аргумент этой агрегатной функции всегда должен иметь числовой тип данных. Перед применением функции SUM все значения NULL удаляются из ее аргумента. Использование агрегатной функции AVG показано в примере 6.30.

Пример 6.30. Вычисление среднего значения бюджета для всех бюджетов, превышающих $100 000

```
USE sample;
SELECT AVG(budget) avg_budget
   FROM project
   WHERE budget > 100000;
```

Результат выполнения запроса:

avg_budget
153250

Агрегатные функции *COUNT* и *COUNT_BIG*

Агрегатная функция COUNT имеет две разные формы:

COUNT([DISTINCT] *col_name*)
COUNT(*)

Первая форма функции подсчитывает количество значений в столбце *col_name*. Если в запросе используется ключевое слово DISTINCT, перед применением функции COUNT удаляются все повторяющиеся значения столбца. При подсчете количества значений столбца эта форма функции COUNT не принимает во внимание значения NULL.

Использование первой формы агрегатной функции COUNT показано в примере 6.31.

Пример 6.31. Подсчет количества разных должностей для каждого проекта

```
USE sample;
SELECT project_no, COUNT(DISTINCT job) job_count
   FROM works_on
   GROUP BY project_no;
```

Результат выполнения этого запроса:

project_no	job_count
p1	3
p2	1
p3	3

Как можно видеть в результате выполнения запроса, представленного в примере 6.31, значения NULL функцией COUNT не принимались во внимание. (Сумма всех значений столбца должностей получилась равной 7, а не 11, как должно быть.)

Вторая форма функции COUNT, т. е. функция COUNT(*) подсчитывает количество строк в таблице. А если инструкция SELECT запроса с функцией COUNT(*) содержит предложение WHERE с условием, функция возвращает количество строк, удовлетво-

ряющих указанному условию. В отличие от первого варианта функции COUNT вторая форма не игнорирует значения NULL, поскольку эта функция оперирует строками, а не столбцами. В примере 6.32 демонстрируется использование функции COUNT(*).

Пример 6.32. Подсчет количества должностей во всех проектах

```
USE sample;
SELECT job, COUNT(*) job_count
   FROM works_on
   GROUP   BY   job;
```

Результат выполнения запроса:

Job	job_count
NULL	3
Analyst	2
Clerk	4
Manager	2

Функция COUNT_BIG аналогична функции COUNT. Единственное различие между ними заключается в типе возвращаемого ими результата: функция COUNT_BIG всегда возвращает значения типа BIGINT, тогда как функция COUNT возвращает значения данных типа INTEGER.

Статистические агрегатные функции

Следующие функции составляют группу статистических агрегатных функций:

- VAR — вычисляет статистическую дисперсию всех значений, представленных в столбце или выражении;
- VARP — вычисляет статистическую дисперсию совокупности всех значений, представленных в столбце или выражении;
- STDEV — вычисляет среднеквадратическое отклонение (который рассчитывается как квадратный корень из соответствующей дисперсии) всех значений столбца или выражения;
- STDEVP — вычисляет среднеквадратическое отклонение совокупности всех значений столбца или выражения.

Примеры использования статистических агрегатных функций рассмотрены в *главе 23*.

Агрегатные функции, определяемые пользователем

Компонент Database Engine также поддерживает реализацию функций, определяемых пользователем. Эта возможность позволяет пользователям дополнить системные агрегатные функции функциями, которые они могут реализовывать и устанав-

ливать самостоятельно. Эти функции представляют специальный класс определяемых пользователем функций и подробно рассматриваются в *главе 8*.

Предложение *HAVING*

В предложении HAVING определяется условие, которое применяется к группе строк. Таким образом, это предложение имеет такой же смысл для групп строк, что и предложение WHERE для содержимого соответствующей таблицы. Синтаксис предложения HAVING следующий:

```
HAVING condition
```

Здесь параметр *condition* содержит агрегатные функции или константы.

Использование предложения HAVING совместно с агрегатной функцией COUNT(*) показано в примере 6.33.

Пример 6.33. Выборка номеров проектов, в которых участвует меньше чем четыре сотрудника

```
USE sample;
SELECT project_no
   FROM works_on
   GROUP BY project_no
   HAVING COUNT(*) < 4;
```

Результат выполнения этого запроса:

Project_no
P3

В примере 6.33 система посредством предложения GROUP BY группирует все строки по значениям столбца project_no. После этого подсчитывается количество строк в каждой группе и выбираются группы, содержащие менее четырех строк (три или меньше).

Предложение HAVING можно также использовать без агрегатных функций, как это показано в примере 6.34.

Пример 6.34. Группирование строк таблицы works_on по должности и устранение тех должностей, которые не начинаются с буквы "M"

```
USE sample;
SELECT job
   FROM works_on
   GROUP BY job
   HAVING job LIKE 'M%';
```

Результат выполнения этого запроса:

```
Job
---------------------------------------
Manager
```

Предложение HAVING можно также использовать без предложения GROUP BY, хотя это не является распространенной практикой. В таком случае все строки таблицы возвращаются в одной группе.

Предложение *ORDER BY*

Предложение ORDER BY определяет порядок сортировки строк результирующего набора, возвращаемого запросом. Это предложение имеет следующий синтаксис:

```
ORDER BY {[col_name | col_number [ASC | DESC]]} , ...
```

Порядок сортировки задается в параметре *col_name*. Параметр *col_number* является альтернативным указателем порядка сортировки, который определяет столбцы по порядку их вхождения в список выборки инструкции SELECT (1 — первый столбец, 2 — второй столбец и т. д.). Параметр ASC определяет сортировку в восходящем порядке, а параметр DESC — в нисходящем. По умолчанию применяется параметр ASC.

ПРИМЕЧАНИЕ

Имена столбцов в предложении ORDER BY не обязательно должны быть указаны в списке столбцов выборки. Но это не относится к запросам типа SELECT DISTINCT, т. к. в таких запросах имена столбцов, указанные в предложении ORDER BY, также должны быть указаны в списке столбцов выборки. Кроме этого, это предложение не может содержать имен столбцов из таблиц, не указанных в предложении FROM.

Как можно видеть по синтаксису предложения ORDER BY, сортировка результирующего набора может выполняться по нескольким столбцам. Такая сортировка показана в примере 6.35.

Пример 6.35. Выборка номеров отделов и фамилий и имен сотрудников для сотрудников, чей табельный номер меньше чем 20 000, а так же с сортировкой по фамилии и имени

```
USE sample;
SELECT emp_fname, emp_lname, dept_no
   FROM employee
   WHERE emp_no < 20000
   ORDER BY emp_lname, emp_fname;
```

Результат выполнения этого запроса:

Emp_fname	emp_lname	dept_no
John	Barrimore	d1
Elsa	Bertoni	d2
Elke	Hansel	d2
Ann	Jones	d3

Столбцы в предложении ORDER BY можно указывать не по их именам, а по порядку в списке выборки. Соответственно, предложение в примере 6.35 можно переписать таким образом:

```
ORDER BY 2,1
```

Такой альтернативный способ указания столбцов по их позиции вместо имен применяется, если критерий упорядочивания содержит агрегатную функцию. (Другим способом является использование наименований столбцов, которые тогда отображаются в предложении ORDER BY.) Однако в предложении ORDER BY рекомендуется указывать столбцы по их именам, а не по номерам, чтобы упростить обновление запроса, если в списке выборки придется добавить или удалить столбцы. Указание столбцов в предложении ORDER BY по их номерам показано в примере 6.36.

Пример 6.36. Для каждого номера проекта выбрать номер проекта и количество участвующих в нем сотрудников, упорядочив результат в убывающем порядке по числу сотрудников

```
USE sample;
SELECT project_no, COUNT(*) emp_quantity
   FROM works_on
   GROUP BY project_no
   ORDER BY 2 DESC
```

Результат выполнения этого запроса:

project_no	emp_quantity
P1	4
p2	4
p3	3

Язык Transact-SQL при сортировке в возрастающем порядке помещает значения NULL в начале списка, и в конце списка — при убывающем.

Использование предложения *ORDER BY* для разбиения результатов на страницы

Отображение результатов запроса на текущей странице можно или реализовать в пользовательском приложении, или же дать указание осуществить это серверу базы данных. В первом случае все строки базы данных отправляются приложению,

чьей задачей является отобрать требуемые строки и отобразить их. Во втором случае, со стороны сервера выбираются и отображаются только строки, требуемые для текущей страницы. Как можно предположить, создание страниц на стороне сервера обычно обеспечивает лучшую производительность, т. к. клиенту отправляются только строки, необходимые для отображения.

Для поддержки создания страниц на стороне сервера в SQL Server 2012 вводится два новых предложения инструкции SELECT: OFFSET и FETCH. Применение этих двух предложений демонстрируется в примере 6.37. Здесь из базы данных AdventureWorks2012 извлекается идентификатор бизнеса, название должности и день рождения всех сотрудников женского пола с сортировкой результата по названию должности в возрастающем порядке. Результирующий набор строк разбивается на 10-строчные страницы и отображается третья страница.

Пример 6.37. Применение предложений OFFEST и FETCH

```
USE AdventureWorks2012;
SELECT BusinessEntityID, JobTitle, BirthDate
FROM HumanResources.Employee
WHERE Gender = 'F'
ORDER BY JobTitle
OFFSET 20 ROWS
FETCH NEXT 10 ROWS ONLY;
```

ПРИМЕЧАНИЕ

Другие примеры использования предложения OFFSET приводятся в *главе 23* (см. примеры 23.24 и 23.25).

В предложении OFFSET указывается количество строк результата, которые нужно пропустить в отображаемом результате. Это количество вычисляется после сортировки строк предложением ORDER BY. В предложении FETCH NEXT указывается количество удовлетворяющих условию WHERE и отсортированных строк, которое нужно возвратить. Параметром этого предложения может быть константа, выражение или результат другого запроса. Предложение FETCH NEXT аналогично предложению FETCH FIRST.

Основной целью при создании страниц на стороне сервера является возможность реализация общих страничных форм, используя переменные. Эту задачу можно выполнить посредством пакета SQL Server. Соответствующий пример приводится в *главе 8* (см. пример 8.5).

Инструкция *SELECT* и свойство *IDENTITY*

Свойство IDENTITY позволяет определить значения для конкретного столбца таблицы в виде автоматически возрастающего счетчика. Это свойство могут иметь столбцы численного типа данных, такого как TINYINT, SMALLINT, INT и BIGINT. Для

такого столбца таблицы компонент Database Engine автоматически создает последовательные значения, начиная с указанного стартового значения. Таким образом, свойство IDENTITY можно использовать для создания однозначных числовых значений для выбранного столбца.

Таблица может содержать только один столбец со свойством IDENTITY. Владелец таблицы имеет возможность указать начальное значение и шаг приращения, как это показано в примере 6.38.

Пример 6.38. Использование свойства IDENTITY

```
USE sample;
CREATE TABLE product
   (product_no INTEGER IDENTITY(10000, 1) NOT NULL,
    product_name CHAR(30) NOT NULL,
    price MONEY);
SELECT $identity
   FROM product
   WHERE product_name = 'Soap';
```

Результатом этого запроса может быть следующее:

product_no
10005

В примере 6.38 сначала создается таблица product, содержащая столбец product_no со свойством IDENTITY. Значения в столбце product_no создаются автоматически системой, начиная с 10 000 и увеличиваясь с единичным шагом для каждого последующего значения: 10 000, 10 001, 10 002 и т. д.

Со свойством IDENTITY связаны некоторые системные функции и переменные. Например, в коде примера 6.38 используется переменная $identity. Как можно видеть по результатам выполнения этого кода, эта переменная автоматически ссылается на свойство IDENTITY.

Начальное значение и шаг приращения столбца со свойством IDENTITY можно узнать с помощью функций IDENT_SEED и IDENT_INCR соответственно. Применяются эти функции следующим образом:

```
SELECT IDENT_SEED('product'), IDENT_INCR('product)'
```

Как уже упоминалось, значения IDENTITY устанавливаются автоматически системой. Но пользователь может указать явно свои значения для определенных строк, присвоив параметру IDENTITY_INSERT значение ON перед вставкой явного значения:

```
SET IDENTITY_INSERT table_name ON
```

ПРИМЕЧАНИЕ

Поскольку с помощью параметра IDENTITY_INSERT для столбца со свойством IDENTITY можно установить любое значение, в том числе и повторяющееся, свойство IDENTITY обычно не обеспечивает принудительную уникальность значений столбца. Поэтому для принудительного обеспечения уникальности значений столбца следует применять ограничения UNIQUE или PRIMARY KEY.

При вставке значений в таблицу после присвоения параметру IDENTITY_INSERT значения ON система создает следующее значение столбца IDENTITY, увеличивая наибольшее текущее значение этого столбца.

Оператор *CREATE SEQUENCE*

Применение свойства IDENTITY имеет несколько значительных недостатков, наиболее существенными из которых являются следующие:

- применение свойства ограничивается указанной таблицей;
- новое значение столбца нельзя получить иным способом, кроме как применив его;
- свойство IDENTITY можно указать только при создании столбца.

По этим причинам в SQL Server 2012 вводятся последовательности, которые обладают той же семантикой, что и свойство IDENTITY, но при этом не имеют ранее перечисленных недостатков. В данном контексте *последовательностью* называется функциональность базы данных, позволяющая указывать значения счетчика для разных объектов базы данных, таких как столбцы и переменные.

Последовательности создаются с помощью инструкции CREATE SEQUENCE. Инструкция CREATE SEQUENCE определена в стандарте SQL и поддерживается другими реляционными системами баз данных, такими как IBM DB2 и Oracle.

В пример 6.39 показано создание последовательности в SQL Server.

Пример 6.39. Использование оператора CREATE SEQUENCE

```
USE sample;
CREATE SEQUENCE dbo.Sequence1
   AS INT
   START WITH 1 INCREMENT BY 5
   MINVALUE 1 MAXVALUE 256
   CYCLE;
```

В примере 6.39 значения последовательности Sequence1 создаются автоматически системой, начиная со значения 1 с шагом 5 для каждого последующего значения. Таким образом, в предложении START указывается начальное значение, а в предложении INCREMENT — шаг. (Шаг может быть как положительным, так и отрицательным.)

В следующих двух, необязательных, предложениях MINVALUE и MAXVALUE указываются минимальное и максимальное значение объекта последовательности. (Обратите внимание, что значение MINVALUE должно быть меньшим или равным начальному значению, а значение MAXVALUE не может быть большим, чем верхний предел типа данных, указанных для последовательности.) В предложении CYCLE указывается, что последовательность повторяется с начала по превышению максимального (или минимального для последовательности с отрицательным шагом) значения. По умолчанию это предложение имеет значение NO CYCLE, что означает, что превышение максимального или минимального значения последовательности вызывает исключение.

Основной особенностью последовательностей является их независимость от таблиц, т. е. их можно использовать с любыми объектами базы данных, такими как столбцы таблицы или переменные. (Это свойство положительно влияет на хранение и, соответственно, на производительность. Определенную последовательность хранить не требуется; сохраняется только ее последнее значение.)

Новые значения последовательности создаются с помощью выражения NEXT VALUE FOR, применение которого показано в примере 6.40.

Пример 6.40. Создание новых значений последовательности

```
USE sample;
SELECT NEXT VALUE FOR dbo.Sequence1;
SELECT NEXT VALUE FOR dbo.Sequence1;
```

Результат выполнения этого запроса:

```
1
6
```

С помощью выражения NEXT VALUE FOR можно присвоить результат последовательности переменной или ячейке столбца. В примере 6.41 показано использование этого выражения для присвоения результатов столбцу.

Пример 6.41. Использование выражения NEXT VALUE FOR

```
USE sample;
CREATE TABLE dbo.table1
   (column1 INT NOT NULL PRIMARY KEY,
    column2 CHAR(10));
   INSERT INTO dbo.table1 VALUES (NEXT VALUE FOR dbo.sequence1, 'A');
   INSERT INTO dbo.table1 VALUES (NEXT VALUE FOR dbo.sequence1, 'B');
```

В примере 6.41 сначала создается таблица table1, состоящая из двух столбцов. Далее, две инструкции INSERT вставляют в эту таблицу две строки. (Синтаксис инструкции INSERT см. в *главе 7*.) Первые две ячейки первого столбца будут иметь

значения 11 и 16. (Эти два значения соответствуют значениям, созданным в примере 6.40.)

В примере 6.42 показано использование представления каталога sys.sequences для просмотра текущего значения последовательности, не используя его. (Представления просмотра каталога подробно рассматриваются в *главе 9*.)

Пример 6.42. Просмотр текущего значения последовательности

```
USE sample;
SELECT current_value
   FROM sys.sequences WHERE name = 'Sequence1';
```

ПРИМЕЧАНИЕ

Обычно выражение NEXT VALUE FOR применяется в инструкции INSERT (см. *главу 7*), чтобы система вставляла созданные значения. Это выражение также можно использовать, как часть многострочного запроса с помощью предложения OVER (см. пример 23.8 в *главе 23*).

Для изменения свойства существующей последовательности применяется инструкция ALTER SEQUENCE. Одно из наиболее важных применений этой инструкции связано с параметром RESTART WITH, который переустанавливает указанную последовательность. В примере 6.43 показано использование инструкции ALTER SEQUENCE для переустановки почти всех свойств последовательности Sequence1.

Пример 6.43. Переустановка свойств последовательности

```
USE sample;
ALTER SEQUENCE dbo.sequence1
   RESTART WITH 100
   INCREMENT BY 50
   MINVALUE 50
   MAXVALUE 200
   NO CYCLE;
```

Удаляется последовательность с помощью инструкции DROP SEQUENCE.

Операторы работы с наборами

Кроме операторов, рассмотренных ранее в этой главе, язык Transact-SQL поддерживает еще три оператора работы с наборами:

- UNION;
- INTERSECT;
- EXCEPT.

Оператор *UNION*

Оператор объединяет результаты двух или более запросов в один результирующий набор, в который входят все строки, принадлежащие всем запросам в объединении. Соответственно, результатом объединения двух таблиц является новая таблица, содержащая все строки, входящие в одну из исходных таблиц или в обе эти таблицы.

Общая форма оператора UNION выглядит таким образом:

select_1 UNION [ALL] *select_2* {[UNION [ALL] *select_3*]}...

Параметры *select_1*, *select_2*, ... представляют собой инструкции SELECT, которые создают объединение. Если используется параметр ALL, отображаются все строки, включая дубликаты. В операторе UNION параметр ALL имеет то же самое значение, что и в списке выбора SELECT, но с одним отличием: для списка выбора SELECT этот параметр применяется по умолчанию, а для оператора UNION его нужно указывать явно.

В своей исходной форме база данных sample не подходит для демонстрации применения оператора UNION. Поэтому в этом разделе создается новая таблица employee_enh, которая идентична существующей таблице employee, но имеет дополнительный столбец domicile. В этом столбце указывается место жительства сотрудников.

Новая таблица employee_enh выглядит следующим образом:

emp_no	emp_fname	emp_lname	dept_no	domicile
25348	Matthew	Smith	d3	San Antonio
10102	Ann	Jones	d3	Houston
18316	John	Barrimore	d1	San Antonio
29346	James	James	d2	Seattle
9031	Elke	Hansel	d2	Portland
2581	Elsa	Bertoni	d2	Tacoma
28559	Sybil]	Moser	dl	Houston

Создание таблицы employee_enh предоставляет нам удобный случай продемонстрировать использование предложения INTO в инструкции SELECT. Инструкция SELECT INTO выполняет две операции. Сначала создается новая таблица со столбцами, перечисленными в списке выбора SELECT. Потом строки исходной таблицы вставляются в новую таблицу. Имя новой таблицы указывается в предложении INTO, а имя таблицы-источника указывается в предложении FROM.

В примере 6.44 показано создание таблицы employee_enh.

Пример 6.44. Применение инструкции SELECT INTO для создания новой таблицы

```
USE sample;
SELECT emp_no, emp_fname, emp_lname, dept_no
   INTO  employee_enh
   FROM employee;
ALTER TABLE employee_enh
   ADD domicile CHAR(25) NULL;
```

В примере 6.44 инструкция SELECT INTO создает таблицу employee_enh, вставляет в нее все строки из таблицы-источника employee, после чего инструкция ALTER TABLE добавляет в новую таблицу столбец domicile.

Но добавленный столбец domicile не содержит никаких значений. Значения в этот столбец можно вставить посредством среды Management Studio (см. *главу 3*) или же с помощью следующего кода:

```
USE sample;
UPDATE employee_enh SET domicile = 'San Antonio'
   WHERE emp_no = 25348;
UPDATE employee_enh SET domicile = 'Houston'
   WHERE emp_no = 10102;
UPDATE employee_enh SET domicile = 'San Antonio'
   WHERE emp_no = 18316;
UPDATE employee_enh SET domicile = 'Seattle'
   WHERE emp_no = 29346;
UPDATE employee_enh SET domicile = 'Portland'
   WHERE emp_no = 9031;
UPDATE employee_enh SET domicile = 'Tacoma'
   WHERE emp_no = 2581;
UPDATE employee_enh SET domicile = 'Houston'
   WHERE emp_no = 28559;
```

Теперь мы готовы продемонстрировать использование инструкции UNION. В примере 6.45 показан запрос для создания соединения таблиц employee_enh и department, используя эту инструкцию.

Пример 6.45. Объединение таблиц с помощью инструкции UNION

```
USE sample;
SELECT domicile
   FROM employee_enh
UNION
SELECT location
   FROM department;
```

Результат выполнения этого запроса:

Domicile
San Antonio
Houston
Portland
Tacoma
Seattle
Dallas

Объединять с помощью инструкции UNION можно только совместимые таблицы. Под совместимыми таблицами имеется в виду, что оба списка столбцов выборки должны содержать одинаковое число столбцов, а соответствующие столбцы должны иметь совместимые типы данных. (В отношении совместимости типы данных INT и SMALLINT не являются совместимыми.)

Результат объединения можно упорядочить, только используя предложение ORDER BY в последней инструкции SELECT, как это показано в примере 6.46. Предложения GROUP BY и HAVING можно применять с отдельными инструкциями SELECT, но не в самом объединении.

Пример 6.46. Упорядочивание результатов объединения двух таблиц

```
USE sample;
SELECT emp_no
   FROM employee
   WHERE dept_no = 'd1'
UNION
SELECT emp_no
   FROM works_on
   WHERE enter_date < '01.01.2007'
ORDER BY 1;
```

Запрос в примере 6.46 осуществляет выборку сотрудников, которые или работают в отделе d1, или начали работать над проектом до 1 января 2007 г. Этот запрос возвращает следующий результат:

emp_no
9031
10102
18316
28559
29346

ПРИМЕЧАНИЕ

Оператор UNION поддерживает параметр ALL. При использовании этого параметра дубликаты не удаляются из результирующего набора.

Вместо оператора UNION можно применить оператор OR, если все инструкции SELECT, соединенные одним или несколькими операторами UNION, ссылаются на одну и ту же таблицу. В таком случае набор инструкций SELECT заменяется одной инструкцией SELECT с набором операторов OR.

Операторы *INTERSECT* и *EXCEPT*

Два других оператора для работы с наборами, INTERSECT и EXCEPT, определяют пересечение и разность соответственно. Под пересечением в данном контексте имеется набор строк, которые принадлежат к обеим таблицам. А разность двух таблиц определяется как все значения, которые принадлежат к первой таблице и не присутствуют во второй. В примере 6.47 показано использование оператора INTERSECT.

Пример 6.47. Применение оператора INTERSECT

```
USE sample;
SELECT emp_no
   FROM employee
   WHERE dept_no = 'd1'
INTERSECT
SELECT emp_no
   FROM works_on
   WHERE enter_date < '01.01.2008';
```

Результат выполнения этого запроса:

emp_no
18316
28559

ПРИМЕЧАНИЕ

Язык Transact-SQL не поддерживает использование параметра ALL ни с оператором INTERSECT, ни с оператором EXCEPT.

Использование оператора EXCEPT показано в примере 6.48.

Пример 6.48. Применение оператора EXCEPT

```
USE sample;
SELECT emp_no
   FROM employee
   WHERE dept_no = 'd3'
EXCEPT
```

```
SELECT emp_no
   FROM works_on
   WHERE enter_date > '01.01.2008';
```

Результат выполнения этого запроса:

emp_no
10102
25348

ПРИМЕЧАНИЕ

Следует помнить, что эти три оператора над множествами имеют разный приоритет выполнения: оператор INTERSECT имеет наивысший приоритет, за ним следует оператор EXCEPT, а оператор UNION имеет самый низкий приоритет. Невнимательность к приоритету выполнения при использовании нескольких разных операторов для работы с наборами может повлечь неожиданные результаты.

Выражения *CASE*

В области прикладного программирования баз данных иногда требуется модифицировать представление данных. Например, людей можно подразделить, закодировав их по их социальной принадлежности, используя значения 1, 2 и 3, обозначив так мужчин, женщин и детей соответственно. Такой прием программирования может уменьшить время, необходимое для реализации программы. Выражение CASE языка Transact-SQL позволяет с легкостью реализовать такой тип кодировки.

ПРИМЕЧАНИЕ

В отличие от большинства языков программирования, CASE не является инструкцией, а выражением. Поэтому выражение CASE можно использовать почти везде, где язык Transact-SQL позволяет применять выражения.

Выражение CASE имеет две формы:

- простое выражение CASE;
- поисковое выражение CASE.

Синтаксис простого выражения CASE следующий:

```
CASE expression_1
   {WHEN expression_2 THEN result_1}...
     [ELSE result_n]
END
```

Инструкция с простым выражением CASE сначала ищет в списке всех выражений в предложении WHEN первое выражение, совпадающее с выражением *expression_1*, после чего выполняет соответствующее предложение THEN. В случае отсутствия в списке WHEN совпадающего выражения, выполняется предложение ELSE.

Синтаксис поискового выражения CASE следующий:

```
CASE
   {WHEN condition_1 THEN result_1} ...
      [ELSE result_n]
END
```

В данном случае выполняется поиск первого отвечающего требованиям условия, после чего выполняется соответствующее предложение THEN. Если ни одно из условий не отвечает требованиям, выполняется предложение ELSE. Применение поискового выражения CASE показано в примере 6.49.

Пример 6.49. Применение поискового выражения CASE

```
USE sample;
SELECT project_name,
   CASE
      WHEN budget > 0 AND budget < 100000 THEN 1
      WHEN budget >= 100000 AND budget < 200000 THEN 2
      WHEN budget >= 200000 AND budget < 300000 THEN 3
      ELSE 4
   END budget_weight
FROM project;
```

Результат выполнения этого запроса:

project_name	budget_weight
Apollo	2
Gemini	1
Mercury	2

В примере 6.49 взвешиваются бюджеты всех проектов, после чего отображаются вычисленные их весовые коэффициенты вместе с соответствующими наименованиями проектов.

В примере 6.50 показан другой способ применения выражения CASE, где предложение WHEN содержит вложенные запросы, составляющие часть выражения.

Пример 6.50. Применение выражения CASE с вложенными запросами

```
USE sample;
SELECT project_name,
   CASE
      WHEN p1.budget < (SELECT AVG(p2.budget) FROM project p2)
         THEN 'below average'
      WHEN p1.budget = (SELECT AVG(p2.budget) FROM project p2)
         THEN 'on average'
```

```
        WHEN p1.budget > (SELECT AVG(p2.budget) FROM project p2)
           THEN 'above average'
     END budget_category
  FROM project p1;
```

Результат выполнения этого запроса следующий (где `below average` обозначает "ниже среднего", а `above average` — "выше среднего"):

project_name	budget_category
Apollo	below average
Gemini	below average
Mercury	above average

Подзапросы

Во всех рассмотренных ранее в этой главе примерах значения столбцов сравниваются с выражением, константой или набором констант. Кроме таких возможностей сравнения язык Transact-SQL позволяет сравнивать значения столбца с результатом другой инструкции `SELECT`. Такая конструкция, где предложение `WHERE` инструкции `SELECT` содержит одну или больше вложенных инструкций `SELECT`, называется *подзапросом* (subquery). Первая инструкция `SELECT` подзапроса называется *внешним запросом* (outer query), а внутренняя инструкция (или инструкции) `SELECT`, используемая в сравнении, называется *вложенным запросом* (inner query). Первым выполняется вложенный запрос, а его результат передается внешнему запросу.

ПРИМЕЧАНИЕ

Вложенные запросы также могут содержать инструкции `INSERT`, `UPDATE` и `DELETE`, которые рассматриваются далее в этой книге.

Существует два типа подзапросов:

- независимые;
- связанные.

В независимых подзапросах вложенный запрос логически выполняется ровно один раз. Связанный запрос отличается от независимого тем, что его значение зависит от переменной, получаемой от внешнего запроса. Таким образом, вложенный запрос связанного подзапроса выполняется каждый раз, когда система получает новую строку от внешнего запроса. В этом разделе приводится несколько примеров независимых подзапросов. Связанные подзапросы рассматриваются далее в этой главе совместно с оператором соединения.

Независимый подзапрос может применяться со следующими операторами:

- операторами сравнения;
- оператором `IN`;
- операторами `ANY` и `ALL`;

Подзапросы и операторы сравнения

Использование оператора равенства (=) в независимом подзапросе показано в примере 6.51.

Пример 6.51. Выборка имен и фамилий сотрудников отдела `Research`

```
USE sample;
SELECT emp_fname, emp_lname
   FROM employee
   WHERE dept_no =
      SELECT dept_no
         FROM department
         WHERE dept_name = 'Research');
```

Результат выполнения этого запроса:

emp_fname	emp_lname
John	Barrimore
Sybill	Moser

В примере 6.51 сначала выполняется вложенный запрос, возвращая номер отдела разработки (`d1`). После выполнения внутреннего запроса подзапрос в примере 6.51 можно представить следующим эквивалентным запросом:

```
USE sample
SELECT emp_fname, emp_lname
   FROM employee
   WHERE dept_no = 'd1';
```

В подзапросах можно также использовать любые другие операторы сравнения, при условии, что вложенный запрос возвращает в результате одну строку. Это очевидно, поскольку невозможно сравнить конкретные значения столбца, возвращаемые внешним запросом, с набором значений, возвращаемым вложенным запросом. В последующем разделе рассматривается, как можно решить проблему, когда результат вложенного запроса содержит набор значений.

Подзапросы и оператор *IN*

Оператор `IN` позволяет определить набор выражений (или констант), которые затем можно использовать в поисковом запросе. Этот оператор можно использовать в подзапросах при таких же обстоятельствах, т. е. когда вложенный запрос возвращает набор значений. Использование оператора `IN` в подзапросе показано в примере 6.52.

Пример 6.52. Получение всей информации о сотрудниках, чей отдел находится в Далласе

```
USE sample;
SELECT *
   FROM employee
   WHERE dept_no IN
   (SELECT dept_no
      FROM department
      WHERE location = 'Dallas');
```

Результат выполнения этого запроса:

emp_no	emp_fname	emp_lname	dept_no
25348	Matthew	Smith	d3
10102	Ann	Jones	d3
18316	John	Barrimore	d1
28559	Sybil]	Moser	dl

Каждый вложенный запрос может содержать свои вложенные запросы. Подзапросы такого типа называются *подзапросами с многоуровневым вложением*. Максимальная глубина вложения (т. е. количество вложенных запросов) зависит от объема памяти, которым компонент Database Engine располагает для каждой инструкции SELECT. В случае подзапросов с многоуровневым вложением система сначала выполняет самый глубокий вложенный запрос и возвращает полученный результат запросу следующего высшего уровня, который в свою очередь возвращает свой результат запросу следующего уровня над ним и т. д. Конечный результат выдается запросом самого высшего уровня.

Запрос с несколькими уровнями вложенности показан в примере 6.53.

Пример 6.53. Выборка фамилий всех сотрудников, работающих над проектом Apollo

```
USE sample;
SELECT emp_lname
   FROM employee
   WHERE emp_no IN
   (SELECT emp_no
      FROM works_on
      WHERE project_no IN
      (SELECT project_no
         FROM project
         WHERE project_name = 'Apollo'));
```

Результат выполнения этого запроса:

emp_lname
Jones
James
Bertoni
Moser

В примере 6.53 самый глубокий вложенный запрос выбирает из таблицы `project_no` значение `p1`. Этот результат передается следующему вышестоящему запросу, который обрабатывает столбец `project_no` в таблице `works_on`. Результатом этого запроса является набор табельных номеров сотрудников: (`10102`, `29346`, `9031`, `28559`). Наконец, самый внешний запрос выводит фамилии сотрудников, чьи номера были выбраны предыдущим запросом.

Подзапросы и операторы *ANY* и *ALL*

Операторы `ANY` и `ALL` всегда используются в комбинации с одним из операторов сравнения. Оба оператора имеют одинаковый синтаксис:

column_name operator [ANY|ALL] *query*

Параметр *operator* обозначает оператор сравнения, а параметр *query* — вложенный запрос.

Оператор `ANY` возвращает значение `TRUE` (истина), если результат соответствующего вложенного запроса содержит хотя бы одну строку, удовлетворяющую условию сравнения. Ключевое слово `SOME` является синонимом `ANY`. Использование оператора `ANY` показано в примере 6.54.

Пример 6.54. Выборка табельного номера, номера проекта и названия должности для сотрудников, которые не затратили большую часть своего времени при работе над одним из проектов

```
USE sample;
SELECT DISTINCT emp_no, project_no, job
   FROM works_on
   WHERE enter_date > ANY
   (SELECT  enter_date
      FROM works_on);
```

Результат выполнения этого запроса:

emp_no	project_no	Job
2581	p3	Analyst
9031	p1	Manager
9031	p3	Clerk
10102	p3	Manager

```
18316        p2              NULL
25348        p2              Clerk
28559        p1              NULL
28559        p2              Clerk
29346        p1              Clerk
29346        p2              NULL
```

В примере 6.54 каждое значение столбца enter_date сравнивается со всеми другими значениями этого же столбца. Для всех дат этого столбца, за исключением самой ранней, сравнение возвращает значение TRUE (истина), по крайней мере, один раз. Строка с самой ранней датой не попадает в результирующий набор, поскольку сравнение ее даты со всеми другими датами никогда не возвращает значение TRUE (истина). Иными словами, выражение enter_date > ANY (SELECT enter_date FROM works_on) возвращает значение TRUE, если в таблице works_on имеется *любое* количество строк (одна или больше), для которых значение столбца enter_date меньше, чем значение enter_date текущей строки. Этому условию удовлетворяют все значения столбца enter_date, за исключением наиболее раннего.

Оператор ALL возвращает значение TRUE, если вложенный запрос возвращает все значения, обрабатываемого им столбца.

ПРИМЕЧАНИЕ

Настоятельно рекомендуется избегать использования операторов ANY и ALL. Любой запрос с применением этих операторов можно сформулировать лучшим образом посредством функции EXISTS, которая рассматривается далее в этой главе (см. далее *разд. "Подзапросы и функция EXISTS"* этой главы). Кроме этого, семантическое значение оператора ANY можно легко принять за семантическое значение оператора ALL и наоборот.

Временные таблицы

Временная таблица — это объект базы данных, который хранится и управляется системой базы данных на временной основе. Временные таблицы могут быть локальными или глобальными. Локальные временные таблицы представлены физически, т. е. они хранятся в системной базе данных tempdb. Имена временных таблиц начинаются с префикса #, например #table_name.

Временная таблица принадлежит создавшему ее сеансу, и видима только этому сеансу. Временная таблица удаляется по завершению создавшего ее сеанса. (Также локальная временная таблица, определенная в хранимой процедуре, удаляется по завершению выполнения этой процедуры.)

Глобальные временные таблицы видимы любому пользователю и любому соединению и удаляются после отключения от сервера базы данных всех обращающихся к ним пользователей. В отличие от локальных временных таблиц имена глобальных временных таблиц начинаются с префикса ##.

В примерах 6.55 и 6.56 показано создание временной таблицы, называющейся project_temp, используя две разные инструкции языка Transact-SQL.

Пример 6.55. Создание временной таблицы посредством инструкции CREATE TABLE

```
USE sample;
CREATE TABLE #project_temp
   (project_no CHAR(4) NOT NULL,
   project_name CHAR(25) NOT NULL);
```

Пример 6.56. Создание временной таблицы посредством инструкции SELECT

```
USE sample;
SELECT project_no, project_name
   INTO #project_temp1
   FROM project;
```

Примеры 6.55 и 6.56 похожи в том, что в обоих создается локальная временная таблица #project_temp and #project_temp1. При этом таблица, созданная в примере 6.55 инструкцией CREATE TABLE, остается пустой, а созданная в примере инструкцией SELECT заполняется данными из таблицы project.

Оператор соединения *JOIN*

В предшествующих разделах этой главы мы рассмотрели применение инструкции SELECT для выборки данных из одной таблицы базы данных. Если бы возможности языка Transact-SQL ограничивались поддержкой только таких простых инструкций SELECT, то присоединение в запросе двух или больше таблиц для выборки из них данных было бы невозможно. Следственно, все данные базы данных требовалось бы хранить в одной таблице. Хотя такой подход является вполне возможным, ему присущ один значительный недостаток — хранимые таким образом данные характеризуются высокой избыточностью.

Язык Transact-SQL устраняет этот недостаток, предоставляя для этого оператор соединения JOIN, который позволяет извлекать данные более чем из одной таблицы. Этот оператор, наверное, является наиболее важным оператором для реляционных систем баз данных, поскольку благодаря ему имеется возможность распределять данные по нескольким таблицам, обеспечивая, таким образом, важное свойство систем баз данных — отсутствие избыточности данных.

ПРИМЕЧАНИЕ

Оператор UNION также позволяет выполнять запрос по нескольким таблицам. Но этот оператор позволяет присоединить несколько инструкций SELECT, тогда как оператор соединения JOIN соединяет несколько таблиц с использованием всего лишь одной инструкции SELECT. Кроме этого, оператор UNION объединяет строки таблиц, в то время как оператор JOIN соединяет столбцы (как мы увидим далее в этой главе).

Оператор соединения также можно применять с базовыми таблицами и представлениями. В этой главе рассматривается соединение базовых таблиц, а соединения представлений рассматриваются в *главе 11*.

Оператор соединения `JOIN` имеет несколько разных форм. В этом разделе рассматриваются следующие основные формы этого оператора:

- естественное соединение;
- декартово произведение или перекрестное соединение;
- внешнее соединение;
- тета-соединение, самосоединение и полусоединение.

Прежде чем приступить к рассмотрению разных форм соединений, в этом разделе мы рассмотрим разные варианты оператора соединения `JOIN`.

Две синтаксические формы реализации соединений

Для соединения таблиц можно использовать две разные синтаксические формы оператора соединения:

- явный синтаксис соединения (синтаксис соединения ANSI SQL:1992);
- неявный синтаксис соединения (синтаксис соединения "старого стиля").

Синтаксис соединения ANSI SQL:1992 был введен стандартом SQL92 и определяет операции соединения явно, т. е. используя соответствующее имя для каждого типа операции соединения. При явном объявлении соединения используются следующие ключевые слова:

- `CROSS JOIN`;
- `[INNER] JOIN`;
- `LEFT [OUTER] JOIN`;
- `RIGHT [OUTER] JOIN`;
- `FULL [OUTER] JOIN`.

Ключевое слово `CROSS JOIN` определяет декартово произведение двух таблиц. Ключевое слово `INNER JOIN` определяет естественное соединение двух таблиц, а `LEFT OUTER JOIN` и `RIGHT OUTER JOIN` определяют одноименные операции соединения. Наконец, ключевое слово `FULL OUTER JOIN` определяет соединение правого и левого внешнего соединений. Все эти операции соединения рассматриваются в последующих разделах.

Неявный синтаксис оператора соединения является синтаксисом "старого стиля", где каждая операция соединения определяется неявно посредством предложения `WHERE`, используя так называемые *столбцы соединения* (см. вторую инструкцию в примере 6.57).

ПРИМЕЧАНИЕ

Для операций соединения рекомендуется использовать явный синтаксис, т. к. это повышает надежность запросов. По этой причине во всех примерах в этой главе, связанные с операциями соединения, используются формы явного синтаксиса. Но в нескольких первых примерах также будет продемонстрирован и синтаксис "старого стиля".

Естественное соединение

Наилучшим способом объяснить *естественное соединение* можно посредством примера 6.57.

ПРИМЕЧАНИЕ

Термины "естественное соединение" (natural join) и "соединение по эквивалентности" (equi-join) часто используют синонимично, но между ними есть небольшое различие. Операция соединения по эквивалентности всегда имеет одну или несколько пар столбцов с идентичными значениями в каждой строке. Операция, которая устраняет такие столбцы из результатов операции соединения по эквивалентности, называется *естественным соединением*.

Запрос в примере 6.57 возвращает всю информацию обо всех сотрудниках: имя и фамилию, табельный номер, а также имя, номер и местонахождение отдела, при этом для номера отдела отображаются дубликаты столбцов из разных таблиц.

Пример 6.57. Явный синтаксис оператора соединения

```
USE sample;
SELECT employee.*, department.*
   FROM employee INNER JOIN department
   ON employee.dept_no = department.dept_no;
```

В примере 6.57 в инструкции `SELECT` для выборки указаны все столбцы таблиц для сотрудника `employee` и отдела `department`. Предложение `FROM` инструкции `SELECT` определяет соединяемые таблицы, а также явно указывает тип операции соединения — `INNER JOIN`. Предложение `ON` является частью предложения `FROM` и указывает соединяемые столбцы в обеих таблицах. Выражение `employee.dept_no = department.dept_no` определяет условие соединения, а оба столбца условия называются *столбцами соединения*.

Эквивалентный запрос с применением неявного синтаксиса ("старого стиля") будет выглядеть следующим образом:

```
USE sample;
SELECT employee.*, department.*
   FROM employee, department
   WHERE employee.dept_no = department.dept_no;
```

Эта форма синтаксиса имеет два значительных различия с явной формой: список соединяемых таблиц указывается в предложении FROM, а соответствующее условие соединения указывается в предложении WHERE посредством соединяемых столбцов.

Результат выполнения этого запроса:

emp_no	emp_fname	emp_lname	dept_no	dept_no	dept_name	location
25348	Matthew	Smith	d3	d3	Marketing	Dallas
10102	Ann	Jones	d3	d3	Marketing	Dallas
18316	John	Barrimore	d1	d1	Research	Dallas
29346	James	James	d2	d2	Accounting	Seattle
9031	Elsa	Bertoni	d2	d2	Accounting	Seattle
2581	Elke	Hansel	d2	d2	Accounting	Seattle
28559	Sybill	Moser	dl	dl	Research	Dallas

ПРИМЕЧАНИЕ

Настоятельно рекомендуется использовать подстановочный знак * в списке выбора SELECT только при работе с SQL в интерактивном режиме, и избегать его применения в прикладных программах.

На примере 6.57 можно проиллюстрировать принцип работы операции соединения. Но при этом следует иметь в виду, что это всего лишь представление о процессе соединения, т. к. в действительности компонент Database Engine выбирает реализацию операции соединения из нескольких возможных стратегий. Представьте себе, что каждая строка таблицы employee соединена с каждой строкой таблицы department. В результате получится таблица с семью столбцами (4 столбца из таблицы employee и 3 из таблицы department) и 21 строкой (табл. 6.1).

Таблица 6.1. *Результат декартового произведения таблиц* employee *и* department

emp_no	emp_fname	emp_lname	dept_no	dept_no	dept_name	location
*25348	Matthew	Smith	d3	dl	Research	Dallas
*10102	Ann	Jones	d3	d1	Research	Dallas
18316	John	Barrimore	d1	d1	Research	Dallas
*29346	James	James	d2	d1	Research	Dallas
*9031	Elsa	Bertoni	d2	dl	Research	Dallas
*2581	Elke	Hansel	d2	dl	Research	Dallas
28559	Sybill	Moser	dl	dl	Research	Dallas
*25348	Matthew	Smith	d3	d2	Accounting	Seattle
*10102	Ann	Jones	d3	d2	Accounting	Seattle
*18316	John	Barrimore	d1	d2	Accounting	Seattle

Таблица 6.1 *(окончание)*

emp_no	emp_fname	emp_lname	dept_no	dept_no	dept_name	location
29346	James	James	d2	d2	Accounting	Seattle
9031	Elsa	Bertoni	d2	d2	Accounting	Seattle
2581	Elke	Hansel	d2	d2	Accounting	Seattle
*28559	Sybill	Moser	d1	d2	Accounting	Seattle
25348	Matthew	Smith	d3	d3	Marketing	Dallas
10102	Ann	Jones	d3	d3	Marketing	Dallas
*18316	John	Barrimore	d1	d3	Marketing	Dallas
*29346	James	James	d2	d3	Marketing	Dallas
*9031	Elsa	Bertoni	d2	d3	Marketing	Dallas
*2581	Elke	Hansel	d2	d3	Marketing	Dallas
*28559	Sybill	Moser	d1	d3	Marketing	Dallas

Далее, из табл. 6.1 удаляются все строки, которые не удовлетворяют условию соединения `employee.dept_no = department.dept_no`. В табл. 6.1 эти строки обозначены символом `*`. Оставшиеся строки представляют результат примера 6.57. Соединяемые столбцы должны иметь идентичную семантику, т. е. оба столбца должны иметь одинаковое логическое значение. Соединяемые столбцы не обязательно должны иметь одинаковое имя (или даже одинаковый тип данных), хотя часто так и бывает.

ПРИМЕЧАНИЕ

Система базы данных не может определить логическое значение столбца. Например, она не может определить, что между столбцами номера проекта и табельного номера сотрудника нет ничего общего, хотя оба они имеют целочисленный тип данных. Поэтому система базы данных может только проверить тип данных и длину строк. Компонент Database Engine требует, что соединяемые столбцы имели совместимые типы данных, например `INT` и `SMALLINT`.

База данных `sample` содержит три пары столбцов, где каждый столбец в паре имеет одинаковое логическое значение (а также одинаковые имена). Таблицы `employee` и `department` можно соединить по столбцам `employee.dept_no` и `department.dept_no`. Столбцами соединения таблиц `employee` и `works_on` являются столбцы `employee.emp_no` и `works_on.emp_no`. Наконец, таблицы `project` и `works_on` можно соединить по столбцам `project.project_no` и `works_on.project_no`.

Имена столбцов в инструкции `SELECT` можно уточнить. В данном контексте под уточнением имеется в виду, что во избежание неопределенности относительно того, какой таблице принадлежит столбец, в имя столбца включается имя его таблицы (или псевдоним таблицы), отделенное точкой: *`table_name.column_name`* (*`имя_таблицы.имя_столбца`*).

В большинстве инструкций SELECT столбцы не требуют уточнения, хотя обычно рекомендуется применять уточнение столбцов с целью улучшения понимания кода. Если же имена столбцов в инструкции SELECT неоднозначны (как, например, столбцы dept_no в таблицах employee и department в примере 6.57) использование уточненных имен столбцов является *обязательным*.

В инструкции SELECT с операцией соединения, кроме условия соединения предложение WHERE может содержать и другие условия, как это показано в примере 6.58.

Пример 6.58. Выборка полной информации сотрудников, работающих над проектом Gemini

Явный синтаксис соединения:

```
USE sample;
SELECT emp_no, project.project_no, job, enter_date, project_name, budget
   FROM works_on JOIN project
   ON project.project_no = works_on.project_no
   WHERE project_name = 'Gemini';
```

Неявный синтаксис соединения ("старый стиль"):

```
USE sample;
SELECT emp_no, project.project_no, job, enter_date, project_name, budget
   FROM works_on, project
   WHERE project.project_no = works_on.project_no
   AND project_name = 'Gemini';
```

ПРИМЕЧАНИЕ

Использование уточненных имен столбцов emp_no, project_name, job и budget в примере 6.58 не является обязательным, поскольку в данном случае нет никакой двусмысленности в отношении их имен.

Обе формы запроса возвращают одинаковый результат:

emp_no	project_no	job	enter_date	project_name	budget
25348	p2	Clerk	2007-02-15	Gemini	95000.0
18316	p2	NULL	2007-06-01	Gemini	95000.0
29346	p2	NULL	2006-12-15	Gemini	95000.0
28559	p2	Clerk	2008-02-01	Gemini	95000.0

В дальнейшем во всех примерах будет использоваться только явный синтаксис соединения.

В примере 6.59 показано еще одно применение внутреннего соединения.

Пример 6.59. Выборка номера отдела сотрудников, приступивших к работе над проектами 15 октября 2007 г.

```
USE sample;
SELECT dept_no
   FROM employee JOIN works_on
   ON employee.emp_no = works_on.emp_no
   WHERE enter_date = '10.15.2007';
```

Результат выполнения этого запроса:

dept_no
d2

Соединение более чем двух таблиц

Теоретически количество таблиц, которые можно соединить в инструкции SELECT, неограниченно. (Но одно условие соединения совмещает только две таблицы!) Однако для компонента Database Engine количество соединяемых таблиц в инструкции SELECT ограничено 64 таблицами.

В примере 6.60 показано соединение трех таблиц базы данных sample.

Пример 6.60. Выборка имен и фамилий всех аналитиков (job = 'analyst'), чей отдел находится в Сиэтле (location = 'Seattle')

```
USE sample;
SELECT emp_fname, emp_lname
  FROM works_on JOIN employee ON works_on.emp_no=employee.emp_no
                JOIN department ON employee.dept_no=department.dept_no
  AND location = 'Seattle'
  AND job = 'analyst';
```

Результат выполнения этого запроса:

emp_fname	emp_lname
Elke	Hansel

Результат запроса, приведенного в примере 6.60, можно получить только в том случае, если соединить, по крайней мере, три таблицы: works_on, employee и department. Эти таблицы можно соединить, используя две пары столбцов соединения:

```
(works_on.emp_no, employee.emp_no)
(employee.dept_no, department.dept_no)
```

В примере 6.61 показано соединение четырех таблиц базы данных sample.

Пример 6.61. Выборка наименований проектов (с удалением избыточных дубликатов), в которых участвуют сотрудники бухгалтерии (отдел Accounting)

```
USE sample;
SELECT DISTINCT project_name
   FROM project JOIN works_on
   ON project.project_no = works_on.project_no
      JOIN employee ON works_on.emp_no = employee.emp_no
      JOIN department ON employee.dept_no = department.dept_no
   WHERE dept_name = 'Accounting';
```

Результат выполнения этого запроса соединения будет следующим:

project_name
Apollo
Gemini
Mercury

Обратите внимание, что для осуществления естественного соединения трех таблиц используется два условия соединения, каждое из которых соединяет по две таблицы. А при соединении четырех таблиц таких условий соединения требуется три. В общем, чтобы избежать получения декартового продукта при соединении n таблиц, требуется применять $n - 1$ условий соединения. Конечно же, допустимо использование более чем $n - 1$ условий соединения, а также других условий, для того чтобы еще больше уменьшить количество элементов в результирующем наборе данных.

Декартово произведение

В предшествующем разделе мы рассмотрели возможный способ создания естественного соединения. На первом шаге этого процесса каждая строка таблицы employee соединяется с каждой строкой таблицы department. Эта операция называется *декартовым произведением* (cartesian product). Запрос для создания соединения таблиц employee и department, используя декартово произведение, показан в примере 6.62.

Пример 6.62. Соединение таблиц декартовым произведением

```
USE sample;
SELECT  employee.*, department.*
   FROM employee CROSS JOIN department;
```

Результирующий набор выполнения примера 6.62 показан ранее в табл. 6.1.

Декартово произведение соединяет каждую строку первой таблицы с каждой строкой второй. В общем, результатом декартового произведения первой таблицы

с n строками и второй таблицы с m строками будет таблица с $n \times m$ строками. Таким образом, результирующий набор запроса в примере 6.62 имеет $7 \times 3 = 21$ строку.

На практике декартово произведение применяется крайне редко. Иногда пользователи получают декартово произведение двух таблиц, когда они забывают включить условие соединения в предложении `WHERE` при использовании неявного синтаксиса соединения "старого стиля". В таком случае полученный результат не соответствует ожидаемому, т. к. содержит лишние строки. Наличие неожидаемо большого количества строк в результате служит признаком того, что вместо требуемого естественного соединения двух таблиц было получено декартово произведение.

Внешнее соединение

В предшествующих примерах естественного соединения, результирующий набор содержал только те строки с одной таблицы, для которых имелись соответствующие строки в другой таблице. Но иногда кроме совпадающих строк бывает необходимым извлечь из одной или обеих таблиц строки без совпадений. Такая операция называется *внешним соединением* (outer join).

В примерах 6.63 и 6.64 показано различие между естественным соединением и соответствующим ему внешним соединением. (Во всех примерах в этом разделе используется таблица `employee_enh`.)

Пример 6.63. Выборка всей информации для сотрудников, которые проживают и работают в одном и том же городе

```
USE sample;
SELECT employee_enh.*, department.location
   FROM employee_enh JOIN department
      ON domicile = location;
```

Результат выполнения этого запроса:

emp_no	emp_fname	emp_lname	dept_no	domicile	location
29346	James	James	d2	Seattle	Seattle

В примере 6.63 получение требуемых строк осуществляется посредством естественного соединения. Если бы в этот результат потребовалось включить сотрудников, проживающих в других местах, то нужно было применить левое внешнее соединение. Данное внешнее соединение называется *левым* потому, что оно возвращает все строки из таблицы с *левой* стороны оператора сравнения, независимо от того, имеются ли совпадающие строки в таблице с правой стороны. Иными словами, данное внешнее соединение возвратит строку с левой таблицы, даже если для нее нет совпадения в правой таблице, со значением `NULL` соответствующего столбца для всех строк с несовпадающим значением столбца другой, правой, таблицы (см. пример 6.64). Для выполнения операции левого внешнего соединения компонент Database Engine использует оператор `LEFT OUTER JOIN`.

Операция *правого* внешнего соединения аналогична левому, но возвращаются все строки таблицы с *правой* части выражения. Для выполнения операции правого внешнего соединения компонент Database Engine использует оператор `RIGHT OUTER JOIN`.

Пример 6.64. Выборка сотрудников (с включением полной информации) для таких городов, в которых сотрудники или только проживают, или проживают и работают

```
USE sample;
SELECT employee_enh.*, department.location
   FROM employee_enh LEFT OUTER JOIN department
      ON domicile = location;
```

Результат выполнения этого запроса:

emp_no	emp_fname	emp_lname	dept_no	domicile	location
25348	Matthew	Smith	d3	San Antonio	NULL
10102	Ann	Jones	d3	Houston	NULL
18316	John	Barrimore	d1	San Antonio	NULL
29346	James	James	d2	Seattle	Seattle
9031	Elsa	Bertoni	d2	Portland	NULL
2581	Elke	Hansel	d2	Tacoma	NULL
28559	Sybil]	Moser	dl	Houston	NULL

Как можно видеть в результате выполнения запроса, когда для строки из левой таблицы (в данном случае `employee_enh`) нет совпадающей строки в правой таблице (в данном случае `department`), операция левого внешнего соединения все равно возвращает эту строку, заполняя значением `NULL` все ячейки соответствующего столбца для несовпадающего значения столбца правой таблицы. Применение правого внешнего соединения показано в примере 6.65.

Пример 6.65. Выборка отделов (с включением полной информации о них) для таких городов, в которых сотрудники или только проживают, или проживают и работают

```
USE sample;
SELECT employee_enh.domicile, department.*
   FROM employee_enh RIGHT OUTER JOIN department
      ON domicile =location;
```

Результат выполнения этого запроса:

domicile	dept_no	dept_name	location
Seattle	d2	Accounting	Seattle
NULL	dl	Research	Dallas
NULL	d3	Marketing	Dallas

Кроме левого и правого внешнего соединения, также существует полное внешнее соединение, которое является объединением левого и правого внешних соединений. Иными словами, результирующий набор такого соединения состоит из всех строк обеих таблиц. Если для строки одной из таблиц нет соответствующей строки в другой таблице, всем ячейкам строки второй таблицы присваивается значение NULL. Для выполнения операции правого внешнего соединения используется оператор FULL OUTER JOIN.

Любую операцию внешнего соединения можно эмулировать, используя оператор UNION совместно с функцией NOT EXISTS. Таким образом, запрос, показанный в примере 6.66, эквивалентен запросу левого внешнего соединения (см. пример 6.64). В данном запросе осуществляется выборка сотрудников (с включением полной информации) для таких городов, в которых сотрудники или только проживают или проживают и работают.

Пример 6.66. Эмуляция левого внешнего соединения

```
USE sample;
SELECT employee_enh.*, department.location
   FROM employee_enh JOIN department
   ON domicile = location
UNION
SELECT employee_enh.*, 'NULL'
   FROM employee_enh
   WHERE NOT EXISTS
   (SELECT *
      FROM department
      WHERE location = domicile);
```

Первая инструкция SELECT объединения определяет естественное соединение таблиц employee_enh и department по столбцам соединения domicile и location. Эта инструкция возвращает все города для всех сотрудников, в которых сотрудники и проживают и работают. Дополнительно, вторая инструкция SELECT объединения возвращает все строки таблицы employee_enh, которые не отвечают условию в естественном соединении.

Другие формы операций соединения

В предшествующих разделах мы рассмотрели наиболее важные формы соединения. Но существуют и другие формы этой операции, которые мы рассмотрим в этом разделе. А именно:

- тета-соединение;
- самосоединение;
- полусоединение.

Эти формы рассматриваются в следующих подразделах.

Тета-соединение

Условие сравнения столбцов соединения не обязательно должно быть равенством, но может быть любым другим сравнением. Соединение, в котором используется общее условие сравнения столбцов соединения, называется *тета-соединением*. В примере 6.67 показана операция тета-соединения, в которой используется условие "меньше чем". Данный запрос возвращает все комбинации информации о сотрудниках и отделах для тех случаев, когда место проживания сотрудника по алфавиту идет перед месторасположением любого отдела, в котором работает этот служащий.

Пример 6.67. Операция тета-соединения с условием "меньше чем"

```
USE sample;
SELECT emp_fname, emp_lname, domicile, location
   FROM employee_enh JOIN department
   ON domicile < location;
```

Результат выполнения этого запроса:

emp_fname	emp_lname	domicile	location
Matthew	Smith	San Antonio	Seattle
Ann	Jones	Houston	Seattle
John	Barrimore	San Antonio	Seattle
Elsa	Bertoni	Portland	Seattle
Sybill	Moser	Houston	Seattle

В примере 6.67 сравниваются соответствующие значения столбцов `domicile` и `location`. В каждой строке результата значение столбца `domicile` сравнивается в алфавитном порядке с соответствующим значением столбца `location`.

Самосоединение, или соединение таблицы самой с собой

Кроме соединения двух или больше разных таблиц, операцию естественного соединения можно применить к одной таблице. В данной операции таблица соединяется сама с собой, при этом один столбец таблицы сравнивается сам с собой. Сравнивание столбца с самим собой означает, что в предложении `FROM` инструкции `SELECT` имя таблицы употребляется дважды. Поэтому необходимо иметь возможность ссылаться на имя одной и той же таблицы дважды. Это можно осуществить, используя, по крайней мере, один псевдоним. То же самое относится и к именам столбцов в условии соединения в инструкции `SELECT`. Для того чтобы различить столбцы с одинаковыми именами, необходимо использовать уточненные имена. Соединение таблицы с самой собой демонстрируется в примере 6.68.

Пример 6.68. Выборка всех отделов (с полной информацией), расположенных в том же самом месте, как и, по крайней мере, один другой отдел

```
USE sample;
SELECT t1.dept_no, t1.dept_name, t1.location
FROM department t1 JOIN department t2
ON t1.location = t2.location
WHERE t1.dept_no <> t2.dept_no;
```

Результат выполнения этого запроса:

dept_no	dept_name	location
d3	Marketing	Dallas
d1	Research	Dallas

В примере 6.68 предложение `FROM` содержит два псевдонима для таблицы `department`: `t1` и `t2`. Первое условие в предложении `WHERE` определят столбцы соединения, а второе — удаляет ненужные дубликаты, обеспечивая сравнение каждого отдела с *другими* отделами.

Полусоединение

Полусоединение похоже на естественное соединение, но возвращает только набор всех строк из одной таблицы, для которой в другой таблице есть одно или несколько совпадений. Использование полусоединения показано в примере 6.69.

Пример 6.69. Использование полусоединения

```
USE sample;
SELECT emp_no, emp_lname, e.dept_no
   FROM employee e JOIN department d
   ON e.dept_no = d.dept_no WHERE location = 'Dallas';
```

Результат выполнения запроса:

emp_no	Emp_lname	dept_no
25348	Smith	d3
10102	Jones	d3
18316	Barrimore	d1
28559	Moser	d1

Как можно видеть в примере 6.69, список выбора `SELECT` в полусоединении содержит только столбцы из таблицы `employee`. Это и есть характерной особенностью операции полусоединения. Эта операция обычно применяется в распределенной обработке запросов, чтобы свести к минимуму объем передаваемых данных. Компонент Database Engine использует операцию полусоединения для реализации функциональности, называющейся *соединением типа "звезда"* (см. *главу 25*).

Связанные подзапросы

Подзапрос называется *связанным* (correlated), если любые значения вложенного запроса зависят от внешнего запроса. В примере 6.70 показано использование связанного подзапроса.

Пример 6.70. Выборка фамилий всех сотрудников, работающих над проектом p3

```
USE sample;
SELECT emp_lname
   FROM employee
   WHERE 'p3' IN
   (SELECT project_no
      FROM works_on
      WHERE works_on.emp_no = employee.emp_no);
```

Результат выполнения этого запроса:

emp_lname
Jones
Bertoni
Hansel

В примере 6.70 вложенный запрос должен логически выполниться несколько раз, поскольку он содержит столбец emp_no, который принадлежит таблице employee во внешнем запросе, и значение столбца emp_no изменяется каждый раз, когда проверяется другая строка таблицы employee во внешнем запросе.

Давайте проследим, как система может выполнять запрос в примере 6.70. Сначала система выбирает первую строку таблицы employee (для внешнего запроса) и сравнивает табельный номер сотрудника в этом столбце (25348) со значениями столбца works_on.emp_no вложенного запроса. Поскольку для этого сотрудника имеется только одно значение project_no равное p2, вложенный запрос возвращает значение p2. Это единственное значение результирующего набора вложенного запроса не равно значению p3 внешнего запроса, условие внешнего запроса (WHERE 'p3' IN...) не удовлетворяется и, следовательно, внешний запрос не возвращает никаких строк для этого сотрудника. Далее система берет следующую строку таблицы employee и снова сравнивает номера сотрудников в обеих таблицах. Для этой строки в таблице works_on есть две строки, для которых значение project_no равно p1 и p3 соответственно. Следовательно, вложенный запрос возвращает результат p1 и p3. Значение одного из элементов этого результирующего набора равно константе p3, поэтому условие удовлетворяется, и отображается соответствующее значение второй строки столбца emp_lname (Jones). Такой же обработке подвергаются все остальные строки таблицы employee, и в конечном результате возвращается набор из трех строк.

В следующем разделе приводятся дополнительные примеры по связанным подзапросам.

Подзапросы и функция *EXISTS*

Функция EXISTS принимает вложенный запрос в качестве аргумента и возвращает значение FALSE, если вложенный запрос не возвращает строк и значение TRUE в противном случае. Рассмотрим работу этой функции на нескольких примерах, начиная с примера 6.71.

Пример 6.71. Выборка фамилий всех сотрудников, работающих над проектом p1

```
USE sample;
SELECT emp_lname
   FROM employee
   WHERE EXISTS (SELECT *
      FROM works_on
      WHERE employee.emp_no = works_on.emp_no
      AND project_no = 'p1');
```

Результат выполнения этого запроса:

emp_lname
Jones
James
Bertoni
Moser

Вложенный запрос функции EXISTS почти всегда зависит от переменной с внешнего запроса. Поэтому функция EXISTS обычно определяет связанный подзапрос.

Давайте проследим, как Database Engine может обрабатывать запрос в примере 6.71. Сначала внешний запрос рассматривает первую строку таблицы employee (сотрудник Smith). Далее функция EXISTS определяет, есть ли в таблице works_on строки, чьи номера сотрудников совпадают с номером сотрудника в текущей строке во внешнем запросе и чей project_no равен p1. Поскольку сотрудник Smith не работает над проектом p1, вложенный запрос возвращает пустой набор, вследствие чего функция EXISTS возвращает значение FALSE. Таким образом, сотрудник Smith не включается в конечный результирующий набор. Этому процессу подвергаются все строки таблицы employee, после чего выводится конечный результирующий набор.

В примере 6.72 показано использование функции NOT EXISTS.

Пример 6.72. Выборка фамилий сотрудников, чей отдел не расположен в Сиэтле

```
USE sample;
SELECT emp_lname
```

```
FROM employee
WHERE NOT EXISTS (SELECT *
FROM department
WHERE employee.dept_no = department.dept_no
AND location = 'Seattle');
```

Результат выполнения этого запроса будет следующим:

emp_lname
Smith
Jones
Barrimore
Moser

Список выбора инструкции SELECT во внешнем запросе с функцией EXISTS не обязательно должен быть в форме SELECT *, как в предыдущем примере. Можно использовать альтернативную форму SELECT colum_list, где column_list представляет список из одного или нескольких столбцов таблицы. Обе формы равнозначны, потому что функция EXIST только проверяет на наличие (или отсутствие) строк в результирующем наборе. По этой причине в данном случае правильнее использовать форму SELECT *.

Что использовать, соединения или подзапросы?

Почти все инструкции SELECT для соединения таблицы посредством оператора соединения можно заменить инструкциями подзапроса и наоборот. Конструкция инструкции SELECT с использованием оператора соединения часто более удобно читаемая и легче понимаемая, а также может помочь компоненту Database Engine найти более эффективную стратегию для выборки требуемых данных. Но некоторые задачи легче поддаются решению посредством подзапросов, а другие при помощи соединений.

Преимущества подзапросов

Подзапросы будет более выгодно использовать в таких случаях, когда требуется вычислить агрегатное значение "на лету" и использовать его в другом запросе для сравнения. Это показано в примере 6.73.

Пример 6.73. Выборка табельных номеров сотрудников и дат начала их работы над проектом (enter_date) для всех сотрудников, у которых дата начала работы равна самой ранней дате

```
USE sample;
SELECT emp_no, enter_date
   FROM works_on
   WHERE enter_date = (SELECT min(enter_date)
      FROM works_on);
```

Решить эту задачу с помощью соединения будет нелегко, поскольку для этого нужно поместить агрегатную функцию в предложении WHERE, а это не разрешается. (Эту задачу можно решить, используя два отдельных запроса по отношении к таблице works_on.)

Преимущества соединений

Использовать соединения вместо подзапросов выгоднее в тех случаях, когда список выбора инструкции SELECT в запросе содержит столбцы более чем из одной таблицы. Это показано в примере 6.74.

Пример 6.74. Выборка информации о всех сотрудниках (табельный номер, фамилия и должность), которые начали участвовать в работе над проектом 15 октября 2007 г.

```
USE sample;
SELECT employee,emp_no, emp_lname, job
   FROM employee, works_on
   WHERE employee.emp_no = works_on.emp_no
   AND enter_date = '10.15.2007';
```

Список выбора инструкции SELECT в запросе примера 6.74 содержит столбцы emp_no и emp_lname из таблицы employee и столбец job из таблицы works_on. По этой причине решение с применением подзапроса возвратило бы ошибку, поскольку подзапросы могут отображать информацию только из внешней таблицы.

Табличные выражения

Табличными выражениями называются подзапросы, которые используются там, где ожидается наличие таблицы. Существует два типа табличных выражений:

- производные таблицы;
- обобщенные табличные выражения.

Эти две формы табличных выражений рассматриваются в следующих подразделах.

Производные таблицы

Производная таблица (derived table) — это табличное выражение, входящее в предложение FROM запроса. Производные таблицы можно применять в тех случаях, когда использование псевдонимов столбцов не представляется возможным, поскольку транслятор SQL обрабатывает другое предложение до того, как псевдоним станет известным. В примере 6.75 показана попытка использовать псевдоним столбца в ситуации, когда другое предложение обрабатывается до того, как станет известным псевдоним.

Пример 6.75. Неправильная попытка выбрать все группы месяцев из значений столбца enter_date таблицы works_on

```
USE sample;
SELECT MONTH(enter_date) as enter_month
   FROM works_on
   GROUP BY enter_month;
```

Попытка выполнить этот запрос выдаст следующее сообщение об ошибке:

```
Message 207: Level 16, State 1, Line 4 The invalid column 'enter_month'
```

(Сообщение 207: уровень 16, состояние 1, строка 4 Недопустимое имя столбца enter_month)

Причиной ошибки является то обстоятельство, что предложение GROUP BY обрабатывается до обработки соответствующего списка инструкции SELECT, и при обработке этой группы псевдоним столбца enter_month неизвестен.

Эту проблему можно решить, используя производную таблицу, содержащую предшествующий запрос (без предложения GROUP BY), поскольку предложение FROM исполняется перед предложением GROUP BY (пример 6.76).

Пример 6.76. Правильная конструкция запроса из примера 6.75

```
USE sample;
SELECTenter_month
   FROM (SELECT MONTH(enter_date) as enter_month
   FROM works_on) AS m
   GROUP BY enter_month;
```

Результат выполнения этого запроса будет таким:

enter_month
1
2
4
6
8
10
11
12

Обычно табличное выражение можно разместить в любом месте инструкции SELECT, где может появиться имя таблицы. (Результатом табличного выражения всегда является таблица или, в особых случаях, выражение.) В примере 6.77 показывается использование табличного выражения в списке выбора инструкции SELECT.

Пример 6.77. Использование табличного выражения

```
USE sample;
SELECT w.job, (SELECT e.emp_lname
      FROM employee e WHERE e.emp_no = w.emp_no) AS name
   FROM works_on w
   WHERE w.job IN('Manager', 'Analyst');
```

Результат выполнения этого запроса:

Job	name
Analyst	Jones
Manager	Jones
Analyst	Hansel
Manager	Bertoni

Обобщенные табличные выражения

Обобщенным табличным выражением (ОТВ) (Common Table Expression — сокращенно CTE) называется именованное табличное выражение, поддерживаемое языком Transact-SQL. Обобщенные табличные выражения используются в следующих двух типах запросов:

- нерекурсивных;
- рекурсивных.

Эти два типа запросов рассматриваются в следующих далее разделах.

ПРИМЕЧАНИЕ

Обобщенные табличные выражения также используются оператором `APPLY`, который позволяет вызывать функцию с табличным значением для каждой строки, возвращаемой внешним табличным выражением запроса. Оператор `APPLY` рассматривается в *главе 8*.

ОТВ и нерекурсивные запросы

Нерекурсивную форму ОТВ можно использовать в качестве альтернативы производным таблицам и представлениям. Обычно ОТВ определяется посредством предложения `WITH` и дополнительного запроса, который ссылается на имя, используемое в предложении `WITH` (см. пример 6.79).

ПРИМЕЧАНИЕ

В языке Transact-SQL значение ключевого слова `WITH` неоднозначно. Чтобы избежать неопределенности, инструкцию, предшествующую оператору `WITH`, следует завершать точкой с запятой (`;`).

В примерах 6.78 и 6.79 показано использование ОТВ в нерекурсивных запросах. В примере 6.78 используется стандартное решение, в примере 6.79 та же задача решается посредством нерекурсивного запроса.

Пример 6.78. Стандартное решение

```
USE AdventureWorks;
SELECT SalesOrderID
   FROM Sales.SalesOrderHeader
      WHERE TotalDue > (SELECT AVG(TotalDue)
            FROM Sales.SalesOrderHeader
            WHERE YEAR(OrderDate) = '2002')
         AND Freight > (SELECT AVG(TotalDue)
            FROM Sales.SalesOrderHeader
            WHERE YEAR(OrderDate) = '2002')/2.5;
```

Запрос в примере 6.78 выбирает заказы, чьи общие суммы налогов (`TotalDue`) большие, чем среднее значение по всем налогам, и плата за перевозку (`Freight`) которых больше чем 40% среднего значения налогов. Основным свойством этого запроса является его объемистость, поскольку вложенный запрос требуется писать дважды. Одним из возможных способов уменьшить объем конструкции запроса будет создать представление, содержащее вложенный запрос. Но это решение несколько сложно, поскольку требует создания представления, а потом его удаления после окончания выполнения запроса. Лучшим подходом будет создать ОТВ. В примере 6.79 показывается использование нерекурсивного ОТВ, которое сокращает определение запроса, приведенного в примере 6.78.

Пример 6.79. Использование ОТВ для сокращения объема запроса

```
USE AdventureWorks;
WITH price_calc(year_2002) AS
   (SELECT AVG(TotalDue)
      FROM Sales.SalesOrderHeader
      WHERE YEAR(OrderDate) = '2002')
SELECT SalesOrderID
   FROM Sales.SalesOrderHeader
   WHERE TotalDue > (SELECT year_2002  FROM price_calc)
AND Freight > (SELECT year_2002  FROM price_calc)/2.5;
```

Синтаксис предложения `WITH` в нерекурсивных запросах имеет следующий вид:

```
WITH cte_name (column_list) AS
(inner_query)
outer_query
```

Параметр `cte_name` представляет имя ОТВ, которое определяет результирующую таблицу, а параметр `column_list` — список столбцов табличного выражения.

(В примере 6.79 ОТВ называется `price_calc` и имеет один столбец — `year_2002`.) Параметр `inner_query` представляет инструкцию `SELECT`, которая определяет результирующий набор соответствующего табличного выражения. После этого определенное табличное выражение можно использовать во внешнем запросе `outer_query`. (Внешний запрос в примере 6.79 использует ОТВ `price_calc` и ее столбец `year_2002`, чтобы упростить употребляющийся дважды вложенный запрос.)

ОТВ и рекурсивные запросы

ПРИМЕЧАНИЕ

В этом разделе представляется материал повышенной сложности. Поэтому при первом его чтении рекомендуется его пропустить и вернуться к нему позже.

Посредством ОТВ можно реализовывать рекурсии, поскольку ОТВ могут содержать ссылки на самих себя. Основной синтаксис ОТВ для рекурсивного запроса выглядит таким образом:

```
WITH cte_name (column_list) AS
   (anchor_member
   UNION ALL
   recursive_member)
outer_query
```

Параметры `cte_name` и `column_list` имеют такое же значение, как и в ОТВ для нерекурсивных запросов. Тело предложения `WITH` состоит из двух запросов, объединенных оператором `UNION ALL`. Первый запрос вызывается только один раз, и он начинает накапливать результат рекурсии. Первый операнд оператора `UNION ALL` не ссылается на ОТВ (см. пример 6.80). Этот запрос называется *опорным запросом* или *источником*.

Второй запрос содержит ссылку на ОТВ и представляет ее рекурсивную часть. Вследствие этого он называется *рекурсивным членом*. В первом вызове рекурсивной части ссылка на ОТВ представляет результат опорного запроса. Рекурсивный член использует результат первого вызова запроса. После этого система снова вызывает рекурсивную часть. Вызов рекурсивного члена прекращается, когда предыдущий его вызов возвращает пустой результирующий набор.

Оператор `UNION ALL` соединяет накопившиеся на данный момент строки, а также дополнительные строки, добавленные текущим вызовом рекурсивного члена. (Наличие оператора `UNION ALL` означает, что повторяющиеся строки не будут удалены из результата.)

Наконец, параметр `outer_query` определяет внешний запрос, который использует ОТВ для получения всех вызовов объединения обеих членов.

Для демонстрации рекурсивной формы ОТВ мы используем таблицу, определенную и заполненную кодом, показанным в примере 6.80.

Пример 6.80. Создание и заполнение таблицы для демонстрации рекурсивной формы ОТВ

```
USE sample;
CREATE TABLE airplane
   (containing_assembly VARCHAR(10),
    contained_assembly VARCHAR(10),
    quantity_contained INT,
    unit_cost DECIMAL (6,2));
insert into airplane values ('Airplane', 'Fuselage',1, 10);
insert into airplane values ('Airplane', 'Wings', 1, 11);
insert into airplane values ('Airplane', 'Tail',1, 12);
insert into airplane values ('Fuselage', 'Cockpit', 1, 13);
insert into airplane values ('Fuselage', 'Cabin', 1, 14);
insert into airplane values ('Fuselage', 'Nose',1, 15);
insert into airplane values ('Cockpit', NULL, 1,13);
insert into airplane values ('Cabin', NULL, 1, 14);
insert into airplane values ('Nose', NULL, 1, 15);
insert into airplane values ('Wings', NULL,2, 11);
insert into airplane values ('Tail', NULL, 1, 12);
```

Таблица `airplane` состоит из четырех столбцов. Столбец `containing_assembly` определяет сборку, а столбец `contained_assembly` — части (одна за другой), которые составляют соответствующую сборку. На рис. 6.1 приведена графическая иллюстрация возможного вида самолета и его составляющих частей.

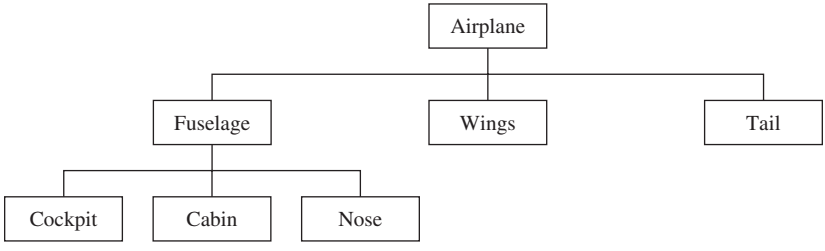


Рис. 6.1. Графическое представление самолета и его составляющих

Таблица `airplane` содержит 11 строк, которые показаны в табл. 6.2. Эти строки вставляются в таблицу `airplane` инструкциями `INSERT` (см. пример 6.80).

Таблица 6.2. *Содержимое таблицы* `airplane`

Airplane	Fuselage	1	10
Airplane	Wings	1	11
Airplane	Tail	1	12
Fuselage	Cockpit	1	13

Таблица 6.2 *(окончание)*

Fuselage	Cabin	1	14
Fuselage	Nose	1	15
Cockpit	NULL	1	13
Cabin	NULL	1	14
Nose	NULL	1	15
Wings	NULL	2	11
Tail	NULL	1	12

В примере 6.81 показано применение предложения `WITH` для определения запроса, который вычисляет общую стоимость каждой сборки.

Пример 6.81. Вычисление общей стоимости каждой сборки

```
USE sample;
WITH list_of_parts(assembly1, quantity, cost) AS
   (SELECT containing_assembly, quantity_contained, unit_cost
      FROM airplane
      WHERE contained_assembly IS NULL
   UNION ALL
   SELECT a.containing_assembly, a.quantity_contained,
   CAST(1.quantity*1.cost AS DECIMAL(6,2))
   FROM list_of_parts 1, airplane a
   WHERE 1.assembly1 = a.contained_assembly)
SELECT * FROM list_of_parts;
```

Предложение `WITH` определяет список ОТВ с именем `list_of_parts`, состоящий из трех столбцов: `assembly`, `quantity` и `cost`. Первая инструкция `SELECT` в примере 6.81 вызывается только один раз, чтобы сохранить результаты первого шага процесса рекурсии.

Инструкция `SELECT` в последней строке примера 6.81 отображает следующий результат:

```
Assembly        quantity      costs
Cockpit         1             13.00
Cabin           1             14.00
Nose            1             16.500
Wings           2             11.00
Tail            1             12.00
Airplane        1             12.00
Airplane        1             22.00
Fuselage        1             16.500
Airplane        1             16.500
```

```
Fuselage        1            14.00
Airplane        1            14.00
Fuselage        1            13.00
Airplane        1            13.00
```

Первые пять строчек этого результата являются результирующим набором первого вызова опорного члена запроса в примере 6.81, а все остальные строчки — результатом рекурсивного члена (вторая часть) запроса в этом примере. Рекурсивный член запроса вызывается дважды: первый раз для сборки фюзеляжа (fuselage), а второй раз для всего самолета (airplane).

Запрос в примере 6.82 вычисляет стоимость каждой сборки со всеми ее составляющими.

Пример 6.82. Вычисление стоимости каждой сборки с ее составляющими

```
USE sample;
WITH list_of_parts(assembly, quantity, cost) AS
   (SELECT containing_assembly, quantity_contained, unit_cost
   FROM airplane
   WHERE contained_assembly IS NULL
UNION ALL
SELECT a.containing_assembly, a.quantity_contained,
   CAST(1.quantity*1.cost AS DECIMAL(6,2))
   FROM list_of_parts 1, airplane a
   WHERE 1.assembly = a.contained_assembly)
SELECT assembly, SUM(quantity) parts, SUM(cost) sum_cost
   FROM  list_of_parts
   GROUP BY assembly;
```

Результат выполнения этого запроса дает следующий результат:

Assembly	parts	sum_cost
Airplane	5	76.00
Cabin	1	14.00
Cockpit	1	13.00
Fuselage	3	42.00
Nose	1	16.500
Tail	1	12.00
Wings	2	11.00

В рекурсивных запросах на ОТВ налагается несколько следующих ограничений:

- определение ОТВ должно содержать, по крайней мере, две инструкции `SELECT` (опорный член и рекурсивный член), объединенные оператором `UNION ALL`;

- опорный член и рекурсивный член должны иметь одинаковое количество столбцов (это является прямым следствием использования оператора `UNION ALL`);
- столбцы в рекурсивном члене должны иметь такой же тип данных, как и соответствующие столбцы в опорном члене;
- предложение `FROM` рекурсивного члена должно ссылаться на имя ОТВ только один раз;
- определение рекурсивного члена не может содержать следующие параметры: `SELECT DISTINCT`, `GROUP BY`, `HAVING`, агрегатные функции, `TOP` и подзапросы. Кроме этого, единственным типом операции соединения, разрешенной в определении запроса, является внутреннее соединение.

Резюме

В этой главе мы рассмотрели все возможности инструкции `SELECT`, касающиеся выборки данных из одной или нескольких таблиц. Каждая инструкция `SELECT`, которая извлекает данные из таблицы, должна содержать как минимум список столбцов и предложение `FROM`. Предложение `FROM` указывает таблицу или таблицы, из которых выбираются данные. Наиболее важным необязательным предложением является предложение `WHERE`, содержащее одно или несколько условий, объединяемых логическими операторами `AND`, `OR` или `NOT`. Таким образом, условия в предложении `WHERE` накладывают ограничения на выбираемые строки.

Упражнения

Упражнение 6.1

Выполните выборку всех строк из таблицы `works_on`.

Упражнение 6.2

Выполните выборку табельных номеров всех сотрудников с должностью клерк (`clerk`).

Упражнение 6.3

Выполните выборку табельных номеров всех сотрудников, которые работают над проектом `p2` и чей табельный номер меньше, чем 10 000. Решите эту задачу, используя два различных, но эквивалентных запроса с инструкцией `SELECT`.

Упражнение 6.4

Выполните выборку табельных номеров всех сотрудников, которые не приступили к работе над проектом в 2007 г.

Упражнение 6.5

Выполните выборку табельных номеров всех сотрудников проекта `p1` с ведущими должностями (т. е. аналитик — `analyst` и менеджер — `manager`).

Упражнение 6.6

Выполните выборку всех сотрудников проекта `p2`, чья должность еще не определена.

Упражнение 6.7

Выполните выборку табельных номеров и фамилий сотрудников, чьи имена содержат две буквы "t".

Упражнение 6.8

Выполните выборку табельных номеров и имен всех сотрудников, у которых вторая буква фамилии "o" или "a" (буквы английские) и последние буквы фамилии "es".

Упражнение 6.9

Выполните выборку табельных номеров сотрудников, чьи отделы расположены в Сиэтле (`Seattle`).

Упражнение 6.10

Выполните выборку фамилий и имен сотрудников, которые приступили к работе над проектами 4 января 2007 г.

Упражнение 6.11

Сгруппируйте все отделы по их местонахождению.

Упражнение 6.12

Объясните разницу между предложениями `DISTINCT` и `GROUP BY`.

Упражнение 6.13

Как предложение `GROUP BY` обрабатывает значения `NULL`? Подобна ли эта обработка обычной обработке этих значений?

Упражнение 6.14

Объясните разницу между агрегатными функциями `COUNT(*)` и `COUNT(column)`.

Упражнение 6.15

Выполните выборку наибольшего табельного номера сотрудника.

Упражнение 6.16

Выполните выборку должностей, занимаемых больше, чем двумя сотрудниками.

Упражнение 6.17

Выполните выборку табельных номеров сотрудников, которые или имеют должность клерк `clerk`, или работают в отделе `d3`.

Упражнение 6.18

Объясните, почему следующий запрос неправильный.

```
SELECT project_name
  FROM project
   WHERE project_no =
      (SELECT project_no FROM works_on WHERE Job = 'Clerk')
```

Исправьте синтаксис запроса.

Упражнение 6.19

Каково практическое применение временных таблиц?

Упражнение 6.20

Объясните разницу между глобальными и локальными временными таблицами.

ПРИМЕЧАНИЕ

В решениях всех последующих упражнений по операции соединения используйте явный синтаксис.

Упражнение 6.21

Создайте следующие соединения таблиц `project` и `works_on`, выполнив:

- естественное соединение;
- декартово произведение.

Упражнение 6.22

Сколько условий соединения необходимо для соединения в запросе *n* таблиц?

Упражнение 6.23

Выполните выборку табельного номера сотрудника и должности для всех сотрудников, работающих над проектом `Gemini`.

Упражнение 6.24

Выполните выборку имен и фамилий всех сотрудников, работающих в отделе `Research` или `Accounting`.

Упражнение 6.25

Выполните выборку всех дат начала работы для всех клерков (`clerk`), работающих в отделе `d1`.

Упражнение 6.26

Выполните выборку всех проектов, над которыми работают двое или больше сотрудников с должностью клерк (`clerk`).

Упражнение 6.27

Выполните выборку имен и фамилий сотрудников, которые имеют должность менеджер (`manager`) и работают над проектом `Mercury`.

Упражнение 6.28

Выполните выборку имен и фамилий всех сотрудников, которые начали работать над проектом одновременно, по крайней мере, еще с одним другим сотрудником.

Упражнение 6.29

Выполните выборку табельных номеров сотрудников, которые живут в том же городе, где находится их отдел. (Используйте расширенную таблицу `enployee_enh` базы данных `sample`.)

Упражнение 6.30

Выполните выборку табельных номеров всех сотрудников, работающих в отделе маркетинга `marketing`. Создайте два равнозначных запроса, используя:

- оператор соединения;
- связанный подзапрос.

Глава 7

Модифицирование содержимого таблиц

- **Инструкция *INSERT***
- **Инструкция *UPDATE***
- **Инструкция *DELETE***
- **Другие инструкции и предложения Transact-SQL для модификации таблиц**

Кроме инструкции SELECT, которая была рассмотрена в *главе 6*, язык манипуляции данными DML (Data Manipulation Language) содержит три другие инструкции: INSERT, UPDATE и DELETE. Подобно инструкции SELECT эти три инструкции оперируют либо таблицами, либо представлениями. В этой главе эти инструкции рассматриваются применительно к таблицам, иллюстрируя их использование на примерах. Кроме этого, рассматриваются две другие инструкции: TRUNCATE TABLE и MERGE, а также предложение OUTPUT. В то время как инструкция TRUNCATE TABLE является расширением Transact-SQL стандарта SQL, инструкция MERGE является стандартной возможностью SQL Server. Предложение OUTPUT позволяет явно отображать вставленные или обновленные строки.

Инструкция *INSERT*

Инструкция INSERT вставляет строки (или части строк) в таблицу. Существует две разные формы этой инструкции:

```
INSERT [INTO] tab_name [(col_list)]
   DEFAULT VALUES | VALUES ({DEFAULT NULL expression} [ ,...n])

INSERT INTO tab_name | view_name [(col_list)]
   {select_statement | execute_statement}
```

Первая форма инструкции позволяет вставить в таблицу одну строку (или часть ее). А вторая форма инструкции INSERT позволяет вставить в таблицу результирующий набор инструкции SELECT или хранимой процедуры, выполняемой посредством инструкции EXECUTE. Хранимая процедура должна возвращать данные для вставки в таблицу. Применяемая с инструкцией INSERT инструкция SELECT может выбирать значения из другой или той же самой таблицы, в которую вставляются данные, при условии совместимости типов данных соответствующих столбцов.

Для обеих форм тип данных каждого вставляемого значения должен быть совместимым с типом данных соответствующего столбца таблицы. Все строковые и временные данные должны быть заключены в кавычки; численные значения заключать в кавычки не требуется.

Вставка одной строки

Для обеих форм инструкции INSERT явное указание списка столбцов не является обязательным. Отсутствие списка столбцов равнозначно указанию всех столбцов таблицы.

Параметр DEFAULT VALUES вставляет значения по умолчанию для всех столбцов. В столбцы с типом данных TIMESTAMP или свойством IDENTITY по умолчанию вставляются значения, автоматически создаваемые системой. Для столбцов других типов данных вставляется соответствующее ненулевое значение по умолчанию, если таково имеется, или NULL в противном случае. Если для столбца значения NULL не разрешены и для него не определено значение по умолчанию, выполнение инструкции INSERT завершается ошибкой и выводится соответствующее сообщение.

В примерах 7.1—7.4 вставляются строки в четыре таблицы базы данных sample, демонстрируя использование инструкции INSERT для вставки небольшого объема данных в базу данных.

Пример 7.1. Загрузка данных в таблицу employee

```
USE sample;
INSERT INTO employee VALUES (25348, 'Matthew', 'Smith','d3');
INSERT INTO employee VALUES (10102, 'Ann', 'Jones','d3');
INSERT INTO employee VALUES (18316, 'John', 'Barrimore', 'd1');
INSERT INTO employee VALUES (29346, 'James', 'James', 'd2');
INSERT INTO employee VALUES (9031, 'Elsa', 'Bertoni', 'd2');
INSERT INTO employee VALUES (2581, 'Elke' , 'Hansel', 'd2') ;
INSERT INTO employee VALUES (28559, 'Sybill', 'Moser', 'd1');
```

Пример 7.2. Загрузка данных в таблицу department

```
USE sample;
INSERT INTO department VALUES ('d1', 'Research', 'Dallas');
INSERT INTO department VALUES ('d2', 'Accounting', 'Seattle');
INSERT INTO department VALUES ('d3', 'Marketing', 'Dallas');
```

Пример 7.3. Загрузка данных в таблицу `project`

```
USE sample;
INSERT INTO project VALUES ('p1', 'Apollo', 120000.00);
INSERT INTO project VALUES ('p2', 'Gemini', 95000.00);
INSERT INTO project VALUES ('p3', 'Mercury', 186500.00);
```

Пример 7.4. Загрузка данных в таблицу `works_on`

```
USE sample;
INSERT INTO works_on VALUES (10102,'p1', 'Analyst', '2006.10.1');
INSERT INTO works_on VALUES (10102, 'p3', 'Manager', '2008.1.1');
INSERT INTO works_on VALUES (25348, 'p2', 'Clerk', '2007.2.15');
INSERT INTO works_on VALUES (18316, 'p2', NULL, '2007.6.1');
INSERT INTO works_on VALUES (29346, 'p2', NULL, '2006.12.15');
INSERT INTO works_on VALUES (2581, 'p3', 'Analyst', '2007.10.15');
INSERT INTO works_on VALUES (9031, 'p1', 'Manager', '2007.4.15');
INSERT INTO works_on VALUES (28559, 'p1', 'NULL', '2007.8.1');
INSERT INTO works_on VALUES (28559, 'p2', 'Clerk', '2008.2.1');
INSERT INTO works_on VALUES (9031, 'p3', 'Clerk', '2006.11.15');
INSERT INTO works_on VALUES (29346, 'p1','Clerk', '2007.1.4');
```

Существует два разных способа вставки значений в новую строку, которые показаны в примерах 7.5—7.7.

Пример 7.5. Явная вставка значения `NULL`

```
USE sample;
INSERT INTO employee VALUES (15201, 'Dave', 'Davis', NULL);
```

Инструкция `INSERT` в примере 7.5 соответствует инструкциям `INSERT` в примерах 7.1—7.4. Явное использование ключевого слова `NULL` вставляет значение `NULL` в соответствующий столбец.

Чтобы вставить значения в некоторые (но не во все) столбцы таблицы, обычно необходимо явно указать эти столбцы. Не указанные столбцы должны или разрешать значения `NULL`, или для них должно быть определено значение по умолчанию.

Пример 7.6. Вставка данных в часть столбцов таблицы

```
USE sample;
INSERT INTO employee (emp_no, emp_fname, emp_lname)
   VALUES (15201, 'Dave', 'Davis');
```

Примеры 7.5 и 7.6 равнозначны. В таблице `employee` единственным столбцом, разрешающим значения `NULL`, является столбец `dept_no`, а для всех прочих столбцов это значение было запрещено предложением `NOT NULL` в инструкции `CREATE TABLE`.

Порядок значений в предложении VALUE инструкции INSERT может отличаться от порядка, указанного в инструкции CREATE TABLE. В таком случае их порядок должен совпадать с порядком, в котором соответствующие столбцы перечислены в списке столбцов.

Пример 7.7. Вставка данных в порядке, отличающемся от исходного

```
USE sample;
INSERT INTO employee (emp_lname, emp_fname, dept_no, emp_no)
   VALUES ('Davis', 'Dave', 'd1', 15201);
```

Вставка нескольких строк

Вторая форма инструкции INSERT вставляет в таблицу одну или несколько строк, выбранных подзапросом. В примере 7.8 показана вставка строк в таблицу, используя вторую форму инструкции INSERT. В данном случае выполняется запрос по выборке номеров и имен отделов, расположенных в Далласе (Dallas), и загрузка полученного результирующего набора в новую таблицу, созданную ранее.

Пример 7.8. Вставка в таблицу строк, выбранных подзапросом

```
USE sample;
CREATE TABLE dallas_dept
   (dept_no CHAR(4) NOT NULL,
    dept_name CHAR(20) NOT NULL);

INSERT INTO dallas_dept (dept_no, dept_name)
   SELECT dept_no, dept_name
      FROM department
      WHERE location = 'Dallas';
```

Создаваемая в примере 7.8 новая таблица dallas_dept имеет те же столбцы, что и существующая таблица department, за исключением отсутствующего столбца location. Подзапрос в инструкции INSERT выбирает в таблице department все строки, для которых значение столбца location равно 'Dallas', которые затем вставляются в созданную в начале запроса новую таблицу.

Содержимое этой таблицы можно просмотреть посредством следующей инструкции SELECT:

```
SELECT *
   FROM dallas_dept;
```

Результат выполнения этого запроса:

dept_no	dept_name
d1	Research
d3	Marketing

В примере 7.9 показан еще один способ вставки строк в таблицу, используя вторую форму инструкции INSERT. В данном случае выполняется запрос на выборку табельных номеров, номеров проектов и дат начала работы над проектом для всех сотрудников с должностью клерк (clerks), которые работают над проектом p2 с последующей загрузкой полученного результирующего набора в новую таблицу, создаваемую в начале запроса.

Пример 7.9. Вставка в таблицу строк, выбранных подзапросом с составным условием

```
USE sample;
CREATE TABLE clerk_t
   (emp_no INT NOT NULL,
    project_no CHAR(4),
    enter_date DATE);

INSERT INTO clerk_t (emp_no, project_no, enter_date)
   SELECT emp_no, project_no, enter_date
      FROM works_on
      WHERE job = 'Clerk'
      AND project_no = 'p2';
```

Результатом выполнения этого запроса должна быть новая таблица clerk_t, содержащая следующие данные:

emp_no	project_no	enter_date
25348	p2	2007-02-15
28559	p2	2008-02-01

Перед вставкой строк с помощью инструкции INSERT таблицы dallas_dept и clerk_t (в примерах 7.8 и 7.9 соответственно) были пустыми. Если же таблица уже существовала и содержала строки с данными, то к ней были бы добавлены новые строки.

ПРИМЕЧАНИЕ

В примере 7.9 инструкции CREATE TABLE и INSERT можно заменить инструкцией SELECT с предложением INTO (см. пример 6.48).

Конструкторы значений таблицы и инструкция *INSERT*

Конструктор значений таблицы или строки (table (row) value constructor) позволяет вставить в таблицу несколько записей (строк) посредством инструкции языка DML,

такой как, например, INSERT или UPDATE. В примере 7.10 показана вставка в таблицу нескольких строк, используя такой конструктор с помощью инструкции INSERT.

Пример 7.10. Вставка строк посредством конструктора значений таблицы

```
USE sample;
   INSERT INTO department VALUES
      ('d4', 'Human Resources', 'Chicago'),
      ('d.5', 'Distribution', 'New Orleans'),
      ('d6', 'Sales', 'Chicago');
```

В примере 7.10 инструкция INSERT одновременно вставляет три строки в таблицу department, используя конструктор значений таблицы. Как можно видеть, синтаксис этого конструктора довольно простой. Для вставки в таблицу строк с данными посредством конструктора значений таблицы нужно в круглых скобках перечислить значения каждой строки, разделяя как значения каждого списка, так и отдельные списки запятыми.

Инструкция *UPDATE*

Инструкция UPDATE используется для модифицирования строк таблицы. Эта инструкция имеет следующую общую форму:

```
UPDATE tab_name
   {SET column_1 = {expression | DEFAULT | NULL} [,...n]
      [FROM tab_name1 [,...n]]
      [WHERE condition]
```

Строки таблицы tab_name выбираются для изменения в соответствии с условием в предложении WHERE. Значения столбцов каждой модифицируемой строки изменяются с помощью предложения SET инструкции UPDATE, которое соответствующему столбцу присваивает выражение (обычно) или константу. Если предложение WHERE отсутствует, то инструкция UPDATE модифицирует все строки таблицы. (Предложение FROM рассматривается далее в этой главе.)

ПРИМЕЧАНИЕ

С помощью инструкции UPDATE данные можно модифицировать только в одной таблице.

В примере 7.11 инструкция UPDATE изменяет всего лишь одну строку таблицы works_on, поскольку комбинация столбцов emp_no и project_no является первичным ключом этой таблицы и, следственно, она однозначна. В данном примере изменяется должность сотрудника, значение которого было ранее неизвестно или имело значение NULL.

Пример 7.11. Присвоение сотруднику с табельным номером 18316, который работает над проектом p2, должности менеджера (manager)

```
USE sample;
UPDATE works_on
   SET job = 'Manager'
      WHERE emp_no = 18316
      AND project_no = 'p2';
```

В примере 7.12 значения строкам таблицы присваиваются посредством выражения. Запрос пересчитывает бюджеты всех проектов с долларов на английские фунты стерлингов. Валютный курс: 0.51£ за $1.

Пример 7.12. Присвоение значений посредством выражения

```
USE sample;
UPDATE project
   SET budget = budget*0.51;
```

В данном примере изменяются все строки таблицы project, поскольку в запросе отсутствует предложение WHERE. Измененное содержимое таблицы project можно просмотреть с помощью следующего запроса:

```
USE sample;
   SELECT *
      FROM project;
```

Результат выполнения этого запроса:

project_no	project_name	budget
p1	Apollo	61200
p2	Gemini	48450
p3	Mercury	95115

В примере 7.13 в предложении WHERE инструкции UPDATE используется вложенный запрос. Поскольку применяется оператор IN, то этот запрос может возвратить более одной строки.

Пример 7.13. В связи с болезнью сотрудницы Jones во всех ее проектах в столбце ее должности присваивается значение NULL

```
USE sample;
UPDATE works_on
   SET job = NULL
   WHERE emp_no IN (
   SELECT emp_no
      FROM employee
      WHERE emp_lname = 'Jones');
```

Запрос в примере 7.13 можно также выполнить посредством предложения FROM инструкции UPDATE. В предложении FROM указываются имена таблиц, которые обрабатываются инструкцией UPDATE. Все эти таблицы должны быть в дальнейшем соединены. Применение предложения FROM показано в примере 7.14. Логически, этот пример идентичен предыдущему примеру 7.13.

ПРИМЕЧАНИЕ

В языке Transact-SQL предложение FROM является расширением стандарта ANSI SQL.

Пример 7.14. Использование предложения FROM для выполнения аналогичного запроса из примера 7.13

```
USE sample;
UPDATE works_on
   SET job = NULL
      FROM works_on, employee
      WHERE emp_lname = 'Jones'
      AND works_on.emp_no = employee.emp_no;
```

В примере 7.15 показано использование выражения CASE в инструкции UPDATE. (Подробное рассмотрение этого выражения см. в *главе 6*.) В данном примере нужно увеличить бюджет всех проектов на определенное число процентов (20, 10 или 5), в зависимости от исходной суммы бюджета: чем меньше бюджет, тем больше должно быть его процентное увеличение.

Пример 7.15. Использование выражения CASE в инструкции UPDATE

```
USE sample;
UPDATE project
   SET budget = CASE
      WHEN budget >0 and budget < 100000 THEN budget*1.2
      WHEN budget >= 100000 and budget < 200000 THEN budget*1.1
      ELSE budget*1.05
      END
```

Инструкция *DELETE*

Инструкция DELETE удаляет строки из таблицы. Подобно инструкции INSERT, эта инструкция также имеет две различные формы:

```
DELETE FROM table_name
   [WHERE predicate];
```

```
DELETE table_name
   FROM table_name [,...n]
   [WHERE condition];
```

Удаляются все строки, которые удовлетворяют условие в предложении WHERE. Явно перечислять столбцы в инструкции DELETE не то чтобы нет необходимости, а даже не разрешается, поскольку эта инструкция оперирует строками, а не столбцами.

Использование первой формы инструкции DELETE показано в примере 7.16.

Пример 7.16. Удаление из таблицы works_on всех сотрудников, имеющих должность менеджера (manager)

```
USE sample;
DELETE FROM works_on
   WHERE job = 'Manager';
```

Предложение WHERE инструкции DELETE может содержать вложенный запрос, как это показано в примере 7.17.

Пример 7.17. Поскольку сотрудница Moser уволилась, из базы данных удаляются все записи, связанные с ней

```
USE sample;
DELETE FROM works_on
   WHERE emp_no IN
   (SELECT emp_no
      FROM employee
      WHERE emp_lname = 'Moser');

DELETE FROM employee
   WHERE emp_lname = 'Moser';
```

Запрос из примера 7.17 можно также выполнить с помощью предложения FROM, как это показано в примере 7.18. В данном случае семантика этого предложения такая же, как и предложения FROM в инструкции UPDATE.

Пример 7.18. Удаление записей из всей базы данных посредством предложения FROM

```
USE sample;
DELETE works_on
   FROM works_on, employee
   WHERE works_on.emp_no = employee.emp_no
   AND emp_lname = 'Moser';

DELETE FROM employee
   WHERE emp_lname = 'Moser';
```

Использование предложения WHERE в инструкции DELETE не является обязательным. Если это предложение отсутствует, то из таблицы удаляются все строки, как это показано в примере 7.19.

Пример 7.19. Удаление всех строк таблицы

```
USE sample;
DELETE FROM works_on;
```

ПРИМЕЧАНИЕ

Инструкции DELETE и DROP TABLE существенно отличаются друг от друга. Инструкция DELETE удаляет (частично или полностью) содержимое таблицы, тогда как инструкция DROP TABLE удаляет как содержимое, так и схему таблицы. Таким образом, после удаления всех строк посредством инструкции DELETE таблица продолжает существовать в базе данных, а после выполнения инструкции DROP TABLE таблица больше не существует.

Другие инструкции и предложения Transact-SQL для модификации таблиц

Сервер SQL Server поддерживает следующие дополнительные инструкции и предложения для модификации таблиц:

- инструкцию TRUNCATE TABLE;
- инструкцию MERGE;
- предложение OUTPUT.

Эти инструкции и предложение рассматриваются в последующих подразделах главы.

Инструкция *TRUNCATE TABLE*

Инструкция TRUNCATE TABLE является более быстрой версией инструкции DELETE без предложения WHERE. Эта инструкция удаляет все строки таблицы более быстро, чем инструкция DELETE, поскольку она удаляет содержимое постранично, тогда как инструкция DELETE делает это построчно.

ПРИМЕЧАНИЕ

Инструкция TRUNCATE TABLE является расширением Transact-SQL стандарта SQL.

Инструкция TRUNCATE TABLE имеет следующий синтаксис:

```
TRUNCATE TABLE table_name
```

ПРИМЕЧАНИЕ

Для удаления всех строк таблицы следует использовать инструкцию TRUNCATE TABLE. Эта инструкция выполняется значительно быстрее, чем инструкция DELETE, поскольку для нее ведется минимальное протоколирование, и в процессе ее выполнения в журнале делается лишь несколько записей. Тема протоколирования более подробно рассматривается в *главе 13*.

Инструкция *MERGE*

Инструкция MERGE объединяет последовательность инструкций INSERT, UPDATE и DELETE в одну элементарную инструкцию, в зависимости от существования записи (строки). Иными словами, можно синхронизировать две разные таблицы, чтобы модифицировать содержимое таблицы назначения в зависимости от различий, обнаруженных в таблице-источнике.

Основной областью применения для инструкции MERGE является среда хранилищ данных (см. *главу 23*), где таблицы необходимо периодически обновлять, чтобы отражать новые данные, прибывающие с систем оперативной обработки транзакций *OLTP* (On-Line Transaction Processing). Эти данные могут содержать изменения существующих строк таблиц и/или новый строки, которые нужно вставить в таблицы. Если строка в новых данных соответствует записи, которая уже имеется в таблице, выполняется инструкция UPDATE или DELETE. В противном случае выполняется инструкция INSERT.

Альтернативно, вместо инструкции MERGE можно использовать последовательность инструкций INSERT, UPDATE и DELETE, в которых для каждой строки решается, какую операцию выполнять: вставку, удаление или обновление. Но этот подход имеет значительный недостаток, связанный с производительностью: в нем требуется выполнять несколько проходов по данным, а данные обрабатываются по принципу "запись за записью".

Использование инструкции MERGE показано в примерах 7.20 и 7.21. В примере 7.20 создается таблица bonus, содержащая одну строку со значениями p1 и 100.

Пример 7.20. Создание однострочной таблицы

```
USE sample;
CREATE TABLE bonus
   (pr_no CHAR(4),
     bonus SMALLINT DEFAULT 100);
INSERT INTO bonus (pr_no) VALUES ('p1');
```

В созданную в примере 7.20 таблицу bonus в примере 7.21 будут вставлены новые данные из другой таблицы.

Пример 7.21. Вставка новых строк данных в таблицу bonus

```
USE sample;
MERGE INTO bonus B
   USING (SELECT project_no, budget
         FROM project) E
      ON (B.pr_no = E.project_no)
      WHEN MATCHED THEN
         UPDATE SET B.bonus = E.budget * 0.1
      WHEN NOT MATCHED THEN
         INSERT (pr_no, bonus)
            VALUES (E.project_no, E.budget * 0.05);
```

В примере 7.21 инструкция MERGE в зависимости от значений в столбце pr_no модифицирует данные в таблице bonus. Если в столбце pro_no таблицы bonus имеется значение из столбца project_no таблицы project, то выполняется ветвление MATCHED и существующее значение будет обновлено. В противном случае выполняется ветвление NON MATCHED и соответствующая инструкция INSERT вставляет новые строки в таблицу bonus.

После выполнения этой инструкции содержимое таблицы bonus будет выглядеть следующим образом:

pr_no	bonus
P1	12000
p2	4750
p3	9325

По этому результату можно видеть, что значение столбца bonus представляет 10% исходного значения при использовании инструкции UPDATE и 5% при использовании инструкции INSERT.

Предложение *OUTPUT*

По умолчанию единым видимым результатом выполнения инструкции INSERT, UPDATE или DELETE является только сообщение о количестве модифицированных строк, например "3 rows deleted" (удалены 3 строки). Если такой видимый результат не удовлетворяет вашим требованиям, то можно использовать предложение OUTPUT, которое выводит модифицированные, вставленные или удаленные строки.

ПРИМЕЧАНИЕ

Предложение OUTPUT также применимо с инструкцией MERGE, для которой оно выводит все модифицированные строки в виде таблицы (см. примеры 7.25 и 7.26).

Результаты выполненных операций соответствующих инструкций предложение OUTPUT выводит в таблицах inserted и deleted (см. *главу 14*). Кроме этого, чтобы

заполнить таблицы, в предложении OUTPUT требуется использовать выражение INTO. Поэтому для сохранения результата используется табличная переменная.

В примере 7.22 показано использование инструкции OUTPUT с инструкцией DELETE.

Пример 7.22. Применение инструкции OUTPUT

```
USE sample;
DECLARE @del_table TABLE (emp_no INT, emp_lname CHAR(20));
DELETE employee
OUTPUT DELETED.emp_no, DELETED.emp_lname INTO @del_table
WHERE emp_no > 15000;
SELECT * FROM @del_table
```

При условии, что содержимое таблицы находится в исходном состоянии, выполнение запроса в примере 7.22 дает следующий результат:

emp_no	emp_lname
25348	Smith
18316	Barrimore
29346	James
28559	Moser

В примере 7.22 сначала объявляется табличная переменная @del_table с двумя столбцами: emp_ no и emp_lname. (Переменные подробно рассматриваются в *главе 8*.) В этой таблице будут сохранены удаленные строки. Синтаксис инструкции DELETE расширен предложением OUTPUT:

```
OUTPUT DELETED.emp_no, DELETED.emp_lname INTO @del_table
```

Посредством этого предложения система сохраняет удаленные строки в таблице deleted, содержимое которой потом копируется в переменную @del_table.

В примере 7.23 показано использование предложения OUTPUT в инструкции UPDATE.

Пример 7.23. Инструкция UPDATE с предложением OUTPUT

```
USE sample;
DECLARE @update_table TABLE
   (emp_no INT, project_no CHAR(20),old_job CHAR(20),new_job CHAR(20));
UPDATE works_on
SET job = NULL
OUTPUT DELETED.emp_no, DELETED.project_no,
   DELETED.job, INSERTED.job INTO @update_table
WHERE job = 'Clerk';
SELECT * FROM @update_table
```

Результат выполнения этого запроса:

emp_no	project_no	Oldjob	newjob
25348	p2	Clerk	NULL
28559	p2	Clerk	NULL
9031	p3	Clerk	NULL
29346	pi	Clerk	NULL

В следующих примерах приводится использование предложения OUTPUT с инструкцией MERGE.

Допустим, что маркетинговый отдел решил предоставить клиентам скидку в 20% на все велосипеды ("Bikes") стоимостью более $500. В примере 7.24 инструкция SELECT выбирает все продукты стоимостью свыше $500 и вставляет их во временную таблицу temp_PriceList. А последующая инструкция UPDATE выбирает из всех отобранных товаров велосипеды и уменьшает их цену.

Пример 7.24. Уменьшение цены велосипедов на 20%

```
USE AdventureWorks;
SELECT ProductID, Product.Name as ProductName, ListPrice
INTO temp_PriceList
FROM Production.Product
WHERE ListPrice > 500;

UPDATE temp_PriceList
   SET ListPrice = ListPrice * 0.8
      WHERE ProductID IN (SELECT ProductID
      FROM AdventureWorks.Production.Product
         WHERE ProductSubcategoryID IN (SELECT ProductCategoryID
         FROM AdventureWorks.Production.ProductSubcategory
            WHERE ProductCategoryID IN (SELECT ProductCategoryID
            FROM AdventureWorks.Production.ProductCategory
               WHERE Name = 'Bikes')));
```

В примере 7.25 инструкция CREATE TABLE создает новую таблицу, temp_Difference, в которой будет сохраняться результирующий набор инструкции MERGE. После этого инструкция MERGE сравнивает полный список продуктов с новым списком (находящимся в таблице temp_PriceList) и вставляет модифицированные цены для всех велосипедов, используя предложение UPDATE SET. (Кроме вставки новых цен для всех велосипедов, эта инструкция также изменяет значения столбца ModifiedDate для всех продуктов на текущую дату.) Предложение OUTPUT в примере 7.25 записывает старые и новые цены во временную таблицу temp_Difference. Это в дальнейшем (при необходимости) позволит подсчитать агрегатные различия.

Пример 7.25. Инструкция MERGE с предложением UPDATE

```
USE AdventureWorks;
CREATE TABLE temp_Difference
   (old DEC (10,2), new DEC(10,2));
GO
MERGE INTO Production.Product
USING temp_PriceList ON Product.ProductID = temp_PriceList.ProductID
WHEN MATCHED AND Product.ListPrice <> temp_PriceList.ListPrice THEN
UPDATE SET ListPrice = temp_PriceList.ListPrice, ModifiedDate = GETDATE()
WHEN NOT MATCHED BY SOURCE THEN
UPDATE SET ModifiedDate = GETDATE()
OUTPUT DELETED.ListPrice, INSERTED.ListPrice INTO temp_Difference1;
```

В примере 7.26 показано вычисление общих различий, полученных в результате предыдущих модификаций.

Пример 7.26. Вычисление общих различий

```
USE AdventureWorks;
SELECT SUM(old) — SUM(new) AS diff
FROM AdventureWorks.dbo.temp_Difference;
```

Результат выполнения этого запроса:

```
Diff
10773.60
```

Резюме

По большому счету, для модификации таблиц применяются только три инструкции SQL: INSERT, UPDATE и DELETE. Эти инструкции являются общими, поскольку для всех типов вставки строк используется только инструкция INSERT, для всех типов модификации строк — только инструкция UPDATE, а для всех типов удаления строк — только инструкция DELETE.

Нестандартная инструкция TRUNCATE TABLE является всего лишь другой формой инструкции DELETE, с той разницей, что строки удаляются быстрее, чем с использованием инструкции DELETE. Инструкция MERGE — это, по сути, инструкция "UPSERT" (UPDATE + INSERT), совмещающая инструкции UPDATE и INSERT.

В *главах 5—7* мы познакомились с инструкциями SQL, которые принадлежат к языкам DDL и DML. Большинство из этих инструкций можно сгруппировать вместе для создания последовательности инструкций Transact-SQL. Такая последовательность является основой для *хранимых процедур*, которые рассмотрены в следующей главе.

Упражнения

Упражнение 7.1

Вставьте данные для новой сотрудницы по имени `Julia Long`, табельный номер `11111`. Она еще не назначена в какой-либо отдел.

Упражнение 7.2

Создайте новую таблицу `emp_d1_d2` и загрузите в нее из таблицы `employee` всех сотрудников, работающих в отделах `d1` и `d2`. Создайте два разных, но равнозначных решения.

Упражнение 7.3

Создайте новую таблицу для сотрудников, которые приступили к работе над своими проектами в 2007 г., и загрузите в нее соответствующие строки из таблицы `employee`.

Упражнение 7.4

Измените должности всех менеджеров (`Manager`) в проекте `p1` на клерков (`Clerk`).

Упражнение 7.5

Измените значение бюджетов всех проектов на значение `NULL`.

Упражнение 7.6

Измените должность сотрудника с табельным номером `28559` на менеджера (`Manager`) для всех его проектов.

Упражнение 7.7

Повысьте на 10% бюджет проекта, менеджер которого имеет табельный номер `10102`.

Упражнение 7.8

Измените наименование отдела, в котором работает сотрудник по фамилии `James`, на название `Sales`.

Упражнение 7.9

Измените дату начала работы над проектом для сотрудников, которые работают над проектом `p1` и числятся в отделе `Sales` на 12 декабря 2007 г.

Упражнение 7.10

Удалите все отделы, расположенные в Сиэтле (`Seattle`).

Упражнение 7.11

Проект `p3` выполнен. Удалите всю информацию об этом проекте из базы данных `sample`.

Упражнение 7.12

Удалите всю информацию из таблицы `works_on` для всех сотрудников, которые работают в отделах, расположенных в Далласе (`Dallas`).

Глава 8

Хранимые процедуры и определяемые пользователем функции

- **Процедурные расширения**
- **Хранимые процедуры**
- **Определяемые пользователем функции**

В этой главе мы познакомимся с пакетами и подпрограммами. *Пакет* (batch) — это последовательность инструкций и процедурных расширений языка Transact-SQL. А *подпрограмма* (routine) — это хранимая процедура или *определяемая пользователем функция* (*ОПФ*) (User Defined Function — UDF). В начале главы мы рассмотрим все процедурные расширения, поддерживаемые компонентом Database Engine. Далее обсуждается использование процедурных расширений совместно с инструкциями Transact-SQL для реализации пакетов. Пакет можно сохранить в виде объекта базы данных как хранимую процедуру или ОПФ. Одни хранимые процедуры создаются пользователями, другие же предоставляются разработчиками Microsoft и называются *системными хранимыми процедурами*. В отличие от определяемых пользователем хранимых процедур, ОПФ возвращают значение вызывающему их объекту. Все подпрограммы можно реализовать или на языке Transact-SQL, или на другом языке программирования, таком как C# или Visual Basic. В конце главы представлены табличные значения параметров.

Процедурные расширения

В предыдущих главах мы познакомились с инструкциями Transact-SQL языка описания данных DDL и языка манипуляции данными DML. Большинство этих инструкций можно сгруппировать в пакет. Как упоминалось ранее, *пакет* — это последовательность инструкций Transact-SQL и процедурных расширений, которые от-

правляются системе базы данных для совместного их выполнения. Количество инструкций в пакете ограничивается допустимым размером скомпилированного пакетного объекта. Преимущество пакета над группой отдельных инструкций состоит в том, что одновременное исполнение всех инструкций позволяет получить значительное улучшение производительности.

Существует несколько ограничений на включение разных инструкций языка Transact-SQL в пакет. Наиболее важным из них является то обстоятельство, что если пакет содержит инструкцию описания данных `CREATE VIEW`, `CREATE PROCEDURE` или `CREATE TRIGGER`, то он не может содержать никаких других инструкций. Иными словами, такая инструкция должна быть единственной инструкцией пакета.

ПРИМЕЧАНИЕ

Инструкции языка описания данных разделяются с помощью инструкции `GO`.

Каждое процедурное расширение языка Transact-SQL рассматривается по отдельности в следующих разделах.

Блок инструкций

Блок инструкций может состоять из одной или нескольких инструкций языка Transact-SQL. Каждый блок начинается с инструкции `BEGIN` и заканчивается инструкцией `END`, как это показано далее:

```
BEGIN
   statement_1
   statement_2
   ...
END
```

Блок можно разместить внутри инструкции `IF`, чтобы в зависимости от определенного условия разрешить исполнение одной или нескольких инструкций (см. пример 8.1).

Инструкция *IF*

Инструкция `IF` языка Transact-SQL соответствует одноименной инструкции, поддерживаемой почти всеми языками программирования. Инструкция `IF` выполняет одну или несколько составляющих блок инструкций, *если* логическое выражение, следующее после ключевого слова `IF`, возвращает значение `TRUE` (истина). Если же инструкция `IF` содержит оператор `ELSE`, то при условии, что логическое выражение возвращает значение `FALSE` (ложь), выполняется вторая группа инструкций.

ПРИМЕЧАНИЕ

Перед тем как можно исполнять примеры пакетов и ОПФ в этой главе, необходимо заново создать базу данных `sample`.

Пример 8.1. Исполнение блока инструкций посредством инструкции IF

```
USE sample;
IF (SELECT COUNT(*)
        FROM works_on
        WHERE project_no = 'p1'
        GROUP BY project_no ) > 3
   PRINT 'The number of employees in the project p1 is 4 or more'
   ELSE BEGIN
        PRINT 'The following employees work for the project p1'
        SELECT emp_fname, emp_lname
        FROM employee, works_on
        WHERE employee.emp_no = works_on.emp_no
        AND project_no = 'p1'
   END
```

В примере 8.1 демонстрируется использование блока инструкций внутри инструкции IF. Следующее далее логическое выражение инструкции IF

```
(SELECT COUNT(*)
   FROM works_on
   WHERE project_no = 'p1'
   GROUP BY project_no) > 3
```

возвращает значение TRUE (истина) для базы данных sample. Поэтому будет выполняться инструкция PRINT, входящая в часть инструкции IF. Обратите внимание на то обстоятельство, что в этом примере используется подзапрос, чтобы возвратить число строк (посредством агрегатной функции COUNT), удовлетворяющих условию предложения WHERE (project_no='p1').

Результат выполнения примера 8.1:

```
The number of employees in the project p1 is four or more
```

(В проекте p1 задействовано четыре или больше сотрудников.)

ПРИМЕЧАНИЕ

Оператор ELSE инструкции IF в примере 8.1 содержит две инструкции: PRINT и SELECT. Поэтому для выполнения этих инструкций их необходимо заключить в блок между ключевыми словами BEGIN и END. (Инструкция PRINT является процедурным расширением и возвращает определяемое пользователем сообщение.)

Инструкция *WHILE*

Инструкция WHILE выполняет одну или несколько заключенных в блок инструкций, *на протяжении времени, пока* (while) логическое выражение возвращает значение TRUE (истина). Иными словами, если выражение возвращает TRUE, выполняется инструкция или блок инструкций, после чего снова осуществляется проверка выраже-

ния. Этот процесс повторяется до тех пор, пока выражение не возвратит значение FALSE (ложь).

Блок внутри инструкции WHILE может содержать одну или две необязательных инструкций, применяемых для управления выполнением инструкций внутри блока: BREAK или CONTINUE. Инструкция BREAK останавливает выполнение инструкций внутри блока и начинает исполнение инструкций, следующих сразу же после этого блока. А инструкция CONTINUE останавливает выполнение только текущей инструкции в блоке и начинает выполнять его с самого начала.

В примере 8.2 показано использование инструкции WHILE.

Пример 8.2. Использование инструкции WHILE

```
USE sample;
WHILE (SELECT SUM(budget)
        FROM project) < 500000
   BEGIN
        UPDATE project SET budget = budget*1.1
        IF (SELECT MAX(budget)
           FROM project) > 240000
           BREAK
        ELSE CONTINUE
   END
```

В примере 8.2 бюджеты всех проектов увеличиваются на 10% до тех пор, пока общая сумма бюджетов не превысит $500 000. Но выполнение блока прекратится, даже если общая сумма бюджетов будет меньше $500 000, если только бюджет одного из проектов превысит $240 000.

Результат выполнения примера 8.2:

```
(3 rows affected)
(3 rows affected)
(3 rows affected)
```

ПРИМЕЧАНИЕ

Информационные сообщения о результате выполнения запроса наподобие сообщения, используемого в примере 8.2, можно подавить, применив инструкцию SET NOCOUNT ON.

Локальные переменные

Локальные переменные являются важным процедурным расширением языка Transact-SQL. Они применяются для хранения значений любого типа в пакетах и подпрограммах. Локальными они называются по той причине, что они могут быть использованы только в том пакете, в котором они были объявлены. (Компонент Database Engine также поддерживает глобальные переменные, которые уже были рассмотрены в *главе 4*.)

Все локальные переменные пакета объявляются, используя инструкцию DECLARE. (Синтаксис этой инструкции приводится в примере 8.3.) Определение переменной состоит из имени переменной и ее типа данных. Имена локальных переменных в пакете всегда начинаются с префикса @. Присвоение значений локальной переменной осуществляется:

- используя специальную форму инструкции SELECT;
- используя инструкцию SET;
- непосредственно в инструкции DECLARE посредством знака = (например, @extra_budget MONEY = 1500).

Использование первых двух способов присвоения значения локальным переменным показано в примере 8.3.

Пример 8.3. Присвоение значения локальным переменным

```
USE sample;
DECLARE @avg_budget MONEY, @extra_budget MONEY
   SET @extra_budget = 15000
   SELECT @avg_budget = AVG(budget) FROM project
   IF (SELECT budget
        FROM project
        WHERE project_no='p1') < @avg_budget
   BEGIN
        UPDATE project
             SET budget = budget + @extra_budget
             WHERE project_no ='p1'
        PRINT 'Budget for p1 increased by @extra_budget'
   END
   ELSE PRINT 'Budget for p1 unchanged'
```

Результат выполнения:

```
Budget for p1 increased by @extra_budget
```

(Бюджет проекта p1 увеличен на значение локальной переменной @extra_budget.)

Пакет в примере 8.3 вычисляет среднее значение бюджетов всех проектов и сравнивает полученное значение с бюджетом проекта p1. Если бюджет проекта p1 меньше среднего значения всех бюджетов, его значение увеличивается на величину значения локальной переменной @extra_budget.

Прочие процедурные инструкции

Процедурные расширения языка Transact-SQL также содержат следующие инструкции:

- RETURN;
- GOTO;

- RAISEERROR;
- WAITFOR.

Инструкция RETURN выполняет ту же самую функцию внутри пакета, что и инструкция BREAK внутри блока WHILE. Иными словами, инструкция RETURN останавливает выполнение пакета и начинает исполнение первой инструкции, следующей за пакетом.

Инструкция GOTO передает управление при выполнении пакета инструкции Transact-SQL внутри пакета, обозначенной маркером. Инструкция RAISEERROR выводит определенное пользователем сообщение об ошибке и устанавливает флаг системной ошибки. Номер ошибки в определяемом пользователем сообщении должен быть больше, чем 50 000, т. к. все номера ошибок меньшие или равные 50 000 определены системой и зарезервированы компонентом Database Engine. Значения номеров ошибок сохраняются в глобальной переменной @@error. (Использование инструкции RAISEERROR показано в примере 17.3.)

Инструкция WAITFOR определяет или период времени (с параметром DELAY) или определенное время (с параметром TIME), на протяжении или до которого, соответственно, система должна ожидать, прежде чем исполнять следующую инструкцию пакета. Синтаксис этой инструкции выглядит следующим образом:

```
WAITFOR {DELAY 'time' | TIME 'time' | TIMEOUT 'timeout'}
```

Параметр DELAY указывает системе баз данных ожидать, пока не истечет указанный период времени, а параметр TIME указывает точку во времени, в одном из допустимых форматов, до которой ожидать. Параметр TIMEOUT, за которым следует аргумент *timeout*, задает период времени в миллисекундах, в течение которого надо ожидать прибытия сообщения в очередь. (Использование инструкции WAITFOR показано в примере 13.5.)

Обработка исключений с помощью инструкций *TRY*, *CATCH* и *THROW*

В версиях SQL Server более ранних, чем SQL Server 2005 требовалось наличие кода для обработки ошибок после каждой инструкции Transact-SQL, которая могла бы вызвать ошибку. (Для обработки ошибок можно использовать глобальную переменную @@error, применение которой показано в примере 13.1.) Начиная с версии SQL Server 2005, исключения можно перехватывать для обработки с помощью инструкций TRY и CATCH. В этом разделе сначала объясняется значение понятия "исключение", после чего обсуждается работа этих двух инструкций.

Исключением (exception) называется проблема (обычно ошибка), которая не позволяет продолжать выполнение программы. Выполняющаяся программа не может продолжать исполнение по причине недостаточности информации, необходимой для обработки ошибки в данной части программы. Поэтому задача обработки ошибки передается другой части программы.

Роль инструкции TRY заключается в перехвате исключения. (Поскольку для реализации этого процесса обычно требуется несколько инструкций, то обычно приме-

няется термин "блок TRY", а не "инструкция TRY".) Если же в блоке TRY возникает исключение, компонент системы, называющийся обработчиком исключений, доставляет это исключение для обработки другой части программы. Эта часть программы обозначается ключевым словом CATCH и поэтому называется блоком CATCH.

ПРИМЕЧАНИЕ

Обработка исключений с использованием инструкций TRY и CATCH является общим методом обработки ошибок, применяемым в современных языках программирования, таких как C# и Java.

Обработка исключений с помощью блоков TRY и CATCH предоставляет программисту множество преимуществ, включая следующие:

- исключения предоставляют аккуратный способ определения ошибок без загромождения кода дополнительными инструкциями;
- исключения предоставляют механизм прямой индикации ошибок, вместо использования каких-либо побочных признаков;
- программист может видеть исключения и проверить их в процессе компиляции.

В SQL Server 2012 добавлена еще одна инструкция THROW, имеющая отношение к обработке ошибок. Эта инструкция позволяет вызвать исключение, которое улавливается в блоке обработки исключений. Иными словами, инструкция THROW — это другой механизм возврата, который работает подобно рассмотренной ранее инструкции RAISEERROR.

Использование инструкций TRY, CATCH и THROW для обработки исключений показано в примере 8.4. В данном примере показано, как для обработки исключений оформлять инструкции в пакет и выполнять откат результатов исполнения всей группы инструкций при возникновении ошибки. Для правильного функционирования данного примера требуется наличие ссылочной целостности между таблицами department и employee базы данных sample. По этой причине обе эти таблицы нужно создать, используя ограничения первичного (PRIMARY KEY) и внешнего (FOREIGN KEY) ключей, как это сделано в примере 5.11.

Пример 8.4. Применение инструкций TRY, CATCH и THROW для обработки исключений

```
USE sample;
BEGIN TRY
   BEGIN TRANSACTION
   insert into employee values (11111, 'Ann', 'Smith','d2');
   insert into employee values(22222, 'Matthew', 'Jones','d4');
   --ошибка ссылочной целостности
   insert into employee values(33333, 'John', 'Barrimore', 'd2');
   COMMIT TRANSACTION
   PRINT 'Transaction committed'
END TRY
```

```
BEGIN CATCH
   ROLLBACK
   PRINT 'Transaction rolled back';
   THROW
END CATCH
```

Попытка выполнить пакет, показанный в примере 8.4, будет неудачной, а в результате будет выведено следующее сообщение:

```
Transaction rolled back

Msg 547, Level 16, State 0, Line 4
The INSERT statement conflicted with the FOREIGN KEY constraint
"foreign_emp". The conflict occurred in database "sample",
table "dbo.department", column 'dept_no'.

(Сообщение 547, уровень 16, состояние 0, строка 5
Конфликт инструкции INSERT с ограничением FOREIGN KEY "foreign_emp".
Конфликт произошел в базе данных "sample", таблица "dbo.department", столбец
'dept_no'. Выполнение данной инструкции было прервано.)
```

Выполнение кода в примере 8.4 осуществляется следующим образом. После успешного выполнения первой инструкции `INSERT` попытка исполнения второй инструкции вызывает ошибку нарушения ссылочной целостности. Так как все три инструкции заключены в блок `TRY`, возникает исключение для всего блока и обработчик исключений начинает исполнение блока `CATCH`. Выполнение кода в этом блоке осуществляет откат исполнения всех инструкций в блоке `TRY` и выводит соответствующее сообщение. После этого инструкция `THROW` возвращает управление исполнением вызывающему объекту. Вследствие всего этого содержимое таблицы `employee` не будет изменено.

ПРИМЕЧАНИЕ

Инструкции Transact-SQL `BEGIN TRANSACTION`, `COMMIT TRANSACTION` и `ROLLBACK` начинают, фиксируют и выполняют откат транзакций соответственно. Предмет транзакций, в общем, и эти инструкции, в частности, подробно рассмотрены в *главе 13*.

В примере 8.5 показан пакет, поддерживающий создание страниц на стороне сервера (см. *главу 6*).

Пример 8.5. Использование локальных переменных для создания страниц на стороне сервера

```
USE AdventureWorks2012;
DECLARE
   SPageSize TINYINT = 20,
   @CurrentPage INT = 4;
SELECT BusinessEntitylD, JobTitle, BirthDate
   FROM HumanResource s.Employee
   WHERE Gender = 'F'
```

```
ORDER BY JobTitle
OFFSET (@PageSize * (SCurrentPage — 1)) ROWS
   FETCH NEXT SPageSize ROWS ONLY;
```

В пакете примера 8.5 используется база данных `AdventureWorks2012` и ее таблица `Employee` для демонстрации реализации разбиения результатов запроса на страницы на стороне сервера. Количество строк в странице указывается локальной переменной `@Pagesize` в инструкции `FETCH NEXT`. А другая локальная переменная, `@CurrentPage`, указывает, какую конкретную страницу отображать. В данном примере отображается третья страница.

Хранимые процедуры

Хранимая процедура — это специальный тип пакета инструкций Transact-SQL, созданный, используя язык SQL и процедурные расширения. Основное различие между пакетом и хранимой процедурой состоит в том, что последняя сохраняется в виде объекта базы данных. Иными словами, хранимые процедуры сохраняются на стороне сервера, чтобы улучшить производительность и постоянство выполнения повторяемых задач.

Компонент Database Engine поддерживает хранимые процедуры и системные процедуры. Хранимые процедуры создаются таким же образом, как и все другие объекты баз данных, т. е. при помощи языка DDL. Системные процедуры предоставляются компонентом Database Engine и могут применяться для доступа к информации в системном каталоге и ее модификации. В этом разделе рассматриваются хранимые процедуры, которые определяются пользователями, а системные процедуры рассматриваются в следующей главе.

При создании хранимой процедуры можно определить необязательный список параметров. Таким образом, процедура будет принимать соответствующие аргументы при каждом ее вызове. Хранимые процедуры могут возвращать значение, содержащее определенную пользователем информацию или, в случае ошибки, соответствующее сообщение об ошибке.

Хранимая процедура предварительно компилируется перед тем, как она сохраняется в виде объекта в базе данных. Предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Это свойство хранимых процедур предоставляет важную выгоду, заключающуюся в устранении (почти во всех случаях) повторных компиляций процедуры и получении соответствующего улучшения производительности. Это свойство хранимых процедур также оказывает положительный эффект на объем данных, участвующих в обмене между системой баз данных и приложениями. В частности, для вызова хранимой процедуры объемом в несколько тысяч байтов может потребоваться меньше, чем 50 байтов. Когда множественные пользователи выполняют повторяющиеся задачи с применением хранимых процедур, накопительный эффект такой экономии может быть довольно значительным.

Хранимые процедуры можно также использовать для следующих целей:

- управления авторизацией доступа;
- создания аудиторского следа действий с таблицами баз данных.

Использование хранимых процедур предоставляет возможность управления безопасностью на уровне, значительно превышающем уровень безопасности, предоставляемый использованием инструкций GRANT и REVOKE (см. *главу 12*), с помощью которых пользователям предоставляются разные привилегии доступа. Это возможно вследствие того, что авторизация на выполнение хранимой процедуры не зависит от авторизации на модифицирование объектов, содержащихся в данной хранимой процедуре, как это описано в следующем разделе.

Хранимые процедуры, которые выполняют аудит операций записи и/или чтения таблиц, предоставляют дополнительную возможность обеспечения безопасности базы данных. Используя такие процедуры, администратор базы данных может отслеживать модификации, вносимые в базу данных пользователями или прикладными программами.

Создание и исполнение хранимых процедур

Хранимые процедуры создаются посредством инструкции CREATE PROCEDURE, которая имеет следующий синтаксис:

```
CREATE PROC[EDURE] [schema_name.]proc_name
[({@param1} type1 [VARYING] [= default1] [OUTPUT])] {, ...}
[WITH {RECOMPILE | ENCRYPTION | EXECUTE AS 'user_name'}]
[FOR REPLICATION]
AS batch | EXTERNAL NAME method_name
```

Параметр *schema_name* определяет имя схемы, которая назначается владельцем созданной хранимой процедуры. Параметр *proc_name* определяет имя хранимой процедуры. Параметр *@param1* является параметром процедуры (формальным аргументом), чей тип данных определяется параметром *type1*. Параметры процедуры являются локальными в пределах процедуры, подобно тому, как локальные переменные являются локальными в пределах пакета. *Параметры процедуры* — это значения, которые передаются вызывающим объектом процедуре для использования в ней. Параметр *default1* определяет факультативное значение по умолчанию для соответствующего параметра процедуры. (Значением по умолчанию также может быть NULL.)

Параметр OUTPUT указывает, что параметр процедуры является возвращаемым, и с его помощью можно возвратить значение из хранимой процедуры вызывающей процедуре или системе (см. пример 8.9 далее в этом разделе).

Как уже упоминалось ранее, предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Если же по каким-либо причинам хранимую процедуру требуется компилировать при каждом ее вызове, при объявлении процедуры используется опция WITH RECOMPILE.

ПРИМЕЧАНИЕ

Использование опции WITH RECOMPILE сводит на нет одно из наиболее важных преимуществ хранимых процедур: улучшение производительности благодаря одной компиляции. Поэтому опцию WITH RECOMPILE следует использовать только при частых изменениях используемых хранимой процедурой объектов базы данных.

Предложение EXECUTE AS определяет контекст безопасности, в котором должна исполняться хранимая процедура после ее вызова. Задавая этот контекст, с помощью Database Engine можно управлять выбором учетных записей пользователей для проверки полномочий доступа к объектам, на которые ссылается данная хранимая процедура.

По умолчанию использовать инструкцию CREATE PROCEURE могут только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db_owner или db_ddladmin. Но члены этих ролей могут присваивать это право другим пользователям с помощью инструкции GRANT CREATE PROCEDURE. (Предмет пользовательских разрешений, предопределенных ролей сервера и предопределенных ролей баз данных, рассматривается в *главе 12*.)

В примере 8.6 показано создание простой хранимой процедуры для работы с таблицей project.

Пример 8.6. Создание хранимой процедуры для изменения данных таблицы

```
USE sample;
GO
CREATE PROCEDURE increase_budget (@percent INT=5)
   AS UPDATE project
      SET budget = budget + budget*@percent/100;
```

ПРИМЕЧАНИЕ

Для разделения двух пакетов используется инструкция GO. Инструкцию CREATE PROCEDURE нельзя объединять с другими инструкциями Transact-SQL в одном пакете.

Хранимая процедура increase_budget увеличивает бюджеты для всех проектов на определенное число процентов, определяемое посредством параметра @percent. В процедуре также определяется значение числа процентов по умолчанию (5), которое применяется, если во время выполнения процедуры этот аргумент отсутствует.

ПРИМЕЧАНИЕ

Хранимые процедуры могут обращаться к несуществующим таблицам. Это свойство позволяет выполнять отладку кода процедуры, не создавая сначала соответствующие таблицы и даже не подключаясь к конечному серверу.

В отличие от основных хранимых процедур, которые всегда сохраняются в текущей базе данных, возможно создание временных хранимых процедур, которые всегда помещаются во временную системную базу данных tempdb. Одним из поводов для создания временных хранимых процедур может быть желание избежать повторяющегося исполнения определенной группы инструкций при соединении с базой данных. Можно создавать *локальные* или *глобальные* временные процедуры. Для этого имя локальной процедуры задается с одинарным символом # (#proc_name), а имя глобальной процедуры — с двойным (##proc_name). Локальную временную хранимую процедуру может выполнить только создавший ее пользователь и только в течение соединения с базой данных, в которой она была создана. Глобальную временную процедуру могут выполнять все пользователи, но только до тех пор, пока не завершится последнее соединение, в котором она выполняется (обычно это соединение создателя процедуры).

Жизненный цикл хранимой процедуры состоит из двух этапов: ее создания и ее выполнения. Каждая процедура создается один раз, а выполняется многократно. Хранимая процедура выполняется посредством инструкции EXECUTE пользователем, который является владельцем процедуры или обладает правом EXECUTE для доступа к этой процедуре (см. *главу 12*). Инструкция EXECUTE имеет следующий синтаксис:

```
[[EXEC[UTE]] [@return_status =] {proc_name
  |@proc_name_var}
  {[[@parameter1 =] value|[@parameter1=] @variable [OUTPUT]]|DEFAULT}..
  [WITH RECOMPILE]
```

За исключением параметра *return_status*, все параметры инструкции EXECUTE имеют такое же логическое значение, как и одноименные параметры инструкции CREATE PROCEDURE. Параметр *return_status* определяет факультативную целочисленную переменную, в которой сохраняется состояние возврата процедуры. Значение параметру можно присвоить, используя или константу (*value*), или локальную переменную (*@variable*). Порядок значений именованных параметров не важен, но значения неименованных параметров должны предоставляться в том порядке, в каком они определены в инструкции CREATE PROCEDURE.

Предложение DEFAULT предоставляет значения по умолчанию для параметра процедуры, которое было указано в определении процедуры. Когда процедура ожидает значение для параметра, для которого не было определено значение по умолчанию и отсутствует параметр, либо указано ключевое слово DEFAULT, то происходит ошибка.

ПРИМЕЧАНИЕ

Когда инструкция EXECUTE является первой инструкцией пакета, ключевое слово EXECUTE можно опустить. Тем не менее будет надежнее включать это слово в каждый пакет.

Использование инструкции EXECUTE показано в примере 8.7.

Пример 8.7. Применение инструкции EXECUTE

```
USE sample;
EXECUTE increase_budget 10;
```

Инструкция EXECUTE в примере 8.7 выполняет хранимую процедуру increase_budget (см. пример 8.6), которая увеличивает бюджете всех проектов на 10%.

В примере 8.8 показано создание хранимой процедуры для обработки данных в таблицах employee и works_on.

Пример 8.8. Создание хранимой процедуры для обработки таблиц employee и works_on

```
USE sample;
GO
CREATE PROCEDURE modify_empno (@old_no INTEGER, @new_no INTEGER)
   AS UPDATE employee
     SET emp_no = @new_no
     WERE emp_no = @old_no
   UPDATE works_on
     SET emp_no = @new_no
     WHERE emp_no = @old_no
```

Процедура modify_empno в примере 8.8 иллюстрирует использование хранимых процедур, как часть процесса обеспечения ссылочной целостности (в данном случае между таблицами employee и works_on). Подобную хранимую процедуру можно использовать внутри определения триггера, который собственно и обеспечивает ссылочную целостность (см. пример 14.3).

В примере 8.9 показано использование в хранимой процедуре предложения OUTPUT.

Пример 8.9. Использование в хранимой процедуре предложения OUTPUT

```
USE sample;
GO
CREATE PROCEDURE delete_emp @employee_no INT, @counter INT OUTPUT
   AS SELECT @counter = COUNT(*)
      FROM works_on
      WHERE emp_no = @employee_no
   DELETE FROM employee
      WHERE emp_no = @employee_no
      DELETE FROM works_on
   WHERE emp_no = @employee_no
```

Данную хранимую процедуру можно запустить на выполнение посредством следующих инструкций:

```
DECLARE @quantity INT
EXECUTE delete_emp @employee_no=28559, @counter=@quantity OUTPUT
```

Эта процедура подсчитывает количество проектов, над которыми занят сотрудник с табельным номером @employee_no, и присваивает полученное значение параметру @counter. После удаления всех строк для данного табельного номера из таблиц employee и works_on вычисленное значение присваивается переменной @quantity.

ПРИМЕЧАНИЕ

Значение параметра возвращается вызывающей процедуре только в том случае, если указана опция OUTPUT. В примере 8.9 процедура delete_emp передает вызывающей процедуре параметр @counter, следовательно, хранимая процедура возвращает значение системе. Поэтому параметр @counter необходимо указывать как в опции OUTPUT при объявлении процедуры, так и в инструкции EXECUTE при ее вызове.

Предложение *WITH RESULTS SETS* инструкции *EXECUTE*

В SQL Server 2012 для инструкции EXECUTE вводится предложение WITH RESULT SETS, посредством которого при выполнении определенных условий можно изменять форму результирующего набора хранимой процедуры.

Следующие два примера помогут объяснить это предложение. Пример 8.10 является вводным примером, который показывает, как может выглядеть результат, когда опущено предложение WITH RESULT SETS.

Пример 8.10. Определение хранимой процедуры

```
USE sample;
GO
CREATE PROCEDURE employees_in_dept (@dept CHAR(4))
   AS SELECT emp_no, emp_lname
      FROM employee
      WHERE dept_no IN (SELECT @dept FROM department
                        GROUP BY dept_no)
```

Процедура employees_in_dept — это простая процедура, которая отображает табельные номера и фамилии всех сотрудников, работающих в определенном отделе. Номер отдела является параметром процедуры, и его нужно указать при ее вызове. Выполнение этой процедуры выводит таблицу с двумя столбцами, заглавия которых совпадают с наименованиями соответствующих столбцов таблицы базы данных, т. е. emp_no и emp_lname. Чтобы изменить заглавия столбцов результата (а также их тип данных), в SQL Server 2012 применяется новое предложение WITH RESULT SETS. Применение этого предложения показано в примере 8.11.

Пример 8.11. Применение инструкции EXECUTE с предложением WITH RESULT SETS

```
USE sample;
EXEC employees_in_dept 'd1'
   WITH RESULT SETS
   (([EMPLOYEE NUMBER] INT NOT NULL,
      [NAME OF EMPLOYEE] CHAR(20) NOT NULL));
```

Результат выполнения хранимой процедуры, вызванной таким способом, будет следующим:

EMPLOYEE NUMBER	NAME OF EMPLOYEE
18316	Barrimore
28559	Moser

Как можно видеть, запуск хранимой процедуры с использованием предложения WITH RESULT SETS в инструкции EXECUTE позволяет изменить наименования и тип данных столбцов результирующего набора, выдаваемого данной процедурой. Таким образом, эта новая функциональность предоставляет большую гибкость в исполнении хранимых процедур и помещении их результатов в новую таблицу.

Изменение структуры хранимых процедур

Компонент Database Engine также поддерживает инструкцию ALTER PROCEDURE для модификации структуры хранимых процедур. Инструкция ALTER PROCEDURE обычно применяется для изменения инструкций Transact-SQL внутри процедуры. Все параметры инструкции ALTER PROCEDURE имеют такое же значение, как и одноименные параметры инструкции CREATE PROCEDURE. Основной целью использования этой инструкции является избежание переопределения существующих прав хранимой процедуры.

ПРИМЕЧАНИЕ

Компонент Database Engine поддерживает тип данных CURSOR. Этот тип данных используется для объявления курсоров в хранимых процедурах. *Курсор* — это конструкция программирования, применяемая для хранения результатов запроса (обычно набора строк) и для предоставления пользователям возможности отображать этот результат построчно. Подробное рассмотрение курсоров выходит за рамки тематики этой книги.

Для удаления одной или группы хранимых процедур используется инструкция DROP PROCEDURE. Удалить хранимую процедуру может только ее владелец или члены предопределенных ролей db_owner и sysadmin.

Хранимые процедуры и среда CLR

SQL Server поддерживает общеязыковую среду выполнения *CLR* (Common Language Runtime), которая позволяет разрабатывать различные объекты баз дан-

ных (хранимые процедуры, определяемые пользователем функции, триггеры, определяемые пользователем статистические функции и пользовательские типы данных), применяя языки C# и Visual Basic. Среда CLR также позволяет выполнять эти объекты, используя систему общей среды выполнения.

ПРИМЕЧАНИЕ

Среда CLR разрешается и запрещается посредством опции `clr_enabled` системной процедуры `sp_configure`, которая запускается на выполнение инструкцией `RECONFIGURE`.

В примере 8.12 показано, как можно с помощью системной процедуры `sp_configure` разрешить использование среды CLR.

Пример 8.12. Использование среды CLR посредством системной процедуры `sp_configure`

```
USE sample;
EXEC sp_configure 'clr_enabled', 1
RECONFIGURE
```

Для создания, компилирования и сохранения процедуры с помощью среды CLR требуется выполнить следующую последовательность шагов в указанном порядке:

1. Создать хранимую процедуру на языке C# или Visual Basic, а затем скомпилировать ее, используя соответствующий компилятор.
2. Используя инструкцию `CREATE ASSEMBLY`, создать соответствующий выполняемый файл.
3. Сохранить процедуру в виде объекта сервера, используя инструкцию `CREATE PROCEDURE`.
4. Выполнить процедуру, используя инструкцию `EXECUTE`.

На рис. 8.1 показана графическая схема ранее изложенных шагов.

Далее приводится более подробное описание этого процесса.

Сначала создайте требуемую программу в какой-либо среде разработки, например Visual Studio. Скомпилируйте готовую программу в объектный код, используя компилятор C# или Visual Basic. Этот код сохраняется в файле динамической библиотеки (.dll), который служит источником для инструкции `CREATE ASSEMBLY`, создающей промежуточный выполняемый код. Далее выполните инструкцию `CREATE PROCEDURE`, чтобы сохранить выполняемый код в виде объекта базы данных. Наконец, запустите процедуру на выполнение, используя уже знакомую нам инструкцию `EXECUTE`.

Практическая демонстрация всего процесса приведена в примерах 8.13 по 8.17. В примере 8.13 показан исходный код хранимой процедуры на языке C#.

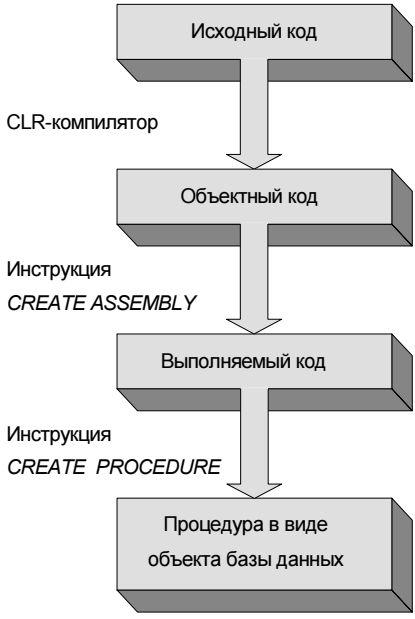


Рис. 8.1. Блок-схема процесса создания хранимой процедуры в среде CLR

Пример 8.13. Исходный код хранимой процедуры на языке C#

```
using System;
using System.Data;
using System.Data.Sql;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;

public partial class StoredProcedures
{[SqlProcedure]
   public static int GetEmployeeCount()
   {
      int iRows;
      SqlConnection conn = new SqlConnection
                              ("Context Connection=true");
      conn.Open();
      SqlCommand sqlCmd = conn.CreateCommand();
      sqlCmd.CommandText = "select count(*) as
                        'Employee Count'" + "from employee";
      iRows = (int)sqlCmd.ExecuteScalar();
      conn.Close();
      return iRows;
   }
};
```

В этой процедуре реализуется запрос для подсчета числа строк в таблице employee. В директивах using в начале программы указываются пространства имен, требуемые для ее выполнения. Применение этих директив позволяет указывать в исходном коде имена классов без явного указания соответствующих пространств имен. Далее определяется класс StoredProcedure, для которого применяется атрибут [SqlProcedure], который информирует компилятор о том, что этот класс является хранимой процедурой. Внутри кода класса определяется метод GetEmployeeCount(). Соединение с системой баз данных устанавливается посредством экземпляра conn класса SQLConnection. Чтобы открыть соединение, применяется метод Open() этого экземпляра. А метод CreateCommand() экземпляра conn позволяет обращаться к экземпляру sqlCmd класса SqlCommnd.

В следующем фрагменте кода:

```
sqlCmd.CommandText =
"select count(*) as 'Employee Count' " + "from employee";
iRows = (int)sqlCmd.ExecuteScalar();
```

используется инструкция SELECT для подсчета количества строк в таблице employee и отображения результата. Текст команды указывается, присваивая свойству CommandText экземпляра SqlCmd экземпляр, возвращаемый методом CreateCommand(). Далее вызывается метод ExecuteScalar() экземпляра SqlCmd. Этот метод возвращает скалярное значение, которое преобразовывается в целочисленный тип данных int и присваивается переменной iRows.

В примере 8.14 показан следующий шаг по реализации хранимой процедуры в среде CLR, а именно ее компиляция.

Пример 8.14. Компиляция исходного кода хранимой процедуры

```
csc /target:library GetEmployeeCount.cs
/reference:"C:\Program Files\Microsoft SQL Server\
                    MSSQL11.MSSQLSERVER\MSSQL\Binn\sqlaccess.dll"
```

В примере 8.14 выполняется компиляция метода GetEmployeeCount() из примера 8.13. (Вообще-то этот код можно использовать для компиляции любой программы на языке C#, указав в нем соответствующее имя файла с ее исходным кодом.) Компилятор C# запускается посредством команды csc, которая вводится в командной строке Windows. Для запуска компилятора этой командой необходимо заблаговременно указать его расположение, используя переменную среды PATH. На момент написания этой книги компилятор C# (файл csc.exe) находится в папке *C:\WINDOWS\Microsoft.NET\Framework*. (Следует выбрать соответствующую версию компилятора.)

Опция /target указывает имя исходной программы на языке C#, а опция /reference определяет конечный файл как .dll, что необходимо для целей компиляции.

В примере 8.15 показан следующий шаг в создании хранимой процедуры: создание выполняемого кода. Прежде чем выполнять код в этом примере, необходимо скопировать полученный в предыдущем примере dll-файл в корневую папку диска C:.

Пример 8.15. Создание выполняемого кода хранимой процедуры

```
USE sample;
GO
CREATE ASSEMBLY GetEmployeeCount
   FROM 'C:\GetEmployeeCount.dll' WITH PERMISSION_SET = SAFE
```

Инструкция `CREATE ASSEMBLY` принимает в качестве ввода управляемый код и создает соответствующий объект, для которого можно создавать хранимые процедуры среды CLR, определяемые пользователем функции и триггеры. Эта инструкция имеет следующий синтаксис:

```
CREATE ASSEMBLY assembly_name [AUTHORIZATION owner_name]
   FROM {dll_file}
   [WITH PERMISSION_SET = {SAFE | EXTERNAL_ACCESS | UNSAFE }]
```

В параметре *assembly_name* указывается имя сборки. В необязательном предложении `AUTHORIZATION` указывается имя роли в качестве владельца этой сборки. В предложении `FROM` указывается путь, где находится загружаемая сборка. (В примере 8.15 dll-файл, созданный из исходной программы, был скопирован из папки *Framework* в корневую папку диска C:.)

Предложение `WITH PERMISSTION_SET` является очень важным предложением инструкции `CREATE AESSMBLY` и всегда должно указываться. В нем определяется набор прав доступа, предоставляемых коду сборки. Набор прав `SAFE` является наиболее ограничивающим. Код сборки, имеющий эти права, не может обращаться к внешним системным ресурсам, таким как файлы. Набор прав `EXTERNAL_ACCESS` позволяет коду сборки обращаться к определенным внешним системным ресурсам, а набор прав `UNSAFE` предоставляет неограниченный доступ к ресурсам, как внутри, так и вне системы базы данных.

ПРИМЕЧАНИЕ

Чтобы сохранить информацию о коде сборке, пользователь должен имеет возможность выполнить инструкцию `CREATE ASSEMBLY`. Владельцем сборки является пользователь (или роль), исполняющий эту инструкцию. Владельцем сборки можно сделать другого пользователя, используя предложение `AUTHORIZAION` инструкции `CREATE SCHEMA`.

Компонент Database Engine также поддерживает инструкции `ALTER ASSEMBLY` и `DROP ASSEMBLY`. Инструкция `ALTER ASSEMBLY` используется для обновления системной папки до последней копии модулей Microsoft .NET Framework, содержащих реализацию сборки. Эта инструкция также добавляет или удаляет файлы, связанные с соответствующей сборкой. Инструкция `DROP ASSEMBLY` удаляет указанную сборку и все связанные с ней файлы из текущей базы данных.

В примере 8.16 показано создание хранимой процедуры на основе управляемого кода, реализованного в примере 8.13.

Пример 8.16. Создание хранимой процедуры с использованием параметра EXTERNAL NAME

```
USE sample;
GO
CREATE PROCEDURE GetEmployeeCount
AS EXTERNAL NAME GetEmployeeCount.StoredProcedures.GetEmployeeCount
```

Инструкция CREATE PROCEDURE в примере 8.16 отличается от такой же инструкции в примерах 8.6 и 8.8 тем, что она содержит параметр EXTERNAL NAME. Этот параметр указывает, что код создается средой CLR. Имя в этом предложении состоит из трех частей:

assembly_name.class_name.method_name

где:

- assembly_name — указывает имя сборки (см. пример 8.15);
- class_name — указывает имя общего класса (см. пример 8.13);
- method_name — необязательная часть, указывает имя метода, который задается внутри класса.

Выполнение процедуры GetEmployeeCount показано в примере 8.17.

Пример 8.17. Выполнение хранимой процедуры GetEmployeeCount

```
USE sample;
DECLARE @ret INT
EXECUTE @ret=GetEmployeeCount
PRINT @ret
```

Инструкция PRINT возвращает текущее количество строк в таблице employee.

Определяемые пользователем функции

В языках программирования обычно имеется два типа подпрограмм:

- хранимые процедуры;
- определяемые пользователем функции (ОПФ).

Как уже было рассмотрено ранее в этой главе, хранимые процедуры состоят из нескольких инструкций и имеют от нуля до нескольких входных параметров, но обычно не возвращают никаких параметров. В отличие от хранимых процедур, функции всегда возвращают одно значение. В этом разделе мы рассмотрим создание и использование определяемых пользователем функций (ОПФ) (*User Defined Functions* — UDF).

Создание и выполнение определяемых пользователем функций

Определяемые пользователем функции создаются посредством инструкции CREATE FUNCTION, которая имеет следующий синтаксис:

```
CREATE FUNCTION [schema_name.]function_name
   [({@param} type [= default]) {,...}
   RETURNS {scalar_type | [@variable] TABLE}
   [WITH {ENCRYPTION | SCHEMABINDING}
   [AS] {block | RETURN (select_statement)}
```

Параметр *schema_name* определяет имя схемы, которая назначается владельцем создаваемой ОПФ, а параметр *function_name* определяет имя этой функции. Параметр *@param* является входным параметром функции (формальным аргументом), чей тип данных определяется параметром *type*. Параметры процедуры — это значения, которые передаются вызывающим объектом определяемой пользователем функции для использования в ней. Параметр *default* определяет факультативное значение по умолчанию для соответствующего параметра функции. (Значением по умолчанию также может быть NULL.)

Предложение RETURNS определяет тип данных значения, возвращаемого ОПФ. Это может быть почти любой стандартный тип данных, поддерживаемый системой баз данных, включая тип данных TABLE. Единственным типом данных, который нельзя указывать, является тип данных TIMESTAMP.

Определяемые пользователем функции могут быть либо *скалярными*, либо *табличными*. Скалярные функции возвращают атомарное (скалярное) значение. Это означает, что в предложении RETURNS скалярной функции указывается один из стандартных типов данных. Функция является табличной, если предложение RETURNS возвращает набор строк (см. *следующий подраздел* для дополнительной информации).

Параметр WITH ENCRYPTION в системном каталоге кодирует информацию, содержащую текст инструкции CREATE FUNCTION. Таким образом, предотвращается несанкционированный просмотр текста, который был использован для создания функции. Данная опция позволяет повысить безопасность системы баз данных.

Альтернативное предложение WITH SCHEMABINDING привязывает ОПФ к объектам базы данных, к которым эта функция обращается. После этого любая попытка модифицировать объект базы данных, к которому обращается функция, претерпевает неудачу. (Привязка функции к объектам базы данных, к которым она обращается, удаляется только при изменении функции, после чего параметр CHEMABINDING больше не задан.)

Для того чтобы во время создания функции использовать предложение SCHEMAMINDING, объекты базы данных, к которым обращается функция, должны удовлетворять следующим условиям,

- все представления и другие ОПФ, к которым обращается определяемая функция, должны быть привязаны к схеме;

- все объекты базы данных (таблицы, представления и ОПФ) должны быть в той же самой базе данных, что и определяемая функция.

Параметр block определяет блок BEGIN/END, содержащий реализацию функции. Последней инструкцией блока должна быть инструкция RETURN с аргументом. (Значением аргумента является возвращаемое функцией значение.) Внутри блока BEGIN/END разрешаются только следующие инструкции:

- инструкции присвоения, такие как SET;
- инструкции для управления ходом выполнения, такие как WHILE и IF;
- инструкции DECLARE, объявляющие локальные переменные;
- инструкции SELECT, содержащие списки столбцов выборки с выражениями, значения которых присваиваются переменным, являющимися локальными для данной функции;
- инструкции INSERT, UPDATE и DELETE, которые изменяют переменные с типом данных TABLE, являющиеся локальными для данной функции.

По умолчанию инструкцию CREATE FUNCTION могут использовать только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db_owner или db_ddladmin. Но члены этих ролей могут присвоить это право другим пользователям с помощью инструкции GRANT CREATE FUNCTION (см. *главу 12*).

В примере 8.18 показано создание функции compute_costs.

Пример 8.18. Создание определяемой пользователем функции

```
-- Эта функция вычисляет возникающие дополнительные общие затраты,
-- при увеличении бюджетов проектов
USE sample;
GO
CREATE FUNCTION compute_costs (@percent INT =10)
RETURNS DECIMAL(16, 2)
   BEGIN
      DECLARE @additional_costs DEC (14,2), @sum_budget dec(16,2)
      SELECT @sum_budget = SUM (budget) FROM project
      SET @additional_costs = @sum_budget * @percent/100
      RETURN @additional_costs
   END
```

Функция compute_costs вычисляет дополнительные расходы, возникающие при увеличении бюджетов проектов. Единственный входной параметр, @percent, определяет процентное значение увеличения бюджетов. В блоке BEGIN/END сначала объявляются две локальные переменные: @additional_costs и @sum_budget, а затем с помощью инструкции SELECT переменной @sum_budget присваивается общая сумма всех бюджетов. После этого функция вычисляет общие дополнительные расходы и посредством инструкции RETURN возвращает это значение.

Вызов определяемой пользователем функции

Определенную пользователем функцию можно вызывать с помощью инструкций Transact-SQL, таких как SELECT, INSERT, UPDATE или DELETE. Вызов функции осуществляется, указывая ее имя с парой круглых скобок в конце, в которых можно задать один или несколько аргументов. *Аргументы* — это значения или выражения, которые передаются входным параметрам, определяемым сразу же после имени функции. При вызове функции, когда для ее параметров не определены значения по умолчанию, для всех этих параметров необходимо предоставить аргументы в том же самом порядке, в каком эти параметры определены в инструкции CREATE FUNCTION.

В примере 8.19 показан вызов функции compute_costs (см. пример 8.18) в инструкции SELECT.

Пример 8.19. Вызов ОПФ в инструкции SELECT

```
USE sample;
SELECT project_no, project_name
   FROM project
   WHERE budget < dbo.compute_costs(25)
```

Результат выполнения:

project_no	project_name
p2	Gemini

Инструкция SELECT в примере 8.19 отображает названия и номера всех проектов, бюджеты которых меньше, чем общие дополнительные расходы по всем проектам при заданном значении процентного увеличения.

ПРИМЕЧАНИЕ

В инструкциях Transact-SQL имена функций необходимо задавать, используя имена, состоящие из двух частей: *schema_name*.*function_name*.

Возвращающие табличное значение функции

Как уже упоминалось ранее, функция является возвращающей табличное значение, если ее предложение RETURNS возвращает набор строк. В зависимости от того, каким образом определено тело функции, возвращающие табличное значение функции классифицируются как *встраиваемые* (inline) и *многоинструкционные* (multistatement). Если в предложении RETURNS ключевое слово TABLE указывается без сопровождающего списка столбцов, такая функция является встроенной. Инструкция SELECT встраиваемой функции возвращает результирующий набор в виде переменной с типом данных TABLE (см. пример 8.20). Многоинструкционная возвращающая табличное значение функция содержит имя, определяющее внутреннюю

переменную с типом данных TABLE. Этот тип данных указывается ключевым словом TABLE, которое следует за именем переменной. В эту переменную вставляются выбранные строки, и она служит возвращаемым значением функции.

Создание возвращающей табличное значение функции показано в примере 8.20.

Пример 8.20. Создание возвращающей табличное значение функции

```
USE sample;
GO
CREATE FUNCTION employees_in_project (@pr_number CHAR(4))
   RETURNS TABLE
   AS RETURN (SELECT emp_fname, emp_lname
      FROM works_on, employee
      WHERE employee.emp_no = works_on.emp_no
      AND project_no = @pr_number)
```

Функция employees_in_project отображает имена всех сотрудников, работающих над определенным проектом, номер которого задается входным параметром @pr_number. Тогда как функция в общем случае возвращает набор строк, предложение RETURNS в определение данной функции содержит ключевое слово TABLE, указывающее, что функция возвращает табличное значение. (Обратите внимание на то, что в примере 8.20 блок BEGIN/END необходимо опустить, а предложение RETURN содержит инструкцию SELECT.)

Использование функции employees_in_project приведено в примере 8.21.

Пример 8.21. Использование возвращающей табличное значение функции

```
USE sample;
SELECT *
   FROM employees_in_project('p3')
```

Результат выполнения:

emp_fname	emp_lname
Ann	Jones
Elsa	Bertoni
Jlke	Hansel

Возвращающие табличное значение функции и инструкция *APPLY*

Реляционная инструкция APPLY позволяет вызывать возвращающую табличное значение функцию для каждой строки табличного выражения. Эта инструкция задается в предложении FROM соответствующей инструкции SELECT таким же образом, как и инструкция JOIN. Инструкция APPLY может быть объединена с табличной функци-

ей для получения результата, похожего на результирующий набор операции соединения двух таблиц. Существует две формы инструкции APPLY:

- CROSS APPLY
- OUTER APPLY

Инструкция CROSS APPLY возвращает те строки из внутреннего (левого) табличного выражения, которые совпадают с внешним (правым) табличным выражением. Таким образом, логически, инструкция CROSS APPLY функционирует так же, как и инструкция INNER JOIN.

Инструкция OUTER APPLY возвращает все строки из внутреннего (левого) табличного выражения. (Для тех строк, для которых нет совпадений во внешнем табличном выражении, он содержит значения NULL в столбцах внешнего табличного выражения.) Логически, инструкция OUTER APPLY эквивалентна инструкции LEFT OUTER JOIN.

Применение инструкции APPLY показано в примерах 8.22—8.23.

Пример 8.22. Создание возвращающей табличное значение функции

```
-- создать функцию
CREATE FUNCTION dbo.fn_getjob(@empid AS INT)
   RETURNS TABLE AS
RETURN
   SELECT job
      FROM works_on
      WHERE emp_no = @empid
         AND job IS NOT NULL AND project_no = 'p1';
```

Функция fn_getjob() в примере 8.22 возвращает набор строк с таблицы works_on. В примере 8.23 этот результирующий набор "соединяется" предложением APPLY с содержимым таблицы employee.

Пример 8.23. "Соединение" двух таблиц посредством предложения APPLY

```
-- используется CROSS APPLY
SELECT E.emp_no, emp_fname, emp_lname, job
   FROM employee as E
   CROSS APPLY dbo.fn_getjob(E.emp_no) AS A
   -- используется OUTER APPLY
   SELECT E.emp_no, emp_fname, emp_lname, job
   FROM employee as E
   OUTER APPLY dbo.fn_getjob(E.emp_no) AS A
```

Результатом выполнения этих двух функций будут следующие две таблицы (отображаются после выполнения второй функции):

emp_no	emp_fname	emp_lname	job
10102	Ann	Jones	Analyst
29346	James	James	Clerk
9031	Elsa	Bertoni	Manager
28559	Sybill	Moser	NULL

emp_no	emp_fname	emp_lname	job
25348	Matthew	Smith	NULL
10102	Ann	Jones	Analyst
18316	John	Barrimore	NULL
29346	James	James	Clerk
9031	Elsa	Bertoni	Manager
2581	Elke	Hansel	NULL
28559	Sybill	Moser	NULL

В первом запросе примера 8.23 результирующий набор возвращающей табличное значение функции `fn_getjob()` из примера 8.22 "соединяется" с содержимым таблицы `employee` посредством инструкции `CROSS APPLY`. Функция `fn_getjob()` играет роль правого ввода, а таблица `employee` — левого. Выражение правого ввода вычисляется для каждой строки левого ввода, а полученные строки комбинируются, создавая конечный результат.

Второй запрос похожий на первый (но в нем используется инструкция `OUTER APPLY`), который логически соответствует операции внешнего соединения двух таблиц.

Возвращающие табличное значение параметры

Во всех версиях сервера, предшествующих SQL Server 2008, задача передачи подпрограмме множественных параметров была сопряжена со значительными сложностями. Для этого сначала нужно было создать временную таблицу, вставить в нее передаваемые значения, и только затем можно было вызывать подпрограмму. Начиная с версии SQL Server 2008, эта задача упрощена, благодаря возможности использования возвращающих табличное значение параметров, посредством которых результирующий набор может быть передан соответствующей подпрограмме.

Использование возвращающего табличное значение параметра показано в примере 8.24.

Пример 8.24. Применение возвращающего табличное значение параметра

```
USE sample;
GO
CREATE TYPE departmentType AS TABLE
   (dept_no CHAR(4), dept_name CHAR(25),location CHAR(30));
GO
```

```
CREATE TABLE #dallasTable
   (dept_no CHAR(4), dept_name CHAR(25),location CHAR(30));
GO
CREATE PROCEDURE insertProc
   @Dallas departmentType READONLY
   AS SET NOCOUNT ON
   INSERT INTO #dallasTable (dept_no, dept_name, location)
   SELECT * FROM @Dallas
GO
DECLARE @Dallas AS departmentType;
INSERT INTO @Dallas( dept_no, dept_name, location)
SELECT * FROM department
WHERE location = 'Dallas'
EXEC insertProc @Dallas;
```

В примере 8.24 сначала определяется табличный тип `departmentType`. Это означает, что данный тип является типом данных `TABLE`, вследствие чего он разрешает вставку строк. В процедуре `insertProc` объявляется переменная `@Dallas` с типом данных `departmentType`. (Предложение `READONLY` указывает, что содержимое этой таблицы нельзя изменять.) В последующем пакете в эту табличную переменную вставляются данные, после чего процедура запускается на выполнение. В процессе исполнения процедура вставляет строки из табличной переменной во временную таблицу `#dallasTable`. Вставленное содержимое временной таблицы выглядит следующим образом:

dept_no	dept_name	location
d1	Research	Dallas
d3	Marketing	Dallas

Использование возвращающих табличное значение параметров предоставляет следующие преимущества:

- упрощается модель программирования подпрограмм;
- уменьшается количество обращений к серверу и получений соответствующих ответов;
- таблица результата может иметь произвольное количество строк.

Изменение структуры определяемых пользователями инструкций

Язык Transact-SQL также поддерживает инструкцию `ALTER FUNCTION`, которая модифицирует структуру определяемых пользователями инструкций (ОПФ). Эта инструкция обычно используется для удаления привязки функции к схеме. Все параметры инструкции `ALTER FUNCTION` имеют такое же значение, как и одноименные параметры инструкции `CREATE FUNCTION`.

Для удаления ОПФ применяется инструкция DROP FUNCTION. Удалить функцию может только ее владелец или член предопределенной роли db_owner или sysadmin.

Определяемые пользователем функции и среда CLR

Материал, рассмотренный ранее в *разд. "Хранимые процедуры и среда CLR" этой главы*, также верен и для определяемых пользователем функций (ОПФ), с одним только различием, что для сохранения ОПФ в виде объекта базы данных используется инструкция CREATE FUNCTION, а не CREATE PROCEDURE. Кроме этого, определяемые пользователем функции также применяются в другом контексте, чем хранимые процедуры, поскольку ОПФ всегда возвращают значение.

В примере 8.25 показан исходный код определяемых пользователем функций (ОПФ), реализованный на языке C#.

Пример 8.25. Исходный код ОПФ на языке C#

```
using System;
using System.Data.Sql;
using System.Data.SqlTypes;
public class budgetPercent
{private const float percent = 10;
   public static SqlDouble computeBudget(float budget)
   {float budgetNew;
      budgetNew = budget * percent;
      return budgetNew;
   }
};
```

В исходном коде определяемых пользователем функций в примере 8.25 вычисляется новый бюджет проекта, увеличивая старый бюджет на определенное количество процентов. (Описание значений элементов программы для данного примера не приводится, поскольку эта программа аналогична программе, приведенной в примере 8.13.) В примере 8.26 приводится инструкция CREATE ASSEMBLY для создания объекта базы данных для ОПФ.

Пример 8.26. Создание объекта базы данных для ОПФ

```
USE sample;
GO
CREATE ASSEMBLY computeBudget
FROM 'C:\computeBudget.dll'
WITH PERMISSION_SET = SAFE
```

Инструкция CREATE FUNCTION в примере 8.27 сохраняет сборку computeBudget в виде объекта базы данных, который в дальнейшем можно использовать в инструкциях

для манипулирования данными. Использование одной из таких инструкций, инструкции SELECT, показано в примере 8.28.

Пример 8.27. Создание объекта базы данных для ОПФ

```
USE sample;
GO
CREATE FUNCTION ReturncomputeBudget (@budget Real)
RETURNS FLOAT
AS EXTERNAL NAME computeBudget.budgetPercent.computeBudget
```

Пример 8.28. Использование объекта базы данных ОПФ с инструкцией SELECT

```
USE sample;
SELECT dbo.ReturncomputeBudget (321.50)
```

Этот запрос возвращает результат 3215.

ПРИМЕЧАНИЕ

Определяемую пользователем функцию можно поместить в разных местах инструкции SELECT. В примере 8.19 она вызывается в предложении WHERE, в примере 8.21 — в предложении FROM, а в примере 8.28 — в списке выбора оператора SELECT.

Резюме

Хранимая процедура представляет собой специальный пакет, созданный либо на языке Transact-SQL, либо используя общеязыковую среду выполнения CLR. Хранимые процедуры используются для следующих целей:

- управления авторизацией доступа;
- создания контрольной записи действий с таблицами баз данных;
- принудительного обеспечения целостности данных и бизнес-правил применительно к модификации данных;
- улучшения производительности повторяемых задач.

Определяемые пользователем функции во многом схожи с хранимыми процедурами. Основное различие между ними заключается в том, что ОПФ не поддерживают параметров, но возвращают одно значение данных, которое может быть таблицей.

Для создания серверных объектов разработчики корпорации Microsoft рекомендуют использовать по умолчанию язык Transact-SQL. Использование среды CLR рекомендуется в качестве альтернативы только в тех случаях, когда программа должна выполнять большой объем вычислений.

В следующей главе рассматривается системный каталог компонента Database Engine.

Упражнения

Упражнение 8.1

Создайте пакет для вставки 3000 строк в таблицу employee. Значения столбца emp_no должны быть однозначными в диапазоне от 1 до 3000. Всем ячейкам столбцов emp_lname, emp_fname и dept_no присваиваются значения "Jane", "Smith" и "d1" соответственно.

Упражнение 8.2

Измените пакет из упражнения 8.1 таким образом, чтобы генерировать случайные значения для столбца emp_no, используя функцию RAND. (Подсказка: для получения случайных значений используйте системные функции DATEPART и GETDATE.)

Глава 9

Системный каталог

- **Введение в системный каталог**
- **Общие интерфейсы**
- **Специализированные интерфейсы**

В этой главе рассматривается системный каталог компонента Database Engine. После вводного раздела следует описание структуры нескольких представлений каталога, каждое из которых позволяет извлекать метаданные. Также в первой части этой главы рассматривается использование представлений и функций динамического управления. Во второй части главы рассматривается четыре альтернативных способа для извлечения метаданных: системные хранимые процедуры, системные функции, функции свойств и информационная схема.

Введение в системный каталог

Системный каталог состоит из таблиц, описывающих структуру объектов базы данных, таких как базовые таблицы, представления, индексы и собственно базы данных. Эти таблицы называются *системными базовыми таблицами*. Компонент Database Engine часто обращается к системному каталогу за информацией, необходимой для правильного функционирования системы.

Компонент Database Engine отличает системные базовые таблицы базы данных `master` от базовых таблиц пользовательских баз данных. Системные таблицы базы данных `master` принадлежат к системному каталогу, а системные таблицы определенной базы данных составляют каталог этой базы данных. Поэтому системные базовые таблицы присутствуют только в одном экземпляре для всей системы (если они принадлежат исключительно к базе данных `master`), тогда как другие таблицы присутствуют в одном экземпляре в каждой базе данных, включая базу данных `master`.

Во всех реляционных системах баз данных системные базовые таблицы имеют такую же логическую структуру, как и базовые таблицы. Поэтому информацию из системных базовых таблиц можно извлекать посредством таких же инструкций Transact-SQL, какие применяются для извлечения информации из базовых таблиц.

ПРИМЕЧАНИЕ

К системным базовым таблицам нельзя обращаться напрямую. Для этого необходимо выполнять запрос на информацию из системного каталога посредством существующих интерфейсов.

Для обращения за информацией к системным базовым таблицам можно использовать несколько разных интерфейсов.

- *Представления каталогов*. Является основным интерфейсом для метаданных, хранящихся в системных базовых таблицах. Метаданные описывают атрибуты объектов в системе баз данных.
- *Динамические административные представления (ДАП) (Dynamic Management Views, DMV) и динамические административные функции (ДАФ) (Dynamic Management Functions, DMF)*. Эти интерфейсы обычно применяются для просмотра активных процессов и содержимого памяти.
- *Информационная схема*. Стандартное решение для доступа к метаданным, которое предоставляет общий интерфейс не только для компонента Database Engine, но и для всех существующих реляционных систем баз данных (при условии, что конкретная система поддерживает информационную схему).
- *Системные функции и функции свойств*. Позволяют извлекать системную информацию. Основная разница между этими двумя типами функций заключается в их структуре. Кроме этого, функции свойств возвращают больше информации, чем системные функции.
- *Системные хранимые процедуры*. Некоторые системные хранимые процедуры можно использовать для получения доступа к содержимому системных баз данных и модификации этого содержимого.

Упрощенная форма системной информации компонента Database Engine и различные интерфейсы для доступа к ней показаны на рис. 9.1.

ПРИМЕЧАНИЕ

В этой главе представляется только обзор системного каталога и способов доступа к метаданным.

Отдельные представления каталога, а также все другие интерфейсы, специфичные для разных тематик (например, индексы, безопасность, и т. п.), рассматриваются в соответствующих главах.

Все эти интерфейсы можно подразделить на две группы: *общие* интерфейсы (представления каталога, динамические административные представления и функции,

информационная схема) и *специализированные* интерфейсы компонента Database Engine (системные хранимые процедуры, системные функции и функции свойств).

ПРИМЕЧАНИЕ

"Общие" означает, что все реляционные системы баз данных поддерживают такие интерфейсы, но используют другую терминологию. Например, в терминологии Oracle, представления каталога и динамические административные представления называются "представления словаря данных" и "представления V$" соответственно.

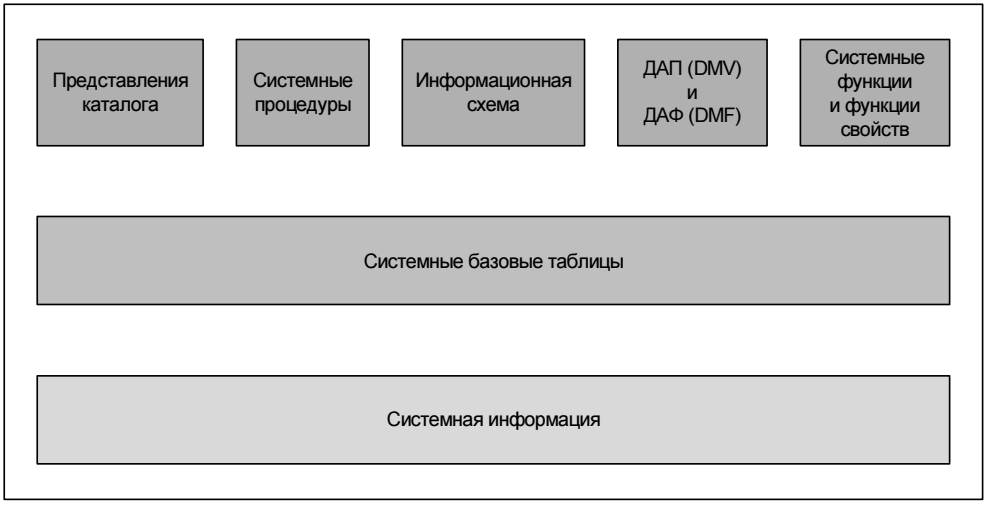


Рис. 9.1. Графическое представление разных интерфейсов для доступа к системному каталогу

Общие интерфейсы рассматриваются в следующем разделе, а специализированные интерфейсы — далее в этой главе.

Общие интерфейсы

Как уже упоминалось, к общим относятся следующие интерфейсы:

- представления каталога;
- динамические административные представления и динамические административные функции;
- информационная схема.

Представления каталога

Представления каталога являются наиболее общим интерфейсом для метаданных и предоставляют самый эффективный способ получения специализированных форм этой информации (см. примеры 9.1—9.3).

Представления каталога принадлежат к схеме sys, поэтому при обращении к этим объектам следует использовать имя этой схемы. В этом разделе рассматриваются три наиболее важных представления каталога:

- sys.objects;
- sys.columns;
- sys.database_principals.

ПРИМЕЧАНИЕ

Описание других представлений можно найти в других главах этой книги или в электронной документации.

Представление каталога sys.objects возвращает строку для каждого объекта, определенного пользователем в схеме пользователя. Существуют два других представления каталога, которые предоставляют доступ к подобной информации: sys.system_objects и sys.all_objects. Представление sys.system_objects возвращает строку для каждого системного объекта, а представление sys.all_objects отображает объединение всех определенных пользователем и системных объектов в области видимости схемы. (Все три представления каталога имеют одинаковую структуру.) Наиболее важные столбцы представления каталога sys.objects перечислены в табл. 9.1.

Таблица 9.1. *Наиболее важные столбцы представления каталога sys.objects*

Имя столбца	Описание
name	Имя объекта
object_id	Идентификационный номер объекта, однозначный в пределах базы данных
schema_id	Идентификатор схемы, содержащей объект
type	Тип объекта

Представление каталога sys.columns возвращает строку для каждого столбца объектов со столбцами, таких как таблицы и представления. Наиболее важные столбцы представления каталога sys.columns перечислены в табл. 9.2.

Таблица 9.2. *Наиболее важные столбцы представления каталога sys.columns*

Имя столбца	Описание
object_id	Идентификатор объекта, которому принадлежит данный столбец
name	Имя столбца
column_id	Идентификатор столбца (однозначный в пределах объекта)

Представление каталога sys.database_principals возвращает строку для каждого принципала системы безопасности (т. е. пользователя, группы или роли в базе данных). (*Принципалы* системы безопасности подробно рассматриваются в *главе 12*.) Наиболее важные столбцы представления каталога sys.database_principals перечислены в табл. 9.3.

Таблица 9.3. *Наиболее важные столбцы представления каталога* sys.database_principals

Имя столбца	Описание
name	Имя принципала системы безопасности
principal_id	Идентификатор принципала (однозначный в пределах базы данных)
type	Тип принципала

ПРИМЕЧАНИЕ

В SQL Server 2012 продолжают поддерживаться так называемые представления совместимости с целью обеспечения обратной совместимости. Каждое представление совместимости имеет такое же имя и такую же структуру, как и соответствующая системная базовая таблица системы SQL Server 2000. Представления совместимости не предоставляют доступ к метаданным, связанным с возможностями, введенными, начиная с версии SQL Server 2005. Эти представления являются устаревшей возможностью и будут изъяты из будущих версий SQL Server.

Запросы к представлениям каталога

Как уже упоминалось ранее в этой главе, все системные таблицы имеют такую же структуру, как и базовые таблицы. Поскольку к системным таблицам нельзя обращаться напрямую, необходимо выполнять запросы к представлениям каталога, соответствующим требуемым системным таблицам. В примерах 9.1—9.3 показано получение информации об объектах баз данных, используя существующие представления каталога.

Пример 9.1. Выборка идентификатора таблицы, идентификатора пользователя и типа таблицы для таблицы employee

```
USE sample;
SELECT object_id, principal_id, type
   FROM sys.objects
   WHERE name = 'employee';
```

Результат исполнения этого запроса:

object_id	principal_id	type
530100929	NULL	U

В столбце `object_id` представления каталога `sys.objects` отображается однозначный идентификационный номер соответствующего объекта базы данных. Значение `NULL` в столбце `principal_id` указывает, что владелец данного объекта тот же самый, что и владелец схемы. Значение `U` в столбце `type` означает пользовательскую таблицу (U — от англ. *user*, пользователь).

Пример 9.2. Выборка имен всех таблиц базы данных `sample`, которые содержат столбец `project_no`

```
USE sample;
SELECT sys.objects.name
   FROM sys.objects INNER JOIN sys.columns
   ON sys.objects.object_id = sys.columns.object_id
   WHERE sys.objects.type = 'U'
   AND sys.columns.name = 'project_no';
```

Результат выполнения этого запроса:

```
name
------------------------------------------------------------
project works_on
```

Пример 9.3. Определение владельца таблицы `employee`

```
SELECT sys.database_principals.name
FROM sys.database_principals INNER JOIN sys.objects
ON sys.database_principals,principal_id = sys.objects.schema_id
WHERE sys.objects.name = 'employee'
AND sys.objects.type = 'U';
```

Результат выполнения этого запроса:

```
name
------------------------------------------------------------
dbo
```

Динамические административные представления и функции

Динамические административные представления (ДАП) (Dynamic Management Views, DMV) и динамические административные функции (ДАФ) (Dynamic Management Functions, DMF) возвращают информацию о состоянии сервера, которую можно применить для наблюдения над активными процессами и, следственно, для настройки производительности системы или для отслеживания действительного состояния системы. В отличие от представлений каталога ДАП (DMV) и ДАФ (DMF) основаны на внутренних структурах системы.

ПРИМЕЧАНИЕ

Основное различие между представлениями каталога и ДАП состоит в их применении: представления каталога предоставляют информацию о метаданных, тогда как ДАП (DMV) и ДАФ (DMF) применяются для доступа к динамическим свойствам системы. Иными словами, ДАП применяются для получения информации о базе данных, об отдельных запросах или отдельных пользователях.

ДАП (DMV) и ДАФ (DMF) принадлежат к схеме `sys`, а их имена состоят из префикса `dm_` и текстовой строки, указывающей категорию, к которой принадлежит ДАП или ДАФ.

В следующем списке перечислены некоторые из этих категорий и даны их краткие описания:

- `sys.dm_db_*` — возвращает информацию о базах данных и их объектах;
- `sys.dm_tran_*` — возвращает информацию, имеющую отношение к транзакциям;
- `sys.dm_io_*` — возвращает информацию о действиях по вводу/выводу;
- `sys.dm_exec_*` — возвращает информацию, связанную с исполнением пользовательского кода.

ПРИМЕЧАНИЕ

В каждой новой версии SQL Server корпорация Microsoft непрерывно увеличивает количество поддерживаемых ДАП. Так SQL Server 2012 содержит 20 новых ДАП, а их общее число равно 155.

В этом разделе рассматриваются два новых ДАП:

- `sys.dm_exec_describe_first_result_set`;
- `sys.dm_db_uncontained_entities`.

Представление `sys.dm_exec_describe_first_result_set` описывает первый результирующий набор группы результирующих наборов. Поэтому ДАП можно использовать при объявлении нескольких последовательных запросов в пакете или хранимой процедуре (см. пример 9.4).

Представление `sys.dm_db_uncontained_entities` отображает все неограниченные объекты, используемые в базе данных.

Неограниченными объектами (uncontained objects) называются объекты, которые пересекают границы приложения в автономной базе данных.

Описание неограниченных объектов и границы приложения см. в *разд. "Автономные базы данных" главы 5*.

Пример 9.4. Использование представления `sys.dm_db_uncontained_entities`

```
USE sample;
GO
CREATE PROC TwoSELECTS
```

```
AS
SELECT emp_no, job from works_on where emp_no
                                  BETWEEN 1000 and 9999;
SELECT emp_no, emp_lname FROM employee  where emp_fname LIKE 'S%';
GO
SELECT is_hidden hidden, column_ordinal ord,
                         name, is_nullable nul, system_type_id id
   FROM sys.dm_exec_describe_first_result_set
                                 ('TwoSELECTS1, NULL, 0);
```

Результат выполнения этого запроса:

Hidden	ord	Name	nul	id
0	1	emp_no	0	56
0	2	Job	1	175

Процедура, сохраненная в примере 9.4, содержит две инструкции `SELECT`, осуществляющих выборку из базы данных `sample`. Последующий запрос отображает несколько свойств из результата первого запроса, используя для этого представление `sys.dm_exec_describe_first_result_set`.

ПРИМЕЧАНИЕ

В последующих главах этой книги рассматриваются многие другие ДАП (DMV) и ДАФ (DMF). Например, в следующей главе рассматриваются ДАП и ДАФ для работы с индексами, а в *главе 13* — связанные с транзакциями.

Информационная схема

Информационная схема состоит из представлений, доступных только для чтения, которые предоставляют информацию обо всех таблицах, представлениях и столбцах компонента Database Engine, к которым имеется доступ. В отличие от системного каталога, который управляет метаданными применительно к системе, как к единому целому, информационная схема в основном управляет средой базы данных.

ПРИМЕЧАНИЕ

Информационная схема была впервые представлена в стандарте SQL92. Компонент Database Engine предоставляет представления информационной схемы, чтобы разработанные на других системах баз данных приложения смогли получить свой системный каталог, не используя его прямым образом. Эти стандартные представления используют разную терминологию, поэтому при использовании имен столбцов имейте в виду, что "каталог" является синонимом базы данных, а "домен" — синонимом пользовательского типа данных.

В следующих разделах описываются наиболее важные представления информационной схемы.

Представление *information_schema.tables*

Представление `information_schema.tables` возвращает одну строку для каждой таблицы в текущей базе данных, к которой пользователь имеет доступ. Это представление извлекает информацию из системного каталога, используя представление каталога `sys.objects`. В табл. 9.4 дано описание четырех столбцов этого представления.

Таблица 9.4. *Представление* `information_schema.tables`

Столбец	Описание
TABLE_CATALOG	Имя каталога (базы данных), к которому принадлежит представление
TABLE_SCHEMA	Имя схемы, к которой принадлежит представление
TABLE_NAME	Наименование таблицы
TABLE_TYPE	Тип таблицы (может быть BASE TABLE или VIEW)

Представление *information_schema.columns*

Представление `information_schema.columns` возвращает одну строку для каждого столбца, доступного текущему пользователю в текущей базе данных. Это представление извлекает информацию из системного каталога, используя представления каталога `sys.columns` и `sys.objects`. В табл. 9.5 приведено описание шести наиболее важных столбцов этого представления.

Таблица 9.5. *Представление* `information_schema.columns`

Столбец	Описание
TABLE_CATALOG	Имя каталога (базы данных), к которому принадлежит столбец
TABLE_SCHEMA	Имя схемы, к которой принадлежит столбец
TABLE_NAME	Имя таблицы, к которой принадлежит столбец
COLUMN_NAME	Имя столбца
ORDINAL_POSITION	Номер по порядку столбца
DATAJYPE	Тип данных столбца

Специализированные интерфейсы

В предшествующем разделе мы рассмотрели получение доступа к системным базовым таблицам, используя общие интерфейсы. Но системную информацию можно также получить, используя один из следующих специализированных механизмов компонента Database Engine:

- системные хранимые процедуры;
- системные функции;
- функции свойств.

Эти интерфейсы рассматриваются в последующих подразделах.

Системные хранимые процедуры

Системные хранимые процедуры применяются для выполнения многих административных и пользовательских задач, таких как переименование объектов базы данных, идентификация пользователей и мониторинг авторизации и ресурсов. Для извлечения и модифицирования системной информации почти все существующие системные хранимые процедуры используют системные базовые таблицы.

ПРИМЕЧАНИЕ

Наиболее важным свойством системных хранимых процедур является предоставляемая ими возможность легко и надежно модифицировать системные базовые таблицы.

В этом разделе рассматриваются две системные хранимые процедуры: `sp_help` и `sp_configure`.

Некоторые из системных хранимых процедур обсуждались в предшествующих главах с соответствующей тематикой; дополнительные процедуры рассматриваются в дальнейших главах.

Системная процедура `sp_help` возвращает сведения об объектах базы данных. Параметром для этой процедуры может служить имя любого объекта базы данных или типа данных. При выполнении процедуры `sp_help` без параметра она возвращает информацию обо всех объектах текущей базы данных.

Хранимая системная процедура `sp_configure` возвращает или изменяет глобальные конфигурационные параметры текущего сервера.

В примере 9.5 показано использование системной процедуры `sp_configure`.

Пример 9.5. Использование системной процедуры `sp_configure`

```
USE sample;
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE WITH OVERRIDE;
EXEC sp_configure 'fill factor', 100;
RECONFIGURE WITH OVERRIDE;
```

Обычно, доступ к дополнительным конфигурационным опциям SQL Server не разрешен. Поэтому первая инструкция `EXECUTE` в примере 9.5 дает указание системе разрешить изменения дополнительных параметров. Следующая инструкция, `RECONFIGURE WITH OVERRIDE`, устанавливает это разрешение. Теперь можно изменять

значения дополнительных параметров. В примере 9.5 параметру `fill factor` присваивается значение `100` (инструкция `EXEC`), после чего выполняется установка этого изменения (инструкция `RECONFIGURE`). (Параметр `fill factor` указывает объем хранилища в процентах для страниц индексов и подробно рассматривается в *главе 10*.)

Системные функции

Системные функции рассматриваются в *главе 5*. Некоторые из этих функций можно использовать для доступа к базовым таблицам. В примере 9.6 показано использование инструкции `SELECT` для выборки одной и той же информации, используя два разных интерфейса.

Пример 9.6. Два разных способа выборки информации

```
USE sample;
SELECT object_id
FROM sys.objects
WHERE name = 'employee';
SELECT object_id('employee');
```

В примере 9.6 вторая инструкция `SELECT` возвращает идентификатор таблицы `employee`, используя системную функцию `object_id`. (Полученную информацию можно сохранить в переменной и применить в качестве параметра при вызове команды или системной хранимой процедуры.)

Далее приводится список некоторых системных функций для доступа к системным базовым таблицам. Назначение этих функций должно быть понятно с их названий.

- `OBJECT_ID(`*`object_name`*`)`
- `OBJECT_NAME(`*`object_id`*`)`
- `USER_ID([user_name])`
- `USER_NAME([`*`user_id`*`])`
- `DB_ID([`*`db_name`*`])`
- `DB_NAME([`*`db_id`*`])`

Функции свойств

Функции свойств возвращают свойства объектов баз данных, типов данных и файлов. Обычно функции свойств могут возвратить больше информации, чем системные функции, т. к. функции свойств поддерживают десятки свойств (в виде параметров), которые можно явно указывать.

Почти все функции свойств возвращают одно из следующих значений: `0`, `1` или `NULL`. Возвращение функцией свойств значения `0` означает, что объект не обладает данным свойством. Возвращение функцией свойств значения `1` означает, что объ-

ект обладает данным свойством. А возвращение значения NULL означает, что системе неизвестно, обладает ли данный объект указанным свойством.

Компонент Database Engine поддерживает, среди прочих, следующие функции свойств:

- OBJECTPROPERTY(*id*, *property*)
- COLUMNPROPERTY(*id*, *column*, *property*)
- FILEPROPERTY(*filename*, *property*)
- TYPEPROPERTY(*type*, *property*)

Функция OBJECTPROPERTY возвращает информацию об объектах текущей базы данных (см. упражнение 9.2). Функция COLUMNPROPERTY возвращает информацию о столбце или о параметре процедуры. Функция FILEPROPERTY возвращает значение указанного свойства для заданного имени файла. Функция TYPEPROPERTY возвращает информацию о типе данных. (Описание поддерживаемых свойств для всех функций свойств см. в *электронной документации*.)

Резюме

Системный каталог представляет собой набор системных базовых таблиц базы данных master и существующих пользовательских баз данных. Обычно, пользователи не могут обращаться к системным таблицам напрямую. Компонент Database Engine поддерживает несколько разных интерфейсов, которые можно использовать для запросов данных из системного каталога. Наиболее общим интерфейсом для получения системной информации являются представления каталога. Динамические административные представления (ДАП) (Dynamic Management Views, DMV) и динамические административные функции (ДАФ) (Dynamic Management Functions, DMF) подобны представлениям каталога, но применяются для доступа к динамическим свойствам системы. Системные хранимые процедуры предоставляют легкий и надежный доступ к системным базовым таблицам для чтения и записи. Для изменения системной информации настоятельно рекомендуется использовать исключительно системные хранимые процедуры.

Информационная схема — это коллекция представлений, определенных для системных базовых таблиц, которая предоставляет единообразный метод доступа к системному каталогу для всех приложений баз данных, разработанных на других системах баз данных. Использование информационной схемы рекомендуется, если планируется перенос системы базы данных на другой тип системы баз данных.

В следующей главе мы познакомимся с индексами баз данных.

Упражнения

Упражнение 9.1

Используя представления каталога, определите путь и имя файла базы данных sample.

Упражнение 9.2

Используя представления каталога, определите количество определенных ограничений для обеспечения целостности для таблицы `employee` базы данных `sample`.

Упражнение 9.3

Используя представления каталога, определите наличие или отсутствие ограничения для обеспечения целостности для столбца `dept_no` таблицы `employee`.

Упражнение 9.4

Используя информационную схему, выполните выборку всех пользовательских таблиц базы данных `AdventureWorks2012`.

Упражнение 9.5

Используя информационную схему, определите все столбцы таблицы `employee`, их порядок и тип данных.

Глава 10

Индексы

- **Общие сведения**
- **Язык Transact-SQL и индексы**
- **Рекомендации по созданию и использованию индексов**
- **Специальные типы индексов**

В этой главе рассматриваются индексы и их роль в оптимизации времени выполнения запросов. В первой части главы обсуждаются разные формы индексов и способы их хранения. В основной части главы исследуются три основные инструкции языка Transact-SQL, применяемые для работы с индексами: `CREATE INDEX`, `ALTER INDEX` и `DROP INDEX`. Далее рассматривается фрагментация индексов ее влияния на производительность системы. После этого дается несколько общих рекомендаций по созданию индексов. В последней части главы описывается несколько специальных типов индексов.

Общие сведения

Системы баз данных обычно используют индексы для обеспечения быстрого доступа к реляционным данным. *Индекс* представляет собой отдельную физическую структуру данных, которая позволяет получать быстрый доступ к одной или нескольким строкам данных. Таким образом, правильная настройка индексов является ключевым аспектом улучшения производительности запросов.

Индекс базы данных во многом сходен с индексом (алфавитным указателем) книги. Когда нам нужно быстро найти какою-либо тему в книге, мы сначала смотрим в индексе, на каких страницах книги эта тема рассматривается, а потом сразу же открываем нужную страницу. Подобным образом, при поиске определенной строки таблицы компонент Database Engine обращается к индексу, чтобы узнать ее физическое местонахождение.

Но между индексом книги и индексом базы данных есть две существенные разницы.

- Читатель книги имеет возможность самому решать, использовать ли индекс в каждом конкретном случае или нет. Пользователь базы данных такой возможности не имеет, и за него это решение принимает компонент системы, называемый *оптимизатором запросов*. (Пользователь может манипулировать использованием индексов посредством подсказок индексов, но эти подсказки рекомендуется применять только в ограниченном числе специальных случаев. Подсказки индексов рассматриваются в *главе 19*.)
- Индекс для определенной книги создается вместе с книгой, после чего он больше не изменяется. Это означает, что индекс для определенной темы всегда будет указывать на один и тот же номер страницы. В противоположность, индекс базы данных может меняться при каждом изменении соответствующих данных.

Если для таблицы отсутствует подходящий индекс, для выборки строк система использует метод сканирования таблицы. Выражение *сканирование таблицы* означает, что система последовательно извлекает и исследует каждую строку таблицы (от первой до последней), и помещает строку в результирующий набор, если для нее удовлетворяется условие поиска в предложении `WHERE`. Таким образом, все строки извлекаются в соответствии с их физическим расположением в памяти. Этот метод менее эффективен, чем доступ с использованием индексов, как объясняется далее.

Индексы сохраняются в дополнительных структурах базы данных, называющихся *страницами индексов*. (Структура страниц индексов очень похожа на структуру страниц данных, как мы увидим это в *главе 15*.)

Для каждой индексируемой строки имеется *элемент индекса* (index entry), который сохраняется на странице индексов. Каждый элемент индекса состоит из ключа индекса и указателя. Вот поэтому элемент индекса значительно короче, чем строка таблицы, на которую он указывает. По этой причине количество элементов индекса на каждой странице индексов намного больше, чем количество строк в странице данных. Это свойство индексов играет очень важную роль, поскольку количество операций ввода/вывода, требуемых для прохода по страницам индексов, значительно меньше, чем количество операций ввода/вывода, требуемых для прохода по соответствующим страницам данных. Иными словами, для сканирования таблицы, скорей всего, потребовалось бы намного больше операций ввода/вывода, чем для сканирования индекса этой таблицы.

Индексы компонента Database Engine создаются, используя структуру данных сбалансированного дерева B^+. B^+-дерево имеет древовидную структуру, в которой все самые нижние узлы (листья) находятся на расстоянии одинакового количества уровней от вершины (корневого узла) дерева. Это свойство поддерживается даже тогда, когда в индексированный столбец добавляются или удаляются данные.

На рис. 10.1 показана структура B^+-дерева для таблицы `employee` и прямой доступ к строке в этой таблице со значением `25348` для столбца `emp_no`. (Предполагается, что таблица `employee` проиндексирована по столбцу `emp_no`.) На этом рисунке можно также видеть, что B^+-дерево состоит из корневого узла, листьев дерева (узлов-листьев) и промежуточных узлов, количество которых может быть от нуля и больше.

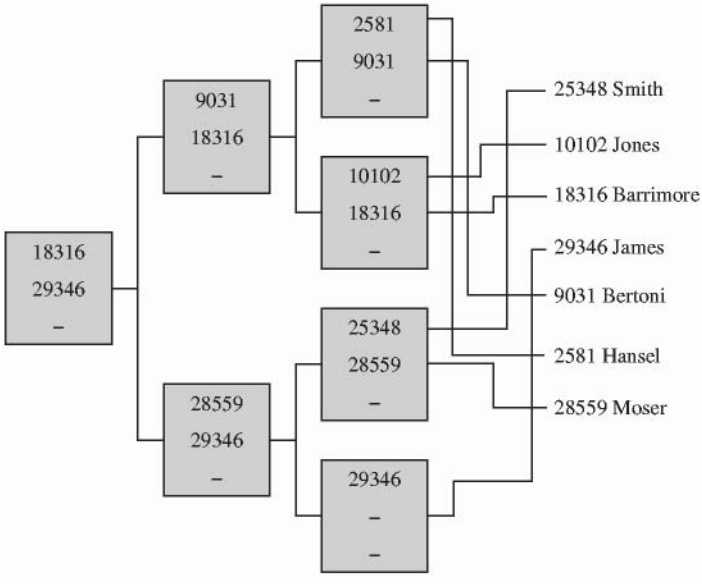


Рис. 10.1. B+-дерево для столбца `emp_no` таблицы `employee`

Поиск в этом дереве значения `25348` можно выполнить следующим образом. Начиная с корня дерева, выполняется поиск наименьшего значения ключа, большего или равного требуемому значению. Таким образом, в корневом узле таким значением будет `29346`, поэтому делается переход на промежуточный узел, связанный с этим значением. В этом узле заданным требованиям отвечает значение `28559`, вследствие чего выполняется переход на листья дерева, связанный с этим значением. Этот узел и содержит искомое значение `25348`. Определив требуемый индекс, мы можем извлечь его строку из таблицы данных с помощью соответствующих указателей. (Альтернативным эквивалентным подходом будет поиск меньшего или равного значения индекса.)

Индексированный поиск обычно является предпочтительным методом поиска в таблицах с большим количеством строк по причине его очевидного преимущества. Используя индексированный поиск, мы можем найти любую строку в таблице за очень короткое время, применив лишь несколько операций ввода/вывода. А последовательный поиск (т. е. сканирование таблицы от первой строки до последней) требует тем больше времени, чем дальше находится требуемая строка.

В следующих разделах мы рассмотрим два существующих типа индексов, кластеризованные и некластеризованные, а также научимся создавать индексы.

Кластеризованные индексы

Кластеризованный индекс определяет физический порядок данных в таблице. Компонент Database Engine позволяет создавать для таблицы лишь один кластеризованный индекс, т. к. строки таблицы нельзя упорядочить физически более чем одним способом. Поиск с использованием кластеризованного индекса выполняется

от корневого узла B^+-дерева по направлению к листьям дерева, которые связаны между собой в двунаправленный связанный список (doubly linked list), называющийся *цепочкой страниц* (page chain). Важным свойством кластеризованного индекса является та особенность, что его листья дерева (узлы-листья) содержат страницы данных. (Узлы кластеризованного индекса всех других уровней содержат страницы индекса.) Таблица, для которой определен кластеризованный индекс (явно или неявно), называется *кластеризованной таблицей*. Структура B^+-дерева кластеризованного индекса показана на рис. 10.2.

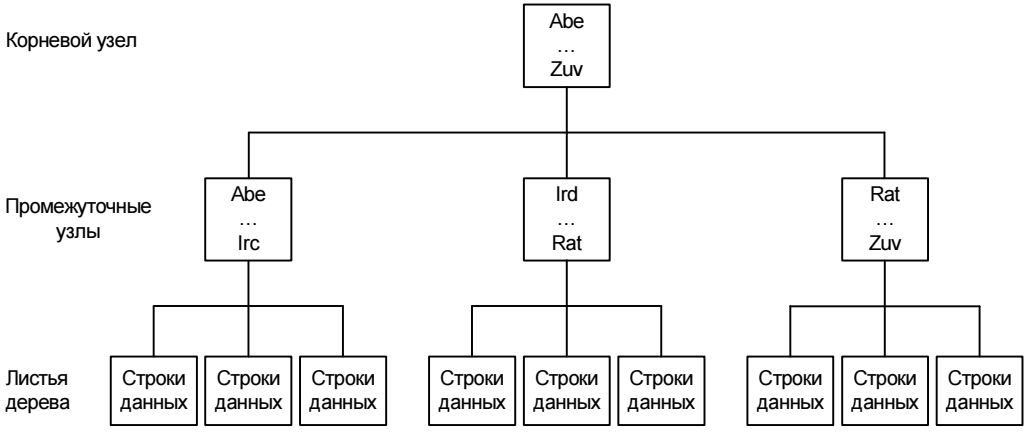


Рис. 10.2. Физическая структура кластеризованных индексов

Кластеризованный индекс создается по умолчанию для каждой таблицы, для которой с помощью ограничения первичного ключа определен первичный ключ. Кроме этого, каждый кластеризованный индекс однозначен по умолчанию, т. е. в столбце, для которого определен кластеризованный индекс, каждое значение данных может встречаться только один раз. Если кластеризованный индекс создается для столбца, содержащего повторяющиеся значения, система баз данных принудительно обеспечивает однозначность, добавляя четырехбайтовый идентификатор к строкам, содержащим дубликаты значений.

ПРИМЕЧАНИЕ

Кластеризованные индексы обеспечивают очень быстрый доступ к данным, когда запрос осуществляет поиск в диапазоне значений (см. *главу 19*).

Некластеризованные индексы

Структура некластеризованного индекса точно такая же, как и кластеризованного, но с двумя важными отличиями:

- некластеризованный индекс не изменяет физическое упорядочивание строк таблицы;

- сраницы листьев некластеризованного индекса состоят из ключей индекса и закладок.

Если для таблицы определить один или более некластеризованных индексов, физический порядок строк этой таблицы не будет изменен. Для каждого некластеризованного индекса компонент Database Engine создает дополнительную индексную структуру, которая сохраняется в индексных страницах. Структура B+-дерева некластеризованного индекса показана на рис. 10.3.

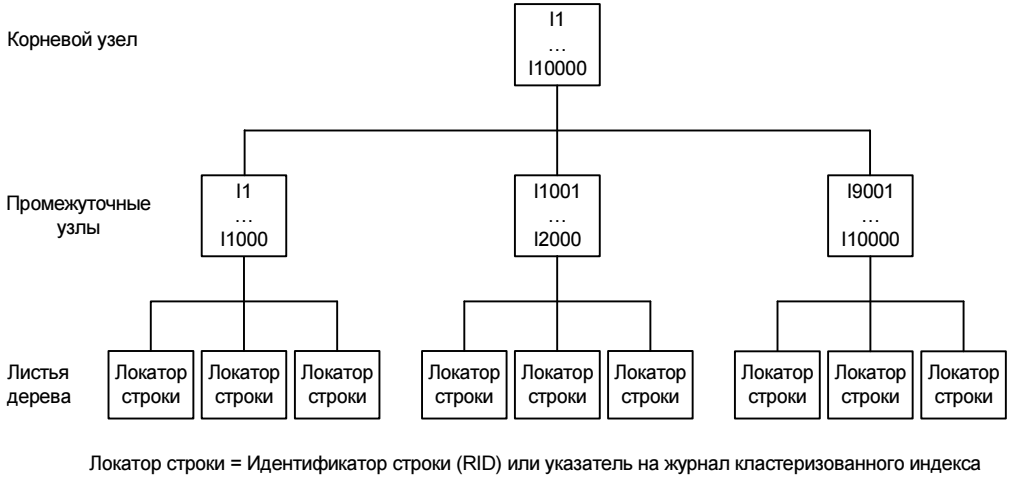


Рис. 10.3. Структура некластеризованного индекса

Закладка в некластеризованном индексе указывает, где находится строка, соответствующая ключу индекса. Составляющая закладки ключа индекса может быть двух видов, в зависимости от того, является ли таблица кластеризованной таблицей или кучей (heap). (Согласно терминологии SQL Server, *кучей* называется таблица без кластеризованного индекса.) Если существует кластеризованный индекс, то закладка некластеризованного индекса показывает B^+-дерево кластеризованного индекса таблицы. Если таблица не имеет кластеризованного индекса, закладка идентична идентификатору строки (RID — Row Identifier), состоящего из трех частей: адреса файла, в котором хранится таблица, адреса физического блока (страницы), в котором хранится строка, и смещения строки в странице.

Как уже упоминалось ранее, поиск данных с использованием некластеризованного индекса можно осуществлять двумя разными способами, в зависимости от типа таблицы:

- *куча* — прохождение при поиске по структуре некластеризованного индекса, после чего строка извлекается, используя идентификатор строки;
- *кластеризованная таблица* — прохождение при поиске по структуре некластеризованного индекса, после чего следует прохождение по соответствующему кластеризованному индексу.

В обоих случаях количество операций ввода/вывода довольно велико, поэтому следует подходить к проектированию некластеризованного индекса с осторожностью, и применять его только в том случае, если есть уверенность, что его использование существенно повысит производительность.

Язык Transact-SQL и индексы

Теперь, когда мы познакомились с физической структурой индексов, в этом разделе рассмотрим, как их создавать, изменять и удалять, а также как получать информацию о фрагментации индексов и редактировать информацию об индексах. Все это подготовит нас к последующему обсуждению использования индексов для улучшения производительности системы.

Создание индексов

Индекс для таблицы создается с помощью инструкции `CREATE INDEX`. Эта инструкция имеет следующий синтаксис:

```
CREATE [UNIQUE] [CLUSTERED |NONCLUSTERED1 INDEX index_name
   ON table_name (column1 [ASC DESC],...)
      [INCLUDE (colunm_name [,... ])]
   [WITH
      [FILLFACTOR=n]
      [[,] PAD_INDEX = {ON | OFF}]
      [[,] DROP_EXISTING = {ON | OFF}]
      [[,] SORT_IN_TEMPDB = {ON | OFF}]
      [[,] IGNORE_DUP_KEY = {ON | OFF}]
      [[,] ALLOW_ROW_LOCKS = {ON | OFF}]
      [[,] ALLOW_PAGE_LOCKS = {ON | OFF}]
      [[,] STATISTICS_NORECOMPUTE = {ON | OFF}]
      [[,] ONLINE = {ON | OFF}]]
      [ON file_group | "default"]
```

Параметр *`index_name`* задает имя создаваемого индекса. Индекс можно создать для одного или больше столбцов одной таблицы, обозначаемой параметром *`table_name`*. Столбец, для которого создается индекс, указывается параметром *`column1`*. Числовой суффикс этого параметра указывает на то, что индекс можно создать для нескольких столбцов таблицы. Компонент Database Engine также поддерживает создание индексов для представлений. Такие представления, называемые, соответственно, *индексированными представлениями*, рассматриваются в *главе 11*.

ПРИМЕЧАНИЕ

Можно проиндексировать любой столбец таблицы. Это означает, что столбцы, содержащие значения типа данных `VARBINARY(max)`, `BIGINT` и `SQL_ VARIANT`, также могут быть индексированы.

Индекс может быть простым или составным. *Простой индекс* создается по одному столбцу, а *составной индекс* — по нескольким столбцам. Для составного индекса существуют определенные ограничения, связанные с его размером и количеством столбцов. Индекс может иметь максимум 900 байтов и не более 16 столбцов.

Параметр `UNIQUE` указывает, что проиндексированный столбец может содержать только однозначные (т. е. неповторяющиеся) значения. В однозначном составном индексе однозначной должна быть комбинация значений всех столбцов каждой строки. Если ключевое слово `UNIQUE` не указывается, то повторяющиеся значения в проиндексированном столбце (столбцах) разрешаются.

Параметр `CLUSTERED` задает кластеризованный индекс, а параметр `NONCLUSTED` (применяется по умолчанию) указывает, что индекс не изменяет порядок строк в таблице. Компонент Database Engine разрешает для таблицы максимум 249 некластеризованных индексов.

Возможности компонента Database Engine были расширены, позволяя поддержку индексов с убывающим порядком значений столбцов. Параметр `ASC` после имени столбца указывает, что индекс создается с возрастающим порядком значений столбца, а параметр `DESC` означает убывающий порядок значений столбца индекса. Таким образом, в использовании индекса предоставляется большая гибкость. С убывающим порядком следует создавать составные индексы на столбцах, значения которых упорядочены в противоположных направлениях.

Параметр `INCLUDE` позволяет указать неключевые столбцы, которые добавляются к страницам листьев некластеризованного индекса. Имена столбцов в списке `INLCUDE` не должны повторяться, и столбец нельзя использовать одновременно как ключевой и неключевой. Чтобы по-настоящему понять полезность параметра `INCLUDE`, нужно понимать, что собой представляет покрывающий индекс (covering index). Если все столбцы запроса включены в индекс, то можно получить значительное повышение производительности, т. к. оптимизатор запросов может определить местонахождение всех значений столбцов по страницам индекса, не обращаясь к данным в таблице. Такая возможность называется *покрывающим индексом* или *покрывающим запросом*. Поэтому включение в страницы листьев некластеризованного индекса дополнительных неключевых столбцов позволит получить больше покрывающих запросов, при этом их производительность будет значительно повышена. (Более подробно этот предмет рассматривается далее в *разд. "Покрывающий индекс" этой главы*, в котором также приводится пример обработки покрывающего индекса оптимизатором.)

Параметр `FILEFACTOR=n` задает заполнение в процентах каждой страницы индекса во время его создания. Значение параметра `FILEFACTOR` можно установить в диапазоне от 1 до 100. При значении `n=100` каждая страница индекса заполняется на 100%, т. е. существующая страница листа так же, как страница, не относящаяся к листу, не будет иметь свободного места для вставки новых строк. Поэтому это значение рекомендуется применять только для статических таблиц. (Значение по умолчанию, `n=0`, означает, что страницы листьев индекса заполняются полностью, а каждая из промежуточных страниц содержит свободное место для одной записи.)

При значении параметра FILEFACTOR между 1 и 99 страницы листьев создаваемой структуры индекса будут содержать свободное место. Чем больше значение n, тем меньше свободного места в страницах листьев индекса. Например, при значении n=60 каждая страница листьев индекса будет иметь 40% свободного места для вставки строк индекса в дальнейшем. (Строки индекса вставляются посредством инструкции INSERT или UPDATE.) Таким образом, значение n=60 будет разумным для таблиц, данные которых подвергаются довольно частым изменениям. При значениях параметра FILEFACTOR между 1 и 99 промежуточные страницы индекса содержат свободное место для одной записи каждая.

ПРИМЕЧАНИЕ

После создания индекса в процессе его использования значение FILEFACTOR не поддерживается. Иными словами, оно только указывает объем зарезервированного места с имеющимися данными при задании процентного соотношения для свободного места. Для восстановления исходного значения параметра FILEFACTOR применяется инструкция ALTER INDEX, которая рассматривается далее в этой главе.

Параметр PAD_INDEX тесно связан с параметром FILEFACTOR. Параметр FILEFACTOR в основном задает объем свободного пространства в процентах от общего объема страниц листьев индекса. А параметр PAD_INDEX указывает, что значение параметра FILEFACTOR применяется как к страницам индекса, так и к страницам данных в индексе.

Параметр DROP_EXISTING позволяет повысить производительность при воспроизведении кластеризованного индекса для таблицы, которая также имеет некластеризованный индекс. Более подробную информацию см. далее в *разд. "Пересоздание индекса" этой главы*.

Параметр SORT_IN_TEMPDB применяется для помещения в системную базу данных tempdb данных промежуточных операций сортировки, применяющихся при создании индекса. Это может повысить производительность, если база данных tempdb размещена на другом диске, чем данные.

Параметр IGNORE_DUP_KEY разрешает системе игнорировать попытку вставки повторяющихся значений в индексированные столбцы. Этот параметр следует применять только для того, чтобы избежать прекращения выполнения длительной транзакции, когда инструкции INSERT вставляет дубликат данных в индексированный столбец. Если этот параметр активирован, то при попытке инструкции INSERT вставить в таблицу строки, нарушающие однозначность индекса, система базы данных вместо аварийного завершения выполнения всей инструкции просто выдает предупреждение. При этом компонент Database Engine *не вставляет* строки с дубликатами значений ключа, а просто игнорирует их и добавляет правильные строки. Если же этот параметр не установлен, то выполнение всей инструкции будет аварийно завершено.

Когда параметр ALLOW_ROW_LOCKS активирован (имеет значение ON), система применяет блокировку строк. Подобным образом, когда активирован параметр

ALLOW_PAGE_LOCKS, система применяет блокировку страниц. (Блокировка страниц и строк рассматривается в *главе 13*.)

Параметр STATISTICS_NORECOMPUTE определяет состояние автоматического перерасчета статистики указанного индекса. (Статистика рассматривается в *главе 19*.)

Активированный параметр ONLINE позволяет создавать, пересоздавать (rebuild) и удалять индекс в диалоговом режиме. Данный параметр позволяет в процессе изменения индекса одновременно изменять данные основной таблицы или кластеризованного индекса и любых связанных индексов. Например, в процессе пересоздания кластеризованного индекса можно продолжать обновлять его данные и выполнять запросы по этим данным.

Параметр ON создает указанный индекс или на файловой группе по умолчанию (значение default), или на указанной файловой группе (значение file_group).

ПРИМЕЧАНИЕ

Перед тем как запускать на выполнение примеры запросов этой главы, необходимо заново создать базу данных sample.

В примере 10.1 показано создание некластеризованного индекса.

Пример 10.1. Создание индекса для emp_no таблицы employee

```
USE sample;
CREATE INDEX i_empno ON employee (emp_no);
```

Создание однозначного составного индекса показано в примере 10.2.

Пример 10.2. Создание составного индекса для столбцов emp_no и project_no таблицы works_on

```
USE sample;
CREATE UNIQUE INDEX i_empno_prno
ON works_on (emp_no, project_no)
WITH FILLFACTOR= 80;
```

В примере 10.2 значения в каждом столбце должны быть однозначными. При создании индекса заполняется 80% пространства каждой страницы листьев индекса.

Создание однозначного индекса для столбца невозможно, если этот столбец содержит повторяющиеся значения. Такой индекс можно создать лишь в том случае, если каждое значение (включая значение NULL) встречается в столбце только один раз. Кроме этого, любая попытка вставить или изменить существующее значение данных в столбец, включенный в существующий уникальный индекс, будет отвергнута системой в случае дублирования значения.

Получение информации о фрагментации индекса

В течение жизненного цикла индекса он может подвергнуться *фрагментации*, вследствие чего процесс хранения данных в страницах индекса станет неэффективным. Существует два типа фрагментации индекса: внутренняя фрагментация и внешняя фрагментация. Внутренняя фрагментация определяет объем данных, хранящихся в каждой странице, а внешняя фрагментация возникает при нарушении логического порядка страниц.

Для получения информации о внутренней фрагментации индекса применяется динамическое административное представление (ДАП), называемое `sys.dm_db_index_physical_stats`. Это ДАП (DMV) возвращает информацию об объеме и фрагментации данных и индексов указанной страницы. Для каждой страницы возвращается одна строка для каждого уровня B^+-дерева. С помощью этого ДАП можно получить информацию о степени фрагментации строк в страницах данных, на основе которой можно принять решение о необходимости реорганизации данных.

Использование представления `sys.dm_db_index_physical_stats` показано в примере 10.3. (Прежде чем запускать пакет в примере 10.3 на выполнение, необходимо удалить все существующие индексы таблицы `works_on`. Для удаления индексов используется инструкция `DROP INDEX`, применение которой показано в примере 10.4.)

Пример 10.3. Использование ДАП `sys.dm_db_index_physical_stats`

```
DECLARE @db_id INT;
DECLARE @tab_id INT;
DECLARE @ind_id INT;
SET @db_id = DB_ID('sample');
SET @tab_id = OBJECT_ID('employee');
SELECT avg_fragmentation_in_percent, avg_page_space_used_in_percent
   FROM sys.dm_db_index_physical_stats
   (@db_id, @tab_id, NULL, NULL, NULL)
```

Как видно из примера 10.3, представление `sys.dm_db_index_physical_stats` имеет пять параметров. Первые три параметра определяют идентификаторы текущей базы данных, таблицы и индекса соответственно. Четвертый параметр задает идентификатор раздела (см. *главу 25*), а последний определяет уровень сканирования, применяемый для получения статистической информации. (Значение по умолчанию для определенного параметра можно указать посредством значения `NULL`.)

Наиболее важными из столбцов этого представления являются столбцы `avg_fragmentation_in_percent` и `avg_ page_space_used_in_percent`. В первом указывается средний уровень фрагментации в процентах, а во втором определяется объем занятого пространства в процентах.

Редактирование информации индекса

После ознакомления с информацией о фрагментации индекса, как было рассмотрено в предыдущем разделе, эту и другую информацию индекса можно редактировать с помощью следующих системных средств:

- представления каталога `sys.indexes`;
- представления каталога `sys.index_columns`;
- системной процедуры `sp_helpindex`;
- функции свойств `OBJECTPROPERTY`;
- среды управления Management Studio сервера SQL Server;
- динамического административного представления (ДАП) `sys.dm_db_index_usage_stats`;
- динамического административного представления (ДАП) `sys.dm_db_missing_index_details`.

Представление каталога `sys.indexes` содержит строку для каждого индекса и строку для каждой таблицы без кластеризованного индекса. Наиболее важными столбцами этого представления каталога являются столбцы `object_id`, `name` и `index_id`. Столбец `object_id` содержит имя объекта базы данных, которой принадлежит индекс, а столбцы `name` и `index_id` содержат имя и идентификатор этого индекса соответственно.

Представление каталога `sys.index_columns` содержит строку для каждого столбца, являющегося частью индекса или кучи. Эту информацию можно использовать совместно с информацией, полученной посредством представления каталога `sys.indexes`, для получения дополнительных сведений о свойствах указанного индекса.

Системная процедура `sp_helpindex` возвращает данные об индексах таблицы, а также статистическую информацию для столбцов. Эта процедура имеет следующий синтаксис:

```
sp_helpindex [@db_object = ] 'name'
```

Здесь переменная *`@db_object`* представляет имя таблицы.

Применительно к индексам, функция свойств `OBJECTPROPERTY` имеет два свойства: `IsIndexed` и `IsIndexable`. Первое свойство предоставляет сведения о наличии индекса у таблицы или представления, а второе указывает, поддается ли таблица или представление индексированию.

Для редактирования информации существующего индекса с помощью среды SQL Server Management Studio выберите требуемую базу данных в папке Databases (базы данных), разверните узел Tables (таблицы), в этом узле разверните требуемую таблицу и ее папку Indexes (индексы). В папке таблицы отобразится список всех существующих индексов для данной таблицы. Двойной щелчок мышью по индексу откроет диалоговое окно **Index Properties** со свойствами этого индекса.

(Создать новый индекс или удалить существующий можно также с помощью среды Management Studio.)

Представление `sys.dm_db_index_usage_stats` возвращает подсчет разных типов операций с индексами и время последнего выполнения каждого типа операции. Каждая отдельная операция поиска, просмотра или обновления по указанному индексу при исполнении одного запроса считается использованием индекса и увеличивает на единицу значение соответствующего счетчика в этом ДАП (DMV). Таким образом можно получить общую информацию о частоте использования индекса, чтобы на ее основе определить, какие индексы используются больше, а какие меньше.

Представление `sys.dm_db_missing_index_details` возвращает подробную информацию о столбцах таблицы, для которых отсутствуют индексы. Наиболее важными столбцами этого ДАП являются столбцы `index_handle` и `object_id`. Значение в первом столбце определяет конкретный отсутствующий индекс, а во втором — таблицу, в которой отсутствует индекс.

Изменение индексов

Компонент Database Engine является одной из немногих систем баз данных, которые поддерживают инструкцию `ALTER INDEX`. Эту инструкцию можно использовать для выполнения операций по обслуживанию индекса. Синтаксис инструкции `ALTER INDEX` очень сходен с синтаксисом инструкции `CREATE INDEX`. Иными словами, эта инструкция позволяет изменять значения параметров `ALLOW_ROW_LOCKS`, `ALLOW_PAGE_LOCKS`, `IGNORE_DUP_KEY` и `STATISTICS_NORECOMPUTE`, которые были описаны ранее при рассмотрении инструкции `CREATE INDEX`.

Кроме вышеперечисленных параметров, инструкция `ALTER INDEX` поддерживает три другие параметра:

- параметр `REBUILD`, используемый для пересоздания индекса;
- параметр `REORGANIZE`, используемый для реорганизации страниц листьев индекса;
- параметр `DISABLE`, используемый для отключения индекса.

Эти три параметра рассматриваются в следующих подразделах.

Пересоздание индекса

При любом изменении данных, используя инструкции `INSERT`, `UPDATE` или `DELETE`, возможна фрагментация данных. Если эти данные проиндексированы, то также возможна фрагментация индекса, когда информация индекса оказывается разбросанной по разным физическим страницам. В результате фрагментации данных индекса компонент Database Engine может быть вынужден выполнять дополнительные операции чтения данных, что понижает общую производительность системы. В таком случае требуется пересоздать (rebuild) все фрагментированные индексы.

Это можно сделать двумя способами:

- посредством параметра REBUILD инструкции ALTER INDEX;
- посредством параметра DROP_EXISTING инструкции CREATE INDEX.

Параметр REBUILD применяется для пересоздания индексов. Если для этого параметра вместо имени индекса указать ALL, будут вновь созданы все индексы таблицы. (Разрешив динамическое пересоздание индексов, вам не нужно будет удалять и создавать их заново.)

Параметр DROP_EXISTING инструкции CREATE INDEX позволяет повысить производительность при пересоздании кластеризованного индекса таблицы, которая также имеет некластеризованные индексы. Он указывает, что существующий кластеризованный или некластеризованный индекс нужно удалить и создать заново указанный индекс. Как упоминалось ранее, каждый некластеризованный индекс в кластеризованной таблице содержит в своих листьях дерева соответствующие значения кластеризованного индекса таблицы. По этой причине при удалении кластеризованного индекса таблицы требуется создать вновь все ее некластеризованные индексы. Использование параметра DROP_EXISTING позволяет избежать повторного пересоздания некластеризованных индексов.

ПРИМЕЧАНИЕ

Параметр DROP_EXISTING более мощный, чем параметр REBUILD, поскольку он более гибкий и предоставляет несколько опций, таких как изменение столбцов, составляющих индекс, и изменение некластеризованного индекса в кластеризованный.

Реорганизация страниц листьев индекса

Параметр REORGANIZE инструкции ALTER INDEX задает реорганизацию страниц листьев указанного индекса, чтобы физический порядок страниц совпадал с их логическим порядком — слева направо. Это удаляет определенный объем фрагментации индекса, повышая его производительность.

Отключение индекса

Параметр DISABLE отключает указанный индекс. Отключенный индекс недоступен для применения, пока он не будет снова включен. Обратите внимание, что отключенный индекс не изменяется при внесении изменений в соответствующие данные. По этой причине, чтобы снова использовать отключенный индекс, его нужно полностью создать вновь. Для включения отключенного индекса применяется параметр REBUILD инструкции ALTER TABLE.

ПРИМЕЧАНИЕ

При отключенном кластеризованном индексе таблицы данные этой таблицы будут недоступны, так как все страницы данных таблицы с кластеризованным индексом хранятся в его листьях дерева.

Удаление и переименование индексов

Для удаления индексов в текущей базе данных применяется инструкция DROP INDEX. Обратите внимание, что удаление кластеризованного индекса таблицы может быть очень ресурсоемкой операцией, т. к. потребуется пересоздать все некластеризованные индексы. (Все некластеризованные индексы используют ключ индекса кластеризованного индекса, как указатель в своих страницах листьев.) Использование инструкции DROP INDEX для удаления индекса показано в примере 10.4.

Пример 10.4. Удаление индекса i_empno, созданного в примере 10.1

```
USE sample;
DROP INDEX i_empno ON employee;
```

Инструкция DROP INDEX имеет дополнительный параметр, MOVE TO, значение которого аналогично параметру ON инструкции CREATE INDEX. Иными словами, с помощью этого параметра можно указать, куда переместить строки данных, находящиеся в страницах листьев кластеризованного индекса. Данные перемещаются в новое место в виде кучи. Для нового места хранения данных можно указать или файловую группу по умолчанию, или именованную файловую группу.

ПРИМЕЧАНИЕ

Инструкцию DROP INDEX нельзя использовать для удаления индексов, которые создаются неявно системой для ограничений целостности, таких индексов, как PRIMARY KEY и UNIQUE. Чтобы удалить такие индексы, нужно удалить соответствующее ограничение.

Индексы можно переименовывать с помощью системной процедуры sp_rename, которая рассматривается в *главе 5*.

ПРИМЕЧАНИЕ

Индексы можно также создавать, изменять и удалять в среде Management Studio с помощью диаграмм баз данных или обозревателя объектов. Но самым простым способом будет использовать папку Indexes (индексы) требуемой таблицы. Управление индексами в среде Management Studio аналогично управлению таблицами в этой среде. (Подробную информацию см. в *главе 3*.)

Рекомендации по созданию и использованию индексов

Хотя компонент Database Engine не накладывает никаких практических ограничений на количество индексов, по паре причин это количество следует ограничивать. Во-первых, каждый индекс занимает определенный объем дискового пространства,

следовательно, существует вероятность того, что общее количество страниц индекса базы данных может превысить количество страниц данных в базе. Во-вторых, в отличие от получения выгоды при использовании индекса для выборки данных, вставка и удаление данных такой выгоды не предоставляют по причине необходимости обслуживания индекса. Чем больше индексов имеет таблица, тем больший требуется объем работы по их реорганизации. Общим правилом будет разумно выбирать индексы для частых запросов, а затем оценивать их использование.

Некоторые рекомендации по созданию и использованию индексов предоставляются в этом разделе.

ПРИМЕЧАНИЕ

Последующие рекомендации являются всего лишь общими правилами. В конечном итоге их эффективность будет зависеть от способа использования базы данных на практике и типа наиболее часто выполняемых запросов. Индексирование столбца, который никогда не будет использоваться, не принесет никакой пользы.

Индексы и условия предложения *WHERE*

Если предложение `WHERE` инструкции `SELECT` содержит условие поиска с одним столбцом, то для этого столбца следует создать индекс. Это особенно рекомендуется при высокой селективности условия. Под *селективностью* (selectivity) условия имеется в виду соотношение количества строк, удовлетворяющих условию, к общему количеству строк в таблице. Высокой селективности соответствует меньшему значению этого соотношения. Обработка поиска с использованием индексированного столбца будет наиболее успешной при селективности условия, не превышающей 5%.

Столбец не следует индексировать при постоянном уровне селективности условия 80% или более. В таком случае для страниц индекса потребуются дополнительные операции ввода/вывода, которые уменьшат любую экономию времени, достигаемую за счет использования индексов. В этом случае быстрее выполнять поиск сканированием таблицы, что и будет обычно выбрано оптимизатором запросов, делая индекс бесполезным.

Если условие поиска часто используемого запроса содержит операторы `AND`, лучше всего будет создать составной индекс по всем столбцам таблицы, указанным в предложении `WHERE` инструкции `SELECT`. Создание такого составного индекса показано в примере 10.5.

Пример 10.5. Создание составного индекса по всем столбцам предложения `WHERE`

```
USE sample;
CREATE INDEX i_works ON works_on(emp_no, enter_date);
SELECT emp_no, project_no, enter_date
   FROM works_on
   WHERE emp_no = 29346 AND enter_date='1.4.2006';
```

В этом запросе оператором AND соединены два условия, поэтому для обоих столбцов в этих условиях следует создать составной некластеризованный индекс.

Индексы и оператор соединения

В случае операции соединения рекомендуется создавать индекс для каждого соединяемого столбца. Соединяемые столбцы часто представляют первичный ключ одной из таблицы и соответствующий внешний ключ другой таблицы. Если указываются ограничения для обеспечения целостности PRIMARY KEY и FOREIGN KEY для соответствующих соединяемых столбцов, следует создать только некластеризованный индекс для столбца внешнего ключа, т. к. система неявно создаст кластеризованный индекс для столбца первичного ключа.

В примере 10.6 показано создание индексов, которые будут использованы, если у вас есть запрос с операцией соединения и дополнительным фильтром.

Пример 10.6. Запрос, содержащий операцию соединения и дополнительный фильтр

```
USE sample;
SELECT emp_lname, emp_fname
   FROM employee, works_on
   WHERE employee.emp_no = works_on.emp_no
   AND enter_date = '10.15.2007';
```

Для запроса в примере 10.6 рекомендуется создать два отдельных индекса для столбца emp_no как в таблице employee, так и в таблице works_on. Кроме этого, следует создать дополнительный индекс для столбца enter_date.

Покрывающий индекс

Как уже упоминалось ранее, включение *всех* столбцов запроса в индекс может значительно повысить производительность запроса. Создание такого индекса, называемого *покрывающим* (covering), показано в примере 10.7.

Пример 10.7. Создание покрывающего индекса

```
USE AdventureWorks2012;
GO
DROP INDEX Person.Address.IX_Address_StateProvinceID;
GO
CREATE INDEX i_address_zip
   ON Person.Address (PostalCode)
   INCLUDE (City, StateProvinceID);
GO
SELECT City, StateProvinceID
   FROM Person.Address
   WHERE PostalCode = 84407;
```

В примере 10.7 в первую очередь из таблицы `Address` удаляется индекс `IX_Address_StateProvinceID`. Затем создается новый индекс, который помимо столбца `PostalCode` включает два дополнительных столбца. Наконец, инструкция `SELECT` в конце примера показывает запрос, покрываемый индексом. Для этого запроса системе нет необходимости выполнять поиск данных в страницах данных, поскольку оптимизатор запросов может найти все значения столбцов в страницах листьев некластеризованного индекса.

ПРИМЕЧАНИЕ

Покрывающие индексы рекомендуется применять по той причине, что страницы индексов обычно содержат намного больше записей, чем соответствующие страницы данных. Кроме этого, для того чтобы использовать этот метод, фильтруемые столбцы должны быть первыми ключевыми столбцами в индексе.

Специальные типы индексов

Компонент Database Engine позволяет создавать следующие специальные типы индексов:

- индексированные представления;
- фильтруемые индексы;
- индексы для вычисляемых столбцов;
- секционированные индексы;
- индексы сохранения столбца;
- XML-индексы;
- полнотекстовые индексы.

Индексированные представления основаны на представлениях и поэтому рассматриваются в следующей главе. Фильтруемые индексы похожи на индексированные представления. Информацию об этих индексах можно найти в электронной документации. Секционированные индексы используются с секционированными таблицами и поэтому рассматриваются в *главе 25*. Индексы сохранения столбца (column store indices) являются одной из наиболее важных новых возможностей SQL Server 2012 и рассматриваются в *главе 25*. Индексы XML обсуждаются в *главе 26*, а полнотекстовые индексы — в *главе 28*.

В этом разделе рассматриваются вычисляемые столбцы и связанные с ними индексы.

Вычисляемым столбцом называется столбец таблицы, в котором сохраняются результаты вычислений данных таблицы. Такой столбец может быть виртуальным или постоянным. Эти два типа столбцов рассмотрены в следующих далее подразделах.

Виртуальные вычисляемые столбцы

Вычисляемый столбец, который не имеет соответствующего кластеризованного индекса, является логическим, т. е. он физически на жестком диске не хранится. Таким образом, он вычисляется при каждом обращении к строке. Использование виртуальных вычисляемых столбцов показано в примере 10.8.

Пример 10.8. Использование виртуальных вычисляемых столбцов

```
USE sample;
CREATE TABLE orders
   (orderid INT NOT NULL,
   price MONEY NOT NULL,
   quantity INT NOT NULL,
   orderdate DATETIME NOT NULL,
   total AS price * quantity,
   shippeddate AS DATEADD (DAY, 7, orderdate));
```

Таблица `orders` в примере 10.8 имеет два виртуальных вычисляемых столбца: `total` и `shippeddate`. Столбец `total` вычисляется с использованием двух других столбцов, `price` и `quantity`, а столбец `shippeddate` вычисляется при использовании функции `DATEADD` и столбца `orderdate`.

Постоянные вычисляемые столбцы

Компонент Database Engine позволяет создавать индексы для детерминированных вычисляемых столбцов, где базовые столбцы имеют точные типы данных. (Вычисляемый столбец называется *детерминированным*, если всегда возвращаются одни и те же значения для одних и тех же данных таблицы.)

Индексированный вычисляемый столбец может быть создан только в том случае, если следующим параметрам инструкции `SET` присвоено значение `ON` (эти параметры обеспечивают детерминированность столбца):

- `QUOTED_IDENTIFIER`;
- `CONCAT_NULL_YIELDS_NULL`;
- `ANSI_NULLS`;
- `ANSI_PADDING`;
- `ANSI_WARNINGS`.

Кроме этого, параметру `NUMERIC_ROUNDABORT` нужно присвоить значение `OFF`.

Если для вычисляемого столбца создать кластеризованный индекс, то значения столбца будут существовать физически в соответствующих строках таблицы, поскольку страницы листьев кластеризованного индекса содержат строки данных (см. *разд. "Кластеризованные индексы" ранее в этой главе*).

В примере 10.9 показано создание кластеризованного индекса для вычисляемого столбца `total` из примера 10.8.

Пример 10.9. Создание кластеризованного индекса для вычисляемого столбца

```
CREATE CLUSTERED INDEX i1 ON orders (total);
```

После выполнения инструкции `CREATE INDEX` вычисляемый столбец `total` будет присутствовать в таблице физически. Это означает, что все обновления базовых столбцов вычисляемого столбца будут вызывать его обновление.

ПРИМЕЧАНИЕ

Столбец можно сделать постоянным и другим способом, используя параметр `PERSISTED`. Этот параметр позволяет задать физическое наличие вычисляемого столбца, даже не создавая соответствующего кластеризованного индекса. Эта возможность требуется для создания физических вычисляемых столбцов, которые создаются на столбцах с приблизительным типом данных (`FLOAT` или `REAL`). (Как упоминалось ранее, индекс для вычисляемого столбца можно создать только в том случае, если его базовые столбцы имеют точный тип данных.)

Резюме

Индексы применяются для того, чтобы обеспечить более эффективный доступ к данным. Они могут оказывать влияние не только на производительность инструкции `SELECT`, но также на производительность инструкций `INSERT`, `UPDATE` и `DELETE`. Существуют кластеризованные и некластеризованные, однозначные и неоднозначные, а также простые и составные индексы. Кластеризованный индекс физически сортирует строки таблицы в порядке указанного столбца (или столбцов). Однозначный индекс указывает, что каждое значение может встречаться в таблице только один раз. Составной индекс создается по нескольким столбцам.

Прекрасным средством, относящимся к индексам, является Database Engine Tuning Advisor (DTA), который, среди прочего, анализирует образчик действительной загруженности (предоставляемой либо пользователем с помощью файла сценария, либо приложением SQL Server Profiler посредством записанного файла трассировки) и рекомендует, какие индексы следует добавить или удалить на основе данной загруженности. Настоятельно рекомендуется использовать DTA. Дополнительную информацию о SQL Server Profiler и DTA см. в *главе 20*.

В следующей главе мы рассмотрим концепцию представления.

Упражнения

Упражнение 10.1

Создайте некластеризованный индекс для столбца `enter_date` таблицы `works_on`, с заполнением пространства каждой страницы листьев индекса на 70%.

Упражнение 10.2

Создайте однозначный составной индекс для столбцов `emp_lname` и `emp_fname` таблицы `employee`. Будет ли какая-либо разница, если изменить порядок столбцов в составном индексе?

Упражнение 10.3

Как удалить индекс, который был неявно создан для первичного ключа таблицы?

Упражнение 10.4

Изложите достоинства и недостатки индексов.

В следующих четырех упражнениях создайте индексы, которые повысят производительность запросов. Предполагается, что все таблицы базы данных `sample`, используемые в этих упражнениях, имеют большое количество строк.

Упражнение 10.5

```
SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE emp_lname = 'Smith'
```

Упражнение 10.6

```
SELECT emp_no, emp_fname, emp_lname
   FROM employee
   WHERE emp_lname = 'Hansel'
   AND emp_fname = 'Elke'
```

Упражнение 10.7

```
SELECT job
   FROM works_on, employee
   WHERE employee.emp_no = works_on.emp_no
```

Упражнение 10.8

```
SELECT emp_lname, emp_fname
   FROM employee, department
   WHERE employee.dept_no = department.dept_no
   AND dept_name = 'Research'
```

Глава 11

Представления

- **Инструкции языка DDL и представления**
- **Инструкции языка DML и представления**
- **Индексированные представления**

Данная глава полностью посвящена рассмотрению объектов базы данных, которые называются *представлениями*. Структура этой главы совпадает со структурой *глав 5—7*, в которых описаны инструкции языка DML для базовых таблиц. В первом разделе этой главы рассматриваются инструкции языка DDL для работы с представлениями: `CREATE VIEW`, `ALTER VIEW` и `DROP VIEW`. Во второй части главы обсуждается использование инструкций языка DML `SELECT`, `INSERT`, `UPDATE` и `DELETE` для работы с представлениями. Из них инструкция `SELECT` рассматривается отдельно от остальных трех. В отличие от базовых таблиц, на выполнение модификаций представлений накладываются определенные ограничения. Эти ограничения описываются в конце каждого соответствующего раздела.

В последнем основном разделе этой главы обсуждается другая форма представления, называемая *индексированным представлением*. Этот тип индекса материализует соответствующий запрос и позволяет значительно повысить производительность запросов с агрегированным данным.

Инструкции DDL и представления

В предыдущих главах инструкции DDL и DML рассматривались применительно к базовым таблицам. Данные базовой таблицы хранятся на диске. В отличие от базовых таблиц, представления по умолчанию не существуют физически, т. е. их содержимое не сохраняется на диске. Это не относится к так называемым индексированным представлениям, которые рассматриваются далее в этой главе. *Представления* — это объекты базы данных, которые всегда создаются на основе одной или

более базовых таблиц (или других представлений), используя информацию метаданных. Эта информация (включая имя представления и способ получения строк из базовых таблиц) — все, что сохраняется физически для представления. По этой причине представления также называются *виртуальными таблицами*.

Создание представления

Представление создается посредством инструкции `CREATE VIEW`, синтаксис которой выглядит следующим образом:

```
CREATE VIEW view_name [(column_list)]
   [WITH {ENCRYPTION SCHEMABINDING | VIEW_METADATA}]
   AS select_statement
   [WITH CHECK OPTION]
```

ПРИМЕЧАНИЕ

Инструкция `CREATE VIEW` должна быть единственной инструкцией пакета. (Это означает, что эту инструкцию следует отделять от других инструкций группы посредством инструкции `GO`.)

Параметр *`view_name`* задает имя определяемого представления, а в параметре *`column_list`* указывается список имен, которые будут использоваться в качестве имен столбцов представления. Если этот необязательный параметр опущен, то используются имена столбцов таблиц, по которым создается представление. Параметр *`select_statement`* задает инструкция `SELECT`, которая извлекает строки и столбцы из таблиц (или других представлений). Параметр `WITH ENCRYPTION` задает шифрование инструкции `SELECT`, повышая таким образом уровень безопасности системы баз данных.

Предложение `SCHEMABINDING` привязывает представление к схеме таблицы, по которой оно создается. Когда это предложение указывается, имена объектов баз данных в инструкции `SELECT` должны состоять из двух частей, т. е. в виде *`owner.db_object`*, где *`owner`* — владелец, а *`db_object`* может быть таблицей, представлением или определяемой пользователем функцией.

Любая попытка модифицировать структуру представлений или таблиц, на которые ссылается созданное таким образом представление, будет неудачной. Чтобы такие таблицы или представления можно было модифицировать (инструкцией `ALTER`) или удалять (инструкцией `DROP`), нужно удалить это представление или убрать из него предложение `SCHEMABINDING`. Предложение `WITH CHECK OPTION` подробнее рассматривается в *разд. "Инструкция INSERT и представления" далее в этой главе*.

Когда при создании представления указывается параметр `VIEW_METADATA`, все его столбцы можно обновлять (за исключением столбцов с типом данных `TIMESTAMP`), если представление имеет триггеры `INSERT` или `UPDATE INSTEAD OF`. Триггеры подробнее рассматриваются в *главе 14*.

ПРИМЕЧАНИЕ

Инструкция SELECT в представлении не может содержать предложение ORDER BY или параметр INTO. Кроме этого, по временным таблицам нельзя выполнять запросы.

Представления можно использовать для разных целей.

- Для ограничения использования определенных столбцов и/или строк таблиц. Таким образом, представления можно использовать для управления доступом к определенной части одной или нескольких таблиц.
- Для скрытия подробностей сложных запросов. Если для приложения базы данных требуются запросы со сложными операциями соединения, создание соответствующих представлений может упростить такие запросы.
- Для ограничения вставляемых или обновляемых значений некоторым диапазоном.

В примере 11.1 показано создание представления.

Пример 11.1. Создание представления

```
USE sample;
GO
CREATE VIEW v_clerk
   AS SELECT emp_no, project_no, enter_date
      FROM works_on
      WHERE job = 'Clerk';
```

Запрос в примере 11.1 выбирает из таблицы works_on строки, удовлетворяющие условию job='Clerk'. Представление v_clerk определяется строками и столбцами, возвращаемыми этим запросом. В табл. 11.1 отображена таблица works_on, в которой строки, выбранные в представлении v_clerk, выделены жирным шрифтом.

Таблица 11.1. *Базовая таблица* works_on

emp_no	project_no	job	enter_date
10102	p1	Analyst	2006.10.1 00:00:00
10102	p3	Manager	2008.1.1 00:00:00
25348	**p2**	**Clerk**	**2007.2.15 00:00:00**
18316	p2	NULL	2007.6.1 00:00:00
29346	p2	NULL	2006.12.15 00:00:00
2581	p3	Analyst	2007.10.15 00:00:00
9031	p1	Manager	2007.4.15 00:00:00
28559	p1	NULL	2007.8.1 00:00:00
28559	**p2**	**Clerk**	**2008.2.1 00:00:00**

Таблица 11.1 *(окончание)*

emp_no	project_no	job	enter_date
9031	**p3**	**Clerk**	**2006.11.15 00:00:00**
29346	**p1**	**Clerk**	**2007.1.4 00:00:00**

Запрос в примере 11.1 задает выборку строк, т. е. он создает горизонтальное подмножество базовой таблицы `works_on`. Возможно также создание представления с ограничениями на включаемые в него столбцы и строки. Создание такого представления показано в примере 11.2.

Пример 11.2. Ограничение столбцов, выбираемых в представление

```
USE sample;
GO
CREATE VIEW v_without_budget
   AS SELECT project_no, project_name
      FROM project;
```

Запрос в примере 11.2 выбирает для включения в представление `v_without_budget` все столбцы таблицы `project`, за исключением столбца `budget`.

Как уже упоминалось ранее, в общем формате инструкции `CREATE VIEW` не обязательно указывать имена столбцов представления. Однако, с другой стороны, в приведенных далее двух случаях обязательно требуется явно указывать имена столбцов:

- если столбец представления создается из выражения или агрегатной функции;
- если два или больше столбцов представления имеют одинаковое имя в базовой таблице.

В примере 11.3 показано создание представления, для которого явно указываются имена столбцов.

Пример 11.3. Явное задание имен столбцов для представления

```
USE sample;
GO
CREATE VIEW v_count(project_no, count_project)
   AS SELECT project_no, COUNT(*)
      FROM works_on
      GROUP BY project_no;
```

В примере 11.3 имена столбцов представления `v_count` должны быть указаны явно по той причине, что инструкция `SELECT` содержит агрегатную функцию `COUNT(*)`, которая требует, чтобы все столбцы представления были именованы.

Не требуется явно указывать список столбцов в инструкции CREATE VIEW, если применить заглавия столбцов, как это показано в примере 11.4.

Пример 11.4. Создание представления с заглавиями столбцов

```
USE sample;
GO
CREATE VIEW v_count1
   AS SELECT project_no, COUNT(*) count_project
      FROM works_on
      GROUP BY project_no;
```

Представление можно создать из другого представления, как показано в примере 11.5.

Пример 11.5. Создание представления по другому представлению

```
USE sample;
GO
CREATE VIEW v_project_p2
   AS SELECT emp_no
      FROM v_clerk
      WHERE project_no ='p2';
```

Представление v_project_p2 в примере 11.5 создается из представления v_clerk (см. пример 11.1). Все запросы, использующие представление v_project_p2, преобразовываются в эквивалентные запросы к базовой таблице works_on.

Представления можно также создавать посредством среды Management Studio. Для этого выберите в обозревателе объектов базу данных, в которой требуется создать представление, щелкните в ней правой кнопкой мыши узел **Views** (Представления) и в открывшемся контекстном меню выберите пункт **New View** (Новое представление). Откроется редактор представлений, в котором можно выполнять следующие действия:

- выбрать базовые таблицы и строки в этих таблицах для создания представления;
- присвоить представлению имя и определить условия в предложении WHERE соответствующего запроса.

Изменение и удаление представлений

Для изменения определения представления в языке Transact-SQL применяется инструкция ALTER VIEW. Синтаксис этой инструкции аналогичен синтаксису инструкции CREATE VIEW, применяющейся для создания представления.

Использование инструкции ALTER VIEW позволяет избежать переназначения существующих разрешений для представления. Кроме этого, изменение представления

посредством этой инструкции не влияет на объекты базы данных, зависящие от этого представления. Если же модифицировать представление, сначала удалив его (инструкция DROP VIEW), а затем создав новое представление с требуемыми свойствами (инструкция CREATE VIEW), то все объекты базы данных, которые ссылаются на это представление, не будут работать должным образом, по крайней мере, в период времени между удалением представления и его воссоздания.

Использование инструкции ALTER VIEW показано в примере 11.6.

Пример 11.6. Изменение представления посредством инструкции ALTER VIEW

```
USE sample;
GO
ALTER VIEW v_without_budget
   AS SELECT project_no, project_name
      FROM project
      WHERE project_no >= 'p3';
```

В примере 11.6 инструкция ALTER VIEW расширяет инструкцию SELECT в представлении v_without_budget (см. пример 11.2) новым условием в предложении WHERE.

ПРИМЕЧАНИЕ

Инструкцию ALTER VIEW можно также применять и для изменения индексированных представлений. Эта инструкция удаляет все индексы для таких представлений.

Инструкция DROP VIEW удаляет из системных таблиц определение указанного в ней представления. Применение этой инструкции показано в примере 11.7.

Пример 11.7. Удаление представления посредством инструкции DROP VIEW

```
USE sample;
GO
DROP VIEW v_count;
```

При удалении представления инструкцией DROP VIEW все другие представления, основанные на удаленном, также удаляются, как показано в примере 11.8.

Пример 11.8. Неявное удаление представления

```
USE sample;
GO
DROP VIEW v_clerk;
```

В примере 11.8 инструкция DROP VIEW явно удаляет представление v_clerk, при этом неявно удаляя представление v_project_p2, основанное на представлении v_clerk

(см. пример 11.5). Теперь попытка выполнить запрос по представлению v_project_p2 возвратит сообщение об ошибке: "Invalid Object name (Недействительное имя: 'v_clerk)".

ПРИМЕЧАНИЕ

При удалении базовой таблицы представления, основанные на ней, не удаляются автоматически. Это означает, что все представления для удаленной таблицы нужно удалять явно, используя инструкцию DROP VIEW. С другой стороны, представления удаленной таблицы можно снова использовать на новой таблице, имеющей такую же логическую структуру, как и удаленная.

Редактирование информации о представлениях

Наиболее важным представлением каталога применительно к представлениям является sys.objects. Как уже упоминалось, это представление каталога содержит информацию касательно всех объектов в текущей базе данных. Все строки этого представления со значением V в столбце type содержат информацию о представлениях.

А представление каталога sys.views содержит дополнительную информацию о существующих представлениях. Наиболее важным столбцом этого представления является столбец with_check_option, который информирует, указано или нет предложение WITH CHECK OPTION.

Запрос для определенного представления можно отобразить посредством системной процедуры sp_helptext.

Инструкции DML и представления

Выборка информации из представлений и ее модификация осуществляются посредством тех же самых инструкций языка Transact-SQL, что и для выборки и модификации информации из базовых таблиц. Эти инструкции, имеющие отношения к представлениям, рассматриваются в последующих подразделах.

Выборка информации из представления

Для всех практических целей представление это то же самое, что и любая базовая таблица базы данных. Поэтому выборку информации из представления можно рассматривать, как преобразование инструкций запроса по представлению в эквивалентные операции по базовым таблицам, на основе которых создано это представление. Такой запрос на выборку данных из представления показан в примере 11.9.

Пример 11.9. Запрос на выборку данных из представления

```
USE sample;
GO
CREATE VIEW v_d2
```

```
      AS SELECT emp_no, emp_lname
         FROM employee
         WHERE dept_no ='d2';
   GO
   SELECT emp_lname
      FROM v_d2
      WHERE emp_lname LIKE 'J%';
```

Результат выполнения этого запроса:

```
emp_lname
-------------------------------------------------------------
James
```

Инструкция `SELECT` в примере 11.9 трансформируется в следующую эквивалентную форму с использованием таблицы из представления `v_d2`:

```
SELECT emp_lname
   FROM employee
   WHERE emp_lname LIKE 'J%'
   AND dept_no ='d2';
```

В следующих трех разделах мы рассмотрим использование представлений с тремя другими инструкциями языка DML: `INSERT`, `UPDATE` и `DELETE`. Модифицирование данных посредством этих инструкций подобно выборке данных. Единственное отличие состоит в том, что для представления, используемого для вставки, модифицирования и удаления данных из таблицы, на основе которой оно создано, существуют некоторые ограничения.

Инструкция *INSERT* и представление

Инструкцию `INSERT` можно применять с представлением, как если бы оно было обычной базовой таблицей. Вставляемые в представление строки в действительности вставляются в таблицу в основе представления.

В примере 11.10 создается представление `v_dept`, которое содержит первые два столбца таблицы `department`. Последующая инструкция `INSERT` вставляет две строки в таблицу, связанную с представлением, используя значения `d4` и `Development`. Столбцу `location`, который не вошел в представление `v_dept`, присваивается значение `NULL`.

Пример 11.10. Вставка строк в представление

```
USE sample;
GO
CREATE VIEW v_dept
   AS SELECT dept_no, dept_name
      FROM department;
GO

INSERT INTO v_dept
   VALUES('d4', 'Development');
```

При использовании представления обычно возможно вставить строку, которая не удовлетворяет условиям в предложении WHERE запроса представления. Чтобы ограничить вставку только строками, которые удовлетворяют условиям запроса, применяется предложение WITH CHECK OPTION. При использовании этого предложения компонент Database Engine проверяет каждую вставляемую строку на удовлетворение условий предложения WHERE. Если это предложение отсутствует, такая проверка не выполняется, вследствие чего каждая вставляемая в представление строка также вставляется в таблицу в его основе. Это может вызвать путаницу, когда строка вставляется в представление, но впоследствии не возвращается из этого представления инструкцией SELECT, т. к. для нее принудительно выполняются условия предложения WHERE. Предложение WITH CHECK OPTION также применяется и с инструкцией UPDATE.

В примерах 11.11—11.12 показана разница между применением и неприменением предложения WITH CHECK OPTION соответственно.

Пример 11.11. Запрос на вставку строк в представление с применением предложения WITH CHECK OPTION

```
USE sample;
GO
CREATE VIEW v_2006_check
   AS SELECT emp_no, project_no, enter_date
      FROM works_on
      WHERE enter_date BETWEEN '01.01.2006' AND '12.31.2006'
      WITH CHECK OPTION;
GO

INSERT INTO v_2006_check
   VALUES (22334, 'p2', '1.15.2007');
```

В примере 11.11 система проверяет, соответствует ли вставляемое в столбец enter_date значение True (истина) при вычислении условия в предложении WHERE инструкции SELECT. Если вставляемое значение не удовлетворяет этим условиям, строка не вставляется.

Пример 11.12. Запрос на вставку строк в представление без применения предложения WITH CHECK OPTION

```
USE sample;
GO
CREATE VIEW v_2006_nocheck
   AS SELECT emp_no, project_no, enter_date
      FROM works_on
      WHERE enter_date BETWEEN '01.01.2006' AND '12.31.2006';
GO
```

```
INSERT INTO v_2006_nocheck
   VALUES (22334, 'p2', '1.15.2007');
SELECT *
   FROM v_2006_nocheck;
```

В результате выполнения этого запроса будут вставлены следующие строки:

emp_no	project_no	enter_date
10102	p1	2006-10-01
29346	p2	2006-12-15
9031	p3	2006-11-15

Поскольку в примере 11.12 предложение WITH CHECK OPTION не применяется, инструкция будет выполнена, и строка вставляется в основную таблицу works_on. Обратите внимание на тот факт, что вставленная строка не будет возвращена инструкцией SELECT, поскольку ее нельзя извлечь посредством представления v_2006_nocheck.

Вставку строк в таблицу, на которой основано представление, нельзя выполнить, если это представление содержит одну из следующих возможностей:

- предложение FROM в определении представления содержит более чем одну таблицу, и список столбцов содержит столбцы более чем из одной таблицы;
- столбец в представлении создается из агрегатной функции;
- инструкция SELECT в представлении содержит предложение GROUP BY или параметр DISTINCT;
- столбец в представлении создается из константы или выражения.

В примере 11.13 показано представление, которое нельзя использовать для вставки строк в таблицу, на которой основано это представление.

Пример 11.13. Представление, неприменимое для вставки строк в таблицу в своей основе

```
USE sample;
GO
CREATE VIEW v_sum(sum_of_budget)
   AS SELECT SUM(budget)
   FROM project;
GO

SELECT *
   FROM v_sum;
```

Запрос в примере 11.13 создает представление v_sum, которое содержит агрегатную функцию SUM() в инструкции SELECT. Поскольку представление в этом примере воз-

вращает результат объединения нескольких строк (а не одну строку таблицы project), то нет смысла пытаться вставить одну строку в базовую таблицу, используя это представление.

Инструкция *UPDATE* и представление

Инструкцию UPDATE можно применять с представлением, как будто бы это была базовая таблица. При модифицировании строк представления также модифицируется содержимое таблицы в его основе.

Запрос в примере 11.14 создает представление, посредством которого затем модифицируется таблица works_on.

Пример 11.14. Представление, модифицирующее базовую таблицу в своей основе

```
USE sample;
GO
CREATE VIEW v_p1
   AS SELECT emp_no, job
      FROM works_on
      WHERE project_no = 'p1';
GO
UPDATE v_p1
   SET job = NULL
   WHERE job = 'Manager';
```

Операцию обновления представления v_p1 в примере 11.14 можно рассматривать эквивалентной выполнению следующей инструкции UPDATE:

```
UPDATE works_on
   SET job = NULL
   WHERE job = 'Manager'
   AND project_no = 'p1'
```

Логическое значение предложения WITH CHECK OPTION для инструкции UPDATE имеет такое же значение, как и для инструкции INSERT. Использование предложения WITH CHECK OPTION в инструкции UPDATE показано в примере 11.15.

Пример 11.15. Использование предложения WITH CHECK OPTION в инструкции UPDATE

```
USE sample;
GO
CREATE VIEW v_100000
   AS SELECT project_no, budget
      FROM project
      WHERE budget > 100000
      WITH CHECK OPTION;
GO
```

```
UPDATE v_100000
   SET budget = 93000
   WHERE project_no = 'p3';
```

В примере 11.15 компонент Database Engine проверяет, будет ли измененное значение столбца budget давать значение True в условии предложения WHERE инструкции SELECT. Попытка изменения значения завершается неудачей, поскольку условие не удовлетворяется, т. е. вставляемое значение 93000 не больше, чем значение 100000.

Модификацию столбцов таблицы, на которой основано представление, нельзя выполнить, если это представление содержит одну из следующих возможностей:

- предложение FROM в определении представления включает более чем одну таблицу, и список столбцов содержит столбцы из более чем одной таблицы;
- столбец представления создается из агрегатной функции;
- инструкция SELECT в представлении содержит предложение GROUP BY или параметр DISTINCT;
- столбец в представлении создается из константы или выражения.

В примере 11.16 показано представление, которое нельзя использовать для изменения значений в таблице, на которой основано представление.

Пример 11.16. Представление, неприменимое для изменения строк в таблице, на которой основано представление

```
USE sample;
GO
CREATE VIEW v_uk_pound (project_number, budget_in_pounds)
   AS SELECT project_no, budget*0.65
      FROM project
      WHERE budget > 100000;
GO

SELECT *
   FROM v_uk_pound;
```

Результат выполнения этого запроса:

project_number	budget_in_pounds
p1	78000
p3	121225

В примере 11.16 представление v_uk_pound нельзя использовать с инструкцией UPDATE (или с инструкцией INSERT), поскольку значения столбца budget_in_pound являются результатом вычисления арифметического выражения, а не первоначальными значениями столбца таблицы, на которой основано это представление.

Инструкция *DELETE* и представление

С помощью представления можно удалить строки из таблицы, на которой оно основано, как это показано в примере 11.17.

Пример 11.17. Удаление строк из таблицы посредством ее представления

```
USE sample;
GO
CREATE VIEW v_project_p1
   AS SELECT emp_no, job
      FROM works_on
      WHERE project_no = 'p1';
GO

DELETE FROM v_project_p1
   WHERE job = 'Clerk';
```

Запрос в примере 11.17 создает представление, посредством которого затем удаляются строки из таблицы `works_on`. Удаление строк из таблицы, на которой основано представление, невозможно, если:

- предложение `FROM` в определении представления содержит более чем одну таблицу, и список столбцов содержит столбцы более чем из одной таблицы;
- столбец в представлении создается из агрегатной функции;
- инструкция `SELECT` представления содержит предложение `GROUP BY` или параметр `DISTINCT`.

В отличие от инструкций `INSERT` и `UPDATE`, инструкция `DELETE` допускает значения, получаемые из констант или выражений, в столбце представления, используемого для удаления строк из таблицы, на которой оно основано.

В примере 11.18 показано представление, посредством которого можно удалять строки, но не вставлять строки или изменять значения столбцов.

Пример 11.18. Представление, позволяющее удалять, но не вставлять строки или изменять значения столбцов

```
USE sample;
GO
CREATE VIEW v_budget (budget_reduction)
   AS SELECT budget*0.9
      FROM project;
GO

DELETE FROM v_budget;
```

Инструкция `DELETE` в примере 11.18 удаляет все строки таблицы `project`, на которой основано представление `v_budget`.

Индексированные представления

Как вы уже знаете из *главы 10*, существует несколько специальных типов индексов. Одним из таких специальных типов индексов являются индексированные представления, которые и рассматриваются в этом разделе.

Определение представления всегда содержит запрос, играющий роль фильтра. Если представление не имеет индексов, то компонент Database Engine динамически создает результирующий набор из всех запросов, которые обращаются к представлению. (Выражение "динамически" здесь означает, что модифицированное содержимое таблицы будет всегда отображаться в соответствующем представлении.) Кроме этого, если представление содержит вычисления по одному или больше столбцов таблицы, то эти вычисления выполняются при каждом обращении к представлению.

Если инструкция `SELECT` представления обрабатывает большое количество строк из одной или более таблиц, динамическое создание результирующего набора запроса может понизить уровень производительности запроса. Если подобное представление часто используется в запросах, уровень производительности можно значительно повысить, создав кластеризованный индекс для этого представления (см. *следующий раздел*). Создание кластеризованного индекса означает, что система материализует динамические данные в страницах листьев структуры индекса.

Компонент Database Engine позволяет создавать индексы для представлений. Такие представления называются *индексированными* или *материализованными представлениями*. Результирующий набор, возвращаемый представлением с кластеризованным индексом, сохраняется в базе данных таким же образом, как и таблица с кластеризованным индексом. Это означает, что узлы листьев B^+-дерева кластеризованного индекса содержат страницы данных (см. также описание кластеризованной таблицы в *главе 10*).

ПРИМЕЧАНИЕ

Индексированные представления создаются посредством синтаксических расширений инструкций `CREATE INDEX` и `CREATE VIEW`. В инструкции `CREATE INDEX` вместо имени таблицы указывается имя представления. Синтаксис инструкции `CREATE VIEW` расширяется предложением `SCHEMABINDING`. Дополнительную информацию по расширениям этой инструкции см. в описании, приведенном в начале этой главы.

Создание индексированного представления

Индексированное представление создается в два этапа.

1. Создается представление посредством инструкции `CREATE VIEW` с предложением `SCHEMABINDING`.
2. Создается кластеризованный индекс для этого представления.

В примере 11.19 показан первый шаг создания индексированного представления — создание представления. В этом примере предполагается, что таблица works_on имеет очень большой размер.

Пример 11.19. Создание представления для последующего индексирования

```
USE sample;
GO
CREATE VIEW v_enter_month
   WITH SCHEMABINDING
   AS SELECT emp_no, DATEPART(MONTH, enter_date) AS enter_month
      FROM dbo.works_on;
```

Таблица works_on базы данных sample содержит столбец enter_date, который представляет дату начала работы сотрудника над соответствующим проектом. Всех сотрудников, которые начали работать над проектами в указанный месяц, можно выбрать с помощью представления, представленного в примере 11.19. Для выборки этого результирующего набора Database Engine не может использовать индекс таблицы, поскольку индекс для столбца enter_date будет определять значения этого столбца по полной дате, а не только по месяцу. В таком случае можно воспользоваться индексированным представлением, создание которого показано в примере 11.20.

Пример 11.20. Создание индекса для представления

```
USE sample;
GO
CREATE UNIQUE CLUSTERED INDEX
   c_workson_deptno ON v_enter_month (enter_month, emp_no);
```

Чтобы создать представление индексированным, необходимо создать однозначный (уникальный) кластеризованный индекс для столбца (столбцов) этого представления. (Как уже упоминалось ранее, кластеризованный индекс является единственным типом индекса, который содержит значения данных в своих страницах листьев.) После создания такого индекса система баз данных выделяет память для этого представления, после чего можно создавать любое число некластеризованных индексов, поскольку теперь это представление рассматривается как (базовая) таблица.

Индексированное представление можно создать только в том случае, если оно является детерминированным, т. е. представление всегда возвращает один и тот же результирующий набор. Для этого следующим параметрам инструкции SET нужно присвоить значение ON:

- QUOTED_IDENTIFIER;
- CONCAT_NULL_YIELDS_NULL;
- ANSI_NULLS;

- ANSI_PADDING;
- ANSI_WARNINGS.

Кроме этого, параметру NUMERIC_ROUNDABORT нужно присвоит значение OFF.

Проверить, установлены ли должным образом параметры в предыдущем списке, можно несколькими способами, которые рассматриваются в *разд. "Редактирование информации, связанной с индексированными представлениями" далее в этой главе*.

Чтобы создать индексированное представление, представление должно отвечать следующим требованиям:

- все используемые в представлении функции (как системные, так и определяемые пользователем) должны быть детерминированными, т. е. для одних и тех же аргументов они всегда должны возвращать один и тот же результат;
- представление должно ссылаться только на базовые таблицы;
- представление и ссылки на базовую таблицу (таблицы) должны иметь одного владельца и принадлежать к одной и той же базе данных;
- представление должно быть создано с опцией SCHEMABINDING. Эта опция связывает представление со схемой, содержащей базовые таблицы, лежащие в основе представления;
- определенные пользователем функции, на которые ссылается представление, должны быть созданы с предложением SCHEMABINDING;
- инструкция SELECT в представлении не должна содержать следующие предложения, параметры и прочие элементы: DISTINCT, UNION, TOP, ORDER BY, MIN, MAX, COUNT, OUTER, SUM (для выражений, допускающих значения NULL), подзапросы или производные таблицы.

Удовлетворение всех этих требований можно проверить посредством функции свойств objectproperty с параметром свойств IsIndexable, как показано в примере 11.21. Если функция возвращает значение 1, то представление удовлетворяет всем требованиям для создания для него индекса.

Пример 11.21. Проверка удовлетворения требований для создания индекса представления

```
USE sample;
SELECT objectproperty(object_id('v_enter_month'), 'IsIndexable');
```

Модифицирование структуры индексированного представления

Чтобы удалить однозначный кластеризованный индекс в индексированном представлении, необходимо также удалить все его некластеризованные индексы. После удаления кластеризованного индекса представления система рассматривает его как обычное представление.

ПРИМЕЧАНИЕ

При удалении индексированного представления также удаляются все его индексы.

Если вы хотите изменить обычное представление на индексированное, то для него вам нужно создать кластеризованный индекс. Чтобы сделать это, вы сначала должны указать предложение SCHEMABINDING. Представление можно удалить, а потом воссоздать, указав предложение SCHEMABINDING в инструкции CREATE SCHEMA, или же можно создать другое представление, которое имеет такой же текст, как и существующее представление, но имеет другое имя.

ПРИМЕЧАНИЕ

При создании представления с другим именем необходимо обеспечить, чтобы это представление отвечало всем требованиям для индексированных представлений, описанных в предшествующем разделе.

Редактирование информации, связанной с индексированными представлениями

Проверить, активирован ли какой-либо параметр инструкции SET (список параметров см. в *разд. "Создание индексированного представления" ранее в этой главе*), можно с помощью функции свойств sessionproperty. Если функция возвращает значение 1, то указанный параметр установлен (т. е. имеет значение ON). В примере 11.22 показано использование этой функции для проверки значения параметра QUOTED_IDENTIFIER.

Пример 11.22. Проверка значения параметра QUOTED_IDENTIFIER

```
SELECT sessionproperty ('QUOTED_IDENTIFIER');
```

Наиболее простым способом является использование динамически административного представления sys.dm_exec_session, поскольку оно позволяет получить значения всех параметров инструкции SET, используя только один запрос. (Опять же, если значение столбца равно 1, то соответствующий параметр активирован.) В примере 11.23 демонстрируется использование этой функции для получения значений первых четырех параметров инструкции SET, перечисленных в *разд. "Создание индексированного представления"*. (Глобальная переменная @@spid рассматривается в *главе 4*.)

Пример 11.23. Определение значения параметров инструкции SET посредством динамического административного представления sys.dm_exec_session

```
USE sample;
SELECT quoted_identifier, concat_rmll_yields_null, ansi_nulls,
                                                    ansi_padding
```

```
FROM sys.dm_exec_sessions
WHERE session_id = @@spid;
```

Узнать, материализовано ли представление, т. е. использует ли оно дисковое пространство или нет, можно с помощью системной процедуры sp_spaceused. Результат выполнения запроса в примере 11.24 показывает, что представление v_enter_month использует область памяти как для данных, так и для определенного индекса.

Пример 11.24. Запрос для проверки использования хранилища представлением

```
USE sample;
EXEC sp_spaceused 'v_enter_month';
```

Этот запрос возвращает следующий результирующий набор:

Name	rows	Reserved	data	index_size	unused
v_enter_month	11	16KB	8KB	8KB	0KB

Преимущества индексированных представлений

Кроме возможного повышения уровня производительности для сложных представлений, к которым часто обращаются запросы, применение индексированных представлений имеет два других преимущества:

- индекс представления может быть использован даже в том случае, если в представлении явно не указана ссылка на предложение FROM;
- все изменения данных отражаются в соответствующих индексированных представлениях.

Возможно, самой важной особенностью индексированных представлений является то, что у запроса в представлении нет явного указания на использование индекса в этом представлении. Иными словами, если запрос содержит ссылку на столбцы в базовой таблице (или таблицах), которые также существуют в индексированных представлениях, и оптимизатор запросов определит, что самым лучшим способом выполнения запроса будет использование индексированного представления, то он выбирает индексы представления таким образом, как и индексы таблиц, когда запрос не ссылается на них явно.

При создании индексированного представления его результирующий набор сохраняется на диске (одновременно с созданием индекса). Таким образом, все данные, которые изменяются в базовых таблицах, также изменяются в соответствующем результирующем наборе индексированного представления.

Кроме всех преимуществ, которые можно получить благодаря использованию индексированных представлений, имеется также и потенциальный недостаток: индексы индексированных представлений обычно более сложны в обслуживании, чем индексы базовых таблиц. Причиной этому является то, что структура однозначного

кластеризованного индекса индексированного представления более сложна, чем структура соответствующего индекса базовой таблицы.

Можно значительно повысить уровень производительности следующих далее запросов, если проиндексировать представления, к которым они обращаются:

- запросы, которые обрабатывают большое количество строк и содержат операции соединения или агрегатные функции;
- операции соединения и агрегатные функции, которые часто выполняются в одном или нескольких запросах.

Если запрос ссылается на обычное представление, и системе баз данных требуется обработать большое количество строк, используя операцию соединения, то оптимизатор обычно выбирает менее оптимальный метод соединения. Но если для этого представления определить кластеризованный индекс, то уровень производительности запроса можно значительно повысить, т. к. оптимизатор запросов может использовать наиболее подходящий метод. (То же самое относится и к агрегатным функциям.)

Даже если запрос, который обращается к обычному представлению, и не обрабатывает большое количество строк, все равно, в случае частого использования такого запроса, применение индексированного представления может быть очень полезным. То же самое относится и к группе запросов, которые соединяют одни и те же таблицы или используют один и тот же тип агрегатных функций.

ПРИМЕЧАНИЕ

Начиная с версии SQL Server 2008 R2, Microsoft предоставляет альтернативное решение, взамен индексированных представлений, которое называется *фильтруемыми индексами*. Фильтруемые индексы представляют собой особую форму некластеризованных индексов, в которой индекс сужается, используя условие в конкретном запросе. Использование фильтруемых индексов имеет несколько преимуществ над использованием индексированных представлений.

Резюме

Представления можно использовать для ряда разных целей:

- для ограничения использования определенных столбцов и/или строк таблицы, т. е. для управления доступом к определенным частям одной или более таблиц;
- для скрытия подробностей сложных запросов;
- для ограничения добавляемых или изменяемых значений в определенных диапазонах.

Выборка информации из представлений и ее модификация осуществляется посредством таких же инструкций языка Transact-SQL, что и для выборки и модификации информации из базовых таблиц. Запрос по представлению всегда преобразовывается в запрос по таблице, на которой основано данное представление. Операция обновления обрабатывается подобно операции выборки. Единственное отличие со-

стоит в том, что для представления, используемого для вставки, модифицирования и удаления данных из таблицы, на основе которой оно создано, существуют некоторые ограничения. Но даже несмотря на это, способ, которым компонент Database Engine выполняет модификацию строк и столбцов, является более эффективным, чем те способы, которыми другие системы реляционных баз данных осуществляют подобную модификацию.

Индексированные представления применяются для повышения уровня эффективности определенных запросов. Представление становится индексированным, когда для него создается однозначный кластеризованный индекс, его результирующий набор физически сохраняется точно таким же образом, как и базовая таблица.

В следующей главе мы рассмотрим вопросы обеспечения безопасности компонента Database Engine.

Упражнения

Упражнение 11.1

Создайте представление, содержащее данные для всех сотрудников, которые работают в отделе `d1`.

Упражнение 11.2

Создайте представление для таблицы `project`, чтобы сотрудники могли просматривать все данные этой таблицы, за исключением столбца `budget`.

Упражнение 11.3

Создайте представление, содержащее имя и фамилии всех сотрудников, которые начали работать над своими проектами во второй половине 2007 г.

Упражнение 11.4

Выполните упражнение 11.3, переименовав исходные столбцы `emp_fname` и `emp_lname` в `first` и `last` соответственно.

Упражнение 11.5

Используя представление в упражнении 11.1, отобразите полные сведения для всех сотрудников, чья фамилия начинается с буквы "M".

Упражнение 11.6

Создайте представление, содержащее полные сведения для всех проектов, над которыми работает сотрудник по фамилии `Smith`.

Упражнение 11.7

Модифицируйте условие представления в упражнении 11.1 (используя инструкцию `ALTER VIEW`), чтобы представление возвращало данные всех сотрудников, которые работают в отделе `d1` или `d2` или в обоих отделах.

Упражнение 11.8

Удалите представление, созданное в упражнении 11.3. Что произойдет вследствие этого с представлением, созданным в упражнении 11.4?

Упражнение 11.9

Используя представление из упражнения 11.2, вставьте данные для нового проекта, номер которого `p2`, а имя — `Moon`.

Упражнение 11.10

Создайте представление (с предложением `WITH CHECK OPTION`), содержащее имена и фамилии всех сотрудников, чей табельный номер меньше, чем `10000`. Используя это представление, вставьте данные для нового сотрудника по фамилии `Kohn`, табельный номер которого `22123` и который работает в отделе `d3`.

Упражнение 11.11

Выполните упражнение 11.10, не используя предложение `WITH CHECK OPTION`, и определите разницу относительно вставки новых данных.

Упражнение 11.12

Создайте представление (с предложением `WITH CHECK OPTION`), содержащее полные сведения из таблицы `works_on` для всех сотрудников, которые начали работать над своими проектами в 2007 и 2008 гг. Затем для сотрудника с табельным номером `29346` измените дату начала работы над проектом на 06/01/2006.

Упражнение 11.13

Выполните упражнение 11.12, не используя предложение `WITH CHECK OPTION`, и определите разницу касательно модифицирования данных.

Глава 12

Система безопасности Database Engine

- **Аутентификация**
- **Схемы**
- **Безопасность баз данных**
- **Роли**
- **Авторизация**
- **Отслеживание изменений**
- **Безопасность данных и представления**

Эта глава начинается с краткого обзора наиболее важных понятий безопасности баз данных, после чего рассматриваются конкретные возможности системы безопасности компонента Database Engine. Наиболее важными концепциями безопасности баз данных являются следующие:

- аутентификация;
- шифрование;
- авторизация;
- отслеживание изменений.

Аутентификация заключается в предоставлении ответа на следующий вопрос: "Имеет ли данный пользователь законное право на доступ к системе?" Таким образом, данная концепция безопасности определяет процесс проверки подлинности учетных данных пользователя, чтобы не допустить использование системы несанкционированными пользователями. Аутентификацию можно реализовать путем запроса, требуя, чтобы пользователь предоставил, например, следующее:

- нечто, что известно пользователю (обычно пароль);
- нечто, что принадлежит пользователю, например, магнитную карту или идентификационную карту;

- физические характеристики пользователя, например, подпись или отпечатки пальцев.

Наиболее применяемый способ подтверждения аутентификации реализуется посредством использования имени пользователя и пароля. Система проверяет достоверность этой информации, чтобы решить, имеет ли данный пользователь законное право на доступ к системе или нет. Этот процесс может быть усилен применением шифрования.

Шифрование данных представляет собой процесс кодирования информации таким образом, что она становится непонятной, пока не будет расшифрована пользователем. Для шифрования данных применяются несколько способов, которые рассматриваются в *разд. "Шифрование" далее в этой главе*.

Авторизация — это процесс, который применяется к пользователям, уже получившим доступ к системе, пройдя через процесс аутентификации, чтобы определить их права на использование определенных ресурсов.

Отслеживание изменений означает отслеживание и документирование действий несанкционированных пользователей. Иными словами, документируются все операции вставки, обновления и удаления, применяемые к объектам базы данных. Задокументированные таким образом операции в дальнейшем могут быть проверены авторизованными пользователями. (Данный процесс полезен для защиты системы от пользователей, которые имеют повышенные права.)

Прежде чем приступить к рассмотрению этих четырех концепций безопасности, сначала ознакомимся с кратким определением модели безопасности, которая применяется в SQL Server. Данная модель безопасности состоит из трех разных категорий, которые взаимодействуют друг с другом:

- *Принципалы* (principals). Субъекты, которые имеют разрешение на доступ к определенной сущности. Типичными принципалами являются учетные записи Windows и SQL Server. Кроме этих принципалов, также существуют группы Windows и роли SQL Server. Группа Windows — это коллекция учетных записей и групп Windows. Присвоение учетной записи пользователя членство в группе дает этому пользователю все разрешения, предоставленные данной группе. Подобным образом роль является коллекцией учетных записей.
- *Защищаемые объекты* (securables). Ресурсы, доступ к которым регулируется системой авторизации базы данных. Большинство защищаемых объектов создают иерархию, что означает, что некоторые из них могут быть заключенными внутри других. Большинство защищаемых объектов имеют определенное число разрешений, применимых к ним. (Защищаемые объекты подробно рассмотрены *далее в этой главе*.)
- *Разрешения* (permissions). Каждый защищаемый объект имеет связанные с ним разрешения, которые могут быть предоставлены принципалу. Разрешения обсуждаются в *разд. "Авторизация" далее в этой главе*. (Список всех разрешений и их соответствующих защищаемых объектов приводится в табл. 12.3 *далее в этой главе*.)

Аутентификация

Система безопасности компонента Database Engine состоит из двух разных подсистем безопасности:

- системы безопасности Windows;
- системы безопасности SQL Server.

Система безопасности Windows определяет безопасность на уровне операционной системы, т. е. метод, посредством которого пользователи входят в систему Windows, используя свои учетные записи Windows. (Аутентификация посредством этой подсистемы также называется *аутентификацией Windows*.)

Система безопасности SQL Server определяет дополнительную безопасность, требуемую на уровне системы баз данных, т. е. способ, посредством которого пользователи, уже вошедшие в операционную систему, могут подключаться к серверу базы данных. Система безопасности SQL Server определяет регистрационное имя входа (login) в SQL Server (или просто называемое *логином*), которое создается в системе и ассоциируется с определенным паролем. Некоторые регистрационные имена входа в SQL Server идентичны существующим учетным записям Windows. (Аутентификация посредством этой подсистемы также называется аутентификацией SQL Server.)

На основе этих двух подсистем безопасности компонент Database Engine может работать в одном из следующих режимов аутентификации:

- в режиме Windows;
- в смешанном режиме.

Режим Windows требует, чтобы пользователи входили в систему баз данных исключительно посредством своих учетных записей Windows. Система принимает данные учетной записи пользователя, полагая, что они уже были проверены и одобрены на уровне операционной системы. Такой способ подключения к системе баз данных называется *доверительным соединением* (trusted connection), т. к. система баз данных доверяет, что операционная система уже проверила подлинность учетной записи и соответствующего пароля.

Смешанный режим позволяет пользователям подключаться к компоненту Database Engine посредством аутентификации Windows или аутентификации SQL Server. Это означает, что некоторые учетные записи пользователей можно настроить для использования подсистемы безопасности Windows, а другие, вдобавок к этому, могут использовать также и подсистему безопасности SQL Server.

ПРИМЕЧАНИЕ

Аутентификация SQL Server предоставляется исключительно в целях обратной совместимости. Поэтому следует использовать аутентификацию Windows.

Реализация режима аутентификации

Выбор одного из доступных режимов аутентификации осуществляется посредством среды SQL Server Management Studio. (Среда Management Studio подробно рассмотрена в *главе 3*.) Чтобы установить режим аутентификации Windows, щелкните правой кнопкой сервер баз данных и в контекстном меню выберите пункт **Properties**. Откроется диалоговое окно **Server Properties**, в котором нужно выбрать страницу **Security**, а на ней **Windows Authentication Mode**. Для выбора смешанного режима в этом же диалоговом окне **Server Properties** вам нужно выбрать **Server and Windows Authentication Mode**.

После успешного подключения пользователя к компоненту Database Engine его доступ к объектам базы данных становится независимым от использованного способа аутентификации для входа в базу данных — аутентификации Window или SQL Server аутентификации.

Прежде чем приступить к изучению настройки безопасности сервера базы данных, вам необходимо понять политики и механизмы шифрования, которые рассмотрены в следующих разделах.

Шифрование данных

Шифрование — это процесс приведения данных в запутанное непонятное состояние, вследствие чего повышается уровень их безопасности. Обычно конкретная процедура шифрования осуществляется с использованием определенного алгоритма. Наиболее важный алгоритм шифрования называется RSA, по первым буквам фамилий его создателей — Rivers, Shamir и Adelman.

Компонент Database Engine обеспечивает безопасность данных посредством иерархии уровней шифрования и инфраструктуры управления ключами. Каждый уровень защищает следующий за ним уровень шифрования, используя комбинацию сертификатов, асимметричных и симметричных ключей (рис. 12.1).

На рис. 12.1 *главный сервисный ключ* задает ключ, который управляет всеми другими ключами и сертификатами. Главный сервисный ключ создается автоматически при установке компонента Database Engine. Этот ключ зашифрован с помощью API-интерфейса защиты данных Windows (DPAPI — Data Protection API).

Важным свойством главного сервисного ключа является то, что он управляется системой. Хотя системный администратор может выполнять разные задачи по обслуживанию ключа, ему следует выполнять лишь одну из них — резервное копирование главного сервисного ключа, чтобы его можно было восстановить в случае повреждения.

Как можно видеть на рис. 12.1, *главный сервисный ключ базы данных* является корневым объектом шифрования на уровне базы данных для всех ключей, сертификатов и данных. Каждая база данных имеет один главный ключ базы данных, который создается посредством инструкции `CREATE MASTER KEY` (см. пример 12.1). Поскольку главный ключ базы данных защищен главным сервисным ключом системы, то система может автоматически расшифровывать главный ключ базы данных.

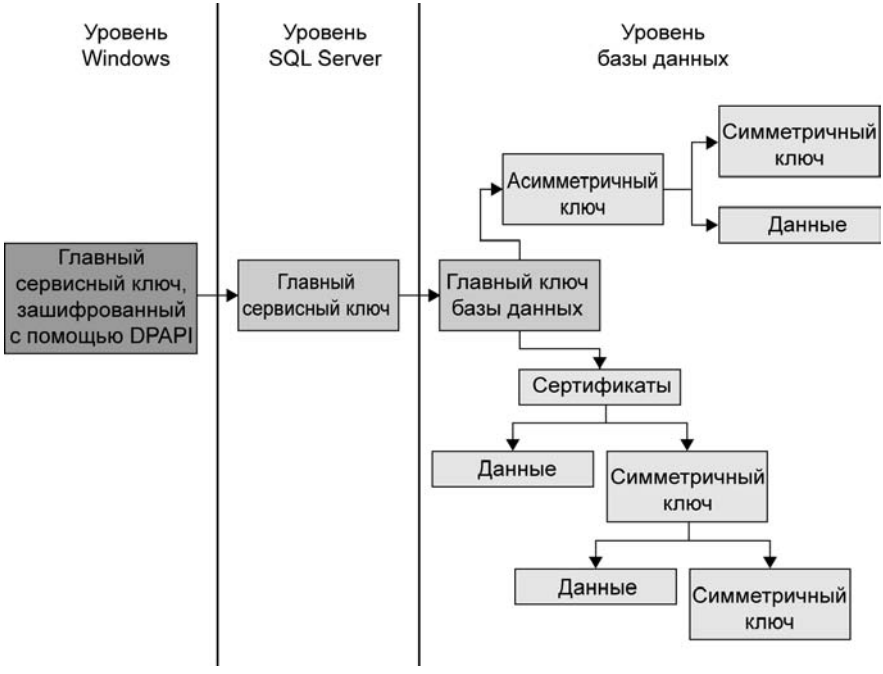


Рис. 12.1. Иерархия уровней шифрования компонента Database Engine

Существующий главный ключ базы данных можно использовать для создания пользовательских ключей. Существует три формы пользовательских ключей:

- симметричные ключи;
- асимметричные ключи;
- сертификаты.

Эти формы пользовательских ключей рассмотрены в следующих далее подразделах.

Симметричные ключи

В системе шифрования с использованием симметричного ключа оба участника обмена — отправитель и получатель сообщения — применяют один и тот же ключ. Иными словами, для шифрования информации и ее обратного расшифровывания используется этот единственный ключ.

Использование симметричных ключей имеет много преимуществ и один недостаток. Одним из преимуществ использования симметричных ключей является то обстоятельство, что с их помощью можно защитить значительно больший объем данных, чем с помощью двух других типов ключей. Кроме этого, использование ключей этого типа намного быстрее, чем использование несимметричных ключей.

Но с другой стороны, использование симметричного ключа в среде распределенной системы может сделать задачу обеспечения целостности шифрования почти невыполнимой, поскольку один и тот же ключ применяется на обоих концах обмена

данными. Таким образом, основной рекомендацией является то, что симметричные ключи должны использоваться только с теми приложениями, в которых зашифрованные данные сохраняются в одном месте.

Язык Transact-SQL поддерживает несколько инструкций и системных функций применительно к симметричным ключам. В частности, для создания симметричного ключа применяется инструкция `CREATE SYMMETRIC KEY`, а для удаления существующего симметричного ключа — инструкция `DROP SYMMETRIC KEY`. Прежде чем симметричный ключ можно использовать для шифрования данных или для защиты другого ключа, его нужно открыть. Для этого используется инструкция `OPEN SYMMETRIC KEY`.

После того как вы откроете симметричный ключ, вам нужно для шифрования использовать системную функцию `EncryptByKey`. Эта функция имеет два входных параметра: идентификатор ключа и текст, который требуется зашифровать. Для расшифровки зашифрованной информации применяется системная функция `DecryptByKey`.

ПРИМЕЧАНИЕ

Подробное описание всех инструкций Transact-SQL, связанных с симметричными ключами, а также системных функций `EncryptByKey` и `DecryptByKey` см. в *электронной документации*.

Асимметричные ключи

В случае если у вас имеется распределенная система или если использование симметричных ключей не обеспечивает достаточного уровня безопасности данных, то следует использовать асимметричные ключи. Асимметричный ключ состоит из двух частей: *личного закрытого ключа* (private key) и соответствующего *общего открытого ключа* (public key). Каждый из этих ключей может расшифровывать данные, зашифрованные другим ключом. Благодаря наличию личного закрытого ключа асимметричное шифрование обеспечивает более высокий уровень безопасности данных, чем симметричное шифрование.

Язык Transact-SQL поддерживает несколько инструкций и системных функций применительно к асимметричным ключам. В частности, для создания нового асимметричного ключа применяется инструкция `CREATE ASYMMETRIC KEY`, а для изменения свойств асимметричного ключа используется инструкция `ALTER ASYMMETRIC KEY`. Для удаления асимметричного ключа применяется инструкция `DROP ASYMMETRIC KEY`.

После того как вы создали асимметричный ключ, для шифрования данных используйте системную функцию `EncryptByAsymKey`. Эта функция имеет два входных параметра: идентификатор ключа и текст, который требуется зашифровать. Для расшифровки информации, зашифрованной с использованием асимметричного ключа, применяется системная функция `DecryptByAsymKey`.

ПРИМЕЧАНИЕ

Подробное описание всех инструкций Transact-SQL, связанных с асимметричными ключами, а также системных функций `EncryptByAsymKey` и `DecryptByAsymKey` см. в *электронной документации.*

Сертификаты

Сертификат открытого ключа, или просто сертификат, представляет собой предложение с цифровой подписью, которое привязывает значение открытого ключа к определенному лицу, устройству или службе, которая владеет соответствующим открытым ключом. Сертификаты выдаются и подписываются *центром сертификации* (Certification Authority (CA)). Сущность, которая получает сертификат из центра сертификации (CA), является субъектом данного сертификата (certificate subject).

ПРИМЕЧАНИЕ

Между сертификатами и асимметричными ключами нет значительной функциональной разницы. Как первый, так и второй используют алгоритм RSA. Основная разница между ними заключается в том, что асимметричные ключи создаются вне сервера.

Сертификаты содержат следующую информацию:

- значение открытого ключа субъекта;
- информацию, идентифицирующую субъекта;
- информацию, идентифицирующую издателя сертификата;
- цифровую подпись издателя сертификата.

Основным достоинством сертификатов является то, что они освобождают хосты от необходимости содержать набор паролей для отдельных субъектов. Когда хост, например, безопасный веб-сервер, обозначает издателя как надежный центр авторизации, этот хост неявно доверяет, что данный издатель выполнил проверку личности субъекта сертификата.

ПРИМЕЧАНИЕ

Сертификаты предоставляют самый высший уровень шифрования в модели безопасности компонента Database Engine. Алгоритмы шифрования с использованием сертификатов требуют большого объема процессорных ресурсов. По этой причине сертификаты следует использовать при реальной необходимости.

Наиболее важной инструкцией применительно к сертификатам является инструкция `CREATE CERTIFICATE`. Использование этой инструкции показано в примере 12.1.

Пример 12.1. Создание сертификата посредством инструкции CREATE CERTIFICATE

```
USE sample;
CREATE MASTER KEY
ENCRYPTION BY PASSWORD = 'pls4w9d16!'
GO
CREATE CERTIFICATE cert01
   WITH SUBJECT = 'Certificate for dbo';
```

Чтобы создать сертификат без параметра ENCRYPTION BY, сначала нужно создать главный ключ базы данных. (Все инструкции CREATE CERTIIFCATE, которые не содержат этот параметр, защищаются главным ключом базы данных.) По этой причине первой инструкцией в примере 12.1 является инструкция CREATE MASTER KEY. После этого инструкция CREATE CERTIFICATE используется для создания нового сертификата cert01, владельцем которого является объект dbo базы данных sample, если этот объект является текущим пользователем.

Редактирование пользовательских ключей

Наиболее важными представлениями каталога применительно к шифрованию являются следующие:

- sys.symmetric_keys;
- sys.asymmetric_keys;
- sys.certificates;
- sys.database_principals.

Первые три представления каталога предоставляют информацию обо всех симметричных ключах, всех асимметричных ключах и всех сертификатах, установленных в текущей базе данных, соответственно. Представление каталога sys.database_principals предоставляет информацию обо всех принципалах в текущей базе данных. Последнее представление каталога можно соединить с любым из первых трех, чтобы получить информацию о владельце определенного ключа.

В примере 12.2 показано получение информации о существующих сертификатах. Подобным образом можно получить информацию о симметричных и асимметричных ключах.

Пример 12.2. Получение информации о существующих сертификатах

```
SELECT p.name, c.name, certificate_id
   FROM sys.database_principals p, sys.certificates c
   WHERE p,principal_id = p,principal_id
```

Часть результата этого запроса:

```
Public                    Cert01      256
Dbo                       Cert01      256
Guest                     Cert01      256
INFORMATION_SCHEMA        cert01      256
Sys                       cert01      256
db_owner                  cert01      256
db_accessadmin            cert01      256
db_securityadmin          cert01      256
```

Расширенное управление ключами SQL Server

Следующим шагом к обеспечению более высокого уровня безопасности ключей является использование расширенного управления ключами (Extensible Key Management (EKM)). Основными целями расширенного управления ключами являются следующие:

- повышение безопасности ключей посредством выбора поставщика функций шифрования;
- общее управление ключами по всему предприятию.

Расширенное управление ключами позволяет сторонним разработчикам регистрировать свои устройства в компоненте Database Engine. Когда такие устройства зарегистрированы, регистрационные имена (logins) в SQL Server могут использовать хранящиеся в них ключи шифрования, а также эффективно использовать продвинутые возможности шифрования, поддерживаемые этими модулями. Расширенное управление ключами позволяет защитить данные от доступа администраторов базы данных (за исключением членов группы `sysadmin`). Таким образом система защищается от пользователей с повышенными привилегиями. Данные можно зашифровывать и расшифровывать, используя инструкции шифрования языка Transact-SQL, а SQL Server может использовать внешнее устройство расширенного управления ключами для хранения ключей.

Способы шифрования данных

SQL Server поддерживает два способа шифрования данных:

- шифрование на уровне столбцов;
- прозрачное шифрование данных.

Шифрование на уровне столбцов позволяет шифровать конкретные столбцы данных. Для реализации этого способа шифрования используется несколько пар сопряженных функций. Далее мы не будем рассматривать этот метод шифрования, поскольку его реализация является сложным ручным процессом, требующим внесения изменений в приложение.

Прозрачное шифрование данных является новой возможностью базы данных, которая зашифровывает файлы базы данных автоматически, не требуя внесения изме-

нений в какие-либо приложения. Таким образом можно предотвратить доступ к информации базы данных несанкционированным лицам, даже если они смогут получить в свое распоряжение основные или резервные файлы базы данных.

Файлы базы данных зашифровываются на уровне страниц. Страницы зашифрованной базы данных шифруются перед тем, как они записываются на диски и расшифровываются при считывании с диска в память.

Как и большинство других методов шифрования, прозрачное шифрование данных основано на ключе шифрования. В нем используется симметричный ключ, посредством которого и защищается база данных.

Применения прозрачного шифрования для защиты определенной базы данных реализуется в четыре этапа:

1. Используя инструкцию `CREATE MASTER KEY`, *создается главный ключ базы данных*. (Применение этой инструкции показано в примере 12.1.)
2. С помощью инструкции `CREATE CERTIFICATE` *создается сертификат* (см. пример 12.1).
3. Используя инструкцию `CREATE DATABASE ENCRYPTION KEY`, *создается ключ шифрования*.
4. *Выполняется конфигурирование базы данных* для использования шифрования. (Этот шаг можно реализовать, присвоив параметру `ENCRYPTION` инструкции `ALTER DATABASE` значение `ON`.)

Настройка безопасности компонента Database Engine

Настройку безопасности компонента Database Engine можно выполнить одним из следующих способов:

- с помощью среды управления Management Studio сервера SQL Server;
- используя инструкции языка Transact-SQL.

Эти два метода рассматриваются в последующих подразделах.

Управление безопасностью с помощью среды Management Studio

Чтобы с помощью среды Management Studio создать новое регистрационное имя, разверните в обозревателе объектов узел сервера, затем разверните папку "Security", в этой папке щелкните правой кнопкой папку "Logins" и в контекстном меню выберите опцию **New Login**. Откроется диалоговое окно **Login — New** (рис. 12.2).

Первым делом нужно решить, какой способ аутентификации применять: Windows или SQL Server. В случае выбора аутентификации Windows, в качестве регистрационного имени (**Login name**) необходимо указать действительное имя пользователя Windows в форме *domain\user_name* (домен\имя_пользователя). А если выбрана аутентификация SQL Server, необходимо ввести новое регистрационное имя (**Login**

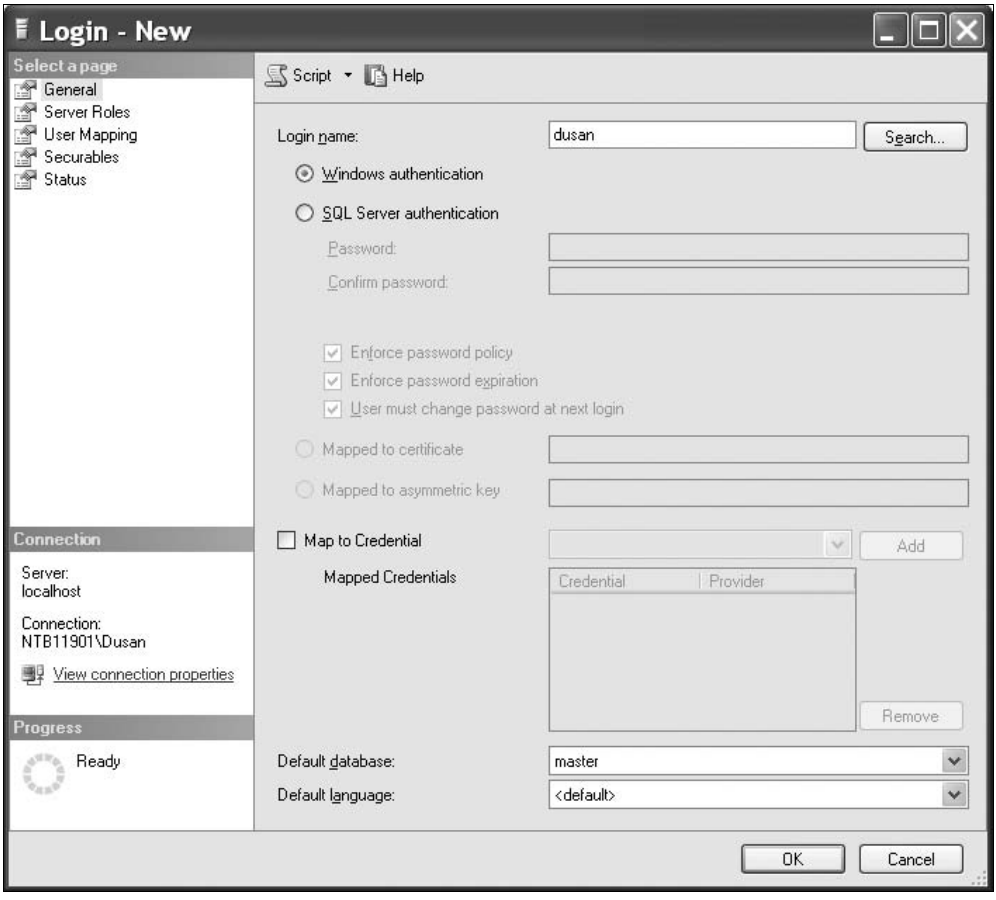


Рис. 12.2. Диалоговое окно **Login — New**

name) и соответствующий пароль (**Password**). Факультативно можно также указать базу данных и язык по умолчанию. *База данных по умолчанию* — это база данных, к которой пользователь автоматически подключается сразу же после входа в компонент Database Engine. Выполнив все эти действия, пользователь может входить в систему под этой новой учетной записью.

Управление безопасностью посредством инструкций Transact-SQL

Для управления безопасностью компонента Database Engine применяются три инструкции языка Transact-SQL: `CREATE LOGIN`, `ALTER LOGIN` и `DROP LOGIN`.

Инструкция `CREATE LOGIN` создает новое регистрационное имя входа в SQL Server. Синтаксис этой инструкции следующий:

```
CREATE LOGIN login_name
{WITH option_list1 |
FROM {WINDOWS [WITH option_list2 [,...]]
| CERTIFICATE certname | ASYMMETRIC KEY key_name }}
```

В параметре *login_name* указывается создаваемое регистрационное имя. Как можно видеть в синтаксисе этой инструкции, в предложении WITH можно указать один или несколько параметров для регистрационного имени или указать в предложении FROM сертификат, асимметричный ключ или учетную запись пользователя Windows, связанную с соответствующим регистрационным именем.

В списке *option_list1* указывается несколько параметров, наиболее важным из которых является параметр PASSWORD, который задает пароль для данного регистрационного имени (пример 12.3). (Другие возможные параметры — DEFAULT_DATABASE, DEFAULT_LANGUAGE и CHECK_EXPIRATION.)

Как видно из синтаксиса инструкции CREATE LOGIN, предложение FROM может содержать один из следующих параметров:

- WINDOWS — указывает, что данное регистрационное имя соотносится с существующей учетной записью пользователя Window (см. пример 12.4). Этот параметр можно указать с другими подпараметрами, такими как DEFAULT_DATABASE и DEFAULT_LANGUAGE;
- SERTIFICATE — задает имя сертификата для привязки к данному регистрационному имени;
- ASYMMETRIC KEY — задает имя асимметричного ключа для привязки к данному регистрационному имени. (Сертификат и асимметричный ключ уже должны присутствовать в базе данных master.)

В примерах 12.3 и 12.4 показано создание разных форм регистрационного имени. В примере 12.3 создается регистрационное имя mary с паролем you1know4it9!.

Пример 12.3. Создание нового регистрационного имени

```
USE sample;
CREATE LOGIN mary WITH PASSWORD = 'you1know4it9!';
```

В примере 12.4 создается регистрационное имя pete, которое сопоставляется с учетной записью пользователя Windows с таким же самым именем пользователя.

Пример 12.4. Создание регистрационного имени, сопоставленного с учетной записью Windows

```
USE sample;
CREATE LOGIN [NTB11901\pete] FROM WINDOWS;
```

ПРИМЕЧАНИЕ

Для конкретной системной среды нужно должным образом изменить имя компьютера и имя пользователя (в примере 12.4 это NTB11901 и pete соответственно).

Вторая инструкция языка Transact-SQL для обеспечения безопасности — ALTER LOGIN — изменяет свойства определенного регистрационного имени. С по-

мощью этой инструкции можно изменить текущий пароль и его конечную дату действия, параметры доступа, базу данных по умолчанию и язык по умолчанию. Также можно задействовать или отключить определенное регистрационное имя.

Наконец, инструкция `DROP LOGIN` применяется для удаления существующего регистрационного имени. Однако регистрационное имя, которое ссылается (владеет) на другие объекты, удалить нельзя.

Схемы

Схемы используются в модели безопасности компонента Database Engine для упрощения взаимоотношений между пользователями и объектами, и, следовательно, схемы имеют очень большое влияние на взаимодействие пользователя с компонентом Database Engine. В этом разделе рассматривается роль схем в безопасности компонента Database Engine. В первом подразделе описывается взаимодействие между схемами и пользователями, а во втором обсуждаются все три инструкции языка Transact-SQL, применяемые для создания и модификации схем.

Разделение пользователей и схем

Схема — это коллекция объектов базы данных, имеющая одного владельца и формирующая одно пространство имен. (Две таблицы в одной и той же схеме не могут иметь одно и то же имя.) Компонент Database Engine поддерживает именованные схемы с использованием понятия *принципала* (principal). Как уже упоминалось, принципалом может быть любой:

- индивидуальный принципал;
- групповой принципал.

Индивидуальный принципал представляет одного пользователя, например, в виде регистрационного имени или учетной записи пользователя Windows. Групповым принципалом может быть группа пользователей, например, роль или группа Windows. Принципалы владеют схемами, но владение схемой может быть с легкостью передано другому принципалу без изменения имени схемы.

Отделение пользователей базы данных от схем дает значительные преимущества, такие как:

- один принципал может быть владельцем нескольких схем;
- несколько индивидуальных принципалов могут владеть одной схемой посредством членства в ролях или группах Windows;
- удаление пользователя базы данных не требует переименования объектов, содержащихся в схеме этого пользователя.

Каждая база данных имеет схему по умолчанию, которая используется для определения имен объектов, ссылки на которые делаются без указания их полных уточненных имен. В схеме по умолчанию указывается первая схема, в которой сервер базы данных будет выполнять поиск для разрешения имен объектов. Для настройки

и изменения схемы по умолчанию применяется параметр DEFAULT_SCHEMA инструкции CREATE USER или ALTER USER. Если схема по умолчанию DEFAULT_SCHEMA не определена, в качестве схемы по умолчанию пользователю базы данных назначается схема dbo. Все схемы по умолчанию подробно описаны в *разд. "Схемы базы данных по умолчанию" далее в этой главе*.

Инструкции языка DDL для работы со схемами

Для работы со схемами используются следующие три инструкции языка Transact-SQL:

- CREATE SCHEMA;
- ALTER SCHEMA;
- DROP SCHEMA.

Эти инструкции рассмотрены далее в следующих подразделах.

Инструкция *CREATE SCHEMA*

В примере 12.5 показано создание схемы и ее использование для управления безопасностью базы данных.

ПРИМЕЧАНИЕ

Прежде чем выполнять пример 12.5, необходимо создать пользователей базы данных peter и mary. Это можно сделать, выполнив код из примера 12.7 в *разд. "Управление безопасностью базы данных посредством инструкций языка Transact-SQL" далее в этой главе*.

Пример 12.5. Создание схемы и ее применение

```
USE sample;
GO
CREATE SCHEMA my_schema AUTHORIZATION peter
GO
CREATE TABLE product
   (product_no CHAR(10) NOT NULL UNIQUE,
   product_name CHAR(20) NULL,
   price MONEY NULL);
GO
CREATE VIEW product_info
   AS SELECT product_no, product_name
      FROM product;
GO
GRANT SELECT TO mary;
DENY UPDATE TO mary;
```

В примере 12.5 создается схема my_schema, содержащая таблицу product и представление product_info. Пользователь базы данных peter является принципалом уровня базы данных, а также владельцем схемы. (Владелец схемы указывается посредством параметра AUTHORIZATION. Принципал может быть владельцем других схем и не может использовать текущую схему в качестве схемы по умолчанию.)

ПРИМЕЧАНИЕ

Две другие инструкции, применяемые для работы с разрешениями для объектов базы данных, GRANT и DENY, подробно рассматриваются далее в этой главе. В примере 12.5 инструкция GRANT предоставляет инструкции SELECT разрешения для всех создаваемых в схеме объектов, тогда как инструкция DENY запрещает инструкции UPDATE разрешения для всех объектов схемы.

С помощью инструкции CREATE SCHEMA можно создать схему, сформировать содержащиеся в этой схеме таблицы и представления, а также предоставить, запретить или удалить разрешения на защищаемый объект. Как упоминалось ранее, защищаемые объекты — это ресурсы, доступ к которым регулируется системой авторизации SQL Server. Существует три основные области защищаемых объектов: сервер, база данных и схема, которые содержат другие защищаемые объекты, такие как регистрационные имена, пользователи базы данных, таблицы и хранимые процедуры.

Инструкция CREATE SCHEMA является *атомарной*. Иными словами, если в процессе выполнения этой инструкции происходит ошибка, не выполняется ни одна из содержащихся в ней подынструкций.

Порядок указания создаваемых в инструкции CREATE SCHEMA объектов базы данных может быть произвольным, с одним исключением: представление, которое ссылается на другое представление, должно быть указано после представления, на которое оно ссылается.

Принципалом уровня базы данных может быть пользователь базы данных, роль или роль приложения. (Роли и роли приложения рассматриваются в *разд. "Роли" далее в этой главе*.) Принципал, указанный в предложении AUTHORIZATION инструкции CREATE SCHEMA, является владельцем всех объектов, созданных в этой схеме. Владение содержащихся в схеме объектов можно передавать любому принципалу уровня базы данных посредством инструкции ALTER AUTHORIZATION.

Для исполнения инструкции CREATE SCHEMA пользователь должен обладать правами базы данных CREATE SCHEMA. Кроме этого, для создания объектов, указанных в инструкции CREATE SCHEMA, пользователь должен иметь соответствующие разрешения CREATE.

Инструкция *ALTER SCHEMA*

Инструкция ALTER SCHEMA перемещает объекты между разными схемами одной и той же базы данных. Инструкция ALTER SCHEMA имеет следующий синтаксис:

```
ALTER SCHEMA schema_name TRANSFER object_name
```

Использование инструкции ALTER SCHEMA показано в примере 12.6.

Пример 12.6. Применение инструкции ALTER SCHEMA

```
USE AdventureWorks2012;
ALTER SCHEMA HumanResources TRANSFER Person.Contact;
```

В примере 12.6 изменяется схема HumanResources базы данных AdventureWorks2012, перемещая в нее таблицу Contact из схемы Person этой же базы данных.

Инструкцию ALTER SCHEMA можно использовать для перемещения объектов между разными схемами только одной и той же базы данных. (Отдельные объекты в схеме можно изменить посредством инструкции ALTER TABLE или ALTER VIEW.)

Инструкция *DROP SCHEMA*

Для удаления схемы из базы данных применяется инструкция DROP SCHEMA. Схему можно удалить только при условии, что она не содержит никаких объектов. Если схема содержит объекты, попытка выполнить инструкцию DROP SCHEMA будет неуспешной.

Как указывалось ранее, владельца схемы можно изменить посредством инструкции ALTER AUTHORIZATION, которая изменяет владение сущностью.

ПРИМЕЧАНИЕ

Язык Transact-SQL не поддерживает инструкции CREATE AUTHORIZATION и DROP AUTHORIZATION. Владелец схемы указывается с помощью инструкции CREATE SCHEMA.

Безопасность базы данных

Пользователь может войти в систему баз данных, используя учетную запись пользователя Windows или регистрационное имя входа в SQL Server. Для последующего доступа и работы с определенной базой данных пользователь также должен иметь учетную запись пользователя базы данных. Для работы с каждой отдельной базой данных требуется иметь учетную запись пользователя именно для этой базы данных. Учетную запись пользователя базы данных можно сопоставить с существующей учетной записью пользователя Windows, группой Windows (в которой пользователь имеет членство), регистрационным именем или ролью.

Управлять безопасностью баз данных можно с помощью:

- среды Management Studio;
- инструкций языка Transact-SQL.

Оба эти способа рассматриваются в следующих подразделах.

Управление безопасностью базы данных с помощью среды Management Studio

Чтобы добавить пользователя базы данных с помощью среды Management Studio, разверните узел сервера и в нем папку "Databases", в этой папке разверните узел требуемой базы данных, а в ней папку "Security". Щелкните правой кнопкой мыши папку "Users" и в контекстном меню выберите пункт **New User**. Откроется диалоговое окно **Database User — New** (рис. 12.3), в котором следует ввести имя пользователя **User name** и выбрать соответствующее регистрационное имя **Login name**.

Факультативно можно выбрать схему по умолчанию для данного пользователя.

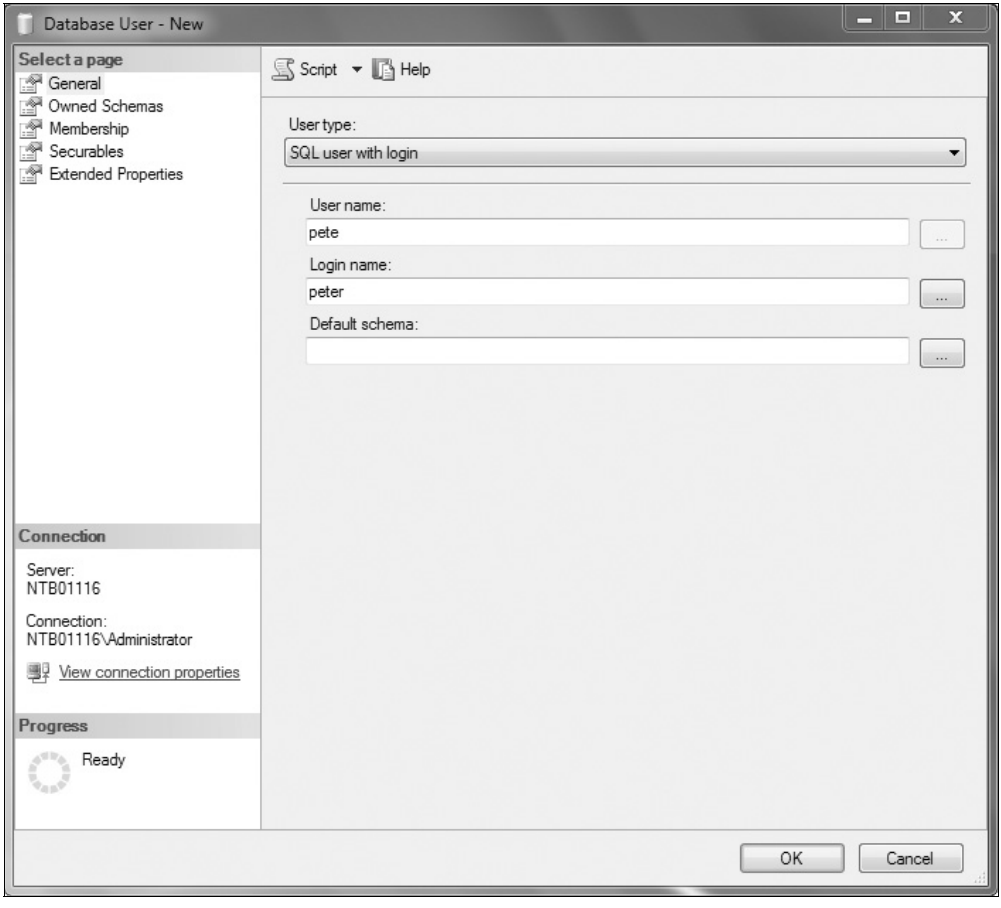


Рис. 12.3. Диалоговое окно **Database User — New**

Управление безопасностью базы данных посредством инструкций языка Transact-SQL

Для добавления пользователя в текущую базу данных используется инструкция `CREATE USER`.

Синтаксис этой инструкции выглядит таким образом:

```
CREATE USER user_name
   [FOR {LOGIN login|CERTIFICATE cert_name | ASYMMETRIC KEY key_name}]
      [WITH DEFAULT_SCHEMA = schema_name]
```

Параметр *user_name* определяет имя, по которому пользователь идентифицируется в базе данных, а в параметре *login* указывается регистрационное имя, для которого создается данный пользователь. В параметрах *cert_name* и *key_name* указываются соответствующий сертификат и асимметричный ключ соответственно. Наконец, в параметре WITH DEFAULT_SCHEMA указывается первая схема, с которой сервер базы данных будет начинать поиск для разрешения имен объектов для данного пользователя базы данных.

Применение инструкции CREATE USER показано в примере 12.7.

Пример 12.7. Создание пользователя

```
USE sample;
CREATE USER peter FOR LOGIN [NTB1901\pete];
CREATE USER mary FOR LOGIN mary WITH DEFAULT_SCHEMA =
my_schema;
```

ПРИМЕЧАНИЕ

Для успешного выполнения на вашем компьютере первой инструкции примера 12.7 требуется сначала создать учетную запись Windows для пользователя pete и вместо домена (сервера) NTB11901 указать имя вашего сервера.

В примере 12.7 первая инструкция CREATE USER создает пользователя базы данных peter для пользователя pete учетной записи Windows. Схемой по умолчанию для пользователя pete будет dbo, поскольку для параметра DEFAULT SCHEMA значение не указано. (Схемы по умолчанию подробно описываются в *разд. "Схемы базы данных по умолчанию" далее в этой главе*.)

Вторая инструкция CREATE USER создает нового пользователя базы данных mary. Схемой по умолчанию для этого пользователя будет схема my_schema. (Параметру DEFAULT_SCHEMA можно присвоить в качестве значения схему, которая в данное время не существует в базе данных.)

ПРИМЕЧАНИЕ

Каждая база данных имеет своих конкретных пользователей. Поэтому инструкцию CREATE USER необходимо выполнить для каждой базы данных, для которой должна существовать учетная запись пользователя. Кроме этого, для определенной базы данных регистрационное имя входа в SQL Server может иметь только одного пользователя базы данных.

С помощью инструкции ALTER USER можно изменить имя пользователя базы данных, изменить схему пользователя по умолчанию или переопределить пользователя

с другим регистрационным именем. Подобно инструкции CREATE USER, пользователю можно присвоить схему по умолчанию прежде, чем она создана.

Для удаления пользователя из текущей базы данных применяется инструкция DROP USER. Пользователя, который является владельцем защищаемых объектов (объектов базы данных), удалить нельзя.

Схемы базы данных по умолчанию

Каждая база данных в системе имеет следующие схемы по умолчанию:

- guest;
- dbo;
- INFORMATION_SCHEMA;
- sys.

Компонент Database Engine позволяет пользователям, которые не имеют учетной записи пользователя, работать с базой данной, используя схему guest. (Каждая созданная база данных имеет эту схему.) Для схемы guest можно применять разрешения таким же образом, как и для любой другой схемы. Кроме этого, схему guest можно удалить из любой базы данных, кроме системных баз данных master и tempdb.

Каждый объект базы данных принадлежит одной, и только одной схеме, которая является схемой по умолчанию для данного объекта. Схема по умолчанию может быть определена явно или неявно. Если при создании объекта его схема по умолчанию не определена явно, этот объект принадлежит к схеме dbo. Кроме этого, при использовании принадлежащей ему базы данных регистрационное имя всегда имеет специальное имя пользователя dbo.

Вся информация о схемах содержится в схеме INFORMATION_SCHEMA (см. *главу 11*). Схема sys, как можно догадаться, содержит системные объекты, такие как представления каталога.

Роли

Когда нескольким пользователям требуется выполнять подобные действия в определенной базе данных и при этом они не являются членами соответствующей группы Windows, то можно воспользоваться *ролью базы данных*, задающей группу пользователей базы данных, которые могут иметь доступ к одним и тем же объектам базы данных.

Членами роли базы данных могут быть любые из следующих:

- группы и учетные записи Windows;
- регистрационные имена входа в SQL Server;
- другие роли.

Архитектура безопасности компонента Database Engine включает несколько "системных" ролей, которые имеют специальные явные разрешения. Кроме ролей, определяемых пользователями, существует два типа предопределенных ролей:

- фиксированные серверные роли;
- фиксированные роли базы данных.

Помимо их в следующих разделах также рассматриваются и такие типы ролей:

- роли приложений;
- определяемые пользователем серверные роли;
- определяемые пользователем роли баз данных.

Фиксированные серверные роли

Фиксированные серверные роли определяются на уровне сервера и поэтому находятся вне баз данных, принадлежащих серверу баз данных. В табл. 12.1 приводится список фиксированных серверных ролей и краткое описание действий, которые могут выполнять члены этих ролей.

12.1. *Фиксированные серверные роли*

sysadmin	
serveradmin	
setupadmin	
securityadmin	CREATE DATABASE !
processadmin	
dbcreator	"
diskadmin	

Управление фиксированными серверными ролями

Членов фиксированной серверной роли можно добавлять и удалять двумя способами:

- используя среду Management Studio;
- используя инструкции языка Transact-SQL.

Чтобы добавить регистрационное имя в члены фиксированной серверной роли посредством среды Management Studio, разверните в обозревателе объектов узел сервера, в нем папку "Security", а в ней разверните папку "Server Roles". Щелкните правой кнопкой мыши роль, в которую требуется добавить регистрационное имя, и

в появившемся контекстном меню выберите пункт **Properties**. После открытия диалогового окна **Server Role Properties — sysadmin** (рис. 12.4) нажмите кнопку **Add**.

Затем в открывшемся диалоговом окне нажмите кнопку **Browse** и выберите регистрационное имя, которое нужно добавить в серверную роль. Теперь это регистрационное имя является членом данной серверной роли и наследует все параметры доступа, предоставленные этой роли.

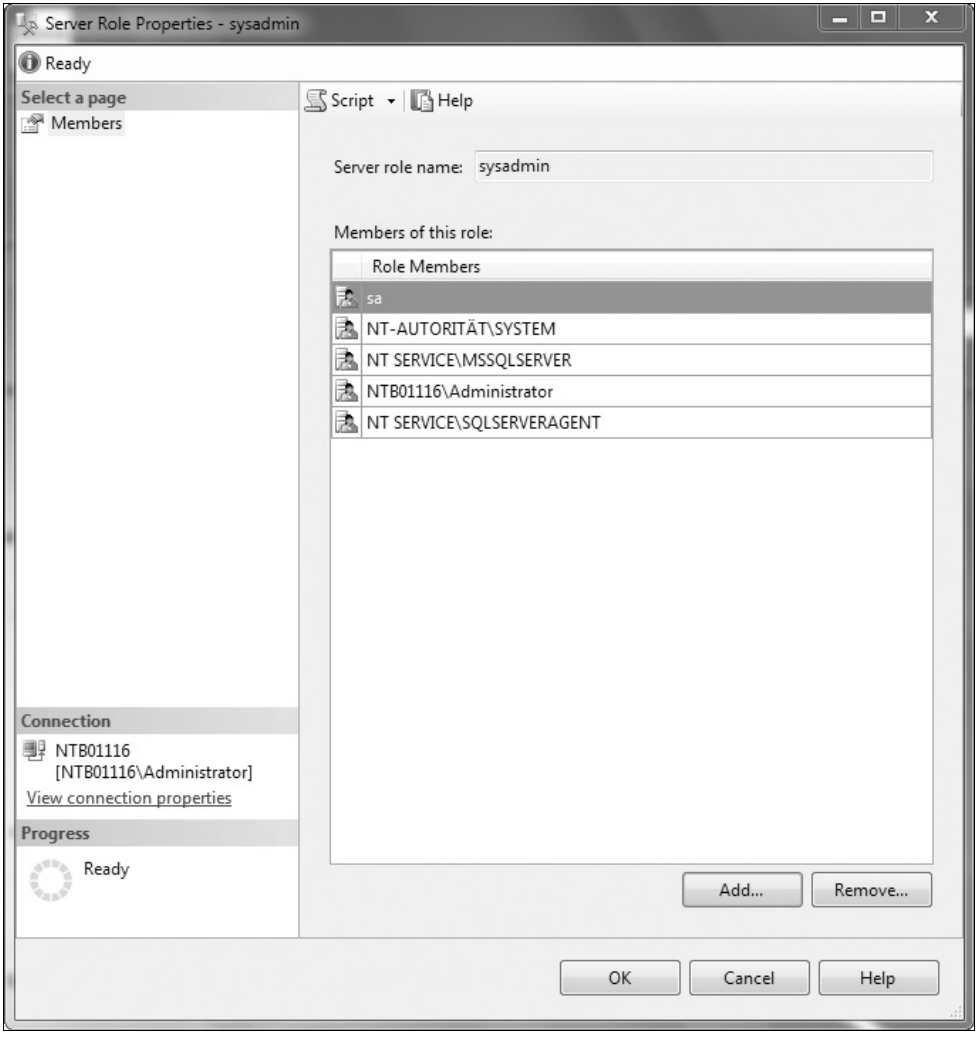


Рис. 12.4. Диалоговое окно **Server Role Properties — sysadmin**

Для добавления и удаления членов в фиксированные серверные роли используются инструкции языка Transact-SQL `CREATE SERVER ROLE` и `DROP SERVER ROLE` соответственно. А для изменения членства в серверной роли используется инструкция `ALTER SERVER ROLE`. Использование инструкций `CREATE SERVER ROLE` и `ALTER SERVER ROLE` показано в примере 12.9 далее в этой главе.

ПРИМЕЧАНИЕ

Фиксированные серверные роли нельзя добавлять, удалять или переименовывать. Кроме этого, только члены фиксированных серверных ролей могут выполнять системные процедуры для добавления или удаления регистрационного имени в роли.

Регистрационное имя *sa*

Регистрационное имя `sa` является регистрационным именем системного администратора. В версиях более ранних, чем SQL Server 2005, в которых роли отсутствовали, регистрационное имя `sa` предоставляло все возможные разрешения для задач системного администрирования. В более новых версиях SQL Server регистрационное имя `sa` включено единственно с целью обратной совместимости. Это регистрационное имя всегда является членом фиксированной серверной роли `sysadmin` и его нельзя удалить из этой роли.

ПРИМЕЧАНИЕ

Регистрационное имя `sa` следует использовать только в тех случаях, когда нет другого способа войти в систему базы данных.

Фиксированные роли базы данных

Фиксированные роли базы данных определяются на уровне базы данных и поэтому существуют в каждой базе данных, принадлежащей серверу баз данных. В табл. 12.2 приводится список фиксированных ролей базы данных и краткое описание действий, которые могут выполнять члены этих ролей.

12.2. *Фиксированные роли базы данных*

db_owner	# $, $
db_accessadmin	# $, $ $ $
db_datareader	# $, $ $
db_datawriter	# $, $, $ $ $ $
db_ddladmin	# $, $ DDL
db_securityadmin	# $, $,
db_backupoperator	# $, $

12.2 *(окончание)*

db_denydatareader	# $, $
db_denydatawriter	# $, $

Члены фиксированных ролей баз данных могут выполнять разные действия. Подробную информацию о действиях, которые могут выполнять члены каждой фиксированной роли базы данных см. в *электронной документации*.

Кроме перечисленных в табл. 12.2 фиксированных ролей базы данных, существует специальная фиксированная роль базы данных `public`, которая рассмотрена в следующем подразделе.

Фиксированная роль базы данных *public*

Фиксированная роль базы данных `public` является специальной ролью, членом которой являются все законные пользователи базы данных. Она охватывает все разрешения по умолчанию для пользователей базы данных. Это позволяет предоставить всем пользователям, которые не имеют должных разрешений, набор разрешений (обычно ограниченный). Роль `public` предоставляет все разрешения по умолчанию для пользователей базы данных и не может быть удалена. Пользователям, группам или ролям нельзя присвоить членство в этой роли, поскольку они имеют его по умолчанию. Применение роли `public` показано в примере 12.19 далее в этой главе.

По умолчанию роль `public` разрешает пользователям выполнять следующие действия:

- просматривать системные таблицы и отображать информацию из системной базы данных `master`, используя определенные системные процедуры;
- выполнять инструкции, для которых не требуются разрешения, например, `PRINT`.

Присвоения пользователю членства в фиксированной роли базы данных

Чтобы присвоить пользователю базы данных членство в фиксированной роли базы данных с помощью среды Management Studio, разверните сервер и папку "Databases", а в ней базу данных, затем разверните папку "Security", "Roles" и папку "Databases Roles". Щелкните правой кнопкой мыши роль, в которую требуется добавить пользователя, и в контекстном меню выберите пункт **Properties**. В диалоговом окне свойств роли базы данных нажмите кнопку **Add** и выберите пользователей, которым нужно присвоить членство в этой роли. Теперь этот пользователь является членом данной роли базы данных и наследует все параметры доступа, предоставленные этой роли.

Роли приложений

Роли приложения позволяют принудительно обеспечивать безопасность для определенного приложения. Иными словами, роли приложения позволяют приложению взять на себя ответственность за аутентификацию пользователя, вместо того, чтобы это делала система баз данных. Например, если служащие компании могут изменять данные о сотрудниках только посредством какого-либо приложения (а не посредством инструкций языка Transact-SQL или какого-либо другого средства), для этого приложения можно создать роль приложения.

Роли приложений существенно отличаются от всех других типов ролей. Во-первых, роли приложений не имеют членов, поскольку они используют только приложения, и поэтому им нет необходимости предоставлять разрешения непосредственно пользователям. Во-вторых, для активации роли приложения требуется пароль.

Когда приложение активирует для сеанса роль приложения, этот сеанс утрачивает все разрешения, применимые к именам ввода, учетным записям пользователей, группам пользователей или ролям во всех базах данных. Так как эти роли применимы только к базе данных, в которой они находятся, сеанс может получить доступ к другой базе данных только посредством разрешений, предоставленных пользователю `guest` базы данных, к которой требуется доступ. Поэтому, если база данных не имеет пользователя `guest`, сеанс не может получить доступ к этой базе данных.

В следующих двух подразделах описывается управление ролями приложений.

Управление ролями приложений посредством среды Management Studio

Чтобы создать роль приложения с помощью среды Management Studio, разверните узел сервера, папку "Databases", требуемую базу данных, папку "Security". Щелкните правой кнопкой папку "Roles", в появившемся контекстном меню выберите пункт **New**, а во вложенном меню выберите пункт **New Application Role**. В открывшемся диалоговом окне **Application Role — New** введите в соответствующие поля имя новой роли приложения, пароль и, необязательно, схему по умолчанию. Нажмите кнопку **OK**, чтобы сохранить роль.

Управление ролями приложений посредством инструкций Transact-SQL

Для создания, изменения и удаления ролей приложений применяются инструкции языка Transact-SQL `CREATE APPLICATION ROLE`, `ALTER APPLICATION ROLE` и `DROP APPLICATION ROLE` соответственно.

Инструкция `CREATE APPLICATION ROLE`, создающая роль приложения для текущей базы данных, имеет два параметра. В первом параметре указывается пароль роли, а во втором — схема по умолчанию, т. е. первая схема, к которой будет обращаться сервер для разрешения имен объектов для этой роли.

В примере 12.8 показано создание роли приложения `weekly_reports` в базе данных `sample`.

Пример 12.8. Создание роли приложения

```
USE sample;
CREATE APPLICATION ROLE weekly_reports
WITH PASSWORD ='x1y2z3w4!',
        DEFAULT_SCHEMA = my_schema;
```

Инструкция ALTER APPLICATION ROLE применяется для изменения имени, пароля или схемы по умолчанию существующей роли приложения. Синтаксис этой инструкции очень сходен с синтаксисом инструкции CREATE APPLICATION ROLE. Для выполнения инструкции ALTER APPLICATION ROLE для этой роли необходимо иметь разрешение ALTER.

Для удаления роли приложения используется инструкция DROP APPLICATION ROLE. Роль приложения нельзя удалить, если она владеет какими-либо объектами (защищаемыми объектами).

Активация ролей приложений

При установлении соединения необходимо выполнить системную процедуру sp_setupprole, чтобы активировать разрешения, связанные с ролью приложения. Эта процедура имеет следующий синтаксис:

```
sp_setapprole [@rolename =] 'role',
      [@password =] 'password'
      [,[@encrypt =] 'encrypt_style']
```

В параметре *role* указывается имя роли приложения, определенного в текущей базе данных, в параметре *password* — пароль для этой роли, а в параметре *encrypt_style* — метод шифрования, указанный для пароля.

При активации роли приложения с помощью системной процедуры sp_setupprole необходимо иметь в виду следующее:

- активированную роль приложения нельзя деактивировать в текущей базе данных, пока сеанс не отсоединится от системы;
- роль приложения всегда привязана к базе данных, т. е. ее область видимости ограничивается текущей базой данных. Если в течение сеанса изменить текущую базу данных, то в ней можно будет выполнять действия, зависящие от разрешений в этой базе данных.

ПРИМЕЧАНИЕ

Конструкция ролей приложений в SQL Server 2012 не является оптимальной вследствие своей неоднородности. В частности, роли приложения создаются посредством инструкций языка Transact-SQL, после чего они активируются с помощью системной процедуры.

Определяемые пользователем роли сервера

Определяемые пользователем роли сервера впервые стали применяться в SQL Server 2012. Для создания и удаления этих ролей используются инструкции языка Transact-SQL `CREATE SERVER ROLE` и `DROP SERVER ROLE` соответственно. Для добавления или удаления членов роли сервера используется инструкция `ALTER SERVER ROLE`. Использование инструкций `CREATE SERVER ROLE` и `ALTER SERVER ROLE` показано в примере 12.9.

Пример 12.9. Создание определяемой пользователем роли сервера и добавление в нее нового члена

```
USE master;
GO
CREATE SERVER ROLE programadmin;
ALTER SERVER ROLE programadmin ADD MEMBER mary;
```

Определяемые пользователем роли баз данных

Обычно определяемые пользователем роли базы данных применяются, когда группе пользователей базы данных требуется выполнять общий набор действий в базе данных и отсутствует применимая группа пользователей Windows. Для создания, изменения и удаления этих ролей применяется или среда Management Studio, или инструкции языка Transact-SQL `CREATE ROLE`, `ALTER ROLE` и `DROP ROLE`. Управление определяемыми пользователем ролями базы данных рассматривается в следующих двух подразделах.

Управление определяемыми пользователем ролями базы данных с помощью среды Management Studio

Чтобы создать определяемую пользователем роль базы данных с помощью среды Management Studio, разверните узел сервера, папку "Databases", требуемую базу данных, папку "Security". Щелкните правой кнопкой папку "Roles", в появившемся контекстном меню выберите пункт **New**, а во вложенном меню выберите пункт **New Database Role**. В открывшемся диалоговом окне **Database Role — New** (рис. 12.5) введите в соответствующее поле имя новой роли.

Нажмите кнопку **Add**, чтобы добавить членов в новую роль. Выберите требуемых членов (пользователей и/или другие роли) новой роли базы данных и нажмите кнопку **OK**.

Управление определяемыми пользователем ролями базы данных с помощью инструкций Transact-SQL

Для создания новой определяемой пользователем роли базы данных в текущей базе данных применяется инструкция `CREATE ROLE`. Синтаксис этой инструкции выглядит таким образом:

```
CREATE ROLE role_name [AUTHORIZATION owner_name]
```

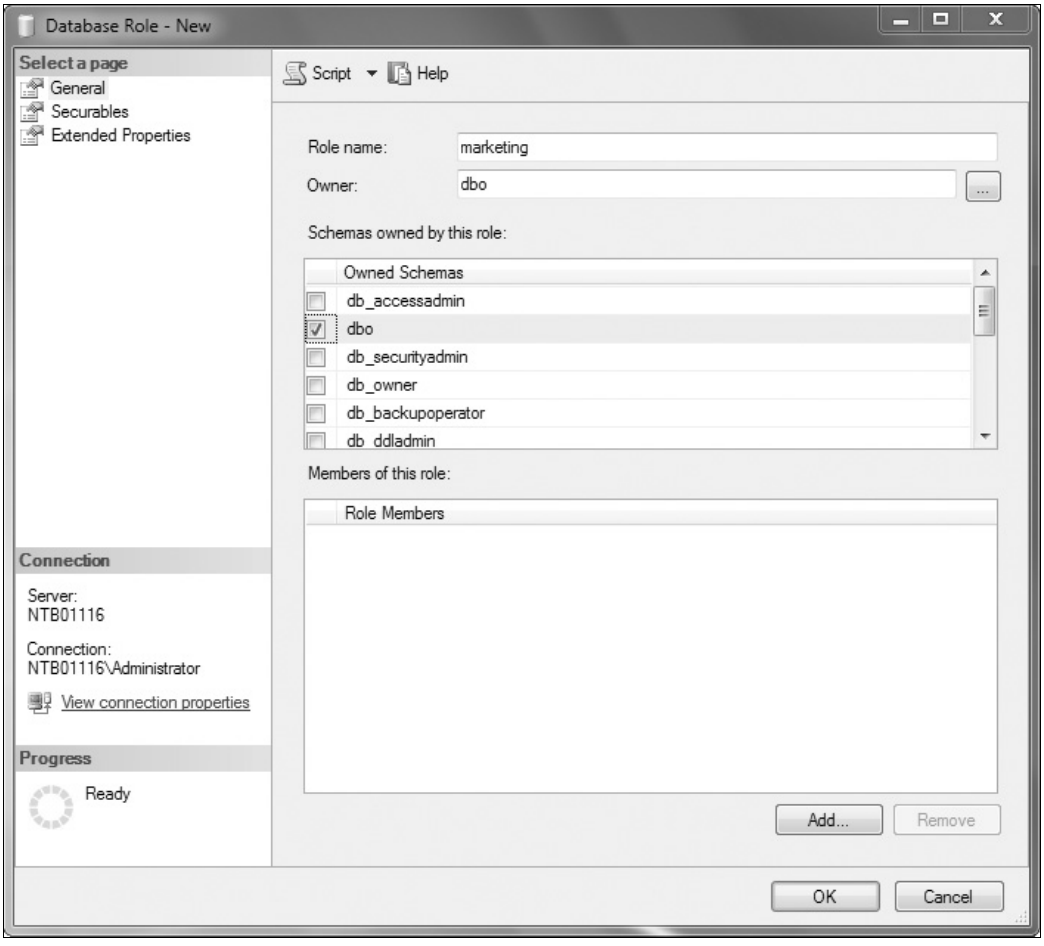


Рис. 12.5. Диалоговое окно **Database Role — New**

В параметре *role_name* инструкции указывается имя создаваемой определяемой пользователем роли, а в параметре *owner_name* — пользователь базы данных или роль, которая будет владельцем новой роли. (Если пользователь не указан, владельцем роли будет пользователь, исполняющий инструкцию `CREATE ROLE`.)

Для изменения имени определяемой пользователем роли базы данных применяется инструкция `ALTER ROLE`, а для удаления роли из базы данных — инструкция `DROP ROLE`. Роль, которая является владельцем защищаемых объектов (т. е. объектов базы данных), удалить нельзя. Чтобы удалить такую роль, сначала нужно изменить владельца этих объектов.

В примере 12.10 показано создание определяемой пользователем роли базы данных и добавление в нее членов.

Пример 12.10. Создание определяемой пользователем роли базы данных

```
USE sample;
CREATE ROLE marketing AUTHORIZATION peter;
GO
ALTER ROLE marketing ADD MEMBER 'peter';
ALTER ROLE marketing ADD MEMBER 'mary';
```

В примере 12.10 сначала создается определяемая пользователем роль базы данных `marketing`, а затем, предложением `ADD MEMBER` инструкции `ALTER ROLE`, в нее добавляются два члена — `peter` и `mary`.

Авторизация

Выполнять инструкции или осуществлять операции с объектами баз данных могут только авторизованные пользователи. Попытка выполнения любой из этих задач неавторизованным пользователем будет неудачной. Для выполнения задач, связанных с авторизацией, используются следующие три инструкции языка Transact-SQL:

- `GRANT`;
- `DENY`;
- `REVOKE`.

Прежде чем приступить к рассмотрению этих инструкций, будет полезным освежить наиболее важные факты о модели безопасности компонента Database Engine. Итак, эта модель разделяет мир сервера базы данных на принципалов безопасности и защищаемые объекты. Каждый защищаемый объект имеет связанные с ним разрешения, которые могут быть предоставлены принципалу. Принципалы, такие как отдельные лица, группы или приложения, могут обращаться к защищаемым объектам. *Защищаемые объекты* — это ресурсы, доступ к которым регулируется подсистемой авторизации. Существует три основных класса защищаемых объектов: сервер, база данных и схема, которые содержат другие защищаемые объекты, такие как регистрационные имена, пользователи базы данных, таблицы и хранимые процедуры.

Инструкция GRANT

Инструкция `GRANT` предоставляет разрешения принципалам на защищаемые объекты. Эта инструкция имеет следующий синтаксис:

```
GRANT {ALL [PRIVILEGES]} permission_list
   [ON [class::] securable] TO principal_list [WITH GRANT OPTION]
   [AS principal ]
```

Предложение `ALL` означает, что указанному принципалу предоставляются все применимые к указанному защищаемому объекту разрешения. (Список конкретных защищаемых объектов см. в *электронной документации*.) В параметре `permission_`

`list` указываются разрешаемые инструкции или объекты (разделенные запятыми), а в параметре `class` — класс или имя защищаемого объекта, для которого предоставляются разрешения. Предложение `ON` *`securable`* указывает защищаемый объект, на который предоставляется разрешение (см. пример 12.15 далее в этом разделе). В параметре *`prinicpal_list`* перечисляются все учетные записи (разделенные запятыми), которым предоставляются разрешения. Параметр *`principal`* и составляющие списка *`principal_list`* могут быть учетной записью пользователя Windows, регистрационным именем или учетной записью пользователя, сопоставленной с сертификатом, регистрационным именем, сопоставленным с асимметричным ключом, пользователем базы данных, ролью базы данных или ролью приложения.

В табл. 12.3 приводятся разрешения и защищаемые объекты, к которым они применяются, а также краткое описание предоставляемых каждым разрешением возможностей.

ПРИМЕЧАНИЕ

В табл. 12.3 перечислены только наиболее важные разрешения. Так как модель безопасности компонента Database Engine является иерархической, то она содержит многие гранулярные разрешения, которые не отображены в списке. Описание этих разрешений см. в *электронной документации*.

12.3. *Разрешения и соответствующие защищаемые объекты*

`SELECT`	% , - , , & &	# ! $ ( ) . ' ! - $ $, . (* ,)
`INSERT`	% , - ,	# ! $ $
`UPDATE`	% , - ,	# ! $ $ - . ' ! - $ $, . (* , - -)
`DELETE`	% , - ,	# ! $ $
`REFERENCES`	+ $ - (SQL CLR), , ,	# ! $ & $ $, $ $ - SELECT /

12. 3 *(окончание)*

EXECUTE	0 (SQL CLR), $ (SQL CLR),	# ! $ $ - - $
CONTROL	0 (SQL CLR), $ (SQL CLR),	# ! , - ! $; $, & & 1 . # , - CONTROL, ! ! $ - $ & & 1 . 2 CONTROL - CONTROL & & 1 / (. - 34.36)
ALTER	0 (SQL CLR), $ (SQL CLR), ,	# ! $ $ - ( ) & & - 1 . / - , ! - ALTER, CREATE DROP & & 1
TAKE OWNERSHIP	0 (SQL CLR), $ (SQL CLR), , ,	# ! $ $ $ & & 1 , -
VIEW DEFINITION	0 (SQL CLR), $ (SQL CLR), , ,	# ! $ $ & & 1 (. 34.35)
CREATE ($)	7	# ! $ $ & & 1
CREATE (DB securable)	7	# ! $ $ & & 1

Применение инструкции GRANT показано в примерах 12.11—12.17. Для начала, в примере 12.11 показано использование разрешения CREATE.

Пример 12.11. Использование разрешения CREATE

```
USE sample;
GRANT CREATE TABLE, CREATE PROCEDURE
                    TO peter, mary;
```

В примере 12.11 пользователям peter и mary дается право на выполнение инструкций языка Transact-SQL CREATE TABLE и CREATE PROCEDURE. (Как можно видеть в этом примере, инструкция GRANT для разрешения CREATE не включает параметр ON.)

В примере 12.12 пользователю mary предоставляется возможность для создания определяемых пользователем функций в базе данных sample.

Пример 12.12. Предоставление разрешения для создания определяемых пользователем функций

```
USE   sample;
GRANT CREATE FUNCTION
             TO mary;
```

В примере 12.13 показано использование разрешения SELECT в инструкции GRANT.

Пример 12.13. Применение разрешения SELECT в инструкции GRANT

```
USE sample;
GRANT SELECT ON employee
          TO peter, mary;
```

В примере 12.13 пользователи peter и mary получают разрешение на чтение строк из таблицы employee.

ПРИМЕЧАНИЕ

Когда разрешение дается учетной записи пользователя Windows или регистрационному имени, это разрешение распространяется только на данную учетную запись (регистрационное имя). С другой стороны, разрешение, предоставленное группе или роли, распространяется на всех пользователей данной группы или роли.

В примере 12.14 показано использование разрешения UPDATE в инструкции GRANT.

Пример 12.14. Применение разрешения UPDATE в инструкции GRANT

```
USE sample;
GRANT UPDATE ON works_on (emp_no, enter_date)
                            TO peter;
```

После выполнения инструкции GRANT в примере 12.14 пользователь peter может модифицировать значения столбцов emp_no и enter_date таблицы works_on.

В примере 12.15 показано использование разрешения VIEW DEFINITION, которое предоставляет пользователям доступ для чтения метаданных.

Пример 12.15. Предоставление доступа для чтения метаданных

```
USE sample;
GRANT VIEW DEFINITION ON OBJECT::employee TO peter;
GRANT VIEW DEFINITION ON SCHEMA::dbo TO peter;
```

В примере 12.15 показаны две инструкции для разрешений VIEW DEFINITION. Первая из них предоставляет пользователю peter разрешение на просмотр метаданных таблицы employee базы данных sample. (В предложении ON OBJECT указывается защищаемый объект базы данных. Посредством этого предложения можно предоставлять разрешения для работы с конкретными объектами, такими как таблицы, представления и хранимые процедуры.) Благодаря иерархической структуре защищаемых объектов, защищаемый объект более высокого уровня можно использовать, чтобы расширить область действия разрешения VEIW DEFINITION (или любого другого базового разрешения). Вторая инструкция GRANT PERMISSION в примере 12.15 предоставляет пользователю peter доступ к метаданным всех объектов схемы dbo базы данных sample.

ПРИМЕЧАНИЕ

В версиях до SQL Server 2005 было можно запрашивать информацию по всем объектам базы данных, даже если этими объектами владел другой пользователь. В последующих версиях разрешение VIEW DEFINTION позволяет предоставлять или запрещать доступ к разным частям метаданных, решая, таким образом, какую часть метаданных разрешать для просмотра другим пользователям.

В примере 12.15 показано использование разрешения CONTROL.

Пример 12.16. Применение разрешения CONTROL

```
USE sample;
GRANT CONTROL ON DATABASE::sample TO peter;
```

В примере 12.16 пользователю peter предоставляются, по сути, все определенные права доступа к защищаемому объекту (в данном случае к базе данных sample). Принципалу, которому предоставлено разрешение CONTROL для защищаемого объекта, также неявно предоставляется возможность самому предоставлять разрешения для данного объекта. Иными словами, разрешение CONTROL включает в себя предложение WITH GRANT OPTION (см. пример 12.17). Разрешение CONTROL является самым высшим разрешением, применительно ко многим защищаемым объектам базы данных. Вследствие этого, разрешение CONTROL на уровне определенной области неявно включает разрешение CONTROL для всех защищаемых объектов в этой области. Поэтому разрешение CONTROL пользователя peter для базы данных sample неявно включает в себя все разрешения к этой базе данных, а также все разрешения ко всем сборкам этой базы данных, все разрешения ко всем схемам и объектам базы данных.

По умолчанию, если пользователь *A* предоставляет разрешение пользователю *B*, то пользователь *B* может использовать разрешения при выполнении инструкций языка Transact-SQL в инструкции GRANT. Предложение WITH GRANT OPTION дает пользователю *B* дополнительную возможность предоставлять данное разрешение другим пользователям (пример 12.17).

Пример 12.17. Инструкция GRANT с предложением WITH GRANT OPTION

```
USE sample;
GRANT SELECT ON works_on TO mary
   WITH GRANT OPTION;
```

В примере 12.17 пользователю mary предоставляется разрешение выполнять инструкцию SELECT для выборки строк из таблицы works_on, а также право самому предоставлять это разрешение другим пользователям базы данных sample.

Инструкция DENY

Инструкция DENY запрещает пользователю выполнять указанные действия на указанных объектах. Иными словами, эта инструкция удаляет существующие разрешения для учетной записи пользователя, а также предотвращает получение разрешений пользователем посредством его членства в группе или роли, которое он может получить в будущем. Эта инструкция имеет следующий синтаксис:

```
DENY {ALL [PRIVILEGES]} | permission_list
   [ON [class::] securable] TO principal_list
   [CASCADE] [AS principal]
```

Все параметры инструкции DENY имеют точно такое же логическое значение, как и одноименные параметры инструкции GRANT. Инструкция DENY имеет дополнительный параметр CASCADE, в котором указывается, что разрешения, запрещенные пользователю *A*, будут также запрещены пользователям, которым он их предоставил. Если в инструкции DENY параметр CASCADE опущен, и при этом ранее были предоставлены разрешения для соответствующего объекта с использованием предложения WITH GRANT OPTION, исполнение инструкции DENY завершается ошибкой.

Инструкция DENY блокирует разрешения, полученные пользователем, группой или ролью посредством их членства в группе или роли. Это означает, что если член группы, которому запрещено разрешение, предоставленное для группы, то этот пользователь будет единственным из группы, кто не сможет использовать это разрешение. С другой стороны, если разрешение запрещено для всей группы, все члены этой группы не смогут пользоваться этим разрешением.

ПРИМЕЧАНИЕ

Инструкцию GRANT можно рассматривать как положительную авторизацию пользователя, а инструкцию DENY — как отрицательную. Обычно инструкция DENY используется для запрещения разрешений, уже предоставленных для группы (или роли), отдельным членам этой группы.

Использование инструкции DENY показано в примерах 12.18 и 12.19.

Пример 12.18. Запрещение пользователю peter двух предоставленных ранее разрешений

```
USE sample;
DENY CREATE TABLE, CREATE PROCEDURE
      TO peter;
```

Инструкция DENY в примере 12.18 отменяет для пользователя peter ранее предоставленные ему разрешения на создание таблиц и процедур.

В примере 12.19 показана негативная авторизация для некоторых пользователей базы данных sample.

Пример 12.19. Запрещение разрешений отдельным пользователям

```
USE   sample;
GRANT SELECT ON project
   TO PUBLIC;
DENY SELECT ON project
   TO peter, mary;
```

Вначале предоставляется разрешение на выборку всех строк из таблицы project всем пользователям базы данных sample. После этого это разрешение отменяется для двух пользователей: peter и mary.

ПРИМЕЧАНИЕ

Запрещение разрешений на более высоком уровне модели безопасности компонента Database Engine аннулирует разрешения, предоставленные на более низком уровне. Например, если разрешение SELECT запрещено на уровне базы данных sample, и это разрешение предполагается для таблицы employee, в результате чего разрешение SELECT будет запрещено для таблицы employee так же, как и для всех других таблиц этой базы данных.

Инструкция REVOKE

Инструкция REVOKE удаляет предоставленное или запрещенное ранее разрешение. Эта инструкция имеет следующий синтаксис:

```
REVOKE [GRANT OPTION FOR]
   {[ALL [PRIVILEGES]] | permission_list]}
   [ON [class::] securable ]
   FROM principal_list [CASCADE] [AS principal]
```

Единственным новым параметром инструкции REVOKE является параметр GRANT OPTION FOR. Все другие параметры этой инструкции имеют точно такое же

логическое значение, как и одноименные параметры инструкций GRANT или DENY. Параметр GRANT OPTION FOR используется для отмены эффекта предложения WITH GRANT OPTION в соответствующей инструкции GRANT. Это означает, что пользователь все еще будет иметь предоставленные ранее разрешения, но больше не сможет предоставлять их другим пользователям.

ПРИМЕЧАНИЕ

Инструкция REVOKE отменяет как "позитивные" разрешения, предоставленные инструкцией GRANT, так и "негативные" разрешения, предоставленные инструкцией DENY. Таким образом, его функцией является нейтрализация указанных ранее разрешений (позитивных и негативных).

Использование инструкции REVOKE показано в примере 12.20.

Пример 12.20. Отмена разрешения на инструкцию SELECT

```
USE sample;
REVOKE SELECT ON project FROM public;
```

Инструкция REVOKE в примере 12.20 отменяет предоставленное разрешение выборки данных для роли public. При этом существующее запрещение разрешения этой инструкции для пользователей peter и mary не удаляется (как это сделано в примере 12.19), поскольку разрешения, предоставленные или запрещенные явно членам группы (или роли), не затрагиваются удалением этих разрешений (положительных или отрицательных) для данной группы.

Управление разрешениями с помощью среды Management Studio

Пользователи базы данных могут выполнять действия, на которые им были предоставлены разрешения. Разрешения на определенные действия установлены в значение G (от GRANT) в столбце state в представлении просмотра каталога sys.database_permissions. Негативная запись в таблице не дает возможность пользователям выполнять деятельность. Значение D (от DENY) в этом столбце state аннулирует разрешение, предоставленное пользователю явно или неявно посредством предоставления его роли, к которой он принадлежит. Таким образом, пользователь не может выполнять данное действие в любом случае. И последнее возможное значение R (от REVOKE) в столбце state означает, что пользователь не имеет никаких явных разрешений, но может выполнять действие, если роли, к которой он принадлежит, предоставлено соответствующее разрешение.

Для управления разрешениями с помощью среды Management Studio разверните сервер, а затем папку "Databases". Щелкните правой кнопкой мыши требуемую базу данных и в контекстном меню выберите пункт **Properties**. В открывшемся диалоговом окне свойств базы данных **Database Properties — sample** (рис. 12.6) выбе-

рите страницу **Permissions**, после чего нажмите кнопку **Search**, чтобы выбрать пользователей, которым предоставить или запретить разрешения. В открывшемся диалоговом окне нажмите кнопку **Browse**, в диалоговом окне **Browse for Objects** выберите требуемых пользователей или роли (например, `guest` и `public`) и нажмите кнопки **OK** соответствующих окон, чтобы добавить выбранных пользователей (роли).

Выбранные таким образом пользователи будут отображены в окне свойств базы данных в поле **Users or roles** (Пользователи и роли) (см. рис. 12.6).

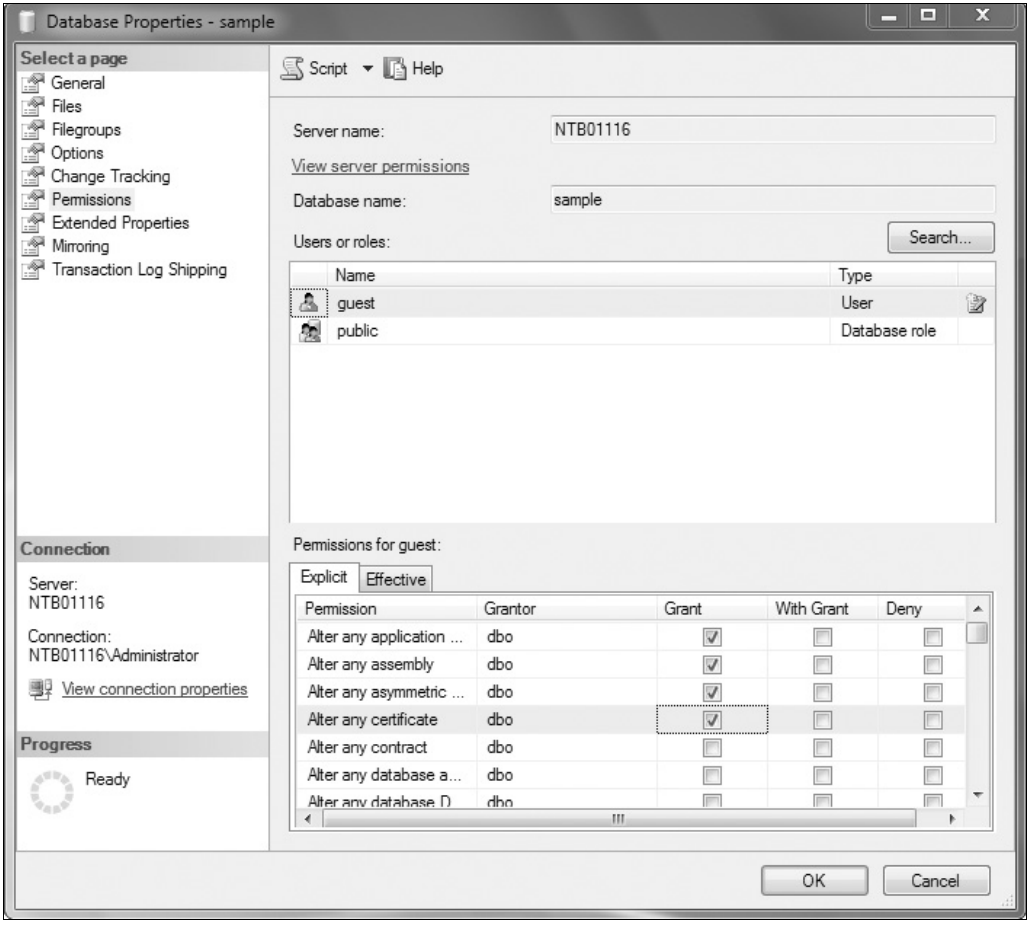


Рис. 12.6. Управление разрешениями для базы данных с помощью среды Management Studio

Теперь для выбранных пользователей можно разрешить или запретить выполнение определенных действий. В частности, для предоставления разрешения, для требуемого действия установите флажок в столбце **Grant**, а для запрещения действия установите для него флажок в столбце **Deny**. Установленный флажок в столбце **With Grant** означает, что получатель разрешения также имеет право предоставлять

его другим пользователям. Отсутствие установленных флажков в столбце разрешений или запрещений означает, что для данного пользователя нет никаких явных разрешений или запрещений на выполнение соответствующих действий.

Предоставление и запрещение разрешений пользователям или ролям для выполнения действий с индивидуальными таблицами базы данных посредством среды Management Studio осуществляется точно так же, как и для всей базы данных: В обозревателе объектов разворачивается вся иерархия папок сервера вплоть до требуемой таблицы, открывается окно свойств этой таблицы, выбираются и добавляются в поле **Users or roles** требуемые пользователи, после чего им предоставляются или запрещаются требуемые разрешения на исполнение определенных действий установкой соответствующих флажков (рис. 12.7).

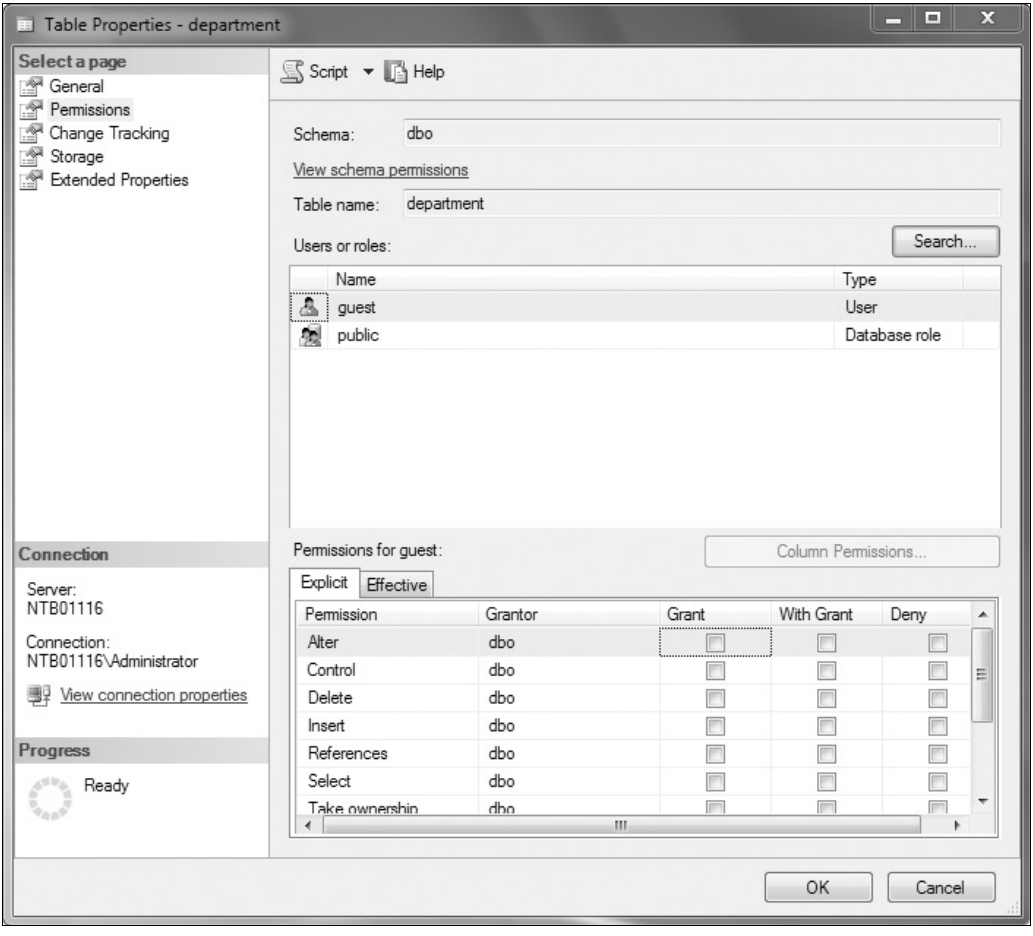


Рис. 12.7. Предоставление и запрещение разрешений для отдельной таблицы базы данных посредством среды Management Studio

Управление авторизацией и аутентификацией для автономных баз данных

Как вам уже известно из *главы 5*, автономные базы данных не имеют конфигурационных зависимостей от экземпляра сервера, на котором они были созданы, и поэтому их можно с легкостью перемещать с одного экземпляра компонента Database Engine на другой. В этом разделе мы рассмотрим, как выполнять аутентификацию пользователей для автономных баз данных. Пользователи автономной базы данных не привязаны к регистрационному имени, т. к. они не имеют внешних зависимостей и привязку для них можно выполнить где-то в другом месте.

Создание такого пользователя показано в примере 12.21.

Пример 12.21. Создание пользователя для автономной базы данных

```
USE my_sample;
CREATE USER my_login WITH PASSWORD = 'x1y2z3w4?';
```

В примере 12.21 создается пользователь `my_login`, который не привязан к какому-либо регистрационному имени. Автономная база данных `my_sample`, для которой создается этот пользователь, была создана в примере 5.20. Попытка создать автономного пользователя в обычной базе данных завершится ошибкой с выводом следующего сообщения об ошибке:

```
Msg 33233, Level 16, State 1, Line 1
You can only create a user with a password in a contained database.
```

(Сообщение 33233, уровень 16, состояние 1, строка 2
В автономной базе данных вы можете создать только пользователя с паролем).

Пользователь, сопоставленный с регистрационным именем входа в SQL Server, преобразовывается в пользователя с паролем автономной базы данных посредством хранимой системной процедуры `sp_migrate_user_to_contained`. Эта системная процедура отделяет пользователя от исходного регистрационного имени входа в SQL Server, что позволяет управлять такими параметрами, как пароль и язык по умолчанию отдельно для автономной базы данных. Данная системная хранимая процедура удаляет зависимости от конкретного экземпляра компонента Database Engine и применяется для перемещения автономной базы данных на другой экземпляр сервера.

Применение этой системной процедуры показано в примере 12.22.

Пример 12.22. Применение системной процедуры `sp_migrate_user_to_contained`

```
USE my_sample;
EXEC sp_migrate_user_to_contained
@username = 'mary_a',
@rename = N'keep_name',
@disablelogin = N'do_not_disable_login';
```

В примере 12.22 выполняется миграция регистрационного имени входа в SQL Server `mary_a` в пользователя с паролем автономной базы данных. При этом имя пользователя не меняется, а имя входа остается активированным.

ПРИМЕЧАНИЕ

Прежде чем выполнять код в примере 12.22, необходимо создать регистрационное имя входа в SQL Server `mary_a` с помощью инструкции `CREATE LOGIN` (см. пример 12.3).

Чтобы узнать, какие части базы данных можно переместить в другой экземпляр сервера, можно воспользоваться динамическим административным представлением `sys.dm_db_uncontained_entities`.

Отслеживание изменений

Понятие *отслеживание изменений* означает документирование всех операций вставок, обновлений и удалений, применяемых к таблицам базы данных. С помощью этих записей можно будет в дальнейшем определить, кто и когда обращался к данным. Отслеживание изменений можно реализовать двумя способами:

- используя триггеры;
- используя систему отслеживания измененных данных.

С помощью триггеров можно создать журнал аудита действий в таблицах базы данных. Применение триггеров рассматривается в *разд. "Триггеры AFTER"* и примере 14.1 в *главе 14*, поэтому в этом разделе мы будем рассматривать только второй подход.

Механизм отслеживания измененных данных позволяет видеть изменения в процессе их возникновения. Основной целью системы отслеживания измененных данных является создание журнала аудита, в котором фиксируется, кто и когда изменяет какие данные, но ее также можно использовать для поддержки *параллельных обновлений* (concurrency update). Суть параллельных обновлений заключается в следующем. Если приложение намеревается модифицировать строку, система отслеживания измененных данных может проверить информацию в журнале изменений, чтобы удостовериться в том, что эта строка не подвергалась изменениям после последнего ее изменения этим приложением.

ПРИМЕЧАНИЕ

Система отслеживания измененных данных предоставляется только с версиями SQL Server Enterprise и Developer.

Прежде чем для таблиц можно создавать экземпляры отслеживания, для базы данных, содержащей эти таблицы, требуется разрешить возможность отслеживания измененных данных. Это осуществляется с помощью системной хранимой про-

цедуры sys.sp_cdc_enable_db, как это показано в примере 12.23. (Исполнять эту процедуру могут только члены фиксированной роли сервера sysadmin.)

Пример 12.23. Разрешение возможности отслеживания измененных данных

```
USE sample;
EXECUTE sys.sp_cdc_enable_db
```

Определить, разрешена ли для базы данных возможность отслеживания измененных данных, можно, исследовав значение столбца is_cdc_enabled представления каталога sys.databases. Если это значение равно 1, возможность отслеживания измененных данных для данной базы данных разрешена.

Когда для базы данных разрешена возможность отслеживания измененных данных, для нее создается схема cdc, пользователь cdc, таблицы метаданных и другие системные объекты. Схема cdc содержит таблицы метаданных для системы отслеживания измененных данных, а также отдельные таблицы отслеживания, которые служат хранилищем для этой системы.

Когда для базы данных разрешена возможность отслеживания измененных данных, можно создать целевую таблицу, в которой будут сохраняться изменения для определенной исходной таблицы. Возможность отслеживания низменных данных для таблицы разрешается посредством системной хранимой процедуры sys.sp_cdc_enable_table. Применение этой системной процедуры показано в примере 12.24.

ПРИМЕЧАНИЕ

Прежде чем разрешать для таблицы возможность отслеживания измененных данных, необходимо запустить агент SQL Server.

Пример 12.24. Разрешение возможности отслеживания измененных данных для таблицы

```
USE sample;
EXECUTE sys.sp_cdc_enable_table
   @source_schema = N'dbo', @source_name = N'works_on',
   @role_name = N'cdc_admin';
```

Системная процедура sys.sp_cdc_enable_table в примере 12.24 разрешает возможность отслеживания измененных данных для указанной исходной таблицы в текущей базе данных. Когда для таблицы разрешена возможность отслеживания измененных данных, все инструкции языка DML считываются из журнала транзакций и сохраняются в соответствующей таблице изменений. В параметре @source_schema в примере 12.24 указывается имя схемы, к которой принадлежит исходная таблица, а в параметре @source_name указывается имя исходной таблицы, для которой разрешается возможность отслеживания измененных данных. Параметр @role_name задает имя роли базы данных, применяемой для разрешения доступа к данным.

Создание экземпляра отслеживания также создает таблицу отслеживания, соответствующую исходной таблице. Для исходной таблицы можно указать до двух экземпляров отслеживания. Для демонстрации работы системы отслеживания измененных данных изменим содержание исходной таблицы `works_on` базы данных `sample` с помощью примера 12.25.

Пример 12.25. Изменение содержимого исходной таблицы

```
USE sample;
INSERT INTO works_on VALUES (10102, 'p2', 'Analyst', NULL);
INSERT INTO works_on VALUES (9031, 'p2', 'Analyst', NULL);
INSERT INTO works_on VALUES (29346, 'p3', 'Clerk', NULL);
```

По умолчанию, для доступа к данным в связанной таблице изменений создается, по крайней мере, одна возвращающая табличное значение функция. Эту функцию можно использовать для запроса всех изменений, осуществляемых на протяжении заданного интервала времени. Имя функции составляется, конкатенируя префикс `cdc.fh_cdc_get_all_changes_` со значением параметра `@capture_instance`. Для данного примера значение этого параметра равно `dbo_works_on`, а запрос показан в примере 12.26.

Пример 12.26. Запрос для отображения измененных данных

```
 USE sample;
SELECT *
 FROM cdc.fn_cdc_get_all_changes_dbo_works_on
  (sys.fn_cdc_get_min_lsn('dbo_works_on'),
   sys.fn_cdc_get_max_lsn(), 'all');
```

Часть результата, выдаваемого после запроса, приведенного в примере 12.26:

__$start_lsn	__$update_mask	emp_no	project_no	job	enter_date
0x0000001C000001EF0003	0x0F	10102	p2	Analyst	NULL
0x00000010000000100003	0x0F	9031	p2	Analyst	NULL
0x00000010000000110003	0x0F	29346	p3	Clerk	NULL

Запрос в примере 12.26 отображает все изменения данных, которые имели место после выполнения трех инструкций `INSERT` в примере 12.25. Для отслеживания всех изменений в исходной таблице на протяжении определенного периода времени можно воспользоваться запросом, приведенным в примере 12.27.

Пример 12.27. Запрос для отслеживания изменений в течение периода времени

```
USE sample;
 DECLARE @from_lsn binary(10), @to_lsn binary(10);
```

```
SELECT @from_lsn =
  sys.fn_cdc_map_time_to_lsn('smallest greater than', GETDATE() - 1);
SELECT @to_lsn =
  sys.fn_cdc_map_time_to_lsn('largest less than or equal', GETDATE());
SELECT * FROM
   cdc.fn_cdc_get_all_changes_dbo_works_on(@from_lsn, @to_lsn, 'all');
```

Единственное, чем отличается запрос в примере 12.27 от запроса в примере 12.26, это тем, что в нем используется два параметра (`@from_lsn` и `@to_lsn`) для указания начала и конца интервала времени. Установка границ временного интервала осуществляется функцией `sys.fh_cdc_map_time_to_lsn()`.

Безопасность данных и представления

Как уже упоминалось в *главе 11*, представления можно использовать для следующих целей:

- для ограничения использования определенных столбцов и/или строк таблиц;
- для скрытия подробностей сложных запросов;
- для ограничения вставляемых или обновляемых значений определенными диапазонами.

Наложение ограничения на использование определенных столбцов и/или строк означает, что механизм представления обеспечивает управление доступом к данным. Например, если таблица `employee` содержит также столбец с информацией о ставке сотрудников, тогда доступ к этому столбцу можно ограничить, используя представление, которое отображает все столбцы таблицы за исключением столбца ставок. Подобным образом разрешение на такую ограниченную выборку данных из таблицы можно предоставить всем пользователям базы данных, используя данное представление, тогда как только небольшое число пользователей (имеющих повышенные права) будут иметь право на доступ ко всем данным таблицы.

В следующих трех примерах показано использование представлений для ограничения доступа к данным.

Пример 12.28. Ограничение доступа к таблице `project` ее двумя столбцами

```
USE sample;
GO
CREATE VIEW v_without_budget
   AS SELECT project_no, project_name
      FROM project;
```

Как показано в примере 12.28, представление `v_without_budget` позволяет разделить пользователей на две группы: тех, кто может просматривать бюджеты всех проектов, и тех, для которых доступ к столбцу `budget` таблицы `projects` не предоставляется.

Пример 12.29. Ограничение на доступ пользователя только к своим вставкам

```
USE sample;
GO
ALTER TABLE employee
   ADD user_name CHAR(60) DEFAULT SYSTEM_USER;
GO
CREATE VIEW v_my_rows
   AS SELECT emp_no, emp_fname, emp_lname, dept_no
      FROM employee
      WHERE user_name = SYSTEM_USER;
```

В примере 12.29 модифицируется схема таблицы `employee`, добавляя в нее новый столбец `user_name`. Теперь при каждой вставке строки в эту таблицу в столбец `user_name` вставляется регистрационное имя пользователя, осуществляющего эту вставку. Создав соответствующие представления, пользователи могут с помощью этих представлений выбирать из таблицы только те строки, которые они сами вставили.

Пример 12.30. Ограничение доступа к строкам и столбцам

```
USE sample;
GO
CREATE VIEW v_analyst
   AS SELECT employee.emp_no, emp_fname, emp_lname
      FROM employee, works_on
      WHERE employee.emp_no = works_on.emp_no
      AND job = 'Analyst';
```

Представление `v_analyst` в примере 12.30 представляет горизонтальное и вертикальное подмножество таблицы `employee`. Иными словами, оно ограничивает доступ к определенным строкам и столбцам этой таблицы.

Резюме

Наиболее важными концепциями безопасности баз данных являются следующие:

- аутентификация;
- шифрование;
- авторизация;
- отслеживание изменений.

Аутентификация является процессом проверки подлинности учетных данных пользователя, чтобы не допустить использование системы несанкционированными пользователями. Наиболее применяемым способом аутентификации является требование, чтобы для входа в систему пользователь предоставил свое имя пользова-

теля и соответствующий пароль. При шифровании данных выполняется кодирование информации таким образом, что содержащиеся данные более не являются понимаемыми до тех пор, пока получатель данных не расшифрует их. Данные можно зашифровывать несколькими разными способами.

В процессе авторизации система определяет, к каким ресурсам данный пользователь может иметь доступ. Для поддержки авторизации компонент Database Engine использует инструкции языка Transact-SQL `GRANT`, `DENY` и `REVOKE`. Отслеживание изменений означает отслеживание и документирование действий несанкционированных пользователей. Данный процесс можно использовать для защиты системы от пользователей, которые имеют расширенные права.

В следующей главе рассматриваются многопользовательские возможности компонента Database Engine и обсуждаются вопросы, связанные с оптимистическими и пессимистическими стратегиями работы в многопользовательском параллельном режиме.

Упражнения

12.1

В чем заключается разница между режимом Windows и смешанным режимом аутентификации?

12.2

Какая разница между регистрационным именем входа в SQL Server и учетной записью пользователя базы данных?

12.3

Создайте регистрационные имена `ann`, `burt` и `chuck` и их соответствующие пароли `a1b2c3d4e5!`, `d4e3f2g1h0!` и `f102gh285!`. Используйте базу данных по умолчанию `sample`. После создания регистрационных имен проверьте их наличие с помощью системного каталога.

12.4

Для регистрационных имен в упражнении 12.3 создайте соответствующие имена пользователей базы данных `s_ann`, `s_burt` и `s_charles`.

12.5

Создайте новую определяемую пользователем роль базы данных `managers` и добавьте в нее трех членов из упражнения 12.4. Отобразите информацию об этой роли и ее членах.

12.6

Используя инструкцию `GRANT`, предоставьте пользователю `s_burt` разрешение на создание таблиц, а пользователю `s_ann` — разрешение на создание хранимых процедур в базе данных `sample`.

12.7

Используя инструкцию GRANT, предоставьте пользователю s_charles разрешение на обновление столбцов emp_fname и emp_lname таблицы employee.

12.8

Используя инструкцию GRANT, предоставьте пользователям s_burt и s_ann разрешение на считывание значений столбцов emp_lname и emp_fname таблицы employee. Подсказка: создайте сначала соответствующее представление.

12.9

Используя инструкцию GRANT, предоставьте определенной пользователем роли managers разрешение на вставку строк в таблицу project.

12.10

Удалите для пользователя s_burt предоставленное ему ранее разрешение SELECT.

12.11

Используя инструкцию языка Transact-SQL, запретите пользователю s_ann вставку строк в таблицу project, как явно, так и неявно (используя разрешения роли).

12.12

Изложите разницу между использованием представлений и инструкций языка Transact-SQL GRANT, DENY и REVOKE, применительно к безопасности.

12.13

Отобразите существующую информацию о пользователе s_ann касательно базы данных sample. Подсказка: используйте системную процедуру sp_helpuser.

13

-
-
-
-
-

,
( ). ,
-
, -
(concurrency). ,
-
, ! -
.
" , -
, $
, # . % -
, , #
, # .
" $ # #
, Database Engine. &
, , -
, #. " $
#, # ACID (Atomicity,
Consistency, Isolation, Durability , , ,
), # Transact-SQL,
, #. "

(locks) ! # : -
, . -
(deadlock), # ,
.
' # , -
,
, # # -
#. -
.
"
,
Database Engine
. * : SNAPSHOT READ
COMMITED, $ # , # -
tempdb ! #.

Модели одновременного конкурентного доступа

Database Engine -
:

- # # # ;
- # # # .

" -
! , -
, . % , , -
! , -
,
+ $ (, , -
), .
. " " , -
,
(shared lock) , -
$. $, -

(exclusive lock) , # #
+ .
/
-
#. Database Engine # -
, ,
, -
#, . ' $ +
- #, ,

$, . +
#
. " , -
.

ПРИМЕЧАНИЕ

Понятие оптимистического одновременного конкурентного доступа обычно определяется в более широком смысле. Работа управления оптимистического одновременного конкурентного доступа основана на предположении маловероятности конфликтов между несколькими пользователями, поэтому разрешается исполнение транзакций без установки блокировок. Только когда пользователь пытается изменить данные, выполняется проверка ресурсов, чтобы определить наличие конфликтов. Если таковые возникли, то приложение требуется перезапустить.

Транзакции

Transact-SQL, -
0 #
,
. ! #.

- INSERT, UPDATE
DELETE .
- $ # Transact-SQL,
, BEGIN TRANSACTION,
COMMIT ROLLBACK.

, 0 . " sample -
Ann Jones # # . 1
. " ,
employee !
works_on. 2 # $,
sample ,
employee !
works_on . / $ # #
Transact-SQL 34.3.

Пример 13.1. Реализация транзакции

```
USE sample;
BEGIN TRANSACTION /* Начало транзакции */
UPDATE employee
  SET emp_no = 39831
   WHERE emp_no = 10102
   IF (@@error <> 0)
      ROLLBACK /*Откат транзакции */
```

```
UPDATE works_on
   SET emp_no = 39831
   WHERE emp_no = 10102
   IF (@@error <> 0)
      ROLLBACK
COMMIT /*Завершение транзакции */
```

, 34.3,
, UPDATE . 5 -
UPDATE
@@error. " $ # # -
-
. (% Transact-SQL BEGIN TRANSACTION, COMMIT
ROLLBACK . " Transact-SQL"
.)

ПРИМЕЧАНИЕ

Язык Transact-SQL поддерживает обработку исключений. Поэтому для обработки исключений в транзакции вместо глобальной переменной `@@error` можно использовать инструкции `TRY` и `CATCH`. Использование этих инструкций рассмотрено в 8.

" ! # # ACID. 1
,
.

Свойства транзакций

! # ,
! ACID (Atomicity, Consistency, Isolation, Durability):

- (Atomicity);
- (Consistency);
- (Isolation);
- (Durability).

#,
+ . 1 ,
, #
! #.
,
. % , -
+ -
.
.
% , + -

. 1 ,
.
# #
: . % , $++ -
. ' $ # ,
, ! -
#$ # .

Инструкции Transact-SQL и транзакции

& Transact-SQL !
#:

- BEGIN TRANSACTION;
- BEGIN DISTRIBUTED TRANSACTION;
- COMMIT [WORK];
- ROLLBACK [WORK];
- SAVE TRANSACTION;
- SET IMPLICIT_TRANSACTIONS.

% BEGIN TRANSACTION . $ #
! :

```
BEGIN TRANSACTION [{transaction_name | @trans_var}
   [WITH MARK ['description']]]
```

" *transaction_name* , -
BEGIN TRANSACTION/COMMIT BEGIN TRANSACTION/ROLLBACK. " *@trans_var*
#, ! # # -
. ' WITH MARK ,
. 6 *description* $, ! $
. " WITH MARK -
. (& + #
. 16 .)

% BEGIN DISTRIBUTED TRANSACTION -
, Microsoft Distributed Transaction Coordinator (MS DTC
Microsoft). ! -
$, -
. ' $ # -
.
, # BEGIN DISTRIBUTED TRANSACTION, $
. (/ 18 .)

% COMMIT WORK , ! -
# BEGIN TRANSACTION. 1 , #

+ . % COMMIT WORK -
# + # $ # . % WORK .

ПРИМЕЧАНИЕ

Язык Transact-SQL также поддерживает инструкцию COMMIT TRANSACTION, которая функционально равнозначна инструкции COMMIT WORK, с той разницей, что она принимает определяемое пользователем имя транзакции. Инструкция COMMIT TRANSACTION является расширением языка Transact-SQL, соответствующим стандарту SQL.

" COMMIT WORK, ROOLBACK WORK -
! . ' $ -
, ,
. "
#. % ROLLBACK WORK # + #
$ # . % WORK .

ПРИМЕЧАНИЕ

Язык Transact-SQL также поддерживает инструкцию ROLLBACK TRANSACTION, которая функционально равнозначна инструкции ROLLBACK WORK, с той разницей, что она принимает определяемое пользователем имя транзакции.

% SAVE TRANSATION .
# $ (savepoint) ,
! # -
. (& # ROLLBACK.)

ПРИМЕЧАНИЕ

Инструкция SAVE TRANSACTION в действительности не фиксирует никаких выполненных изменений данных. Она только создает метку для последующей инструкции ROLLBACK, имеющей такую же метку, как и данная инструкция SAVE TRANSACTION.

% SAVE TRANSACTION 34.7.

Пример 13.2. Создание и использование точки сохранения

```
BEGIN TRANSACTION;
INSERT INTO department (dept_no, dept_name)
   VALUES ('d4', 'Sales');
SAVE TRANSACTION a;
INSERT INTO department (dept_no, dept_name)
   VALUES ('d5', 'Research');
SAVE TRANSACTION b;
INSERT INTO department (dept_no, dept_name)
   VALUES ('d6', 'Management');
```

```
ROLLBACK TRANSACTION b;
INSERT INTO department (dept_no, dept_name)
   VALUES ('d7 ', 'Support');
ROLLBACK TRANSACTION a;
COMMIT TRANSACTION;
```

2 # #, 34.7, INSERT. & # INSERT ! ROLLBACK TRANSACTION b, # INSERT # ROLLBACK TRANSACTION a.

ПРИМЕЧАНИЕ

Инструкция SAVE TRANSACTION в сочетании с инструкцией IF или WHILE является полезной возможностью, позволяющей выполнять отдельные части всей транзакции. С другой стороны, использование этой инструкции противоречит принципу работы с базами данных, гласящему, что транзакция должна быть минимальной длины, поскольку длинные транзакции обычно уменьшают уровень доступности данных.

, Transact-SQL - . & # SQL Database Engine #. - #, - BEGIN TRANSACTION. 1 , - , . 8 + , COMMIT ROLLBACK . 2 + - , , #, .

& # SET IMPLICIT_TRANSACTIONS - ON. 1 # ! . # # , # ! # :

ALTER TABLE	FETCH	REVOKE
CREATE TABLE	GRANT	SELECT
DELETE	INSERT	TRUNCATE TABLE
DROPTABLE	OPEN	UPDATE

% , # ! , $ # .

8 # # BEGIN TRANSACTION, - # COMMIT ROLLBACK. 9 . " , # BEGIN TRANSACTION/COMMIT

BEGIN TRANSACTION/ROLLBACK # #
#. (" -
, -
.) < @@trancount
! .
% BEGIN TRANSACTION, COMMIT ROLLBACK
# . (% ROLLBACK -
# , SAVE TRANSACTION .) % -
# #
# BEGIN TRANSACTON/COMMIT BEGIN TRANSACTION/ROLLBACK.

Журнал транзакций

/ ,
. 1 #
. " # -
, ! . -
, ,
,
.
Database Engine $,
, + # , %
(transaction log). & # # -
#. , #
! # , -
Database Engine #,
, ! .
= # . 2
! , -
! ! #
($ (before image)),
#, . ' ,
-
#, (undo activity).
" # (after
image). ' $ + ,
#, -
. 1 (redo
activity) . (& -
+ # . 16 .)
# # # + ,
# % (log sequence number
LSN). " , ! # ,

, # $ #
.

Блокировка

* # # $+-
+ , ! ! +
. / ! # # , ! #
$ $++ , # . ’ U_1
! , Jim Smith -
. * ! -
, # + $,
. (" -
Jim Smith.) , $
. * $, # ,
.
" U_2 ! -
Jim Smith, # -
, $! # + .
* U_2 , $ -
Jim Smith , ,
. "
.
> -
, -

. & -
! Database Engine,
5?&, . -
, , -
+ $.
+ ,
$ #, -
.
? # :
▫ ;
▫ ;
▫ .
& $, -
. & , -
, .
/ -
! .

ПРИМЕЧАНИЕ

Последующее обсуждение относится к модели пессимистического одновременного конкурентного доступа. Модель оптимистического одновременного конкурентного доступа основана на управлении версиями строк и рассматривается в конце этой главы.

Режимы блокировки

/ . "
, # . &
!
:

- (shared, S);
- (exclusive, X);
- (update, U).

! (shared lock) ( )
. & #
, , # ,
. % ,
-
.
' (exclusive lock) -
. ? $
DML (INSERT, UPDATE DELETE), + .
@ , -
, . .
. 8 (
) # # #
.

ПРИМЕЧАНИЕ

Система баз данных автоматически выбирает соответствующий режим блокировки, в зависимости от типа операции (чтение или запись).

(update lock)
.
, $ 0 -
#. " -
0 . 2 ,
+ 0 , , 0 -
, .
5 0 .

ПРИМЕЧАНИЕ

Блокировка обновления применяется для предотвращения определенных распространенных типов взаимоблокировок. (Взаимоблокировки рассматриваются в конце этого раздела.)

" ! . 34.3.

	Разделяемая	Обновления	Монопольная
Разделяемая	Да	Да	Нет
Обновления	Да	Нет	Нет
Монопольная	Нет	Нет	Нет

34.3 ! : $\#_1$, ! # , $\#_2$, ! . B F& F # , T_2 # , F8 F, .

ПРИМЕЧАНИЕ

Компонент Database Engine также поддерживает и другие типы блокировок, такие как кратковременные блокировки (latch lock) и взаимоблокировки (spin lock). Подробную информацию об этих типах блокировок см. в .

8 ! :

- (shared, S);
- (exclusive, X);
- (intent shared, IS);
- (intent exclusive, IX);
- (shared with intent exclusive, SIX).

/ - . *

(intent lock) , ! # - ! # 0 . , ! 0 , # 0 , # $. 1 # - , , - # , .

" !
. 34.7. 1 ,
. 34.3.

! " #

	S	X	IS	SIX	IX
S	Да	Нет	Да	Нет	Нет
X	Нет	Нет	Нет	Нет	Нет
IS	Да	Нет	Да	Да	Да
SIX	Нет	Нет	Да	Нет	Нет
IX	Нет	Нет	Да	Нет	Да

S — разделяемая, X — монопольная, IS — разделяемая с намерением, SIX — монопольная с намерением, IX — разделяемая с монопольным намерением блокировки.

Гранулярность блокировки

< , # # -
. Database Engine ! -
:

- ;
- ;
- # #;
- ;
- $;
- .

ПРИМЕЧАНИЕ

Система выбирает требуемую гранулярность блокировки автоматически.

, # . ? -
, $.
? , , #
! . ' $ # -
. Database Engine
, # ! -
.

ПРИМЕЧАНИЕ

В кластеризованных таблицах страницы данных хранятся на уровне листьев (кластеризованной) индексной структуры, и поэтому для них вместо блокировки строк применяется блокировка с индексными ключами.

? ,
GH # (. 15). 1 -
, , -
.
< # #
. " ! , , ! -
. 1 ,
, . .
,
. C # , # -
, # # -
. ?
, .

Укрупнение блокировок

2 -
, Database Engine 0 $
. 1
, -
(lock escalation). (,
. ' -
. (" !
K000 .)
* # #, ! # , -
, , ! , -
, $ #, ! -
. @ +
! ALTER TABLE. 1 TABLE
! # :

```
SET (LOCK_ESCALATION = {TABLE | AUTO | DISABLE})
```

' TABLE -
. ' AUTO Database Engine -
, # . 8 -
, DISABLE .
(" Database Engine
, .)

" 34.4 employee.

Пример 13.3. Отмена возможности укрупнения блокировок для таблицы

```
USE sample;
ALTER TABLE employee SET (LOCK_ESCALATION = DISABLE);
```

Настройка блокировок

8 # ! ,
(locking hints) LOCK_TIMEOUT SET. 1 -
! .

Подсказки блокировок

' , #
Database Engine . '
' ,
. (' ! #
.)
" FROM SELECT.
& :

- UPDLOCK #
. "
;
- TABLOCK (TABLOCKX) ( ) -
. " ;
- ROWLOCK ! ! -
! # -
;
- PAGLOCK # -
, ! # ;
- NOLOCK READUNCOMMITTED (. -
$ #);
- HOLDLOCK REPEATABLEREAD (. -
$ #);
- XLOCK , -
. 2 XLOCK # ROWLOCK,
PAGLOCK TABLOCK, -
! ;
- READPAST , Database Engine
, .

ПРИМЕЧАНИЕ

Все эти параметры можно объединять вместе в любом имеющем смысл порядке. Например, комбинация подсказок TABLOCK с PAGLOCK не имеет смысла, поскольку каждая из них применяется для разных ресурсов.

Параметр *LOCK_TIMEOUT*

> 0 , SET LOCK_TIMEOUT. 1 - , 0 . 8 , - , $! :

```
SET LOCK_TIMEOUT 8000
```

2 # $, # ! ! - .

В LOCK_TIMEOUT -1 ( ) , . . - . (' READ PAST LOCK_TIMEOUT.)

Отображение информации о блокировках

8 + sys.dm_tran_locks. 1 - ! + ! - . # ! # - , # . : . < , , . 8 - $! :

- resource_type ;
- resource_database_id + , # - ;
- request_mode ;
- request_status ! .

" 34.Н , ! # sys.dm_tran_locks .

Пример 13.4. Отображение состояния блокировок посредством представления `sys.dm_tran_locks`

```
USE AdventureWorks2012;
SELECT resource_type, DB_NAME(resource_database_id) as db_name,
   request_session_id, request_mode, request_status
   FROM sys.dm_tran_locks
   WHERE request_status = 'WAIT;'
```

Взаимоблокировки

) (deadlock) $ -
, # . " ,
0 , -
, . (" ! , -
, #.) " 34.K
.

ПРИМЕЧАНИЕ

При использовании базы данных небольшого размера, одновременный конкурентный доступ процессов нельзя получить естественным образом, вследствие очень быстрого выполнения каждой транзакции. Поэтому в примере 13.5 используется инструкция `WAITFOR`, чтобы приостановить обе транзакции на десять секунд, чтобы эмулировать взаимоблокировку.

Пример 13.5. Взаимоблокировка двух процессов

```
USE   sample;
BEGIN TRANSACTION
UPDATE works_on
   SET job = 'Manager'
   WHERE emp_no = 18316
   AND project_no = 'p2'
WAITFOR DELAY '00:00:10'
UPDATE employee
   SET emp_lname = 'Green'
   WHERE emp_no = 9031
COMMIT
```

```
BEGIN TRANSACTION
UPDATE employee
   SET dept_no = 'd.2'
   WHERE emp_no = 9031
WAITFOR DELAY '00:00:10'
DELETE FROM works_on
   WHERE emp_no = 18316
   AND project_no = 'p2'
COMMIT
```

2 34.K ,
! !
:

```
Msg 1205, Level 13, State 45
Transaction (Process id 56) was deadlocked with another process and has been
chosen as deadlock victim. Rerun your command.
```

(*Сообщение 1205, уровень 13, состояние 45*

Транзакция (процесс с идентификатором 56) находится во взаимной блокировке с другим процессом и выбрана в качестве потерпевшей взаимоблокировки. Повторите выполнение команды.)

34.K,
, # (,
,) F F
. ' $. 8 -
-
,
(370K), , # .
" , #
F F , SET DEADLOCK_
PRIORITY 73 (-10 10) # .
LOW -5, NORMAL ( ) -
0, HIGH 5. F F -
.

Уровни изоляции

, . 8
, -
, -
. 2 , $ # -
, . * ! -
! #
. ' ! -
! # , , -
, , , ,
#.

Проблемы одновременного конкурентного доступа

2 , ,
, ! :

- #;
- (. " ");
- ;
- + .

,

, #. 1 ,
#

. ' $, #,
#.
'
, , #
$.
" # .
' +
, -
. 8 -
$. & -
+ .

Компонент Database Engine и уровни изоляции

% , , -
, .
Database Engine ! # , -
:

- READ UNCOMMITTED;
- READ COMMITTED;
- REPEATABLE READ;
- SERIALIZABLE;
- SNAPSHOT.

5 READ UNCOMMITTED, REPEATABLE READ SERIALIZABLE
,
SNAPSHOT #
. 5 READ COMMITTED -
. & , -
, SNAPSHOT . "*
" .

Уровень изоляции *READ UNCOMMITTED*

5 READ UNCOMMITTED + -
, ! -
#. $,
! ! .
. "
, #- #
#. 6 $ # # -
, , , -
! ! .

% ,
! , READ UNCOMMITTED :
, + .

ПРИМЕЧАНИЕ

Применение уровня изоляции READ UNCOMMITTED обычно крайне нежелательно и его следует применять только в тех случаях, когда точность данных не представляет важности или когда данные редко подвергаются изменениям.

Уровень изоляции *READ COMMITTED*

, READ COMMITTED + . ' +
,
#. " $ +
$. " + , READ COMMITTED SNAPSHOT
."* " .
, READ COMMITTED,
-
. 2 , . (1 -
.) ! -
, ,
. ' , ,
.
' $
. (* -
.) 1 # # # ,
+ ! .

ПРИМЕЧАНИЕ

Уровень изоляции READ COMMITTED для компонента Database Engine является уровнем изоляции по умолчанию.

Уровень изоляции *REPEATABLE READ*

" READ COMMITTED, REPEATABLE READ -
$ -
, . ' $ -
$ -
! . 8 $
, -
,
, , READ COMMITTED.
1
, ! , -
+ .

Уровень изоляции *SERIALIZABLE*

5 SERIALIZABLE , -
-
, . 1
, ! # #. ’ $ $
! # #
, .

ПРИМЕЧАНИЕ

Уровень изоляции SERIALIZABLE реализуется, используя метод блокировки диапазона ключа. Суть этого метода заключается в блокировке отдельных строк включительно со всем диапазоном строк между ними. Блокировка диапазона ключа блокирует элементы индексов, а не определенные страницы или всю таблицу. В этом случае любые операции модификации другой транзакцией невозможны, вследствие невозможности выполнения требуемых изменений элементов индекса.

" # , -
, ,
. , READ UNCOMMITTED
. # ,
-
#. 5 SERIALIZABLE -
, -
#.

Установка и редактирование уровня изоляции

5 , ! :

- TRANSACTION ISOLATION LEVEL SET;
- .

, TRANSACTION ISOLATION LEVEL -
#, ,
. 6 FROM SELECT ,
, ! # :

- READUNCOMMITTED;
- READCOMMITTED;
- REPEATABLEREAD;
- SERIALIZABLE.

1 ( )
. В FROM SELECT -
! , # SET TRANSACTION ISOLATION LEVEL.

% DBCC USEROPTIONS ! + !
SET, , -
! ISOLATION LEVEL.

Управление версиями строк

Database Engine #
, # . ' -
+ ,
+ #
. ' # +
tempdb # + -
.
(XSN transaction sequence number), $. (' -
XSN # + -
#.) -
! # #, #
tempdb. tempdb
, ! .
tempdb , ! , -
.
5 $++ + #, -
,
. 1 !
, $, !
. 8 -
: , ! #
SNAPSHOT, , + .
5 , , :

- READ COMMITTED SNAPSHOT;
- SNAPSHOT;
- inserted deleted.

5 SNAPSHOT READ COMMITTED SNAPSHOT -
! $ # , inserted deleted 14 .

Уровень изоляции *READ COMMITTED SNAPSHOT*

5 READ COMMITTED SNAPSHOT -
READ COMMITTED, ! . 1
, , + -
, ! $ #
. & #$!

+ -
#. / , , -
.
* ! READ COMMITTED SNAPSHOT ,
#, -
. 8 # , , -
-
.
5 READ COMMITTED SNAPSHOT -
SET ALTER DATABASE. ' $ -
. P -
, # READ COMMITTED, -
READ COMMITTED SNAPSHOT.

Уровень изоляции *SNAPSHOT*

5 SNAPSHOT #,
,
, !
$. $,
SNAPSHOT ! -
, $ # -
#. ' $ +
SNAPSHOT.
SNAPSHOT -
, -
#. " ,
. -
, SNAPSHOT , $
$ # #
, ! ! . 2 -
#, -
+ #
SNAPSHOT . 1 + #
, $ + + -
! .
/ SNAPSHOT ! .
ALLOW_SNAPSHOT_ISOLATION
($, , Management Studio). ' $
, # $,
SET TRANSACTION ISOLATION LEVEL SNAPSHOT.
$, ,
.

Разница между уровнями изоляции *READ COMMITTED SNAPSHOT* и *SNAPSHOT*

$
, SNAPSHOT + #,

. " $, READ
COMMITED SNAPSHOT # # # -
XSN # . #
XSN, -
$, $ -
.
& , READ COMMITTED SNAPSHOT
, -
. 1 + , -
#
! # # $ # -
! # #. (& -
SNAPSHOT -
+ ! ! .)

Резюме

" # # # -
$++ , ! -
! + . ’ 5?&,
Database Engine $! #. -
$ # Transact-
SQL. " . 1 -
, ,
.
& # . R
0 -
. ? ! : ,
. "
,
. & -
, .
Database Engine , # , -
! . 1
! # .

Упражнения

Упражнение 13.1

#V

Упражнение 13.2

" # # #V

Упражнение 13.3

" V

Упражнение 13.4

#V

Упражнение 13.5

, #
Transact-SQLV

Упражнение 13.6

" SAVE TRANSACTIONV

Упражнение 13.7

" # #
V

Упражнение 13.8

@ #V

Упражнение 13.9

" (#
#) # V

Упражнение 3.10

> V

Упражнение 13.11

% READ UNCOMMITTED SERIALIZABLE.

Упражнение 13.12

> V

Упражнение 13.13

F F V @
$ V

Глава 14

Триггеры

- **Введение**
- **Области применения триггеров DML**
- **Триггеры DDL и области их применения**
- **Триггеры и среда CLR**

Эта глава посвящена рассмотрению механизма, который называется *триггером*. В начале главы описываются инструкции Transact-SQL для создания, удаления и изменения триггеров, после чего приводятся примеры разных областей применения триггеров DML. Эти примеры создаются посредством одной из трех инструкций: `INSERT`, `UPDATE` или `DELETE`. Во второй части главы обсуждаются триггеры DDL, которые создаются инструкциями DDL, такими как `CREATE TABLE`. Аналогично после обсуждения триггеров DDL следуют примеры их использования в разных областях применения. В конце главы рассматривается реализация триггеров с использованием общеязыковой исполняющей среды CLR (Common Language Runtime).

Введение

Триггер — это механизм, который вызывается, когда в указанной таблице происходит определенное действие. Каждый триггер имеет следующие основные составляющие:

- имя;
- действие;
- исполнение.

Имя триггера может содержать максимум 128 символов. Действием триггера может быть или инструкция DML (`INSERT`, `UPDATE` или `DELETE`), или инструкция DDL. Таким образом, существует два типа триггеров: триггеры DML и триггеры DDL. Исполни-

тельная составляющая триггера обычно состоит из хранимой процедуры или пакета.

ПРИМЕЧАНИЕ

Компонент Database Engine позволяет создавать триггеры, используя или язык Transact-SQL, или один из языков среды CLR, такой как C# или Visual Basic. В этом разделе описывается создание триггеров посредством языка Transact-SQL. Создание триггеров, используя языки программирования среды CLR, рассматривается в конце этой главы.

Создание триггера DML

Триггер создается с помощью инструкции `CREATE TRIGGER`, которая имеет следующий синтаксис:

```
CREATE TRIGGER [schema_name.] trigger_name
   ON {table_name | view_name}
      [WITH dml_trigger_option [,...]]
   {FOR | AFTER | INSTEAD OF} {[INSERT] [,] [UPDATE] [,] [DELETE]}
   [WITH APPEND] {AS sql_statement | EXTERNAL NAME method_name}
```

ПРИМЕЧАНИЕ

Предшествующий синтаксис относится только к триггерам DML. Триггеры DDL имеют несколько иную форму синтаксиса, которая показана далее в этой главе.

Здесь в параметре *schema_name* указывается имя схемы, к которой принадлежит триггер, а в параметре *trigger_name* — имя триггера. В параметре *table_name* задается имя таблицы, для которой создается триггер. (Также поддерживаются триггеры для представлений, на что указывает наличие параметра *view_name*.)

Также можно задать тип триггера с помощью двух дополнительных параметров: `AFTER` и `INSTEAD OF`. (Параметр `FOR` является синонимом параметра `AFTER`.) Триггеры типа `AFTER` вызываются после выполнения действия, запускающего триггер, а триггеры типа `INSTEAD OF` выполняются вместо действия, запускающего триггер. Триггеры `AFTER` можно создавать только для таблиц, а триггеры `INSTEAD` — как для таблиц, так и для представлений. Примеры для этих обоих типов триггеров приводятся далее в этой главе.

Параметры `INSERT`, `UPDATE` и `DELETE` задают действие триггера. Под действием триггера имеется в виду инструкция Transact-SQL, которая запускает триггер. Допускается любая комбинация этих трех инструкций. Инструкция `DELETE` не разрешается, если используется параметр `IF UPDATE`.

Как можно видеть в синтаксисе инструкции `CREATE TRIGGER`, действие (или действия) триггера указывается в спецификации `AS sql_statement`. (Также можно использовать параметр `EXTERNAL NAME`, который объясняется далее в этой главе.)

ПРИМЕЧАНИЕ

Компонент Database Engine позволяет создавать несколько триггеров для каждой таблицы и для каждого действия (INSERT, UPDATE и DELETE). По умолчанию определенного порядка исполнения нескольких триггеров для данного модифицирующего действия не имеется. (Порядок выполнения можно задать, используя первый и последний триггеры, как описывается далее в этой главе.)

Только владелец базы данных, администраторы DDL и владелец таблицы, для которой определяется триггер, имеют право создавать триггеры для текущей базы данных. (В отличие от разрешений для других типов инструкции CREATE это разрешение не может передаваться.)

Изменение структуры триггера

Язык Transact-SQL также поддерживает инструкцию ALTER FUNCTION, которая модифицирует структуру триггера. Эта инструкция обычно применяется для изменения тела триггера. Все предложения и параметры инструкции ALTER TRIGGER имеют такое же значение, как и одноименные предложения и параметры инструкции CREATE TRIGGER.

Для удаления триггеров в текущей базе данных применяется инструкция DROP TRIGGER.

В следующем разделе рассматриваются таблицы deleted и inserted, которые играют важную роль в действиях триггеров.

Использование виртуальных таблиц *deleted* и *inserted*

При создании действия триггера обычно требуется указать, ссылается ли он на значение столбца до или после его изменения действием, запускающим триггер. По этой причине, для тестирования следствия инструкции, запускающей триггер, используются две специально именованные виртуальные таблицы:

- deleted — содержит копии строк, удаленных из таблицы;
- inserted — содержит копии строк, вставленных в таблицу.

Структура этих таблиц эквивалентна структуре таблицы, для которой определен триггер.

Таблица deleted используется в том случае, если в инструкции CREATE TRIGGER указывается предложение DELETE или UPDATE, а если в этой инструкции указывается предложение INSERT или UPDATE, то используется таблица inserted. Это означает, что для каждой инструкции DELETE, выполненной в действии триггера, создается таблица deleted. Подобным образом для каждой инструкции INSERT, выполненной в действии триггера, создается таблица inserted.

Инструкция UPDATE рассматривается, как инструкция DELETE, за которой следует инструкция INSERT. Поэтому для каждой инструкции UPDATE, выполненной в действии триггера, создается как таблица deleted, так и таблица inserted (в указанной последовательности).

Таблицы inserted и deleted реализуются, используя управление версиями строк, которое рассматривается в *главе 13*. Когда для таблицы с соответствующими триггерами выполняется инструкция DML INSERT, UPDATE или DELETED, для всех изменений в этой таблице всегда создаются версии строк. Когда триггеру требуется информация из таблицы deleted, он обращается к данным в хранилище версий строк. В случае таблицы inserted, триггер обращается к самым последним версиям строк.

ПРИМЕЧАНИЕ

В качестве хранилища версий строк механизм управления версиями строк использует системную базу данных tempdb. По этой причине, если база данных содержит большое число часто используемых триггеров, следует ожидать значительного увеличения объема этой системной базы данных.

Области применения DML-триггеров

В первой части этой главы мы узнали, как создавать триггеры DML и изменять их структуру. Такие триггеры применяются для решения разнообразных задач. В этом разделе мы рассмотрим несколько областей применения триггеров DML, в частности триггеров AFTER и INSTEAD OF.

Триггеры *AFTER*

Как вы уже знаете, триггеры AFTER вызываются после того, как выполняется действие, запускающее триггер. Триггер AFTER задается с помощью ключевого слова AFTER или FOR. Триггеры AFTER можно создавать только для базовых таблиц.

Триггеры этого типа можно использовать для выполнения, среди прочих, следующих операций:

- создания журнала аудита действий в таблицах базы данных (см. пример 14.1);
- реализации бизнес-правил (см. пример 14.2);
- принудительного обеспечения ссылочной целостности (см. примеры 14.3 и 14.4).

Создание журнала аудита

В *главе 12* мы рассмотрели, как выполнять отслеживание изменения данных, используя систему *перехвата изменения данных* CDC (change data capture). Эту задачу можно также решить с помощью триггеров DML. В примере 14.1 показывается, как с помощью триггеров можно создать журнал аудита действий в таблицах базы данных.

Пример 14.1. Создание журнала аудита действий в таблицах базы данных

```
/* Таблица audit_budget используется в качестве журнала аудита действий
в таблице project */
USE sample;
GO
CREATE TABLE audit_budget (project_no CHAR(4) NULL,
   user_name CHAR(16) NULL,
   date DATETIME NULL,
   budget_old FLOAT NULL,
   budget_new FLOAT NULL);
GO
CREATE TRIGGER modify_budget
   ON project AFTER UPDATE
   AS IF UPDATE(budget)
   BEGIN
   DECLARE @budget_old FLOAT
   DECLARE @budget_new FLOAT
   DECLARE @project_number CHAR(4)
   SELECT @budget_old = (SELECT budget FROM deleted)
   SELECT @budget_new = (SELECT budget FROM inserted)
   SELECT @project_number = (SELECT project_no FROM deleted)
   INSERT INTO audit_budget VALUES
   (@project_number,USER_NAME(), GETDATE(), @budget_old, @budget_new)
   END
```

В примере 14.1 создается таблица `audit_budget`, в которой сохраняются все изменения столбца `budget` таблицы `project`. Изменения этого столбца будут записываться в эту таблицу посредством триггера `modify_budget`.

Этот триггер активируется для каждого изменения столбца `budget` с помощью инструкции `UPDATE`. При выполнении этого триггера значения строк таблиц `deleted` и `inserted` присваиваются соответствующим переменным `@budget_old`, `@budget_new` и `@project_ number`. Эти присвоенные значения, совместно с именем пользователя и текущей датой, будут затем вставлены в таблицу `audit_budget`.

ПРИМЕЧАНИЕ

В примере 14.1 предполагается, что за один раз будет обновление только одной строки. Поэтому этот пример является упрощением общего случая, когда триггер обрабатывает многострочные обновления. Реализация такого общего (и сложного) триггера находится вне пределов вводного уровня этой книги.

Если выполнить следующие инструкции Transact-SQL:

```
UPDATE project
   SET budget = 200000
   WHERE project_no = 'p2';
```

то содержимое таблицы audit_budget будет таким:

project_no	user_name	Date	budget_old	budget_new
p2	Dbo	2011-01-31 14:00:05	95000	200000

Реализация бизнес-правил

С помощью триггеров можно создавать бизнес-правила для приложений. Создание такого триггера показано в примере 14.2.

Пример 14.2. Создание триггера для бизнес-правила

```
-- Триггер total_budget является примером использования
-- триггера для реализации бизнес-правила
USE sample;
   GO
   CREATE TRIGGER total_budget
      ON project AFTER UPDATE
      AS IF UPDATE (budget)
         BEGIN
         DECLARE @sum_old1
         FLOAT DECLARE @sum_old2
         FLOAT DECLARE @sum_new FLOAT
         SELECT @sum_new = (SELECT SUM(budget) FROM inserted)
         SELECT @sum_old1 = (SELECT SUM(p.budget)
                     FROM project p WHERE p.project_no
              NOT IN (SELECT d.project_no FROM deleted d))
         SELECT @sum_old2 = (SELECT SUM(budget) FROM deleted)
         IF @sum_new > (@sum_old1 + @sum_old2)*1.5
         BEGIN
            PRINT 'No modification of budgets'
            ROLLBACK TRANSACTION
            END
      ELSE
            PRINT 'The modification of budgets executed'
         END
```

В примере 14.2 создается правило для управления модификацией бюджетов проектов. Триггер total_budget проверяет каждое изменение бюджетов и выполняет только такие инструкции UPDATE, которые увеличивают сумму всех бюджетов не более чем на 50%. В противном случае для инструкции UPDATE выполняется откат посредством инструкции ROLLBACK TRANSACTION.

Принудительное обеспечение ограничений целостности

Как упоминалось в *главе 5*, в системах управления базами данных применяются два типа ограничений для обеспечения целостности данных:

- декларативные ограничения, которые определяются с помощью инструкций языка `CREATE TABLE` и `ALTER TABLE`;
- процедурные ограничения целостности, которые реализуются посредством триггеров.

В обычных ситуациях следует использовать декларативные ограничения для обеспечения целостности, поскольку они поддерживаются системой и не требуют реализации пользователем. Применение триггеров рекомендуется только в тех случаях, для которых декларативные ограничения для обеспечения целостности отсутствуют.

В примере 14.3 показано принудительное обеспечение ссылочной целостности посредством триггеров для таблиц `employee` и `works_on`.

Пример 14.3. Обеспечение ссылочной целостности посредством триггера

```
USE sample;
GO
CREATE TRIGGER workson_integrity
   ON works_on AFTER INSERT, UPDATE
   AS IF UPDATE(emp_no)
      BEGIN
      IF (SELECT employee.emp_no
            FROM employee, inserted
            WHERE employee.emp_no = inserted.emp_no) IS NULL
         BEGIN
         ROLLBACK TRANSACTION
         PRINT 'No insertion/modification of the row'
         END
      ELSE PRINT 'The row inserted/modified'
      END
```

Триггер `workson_integrity` в примере 14.3 проверяет ссылочную целостность для таблиц `employee` и `works_on`. Это означает, что проверяется каждое изменение столбца `emp_no` в ссылочной таблице `works_on`, и при любом нарушении этого ограничения выполнение этой операции не допускается. (То же самое относится и к вставке в столбец `emp_no` новых значений.) Инструкция `ROLLBACK TRANSACTION` во втором блоке `BEGIN` выполняет откат инструкции `INSERT` или `UPDATE` в случае нарушения ограничения для обеспечения ссылочной целостности.

В примере 14.3 триггер выполняет проверку на проблемы ссылочной целостности первого и второго случая (см. *разд. "Возможные проблемы со ссылочной целостностью" главы 5*) между таблицами `employee` и `works_on`. А в примере 14.4 показан

триггер, который выполняет проверку на проблемы ссылочной целостности третьего и четвертого случая между этими же таблицами.

Пример 14.4. Обеспечение ссылочной целостности, случаи 3 и 4

```
USE sample;
GO
CREATE TRIGGER refint_workson2
   ON employee AFTER DELETE, UPDATE
   AS IF UPDATE (emp_no)
   BEGIN
      IF (SELECT COUNT(*)
         FROM WORKS_ON, deleted
         WHERE works_on.emp_no = deleted.emp_no) > 0
      BEGIN
      ROLLBACK TRANSACTION
      PRINT 'No modification/deletion of the row'
      END
      ELSE PRINT 'The row is deleted/modified'
   END
```

Триггеры *INSTEAD OF*

Триггер с предложением `INSTEAD OF` заменяет соответствующее действие, которое запустило его. Этот триггер выполняется после создания соответствующих таблиц `inserted` и `deleted`, но перед выполнением проверки ограничений целостности или каких-либо других действий.

Триггеры `INSTEAD OF` можно создавать как для таблиц, так и для представлений. Когда инструкция Transact-SQL ссылается на представление, для которого определен триггер `INSTEAD OF`, система баз данных выполняет этот триггер вместо выполнения любых действий с любой таблицей. Данный тип триггера всегда использует информацию в таблицах `inserted` и `deleted`, созданных для представления, чтобы создать любые инструкции, требуемые для создания запрошенного события.

Значения столбцов, предоставляемые триггером `INSTEAD OF`, должны удовлетворять определенным требованиям:

- значения не могут задаваться для вычисляемых столбцов;
- значения не могут задаваться для столбцов с типом данных `TIMESTAMP`;
- значения не могут задаваться для столбцов со свойством `IDENTITY`, если только параметру `IDENTITY_INSERT` не присвоено значение `ON`.

Эти требования действительны только для инструкций `INSERT` и `UPDATE`, которые ссылаются на базовые таблицы. Инструкция `INSERT`, которая ссылается на представления с триггером `INSTEAD OF`, должна предоставлять значения для всех столбцов этого представления, не допускающих пустые значения `NULL`. (То же самое от-

носится и к инструкции UPDATE. Инструкция UPDATE, ссылающаяся на представление с триггером INSTEAD OF, должна предоставить значения для всех столбцов представления, которое не допускает пустых значений и на которое осуществляется ссылка в предложении SET.)

В примере 14.5 показана разница в поведении при вставке значений в вычисляемые столбцы, используя таблицу и ее соответствующее представление.

Пример 14.5. Вставка значений в вычисляемые столбцы

```
CREATE VIEW all_orders
   AS SELECT orderid, price, quantity, orderdate, total, shippeddate
      FROM orders;
GO
CREATE TRIGGER tr_orders
   ON all_orders INSTEAD OF INSERT
   AS BEGIN
      INSERT INTO orders
         SELECT orderid, price, quantity, orderdate
            FROM inserted
      END
```

В примере 14.5 используется таблица orders из примера 10.8, содержащая два вычисляемых столбца. Представление all_orders содержит все строки этой таблицы. Это представление используется для задания значения в его столбце, которое соотносится с вычисляемым столбцом в базовой таблице, на которой создано представление. Это позволяет использовать триггер INSTEAD OF, который в случае инструкции INSERT заменяется пакетом, который вставляет значения в базовую таблицу посредством представления all_orders. (Инструкция INSERT, обращающаяся непосредственно к базовой таблице, не может задавать значение вычисляемому столбцу.)

Триггеры *first* и *last*

Компонент Database Engine позволяет создавать несколько триггеров для каждой таблицы или представления и для каждой операции (INSERT, UPDATE и DELETE) с ними. Кроме этого, можно указать порядок выполнения для нескольких триггеров, определенных для конкретной операции. С помощью системной процедуры sp_settriggerorder можно указать, что один из определенных для таблицы триггеров AFTER будет выполняться первым или последним для каждого обрабатываемого действия. Эта системная процедура имеет параметр @order, которому можно присвоить одно из трех значений:

- first — указывает, что триггер является первым триггером AFTER, выполняющимся для модифицирования действия;
- last — указывает, что данный триггер является последним триггером AFTER, выполняющимся для инициирования действия;

- none — указывает, что для триггера отсутствует какой-либо определенный порядок выполнения. (Это значение обычно используется для того, чтобы выполнить сброс ранее установленного порядка выполнения триггера как первого или последнего.)

ПРИМЕЧАНИЕ

Изменение структуры триггера посредством инструкции ALTER TRIGGER отменяет порядок выполнения триггера (первый или последний).

Применение системной процедуры sp_settriggerorder показано в примере 14.6.

Пример 14.6. Применение системной процедуры sp_settriggerorder

```
EXEC sp_settriggerorder @triggername = 'modify_budget',
                        @order = 'first', @stmttype='update'
```

ПРИМЕЧАНИЕ

Для таблицы разрешается определить только один первый и только один последний триггер AFTER. Остальные триггеры AFTER выполняются в неопределенном порядке.

Узнать порядок выполнения триггера можно с помощью следующих средств:

- системной процедуры sp_helptrigger;
- функции OBJECTPROPERTY.

Возвращаемый системной процедурой sp_helptrigger результирующий набор содержит столбец order, в котором указывается порядок выполнения указанного триггера. При вызове функции OBJECTPROPERTY в ее втором параметре указывается значение ExecIsFirstTrigger или ExecIsLastTrigger, а в первом параметре всегда указывается идентификационный номер объекта базы данных. Если указанное во втором параметре свойство имеет значение TRUE, функция возвращает значение 1.

ПРИМЕЧАНИЕ

Поскольку триггер INSTEAD OF исполняется перед тем, как выполняются изменения в его таблице, для триггеров этого типа нельзя указать порядок выполнения "первым" или "последним".

Триггеры DDL и области их применения

В первой части этой главы мы рассмотрели триггеры DML, которые задают действие, предпринимаемое сервером при изменении таблицы инструкциями INSERT, UPDATE или DELETE. Компонент Database Engine также позволяет определять тригге-

ры для инструкций DDL, таких как CREATE DATABASE, DROP TABLE и ALTER TABLE. Триггеры для инструкций DDL имеют следующий синтаксис:

```
CREATE TRIGGER [schema_name.]trigger_name
      ON {ALL SERVER | DATABASE}
   [WITH {ENCRYPTION | EXECUTE AS clause_name]
      {FOR | AFTER} {event_group | event_type | LOGON}
   AS {batch | EXTERNAL NAME method_name}
```

Как можно видеть по их синтаксису, триггеры DDL создаются таким же способом, как и триггеры DML. А для изменения и удаления этих триггеров используются те же инструкции ALTER TRIGGER и DROP TRIGGER, что и для триггеров DML. Поэтому в этом разделе рассматриваются только те параметры инструкции CREATE TRIGGER, которые новые для синтаксиса триггеров DDL.

Первым делом при определении триггера DDL нужно указать его область действия. Предложение DATABASE указывает в качестве области действия триггера DDL текущую базу данных, а предложение ALL SERVER — текущий сервер.

После указания области действия триггера DDL нужно в ответ на выполнение одной или нескольких инструкций DDL указать способ запуска триггера. В параметре event_type указывается инструкция DDL, выполнение которой запускает триггер, а в альтернативном параметре event_group указывается группа событий языка Transact-SQL. Триггер DDL запускается после выполнения любого события языка Transact-SQL, указанного в параметре event_group. Список всех групп и типов событий см. в *электронной документации*. Ключевое слово LOGON указывает триггер входа (см. далее пример 14.8).

Кроме сходства триггеров DML и DDL, между ними также есть несколько различий. Основным различием между этими двумя видами триггеров является то, что для триггера DDL можно задать в качестве его области действия всю базу данных или даже весь сервер, а не всего лишь отдельный объект. Кроме этого, триггеры DDL не поддерживают триггеров INSTEAD OF. Как вы, возможно, уже догадались, для триггеров DDL не требуются таблицы inserted и deleted, поскольку эти триггеры не изменяют содержимого таблиц.

В следующих подразделах подробно рассматриваются две формы триггеров DDL: триггеры уровня базы данных и триггеры уровня сервера.

Триггеры DDL уровня базы данных

В примере 14.7 показано, как можно реализовать триггер DDL, чья область действия распространяется на текущую базу данных.

Пример 14.7. Создание триггера DDL уровня базы данных

```
USE sample;
GO
CREATE TRIGGER prevent_drop_triggers
```

```
ON DATABASE FOR DROP_TRIGGER
AS PRINT 'You must disable "prevent_drop_triggers"
                                    to drop any trigger'
ROLLBACK
```

Триггер в примере 14.7 предотвращает удаление любого триггера для базы данных sample любым пользователем. Предложение DATABASE указывает, что триггер prevent_drop_database является триггером уровня базы данных. Ключевое слово DROP_TRIGGER указывает предопределенный тип события, запрещающий удаление любого триггера.

Триггеры DDL уровня сервера

Триггеры уровня сервера реагируют на серверные события. Триггер уровня сервера создается посредством использования предложения ALL SERVER в инструкции CREATE TRIGGER. В зависимости от выполняемого триггером действия, существует два разных типа триггеров уровня сервера: обычные триггеры DDL и триггеры входа. Запуск обычных триггеров DDL основан на событиях инструкций DDL, а запуск триггеров входа — на событиях входа.

В примере 14.8 демонстрируется создание триггера уровня сервера, который является триггером входа.

Пример 14.8. Создание триггера уровня сервера, являющегося триггером входа

```
USE master;
GO
CREATE LOGIN login_test WITH PASSWORD = 'login_test§$!',
   CHECK_EXPIRATION = ON;
GO
GRANT VIEW SERVER STATE TO login_test;
GO
CREATE TRIGGER connection_limit_trigger
ON ALL SERVER WITH EXECUTE AS 'login_test'
FOR LOGON AS
BEGIN IF ORIGINAL_LOGIN()= 'login_test' AND
   (SELECT COUNT(*) FROM sys.dm_exec_sessions
      WHERE is_user_process = 1 AND
         original_login_name = 'login_test') > 1
   ROLLBACK;
END;
```

В примере 14.8 сначала создается имя входа SQL Server login_test, которое потом используется в триггере уровня сервера. По этой причине, для этого имени входа требуется разрешение VIEW SERVER STATE, которое и предоставляется ему посредством инструкции GRANT. После этого создается триггер connection_limit_trigger. Этот триггер является триггером входа, что указывается ключевым словом LOGON.

С помощью представления sys.dm_exec_sessions выполняется проверка, был ли уже установлен сеанс с использованием имени входа login_test. Если сеанс уже был установлен, выполняется инструкция ROLLBACK. Таким образом имя входа login_test может одновременно установить только один сеанс.

Триггеры и среда CLR

Подобно хранимым процедурам и определяемым пользователем функциям, триггеры можно реализовать, используя общеязыковую среду выполнения (CLR — Common Language Runtime). Триггеры в среде CLR создаются в три этапа:

1. Создается исходный код триггера на языке C# или Visual Basic, который затем компилируется, используя соответствующий компилятор в объектный код (см. примеры 14.9 и 14.10).
2. Объектный код обрабатывается инструкцией CREATE ASSEMBLY, создавая соответствующий выполняемый файл (см. пример 14.11).
3. Посредством инструкции CREATE TRIGGER создается триггер (см. пример 14.12).

Выполнение всех этих трех этапов создания триггера CLR демонстрируется в последующих примерах. В примере 14.9 приводится листинг исходного кода программы на языке C# для триггера из примера 14.1.

ПРИМЕЧАНИЕ

Прежде чем создавать триггер CLR в последующих примерах, сначала нужно удалить триггер prevent_drop_trigger (см. пример 14.7), а затем удалить триггер modify_budget (см. пример 14.1), используя в обоих случаях инструкцию DROP TRIGGER.

Пример 14.9. Исходный код триггера CLR на языке C#

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
public class StoredProcedures
{
  public static void Modify_Budget()
    {
    SqlTriggerContext context = SqlContext.TriggerContext;
    if(context.IsUpdatedColumn(2)) //Бюджет
    {
      float budget_old;
      float budget_new;
      string project_number;
```

```
        SqlConnection conn = new SqlConnection("context
                                                connection=true");
        conn.Open();
        SqlCommand and = conn. CreateCommand ();
        and.CommandText = "SELECT budget FROM DELETED";
        budget_old = (float)Convert.ToDouble(cmd.ExecuteScalar());
        cmd.CommandText = "SELECT budget FROM INSERTED";
        budget_new = (float)Convert.ToDouble(cmd.ExecuteScalar());
        cmd.CommandText = "SELECT project_no FROM DELETED";
        project_number = Convert.ToString(cmd.ExecuteScalar());
        cmd.CommandText = @"INSERT INTO audit_budget
        VALUES(@project_number, USER_NAME(), GETDATE(),
        @budget_old, @budget_new)";
        cmd.Parameters.AddWithValue("@project_number",project_number);
        cmd.Parameters.AddWithValue("@budget_old",budget_old);
        cmd.Parameters.AddWithValue("@budget_new",budget_new);
        cmd.ExecuteNonQuery();
      }
    }
  }
```

Пространство имен `Microsoft.SQLServer.Server` содержит все классы клиентов, которые могут потребоваться программе C#. Классы `SqlTriggerContext` и `SqlFunction` являются членами этого пространства имен. Кроме этого, пространство имен `System.Data.SqlClient` содержит классы `SQLConnection` и `SQLCommand`, которые используются для установления соединения и взаимодействия между клиентом и сервером базы данных. Соединение устанавливается, используя строку соединения `"context connection = true"`:

```
SqlConnection conn = new SqlConnection("context connection=true");
```

Затем определяется класс `StoredProcedure`, который применяется для реализации триггеров. Метод `Modify_Budget()` реализует одноименный триггер.

Экземпляр `context` класса `SQLTriggerContext` позволяет программе получить доступ к виртуальной таблице, создаваемой при выполнении триггера. В этой таблице сохраняются данные, вызвавшие срабатывание триггера. Метод `IsUpdatedColumn()` класса `SQLTriggerContext` позволяет узнать, был ли модифицирован указанный столбец таблицы.

Данная программа содержит два других важных класса: `SQL Connection` и `SQLCommand`. Экземпляр класса `SQLConnection` обычно применяется для установления соединения с базой данных, а экземпляр класса `SQLCommand` позволяет исполнять SQL-инструкции.

Следующие инструкции используют свойство `Parameters` класса `SQLScommand` для отображения параметров, а метод `AddWithValue()` для вставки значения в указанный параметр:

```
cmd.Parameters.AddWithValue("@project_number",project_number);
cmd.Parameters.AddWithValue("@budget_old",budget_old);
cmd.Parameters.AddWithValue("@budget_new",budget_new);
```

Программу в примере 14.9 можно скомпилировать с помощью команды esc. Применение этой команды для компиляции данной программы показано в примере 14.10.

Пример 14.10. Компилирование исходного кода программы триггера

```
esc /target:library Example14_9.cs
/reference:"c:\Program Files\Microsoft
SQLServer\MSSQL11.MSSQLSERVER\MSSQL\Binn\sqlaccess.dll"
```

Подробное описание команды esc см. в *главе 8*.

ПРИМЕЧАНИЕ

Среда CLR включается и отключается посредством опции clr_enabled системной процедуры sp_configure, которая запускается на выполнение инструкцией RECONFIGURE (см. пример 8.9).

Следующий шаг в создании триггера modify_budget показан в примере 14.11. (Для выполнения этой инструкции используйте среду Management Studio.)

Пример 14.11. Создание исполняемого кода триггера

```
CREATE ASSEMBLY Example14_9 FROM
    'C:\Programs\Microsoft SQL Server\assemblies\Example14_9.dll'
     WITH PERMISSION_SET=EXTERNAL_ACCESS
```

Инструкция CREATE ASSEMBLY принимает в качестве ввода управляемый код и создает соответствующий объект, на основе которого создается триггер CLR. Предложение WITH PERMISSION SET в примере 14.11 указывает, что разрешениям доступа присвоено значение EXTERNAL_ACCESS, что не позволяет сборкам получать доступ к внешним ресурсам системы, за исключением только некоторых из них.

Наконец, в примере 14.12 посредством инструкции CREATE TRIGGER создается триггер modify_budget.

Пример 14.12. Создание триггера

```
CREATE TRIGGER modify_budget ON project
   AFTER UPDATE AS
      EXTERNAL NAME Example14_9.StoredProcedures.Modify_Budget
```

Инструкция CREATE TRIGGER в примере 14.12 отличается от такой же инструкции в примерах 14.1 и 14.5 тем, что она содержит параметр EXTERNAL NAME. Этот параметр

указывает, что код создается средой CLR. Имя в этом параметре состоит из трех частей. В первой части указывается имя соответствующей сборки (`Example14_9`), во второй — имя открытого класса, определенного в примере 14.9 (`StoredProcedures`), а в третьей указывается имя метода, определенного в этом классе (`Modify_Budget`).

В примере 14.13 показана реализация на языке C# триггера из примера 14.3.

ПРИМЕЧАНИЕ

Перед тем как создавать триггер CLR в примере 14.13, нужно сначала удалить триггер `workson_integrity` (см. пример 14.3), используя для этого инструкцию `DROP TRIGGER`.

Пример 14.13. Исходный код на языке C# триггера из примера 14.3

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;
public class StoredProcedures
{
  public static void WorksOn_Integrity()
  {
    SqlTriggerContext context = SqlContext.TriggerContext;
    if(context.IsUpdatedColumn(0)) //Emp_No
    {
      SqlConnection conn = new SqlConnection("context
                                          connection=true");
      conn.Open();
      SqlCommand cmd = conn.CreateCommand();
      cmd.CommandText = "SELECT employee.emp_no
          FROM employee, inserted
          WHERE employee.emp_no = inserted.emp_no";
      SqlPipe pipe = SqlContext.Pipe;
      if(cmd.ExecuteScalar() == null)
    {
       System.Transactions.Transaction.Current.Rollback!);
       pipe.Send("No insertion/modification of the row");
    }
    else
      pipe.Send("The row inserted/modified");
    }
  }
}
```

В примере 14.13 содержится только две новые особенности, которые требуют объяснения. Класс `SqlPipe` принадлежит к пространству имен `Microsoft.SQLServer.`

Server и позволяет отправлять сообщения вызывающему объекту, например, как следующее:

```
pipe.Send("No insertion/modification of the row");
```

Свойство Current класса Transaction используется для установки (или получения) текущей транзакции в триггере. В примере 14.13 метод Rollback() применяется для выполнения отката всей транзакции при нарушении ограничения для обеспечения целостности.

В примере 14.14 показано создание сборки и соответствующего триггера на основе исходного кода на C#, приведенного в примере 14.13. (Прежде чем выполнять этот пример, необходимо скомпилировать исходный код программы, используя команду esc, но этот шаг не показан, поскольку он полностью аналогичен выполнению этой команды в примере 14.10.)

Пример 14.14. Создание сборки и соответствующего триггера

```
CREATE ASSEMBLY Example14_13 FROM
   'C:\Programs\Microsoft SQL Server\assemblies\Example14_13.dll'
WITH PERMISSION_SET=EXTERNAL_ACCESS
GO
CREATE TRIGGER workson_integrity ON works_on
AFTER INSERT, UPDATE AS
EXTERNAL NAME Example14_13.StoredProcedures.WorksOn_Integrity
```

Резюме

Триггеры представляют собой механизм базы данных и могут быть двух типов: триггеры DML и триггеры DDL. Триггеры DML задают действие (или действия), которое автоматически исполняется сервером базы данных при изменении таблицы инструкцией INSERT, UPDATE или DELETE. (Триггеры DML нельзя использовать с инструкцией SELECT.) Триггеры DDL запускаются при выполнении инструкций DDL. В зависимости от области действия, существует две формы триггеров этого типа. Предложение DATABASE указывает в качестве области действия триггера DDL текущую базу данных, а предложение ALL SERVER — текущий сервер.

Эта заключительная глава *части II* данной книги. Следующая глава открывает *часть III*; в ней рассматривается системная среда компонента Database Engine.

Упражнения

Упражнение 14.1

Используя триггеры, задайте ссылочную целостность для первичного ключа таблицы department, столбца dept_no, который является внешним ключом таблицы works_on.

Упражнение 14.2

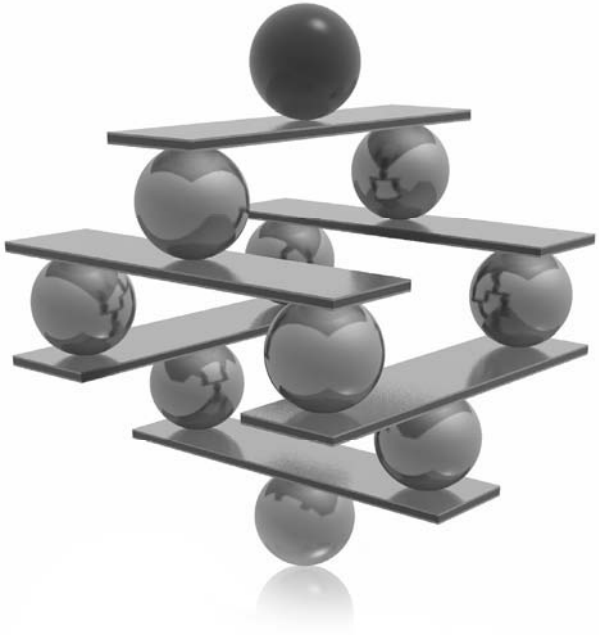
Используя триггеры, задайте ссылочную целостность для первичного ключа таблицы `project`, столбца `project_no`, который является внешним ключом таблицы `works_on`.

Упражнение 14.3

Используя среду CLR, реализуйте триггер из примера 14.4.

Часть III

SQL Server: системное администрирование



Глава 15

Системная среда компонента Database Engine

- **Системные базы данных**
- **Дисковое хранилище**
- **Утилиты и команды DBCC**
- **Управление на основе политик**

В этой главе рассматриваются несколько возможностей системной среды компонента Database Engine. Вначале предоставляется подробное описание системных баз данных, которые создаются в процессе инсталляции сервера. Затем обсуждается хранение данных, исследуя несколько типов дисковых страниц и описывая способы сохранения на диске разных типов данных. После этого представляются системные утилиты `bcp`, `sqlcmd` и `sqlsevr` и системные команды DBCC. В последнем основном разделе главы обсуждается новая технология, введенная в SQL Server 2008, — управление на основе политик.

Системные базы данных

При установке компонента Database Engine создаются следующие системные базы данных:

- `master`;
- `model`;
- `tempdb`;
- `msdb`.

ПРИМЕЧАНИЕ

Существует еще другая, "скрытая" системная база данных, называемая базой данных `resource`, которая применяется для хранения системных объектов, таких как

системные хранимые процедуры и функции. Содержимое этой базы данных обычно применяется для выполнения обновлений системы.

В последующих разделах описываются отдельные системные базы данных.

База данных *master*

База данных `master` является самой важной системной базой данных компонента Database Engine. Она содержит все системные таблицы, требуемые для работы системы баз данных. Например, в этой базе данных хранится информация обо всех других базах данных, управляемых компонентом Database Engine, о соединениях системы с клиентами и об авторизациях пользователей.

Вследствие важности этой системной базы данных, для нее всегда нужно сохранять текущую резервную копию. Кроме этого, база данных `master` подвергается изменениям при каждом выполнении таких операций, как создание пользовательской базы данных или таблицы. По этой причине, для нее нужно осуществлять резервное копирование после каждой такой операции. (Причины, по которым необходимо создавать резервную копию базы данных `master`, рассматриваются более подробно в *разд. "Резервное копирование базы данных master" главы 16*.)

База данных *model*

База данных `model` применяется в качестве шаблона при создании определяемых пользователем баз данных. Она содержит все системные таблицы базы данных `master`, которые требуются для каждой определяемой пользователем базы данных. Системный администратор может изменять свойства базы данных `model`, чтобы приспособить ее к специфическим требованиям своей системы.

ПРИМЕЧАНИЕ

Поскольку база данных `model` используется в качестве модели при создании каждой новой базы данных, то ее можно расширить определенными объектами базы данных и/или разрешениями. Эти новые свойства затем будут наследоваться всеми новыми базами данных. Для расширения или изменения свойств базы данных `model` используется та же самая инструкция `ALTER DATABASE`, что и для модификации пользовательских баз данных.

База данных *tempdb*

База данных `tempdb` используется для хранения временных таблиц и других необходимых временных объектов. Например, в этой базе данных система сохраняет промежуточные результаты вычислений всех сложных выражений. База данных `tempdb` используется всеми базами данных системы. Содержимое этой базы данных удаляется при каждом перезапуске системы.

В базе данных tempdb система сохраняет следующие типы данных:

- пользовательские объекты;
- внутренние объекты;
- историю версий.

Создаваемые пользователями личные и глобальные временные таблицы хранятся в базе данных tempdb. Также в ней хранятся такие объекты, как табличные переменные и табличные значения функций. Система обращается со всеми хранящимися в базе данных tempdb объектами таким же образом, как и с любыми другими объектами базы данных. Это означает, что записи о временных объектах сохраняются в системном каталоге, и информацию о них можно получить с помощью представления каталога sys.objects.

Внутренние объекты аналогичны с пользовательскими объектами, с тем исключением, что их нельзя просматривать с помощью представлений каталога или использовать другие инструменты для получения их метаданных. Существует три типа внутренних объектов: рабочие файлы, рабочие таблицы и элементы сортировки. Рабочие файлы создаются, когда система извлекает информацию, используя определенные операторы. Рабочие таблицы создаются, когда выполняются некоторые операции, такие как буферизация памяти (спулинг) и восстановление баз данных или таблиц с помощью команды DBCC. Наконец, элементы сортировки создаются при выполнении операций сортировки.

База данных tempdb также используется при оптимистическом одновременном конкурентном доступе к данным (см. *главу 13*) для хранения версий строк. Поэтому размер базы данных tempdb увеличивается каждый раз, когда система помимо прочих выполняет следующие определенные операции:

- выполняется триггер;
- выполняется инструкция INSERT, UPDATE или DELETE в базе данных, которая допускает уровень изоляции SNAPSHOT.

ПРИМЕЧАНИЕ

Вследствие оптимистического одновременного конкурентного доступа база данных tempdb используется системой довольно интенсивно. По этой причине следует убедиться, что для нее выделено достаточно места, а также постоянно контролировать объем памяти. (Применение базы данных tempdb для оптимистического одновременного конкурентного доступа к данным описывается в *главе 13*.)

База данных *msdb*

База данных msdb используется компонентом SQL Server Agent для планирования извещений (alert) и задач. Эта системная база данных содержит информацию о планировании задач, обработке исключений, управлении извещениями и системных операторах. Это такая информация, как, например, адреса электронной почты, номера пейджеров и история обо всех операциях резервного копирования и вос-

становления. Более подробно о восстановлении этой системной базы данных см. *главу 16*.

Хранение данных на диске

Архитектура хранилища компонента Database Engine содержит следующие элементы для хранения объектов баз данных:

- страницы;
- экстенты;
- файлы;
- файловые группы.

ПРИМЕЧАНИЕ

Файлы и файловые группы не включены в рассмотрение в этой главе, поскольку они подробно рассмотрены в *главе 5*.

Основной единицей хранилища данных является *страница*. Размер страницы постоянен и составляет 8 Кбайт. Каждая страница имеет заголовок размером в 96 байтов, в котором хранится системная информация. Строки данных размещаются на странице сразу же после заголовка.

Компонент Database Engine поддерживает несколько разных типов страниц, наиболее важными из которых являются следующие:

- страницы данных;
- страницы индексов.

ПРИМЕЧАНИЕ

Страницы данных и индексов в действительности являются физическими составляющими базы данных, где хранятся соответствующие таблицы и индексы. Содержимое базы данных хранится в одном или нескольких файлах, а каждый файл разделен на несколько страниц. Поэтому каждую страницу таблицы или индекса можно однозначно идентифицировать (как физическую единицу базы данных), используя идентификатор базы данных, идентификатор файла базы данных и номер страницы.

При создании таблицы или индекса система выделяет определенный объем дискового пространства для хранения данных таблицы или индекса. После заполнения этого пространства необходимо выделять дополнительную память. Физическая единица дискового пространства, используемая для выделения памяти для таблиц и индексов, называется *экстентом*. Размер экстента составляет восемь последовательно расположенных страниц, или иначе 64 Кбайт. Существует два следующих типа экстентов:

- однородные экстенты;
- смешанные экстенты.

Однородные экстенты содержат данные одной таблицы или индекса, а смешанные могут содержать данные до восьми таблиц или индексов. Система всегда выделяет первые страницы для смешанных экстентов. Затем, если для хранения таблицы или индекса требуется больше, чем восемь страниц, выделяются дополнительные экстенты однородного типа.

Свойства страниц данных

Все типы страниц данных имеют фиксированный размер (8 Кбайт) и состоят из следующих трех частей:

- заголовка страницы;
- пространства для данных;
- таблицы смещений строк.

ПРИМЕЧАНИЕ

В этой главе мы не обсуждаем отдельно свойства страниц индексов, поскольку эти страницы почти идентичны страницам данных.

Эти составляющие страниц данных описываются в следующих далее разделах.

Заголовок страницы

Каждая страница имеет заголовок размером в 96 байтов, в котором хранится системная информация, такая как идентификатор страницы, идентификатор объекта базы данных, которому принадлежит страница, а также указатели на предыдущую и следующую страницы в цепочке страниц. Как вы, возможно, уже догадались, заголовок страницы находится в начале каждой страницы. В табл. 15.1 представлена информация, которая хранится в заголовке страницы.

Таблица 15.1. *Информация, хранящаяся в заголовке страницы*

Информация	Описание
`pageId`	Идентификатор файла базы данных и идентификатор страницы
`level`	Для страниц индексов, уровень страницы (уровень листьев страницы является уровнем 0, страницы первого промежуточного уровня имеют уровень 1 и т. д.)
`flagBits`	Дополнительная информация о странице
`nextPage`	Идентификатор файла базы данных и идентификатор страницы следующей страницы в цепочке (если страница имеет кластеризованный индекс)
`prevPage`	Идентификатор файла базы данных и идентификатор страницы предшествующей страницы в цепочке (если страница имеет кластеризованный индекс)
`objId`	Идентификатор объекта базы данных, которой принадлежит страница
`lsn`	Порядковый номер транзакции в журнале (см. *главу 13*)

Таблица 15.1 *(окончание)*

Информация	Описание
slotCnt	Общее число слотов (сегментов), используемых на данной странице
indexId	Идентификатор индекса страницы (0, если это страница данных)
freeData	Смещение в байтах до первого свободного пространства страницы
pminlen	Количество байтов в части фиксированной длины строк
freeCnt	Количество свободных байтов страницы
reservedCnt	Количество байтов, зарезервированных всеми транзакциями
xactReserved	Количество байтов, зарезервированных самыми последними запущенными транзакциями
xactId	Идентификатор самой последней запущенной транзакции
tornBits	Один бит на сектор для определения незавершенной записи страницы

Пространство для данных

Часть страницы, зарезервированная для данных, имеет переменный размер, зависящий от количества строк в странице и их длины. Для каждой строки страницы создается запись в пространстве, зарезервированном для данных, и запись в таблице смещения строк в конце таблицы. (Строка данных не может выходить за пределы одной страницы, за исключением значения типа VARCHAR(max) и VARBINARY(max), которые хранятся в страницах, выделенных специально для них). Строки сохраняются в последовательном порядке после уже сохраненных строк до полного заполнения страницы. Если в странице нет места для новой строки той же таблицы, эта строка сохраняется в следующей странице цепочки страниц.

Для таблиц, которые имеют только столбцы фиксированного размера, в каждой странице сохраняется одинаковое количество строк. Если в таблице есть хотя бы один столбец переменной длины (например, столбец типа данных VARCHAR), количество строк в странице может колебаться, и система сохраняет в странице столько строк, сколько в нее поместится.

Таблица смещения строк

Эта последняя часть страницы тесно связана с выделенным для данных пространством, поскольку для каждой хранящейся в странице строки имеется соответствующая запись в таблице смещения строк (рис. 15.1).

Таблица смещения строк состоит из 2-байтовых записей, каждая из которых содержит номер строки и смещение в байтах адреса строки на странице. (Порядок записей в таблице смещения строк обратный порядку строк в странице, т. е. эти записи идут справа налево.) Предположим, что каждая строка таблицы имеет фиксированную длину в 36 байт. Тогда первая строка таблицы сохраняется на странице со смещением в 96 байт от начала страницы (начиная с первого байта после заголовка

страницы). Соответствующая запись в таблице смещения строк вносится в последние два байта страницы, указывая номер строки (в первом байте) и смещение строки (во втором байте). Следующая строка сохраняется в следующих 36 байтах страницы, а ее соответствующая запись в таблице смещения байтов вносится в третьем и четвертом от конца страницы байтах, опять указывая номер строки (1) и смещение строки (132).

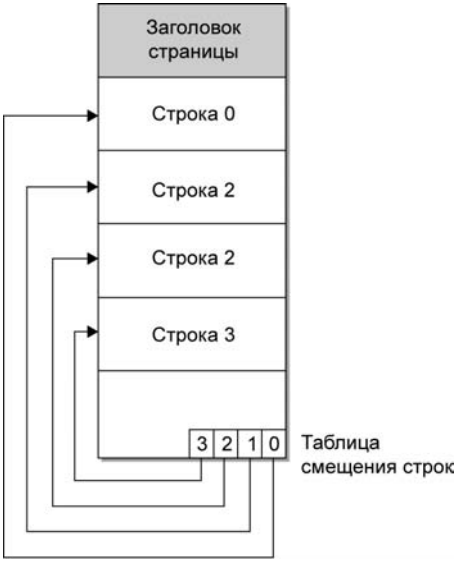


Рис. 15.1. Структура страницы данных

Типы страниц данных

В страницах данных сохраняются, как и следовало ожидать, данные таблицы. Существует два типа страниц данных для хранения данных разных форматов:

- страницы данных последовательных строк (in-row data pages);
- страницы данных переполнения строк (row-overflow data pages).

Страницы данных последовательных строк

О страницах данных последовательных строк нельзя сказать ничего особенного — это просто страницы, в которых удобно сохранять данные и информацию об индексах. Все данные, не являющиеся частью больших объектов, всегда сохраняются в строке. Кроме этого, значения типа `VARCHAR(max)`, `NVARCHAR(max)`, `VARBINARY(max)` и `XML` можно сохранять в строке, если параметру `large value types out of row` системной процедуры `sp_tableoption` присвоено значение `0`. В таком случае все такие значения сохраняются непосредственно в строке данных вплоть до верхнего предела в 8000 байт и при условии, что значение умещается в запись. Если значение не умещается в запись, в строке сохраняется указатель на данные, а сами данные сохраняются вне строки, в хранилище для больших объектов.

Страницы данных переполнения строк

Значения столбцов типа `VARCHAR(max)`, `NVARCHAR(max)` и `VARBINARY(max)` можно сохранять вне фактической страницы данных. Как уже упоминалось, максимальный размер строки страницы данных не может превышать 8 Кбайт, но значения этих типов данных могут превысить этот предел. В таком случае система сохраняет значения этих столбцов вне строки, в дополнительных страницах, называемых *страницами переполнения строк* (row-overflow pages).

Данные сохраняются вне строки только при определенных обстоятельствах. Основным фактором является длина строки таблицы: если она превышает 8060 байтов, некоторые значения столбца сохраняются в страницах переполнения строки. (Значения столбца нельзя разделить для хранения одной части в фактической странице данных, а другой части — в странице переполнения строки.)

В качестве демонстрации сохранения таблицы с большими значениями, в примере 15.1 создается такая таблица, а затем в нее вставляется строка, общая длина столбцов которой превышает 8060 байтов.

Пример 15.1. Сохранение больших данных вне строки

```
USE sample;
CREATE TABLE mytable
       (col1 VARCHAR(1000),
         col2 VARCHAR(3000),
         col3 VARCHAR(3000),
         col4 VARCHAR(3000));
    INSERT INTO mytable
       SELECT REPLICATE('a', 1000), REPLICATE('b', 3000),
          REPLICATE('c', 3000), REPLICATE('d', 3000);
```

В примере 15.1 инструкцией `CREATE TABLE` создается таблица `mytable`, в которую последующей инструкцией `INSERT` вставляется новая строка. Поскольку общая длина столбцов строки составляет 10 000 байтов, то эта строка таблицы не умещается в строку страницы.

В запросе в примере 15.2 используется несколько представлений каталога для отображения информации, связанной с описанием типа страницы.

Пример 15.2. Отображение информации о типе страницы

```
USE sample;
SELECT  rows, type_desc AS page_type, total_pages AS pages
   FROM sys.partitions p JOIN sys.allocation_units a ON
                        p, partition_id = a.container_id
   WHERE object_id = object_id('mytable');
```

Этот запрос возвращает следующий результат:

rows	page_type	pages
1	IN_ROW_DATA	2
1	ROW_OVERFLOW_DATA	2

В примере 15.2 представления каталога `sys.partition` и `sys.allocation_units` объединяются вместе для отображения информации о таблице `mytable` и о хранилище строк этой таблицы. Представление `sys.partition` содержит одну строку для каждой секции таблицы или индекса. (Несекционированные таблицы, такие как `mytable`, содержат только одну секцию.)

Набор страниц одного определенного типа страниц данных называется *выделенным блоком памяти* (allocation unit). В столбце `type_desc` представления `sys.allocation_units` отображаются различные выделенные блоки памяти. Как можно видеть из результата выполнения примера 15.2, для одной строки таблицы `mytable` система выделила (или зарезервировала) две обычные страницы плюс две страницы переполнения строк.

ПРИМЕЧАНИЕ

Производительность системы может быть значительно снижена, если вашим запросам требуется доступ к большому количеству страниц переполнения строк.

Параллельное выполнение задач

Компонент Database Engine может обрабатывать разные задачи базы данных в параллельном режиме. В параллельном режиме могут выполняться следующие задачи:

- массовая загрузка данных;
- резервное копирование;
- выполнение запроса;
- индексы.

Для загрузки данных в параллельном режиме компонент Database Engine применяет утилиту `bcp`. (Утилита `bcp` рассматривается в следующем разделе.) Таблица, в которую загружаются данные, должна не иметь никаких индексов, а операцию загрузки нельзя регистрировать. (Выполнять параллельную загрузку данных в одну таблицу могут только приложения, использующие интерфейсы API на основе ODBC или OLE DB.)

Компонент Database Engine может выполнять резервное копирование баз данных или журналов транзакций на устройства записи (с магнитной лентой или дисковые), используя параллельное "расслоенное" копирование. В этом режиме страницы базы данных считываются несколькими потоками по одному экстенту за раз (см. *главу 16*).

Чтобы улучшить выполнение запросов, компонент Database Engine поддерживает параллельные запросы. Благодаря этой возможности, независимые части инструк-

ции SELECT можно выполнить, используя несколько собственных потоков компьютера. Каждый запланированный для параллельного выполнения запрос в своем плане выполнения содержит оператор обмена. (*Оператором обмена* называется оператор в плане выполнения запроса, который предоставляет управление процессом, перераспределение данных и управление потоком.) Для таких запросов система баз данных создает план параллельного выполнения запроса. Параллельные запросы существенно улучшают производительность инструкций SELECT, которые обрабатывают большие объемы данных.

На многопроцессорных компьютерах компонент Database Engine автоматически использует больше процессоров для выполнения операций с индексами, таких как создание и перестроение индекса. Количество процессоров, применяемых для выполнения одной индексной операции, определяется конфигурационным параметром max degree of parallelism, а также текущей рабочей загрузкой. Если система баз данных определяет, что система занята, перед выполнением инструкции степень параллелизма автоматически понижается.

Утилиты и команда DBCC

Утилиты представляют собой компоненты, обеспечивающие такие возможности, как надежность данных, определение данных и функции сохранения статистики. Далее рассматриваются следующие утилиты:

- bcp;
- sqlcmd;
- sqlservr.

После рассмотрения этих утилит обсуждаются команды DBCC.

Утилита *bcp*

Утилита bcp (Bulk Copy Program, программа массового копирования) представляет собой полезную вспомогательную программу для обмена (копирования) данными между базой данных и файлом данных. Поэтому эта утилита часто применяется для перемещения больших объемов данных в базу данных компонента Database Engine с базы данных другой СУБД и наоборот, используя для этого текстовый файл. Эта утилита имеет следующий синтаксис:

```
bcp [[db_name.]schema_name.]table_name {IN | OUT | QUERYOUT | FORMAT}
        file_name [{-option parameter} ...]
```

В параметре *db_name* указывается имя базы данных, которой принадлежит таблица *table_name*. Параметры IN или OUT указывают направление перемещения данных. Если указан параметр IN, данные копируются с файла *file_name* в таблицу *table_name*, и в обратном направлении, если указан параметр OUT. Параметр FORMAT задает создание файла формата на основе указанных опций. При использовании этого параметра также требуется использовать опцию -f.

ПРИМЕЧАНИЕ

Если указан параметр IN, содержимое файла добавляется к содержимому таблицы базы данных, тогда как параметр OUT перезаписывает содержимое файла содержимым таблицы.

Данные могут копироваться как текст формата SQL Server или как текст формата ASCII. Копирование данных в формате SQL Server означает работу в собственном режиме сервера базы данных, а в формате ASCII — работу в символьном режиме. Параметр -n задает работу в режиме базы данных, а параметр -c — работу в символьном режиме. Собственный режим используется для перемещения данных между системами под управлением Database Engine, а символьный режим — для перемещения данных между экземпляром Database Engine и другими системами баз данных.

В примере 15.3 показано применение утилиты bcp. (Эту команду нужно выполнять из командной строки оперативной системы Windows.)

Пример 15.3. Использование утилиты bcp

```
bcp AdventureWorks2012.Person.Address out "address.txt" -T -c
```

Команда bcp в примере 15.3 экспортирует данные из таблицы address базы данных AdventureWorks2012 в выходной файл *address.txt*. Параметр -T задает использование доверительного соединения. (*Доверительное соединение* означает, что система использует средства безопасности операционной системы вместо проверки подлинности SQL Server.) Параметр -c задает символьный режим, вследствие чего данные сохраняются в ASCII-файл.

ПРИМЕЧАНИЕ

Следует знать, что альтернативой утилиты bcp является инструкция BULK INSERT. Эта инструкция поддерживает все параметры утилиты bcp (хотя ее синтаксис несколько другой) и предоставляет намного лучшую производительность. Инструкция BULK INSERT рассмотрена в *главе 7*.

Для импортирования данных с файла в таблицу базы данных требуется иметь разрешение на выполнение инструкций INSERT и SELECT для этой таблицы. Для экспортирования данных из таблицы в файл базы данных требуется иметь разрешение на выполнение инструкции SELECT для этой таблицы.

Утилита *sqlcmd*

Утилита sqlcmd позволяет вводить инструкции языка Transact-SQL, системные процедуры и файлы сценариев на подсказку в командной строке. Синтаксис утилиты sqlcmd выглядит следующим образом:

```
sqlcmd {option [parameter]} ...
```

В параметре *option* указывается требуемый параметр утилиты, а в параметре *parameter* — значение параметра утилиты, указанного в параметре *option*. Утилита sqlcmd имеет большое число параметров, наиболее важные из которых описаны в табл. 15.2.

Таблица 15.2. *Наиболее важные параметры утилиты sqlcmd*

Параметр	Описание
-S *server_name*[*instance_name*]	Задает имя и экземпляр сервера баз данных, с которым выполняется соединение. Если этот параметр опущен, то соединение выполняется с сервером баз данных, указанным в переменной среде SQLSERVER. Если этой переменной среды не присвоено значение, выполняется соединение с локальным сервером
-U *login_id*	Задает регистрационное имя входа в систему SQL Server. Если этот параметр опущен, то используется значение переменной среды SQLCMDUSER
-P *password*	Задает пароль для регистрационного имени входа. Если параметры -U и -P не указаны, то утилита sqlcmd пытается подключиться к серверу баз данных, используя режим аутентификации Windows. Аутентификация основывается на использовании учетных данных Windows пользователя, который выполняет утилиту sqlcmd
-c *command_end*	Задает завершающий символ пакета (терминатор). Значением по умолчанию для этого параметра является GO. С помощью этого параметра можно задать в качестве терминатора пакета символ точки с запятой (;), который является терминатором по умолчанию почти во всех других системах баз данных
-i *input_file*	Задает имя файла, содержащего пакет или хранимую процедуру. Файл должен содержать (по крайней мере одну) команду терминатора. Вместо -i можно использовать символ <
-o *output_file*	Указывает имя файла, получающего результаты выполнения утилиты. Вместо -o можно использовать символ >
-E	Задает использование доверительного соединения (см. *главу 12*) вместо запроса пароля
-A	Задает подключение к серверу посредством *DAC-подключения* (Dedicated Administrator Connection, выделенное подключение администратора), которое описывается после этой таблицы
-L	Выводит список всех экземпляров базы данных, обнаруженных в сети
-t *seconds*	Задает тайм-аут в секундах. Данный тайм-аут определяет период времени, в течение которого утилита должна ожидать установления подключения, прежде чем считать попытку подключения неуспешной

Таблица 15.2 *(окончание)*

Параметр	Описание
`-?`	Отображает краткую справку по синтаксису параметров утилиты `sqlcmd`
`-d` *`dbname`*	Задает текущую базу данных при запуске утилиты `sqlcmd`

Использование утилиты `sqlcmd` показано в примере 15.4.

Пример 15.4. Выполнение пакета инструкций посредством утилиты `sqlcmd`

```
sqlcmd -S NTB11901 -i C:\ms0510.sql -o C:\ms0510.rpt
```

ПРИМЕЧАНИЕ

Прежде чем выполнять код примера 15.4, в нем нужно указать соответствующее имя сервера вместо используемого в примере и убедиться в наличии файла ввода.

В примере 15.4 пользователь сервера баз данных NTB11900 выполняет пакет инструкций, хранящийся в файле *ms0510.sql*, а результат выполнения этого пакета сохраняется в файле *ms0510.rpt*. В зависимости от используемого режима аутентификации, система может запросить ввести имя пользователя и пароль (аутентификация SQL Server) или просто выполнить эту инструкцию (аутентификация Windows).

Одним из наиболее важных параметров утилиты `sqlcmd` является параметр `-A`. Как описывается в табл. 15.2, этот параметр задает соединение с экземпляром Database Engine через выделенное подключение администратора. Обычно подключение к экземпляру Database Engine осуществляется посредством среды Management Studio. Но в определенных исключительных ситуациях пользователи не могут подключиться к экземпляру таким способом. В таком случае подключение может быть возможным посредством DAC-подключения.

Это специальное подключение, которое администратор базы данных может использовать в случае крайнего уменьшения ресурсов сервера. Даже когда для подключения других пользователей нет достаточных ресурсов, компонент Database Engine будет пытаться освободить ресурсы для подключения DAC. Таким образом администраторы могут осуществлять поиск и устранение неполадок в экземпляре, не прекращая его работу.

Утилита `sqlcmd` поддерживает несколько специфичных команд, которые можно использовать в ней вдобавок к инструкциям языка Transact-SQL. Описание наиболее важных из этих команд приводится в табл. 15.3.

В примере 15.5 показано использование команды `exit` утилиты `sqlcmd`.

Таблица 15.3. *Наиболее важные команды утилиты* `sqlcmd`

Команда	Описание
`:ED`	Запускает текстовый редактор, с помощью которого можно редактировать текущий пакет или последний исполненный пакет. Конкретный тип редактора задается значением переменной среды `SQLCMDEDITOR`. Например, чтобы использовать в качестве текстового редактора Microsoft WordPad, следует ввести команду `SET SQLCMDEDITOR=wordpad`
`:!!`	Выполняет команды операционной системы. Например, команда `:!! dir` отображает содержимое текущей папки
`:r` *`filename`*	Выполняет синтаксический анализ дополнительных инструкций Transact-SQL и команд утилиты `sqlcmd` в файле, указанном в параметре *`filename`*, загружая их в кэш инструкций. Можно указывать несколько команд `:r`. Таким образом, с помощью этой команды можно связывать несколько скриптов с утилитой `sqlcmd`
`:list`	Отображает содержимое кэша инструкций
`:QUIT`	Прекращается сеанс `sqlcmd`
`:EXIT [(statement)]`	Позволяет использовать результат выполнения инструкции `SELECT` в качестве возвращаемого значения утилиты `sqlcmd`

Пример 15.5. Использование команды `exit` утилиты `sqlcmd`

```
1>USE sample;
2>SELECT * FROM project
3>:EXIT(SELECT @@rowcount)
```

Данный пример отображает все строки таблицы `project` и число 3, если эта таблица содержит три строки.

Утилита *sqlservr*

Самым удобным способом запуска экземпляра компонента Database Engine будет автоматический вместе с запуском компьютера. Но в некоторых обстоятельствах может потребоваться запустить сервер базы данных по-иному. На случай таких ситуаций компонент Database Engine предоставляет для запуска экземпляра сервера, среди прочих, утилиту `sqlservr`.

ПРИМЕЧАНИЕ

Запустить экземпляр сервера Database Engine можно также с помощью среды Management Studio или сетевой команды.

Для запуска утилиты `sqlservr` нужно в командной строке ввести следующую команду:

```
sqlservr  option_list
```

В параметре команды *option_list* указываются все аргументы, передаваемые утилите sqlservr при ее запуске. Наиболее важные из этих аргументов приводятся в табл. 15.4.

Таблица 15.4. *Наиболее важные аргументы утилиты* sqlservr

Аргумент	Описание
-f	Экземпляр сервера запускается с минимальной конфигурацией
-m	Экземпляр сервера запускается в однопользовательском режиме. Этот аргумент указывается при проблемах с системой, когда необходимо выполнить на ней техническое обслуживание. (Как правило, этот параметр используется при восстановлении системной базы данных master.)
-s *instance_name*	Указывает экземпляр SQL Server для подключения. Если именованный экземпляр не указан, то запускается экземпляр Database Engine по умолчанию

Команды DBCC

Язык Transact-SQL поддерживает команды *DBCC* (Database Console Commands, консольные команды базы данных), которые выступают в качестве консольных команд базы данных для компонента Database Engine. В зависимости от применяемых параметров, команды DBCC можно сгруппировать по следующим категориям:

- обслуживаемые;
- информационные;
- проверки;
- смешанные.

ПРИМЕЧАНИЕ

В этом разделе рассматриваются только команды проверки. Команды других категорий рассматриваются в связи с их применением. Например, команда DBCC SHOW_STATISTICS рассмотрена в *главе 19*, а команда DBCC USER_OPTIONS — в *главе 13*.

Команды проверки

Команды проверки осуществляют проверку согласованности базы данных. В эту группу входят следующие команды:

- DBCC CHECKALLOC;
- DBCC CHECKTABLE;

- DBCC CHECKCATALOG;
- DBCC CHECKDB.

Команда DBCC CHECKALLOC проверяет выделение памяти для каждого указанного системой экстента, а также существует ли выделенные экстенты, которые не отмечены системой. Таким образом, эта команда выполняет перекрестную проверку экстентов.

Команда DBCC CHECKTABLE проверяет целостность всех страниц и структур, составляющих таблицу или индексируемое представление. Проверки выполняются как на физическую, так и на логическую целостность. Проверки на физическую целостность контролируют целостность физической структуры диска. Проверки на логическую целостность контролируют, среди прочего, наличие для каждой строки базовой таблицы соответствующей строки в каждом некластеризованном индексе и наоборот, а также правильность сортировки индексов. При запуске этой команды с параметром PHYSICAL_ONLY выполняется проверка только физической структуры страницы. Задание этого параметра намного сокращает время выполнения команды и поэтому рекомендуется в случаях частого ее использования на рабочих системах.

Команда DBCC CHECKCATALOG проверяет согласованность каталога в указанной базе данных. Для этого выполняется множество перекрестных проверок между таблицами и системным каталогом. После завершения работы команды DBCC CATALO в журнале ошибок делается соответствующая запись. В случае успешного выполнения команды в сообщении указывается этот факт, а также длительность выполнения команды. В случае прекращения выполнения команды вследствие ошибки, в сообщении указывается факт завершения команды, значение состояния и длительность выполнения команды.

Команда DBCC CHECKDB применяется для проверки физической и логической целостности всех объектов в указанной базе данных. (Собственно говоря, эта команда выполняет все рассмотренные выше команды в порядке их рассмотрения.)

ПРИМЕЧАНИЕ

Все команды DBCC для проверки целостности системы используют технологию snapshot (см. *главу 13*) для обеспечения согласованности транзакций. Иными словами, операции проверки не пресекаются с текущими операциями базы данных, поскольку они используют версии текущих строк.

Управление на основе политик

Компонент Database Engine поддерживает систему для управления одним или несколькими экземплярами, базами данных или другими объектами баз данных, называющуюся *управлением на основе политик* (policy-based management). Но прежде чем приступить к изучению принципов работы этой инфраструктуры, необходимо понимать ее ключевые термины и концепты.

Ключевые термины и концепты управления на основе политик

Ниже приводится список основных терминов управления на основе политик, а далее следует описание основных концептов, связанных с этими терминами:

- управляемый объект;
- набор объектов;
- аспект;
- условие;
- политика;
- категория.

Система управляет сущностями, называющимися *управляемыми объектами* (managed targets), которые могут быть экземплярами сервера баз данных, базами данных, таблицами или индексами. Все управляемые объекты, которые принадлежат экземпляру, формируют иерархию. *Набор объектов* (target set) представляет собой набор управляемых объектов, получаемый в результате применения фильтров к иерархии объектов. Например, если управляемым объектом является таблица, набор объекта будет содержать все индексы, принадлежащие этой таблице.

Аспектом (facet) называется набор логических свойств, которые моделируют поведение или характеристики определенных типов управляемых объектов. Количество и характеристики свойств встроены в аспект и их может удалять или добавлять только создатель аспекта. Некоторые аспекты применимы только к определенным типам управляемых объектов.

Условием (condition) в данном контексте называется логическое выражение, которое определяет набор разрешенных состояний управляемого объекта касательно аспекта. Опять же, некоторые условия применимы только к определенным типам управляемых объектов.

Политика (policy) — это условие вместе с его соответствующим поведением. Политика может иметь только одно условие, и ее можно включить или выключить. Политики управляются пользователями посредством использования категорий.

Политика принадлежит к одной и только одной категории. *Категория* (category) представляет собой группу политик, применяемых с целью предоставления пользователю большего уровня гибкости в случаях, когда используется программное обеспечение сторонних разработчиков. Владельцы базы данных могут подписать базу данных на набор категорий. Базой данных могут управлять только политики из подписанных для данной базы категорий. Все базы данных неявно подписаны на категорию политик по умолчанию.

Применение управления на основе политик

В данном разделе приводится пример использования управления на основе политик. В примере создается политика, условием которой является то, что степень заполнения индексов будет 60% для всех баз данных экземпляра сервера. (Описание степени заполнения индексов `FILLFACTOR` см. в *главе 10*.)

Данное управление на основе политик реализуется в три шага:

1. На основе аспекта создается условие.
2. Создается политика.
3. Политике присваивается категория.

Чтобы создать политику, запустите среду Management Studio и в обозревателе объектов разверните узел сервера, а в узле сервера последовательно разверните папки **Management** и **Policy Management**.

Теперь первым делом создаем условие. Щелкните правой кнопкой папку **Conditions** и в контекстном меню выберите пункт **New Condition**. В открывшемся диалоговом окне **Create New Condition** (рис. 15.2) введите в поле **Name** имя условия (в данном случае имя будет **SetFillFactor**), а затем в раскрывающимся списке **Facet** выберите пункт **Server Configuration**.

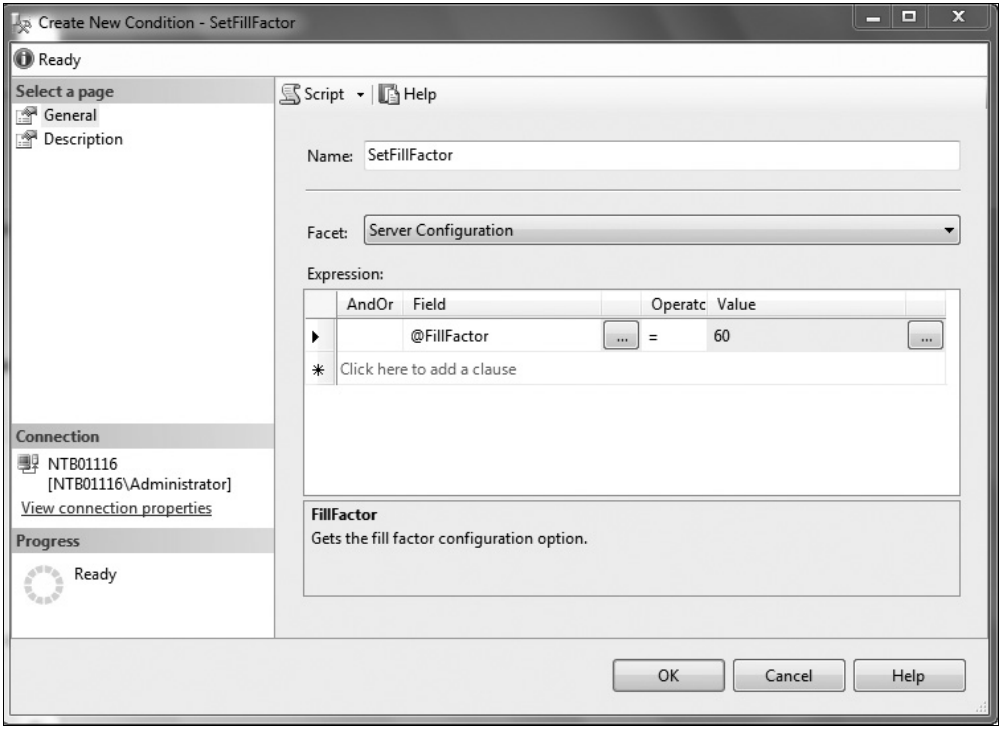


Рис. 15.2. Диалоговое окно **Create New Condition**

(Установка степени заполнения для всех баз данных экземпляра осуществляется в границах сервера и поэтому принадлежит конфигурации сервера.) В раскрывающемся списке столбца **Field** области **Expression** выберите значение **@FillFactor**, а в поле **Operator** выберите знак равенства (=). Наконец, введите значение `60` в поле **Value** и нажмите кнопку **OK**, чтобы сохранить выполненные настройки.

Следующим шагом является создание политики на основе только что созданного условия. В папке **Policy Management** щелкните правой кнопкой мыши папку

Policies и в контекстном меню выберите пункт **New Policy**. В открывшемся диалоговом окне **Create New Policy** (рис. 15.3) введите в поле **Name** имя новой политики (в данном случае **PolicyFillFactor60**).

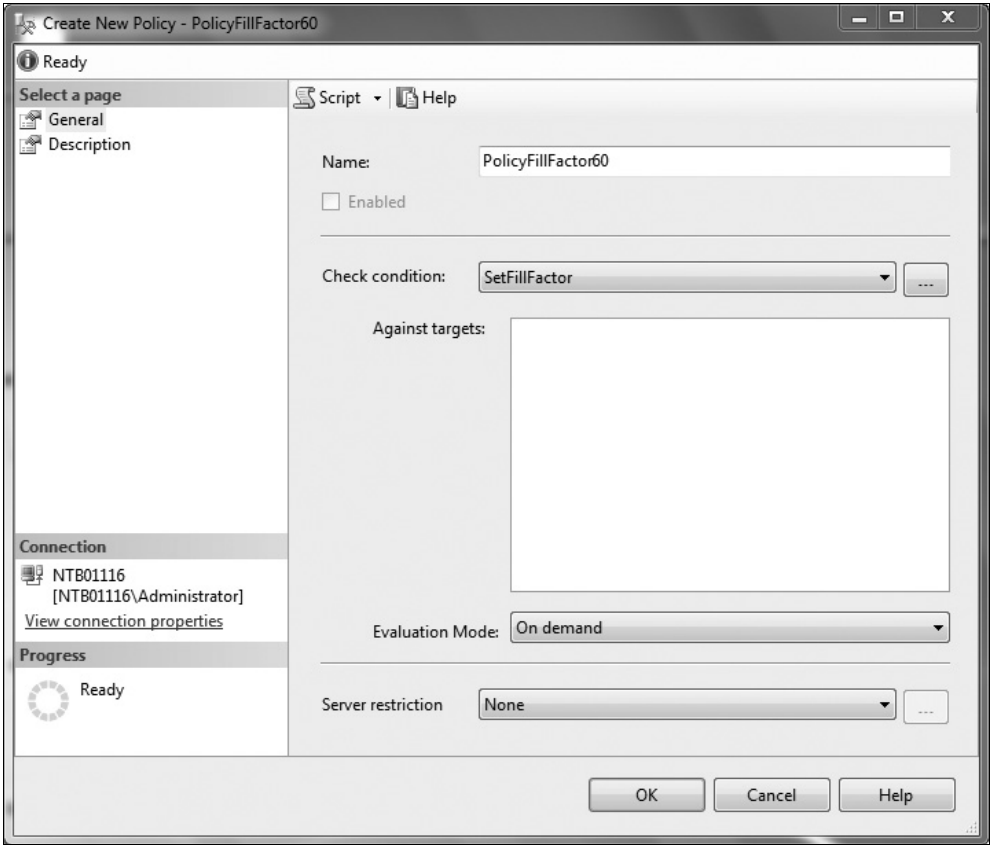


Рис. 15.3. Диалоговое окно **Create New Policy**

В раскрывающемся списке **Check condition** выберите созданное ранее условие (**SetFillFactor**). (Это условие находится в узле **Server Configurations**.) В раскрывающемся списке **Evaluation Mode** выберите пункт **On demand**.

ПРИМЕЧАНИЕ

Политики администраторов могут выполняться политиками по требованию или можно установить возможность автоматического выполнения политики, используя один из существующих режимов выполнения.

Созданную политику нужно поместить в категорию. Для этого выберите в диалоговом окне **Create New Policy** страницу **Description** (рис. 15.4).

Политики можно помещать в категорию **Default** или в более специфичную категорию. (Можно также создать свою категорию, нажав кнопку **New**.)

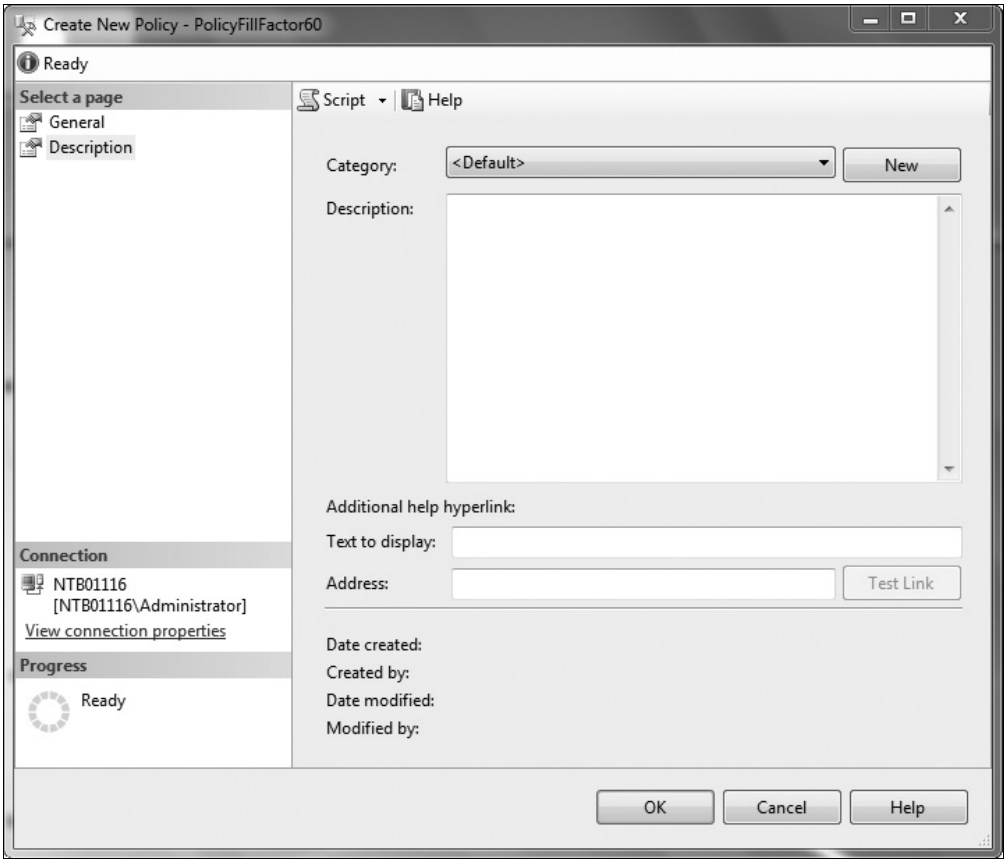


Рис. 15.4. Диалоговое окно **Create New Policy**, страница **Description**

Только что описанный процесс можно таким же образом применять к десяткам разных политик для серверов, баз данных и объектов баз данных.

Резюме

В этой главе мы рассмотрели несколько возможностей системной среды компонента Database Engine:

- системные базы данных;
- сохранение на диске;
- утилиты и команды;
- управление на основе политик.

Системные базы данных содержат системную информацию и информацию высокого уровня обо всей системе баз данных. Наиболее важной системной базой данных является `master`.

Основной единицей хранения данных является страница, размер которой 8 Кбайт. Наиболее важным типом страниц является страница данных. (Формат страниц индексов почти идентичен формату страниц данных.)

Компонент Database Engine поддерживает большое число утилит и команд, из которых в этой главе были рассмотрены три утилиты (`bcp`, `sqlcmd` и `sqlservr`) и команды проверки DBCC.

Начиная с версии SQL Server 2008, SQL Server поддерживает инфраструктуру управления на основе политик. Эта инфраструктура позволяет определять и принуждать следовать политикам для конфигурирования и управления базами данных и объектами баз данных по всему предприятию.

В следующей главе мы рассмотрим, как предотвратить потерю данных, используя резервное копирование и восстановление.

Упражнения

Упражнение 15.1

Где хранятся данные для временных баз данных?

Упражнение 15.2

Измените свойства базы данных `model`, установив для нее размер 4 Мбайта.

Упражнение 15.3

Назовите все ключевые термины управления на основе политик и изложите их роли и как они взаимосвязаны друг с другом.

Упражнение 15.4

Назовите все группы, для которых можно задать условие.

Упражнение 15.5

Создайте политику, которая отключает возможность использования общеязыковой среды выполнения (CLR).

Глава 16

Резервное копирование, восстановление и доступность системы

- **Причины потери данных**
- **Введение в методы резервного копирования**
- **Выполнение резервного копирования**
- **Выполнение восстановления базы данных**
- **Доступность системы**
- **Мастер плана обслуживания**

В этой главе сначала рассматриваются две самые важные задачи, связанные с администрированием системы: резервное копирование базы данных и ее восстановление. *Резервное копирование* (backup) означает процесс создания копии базы (или баз) данных и/или журналов транзакций на отдельных носителях, которые в случае необходимости могут быть использованы для восстановления исходных данных. *Восстановлением* (recovery) называется процесс замены неподтвержденных, несогласованных или потерянных данных данными с резервной копии.

Под *доступностью системы* (system availability) подразумевается свойство, связанное со сведением времени простоя системы базы данных к минимуму. В этой главе подробно рассматриваются такие варианты обеспечения доступности системы, как отказоустойчивая кластеризация (failover clustering), зеркальное отображение базы данных, доставка журналов транзакций (log shipping) и высокий уровень доступности и восстановления после сбоев (HADR — high availability and disaster recovery). Кроме этого, обсуждаются преимущества и недостатки каждого из этих методов.

В конце главы исследуется мастер плана обслуживания — Maintenance Plan Wizard, который предоставляет набор основных задач, требуемых для содержания базы

данных в исправном состоянии. Поэтому его можно использовать, среди прочего, для выполнения резервного копирования и восстановления пользовательских баз данных.

Причины потери данных

Выполнение резервного копирования является мерой предосторожности, которую необходимо предпринимать для того, чтобы предотвратить потерю данных. Причины потери данных можно разбить на следующие группы:

- программные ошибки;
- ошибки администратора (человеческий фактор);
- сбои в работе компьютера (отказ системы);
- неисправность дискового накопителя;
- стихийное бедствие (пожар, наводнение, землетрясение) или кража.

В процессе выполнения программы возможно возникновение условий, вызывающих ненормальное аварийное ее завершение. Такие программные ошибки связаны только с приложениями баз данных и обычно не распространяются на всю систему базы данных. Поскольку причиной таких ошибок является неисправность логики программы, то система базы данных в таких случаях не может выполнить восстановление. Поэтому восстановление должен выполнять программист, используя для обработки таких исключений инструкции `COMMIT` и `ROLLBACK` (см. *главу 13*).

Другой причиной потери данных является "человеческий фактор", иными словами, ошибка пользователя, ведь и администраторы тоже люди. Пользователи, которые имеют достаточно большие полномочия (например, администраторы базы данных), могут случайно удалить или исказить данные (известно много случаев, когда пользователи удаляли не ту базу данных, обновляли или удаляли не те данные, и т. п.). Конечно же, в идеальном случае такое никогда не должно происходить, и вы можете установить такой режим работы, при котором будет маловероятным, что производственные данные будут потеряны.

Однако вам следует осознать, что людям свойственно ошибаться, и данные будут теряться вследствие их ошибок. Самое лучшее, что вы можете сделать в этом отношении, — это предпринять все возможные меры для предотвращения потери данных вследствие человеческого фактора и быть готовым к восстановлению данных в случае такой ситуации.

Сбой в работе компьютера может произойти вследствие различных ошибок в программном или аппаратном обеспечении. В качестве одного примера системного сбоя можно назвать выход из строя оперативной памяти компьютера, вследствие чего ее содержимое будет потеряно. Другим примером можно назвать неисправность жесткого диска, вызванную выходом из строя головки чтения/записи, или когда система ввода/вывода в процессе записи либо чтения данных обнаруживает поврежденные сектора на диске.

В случае стихийного бедствия или кражи необходимо иметь в наличии достаточное количество зарезервированной информации для восстановления потерянных данных. Это обычно достигается хранением носителей с резервной информацией в отдельном от другого оборудования месте, предохраняя их, таким образом, от потери данных в случае кражи и стихийного бедствия.

В большинстве случаев, из только что описанных сбоев, выполнение резервного копирования, рассматриваемое в последующем материале этой главы, может обеспечить восстановление утраченных данных.

Введение в методы резервного копирования

Создание резервной копии базы данных представляет собой выгрузка данных (из базы данных, журнала транзакций или файла) на устройства резервного копирования, поддерживаемые системой. Устройством резервного копирования может быть жесткий диск или накопитель на магнитной ленте. Компонент Database Engine поддерживает как статическое, так и динамическое резервное копирование. *Статическое резервное копирование* означает, что в процессе создания резервной копии единственным активным сеансом, поддерживаемым системой, является сеанс, который создает данную резервную копию. Иными словами, выполнение пользовательских процессов в течение операции резервного копирования не разрешается. *Динамическое резервное копирование* означает, что резервная копия может создаваться без прекращения работы сервера базы данных, отключения пользователей или даже закрытия файлов. (Более того, пользователи даже не будут знать, что выполняется резервное копирование.)

Компонент Database Engine поддерживает четыре следующих метода создания резервных копий:

- полное резервное копирование базы данных;
- разностное резервное копирование;
- резервное копирование журнала транзакций;
- резервное копирование файлов или файловых групп.

Эти методы создания резервных копий рассматриваются в последующих разделах.

Полное резервное копирование базы данных

Полное резервное копирование базы данных (full database backup) фиксирует то состояние базы данных, которое она имела на момент начала выполнения резервного копирования. В процессе создания такой резервной копии система копирует как данные, так и схему всех таблиц базы данных и соответствующие файловые структуры. В случае если резервное копирование выполняется динамически, то копируются все действия, происходящие в процессе создания резервной копии. Таким образом, в резервную копию попадают даже неподтвержденные незафиксированные транзакции.

Разностное резервное копирование

Разностное резервное копирование (differential backup) создает копию только тех частей базы данных, которые были добавлены или изменены после выполнения последнего полного резервного копирования базы данных. (Как и в случае полного резервного копирования, любая деятельность во время динамического разностного резервного копирования также заносится в создаваемую копию.) Достоинством разностного резервного копирования является быстрота его выполнения. Для создания такой резервной копии базы данных требуется существенно меньше времени, чем для полной копии, поскольку объем копируемых данных значительно меньше. Вспомним, что полное резервное копирование базы данных включает копии всех ее страниц.

Резервное копирование журнала транзакций

Резервное копирование журнала транзакций (transaction log backup) учитывает только те данные, которые связаны с изменениями, записанными в журнал транзакций. Таким образом, эта форма резервного копирования основана не на физических составляющих базы данных (страницах), а на логических операциях, т. е. на изменениях, выполненных посредством DML-инструкций: `INSERT`, `UPDATE` и `DELETE`. Опять же, поскольку при этом копируется меньший объем данных, то этот процесс создания резервной копии может быть выполнен значительно быстрее, чем полное или разностное резервное копирование базы данных.

ПРИМЕЧАНИЕ

Выполнять резервное копирование журнала транзакций не имеет смысла, если не было проведено по крайней мере одно, полное резервное копирование базы данных.

Существует две основные причины, по которым следует выполнять резервное копирование журнала транзакций. Первая, чтобы сохранить на защищенном носителе данные, которые изменились со времени последнего резервного копирования журнала транзакций, а вторая, более важная, чтобы должным образом закрыть часть журнала транзакций перед началом новой порции действий его активной части. (Активная часть журнала транзакций содержит записи обо всех неподтвержденных незафиксированных транзакциях.)

Используя полное резервное копирование базы данных и надежные цепочки всех резервных копий журналов транзакций, можно распространить копию базы данных и на другой компьютер. Эту копию базы данных можно будет использовать для восстановления исходной базы данных в случае ее повреждения. (Подобным образом базу данных можно восстановить, используя полную резервную копию и последнюю разностную резервную копию.)

Компонент Database Engine не позволяет сохранять журнал транзакций в том же самом файле, в котором сохраняется база данных. Одной из причин этого является

то, что в случае повреждения этого файла использовать журнал транзакций для восстановления всех изменений, выполненных после последнего резервного копирования, будет невозможно.

Использование журнала транзакций для записи изменений в базе данных является достаточно широко распространенной функциональной возможностью, которая применяется почти во всех реляционных СУБД. Тем не менее, иногда могут возникнуть ситуации, в которых будет полезным эту возможность отключить. Например, обработка большого объема загружаемых данных может длиться несколько часов. В таком случае скорость выполнения программы можно увеличить, отключив протоколирование транзакций. Но с другой стороны, это чревато опасными последствиями, поскольку при этом разрушаются существующие цепочки журналов транзакций. Чтобы обеспечить успешное восстановление базы данных, после окончания загрузки данных настоятельно рекомендуется выполнить полное резервное копирование базы данных.

Одной из самых распространенных причин системных сбоев является переполнение журнала транзакций. Следует иметь в виду, что такая проблема может полностью остановить систему. Если дисковое пространство, выделенное для хранения журнала транзакций, полностью заполнится, то система будет вынуждена прекратить выполнение всех текущих транзакций до тех пор, пока для журнала не будет предоставлено дополнительное пространство. Этой проблемы можно избежать, только выполняя частое резервное копирование журнала транзакций: при каждом закрытии части журнала транзакций и сохранении его на другой носитель эта часть журнала становится повторно используемой, освобождая таким образом дисковое пространство.

ПРИМЕЧАНИЕ

Как разностное резервное копирование, так и резервное копирование журнала транзакций сокращают время, требуемое для создания резервной копии базы данных. Но между ними есть одно важное различие: резервная копия журнала транзакций содержит все множественные изменения строки, выполненные после последнего резервного копирования, тогда как разностная резервная копия содержит только последнее изменение этой строки.

Стоит отметить и другие различия между резервным копированием журнала транзакций и разностным резервным копированием. Преимущество разностного резервного копирования заключается в том, что оно позволяет сэкономить время при восстановлении, поскольку для полного восстановления базы данных требуется ее полная резервная копия и только *последняя* разностная резервная копия. Для полного же восстановления с использованием резервных копий журнала транзакций необходимо использовать полную резервную копию базы данных и все резервные копии журнала транзакций. Недостатком разностного резервного копирования является то, что такая копия не позволяет восстановить данные на определенный момент времени, т. к. в ней не сохраняются промежуточные изменения базы данных.

Резервное копирование файлов или файловых групп

Резервное копирование файлов или файловых групп позволяет вместо полной резервной копии базы данных создать резервную копию только определенных файлов (или файловых групп) базы данных. В таком случае компонент Database Engine создает резервную копию только указанных файлов. Из резервной копии базы данных можно восстанавливать отдельные файлы (или файловые группы), что позволяет выполнять восстановление данных после сбоя, который повлиял только лишь на часть файлов базы данных. Восстановление отдельных файлов или файловых групп можно выполнять из полной резервной копии базы данных или из резервной копии файловой группы. Это означает, что в качестве процедуры резервного копирования можно применять полное резервное копирование базы данных и резервное копирование журнала транзакций и в то же время иметь возможность восстанавливать отдельные файлы (или файловые группы) из этих резервных копий.

ПРИМЕЧАНИЕ

Резервное копирование файлов также называется *резервным копированием на уровне файлов*. Данный тип резервного копирования рекомендуется применять только в случае очень большой базы данных, когда нет времени достаточного для выполнения полного резервного копирования.

Выполнение резервного копирования базы данных

Резервное копирование базы данных можно выполнять, используя следующие средства:

- инструкции языка Transact-SQL;
- интегрированную среду SQL Server Management Studio.

Эти методы рассматриваются в последующих разделах.

Резервное копирование с помощью инструкций Transact-SQL

Все типы резервного копирования можно выполнять, используя только две инструкции языка Transact-SQL:

- `BACKUP DATABASE`;
- `BACKUP LOG`.

Прежде чем приступать к рассмотрению этих двух инструкций Transact-SQL, следует ознакомиться с устройствами, применяемыми для хранения резервной копии.

Типы устройств резервного копирования

Компонент Database Engine позволяет выполнять резервное копирование баз данных, журналов транзакции и файлов на следующие устройства резервного копирования:

- жесткие диски;
- накопители на магнитной ленте.

ПРИМЕЧАНИЕ

Существует еще один тип устройства резервного копирования — *сетевой диск* (network share). Этот тип устройства не будет рассматриваться отдельно, поскольку это просто обычный жесткий диск, который находится на файловом сервере с доступом по сети.

Жесткий диск является наиболее широко применяемым носителем для хранения резервных копий. Дисковые устройства для резервного копирования могут быть как локальными, установленными на компьютере сервера базы данных, так и удаленными, установленными на файловом сервере. Компонент Database Engine позволяет добавлять новую резервную копию в файл, который уже содержит резервные копии этой же или других баз данных. При добавлении нового резервного набора на носитель, уже содержащий резервную копию, ранее записанное содержимое носителя не затрагивается, а новая резервная копия добавляется на него после последней резервной копии. (Резервный набор содержит все сохраненные данные объекта, выбранного для резервного копирования.) По умолчанию компонент Database Engine всегда добавляет новые резервные копии к уже существующему файлу резервных копий.

ВНИМАНИЕ!

Никогда не сохраняйте файл резервной копии на тот же физический диск, на котором хранится база данных или ее журнал транзакций. В случае повреждения диска с базой данных будут повреждены не только файлы базы данных, но и файл резервной копии, вследствие чего восстановление базы данных будет невозможным.

Накопители на магнитной ленте обычно применяются для резервного копирования таким же образом, что и жесткие диски. Но для сохранения резервной копии на накопитель на магнитной ленте, он должен быть подключен к системе локально. Преимущество накопителей на магнитной ленте перед жесткими дисками состоит в простоте их администрирования и использования.

ПРИМЕЧАНИЕ

При хранении резервной копии на сетевом диске всегда следует проверять созданную копию, чтобы убедиться в том, что она не была повреждена вследствие сетевых ошибок.

Инструкция *BACKUP DATABASE*

Инструкция BACKUP DATABASE применяется для выполнения полного резервного копирования или разностного копирования базы данных. Эта инструкция имеет следующий синтаксис:

```
BACKUP DATABASE {db_name | @variable}
TO device_list
[MIRROR TO device_list2]
[WITH | option_list]
```

В параметре *db_name* указывается имя базы данных, для которой выполняется резервное копирование. Имя базы данных также может быть задано в переменной *@variable*. В параметре *device_list* указывается одно или несколько имен устройств, где будет храниться копия резервной копии. Параметр *device_list* может быть списком имен дисковых файлов или магнитных лент. Синтаксис для указания устройства выглядит следующим образом:

```
{logical_device_name @logical_device_name_var}
   | {DISK | TAPE} =
       {'physical_device_name' | @physical_device_ name_var}
```

Здесь имя устройства может быть или логическим именем (или переменной) или физическим именем, начинающимся с ключевых слов DISK или TAPE. (Ключевое слово TAPE будет удалено в будущих версиях SQL Server.)

В предложении MIRROR TO указывается, что сопутствующий набор устройств резервного копирования является зеркалом в зеркальном наборе носителей. Тип и количество указываемых в этом предложении устройств должны совпадать с типом и количеством основных устройств, указанных в предложении TO. Все устройства резервного копирования в зеркальном наборе должны иметь одинаковые свойства. (См. также описание зеркальных носителей в *разд. "Зеркальное отображение базы данных" далее в этой главе*.)

В параметре option_list содержится несколько опций, которые могут быть заданы для различных типов резервного копирования. Наиболее важными из них являются следующие:

- DIFFERENTIAL;
- NOSKIP/SKIP;
- NOINIT/INIT;
- NOFORMAT/FORMAT;
- UNLOAD/NOUNLOAD;
- MEDIANAME и MEDIADESCRIPTION;
- BLOCKSIZE;
- COMPRESSION.

Первая опция, DIFFERENTIAL, задает разностное резервное копирование. Все остальные опции в списке относятся к полному резервному копированию.

Опция SKIP отменяет проверку срока завершения и имени набора резервного копирования, которая обычно выполняется с помощью инструкции BACKUP DATABASE, чтобы предотвратить запись поверх существующих резервных наборов. Опция NOSKIP, используемая по умолчанию, указывает инструкции BACKUP, что должна

быть выполнена проверка даты завершения и имен всех наборов резервных копий, прежде чем выполнять их перезапись.

Опция INIT указывает, что нужно перезаписать все существующие данные на носителе, за исключением заголовка носителя, если таковой имеется. Если при указании этого параметра на носителе имеется резервный набор, срок действия которого не истек, операция резервного копирования завершается неудачей. В таком случае, для перезаписи устройства резервного копирования применяется комбинация опций SKIP и INIT. Если указана опция NOINIT, которая используется по умолчанию, создаваемая резервная копия добавляется к уже имеющимся на носителе резервным наборам.

Опция FORMAT служит для записи заголовка для всех файлов (или томов ленточных носителей), которые используются для резервного копирования. Следственно, эта опция применяется для инициализации носителя информации. Когда при резервном копировании на накопитель на магнитной ленте применяется опция FORMAT, то также неявно указываются опции INIT и SKIP. Подобным образом неявно указывается опция INIT, когда при сохранении резервной копии в файл жесткого диска применяется опция FORMAT. Опция NOFORMAT, которая является опцией по умолчанию, означает, что операция резервного копирования сохранит существующий заголовок носителей и резервные наборы данных на томах носителей, используемых для текущей операции резервного копирования.

Опции UNLOAD и NOUNLOAD применяются только при использовании в качестве носителей резервной копии магнитной ленты. Опция UNLOAD указывает, что после завершения операции резервного копирования необходимо выполнить перемотку магнитной ленты и снять ее с устройства. Эта опция применяется в начале сеанса резервного копирования по умолчанию. Опция NOUNLOAD указывает, что после завершения резервного копирования магнитная лента в устройстве оставляется.

MEDIADESCRIPTION и MEDIANAME указывают описание и имя набора носителей, соответственно. Опция BLOCKSIZE применяется для указания физического размера блока в байтах. Поддерживаются блоки размером в 512, 1024, 2048, 4096, 8192, 16 384, 32 768 и 65 536 (64 Кбайт) байт. Размер блока по умолчанию для накопителей на магнитной ленте — 65 536 байт, и 512 байт для всех других устройств.

Компонент Database Engine поддерживает сжатие резервной копии. Для применения этой возможности в инструкции BACKUP DATABASE указывается опция COMPRESSION. В примере 16.1 показано создание резервной копии базы данных sample со сжатием файла резервного копирования.

Пример 16.1. Создание резервной копии со сжатием

```
USE master;
BACKUP DATABASE sample
TO DISK = 'C:\sample.bak'
WITH INIT, COMPRESSION;
```

ПРИМЕЧАНИЕ

Если при выполнении этого примера система выдаст сообщение об ошибке "Access Denied" (доступ запрещен), укажите для сохранения файла резервной копии другой путь, например *C:\temp\sample.bak*.

Если вы хотите узнать, сжимается ли конкретный файл резервной копии, то используйте инструкцию RESTORE HEADERONLY, которая рассматривается далее в этой главе.

Инструкция *BACKUP LOG*

Инструкция BACKUP LOG применяется для создания резервной копии журнала транзакций. Эта инструкция имеет следующий синтаксис:

```
BACKUP LOG {db_name | @variable}
   TO device_list
      [MIRROR TO device_list2]
   [WITH option_list]
```

Значение параметров *db_name*, *@variable*, *device_list* и *device_list2* точно такое же, как и значение одноименных параметров инструкции BACKUP DATABASE. Параметр *option_list* имеет те же опции, как и одноименный параметр инструкции BACKUP DATABASE, но кроме этого он поддерживает специфические опции журнала транзакций NO_TRUNCATE, NORECOVERY и STANDBY.

Вы должны использовать опцию NO_TRUNCATE, если хотите, чтобы резервное копирование выполнялось без усечения журнала транзакций, т. е. эта опция не удаляет подтвержденные зафиксированные транзакции из журнала. После выполнения инструкции с этой опцией система записывает все недавние операции с базой данных в журнал транзакций. Таким образом, опция NO_TRUNCATE позволяет восстановить данные вплоть до точки сбоя в базе данных.

Опция NORECOVERY создает резервную копию остатка журнала и оставляет базу данных в состоянии восстановления. Эта опция полезна, когда сбой происходит во вторичной базе данных или при сохранении остатка журнала перед выполнением операции восстановления. Опция STANDBY выполняет резервное копирование остатка журнала и оставляет базу данных в режиме "только для чтения" и в состоянии STANDBY. (Операция восстановления и состояние STANDBY рассматриваются далее в этой главе.)

Резервное копирование с помощью интегрированной среды Management Studio

Прежде чем вы начнете выполнять резервное копирование базы данных или журнала транзакций, необходимо указать (или создать) устройства хранения резервной копии. Среда SQL Server Management Studio позволяет одним и тем же образом создавать дисковые устройства и устройства на магнитной ленте. Чтобы создать

устройство любого из этих типов, разверните в обозревателе объектов узел **Server Objects**, щелкните правой кнопкой папку "Backup Devices" и в контекстном меню выберите пункт **New Backup Device**. В открывшемся диалоговом окне **Backup Device** (рис. 16.1) выберите дисковое устройство или устройство на магнитной ленте, установив переключатель **File** или **Tape**, соответственно, и введите имя для этого устройства в поле **Device name**.

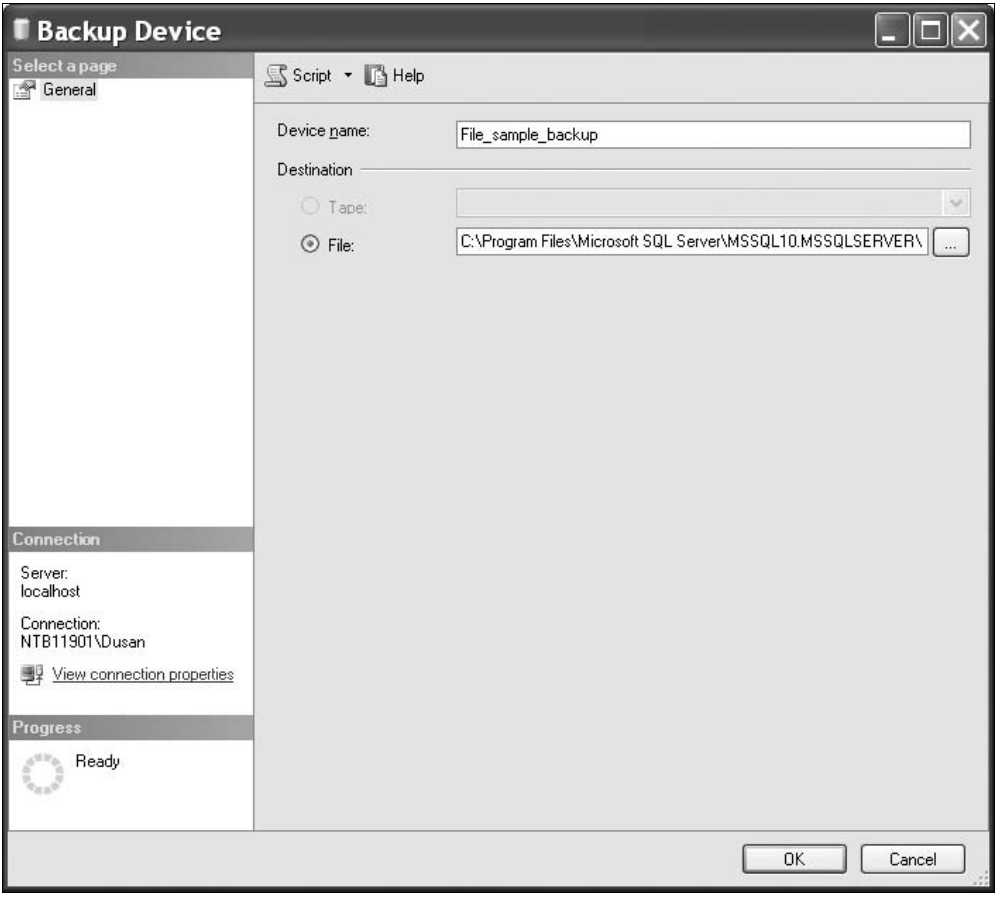


Рис. 16.1. Диалоговое окно **Backup Device**

В первом случае если выбран диск (переключатель **File**), то можно указать расположение файла резервной копии, щелкнув кнопку с тремя точками, расположенную справа от поля для отображения размещения существующих устройств, и выбрав требуемую папку в окне файловой иерархии. Во втором случае, если переключатель **Tape** недоступен для выбора, это означает, что на локальном компьютере отсутствует устройство на магнитных лентах.

После указания устройства для хранения резервной копии можно выполнять резервное копирование базы данных. Разверните в обозревателе объектов узел сервера базы данных, в нем разверните папку "Databases", щелкните правой кнопкой

требуемую базу данных и в контекстном меню выберите последовательность команд **Tasks | Back Up**. Откроется диалоговое окно **Back Up Database** (рис. 16.2).

На странице **General** этого диалогового окна выберите в раскрывающемся списке **Backup type** тип резервного копирования (полное — **Full**, разностное — **Differential** или копирование журнала транзакций — **Transaction Log**), в поле **Name** введите имя резервного набора и, необязательно, введите описание этого набора в поле **Description**. В этой же области можно указать срок действия резервной копии.

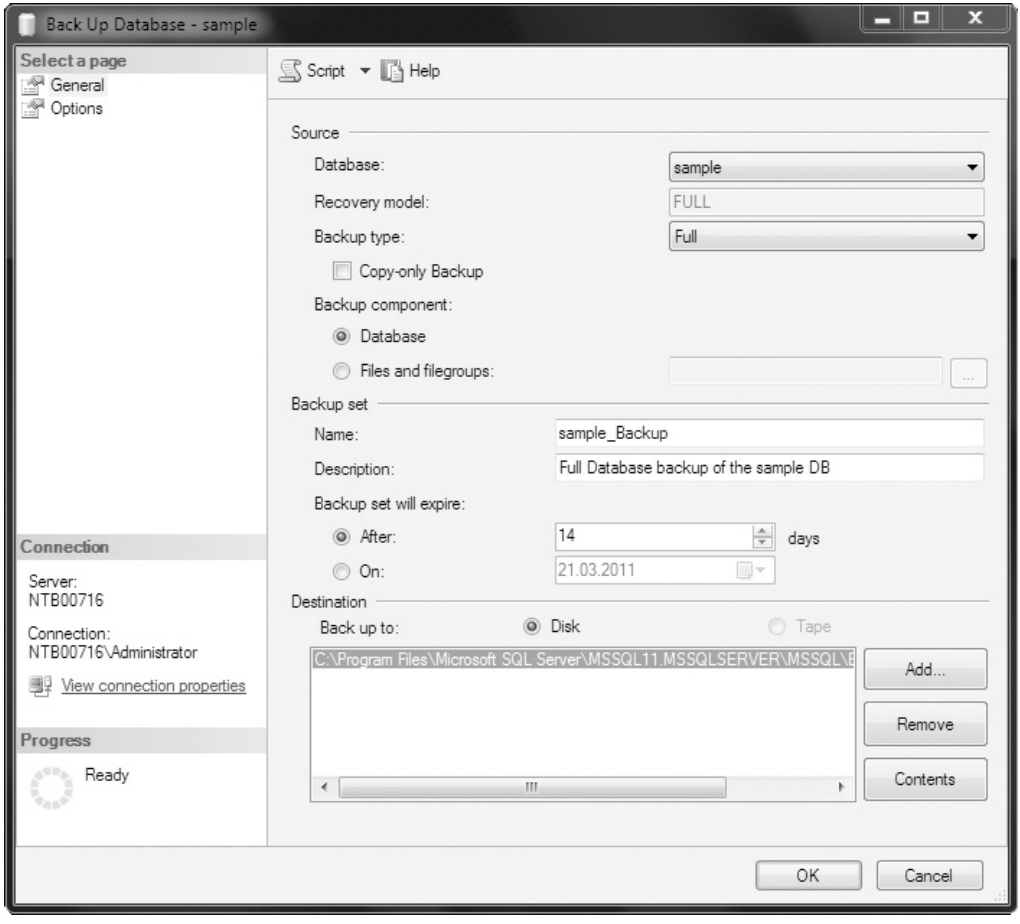


Рис. 16.2. Диалоговое окно **Back Up Database**, страница **General**

В области **Destination** выберите существующее устройство, нажав кнопку **Add**. (Кнопка **Remove** используется для удаления устройств резервного копирования из списка устройств, выбранных для хранения резервной копии.)

Чтобы добавить создаваемую резервную копию к уже существующей копии на выбранном устройстве, на странице **Options** (рис. 16.3) выберите переключатель **Append to the existing backup set**.

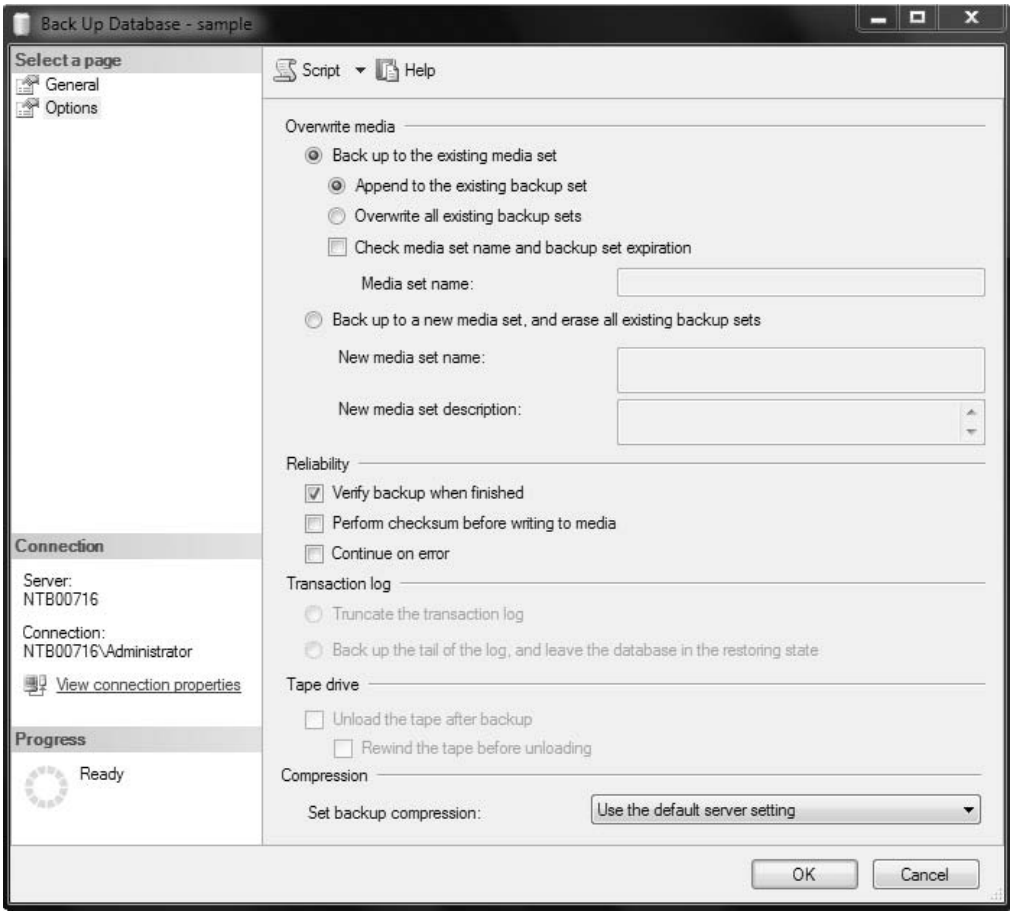


Рис. 16.3. Диалоговое окно **Back Up Database**, страница **Options**

Если в этой же области выбрать переключатель **Overwrite all existing backup sets**, то все существующие на выбранных устройствах резервные копии будут перезаписаны.

Чтобы выполнить проверку сохраненной резервной копии, в области **Reliability** установите флажок **Verify backup when finished**. На странице **Options** можно также задать сохранение резервной копии на новый набор носителей. Для этого в области **Overwrite media** нужно установить переключатель **Back up to a new media set, and erase all existing backup sets**, а затем ввести в соответствующие поля имя и описание набора носителей.

Для создании и проверки разностной резервной копии и резервной копии журнала транзакций в выпадающем списке **Backup type** на странице **General** (см. рис. 16.2) надо выбрать требуемый тип резервного копирования, а затем выполнить те же самые действия, что и для полного резервного копирования базы данных.

После того как вы зададите все требуемые параметры резервного копирования, для того чтобы создать резервную копию, нажмите кнопку **OK**. Чтобы просмотреть

имя, физическое расположение и тип устройств резервного копирования, разверните последовательно узел сервера, папку "Server Objects", папку "Backup Devices", а затем выберите требуемый файл.

Создание графика резервного копирования в среде SQL Server Management Studio

Хорошо спланированный график операций резервного копирования поможет при работе пользователей избежать ситуаций с перебоями в доступе к системе. Среда SQL Server Management Studio для планирования резервного копирования позволяет создавать такие графики, предоставляя простой в использовании графический интерфейс. Использование SQL Server Management Studio для планирования резервного копирования подробно рассматривается в следующей главе.

Для каких баз данных создавать резервную копию?

Нужно регулярно создавать резервную копию для следующих баз данных:

- базы данных `master`;
- все производственные базы данных.

Резервное копирование базы данных *master*

База данных `master` является самой важной базой данных системы, поскольку в ней хранится информация обо всех остальных базах данных системы. По этой причине резервное копирование базы данных `master` следует выполнять на регулярной основе. Кроме этого, создавать резервную копию этой базы данных следует после выполнения определенных инструкций и хранимых процедур, т. к. компонент Database Engine изменяет ее автоматически.

ПРИМЕЧАНИЕ

Для базы данных `master` можно выполнять только полное резервное копирование, поскольку система не поддерживает для нее разностное копирование, копирование журнала транзакций и копирование файлов.

Изменения в базе данных `master` вызываются многими действиями, среди которых можно назвать следующие:

- создание, изменение или удаление базы данных;
- изменение журнала транзакций.

ПРИМЕЧАНИЕ

При отсутствии резервной копии базы данных `master`, в случае необходимости ее восстановления, потребуется полностью воссоздать все системные базы данных, поскольку при повреждении базы данных `master` будут утрачены все ссылки на существующие пользовательские базы данных.

Резервное копирование производственных баз данных

Резервное копирование производственных баз данных следует выполнять на регулярной основе. Кроме этого, следует выполнять резервное копирование любой рабочей базы данных после выполнения следующих действий:

- после создания базы данных;
- после создания индексов;
- после очистки журнала транзакций;
- после выполнения незапротоколированных операций.

Всегда следует выполнять полное резервное копирование сразу же после создания базы данных на случай сбоя в период времени между ее созданием и ее первым запланированным резервным копированием. Помните, что выполнение резервного копирования журнала транзакций не может быть без полного резервного копирования базы данных.

Резервное копирование базы данных после создания одного или большего количества индексов позволяет сэкономить время при восстановлении, поскольку вместе с данными копируются и индексные структуры. Но резервное копирование журнала транзакций после создания индексов такой экономии не предоставляет, т. к. в журнале транзакций записывается только факт создания индекса, а сама измененная структура индекса не записывается.

Резервное копирование базы данных после очистки журнала транзакций необходимо по той причине, что журнал больше не содержит записей операций с базой данных, которые используются для восстановления базы данных. Все операции, которые не записаны в журнале транзакций, называются *незапротоколированными операциями*. Поэтому все изменения, выполненные этими операциями, не могут быть воссозданы в процессе восстановления.

Восстановление базы данных

Каждый раз, когда транзакция запускается на выполнение, компонент Database Engine становится ответственным за выполнение этой транзакции и запись ее изменений в базу данных или же за гарантию того, что транзакция не окажет никакого влияния на базу данных. Такой подход обеспечивает согласованность базы данных в случае сбоя, поскольку сбои повреждают не саму базу данных, а оказывают влияние на транзакции, которые выполнялись во время сбоя. Компонент Database Engine поддерживает как автоматическое, так и ручное восстановление базы данных, которые рассмотрены далее в последующих разделах.

Автоматическое восстановление

Автоматическое восстановление представляет собой средство, позволяющее сохранять работоспособность системы при возникновении ошибок. Компонент Database Engine выполняет автоматическое восстановление при каждом его перезапуске после сбоя или после его останова. Процесс автоматического восстановления прове-

ряет, требуется ли восстановление баз данных, и если обнаруживает такую необходимость, то возвращает каждую базу данных в ее последнее согласованное состояние, используя для этого журнал транзакций.

В процессе автоматического восстановления компонент Database Engine исследует журнал транзакций от последней контрольной точки до точки сбоя системы или останова Database Engine. (*Контрольной точкой* называется последняя точка, в которой все выполненные изменения записываются из оперативной памяти в базу данных. Таким образом, контрольная точка обеспечивает физическую согласованность данных.) Журнал транзакций содержит *подтвержденные (зафиксированные) транзакции*, т. е. транзакции, которые были успешно выполнены (зафиксированы в журнале транзакций), но их результаты еще не были записаны в базу данных, и *неподтвержденные (незафиксированные) транзакции*, т. е. транзакции, которые не были успешно выполнены на момент отключения или сбоя. Компонент Database Engine повторяет все подтвержденные транзакции, в результате чего делает изменения в базе данных постоянными, и отменяет часть неподтвержденных транзакций, которые были выполнены до контрольной точки.

Компонент Database Engine сначала выполняет восстановление базы данных `master`, после чего следует восстановление всех остальных системных баз данных. Процесс завершается восстановлением пользовательских баз данных.

Ручное восстановление

В процессе ручного восстановления применяется полная резервная копия базы данных, с последовательным применением всех резервных копий журнала транзакций в порядке их создания. (Альтернативно, после полной резервной копии можно применить последнюю разностную резервную копию.) Эти действия возвращают базу данных в то же самое (согласованное) состояние, в котором она находилась, когда в последний раз было выполнено последнее резервное копирование журнала транзакций.

При восстановлении базы данных с применением полной резервной копии компонент Database Engine сначала воссоздает все файлы баз данных и размещает их на соответствующих физических носителях. После этого система воссоздает все объекты баз данных.

Компонент Database Engine может выполнять определенные формы восстановления в динамическом режиме (т. е. в процессе работы экземпляра базы данных). Динамическое восстановление повышает уровень доступности системы, т. к. недоступными являются только восстанавливаемые данные. Восстанавливать динамически можно или весь файл базы данных, или файловую группу. (В Microsoft динамическое восстановление называется "восстановлением в режиме онлайн" — online restore.)

Проверка резервного набора на пригодность для восстановления

После выполнения инструкции `BACKUP` заданное устройство резервного копирования (файл на диске или магнитная лента) содержит все данные выбранного для ре-

зервного копирования объекта. Эти данные называются *набором резервного копирования* (backup set). Прежде чем приступать к восстановлению базы данных с резервного набора, необходимо удостовериться в том, что он:

- содержит все данные, которые требуется восстановить;
- является пригодным для использования.

Компонент Database Engine поддерживает набор инструкций Transact-SQL, посредством которых можно проверить соответствие резервного набора этим требованиям. В набор инструкций, среди прочих, входят следующие инструкции:

- `RESTORE LABELONLY`;
- `RESTORE HEADERONLY`;
- `RESTORE FILELISTONLY`;
- `RESTORE VERIFYONLY`.

Эти инструкции подробно рассмотрены в последующем подразделе.

Инструкция *RESTORE LABELONLY*

Инструкция `RESTORE LABELONLY` применяется для отображения информации о заголовке носителя (жесткого диска или магнитной ленты) резервной копии. Выводимая в результате выполнения этой инструкции единственная строка содержит суммарную информацию о заголовке носителя (имя носителя, описание процесса резервного копирования и дату создания резервной копии).

ПРИМЕЧАНИЕ

Инструкция `RESTORE LABELONLY` считывает только файл заголовка, поэтому ее можно использовать, чтобы оперативно узнать содержимое резервного набора.

Инструкция *RESTORE HEADERONLY*

Тогда как инструкция `RESTORE LABELONLY` предоставляет краткую информацию о заголовке файла устройства резервного копирования, инструкция `RESTORE HEADERONLY` предоставляет информацию о хранящихся на этом устройстве резервных копиях. В частности, инструкция выводит одну строку суммарной информации для каждой резервной копии на устройстве резервного копирования. В отличие от инструкции `RESTORE LABELONLY` выполнение инструкции `RESTORE HEADERONLY` для носителя, содержащего несколько резервных копий, может занять достаточно длительное время.

Вывод инструкции содержит столбец `Compressed`, в котором указывается состояние сжатия соответствующего файла резервной копии. Значение `1` в этом столбце означает, что файл сжат.

Инструкция *RESTORE FILELISTONLY*

Эта инструкция возвращает результирующий набор со списком файлов баз данных и журналов транзакций, содержащихся в резервном наборе. Информация возвраща-

ется только для одного резервного набора. Поэтому если носитель содержит несколько резервных копий, необходимо указать позицию той, для которой требуется информация.

Инструкцию `RESTORE FILELISTONLY` следует использовать только в тех случаях, когда неизвестно, какие имеются резервные наборы или где находятся файлы определенного резервного набора. В обоих случаях можно проверить весь или часть носителя, чтобы получить глобальную картину о существующих резервных копиях.

Инструкция *RESTORE VERIFYONLY*

Определив местонахождение требуемой резервной копии, можно выполнять следующий шаг: проверить ее целостность, не запуская процесс восстановления с ее применением. Такую проверку можно выполнить с помощью инструкции `RESTORE VERIFYONLY`, которая проверяет наличие всех носителей резервного копирования (файлов жестких дисков или магнитных лент), а также возможность считывания находящейся на них информации.

В отличие от предшествующих трех инструкций, инструкция `RESTORE VERIFYONLY` поддерживает две особые опции:

- `LOADHISTORY` — задает добавление информации о резервной копии в таблицы историй резервного копирования;
- `STATS` — выводит сообщение после выполнения считывания очередной порции информации и используется для отслеживания масштаба процесса (значением по умолчанию является 10.)

Восстановления баз данных и журналов транзакций с помощью инструкций Transact-SQL

Все операции восстановления с резервной копии можно выполнять, используя только две следующие инструкции языка Transact-SQL:

- `RESTORE DATABASE`;
- `RESTORE LOG`.

Инструкция `RESTORE DATABASE` используется для восстановления с резервной копии базы данных. Синтаксис этой инструкции выглядит таким образом:

```
RESTORE DATABASE {db_name | @variable}
   [FROM device_list]
   [WITH option_list]
```

Здесь в параметре *db_name* инструкции указывается имя восстанавливаемой базы данных. Имя базы данных также можно задать с использованием переменной *@variable*. В параметре *device_list* указывается устройство (или устройства), на котором находится резервная копия базы данных. (Если предложение `FROM` не указывается, то выполняется только автоматическое восстановление, а не восстановление с резервной копии, и при этом нужно указать параметр `RECOVERY`, `NORECOVERY`

или STANDBY. Этот вариант выполнения инструкции можно использовать, если вы хотите переключиться на резервный сервер.) Параметр *device_list* может быть списком имен дисковых файлов или магнитных лент. В параметре *option_list* инструкции указываются опции для разных типов резервного копирования. Наиболее важными из этих опций являются следующие:

- RECOVERY/NORECOVERY/STANDBY;
- CHECKSUM/NO_CHECKSUM;
- REPLACE;
- PARTIAL;
- STOPAT;
- STOPATMARK;
- STOPBEFOREMARK.

Опция RECOVERY указывает компоненту Database Engine выполнить накат всех подтвержденных (зафиксированных) транзакций, а для всех неподтвержденных (незафиксированных) транзакций выполнить откат. После выполнения инструкции с опцией RECOVERY база данных будет находиться в согласованном состоянии и будет готова для использования. Эта опция является опцией по умолчанию.

ПРИМЕЧАНИЕ

Опция RECOVERY используется или с последней восстанавливаемой резервной копией журнала транзакций, или при восстановлении с полной резервной копии базы данных без последующих восстановлений с резервных копий журналов транзакций.

Когда указана опция NORECOVERY, компонент Database Engine не выполняет откат неподтвержденных (незафиксированных) транзакций, поскольку выполнение восстановления будет продолжаться с других резервных копий. После выполнения инструкции с опцией NORECOERY база данных будет недоступной для использования.

ПРИМЕЧАНИЕ

Опция NORECOVERY применяется при восстановлении со всеми, кроме последнего журнала транзакций.

Опция STANDBY является альтернативой параметрам RECOVERY и NORECOVERY и применяется с резервным сервером. (Резервный сервер рассматривается в *разд. "Использование резервного сервера" далее в этой главе*.) Чтобы получить доступ к данным, хранящимся на резервном сервере, вы обычно после восстановления журнала транзакций осуществляете восстановление базы данных. С другой стороны, если вы восстанавливаете базу данных на резервном сервере, то вы не можете для процесса восстановления применять дополнительные журналы с производственного сервера. В этом случае, чтобы предоставить пользователям доступ для чтения к резервному серверу, используется опция STANDBY. Кроме этого, системе разрешается восстанавливать дополнительные журналы транзакций. Указание опции STANDBY подразумевает наличие файла отмены, который используется для выполнения отката изменений после восстановления дополнительных журналов транзакций.

Опция CHECKSUM инициирует проверку как контрольной суммы резервной копии, так и контрольных сумм страниц, если эти суммы имеются. При отсутствии контрольных сумм инструкции восстановления RESTORE выполняются без их проверки. Опция NO_CHECKSUM в операции восстановления явно отключает проверку контрольных сумм.

Опция REPLACE указывает на замену существующей базы данных данными из резервной копии другой базы данных. В этом случае сначала удаляется существующая база данных, а различия в именах файлов базы данных и имени базы данных игнорируются. (Если опция REPLACE не указывается, то система базы данных выполняет безопасную проверку, которая гарантирует, что существующая база данных не будет заменена, если имена файлов базы данных или имя самой базы данных отличаются от соответствующих имен в наборе резервной копии.)

Опция PARTIAL задает операцию частичного восстановления. Эта опция позволяет восстановить часть базы данных, состоящую из ее основной файловой группы и одной или нескольких вторичных файловых групп, которые указываются в дополнительной опции FILEGROUP. (Опцию PARTIAL нельзя применять в инструкции RESTORE LOG.)

Опция STOPAT позволяет восстановить базу данных до состояния, в котором она находилась в конкретный момент времени перед сбоем, указав требуемую точку во времени. Компонент Database Engine восстанавливает все подтвержденные (зафиксированные) транзакции, которые были записаны в журнал транзакций до указанного момента времени. Чтобы восстановить базу данных до определенной точки во времени, выполните инструкцию RESTORE DATABASE с использованием предложения NORECOVERY. Затем, посредством инструкции RESTORE LOG, восстанавливаются все резервные копии журнала транзакций, указывая имя базы данных, носитель, на котором находится резервная копия для восстановления, а также предложение STOPAT. (Если резервная копия журнала транзакций не будет содержать указанной точки во времени, то база данных не будет восстановлена.)

Опции STOPATMARK и STOPBEFOREMARK задают восстановление до определенной метки. Эта тема рассматривается в *разд. "Восстановление до метки" далее в этой главе*.

Инструкция RESTORE DATABASE также применяется для восстановления базы данных с разностной резервной копии. Синтаксис и опции для восстановления с разностной резервной копии такие же, как и для восстановления с полной резервной копии базы данных. При восстановлении с разностной резервной копии компонент Database Engine восстанавливает только те данные, которые изменились после последнего полного резервного копирования. Следственно, прежде чем восстанавливать разностную резервную копию, необходимо выполнить восстановление полной резервной копии.

Инструкция RESTORE LOG применяется для восстановления резервной копии журнала транзакций. Эта инструкция имеет такие же синтаксис и параметры, как и инструкция RESTORE DATABASE.

Восстановление баз данных и журналов транзакций с помощью среды Management Studio

Чтобы восстановить полную резервную копию базы данных, разверните папку "Databases", щелкните правой кнопкой требуемую базу данных и в контекстном меню выберите последовательность команд **Tasks | Restore | Database**. Откроется диалоговое окно **Restore Database** (рис. 16.4).

На странице **General** этого диалогового окна выберите базу данных, в которую (поле **To database**) и из которой (поле **From device**) вы хотите выполнить восстановление. После этого отметьте тот набор резервной копии, который нужно использовать для процесса восстановления.

ПРИМЕЧАНИЕ

Если вы осуществляете восстановление из резервной копии журнала транзакций, не забудьте о правильной последовательности выполнения восстановления разных типов резервных копий. Сначала восстанавливается полная резервная копия базы данных, а затем — все соответствующие резервные копии журнала транзакций в порядке их создания.

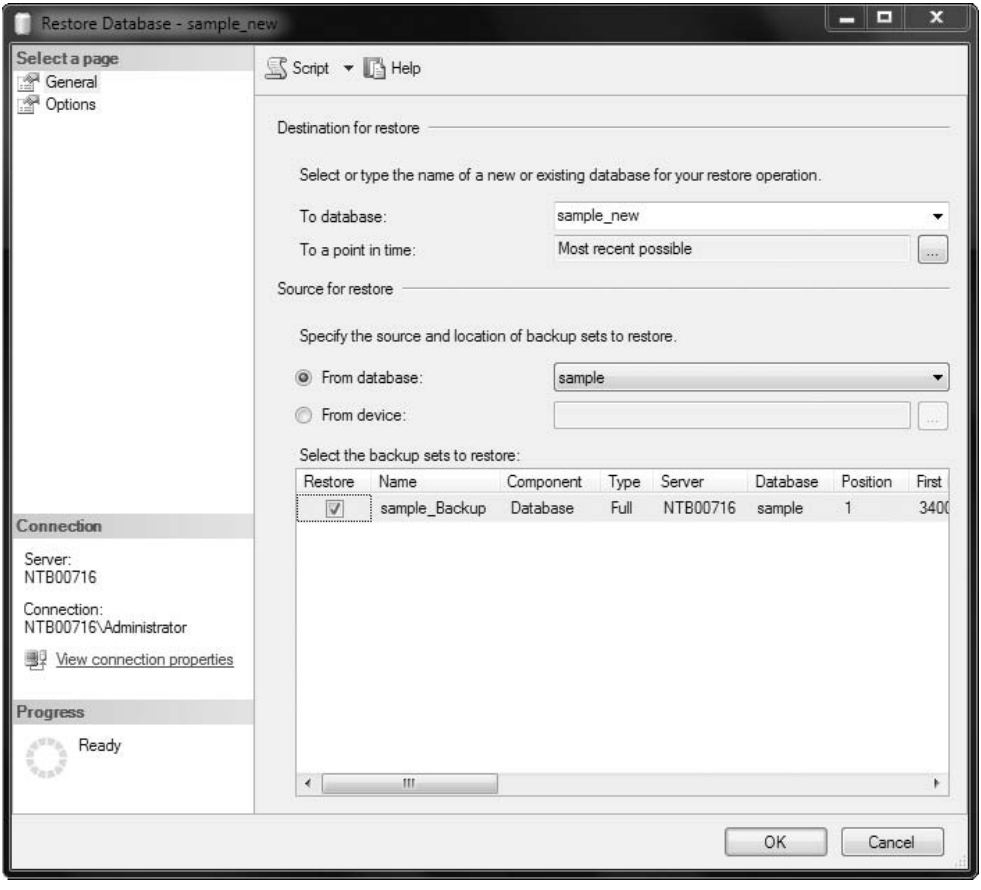


Рис. 16.4. Диалоговое окно **Restore Database**, страница **General**

Чтобы для восстановления базы можно было выбрать необходимые опции, в диалоговом окне **Restore Database** перейдите на страницу **Options** (рис. 16.5).

В верхней части окна выберите один или несколько типов восстановления, а в нижней части окна один из трех режимов восстановления, установив соответствующий переключатель. Выбор первого из этих переключателей **Leave the database ready to use by rolling back uncommitted transactions** указывает компоненту Database Engine выполнить накат всех подтвержденных и откат всех неподтвержденных транзакций. После применения этой опции база данных будет находиться в согласованном состоянии и готова к использованию. Эта опция эквивалентна выполнению инструкции `RESTORE DATABASE` с опцией `RECOVERY`.

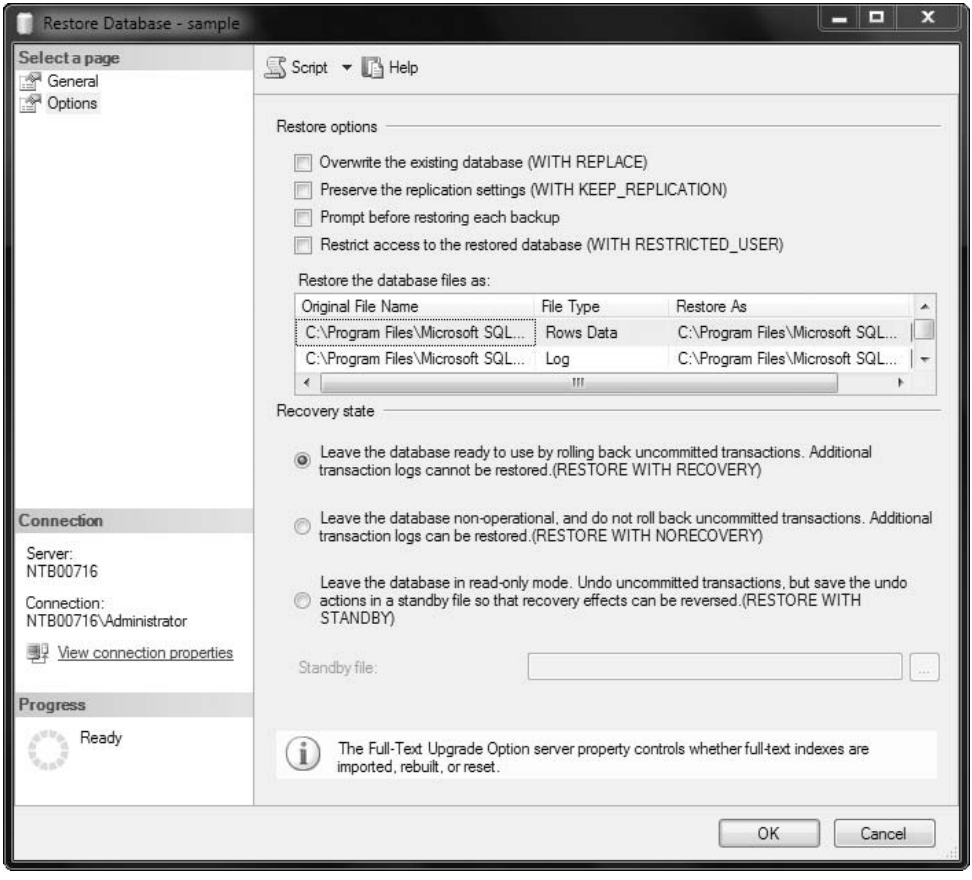


Рис. 16.5. Диалоговое окно **Restore Database**, страница **Options**

ПРИМЕЧАНИЕ

Эту опцию следует использовать при восстановлении только вместе с последней резервной копией журнала транзакций или же при восстановлении с полной резервной копии базы данных, когда не было никаких последующих резервных копий журнала транзакций.

Выбор второго переключателя **Leave the database non-operational, and do not roll back uncommitted transactions** указывает компоненту Database Engine не выполнять откат неподтвержденных транзакций, поскольку вы будете осуществлять восстановление с помощью последующих резервных копий. После применения этой опции база данных будет недоступна для использования, поскольку нужно будет восстановить дополнительные резервные копии журнала транзакций. Эта опция эквивалентна выполнению инструкции `RESTORE DATABASE` с опцией `NORECOVERY`.

ПРИМЕЧАНИЕ

Эта опция используется с восстановлением всех, кроме последней, резервных копий журнала транзакций или с восстановлением разностной резервной копии базы данных.

При выборе третьего переключателя **Leave the database in read-only mode** необходимо указать файл (в поле **Standby file**), который будет использоваться для отката восстановлений. Эта опция эквивалентна выполнению инструкции `RESTORE DATABASE` с опцией `STANDBY`.

Процесс восстановления базы данных из разностной резервной копии эквивалентен процессу восстановления из полной резервной копии базы данных. В этом случае в диалоговом окне **Database Backup** в качестве типа резервной копии нужно выбрать разностную резервную копию (**Differential**). Единственным отличием между восстановлением с помощью полной резервной копии базы данных и восстановлением с помощью разностной резервной копии является то, что для второго типа восстановления можно выбрать только первый из трех переключателей в нижней половине страницы **Options**, т. е. переключатель **Leave the database ready to use by rolling back uncommitted transactions**.

ПРИМЕЧАНИЕ

Если вы выполняете восстановление из разностной резервной копии, вначале необходимо выполнить восстановление из полной резервной копии базы данных, прежде чем осуществлять восстановление соответствующей разностной резервной копии. В отличие от восстановления с использованием резервных копий журнала транзакций, восстанавливается только последняя разностная копия, поскольку она содержит все изменения, выполненные в базе данных после последнего полного резервного копирования.

Чтобы восстановить базу данных с новым именем, разверните в обозревателе объектов папку "Databases", щелкните правой кнопкой требуемую базу данных и в контекстном меню выберите последовательность команд **Tasks | Restore | Database**. На странице **General** диалогового окна **Restore Database** в раскрывающемся списке **To database** введите или выберите имя базы данных, которую нужно создать, а в списке **From database** выберите имя базы данных, чья резервная копия используется для восстановления.

Восстановление до метки

Компонент Database Engine позволяет использовать журнал транзакций для выполнения восстановления до определенной метки. Метки в журнале транзакций соответствуют определенной транзакции и вставляются только в том случае, если транзакция фиксируется. Это позволяет связать метки с определенным объемом выполненной работы и предоставляет возможность восстановления базы данных до точки, которая включает или исключает эту работу.

ПРИМЕЧАНИЕ

Если помеченная транзакция охватывает несколько баз данных на одном и том же сервере баз данных, метки вставляются в журналы всех затронутых баз данных.

Метки в журналы транзакций вставляются, используя с инструкцией `BEGIN TRANSACTION` предложение `WITH MARK`. Поскольку имя метки является тем же самым, что и имя ее транзакции, поэтому требуется задание имени транзакции. (Опция `description` задает текстовое описание метки.)

В журнале транзакций записывается информация об имени метки, ее описание, база данных, пользователь, дата и время, а также порядковый номер транзакции в журнале (LSN — log sequence number). Для возможности повторного использования не требуется уникальность имени транзакции. Информация о дате и времени вместе с именем используется для однозначной идентификации метки.

Восстановление базы данных до определенной метки можно выполнить с помощью инструкции `RESTORE LOG`, используя с ней предложение `STOPATMARK` или `STOPBEFOREMARK`. Предложение `STOPATMARK` указывает процессу восстановления, что нужно выполнить накат до метки и включить транзакцию, содержащую метку. При использовании же предложения `STOPBEFOREMARK` транзакция, содержащая метку, не включается в процесс восстановления.

Оба эти предложения поддерживают параметр `AFTER` *`datetime`*. Если этот параметр не указан, то восстановление останавливается на первой метке с указанным именем. Если же этот параметр указан, то восстановление останавливается на первой метке с указанным именем и с временем, равным или большим, чем указано в *`datetime`*.

Восстановление базы данных *master*

Разрушение данных системной базы данных `master` может иметь губительные последствия для всей системы, поскольку она содержит все системные таблицы, необходимые для работы системы. Процесс восстановления системной базы данных `master` существенно отличается от восстановления пользовательских баз данных.

Повреждение базы данных `master` проявляется разными неполадками, включая следующие:

- невозможность запустить процесс `MSSQLSERVER`;
- ошибки ввода/вывода;
- выполнение команды `DBCC` указывает на такое повреждение.

Существует два разных способа восстановления базы данных master. Более легкий способ, который доступен, только если возможно запустить на выполнение вашу систему базы данных. В этом случае вы просто восстанавливаете базу данных master из полной резервной копии базы данных. Если же систему баз данных запустить нельзя, то остается применить более трудный способ и использовать команду sqlservr.

Для восстановления вашей базы данных master запустите ваш экземпляр сервера в однопользовательском режиме. Лучшим из двух способов моим любимым является использование в командной строке команду sqlservr с опцией -m. Хотя способ восстановления посредством команды sqlservr является более трудным, в большинстве случаев он позволяет восстановить базу данных master. Далее выполняется восстановление базы данных master вместе со всеми другими базами данных, используя последнюю полную резервную копию базы данных.

ПРИМЕЧАНИЕ

Если база данных master подверглась изменениям после последнего полного резервного копирования, то эти изменения вам нужно будет восстановить вручную.

Восстановление других системных баз данных

Процесс восстановления всех других системных баз данных, отличных от базы данных master, аналогичен. Поэтому этот процесс будет рассмотрен на примере базы данных msdb. Восстановление базы данных msdb требуется в случае ее повреждения или после восстановления базы данных master. В случае повреждения базы данных msdb ее следует восстановить, используя существующие резервные копии. Если после создания резервной копии базы данных msdb она подверглась каким-либо изменениям, то эти изменения необходимо воссоздать вручную. (Системная база данных msdb рассмотрена в *главе 15*.)

ПРИМЕЧАНИЕ

Базу данных, к которой обращаются пользователи, восстанавливать нельзя. Поэтому при восстановлении базы данных msdb необходимо остановить службу SQL Server Agent, поскольку она использует эту базу данных.

Модели восстановления

Модель восстановления позволяет вам управлять тем объемом данных, которые вы рискуете потерять в подтвержденных (зафиксированных) транзакциях в случае повреждения базы данных. Она также определяет размер резервных копий журнала и скорость их создания. Кроме этого, выбор модели восстановления влияет на размер журнала транзакций и, следовательно, на объем времени, требуемого для создания резервной копии журнала.

Компонент Database Engine поддерживает следующие модели восстановления:

- модель полного восстановления;
- модель восстановления с неполным протоколированием;
- простую модель восстановления.

Эти модели восстановления рассмотрены далее в последующих разделах.

Модель полного восстановления

В модели полного восстановления (full recovery model) все операции записываются в журнал транзакций. Поэтому эта модель предоставляют полную защиту от потери данных. Это означает, что базу данных можно будет восстановить вплоть до последней подтвержденной (зафиксированной) транзакции, которая была сохранена в журнале транзакций. Кроме этого, данные можно восстановить до любого момента во времени, предшествующего точке сбоя. Чтобы обеспечить такой режим восстановления, полностью протоколируются такие операции, как выполнение инструкции `SELECT INTO` и утилиты `bcp`.

Кроме восстановления до определенного момента во времени, модель полного восстановления позволяет выполнить восстановление до метки в журнале транзакций. Метки в журнале транзакций соответствуют определенной транзакции и вставляются только в том случае, если эта транзакция фиксируется.

Модель полного восстановления также предусматривает протоколирование всех операций, связанных с инструкцией `CREATE INDEX`, подразумевая, что процесс восстановления данных теперь включает и создание индексов. Таким способом воссоздание индексов выполняется быстрее, поскольку их не требуется воссоздавать отдельно.

Недостатком этой модели восстановления является то, что соответствующий журнал транзакций может быть очень большого объема, и файлы на диске, содержащие этот журнал транзакций, могут быть очень быстро заполнены. Кроме этого, для создания резервной копии журнала транзакции такого большого объема потребуется значительно больше времени.

ПРИМЕЧАНИЕ

Если вы используете модель полного восстановления, то журнал транзакций необходимо предохранить от возможных повреждений носителя. По этой причине для хранения журнала транзакций рекомендуется использовать дисковый массив RAID 1. (Дисковый массив RAID 1 обсуждается в *разд. "Применение технологии RAID" далее в этой главе*.)

Модель восстановления с неполным протоколированием

Модель восстановления с неполным протоколированием (bulk-logged recovery model) уменьшает место, занимаемое журналами, за счет неполного протоколирования большинства групповых операций.

Протоколирование следующих далее инструкций сведено к минимуму и неуправляемо для отдельных операций:

- `SELECT INTO`;
- `CREATE INDEX` (включая индексированные представления);
- утилита `bcp` и инструкция `BULK INSERT`.

Хотя групповые операции протоколируются не полностью, выполнять полное резервное копирование базы данных после таких операций не требуется, т. к. резервные копии журнала транзакции, применяемые для восстановления с неполным протоколированием, содержат как запись групповой операции, так и ее результаты. Это упрощает переход от модели полного восстановления к модели восстановления с неполным протоколированием.

Модель восстановления с неполным протоколированием позволяет восстановить базу данных до конца резервной копии журнала транзакций (т. е. до последней подтвержденной транзакции). Кроме этого, если не выполнялось никаких объемных операций, то можно восстанавливать вашу базу данных на любой момент времени. То же самое относится и к операции восстановления до именованной метки в журнале транзакций.

Преимуществом модели восстановления с неполным протоколированием является то, что объемные операции выполняются намного быстрее, чем в модели полного восстановления, поскольку для них не ведется полное протоколирование. С другой стороны, компонент Database Engine создает резервные копии всех подвергшихся изменениям экстентов, включая и сам журнал. Поэтому для резервной копии журнала требуется намного больше места, чем в случае с моделью полного восстановления. Время восстановления с резервной копии журнала также значительно увеличивается.

Простая модель восстановления

В простой модели восстановления журнал транзакций усекается на каждой контрольной точке. Поэтому поврежденную базу данных можно восстановить, только используя полную или разностную резервную копию, поскольку для них не требуются резервные копии журнала транзакций. Стратегия восстановления базы данных с резервной копии для этой модели очень проста: база данных восстанавливается, используя существующую полную резервную копию базы данных и, если имеются разностные резервные копии, восстанавливая самую последнюю из них.

ПРИМЕЧАНИЕ

Использование простой модели восстановления не означает, что протоколирование не ведется совсем. Содержимое журнала транзакций не применяется для целей восстановления, но применяется в контрольных точках, когда все транзакции фиксируются в журнале или выполняется откат.

Преимущество простой модели восстановления состоит в высокой производительности объемных операций и в невысоких требованиях к объему памяти для журна-

ла транзакций. С другой стороны, эта модель требует больше ручной работы, поскольку все изменения после самого последнего полного (или разностного) резервного копирования нужно восстанавливать вручную. Эта модель восстановления не допускает восстановлений до конкретного момента во времени и восстановлений страниц. Кроме этого, восстановление файлов возможно только для вспомогательных файловых групп с доступом только для чтения.

ПРИМЕЧАНИЕ

Простую модель восстановления не следует использовать для производственных баз данных.

Изменение и редактирование модели восстановления

Изменить одну модель восстановления на другую можно с помощью опции RECOVERY в инструкции ALTER DATABASE. Часть синтаксиса инструкции ALTER DATABASE, связанная с моделями восстановления, имеет следующий вид:

```
SET RECOVERY [FULL | BULK_LOGGED | SIMPLE]
```

Существует два способа редактирования текущей модели восстановления базы данных:

- используя функцию свойств databasepropertyx;
- используя представление каталога sys.databases.

Чтобы отобразить текущую модель восстановления для определенной базы данных посредством функции databasepropertyex, при вызове этой функции в первом аргументе ей передается имя требуемой базы данных, а во втором — значение 'recovery'. В примере 16.2 показано использование функции свойств databasepropertyx для отображения модели восстановления базы данных sample. (В качестве результата функция возвращает одно из значений FULL, BULK_LOGGED или SIMPLE.)

Пример 16.2. Отображение текущей модели базы данных посредством функции свойств databasepropertyex

```
SELECT databasepropertyex('sample', 'recovery')
```

Представление каталога sys.databases возвращает в столбце model ту же самую информацию, что и функция свойств databasepropertyex, как это можно видеть в результатах выполнения примера 16.3.

Пример 16.3. Отображение модели восстановления посредством представления каталога sys.databases

```
SELECT name, database_id, recovery_model_desc AS model
   FROM sys.databases
   WHERE name = 'sample'
```

Результат выполнения этого запроса:

Name	database_id	model
Sample	7	FULL

Доступность системы

Обеспечение доступности системы баз данных и собственно баз данных является одной из наиболее важных задач. Для решения этой задачи применяется несколько методов, которые можно разделить на две основные группы: те, которые входят в состав компонента Database Engine, и те, которые не являются таковыми. К методам обеспечения доступности системы, которые не входят в состав компонента Database Engine, относятся следующие:

- использование резервного сервера;
- использование технологии RAID.

Следующие технологии предоставляются системой управления базами данных:

- отказоустойчивая кластеризация;
- зеркальное отображение базы данных;
- пересылка журналов транзакций;
- высокий уровень доступности и восстановления в аварийных ситуациях (HARD);
- репликация.

Эти технологии рассматриваются в последующих разделах (за исключением репликации, которая рассмотрена в *главе 18*).

Использование резервного сервера

Резервный сервер (standby server) представляет собой именно то, что и подразумевается его названием — сервер, находящийся в резерве, на случай если что-либо случится с рабочим производственным сервером (также называемым *первичным сервером*). Файлы, базы данных (системные и пользовательские) и учетные записи пользователей на резервном сервере идентичны этим элементам на производственном сервере.

Реализация резервного сервера осуществляется сначала восстанавливая на нем полную резервную копию базы данных, а затем последовательно восстанавливая резервные копии журнала транзакций производственного сервера, чтобы содержать всю информацию резервного сервера синхронизированной с информацией основного производственного сервера. Для установки резервного сервера присвойте опции баз данных `read only` значение `TRUE`. Эта опция запрещает пользователям любые операции записи в базу данных.

Общая процедура установки копии производственной базы данных на резервном сервере состоит из следующих шагов:

- восстанавливается рабочая база данных, используя инструкцию `RESTORE DATABASE` с предложением `STANDBY`;
- восстанавливаются все, кроме последней, резервные копии журнала транзакций, используя инструкцию `RESTORE LOG` с предложением `STANDBY`;
- восстанавливается последняя резервная копия журнала транзакций, используя инструкцию `RESTORE LOG` с предложением `RECOVERY`. Последняя операция воссоздает базу данных, не создавая при этом файла с прообразами, делая базу данных доступной также и для операций записи.

После восстановления базы данных и журналов транзакций пользователи получают для работы точную копию производственной базы данных. Потеряны будут только транзакции, которые не были зафиксированы при сбое.

ПРИМЕЧАНИЕ

В случае сбоя производственного сервера, автоматического переноса пользовательских процессов на резервный сервер не происходит. Кроме этого, всем пользовательским процессам следует перезапустить все те задачи, транзакции которых не были зафиксированы вследствие сбоя основного производственного сервера.

Использование технологии RAID

Массив дисков *RAID* (Redundant Array of Independent Disks, избыточный массив независимых жестких дисков) представляет собой специальную конфигурацию дисков, в которой несколько приводов дисков составляют единую логическую единицу. Этот способ позволяет расположить каждый файл на нескольких физически отдельных дисках. Иными словами, хотя части файла находятся на разных дисках, система видит этот файл как единое целое. Технология RAID позволяет повысить надежность за счет понижения производительности. Существует шесть основных уровней RAID, от 0 по 5. Только три из этих уровней представляют важность для систем баз данных: уровни 0, 1 и 5.

Система RAID может быть реализована аппаратным или программным образом. Стоимость аппаратной системы RAID более высокая (вследствие необходимости приобретения дополнительных контроллеров дисков), но обычно имеет лучшую производительность. Программная система RAID обычно поддерживается операционной системой. Операционная система Windows поддерживает уровни RAID 0, 1 и 5. Технология RAID оказывает влияние на следующие свойства:

- устойчивость к ошибкам (отказоустойчивость);
- производительность.

Преимущества и недостатки каждого уровня RAID по отношению к этим двум свойствам рассматриваются далее в последующем материале.

Технология RAID предоставляет защиту от сбоев жестких дисков и сопутствующей этим сбоям потери данных тремя способами: чередованием дисков (так же называемым *расслоением дисков*), зеркалированием дисков и контролем по четности. Эти три способа соответствуют уровням RAID 0, 1 и 5 соответственно.

Расслоение дисков. RAID 0

RAID 0 определяет чередование (расслоение) дисков без контроля по четности. При использовании RAID 0 данные разбиваются на несколько блоков, которые записываются одновременно на несколько дисков, что позволяет повысить скорость записи и чтения данных. Вследствие этого RAID 0 является самой быстрой конфигурацией RAID. Недостатком чередования данных на нескольких дисках является полное отсутствие отказоустойчивости: при сбое одного из дисков данные на всех дисках массива становятся недоступными.

Зеркалирование. RAID 1

Зеркалирование представляет собой особую форму расслоения дисков, когда на каждый из дисков записываются идентичные данные. RAID 1 определяет зеркалирование дисков и предохраняет данные от потери вследствие сбоя носителя, помещая копию базы данных (или ее части) на другой диск. Если один из массивов зеркальных дисков выйдет из строя, его файлы можно с легкостью восстановить по их копиям на другом диске массива. Аппаратная реализация зеркалирования дисков имеет более высокую стоимость, но и более высокую скорость доступа к данным. (Кроме этого, аппаратные реализации зеркалирования предоставляют некоторые возможности кэширования, что также повышает пропускную способность массива.) Преимуществом зеркалирования на основе программного решения Windows является то, что оно позволяет зеркалировать разделы диска, тогда как аппаратные решения обычно зеркалируют весь диск.

По сравнению с RAID 0, скорость доступа к данным в RAID 1 намного ниже, но его надежность выше. Кроме этого, стоимость системы RAID 1 выше, чем системы RAID 0, поскольку объем данных одного диска в данном случае сохраняется на двух дисках. Система остается работоспособной при выходе из строя одного диска и может выдержать выход из строя до половины зеркальных дисков, не требуя выключения сервера и восстановления файлов из резервной копии. (RAID 1 предоставляет наилучшую конфигурацию RAID в плане отказоустойчивости.)

Зеркалирование также влияет на производительность операций чтения и записи. В частности, зеркалирование понижает уровень производительности операций записи, поскольку для каждой такой операции необходимо выполнить две дисковые операции ввода/вывода — одну на основной диск, а вторую на зеркальный. С другой стороны, уровень производительности операций чтения повышается, т. к. система может считывать данные с любого диска, в зависимости от того, который из них наименее занят в данный момент.

Контроль по четности. RAID 5

Массив RAID с контролем по четности (parity check), т. е. RAID 5, реализуется, вычисляя контрольную сумму записываемых данных и распределяя блоки данных вместе с информацией контрольной суммы на разные диски массива. В случае сбоя одного из дисков массива, он заменяется новым, а информация на нем восстанавливается по контрольным суммам, записанным на другие диски массива.

Преимущество массива RAID 5 состоит в необходимости иметь только один добавочный диск в дополнение к любому количеству дисков, используемых в обычной конфигурации. Недостатком этой конфигурации является пониженная производительность и отказоустойчивость. Для вычисления и записи информации восстановления требуются дополнительные операции дискового ввода/вывода. (Затраты операций ввода/вывода одинаковые как для зеркальных массивов RAID, так и для массивов с контролем по четности.) Кроме этого, массив RAID 5 допускает выход из строя только одного диска без необходимости выводить массив из работы и восстанавливать данные на нем с резервной копии. Иными словами, выход из строя одновременно двух дисков выводит из строя весь массив, сколько бы дисков в нем не имелось. Так как расслоение дисков с контролем по четности (чем RAID 5 и является) требует дополнительных затрат на вычисление и запись информации восстановления, для RAID 5 требуется четыре дисковых операций ввода/вывода, тогда как для RAID 0 только одна, а для RAID 1 две таких операции.

Зеркальное отображение базы данных

Как уже упоминалось, зеркалирование можно реализовать с помощью аппаратных или программных средств. Преимуществом зеркалирования на основе программного решения является то, что оно позволяет зеркалировать разделы диска, тогда как аппаратные решения обычно зеркалируют весь диск. В этом разделе рассматривается решение Windows для зеркального отображения базы данных и как его можно организовать.

Для зеркального отображения базы данных используются два сервера, и база данных с одного из них будет зеркалироваться на другой. Первый из этих серверов называется *главным* (основным), а второй — *зеркальным* (дублирующим). (Копия базы данных на зеркальном сервере называется *зеркальной базой данных*.)

Зеркальное отображение базы данных разрешает выполнять непрерывную передачу журнала транзакций с главного сервера на зеркальный. Копия операций журнала транзакций записывается в журнал транзакций зеркальной базы данных, и в ней выполняются эти операции. Если главный сервер выходит из строя, то приложения могут переключиться на базу данных на зеркальном сервере, не ожидая завершения процесса ее восстановления. В отличие от отказоустойчивой кластеризации, зеркальный сервер полностью кэширован и готов к работе, будучи синхронизован с главным сервером. Можно реализовать до четырех зеркальных резервных наборов. (Для реализации зеркалирования используется опция `MIRROR TO` в инструкции `BACKUP DATABASE` или в инструкции `BACKUP LOG`.)

При зеркальном отображении базы данных также применяется третий сервер, называющийся *следящим сервером* (witness server). Этот сервер определяет, который

из первых двух серверов является главным, а который зеркальным. Этот сервер применяется только для поддержки автоматического перехода на другой ресурс в случае сбоя первого. (Чтобы разрешить автоматическое переключение, необходимо включить режим синхронной работы, присвоив опции `SAFETY` в инструкции `ALTER DATABASE` значение `FULL`.)

Другим вопросом производительности при зеркальном отображении базы данных является возможность автоматического сжатия отправляемых на зеркальный сервер данных. Если существует возможность получить, по крайней мере, коэффициент сжатия в 12,5%, компонент Database Engine сжимает потоковые данные. Таким образом система уменьшает издержки на передачу данных журнала транзакций с главного сервера на зеркальные.

Отказоустойчивая кластеризация

Отказоустойчивая кластеризация представляет собой процесс, в котором операционная система и система баз данных работают совместно для обеспечения доступности данных в случае сбоя. Отказоустойчивый кластер состоит из группы резервных серверов, называющихся узлами, которые используют общую внешнюю дисковую систему. При сбое одного из узлов кластера экземпляр компонента Database Engine на этой машине выключается, а служба *Microsoft Cluster Service* автоматически перемещает ресурсы с неисправной машины на целевой узел, который имеет идентичную конфигурацию. Процесс перемещения ресурсов с одного узла на другой в кластер происходит с достаточно высокой скоростью.

Преимущество отказоустойчивой кластеризации заключается в том, что она предохраняет систему от аппаратных сбоев, предоставляя механизм автоматического перезапуска системы базы данных на другом узле кластера. С другой стороны, эта система имеет единственную критическую точку, которой является набор дисков, совместно используемых узлами кластера, и не защищает от ошибок в данных. Другим недостатком этой технологии является то, что она не повышает уровень производительности или масштабирования. Иными словами, приложение не может осуществлять масштабирование до возможностей кластера, а только в пределах одного узла.

Таким образом, отказоустойчивая кластеризация предоставляет серверную избыточность, но не избыточность данных. С другой стороны, зеркальное отображение базы данных не предоставляет серверной избыточности, но предоставляет избыточность как баз данных, так и файлов данных.

Доставка журналов транзакций

Доставка (пересылка) журналов транзакций заключается в отправке и использовании в постоянном режиме журналов транзакций с одной базы данных на другую. Это позволяет иметь горячий резервный сервер, а также предоставляет способ выгрузки данных с исходной машины на другие компьютеры-получатели, на которых разрешено только чтение данных. База данных-получатель является точной копией первичной базы данных, поскольку она получает с нее все изменения. Базу данных-

получатель можно сделать новой первичной базой данных, если первичный сервер, который содержит исходную базу данных, выйдет из строя. Когда первичный сервер снова станет доступным, роли серверов можно опять поменять на исходные.

Технология доставки (пересылки) журналов транзакций не поддерживает автоматическую обработку сбоев. Поэтому при сбое сервера базы данных-источника вы должны восстановить базу данных-получатель самостоятельно, вручную или с помощью специального программного кода.

Таким образом, функциональность доставка журналов подобно зеркальному отображению базы данных в том отношении, что она обеспечивает избыточность базы данных. С другой стороны, зеркальное отображение базы данных значительно расширяет возможности функциональности доставки журналов, поскольку она позволяет обновлять базу данных-получатель через прямое соединение в режиме реального времени.

Высокий уровень доступности и восстановления в аварийных ситуациях (HARD)

Зеркальное отображение базы данных как технология достижения высокого уровня доступности имеет несколько недостатков, включая следующие:

- запросы только для чтения не могут выполняться на зеркальной базе данных;
- эту технологию можно применять только на двух экземплярах SQL Server;
- эта технология зеркалирует только объекты внутри базы данных, но объекты такие, как регистрационные имена входа в систему не могут быть защищены с использованием зеркалирования.

С целью устранения этих недостатков зеркалирования базы данных, в SQL Server 2012 вводится новая технология, называемая высоким уровнем доступности и восстановления в аварийных ситуациях (HARD — high availability and disaster recovery). Эта технология позволяет довести до максимума доступность баз данных. Но прежде чем приступить к ее объяснению необходимо рассмотреть концепт групп, реплик и режимов обеспечения доступности.

Группы, реплики и режимы обеспечения доступности

Группа обеспечения доступности (availability group) состоит из набора резервных серверов, которые называются *репликами обеспечения доступности* (availability replicas). Каждая реплика обеспечения доступности содержит локальную копию каждой базы данных в группе обеспечения доступности. На одной из этих реплик, которая называется *первичной репликой*, содержится основная копия каждой базы данных. Первичная реплика делает эти базы данных, называемые *первичными базами данных*, доступными пользователям для операций чтения и записи. Для каждой первичной базы данных другая реплика обеспечения доступности, которая называется *вторичной репликой*, содержит резервную копию базы данных, называемую *вторичной базой данных*.

Реплики обеспечения доступности могут содержаться только на экземплярах SQL Server 2012, находящихся на узлах WSFC (Windows Server Failover Clustering, отказоустойчивая кластеризация Windows Server). Экземпляры сервера SQL Server могут быть или экземплярами отказоустойчивого кластера, или автономными экземплярами. Экземпляр сервера, на котором находится первичная реплика, называется *первичным размещением* (primary location), а экземпляр, на котором находится вторичная реплика, называется *вторичным размещением* (secondary location). Экземпляры, содержащие реплики обеспечения доступности для данной группы обеспечения доступности, должны находиться на отдельных узлах WSFC.

Режим обеспечения доступности (availability mode) — это свойство, которое устанавливается отдельно для каждой реплики обеспечения доступности. Режим обеспечения доступности вторичной реплики определяет, дожидается ли первичная реплика завершения записи вторичной репликой записей в соответствующие журналы транзакций, прежде чем зафиксировать транзакции на диск.

Каждой реплике группы обеспечения доступности присваивается одна из следующих ролей:

- первичная роль;
- вторичная роль;
- определяющаяся роль.

Текущая первичная реплика имеет первичную роль. (В любой данный момент только одна реплика может иметь эту роль.) Каждая вторичная реплика имеет вторичную роль. Определяющаяся (resolving) роль означает, что текущее состояние реплики обеспечения доступности находится в процессе изменения.

В пределах сеанса первичная и вторичная роль могут меняться; этот процесс называется *переключением ролей* (role switching). Переключение ролей включает в себя обработку отказа, которая передает первичную роль вторичной реплике. Этот процесс перемещает роль вторичной реплики на первичную и наоборот. База данных новой первичной реплики становится новой первичной базой данных. Когда бывшая первичная реплика снова становится доступной, ее база данных становится вторичной базой данных.

Конфигурация технологии HADR

Конфигурация технологии HADR очень сложная, поскольку сначала нужно создать двухузловой кластер, описание которого выходит за рамки тематики этой книги. Поэтому приводится только краткое описание требуемых операций.

Для конфигурирования технологии HADR необходимо выполнить следующую последовательность шагов (в указанном порядке):

1. Установить экземпляры баз данных на обоих узлах.
2. Разрешить возможность HADR на обоих экземплярах. Выбрать последовательность команд **SQL Server Configuration Manager | SQL Server Services**, щелкнуть правой кнопкой требуемый экземпляр и в контекстном меню выбрать

опцию **Properties**. В открывшемся окне свойств экземпляра выбрать вкладку **SQL HADR** и на ней установить флажок **Enable SQL HADR Service**.

3. В первичном экземпляре создать группу обеспечения доступности. В обозревателе объектов среды Management Studio развернуть папку "Management", щелкнуть правой кнопкой узел "Availability Groups" и в контекстном меню выбрать опцию **New Availability Group**.
4. Запустить синхронизацию данных, нажав на странице **Results** диалогового окна **New Availability Group** кнопку **Start Data Synchronization**.
5. Протестировать группы обеспечения доступности, создав таблицу на первичной реплике (посредством инструкции `CREATE TABLE` с предложением `ON PRIMARY`) и вставив в нее несколько строк.
6. Протестировать обработку отказа. Для этого развернуть последовательность узлов **Management | Availability Groups | AVG | Availability Replicas**, щелкнуть правой кнопкой вторичную реплику и выбрать Force Failover. После этого первичная и вторичная реплики должны поменяться ролями.

ПРИМЕЧАНИЕ

Подробное описание возможности технологии HADR см. в *электронной документации* и на следующих веб-сайтах: **www.brentozar.com/archive/2010/11/sql-server-denali-database-mimring-rocks/** и **www.7388.info/index.php/article/sql/2011-01-17/5235.html**.

Мастер плана обслуживания

Мастер плана обслуживания Maintenance Plan Wizard определяет набор основных задач, требуемых для содержания базы данных в исправном состоянии. Он обеспечивает качественную работу базы данных, регулярное создание ее резервной копии и отсутствие противоречивости данных.

ПРИМЕЧАНИЕ

Для создания планов обслуживания необходимо быть членом предопределенной роли сервера `sysadmin`.

Чтобы запустить Maintenance Plan Wizard, в обозревателе объектов разверните узел **Management**, щелкните в нем правой кнопкой узел **Maintenance Plans** и в контекстном меню выберите опцию **Maintenance Plan Wizard**. На первой странице мастера Maintenance Plan Wizard перечислены следующие задачи администрирования, которые можно выполнять с его помощью:

- проверять целостность баз данных;
- выполнять обслуживание индексов;
- обновлять статистику баз данных;
- создавать резервные копии баз данных.

ПРИМЕЧАНИЕ

Из всех перечисленных задач будет рассмотрено только создание резервных копий базы данных. Все остальные задачи могут быть выполнены подобным образом.

Нажмите кнопку **Next** и на следующей странице мастера (рис. 16.6) можно выбрать свойства создаваемого плана, ввести его имя и, по желанию, дать его описание.

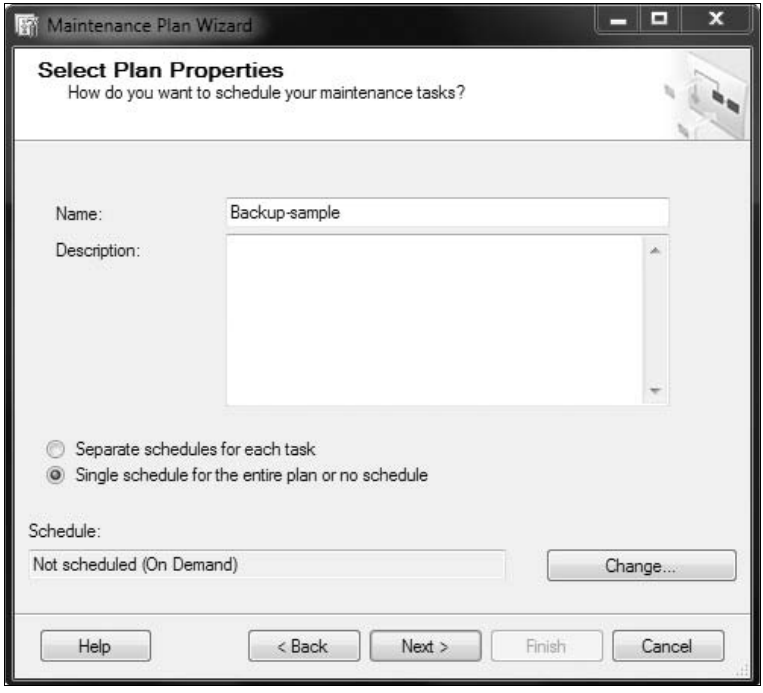


Рис. 16.6. Страница **Select Plan Properties** мастера планов обслуживания **Maintenance Plan Wizard**

Кроме этого, можно выбрать отдельное расписание для каждой задачи или одно общее расписание для всего плана. В данном примере будет запланировано создание резервной копии базы данных `sample`, поэтому присвоим плану имя "Backup-sample" и выберем переключатель **Single schedule for the entire plan or no schedule**. В поле **Schedule** можно создать расписание для исполнения плана, нажав кнопку **Change** справа от него, или же оставить опцию по умолчанию — выполнение плана по требованию. (Создание расписания выполнения плана подробно рассматривается в *главе 17*. В данном примере в этом поле оставьте значение по умолчанию — **Not scheduled (On Demand)**.)

Нажмите кнопку **Next**, чтобы перейти на страницу списка задач, содержащего, среди прочих, задачу создания полной или разностной резервной копии базы данных или резервной копии журнала транзакций. (Описание этих типов резервных копий см. в *разд. "Введение в методы резервного копирования" в начале этой главы*.) Установите флажок **Back Up Database (Full)**, а затем, чтобы перейти на страницу

выбора порядка выполнения задачи обслуживания **Select Maintenance Task Order**, нажмите кнопку **Next**. На этой странице задачи можно перемещать вверх или вниз по списку, устанавливая таким образом порядок их выполнения. Так как в данном случае у нас имеется всего лишь одна задача, то возможен лишь один порядок ее выполнения. Поэтому нажмите кнопку **Next**.

Откроется следующее окно **Define Back Up Database (Full) Task**, в котором можно указать несколько разных параметров создания резервной копии (рис. 16.7).

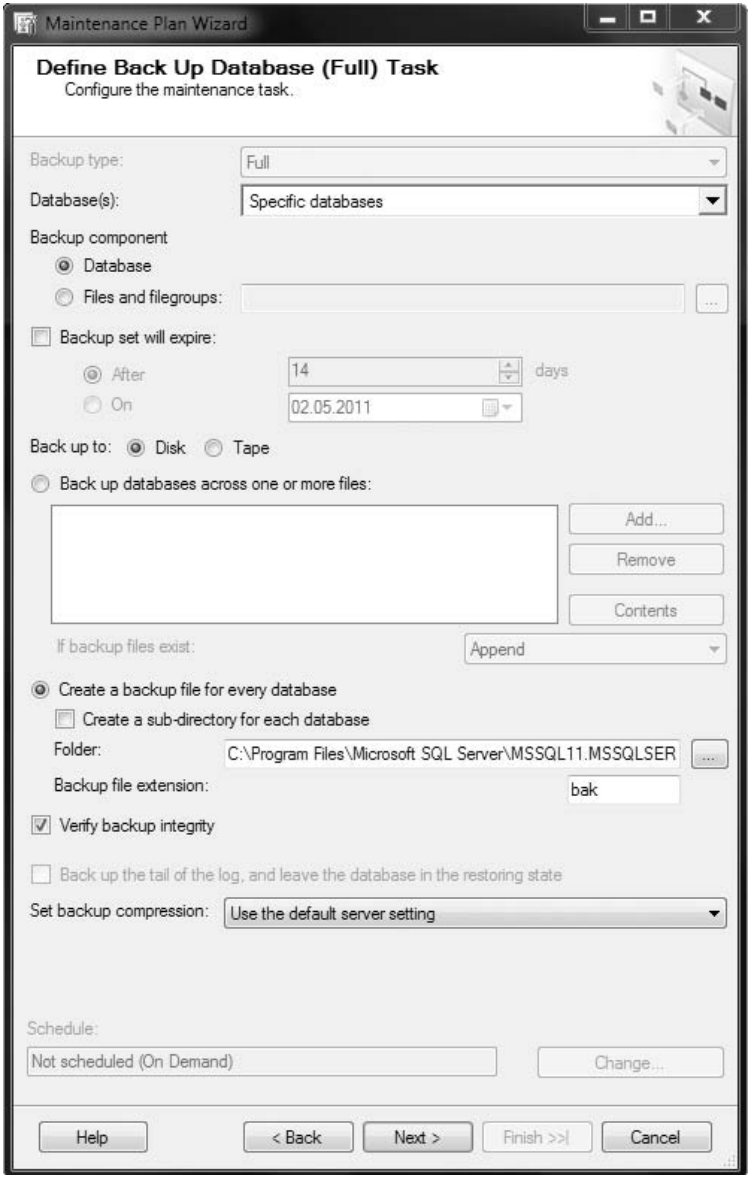


Рис. 16.7. Окно **Define Back Up Database (Full) Task** мастера **Maintenance Plan Wizard**

Сначала нужно выбрать базу данных, для которой будет выполняться создаваемая задача. Затем выбирается место хранения файлов резервной копии, включая тип носителей и их местонахождение. (Можно также указать срок действия резервного набора.)

Следующий параметр, т. е. переключатель **Create a backup file for every database**, позволяет создать отдельный файл для каждой базы данных, указанной в раскрывающемся списке **Database(s)**. Установим этот переключатель, поскольку это является предпочитаемым способом создания резервной копии для нескольких баз данных. Также установим флажок **Verify backup integrity**, чтобы система проверила целостность созданных файлов резервных копий. Завершив работу на этой странице, нажмите кнопку **Next**.

Откроется страница **Select Report Options**, на которой можно выбрать создание отчета с сохранением его в файле и/или с отправкой его по электронной почте. (Отправить отчет по электронной почте можно только существующему оператору, создание которого подробно рассмотрено в *главе 17*.)

Чтобы завершить работу мастера и сохранить созданный план, нажмите кнопку **Finish**. Мастер создаст задачу и выведет отчет с результатами.

Для просмотра истории существующего плана обслуживания разверните узел **Management**, а в нем узел **Maintenance Plan**, щелкните правой кнопкой требуемый план и выберите в контекстном меню пункт **View History**. В результате для просмотра указанного плана откроется диалоговое окно **Log File Viewer**.

Резюме

Системный администратор или владелец базы данных должны периодически выполнять резервное копирование базы данных и ее журнала транзакций. Компонент Database Engine позволяет создавать два типа резервной копии базы данных: полную и разностную. Полное резервное копирование базы данных захватывает состояние базы данных на момент запуска инструкции для ее создания и сохраняет его на носитель резервной копии (файл на диске или магнитную ленту). В разностную резервную копию включаются только те части базы данных, которые были добавлены или изменены после последнего полного резервного копирования базы данных. Преимущество разностного резервного копирования состоит в том, что для его выполнения требуется меньше времени, чем для создания полной резервной копии для этой же базы данных. Система также поддерживает резервное копирование журнала транзакций на носитель резервной копии.

Компонент Database Engine выполняет автоматическое восстановление данных после каждого сбоя системы, не вызывающего повреждений носителей данных. Автоматическое восстановление также выполняется при запуске системы после ее выключения. В процесс автоматического восстановления все подтвержденные (зафиксированные) транзакции, зафиксированные в журнале транзакций, записываются в базу данных, а для всех неподтвержденных транзакций выполняется откат. В случае повреждения носителей базы данных может потребоваться восстановить

ее данные вручную с ее полной резервной копии и резервных копий журнала транзакций. При восстановлении с использованием разностной резервной копии используется только последняя разностная копия (после применения полной резервной копии). При восстановлении базы данных с использованием резервных копий журнала транзакций сначала восстанавливается полная резервная копия базы данных, а затем все резервные копии журнала транзакций в порядке их создания. После восстановления последней резервной копии журнала транзакций база данных будет в том согласованном состоянии, в котором она находилась перед созданием последней резервной копии журнала транзакций.

Компонент Database Engine предоставляет несколько собственных технологий повышения доступности системы управления базами данных и самих баз данных, а именно:

- отказоустойчивую кластеризацию;
- зеркальное отображение базы данных;
- доставку (пересылк) журналов транзакций;
- технологию высокого уровня доступности и восстановления после сбоев (HADR).

В связи с этими технологиями представляют важность следующие три вопроса:

- резервирование серверов;
- резервирование баз данных;
- резервирование файлов данных.

Резервирование серверов означает, что приложение выполняется на двух или более серверах таким образом, чтобы обеспечить отказоустойчивость. (Одним из самых важных технологий реализации резервирования серверов является кластеризация.) Резервирование базы данных означает, что для базы данных и ее приложений гарантируется отказоустойчивость. Резервирование файлов определяется подобным же образом.

Отказоустойчивая кластеризация связана с серверной избыточностью, но не избыточностью баз данных или файлов данных. Доставка журналов транзакций обеспечивает избыточность баз данных, но не избыточность серверов. Недостаток технологии доставки журналов транзакций заключается в том, что она не поддерживает автоматическую обработку отказов.

Зеркальное отображение базы данных не предусматривает серверной избыточности, но обуславливает избыточность как базы данных, так и файлов данных. Зеркальное отображение базы данных значительно расширяет возможности доставки журналов транзакций, поскольку позволяет обновлять базу данных-получатель через прямое соединение в режиме реального времени.

Технология HADR подобна зеркалированию базы данных, но также поддерживает кластеризацию. Таким образом, возможность HADR предоставляет как избыточность серверов, так и избыточность баз и файлов данных. Основной целью техно-

логии HADR является поддержка доступности баз данных, предоставляя при этом возможности восстановления после сбоя.

Мастер плана обслуживания Maintenance Plan Wizard представляет собой инструмент для создания и планирования задач по обслуживанию баз данных. Он позволяет регулярно создавать резервную копию базы данных, обеспечивая таким образом отсутствие в ней противоречивости данных. (Данный мастер рассматривается в этой главе по той причине, что он обычно используется для создания и планирования задач резервного копирования и восстановления с резервной копии.)

В следующей главе рассматриваются все возможности системы, позволяющие автоматизировать задачи администрирования системы.

Упражнения

Упражнение 16.1

Объясните разницу между разностной резервной копией и резервной копией журнала транзакций.

Упражнение 16.2

Когда следует выполнять резервное копирование базы данных?

Упражнение 16.3

Как можно выполнить разностное копирование базы данных `master`?

Упражнение 16.4

Обсудите использование разных технологий RAID применительно к отказоустойчивости базы данных и ее журнала транзакций.

Упражнение 16.5

В чем состоит разница между ручным и автоматическим восстановлением?

Упражнение 16.6

Какую инструкцию следует использовать для проверки пригодности резервной копии для восстановления, не прибегая к восстановлению с этой копии?

Упражнение 16.7

Изложите преимущества и недостатки каждой из трех моделей восстановления.

Упражнение 16.8

Изложите сходство и различие между отказоустойчивой кластеризацией, зеркальным отображением базы данных и доставкой журналов транзакций.

Глава 17

Система автоматизации задач администрирования

- **Запуск агента SQL Server Agent**
- **Создание заданий и операторов**
- **Предупреждающие сообщения**

Одним из самых важных преимуществ системы Database Engine по сравнению с другими реляционными СУБД является ее способность автоматизировать задачи администрирования, понижая таким образом стоимость ее эксплуатации. В качестве примеров наиболее важных задач, которые выполняются на регулярной основе и поэтому должны быть автоматизированы, можно назвать, среди прочих, следующие:

- резервное копирование базы данных и журнала транзакций;
- пересылка данных;
- удаление и создание индексов вновь;
- проверка целостности данных.

Все эти задачи можно автоматизировать для выполнения по регулярному расписанию. Например, задачу резервного копирования можно запланировать для выполнения каждую пятницу в 20:00, а задачу резервного копирования журнала транзакций — ежедневно в 22:00.

К компонентам Database Engine, которые используются в процессах автоматизации, можно отнести следующие:

- службу SQL Server (MSSQLSERVER);
- журнал событий приложений Windows;
- службу SQL Server Agent.

Зачем для автоматизации процессов системе Database Engine требуются эти три компонента? Для автоматизации задач администрирования, служба MSSQLSERVER

требуется для того, чтобы записывать события в журнал событий приложений Windows. Некоторые события записываются автоматически, а некоторые должны активироваться системным администратором (подробное объяснение см. далее в этой главе).

В журнал событий приложений Windows записываются все сообщения от приложений и сообщения от операционной системы Windows и ее компонентов. Роль, которую журнал событий приложений Windows играет в процессе автоматизации, состоит в извещении службы SQL Server Agent о происходящих событиях.

Служба SQL Server Agent также подключается к журналу событий приложений Windows и к службе MSSQLSERVER. Роль службы SQL Server Agent в процессе автоматизации состоит в выполнении действия после получения извещения от журнала событий приложений Windows. Выполняемое действие может быть связанным со службой MSSQLSERVER или каким-либо другим приложением. Совместная работа этих трех компонентов показана на рис. 17.1.

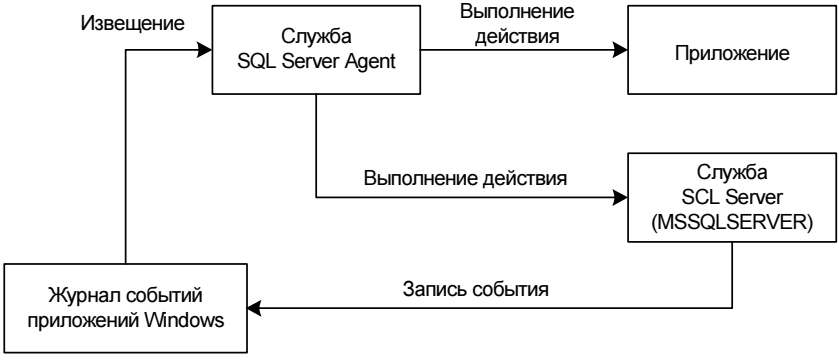


Рис. 17.1. Компоненты автоматизации сервера SQL Server

Запуск службы SQL Server Agent

Служба SQL Server Agent выполняет задания и активирует предупреждающие сообщения. Как мы увидим далее в этой главе, задания и предупреждающие сообщения определяются отдельно и могут выполняться независимо друг от друга. Тем не менее задания и предупреждения могут дополнять друг друга, поскольку задание может вызывать предупреждающее сообщение, и наоборот.

Рассмотрим следующий пример. Предположим, что некоторое задание выполняется для того, чтобы проинформировать системного администратора о неожиданном заполнении журнала транзакций, превышающем допустимый предел. Когда происходит это событие, вызывается связанное с ним предупреждающее сообщение, в ответ на которое системному администратору может быть отправлено сообщение по электронной почте или на пейджер.

Другим примером критического события будет сбой при выполнении резервного копирования журнала транзакций. В таком случае связанное с этим событием пре-

дупреждающее сообщение может запустить задание, которое усекает журнал транзакций. Такая реакция будет адекватной, если причиной сбоя резервного копирования является переполнение журнала транзакций. В других случаях, например, когда будет полностью заполнено устройство, на которое осуществляется резервное копирование, подобное усечение журнала транзакций не даст никакого эффекта. Этот пример показывает тесную связь, которая может существовать между событиями, имеющими подобные признаки.

Служба SQL Server Agent позволяет автоматизировать различные задачи администрирования. Но прежде чем эту службу можно будет использовать, ее нужно запустить. Для этого в обозревателе объектов правой кнопкой мыши щелкните узел службы SQL Server Agent и в появившемся контекстном меню выберите пункт **Start**.

Как уже упоминалось, инициирование предупреждающего сообщения может также включать в себя отправку сообщения оператору (или операторам) по электронной почте, используя компонент Database Mail. Компонент *Database Mail* представляет собой решение уровня предприятия для отправки по электронной почте сообщений от компонента Database Engine. С помощью Database Mail ваши приложения могут отправлять пользователям сообщения по электронной почте. Эти сообщения могут содержать результаты запросов, а также файлы с любого ресурса в сети.

Создание заданий и операторов

Общая процедура для создания задания состоит из трех следующих действий:

1. Создается задание и его шаги.
2. Создается расписание для выполнения задания, если задание не предназначено для выполнения по требованию.
3. Операторы извещаются о статусе задания.

Все эти действия подробно рассматриваются с использованием примера в последующих разделах.

Создание задания и его шагов

Задание может состоять из одного или нескольких шагов. Шаг задания можно определить несколькими способами.

- **Используя инструкции языка Transact-SQL.** Многие шаги заданий содержат инструкции Transact-SQL. Например, для автоматизации резервного копирования базы данных или журнала транзакций применяется инструкция `BACKUP DATABASE` или инструкция `BACKUP LOG` соответственно.
- **Используя операционную систему (CmdExec).** Для некоторых заданий может потребоваться выполнение какой-либо утилиты SQL Server, которая обычно запускается соответствующей командой. Например, чтобы автоматически переместить данные с сервера базы данных в файл данных или наоборот, можно использовать утилиту `bcp`.

- **Запуская программу.** В качестве еще одной альтернативы может потребоваться выполнить программу, которая была разработана с использованием языка Visual Basic или какого-либо другого языка программирования. В этом случае, чтобы служба SQL Server Agent могла найти исполняемый файл этой программы, в текстовом поле **Command** в путь запуска этой программы всегда необходимо включать букву накопителя (диска).

Если задание состоит из нескольких шагов, важно определить, какие действия нужно предпринять в случае сбоя. Обычно компонент Database Engine начинает выполнять следующий шаг задания после успешного завершения выполнения предыдущего. Но в случае сбоя на каком-либо шаге задания все следующие за ним шаги выполняться не будут. Поэтому всегда следует указывать, какие шаги задания необходимо попытаться выполнить снова в случае возникновения сбоя на одном из них. И, конечно же, надо будет устранить причину аварийного завершения выполнения шага. Очевидно, что если причину ошибки не устранить, то простое повторное выполнение задания приведет к той же самой ошибке.

ПРИМЕЧАНИЕ

Количество попыток зависит от типа и содержания выполняемого шага задания (пакет, команда или прикладная программа).

Задание можно создать, используя одно из следующих средств:

- среду SQL Server Management Studio;
- хранимые системные процедуры `sp_add_job` и `sp_add_jobstep`.

Для создания задания резервного копирования базы данных `sample`, рассматриваемого в примере, используется среда SQL Server Management Studio. Чтобы создать это задание, в обозревателе объектов Object Explorer подключитесь к экземпляру сервера, а затем разверните узел этого экземпляра. Запустите службу SQL Server Agent, потом разверните ее узел, щелкните правой кнопкой папку "Jobs" и в контекстном меню выберите пункт **New Job**. В результате откроется диалоговое окно **New Job** (рис. 17.2).

На странице **General** в текстовом поле **Name** введите имя задания. (В данном примере заданию резервного копирования базы данных `sample` присвоим имя `backup_sample`.)

Далее справа от поля **Owner** нажмите кнопку с изображением многоточия (**...**) и выберите владельца, ответственного за выполнение этого задания. В раскрывающемся списке **Category** выберите категорию, которой принадлежит данное задание. При желании в текстовое поле **Description** можно добавить описание задания.

ПРИМЕЧАНИЕ

Для управления несколькими заданиями рекомендуется подразделить их на категории. Это особенно полезно, если задания выполняются в многопользовательской среде.

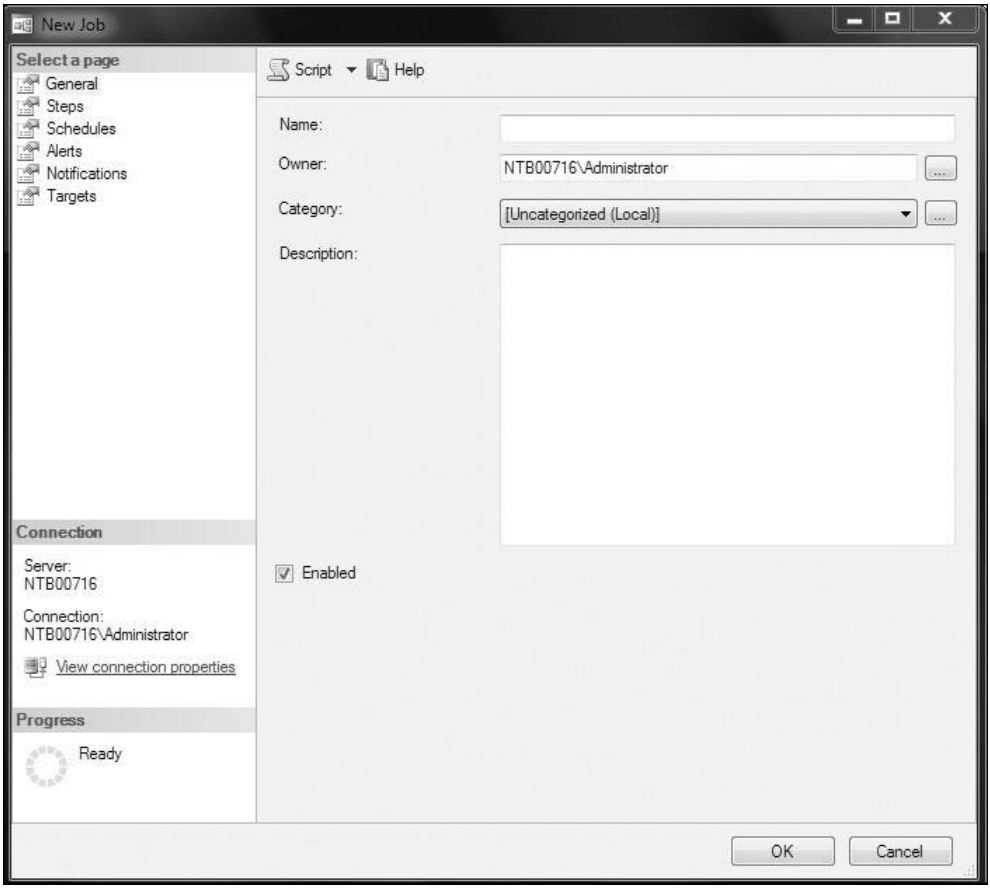


Рис. 17.2. Диалоговое окно **New Job**

Установите флажок **Enabled**, чтобы сделать задание доступным для выполнения.

ПРИМЕЧАНИЕ

Все задания являются доступными по умолчанию. Служба SQL Server Agent отключает задание, если его время выполнения прошло или если прошла конечная дата повторяющегося задания. В обоих случаях задание вручную нужно снова сделать доступным.

Каждое задание должно иметь один или несколько шагов. Поэтому, кроме определения свойств задания, необходимо создать для него, по крайней мере, один шаг, прежде чем его можно будет сохранить. Чтобы определить шаги для задания, в диалоговом окне **New Job** откройте страницу **Steps** и в ней нажмите кнопку **New**. В результате откроется диалоговое окно **New Job Step**, как это показано на рис. 17.3.

Введите имя нового шага, которое для данного примера будет `backup`. В раскрывающемся списке **Type** выберите **Transact-SQL script (T-SQL)**, поскольку резерв-

ное копирование базы данных `sample` будет выполняться с использованием инструкции `BACKUP DATABASE` языка Transact-SQL.

В раскрывающемся списке **Database** выберите базу данных `master`, т. к. при создании резервной копии какой-либо базы данных эта системная база данных должна быть указана в качестве текущей.

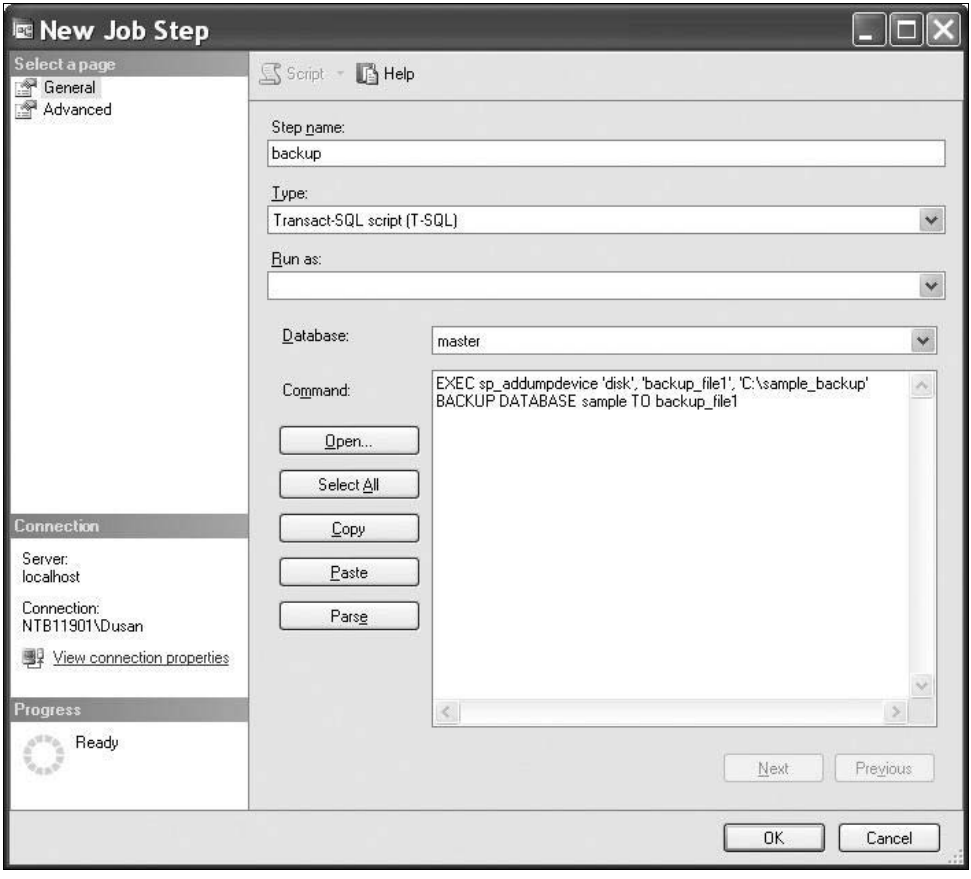


Рис. 17.3. Диалоговое окно **New Job Step**, страница **General**

Инструкцию Transact-SQL можно ввести непосредственно в текстовое поле **Command** или же запустить ее из файла. В первом случае сначала измените путь к файлу резервной копии, а затем введите следующие команды:

```
EXEC sp_addumpdevice 'disk', 'backup_file1', 'C:\sample_backup'
BACKUP DATABASE sample TO backup_file1
```

Как вы, наверное, уже догадались, системная процедура `sp_addumpdevice` добавляет устройство резервного копирования в экземпляр компонента Database Engine. Чтобы запустить инструкцию языка Transact-SQL из файла, нажмите кнопку **Open** и выберите файл, содержащий команду. Проверить правильность синтаксиса инструкций можно, нажав кнопку **Parse**.

Создание расписания задания

Созданное задание можно выполнять по требованию (т. е. вручную пользователем) или запланировать его выполнение по одному или нескольким расписаниям. Запланированное задание может выполняться в указанное время или с заданной периодичностью.

ПРИМЕЧАНИЕ

Каждое задание может иметь несколько расписаний. Например, резервное копирование журнала транзакций можно выполнять по двум разным расписаниям, в зависимости от времени дня. Это означает, что в течение пикового рабочего периода резервное копирование можно выполнять более часто, чем в периоды пониженной нагрузки.

Чтобы создать расписание для существующего задания с помощью среды SQL Server Management Studio, откройте страницу **Schedules** и нажмите кнопку **New**. (Эту страницу можно выбрать или в диалоговом окне **New Job**, если оно еще не было закрыто, или же в окне свойств **Job Properties** для сохраненного задания, которое выглядит точно так же, как и окно **New Job**, показанное на рис. 17.2. Чтобы

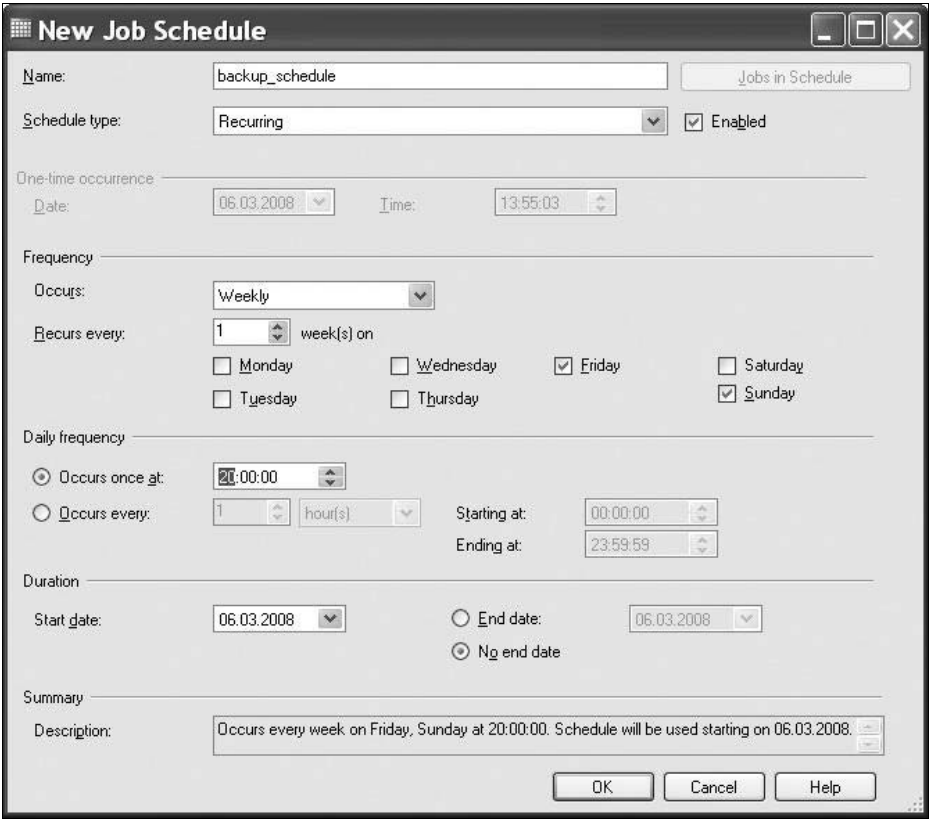


Рис. 17.4. Диалоговое окно **New Job Schedule**

для определенного задания открыть диалоговое окно **Job Properties**, в обозревателе объектов разверните узел службы SQL Server Agent, разверните в нем папку "Jobs" и выполните двойной щелчок мышью по требуемому заданию.) В результате откроется диалоговое окно **New Job Schedule** (рис. 17.4).

ПРИМЕЧАНИЕ

В случае вывода предупреждающего сообщения "The On Access action of the last step will be changed from Get Next Step to Quit with Success" ("Действие On Access последнего шага будет изменено с Get Next Step на Quit with Success"), нажмите кнопку **Yes**.

Для нашего примера, запланируем выполнение задания резервного копирования базы данных `sample` каждую пятницу в 20:00. Для этого введите имя расписания в строку **Name** и в раскрывающемся списке **Schedule type** выберите значение **Recurring**. В области **Frequency** в раскрывающемся списке **Occurs** выберите значение **Weekly**, а также установите флажок **Friday** и очистите флажки для всех других дней. В области **Daily frequency** выберите переключатель **Occurs once at** и установите требуемое время (20:00:00). В области **Duration** в раскрывающемся списке **Start date** выберите начальную дату задания, а затем выберите переключатель **End date** и установите требуемую конечную дату выполнения задания. Если же в расписании задания не должно быть конечной даты, то выберите переключатель **No end date**.

Операторы извещения о состоянии задания

Существует несколько методов извещения о завершении выполнения задания. Например, можно указать системе, чтобы она записывала соответствующее сообщение в журнал событий приложений Windows, в надежде, что системный администратор иногда просматривает этот журнал. Но лучшим решением будет явно известить одного или нескольких операторов системы баз данных по электронной почте, пейджеру или по локальной сети, используя команду `net send`.

Прежде чем назначить оператор заданию, вы должны создать для него запись. Для создания оператора, в обозревателе объектов среды Management Studio разверните узел службы SQL Server Agent, щелкните в нем правой кнопкой мыши папку "Operators" и в контекстном меню выберите пункт **New Operator**. Откроется диалоговое окно **New Operator** (рис. 17.5).

На странице **General** в текстовом поле **Name** введите имя оператора. Далее укажите требуемый способ извещения данного оператора (по электронной почте, на пейджер или по сети через `net send`), введя для этого соответствующие адреса в соответствующие поля. В области **Pager on duty schedule** установите рабочие дни и часы оператора.

Теперь можно задать отправку извещения оператору после завершения исполнения задания (успешного или неуспешного). Для этого опять откройте окно свойств задания, выберите в нем страницу **Notifications** (рис. 17.6) и задайте в ней требуемые параметры для отправки извещения.

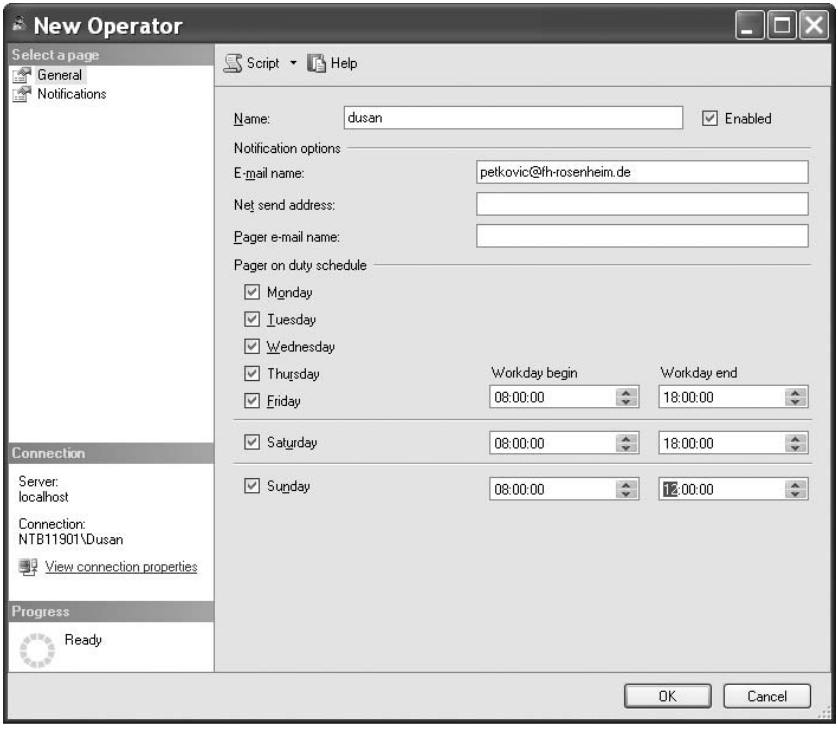


Рис. 17.5. Диалоговое окно **New Operator**

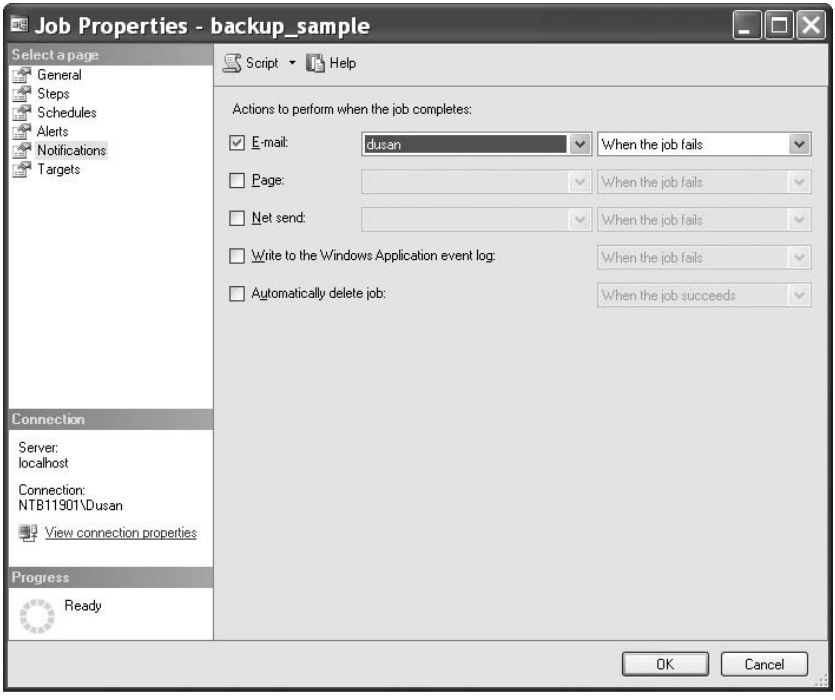


Рис. 17.6. Окно **Job Properties**, страница **Notifications**

Кроме отправки сообщения по электронной почте, на пейджер или по сети через `net send`, на этой странице можно также выбрать вариант создания записи в журнале событий приложений Windows и/или удаления задания.

Просмотр журнала истории задания

Компонент Database Engine сохраняет информацию обо всех действиях задания в таблице `sysjobhistory` системной базы данных `msdb`. Таким образом, эта таблица представляет журнал заданий системы баз данных. Информацию в этой таблице можно просмотреть с помощью среды SQL Server Management Studio. Для этого разверните в обозревателе объектов узел SQL Server Agent, в нем разверните папку "Jobs", щелкните правой кнопкой мыши требуемое задание и выберите в контекстном меню пункт **View History**. Откроется диалоговое окно **Log File Viewer** с информацией для данного задания.

Строки журнала задания отображаются в панели в правой стороне окна и содержат, среди прочего, следующую информацию:

- дату и время запуска каждого шага задания;
- состояние выполнения шага задания — успешное или неуспешное;
- операторы, которым были отправлены извещения;
- длительность выполнения задания;
- ошибки или сообщения, связанные с шагом задания.

По умолчанию журнал заданий может содержать максимум 1000 строк, а количество строк для отдельного задания ограничено значением 100 строк. Журнал заданий автоматически очищается, когда заполняются все его строки. Если имеется несколько заданий, чтобы сохранить информацию для их всех, следует увеличить общее количество строк журнала заданий и/или количество строк для каждого задания. Для этого щелкните правой кнопкой узел **SQL Server Agent** и в контекстном меню выберите пункт **Properties**. В открывшемся диалоговом окне **SQL Server Agent Properties** выберите страницу **History**, и на ней установите новые значения для общего максимального количества строк журнала заданий и для каждого задания. На этой же странице можно установить флажок **Automatically remove agent history** для автоматической очистки журнала и указать период времени, после которого выполнять очистку.

Предупреждающие сообщения

Информация о выполнении заданий и системные сообщения об ошибках сохраняются в журнале событий приложений Windows. Служба SQL Server Agent читает записи в этом журнале и сравнивает их с заданными условиями для инициирования предупреждений. При обнаружении такого совпадения служба SQL Server Agent инициирует предупреждение. Таким образом, предупреждения можно использовать, чтобы реагировать на возможные проблемы (например, заполнение журнала транзакций), различные системные ошибки или определяемые пользователем

ошибки. Прежде чем рассматривать, как создавать предупреждения, в следующем разделе мы рассмотрим предмет системных сообщений об ошибках и два журнала (журнал ошибок службы SQL Server Agent и журнал событий приложений Windows), в которые записываются все системные сообщения и, следовательно, информация о большинстве ошибок.

Сообщения об ошибках

Системные ошибки делятся на четыре категории. Компонент Database Engine предоставляет обширную информацию о каждой ошибке. Эта информация структурирована и включает следующее:

- однозначный номер сообщения об ошибке;
- число в диапазоне от 0 до 25, представляющее уровень серьезности ошибки;
- номер строки, в которой произошла ошибка;
- текст описания ошибки.

ПРИМЕЧАНИЕ

Описание ошибки также может содержать рекомендацию, как решить проблему.

В примере 17.1 показан запрос к несуществующей таблице в базе данных `sample`, что вызывает системную ошибку.

Пример 17.1. Запрос, вызывающий системную ошибку

```
USE sample;
SELECT * FROM authors;
```

Результатом выполнения этого запроса будет следующее сообщение об ошибке:

```
Msg 208, Level 16, State 1, Line 2
Invalid object name 'authors'
```

(Сообщение 208, Уровень 15, Строка 2
Недействительное имя объекта `'authors'`)

Просмотреть информацию о сообщениях об ошибках можно с помощью каталога представления `sys.messages`, наиболее важными столбцами которого являются столбцы `message_id`, `severity` и `text`.

Каждому однозначному номеру ошибки соответствует сообщение об ошибке. (Сообщение об ошибке сохраняется в столбце `text`, а соответствующий номер ошибки — в столбце `message_id` представления каталога `sys.messages`.) В примере 17.1 сообщение, связанное с несуществующим или неправильно указанном объекте базы данных, соответствует номеру ошибки — 208.

Уровень серьезности ошибки (столбец `severity` представления каталога `sys.messages`) представлен диапазоном значений от 0 до 25. Уровни от 0 до 10 обозначают просто информационные сообщения, где ничего не требуется исправлять. Все уровни ошибок от 11 до 16 указывают программные ошибки, которые могут быть разрешены пользователем. Значения уровней 17 и 18 обозначают программные и аппаратные ошибки, которые обычно не завершают выполнение процесса. Все ошибки уровня 19 и выше являются неисправимыми системными ошибками. Соединение программы, вызвавшей такую ошибку, закрывается, после чего ее процесс удаляется.

Сообщения, связанные с программными ошибками (т. е. ошибок с уровнем от 11 до 16) только выводятся на экран. Все системные ошибки (т. е. ошибки уровня 19 и выше) также записываются в журнал.

Чтобы решить проблему с ошибкой, обычно необходимо ознакомиться с ее подробным описанием. Подробные описания ошибок также предоставляются в электронной документации.

Сообщения о системных ошибках записываются в журнал ошибок службы SQL Server Agent и в журнал событий приложений Windows. Эти два журнала рассматриваются в последующих двух разделах.

Журнал ошибок службы SQL Server Agent

Служба SQL Server Agent создает журнал ошибок, в котором по умолчанию записываются сообщения предупреждений и ошибок следующего типа:

- предупреждающие сообщения, предоставляющие информацию о возможных проблемах;
- сообщения об ошибках, требующих вмешательства системного администратора.

Система может вести до десяти журналов службы SQL Server Agent. Текущий журнал называется `Current`, а остальные называются `Archieve` с расширением, указывающим относительный возраст данного журнала. Например, имя `Archieve#1` указывает самый последний архивный журнал ошибок.

Журнал ошибок службы SQL Server Agent является важным источником информации для системного администратора, с помощью которой он может проследить исполнение процесса, вызвавшего ошибку, и определить, какие необходимые меры следует предпринять.

Для просмотра журналов ошибок службы SQL Server Agent последовательно разверните в обозревателе объектов среды Management Studio узел сервера, узел службы SQL Server Agent, а затем папку "Error Logs". Выполните двойной щелчок мышью по требуемому журналу. Откроется диалоговое окно **Log File Viewer** для данного журнала.

Журнал событий приложений Windows

Компонент Database Engine также записывает системные сообщения в журнал событий приложений Windows. В журнал событий приложений Windows записываются все сообщения от приложений и сообщения от операционной системы Windows и ее компонентов. Просмотреть журнал событий приложений Windows можно с помощью средства просмотра событий.

Просмотр ошибок в журнале событий приложений Windows имеет некоторые преимущества над просмотром их в журнале ошибок службы SQL Server Agent. Наиболее важное из них состоит в том, что журнал событий приложений Windows предоставляет дополнительный компонент для поиска требуемых строк.

Для просмотра информации в журнале событий приложений Windows выполните следующую последовательность команд Windows: **Пуск | Панель управления | Система и безопасность | Администрирование | Просмотр событий**. В узле **Журналы Windows** окна средства "Просмотр событий" можно выбрать для просмотра сообщения системы, безопасности и приложений. События системы SQL Server хранятся в узле **Приложение** и обозначаются меткой MSSQLSERVER.

Определение предупреждающих сообщений для обработки ошибок

Чтобы инициировать ответное действие на конкретную ошибку или группу ошибок определенного уровня серьезности, можно определить соответствующее предупреждающее сообщение. Предупреждающие сообщения для конкретной ошибки различаются для системных ошибок и для определяемых пользователем ошибок. (Создание предупреждающих сообщений для определяемых пользователем ошибок рассматривается далее в этой главе.)

В последующем материале этого раздела рассматривается создание предупреждающих сообщений с помощью среды Management Studio.

Создание предупреждающих сообщений для системных ошибок

Для демонстрации создания предупреждающего сообщения о системной ошибке будет использован пример 13.5, в котором две транзакции блокируют друг друга. При взаимоблокировке двух транзакций, транзакция-жертва взаимоблокировки должна быть выполнена повторно. Это можно сделать, помимо других способов, и при помощи предупреждающих сообщений.

Чтобы создать предупреждающее сообщение для обработки взаимоблокировки (или любой другой ошибки), разверните узел службы SQL Server Agent, щелкните правой кнопкой папку "Alerts" и в контекстном меню выберите пункт **New Alert**. В открывшемся диалоговом окне **New Alert** (рис. 17.7) в поле **Name** введите имя сообщения, в раскрывающемся списке **Type** выберите **SQL Server event alert**, а в раскрывающемся списке поля **Database name** выберите **<all databases>**.

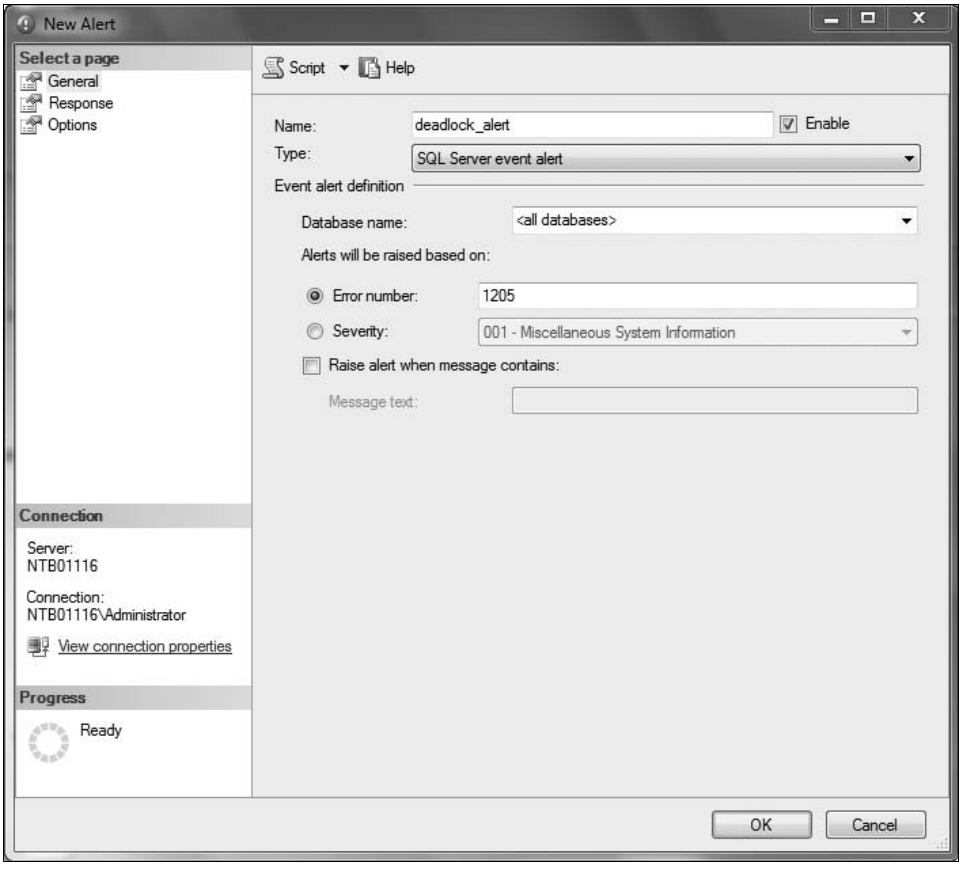


Рис. 17.7. Диалоговое окно **New Alert**, страница **General**

Установите переключатель **Error number** и рядом в поле справа введите значение `1205`. (Этот номер ошибки обозначает проблему взаимоблокировки, где текущий процесс был назначен "жертвой".)

Далее определяется ответное действие для предупреждения. В этом же диалоговом окне выберите страницу **Response** (рис. 17.8).

Сначала установите флажок **Execute job**, а затем выберите задание, которое будет выполняться при появлении предупреждающего сообщения. (Для данного примера указывается задание `deadlock_all_db`, которое перезапускает транзакцию-жертву.) Установите флажок **Notify operators**, а затем в списке области **Operator list** выберите необходимые операторы и метод или методы их извещения (электронная почта, пейджер и/или посредством команды `net send`).

ПРИМЕЧАНИЕ

В предшествующем примере предполагается завершение процесса-жертвы. Но в реальных условиях после ошибки взаимоблокировки 1205 программа сама перезапускает этот процесс.

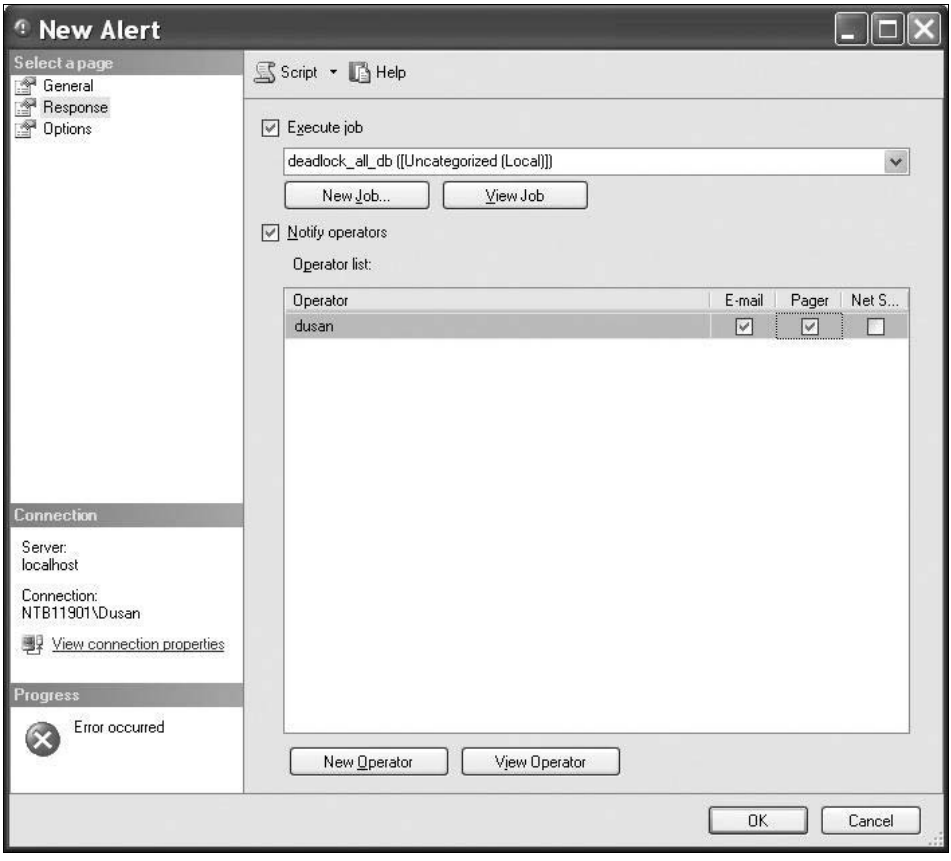


Рис. 17.8. Диалоговое окно **New Alert**, страница **Response**

Создание предупреждающих сообщений для ошибок определенного уровня

Кроме создания предупреждающих сообщений для отдельной ошибки, предупреждающие сообщения можно также создавать в ответ на ошибки определенного уровня серьезности, иначе говоря, для групп ошибок. Как уже упоминалось, каждая системная ошибка принадлежит к одному из 26 (с 0 по 25) уровней серьезности. Чем выше уровень, тем серьезнее ошибка. Ошибки уровней с 20 по 25 являются неисправимыми (фатальными) ошибками. Ошибки уровней с 19 по 25 записываются в журнал событий приложений Windows.

ПРИМЕЧАНИЕ

О возникновении неисправимой ошибки всегда следует извещать оператора.

В качестве примера создания предупреждающего сообщения для ошибки определенного уровня серьезности рассмотрим создание с помощью среды Management Studio сообщения для уровня серьезности 25. Сначала разверните узел службы SQL

Server Agent, щелкните правой кнопкой папку "Alerts" и в контекстном меню выберите пункт **New Alert**. В поле **Name** введите имя для данного предупреждающего сообщения (например, Severity 25 errors). В раскрывающемся списке **Type** выберите **SQL Sever event alert**. В раскрывающемся списке **Database name** выберите базу данных `sample`. Установите переключатель **Severity** и в соответствующем ему раскрывающемся списке выберите **025 – Fatal Error**.

Далее выберите страницу **Response** и на ней укажите одного или нескольких операторов, которых следует известить в случае возникновения ошибки 25-го уровня серьезности.

Создание предупреждающих сообщений для определяемых пользователем ошибок

Кроме создания предупреждающих сообщений для системных ошибок, их также можно создавать и для определяемых пользователем ошибок для отдельных приложений баз данных. С помощью таких сообщений (и предупреждающих сообщений) можно определять решения для проблем, которые могут возникнуть в приложении.

Для создания предупреждающих сообщений для определенных пользователем ошибок выполняются следующие шаги:

1. Создается сообщение об ошибке.
2. Определяется ошибка и ее обработчик в приложении базы данных.
3. Определяется предупреждающее сообщение для ошибки.

В качестве примера рассмотрим создание предупреждающего сообщения, которое инициируется, если дата отправки какой-либо продукции более ранняя, чем дата ее заказа. (Для данного примера используется таблица `sales`, определение которой см. в *главе 5*.)

ПРИМЕЧАНИЕ

Из этих только что названных трех шагов здесь описываются только первые два, поскольку предупреждающие сообщения для ошибок, определяемых пользователем, задаются аналогично сообщениям для системных ошибок.

Создание сообщения об ошибке

Определяемое пользователем сообщение об ошибке можно создать как с помощью среды Management Studio, так и посредством системной хранимой процедуры `sp_addmessage`. В примере 17.2 сообщение об ошибке создается с помощью системной хранимой процедуры `sp_addmessage`.

Пример 17.2. Создание сообщения об ошибке посредством системной процедуры

```
sp_addmessage @msgnum=50010, @severity=16,
@msgtext='The shipping date of a product is earlier than the order date',
@lang='us_english', @with_log='true'
```

В примере 17.2 системная хранимая процедура sp_addmessage создает определяемое пользователем сообщение об ошибке с номером ошибки 50010 (параметр @msgnum) и уровнем 16 (параметр @severity). Все определяемые пользователями сообщения об ошибках хранятся в таблице sysmessages системной базы данных master, и их можно просмотреть с помощью представления каталога sys.messages. В примере 17.2 номер ошибки 50010 был выбран по той причине, что номера всех определяемых пользователями сообщений об ошибках должны быть больше 50000. (Все номера сообщений об ошибках меньше чем 50000 зарезервированы для системы.)

Для каждого определяемого пользователем сообщения об ошибке можно по желанию указать в параметре @lang язык для его отображения. Этой возможностью может потребоваться воспользоваться, если на компьютере установлено несколько языков. (Когда параметр @lang опущен, то для сообщения применяется язык по умолчанию.)

По умолчанию определяемые пользователем ошибки не записываются в журнал событий приложений Windows. С другой стороны, чтобы инициировать предупреждение по этой ошибке, ее необходимо записать в этот журнал. Вынудить запись определяемой пользователем ошибки в журнал событий приложений Windows можно, присвоив параметру @with_log системной процедуры sp_addmessage значение true.

Определение условий ошибки посредством триггеров

Для генерирования ошибки в приложениях баз данных используется инструкция RAISEERROR. Эта инструкция возвращает определенное пользователем сообщение об ошибке и устанавливает флаг в глобальной переменной @@error. (Обработку ошибок можно также выполнять, используя блоки TRY/CATCH.)

В примере 17.3 создается триггер t_date_comp, который возвращает определенную пользователем ошибку номер 50010, если дата заказа продукта более поздняя, чем его дата отправки.

ПРИМЕЧАНИЕ

Для выполнения примера 17.3 необходимо, чтобы существовала таблица sales (см. пример 5.24).

Пример 17.3. Код для определяемой пользователем ошибки и ее обработчика

```
USE sample;
GO
CREATE TRIGGER t_date_comp
   ON sales
   FOR INSERT AS
   DECLARE @order_date DATE
   DECLARE @shipped_date DATE
```

```
SELECT @order_date=order_date, @shipped_date=ship_date FROM INSERTED
   IF @order_date > @shipped_date
      RAISERROR (50010, 16, -1)
```

Теперь, если попытаться вставить в таблицу `sales` строку, в которой дата отправки более ранняя, чем дата заказа, как это делается в следующей инструкции:

```
INSERT INTO sales VALUES (1, '01.01.2007', '01.01.2006')
```

система возвратит определенное в примере 17.3 сообщение об ошибке:

```
Msg 50010, Level 16, State 1, Procedure t_date_comp, Line 8
The shipping date of a product is earlier than the order date
```

Резюме

Компонент Database Engine позволяет автоматизировать и упорядочивать многие задачи администрирования системы, такие как резервное копирование базы данных, перемещение данных и обслуживание индексов. Для выполнения задач автоматизации необходимо, чтобы была запущена служба SQL Server Agent.

Для автоматизации задачи нужно выполнить несколько шагов:

1. Создать задание.
2. Создать операторы.
3. Создать предупреждающие сообщения.

Термины *задача* и *задание* являются синонимами, поэтому, когда создается задача, создается и определенное задание для автоматизирования этой задачи. Самым простым способом создания задания, включая его шаги и расписание выполнения, будет использование SQL Server Management Studio.

О завершении выполнения задания (успешном или неуспешном) можно оповестить одного или несколько операторов, самым легким способом определения которых будет опять же использование среды SQL Server Management Studio.

Предупреждающие сообщения определяются отдельно и могут выполняться независимо от заданий. Предупреждающие сообщения могут обрабатывать отдельные системные ошибки, определяемые пользователем ошибки или группы ошибок уровней серьезности от 0 до 25.

В следующей главе мы рассмотрим репликацию данных.

Упражнения

Упражнение 17.1

Назовите несколько задач администрирования, которые можно было автоматизировать.

Упражнение 17.2

Каким образом можно реализовать резервное копирование журнала транзакций базы данных каждый час во время пиковых часов работы и каждые четыре часа во время внепиковых часов работы?

Упражнение 17.3

Требуется протестировать производительность рабочей базы данных в плане блокировок и узнать, превышает ли время ожидания блокировки 30 сек. Каким образом можно реализовать автоматическое извещение об этом событии?

Упражнение 17.4

Назовите все части сообщения об ошибке SQL Server.

Упражнение 17.5

Какие столбцы представления каталога являются наиболее важными касательно ошибок?

Глава 18

Репликация данных

- **Распределенные данные и методы распределения**
- **Общие сведения о репликации в SQL Server**
- **Управление репликацией**

В наше время рыночные отношения вынуждают большинство компаний настраивать свои компьютеры (и выполняющиеся на них приложения) так, чтобы уделять максимальное внимание своей предпринимательской деятельности и своим клиентам. В результате данные, используемые этими приложениями, должны быть доступны по любому запросу, в любое время и в любом месте. Такая среда данных предоставляется несколькими распределенными базами данных, которые содержат множество копий одной и той же информации.

Переезжающие с места на место продавцы (коммивояжеры) представляют хороший пример использования среды распределенных данных. В течение дня такой продавец обычно запрашивает с базы данных с помощью ноутбука всю необходимую информацию (например, наличие товаров и цены на них), чтобы проинформировать клиента на месте об интересующем его товаре. А в конце дня, закончив обход клиентов, он в своем гостиничном номере опять обращается со своего компьютера к базе данных, на этот раз, чтобы передать в нее данные о заключенных в течение дня сделках.

Рассматривая такой сценарий можно видеть, что среда распределенных данных предоставляет несколько следующих преимуществ по сравнению с централизованной обработкой данных:

- имеется возможность немедленного доступа к данным для пользователей в любое время;
- имеется возможность автономной работы локальных пользователей;
- снижается уровень сетевого трафика;
- уменьшается стоимость безостановочной обработки данных.

С другой стороны, уровень сложности среды распределенных данных намного выше, чем соответствующей централизованной модели и, соответственно, требует дополнительного планирования и администрирования.

Во вступительной части этой главы рассматриваются распределенные транзакции и выполняется сравнительный анализ этих транзакций с репликацией данных, что и является темой этой главы. Затем дается представление об элементах репликации и объясняются существующие типы репликации. В последней части главы описываются три мастера, которые применяются для управления репликацией.

Распределенные данные и методы распределения

Существует два основных метода для распределения данных по нескольким серверам баз данных:

- распределенные транзакции;
- репликация данных.

Распределенная транзакция представляет собой транзакцию, в которой все обновления для всех мест размещения данных, где хранятся распределенные данные, собраны вместе и исполняются синхронно. Для реализации распределенных транзакций системы распределенных баз данных используют метод, называемый *двухфазной фиксацией* (two-phase commit).

Каждая база данных, принимающая участие в распределенной транзакции, имеет свой собственный метод восстановления после сбоев. (Вспомните, что выполняются или все инструкции транзакции или не выполняется ни одна из них.) Менеджер глобального восстановления (называемый *координатором*) координирует обе фазы распределенного процесса.

На первой фазе процесса координатор проверяет, что все участвующие сайты готовы выполнить свою часть распределенной транзакции. Вторая фаза состоит собственно из выполнения транзакции на всех участвующих сайтах. В течение этого процесса любая ошибка на любом сайте приводит к тому, что координатор прекращает выполнение транзакции. В этом случае координатор отправляет сообщение каждому менеджеру восстановления с указанием выполнить отмену той части транзакции, которая была выполнена на данном сайте.

ПРИМЕЧАНИЕ

Координатор распределенных транзакций Microsoft (DTC — Distributed Transaction Coordinator) поддерживает распределенные транзакции, используя метод двухфазной фиксации.

В процессе репликации копии данных распространяются с базы данных-источника на одну или более баз данных-получателей, расположенных на отдельных компьютерах. Вследствие этого, между репликацией данных и распределенными транзак-

циями есть два отличия — отсутствие согласования по времени и наличие задержки по времени.

В отличие от метода распределенных транзакций, в котором все данные распространяются по всем участвующим сайтам одновременно, репликация данных допускает наличие на сайтах в одно и то же время разных данных. Кроме этого, процесс репликации данных является асинхронным. Это означает, что единообразие всех копий данных на всех участвующих сайтах достигается с определенной задержкой. Эта задержка может составлять от нескольких секунд до нескольких дней или даже недель.

В большинстве случаев репликация данных является лучшим решением, чем распределенные транзакции вследствие его большей надежности и низшего уровня накладных расходов. Опыт работы с двухфазной фиксацией показал, что с повышением числа участвующих сайтов администрирование этого подхода становится очень трудным. Кроме этого, повышение числа участвующих сайтов понижает надежность, поскольку с увеличением количества узлов повышается вероятность сбоя какой-либо локальной части распределенной транзакции. А сбой в одной локальной части также означает ненормальное завершение всей распределенной транзакции.

Другой причиной для использования репликации данных вместо централизованных данных является производительность: производительность клиентов на сайте с дублированными данными повышается, поскольку они могут обращаться к данным локально, а не подключаться для этого по сети к центральному серверу баз данных.

Общие сведения о репликации в SQL Server

Обычно репликация основывается на одной из двух различных концепций:

- на использовании журналов транзакций;
- на использовании триггеров.

Как уже упоминалось в *главе 16*, компонент Database Engine хранит все значения измененных строк (как значения до изменений, так и значения после изменений) в системных файлах, называемых журналами транзакций. Если выбранные строки требуется дублировать, система запускает новый процесс, который считывает данные в журнале транзакций и отправляет их одной или нескольким базам данных-получателям.

Другой метод основан на использовании триггеров. Изменение таблицы, содержащей данные, для которых должна быть выполнена репликация, вызывает соответствующий триггер, который в свою очередь создает новую таблицу с данными и начинает процесс репликации.

Обе эти концепции имеют свои преимущества и недостатки. Репликация с использованием журнала транзакций характеризуется лучшей производительностью, по-

скольку процесс, считывающий данные с журнала транзакций, выполняется асинхронно и оказывает незначительное влияние на производительность всей системы. С другой стороны, реализация репликации на основе журнала транзакций является очень сложной, т. к. система баз данных не только должна управлять дополнительными процессами и буферизацией, но также решать проблемы одновременного обращения к журналу транзакций системы и процессов репликации.

ПРИМЕЧАНИЕ

Компонент Database Engine использует обе концепции: метод журнала транзакций для репликации транзакций и триггеры для репликации для слияния обработанных репликаций. Обработка транзакций и слияние репликаций подробно описаны далее в этой главе.

Издатели, распространители и подписчики

Репликация в Database Engine основана на модели *издатель-подписчик*, описывающей различные роли, которые может играть сервер в процессе репликации. Один или несколько серверов публикуют данные, на которые могут подписываться другие серверы. Между ними находится распространитель, который сохраняет все изменения и пересылает их подписчикам. Таким образом, в сценарии репликации узел может играть одну (или несколько) из трех ролей.

- **Издатель (или сервер публикаций).** Поддерживает свои исходные базы данных, предоставляет данные для репликации и отправляет измененные данные распространителю.
- **Распространитель (или сервер распространения).** Получает от издателя и сохраняет все изменения реплицируемых данных и пересылает их соответствующим подписчикам.
- **Подписчик (или сервер подписки).** Получает и сохраняет опубликованные данные.

В процессе репликации сервер базы данных может играть несколько ролей. Например, сервер может одновременно быть и издателем и распространителем. Этот сценарий подходит для процессов с небольшим объемом репликаций и небольшим числом подписчиков. В случае большого количества подписчиков на публикуемую информацию для распространителя может быть выделен отдельный сервер. На рис. 18.1 показан простой сценарий, в котором один экземпляр сервера является одновременно и сервером публикаций, и сервером распространения, а три другие экземпляра являются серверами подписки. (Возможные сценарии репликации подробно рассмотрены в *разд. "Модели репликации" далее в этой главе*.)

ПРИМЕЧАНИЕ

Вы можете выполнять репликацию только баз данных, определенных пользователем.

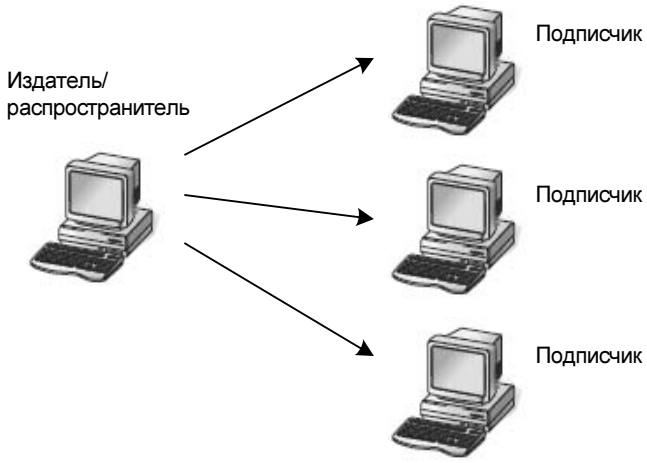


Рис. 18.1. Сценарий публикации с издателем и распространителем, находящимися на одном и том же сервере

Публикации и статьи

Единица публикуемых данных называется *публикацией* (publication). А *статья* (article) содержит данные из таблицы и/или одной или нескольких хранимых процедур. Статья таблицы может быть отдельной таблицей или подмножеством данных таблицы. Статья хранимой процедуры может содержать одну или несколько хранимых процедур, существующих в базе данных на момент публикации.

Публикация состоит из одной или нескольких статей. Публикация может содержать данные только из одной базы данных.

ПРИМЕЧАНИЕ

Публикация является основой подписки. Это означает, что нельзя подписаться непосредственно на статью, поскольку статься всегда является частью публикации.

Фильтром называется процесс, который ограничивает информацию, создавая подмножество. Поэтому публикация содержит один или более следующих элементов, которые определяют типы статей таблицы:

- таблица;
- вертикальный фильтр;
- горизонтальный фильтр;
- комбинация вертикальных и горизонтальных фильтров.

Вертикальный фильтр содержит подмножество столбцов таблицы, а горизонтальный — подмножество строк таблицы.

Публикации тесно связаны с подписками. Подписку можно инициировать двумя разными способами:

- используя принудительную подписку;
- используя подписку по запросу.

При использовании *принудительной подписки* (push subscription) все администрирование по организации подписок выполняется издателем в процессе создания публикации. (Кроме издателя, принудительные подписки также создаются и управляются распространителем.) Принудительные подписки позволяют упростить и централизовать администрирование, поскольку обычный сценарий репликации содержит одного издателя и несколько подписчиков. Преимущество принудительной подписки состоит в более высоком уровне безопасности, потому что управление процессом инициализации осуществляется из одного места. Но с другой стороны, уровень производительности распространителя может понизиться, вследствие одновременного распространения всех подписок.

При *подписке по запросу* (pull subscription) инициирование и управление подпиской осуществляется подписчиком. Подписка по запросу является более избирательной, чем принудительная подписка, поскольку подписчик выбирает сам, на какие публикации подписываться. В отличие от принудительной подписки, подписку по запросу следует использовать для публикаций, не требующих высокого уровня безопасности и с большим количеством подписчиков.

ПРИМЕЧАНИЕ

Типичной формой подписки по запросу является загрузка данных из Интернета.

Существует специальный тип подписки по запросу, называемый *анонимной подпиской* (anonymous subscription). Обычно информация о подписчиках хранится на сервере распространения. Но если требуется уменьшить рабочую нагрузку этого сервера (например, по причине слишком большого количества подписчиков), допускается разрешить подписчикам самим инициировать свои собственные (анонимные) подписки.

База данных *distribution*

База данных `distribution` — это системная база данных, которая устанавливается на сервер распространения при инициировании процесса репликации. Эта база данных содержит все реплицируемые транзакции, предоставленные издателем, которые требуется переслать подписчикам.

Во многих случаях достаточно иметь одну базу данных `distribution`. Но если к одному серверу распространения обращается несколько серверов публикации, для каждого из них можно создать базу данных `distribution`. Таким образом обеспечивается индивидуальность данных, проходящих через каждую базу данных `distribution`.

Агенты

В процессе репликации данных компонент Database Engine использует несколько агентов для управления разными задачами. Система поддерживает, среди прочих, следующие агенты:

- агент моментальных снимков Snapshot Agent;
- агент чтения журнала Log Reader;
- агент распространителя Distribution Agent;
- агент слияния Merge Agent.

Эти агенты рассматриваются в следующих подразделах.

Агент Snapshot Agent

Агент Snapshot Agent создает схемы и данные публикуемых таблиц и сохраняет их на сервер распространения. Схема таблицы и соответствующий файл данных составляют синхронизационный набор, который представляет моментальный снимок таблицы в определенный момент времени. Состояние синхронизации этого набора записывается в базе данных `distribution`. Создает ли агент Snapshot Agent новый файл моментальных снимков при каждом его исполнении, зависит от выбранного типа репликации и от ее параметров.

Агент Log Reader

Если при репликации данных используется журнал транзакций системы, все транзакции, которые содержат подлежащие репликации данные, помечаются для репликации. Компонент, называемый агент Log Reader, выполняет поиск помеченных таким образом транзакций и копирует их из журнала транзакций издателя на сервер распространения. Эти транзакции хранятся в базе данных `distribution`. Каждая база данных, которая использует журнал транзакций для репликации, имеет своего собственного агента Log Reader, работающего на сервере распространения.

Агент Distribution Agent

После сохранения транзакций и моментальных снимков в базе данных `distribution`, их нужно отправить подписчикам. Эту задачу выполняет агент Distribution Agent, который перемещает транзакции и моментальные снимки подписчикам, где они применяются к целевым таблицам в базах данных подписчика.

Задачи агента Distribution Agent для подписок по запросу и принудительных подписок совершенно разные. Для принудительных подписок агент доставляет изменения подписчику, а для подписок по запросу агент извлекает транзакции с сервера распространения. (Все действия, которые изменяют данные на сервере публикаций, применяются к подписчику в хронологическом порядке.)

Агент Merge Agent

Как уже упоминалось, агент Snapshot Agent подготавливает файлы, содержащие схему таблицы и данные, и сохраняет их на сайте распространителе. Поскольку как

издатель, так и подписчик могут обновлять реплицируемые данные, то для того, чтобы отправить эти данные на другие сайты, их необходимо синхронизировать. Эту задачу синхронизации выполняет агент Merge Agent. Иными словами, агент Merge Agent может отправлять реплицируемые данные подписчикам и издателю. Перед началом процесса отправки данных агент Merge Agent также сохраняет соответствующую информацию, которую он использует для отслеживания возможных конфликтов.

Типы репликации

Компонент Database Engine поддерживает следующие типы репликации, которые рассмотрены в последующих разделах:

- репликации транзакций;
- одноранговые репликации транзакций;
- репликации моментальных снимков;
- репликации слиянием.

Репликации транзакций

В репликации транзакций для репликации данных используется журнал транзакций. Все транзакции, содержащие подлежащие репликации данные, помечаются для репликации. Агент Log Reader выполняет поиск помеченных таким образом транзакций и копирует их с журнала транзакций издателя в базу данных `distribution`. Агент Distribution Agent перемещает транзакции подписчикам, где они применяются для целевых таблиц в базах данных подписчика.

ПРИМЕЧАНИЕ

Все таблицы, публикуемые посредством репликации транзакций, должны явно содержать первичный ключ. Первичный ключ требуется для того, чтобы однозначно идентифицировать строки публикуемой таблицы, поскольку в репликации транзакций единицей переноса является строка.

Репликация транзакций может дублировать таблицы (или части таблиц) и хранимые процедуры. Использование хранимых процедур в репликации транзакций повышает производительность, потому что это существенно уменьшает объем пересылаемых по сети данных. Вместо реплицируемых данных подписчикам пересылается только хранимая процедура, которую они и выполняют. Для репликации транзакций можно настроить задержку времени синхронизации между издателем на одной стороне и подписчиками на другой. (Все эти изменения передаются агентами Log Reader и Distribution Agent.)

ПРИМЕЧАНИЕ

Прежде чем может начаться выполнение репликации транзакций, каждому подписчику необходимо отправить копию всей базы данных, что осуществляется посредством моментального снимка.

Специальной формой репликации транзакций является одноранговая репликация транзакций, которая рассматривается в следующем подразделе.

Одноранговая репликация транзакций

При одноранговой репликации транзакций каждый сервер является одновременно издателем, распространителем и подписчиком на одни и те же данные. Иными словами, все серверы содержат одни и те же данные, но каждый сервер является ответственным за модифицирование своего собственного раздела данных. (Обратите внимание, что разделы данных на разных серверах могут пересекаться друг с другом.)

Объяснить одноранговую репликацию транзакций будет лучше всего на примере. Допустим, что компания имеет несколько филиалов в разных городах и что серверы всех филиалов имеют одинаковый набор данных. Кроме этого, весь этот набор данных разбит на подмножества, и каждый сервер может обновлять только свое собственное подмножество данных. Когда данные изменяются на одном из этих серверов, эти изменения дублируются на всех других серверах (подписчиках) в одноранговой сети. (Пользователи в каждом филиале могут читать данные без каких-либо ограничений.)

Преимущества этого вида репликации следующие:

- вся система хорошо поддается масштабированию;
- вся система предоставляет высокий уровень доступности.

Хорошая масштабируемость достигается благодаря тому, что каждый сервер обслуживает только локальных пользователей. (Пользователи могут обновлять только тот раздел данных, который принадлежит их локальному серверу. Для операций чтения данные также хранятся локально.)

Высокий уровень доступности одноранговой репликации транзакций достигается благодаря тому, что если отключится один или несколько серверов, то все другие серверы будут продолжать работать, поскольку все требуемые им данные для операций чтения и записи хранятся локально. Когда отключившийся сервер снова подключается, запускается процесс репликации и сервер получает все изменения данных, которые произошли на других серверах.

Выявление конфликтов при одноранговой репликации

При одноранговой репликации допускается изменение данных любого узла, вследствие чего возможен конфликт изменений данных на разных узлах. (Конфликт может быть вызван изменением строки на нескольких узлах.)

Компонент Database Engine поддерживает возможность выявления конфликтов по всей настроенной топологии. Когда эта возможность разрешена, вызывающее конфликт изменение считается критической ошибкой, которая вызывает сбой агента Distribution Agent. В случае конфликта сценарий находится в неопределенном состоянии до тех пор, пока конфликт не будет разрешен и данные согласованы на всех участвующих серверах.

ПРИМЕЧАНИЕ

Возможность выявления конфликтов можно разрешить с помощью системных процедур `sp_addpublication` и `sp_configure_ peerconflictdetection`.

Конфликты в одноранговой репликации выявляются хранимыми процедурами, которые применяют изменения к каждому узлу, руководствуясь данными в скрытом столбце в каждой публикуемой таблице. Этот скрытый столбец содержит идентификатор, состоящий из однозначного идентификатора узла и версии строки. Процедуры выполняются агентом Distribution Agent и применяют полученные от других одноранговых узлов операции вставки, обновления и удаления. Если при чтении значения скрытого столбца одна из этих процедур обнаруживает конфликт, она вызывает ошибку.

ПРИМЕЧАНИЕ

Доступ к скрытому столбцу имеет только пользователь, который вошел в систему через подключение DAC (Dedicated Administrator Connection, выделенное подключение администратора). (Описание подключения DAC см. в *главе 15*.)

Когда в одноранговой репликации возникает конфликт, инициируется предупреждающее сообщение `Peer-to-peer conflict detection alert`. Это предупреждение следует настроить для отправки извещений при возникновении конфликтов. (Настройка предупреждений и способы извещения операторов рассмотрены в *главе 17*.) Кроме этого, в электронной документации описывается несколько подходов обработки возникающих конфликтов.

ПРИМЕЧАНИЕ

Даже несмотря на то, что в одноранговой репликации разрешено выявление конфликтов, их все же следует пытаться избегать.

Репликация моментальных снимков

Репликация моментальных снимков является самым простым типом репликации, при которой подлежащие публикации данные копируются с издателя всем подписчикам. Разница между репликацией моментальных снимков и репликацией транзакций состоит в том, что в первом случае подписчикам отправляются все опубликованные данные, а во втором — только изменения данных.

ПРИМЕЧАНИЕ

Репликации транзакций и моментальных снимков являются однонаправленными репликациями, а это означает, что изменения в реплицируемых данных осуществляются только на сервере публикаций. Поэтому данные на всех серверах подписки доступны только для чтения, за исключением изменений, вносимых процессом репликации.

В отличие от репликации транзакций репликация моментальных снимков не требует первичных ключей для таблицы. Причина этому очевидна: единицей передачи в репликации моментальных снимков является файл моментального снимка, а не строка таблицы. Другим различием между этими двумя типами репликаций является задержка по времени: репликация моментальных снимков осуществляется периодически, что означает значительную задержку, поскольку издатель отправляет подписчикам все опубликованные данные (измененные и неизмененные).

ПРИМЕЧАНИЕ

Репликация моментальных снимков напрямую не использует базу данных `distribution`. Однако эта база данных содержит информацию о состоянии и другие сведения, которые используются процессом репликации моментальных снимков.

Репликация слиянием

В репликации транзакций и моментальных снимков издатель отправляет данные, а подписчик их получает. (Возможность отправки реплицированных данных подписчиком издателю отсутствует.) Репликация слиянием (merge replication) позволяет издателю и подписчикам обновлять подлежащие репликации данные, вследствие чего в процессе репликации могут возникнуть конфликты.

При использовании сценария репликации слиянием система вносит в схему базы данных публикации три важных изменения:

- для каждой реплицированной строки определяется однозначный столбец;
- добавляется несколько системных таблиц;
- создаются триггеры для таблиц, в которых реплицируются данные.

Компонент Database Engine создает или идентифицирует однозначный столбец в таблице с реплицируемыми данными. Если базовая таблица уже содержит столбец с типом данных `UNIQUEIDENTIFIER` и свойством `ROWGUIDCOL`, для идентификации каждой реплицируемой строки система использует этот столбец. Если в таблице такой столбец отсутствует, то система добавляет столбец `rowguid` с типом данных `UNIQUEIDENTIFIER` и свойством `ROWGUIDCOL`.

ПРИМЕЧАНИЕ

Столбец с типом данных `UNIQUEIDENTIFIER` может содержать несколько одинаковых значений. Но свойство `ROWGUIDCOL` дополнительно указывает, что значения столбца с типом данных `UNIQUEIDENTIFIER` однозначно определяют строки в таблице. Поэтому столбец с типом данных `UNIQUEIDENTIFIER` и свойством `ROWGUIDCOL` содержит однозначные значения для каждой строки на всех компьютерах, объединенных в одну сеть во всем мире, гарантируя таким образом однозначность реплицируемых строк во всех множественных копиях таблицы на сервере издателя и серверах подписки.

Добавление новых системных таблиц предоставляет способ выявления и разрешения любых конфликтов при обновлении. Компонент Database Engine хранит все

изменения реплицируемых данных в системных таблицах слияния msmerge_contents и msmerge_tombstone и соединяет их с таблицей, содержащей реплицируемые данные, чтобы разрешить конфликт.

Компонент Database Engine создает триггеры для таблиц, содержащих реплицируемые данные, на всех сайтах для того, чтобы отслеживать изменения в каждой реплицируемой строке. Эти триггеры определяют вносимые в таблицу изменения и записывают их в системные таблицы msmerge_contents и msmerge_tombstone.

Конфликты выявляются агентом Merge Agent, на основе происхождения столбца в системной таблице msmerge_contents. Разрешение конфликта может быть или на основе приоритета или определяться индивидуально.

Разрешение конфликта на основе приоритета (priority-based conflict resolution) означает, что любой конфликт между новыми и старыми значениями в реплицируемой строке разрешается автоматически, на основе присвоенных приоритетов. (Особым случаем метода разрешения конфликтов на основе приоритетов является метод "выигрывает первый", когда первое по времени изменение реплицируемой строки становится победителем.) Метод на основе приоритетов применяется по умолчанию. Индивидуально определяемый метод разрешения конфликтов использует специализированные триггеры на основе бизнес-правил, определяемых администратором базы данных.

Модели репликации

В предшествующих разделах мы ознакомились с разными типами репликации, которые компонент Database Engine использует для распространения данных между различными узлами. Типы репликации (репликация транзакций, моментальных снимков и слиянием) предоставляют функциональность для обработки реплицируемых данных. *Модели репликации* используются компанией для разработки своих собственных реализаций репликации данных. Каждую модель репликации можно реализовать, используя один или несколько существующих типов репликации. Тип репликации и ее модель обычно указывается одновременно. В зависимости от требований может использоваться несколько моделей репликации. Среди основных моделей репликации можно назвать следующие:

- центральный издатель с распространителем;
- центральный издатель с удаленным распространителем;
- центральный подписчик с множественными издателями;
- множественные издатели с множественными подписчиками.

Эти модели репликации рассматриваются в последующих разделах.

Центральный издатель с распространителем

В модели центрального издателя с распространителем существует один издатель и обычно один распространитель, которые размещены на одном и том же экземпляре сервера Database Engine (см. рис. 18.1). Издатель создает публикации, которые рас-

пространитель отправляет множественным подписчикам. Публикации, создаваемые этой моделью и получаемые подписчиками, обычно имеют доступ "только для чтения".

Преимуществом этой модели является ее простота. По этой причине эта модель обычно применяется для создания копии базы данных, которая затем используется для исполнения интерактивных запросов и для создания простых отчетов. (Другой ситуацией для использования этой модели будет содержание удаленной копии базы данных, которую удаленные системы могут использовать в случае сбоя канала связи.)

С другой стороны, если экземпляр Database Engine настроен таким образом, что максимизированы все системные ресурсы, следует выбрать другую модель репликации данных.

Центральный издатель с удаленным распространителем

Издатель и распространитель могут находиться на одном сервере, если объем публикуемых данных не очень большой. В противном случае, по причине вопросов производительности рекомендуется использовать отдельные серверы для публикации и распространения. (При публикации больших объемов данных распространитель обычно является узким местом.) Модель репликации с центральным издателем и отдельным удаленным распространителем показана на рис. 18.2.

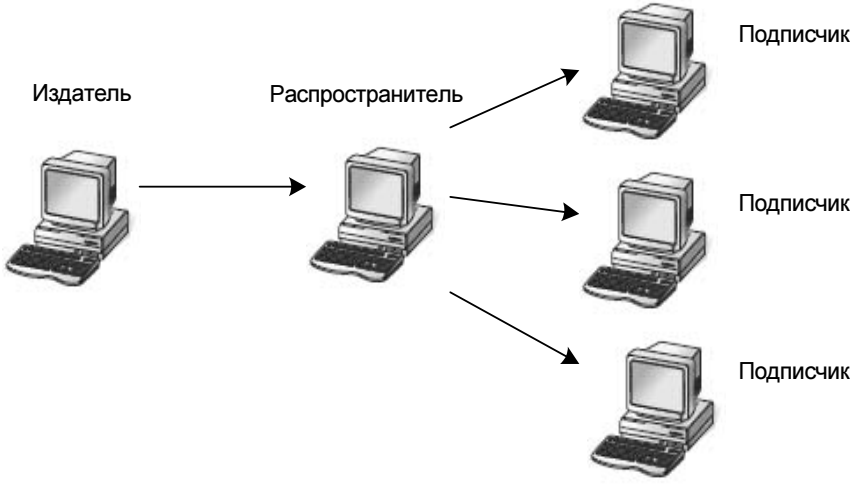


Рис. 18.2. Центральный издатель с удаленным распространителем

ПРИМЕЧАНИЕ

Этот сценарий можно использовать в качестве отправной точки для наращивания количества серверов публикации и/или серверов подписки.

Центральный подписчик с множественными издателями

Рассмотренный в начале главы сценарий с переезжающим с места на место продавцом (коммивояжером), который передает данные в центральный офис, является типичным примером центрального подписчика с множественными издателями. Отправляемые множественными издателями данные собираются на центральном подписчике.

Для этой модели можно использовать или одноранговую репликацию транзакций или репликацию слиянием, в зависимости от применения реплицируемых данных. Если издатели публикуют (и, следственно, обновляют) на подписчике одинаковые данные, то следует использовать репликацию слиянием. А если каждый издатель публикует свои собственные данные, то следует использовать одноранговую репликацию транзакций. В этом случае публикуемые таблицы будут фильтроваться горизонтально, а каждый издатель будет владеть определенным фрагментом таблицы.

Множественные издатели с множественными подписчиками

Модель репликации, в которой некоторые или все серверы, участвующие в процессе репликации, играют роль издателя и подписчика, называется моделью с множественными издателями и множественными подписчиками. В большинстве случаев эта модель включает несколько распространителей, которые обычно размещаются на каждом издателе (рис. 18.3).

Рис. 18.3. Множественные издатели с множественными подписчиками

Эту модель можно реализовать, только используя репликацию слиянием, поскольку публикации изменяются на каждом сервере издателя. (Единым другим способом реализовать эту модель будет использование распределенных транзакций с двухфазной фиксацией.)

Управление репликацией

Все участвующие в репликации серверы баз данных должны быть зарегистрированы. (Регистрация серверов рассмотрена в *главе 3*.) После регистрации серверов баз данных необходимо настроить сервер распространения, сервер(ы) публикации и сервер(ы) подписки. Настройка этих серверов, используя соответствующие мастера, рассматривается в последующих разделах.

ПРИМЕЧАНИЕ

Для репликации вместо учетной записи администратора следует использовать выделенную учетную запись.

Настройка серверов распространения и публикации

Прежде чем устанавливать базы данных публикаций, необходимо установить сервер распространения и настроить базу данных `distribution`. Сервер распространения можно установить с помощью мастера Configure Distribution Wizard. Этот мастер позволяет настроить сервер распространителя и базу данных `distribute` и сделать доступным издателя(ей). В частности, посредством этого мастера вы можете:

- настроить сервер в роли распространителя, который могут использовать другие издатели;
- настроить сервер в роли издателя, который мог бы работать и как собственный распространитель;
- настроить сервер в роли издателя, который также используется другим сервером в качестве распространителя.

В этом разделе рассматривается сценарий репликации базы данных `sample`, используя компьютеры `NTB00716` и `NTB01101`. Первый компьютер будет выполнять роль издателя и распространителя, а второй будет подписчиком. Первым шагом будет использование мастера Configure Distribution Wizard для настройки сервера `NTB00716` в качестве издателя, который также является и собственным распространителем. (Кроме того, с помощью этого мастера также создадим базу данных `distribution`.)

ПРИМЕЧАНИЕ

Для настройки сервера распространения и базы данных `distribution` можно также использовать системные процедуры `sp_adddistributor` и `sp_adddistributiondb`. Системная процедура `sp_adddistributor` устанавливает сервер распространения, создавая новую строку в системной таблице `sysservers`, а процедура `sp_adddistributiondb` создает новую базу данных `distribution` и устанавливает схему распространения.

Чтобы запустить мастер Configure Distribution Wizard, запустите среду Management Studio, разверните экземпляр сервера, щелкните правой кнопкой папку **Replication** и в контекстном меню выберите пункт **Configure Distribution**. Откроется началь-

ное окно мастера Configure Distribution Wizard. Перейдите на страницу **Distributor**, выберите в качества сервера распространения сервер `NTB00716` и нажмите кнопку **Next**. Затем, на странице **Snapshot Folder**, укажите папку для хранения моментальных снимков от издателей, использующих сервер распространения, и нажмите кнопку **Next**. На странице **Distribution Database** выберите имя базы данных `distribution` и файлы журнала а затем нажмите кнопку **Next**. На странице **Publishers** разрешите издателя (в данном примере это будет сервер `NTB00716`), выберите, завершить ли процесс настройки немедленно или же создать файл сценария, чтобы выполнять настройку распространителя позже, а затем нажмите кнопку **Next**. Откроется окно мастера (рис. 18.4) с итоговым обозрением всех шагов, выполненных для настройки сервера `NTB00716` в качестве распространителя и издателя.

ПРИМЕЧАНИЕ

Используя мастер Disable Publishing and Distribution Wizard, уже существующие на сервере издатель и распространитель могут быть запрещены. Для запуска мастера щелкните правой кнопкой папку **Replication** и в контекстном меню выберите пункт **Disable Publishing and Distribution**.

После настройки серверов распространения и публикации требуется настроить процесс публикации. Эта настройка осуществляется с помощью мастера новых публикаций New Publication Wizard и рассматривается в следующем разделе.

Рис. 18.4. Страница **Complete the Wizard** мастера Configure Distribution Wizard

Настройка публикаций

Посредством мастера новых публикаций New Publication Wizard можно выполнять следующие действия:

- выбрать данные и объекты базы данных для репликации;
- отфильтровать публикуемые данные, чтобы подписчики получали только те данные, которые им требуются.

Предположим, что нам необходимо опубликовать данные таблицы `employee` с сервера `NTB00716` на сервер `NTB00716`, используя репликацию моментальных снимков. В данном случае единицей публикации является вся таблица `employee`.

Чтобы создать публикацию, разверните узел сервера публикаций (в данном примере это будет сервер `NTB00716`), в нем разверните папку **Replication**, в ней щелкните правой кнопкой папку **Local Publications** и в контекстном меню выберите пункт **New Publication**. Откроется мастер новых публикаций New Publication Wizard. На первых двух страницах этого мастера выберите базу данных для публикации (`sample`) и тип публикации (в данном примере это будет публикация моментальных снимков) и нажмите кнопку **Next**. Затем выберите, по крайней мере, один объект для публикации (в данном примере выберите всю таблицу `employee`) и нажмите кнопку **Next**. Мастер новых публикаций также позволяет отфильтровать горизонтально (по строкам) или вертикально (по столбцам) данные для публикации. Моментальный снимок выбранных данных можно выполнить сразу же и/или заплани-

Рис. 18.5. Страница **Complete the Wizard** для единицы публикации

ровать его на периодическое выполнение позже. (В нашем примере мы создадим моментальный снимок сразу же.)

На странице **Agent Security** установите настройки безопасности для агента моментальных снимков Snapshot Agent. Для этого нажмите кнопку **Security Settings** и в открывшемся диалоговом окне **Snapshot Agent Security** введите учетную запись пользователя Windows, под которой должен выполняться процесс агента моментальных снимков. (Учетную запись пользователя требуется предоставить в формате *имя_домена\имя_учетной_записи_пользователя*.) Указав учетную запись, нажмите кнопку **Next**. В открывшейся странице **Wizard Action** можно выбрать немедленное окончание процесса конфигурирования и/или создать файл сценария, чтобы создать публикацию позже. Выбрав требуемое действие, нажмите кнопку **Next**. Откроется страница **Complete the Wizard** (рис. 18.5) с кратким изложением всех шагов, предпринятых для настройки таблицы `employee` в качестве единицы публикации.

Последним шагом будет настройка серверов подписки, что и рассматривается в следующем разделе.

Настройка серверов подписки

Разрешение издателю возможности оформлять подписку является задачей, которая относится к подписчикам, но должна выполняться издателем. Для выполнения этой задачи применяется среда Management Studio. Сначала разверните узел сервера публикации, в нем разверните папку **Replication**, в ней щелкните правой кнопкой папку **Local Subscriptions** и в контекстном меню выберите пункт **New Subscriptions**. Будет запущен мастер новых подписок New Subscription Wizard, посредством которого можно:

- создать одну или более подписок на публикацию;
- указать, где и когда выполнять агентов синхронизации подписки.

На странице **Publication** мастера укажите публикацию, на которую требуется создать одну или более подписок, а затем нажмите кнопку **Next**. (Для данного примера следует выбрать публикацию `replicate_employee`, которая была создана ранее мастером новых публикаций.)

На странице **Distribution Agent Location** нужно выбрать требуемый тип подписки: принудительную или по запросу. Принудительная подписка означает центральное управление подписками. Для этого типа репликации установите переключатель **Run all agents at the Distributor** (выполнять всех агентов на распространителе). Чтобы указать подписку по запросу, установите переключатель **Run each agent at its Subscriber** (выполнять каждого агента на его подписчике). Выбрав тип подписки, нажмите кнопку **Next**. (Так как в данном примере подписка отправляется из центрального издателя, установим первый переключатель.)

На странице **Subscribers** нужно указать все серверы подписки. Если на этой странице еще нет серверов подписки, нажмите кнопку **Add Subscriber**, выберите во вложенном меню пункт **Add SQL Server Subscriber**, выберите все серверы, на которые будет выполняться репликация данных, а затем нажмите кнопку **Next**. В завершении работы мастера, на странице **Complete the Wizard**, будут отображены итоговые данные всех шагов, выполненных для настройки подписки.

Резюме

Репликация данных является предпочтительным способом распространения данных, вследствие меньших расходов, чем при использовании распределенных транзакций. В зависимости от используемой модели репликации, компонент Database Engine позволяет выбрать один из четырех типов репликации: репликацию моментальных снимков, репликацию транзакций, репликацию слиянием или одноранговую репликацию. Теоретически, в любой из моделей репликации можно использовать любой тип репликации, хотя каждая (базовая) модель репликации имеет соответствующий тип репликации, который используется с ней в большинстве случаев.

Публикация является наименьшей единицей репликации. Одна база данных может иметь несколько публикаций разного типа репликаций. Однако каждая публикация может соответствовать только одной базе данных.

Чтобы настроить процесс репликации, сначала нужно организовать сервер распространения и базу данных системы распространения и настроить сервер(ы) публикации, а затем определить одну или несколько публикаций. В завершении процесса нужно настроить сервер(ы) подписки. Для выполнения этих задач компонент Database Engine предоставляет три мастера: мастер настройки распространения Configure Distribution Wizard, мастер новых публикаций New Publication Wizard и мастер новых подписок New Subscription Wizard.

В следующих двух главах обсуждается общая производительность системы. В *главе 19* объясняется, как работает оптимизатор запросов компонента Database Engine, а в *главе 20* рассматривается настройка производительности. Эти главы завершают *часть III* данной книги.

Упражнения

Упражнение 18.1

Зачем для репликации требуется первичный ключ? Для каких типов репликации требуется этот ключ?

Упражнение 18.2

Как можно уменьшить объем передаваемых по сети данных и/или размер базы данных?

Упражнение 18.3

При обновлениях рекомендуется избегать конфликтов. Как можно минимизировать число конфликтов?

Упражнение 18.4

Какую задачу выполняют агенты Log Reader Agent, Merge Agent и Snapshot Agent соответственно?

Глава 19

Оптимизатор запросов

- **Этапы обработки запроса**
- **Как работает оптимизация запросов**
- **Инструменты для редактирования стратегии оптимизатора**
- **Подсказки по оптимизации**

При выполнении запроса компонентом Database Engine (или любой другой реляционной системой баз данных) обычно возникает вопрос получения доступа и обработки требуемых для запроса данных с максимальной эффективностью. Компонент системы баз данных, ответственный за это, называется *оптимизатором запросов* (query optimizer).

Задача оптимизатора запросов (или просто оптимизатора) состоит в рассмотрении различных возможных стратегий для выборки данных для конкретного запроса и выбора наиболее эффективной стратегии. Выбранная стратегия называется *планом выполнения запроса*. Оптимизатор принимает решения, принимая во внимание такие факторы, как размер таблиц, к которым направлен запрос, существующие индексы и логические операторы (`AND`, `OR` или `NOT`), используемые в предложении `WHERE`. Совместно эти факторы называются *статистическими данными*.

В начале этой главы приводится описание этапов обработки запроса, а затем подробно излагается работа третьего этапа — оптимизация запроса. Это закладывает основу для практических примеров, рассматриваемых в последующих разделах. После этого рассматриваются разные инструментальные средства для редактирования работы оптимизатора запросов. В конце главы обсуждаются подсказки оптимизации, которые можно предоставлять оптимизатору в особых случаях, когда он не может найти наилучшего решения.

Этапы обработки запроса

Задача оптимизатора заключается в выработке наиболее эффективного плана выполнения для данного запроса. Эта задача выполняется в следующие четыре этапа, которые показаны на рис. 19.1.

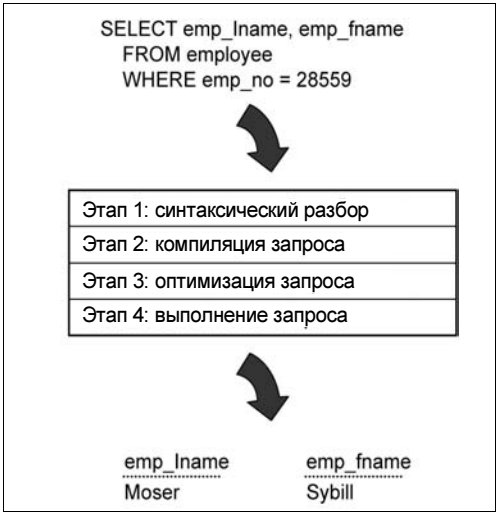


Рис. 19.1. Этапы обработки запроса

ПРИМЕЧАНИЕ

В этой главе рассматривается работа оптимизатора для запросов инструкции `SELECT`. Оптимизатор также применяется для инструкций `INSERT`, `UPDATE` и `DELETE`. Инструкция `INSERT` может содержать подзапрос, а инструкции `UPDATE` и `DELETE` часто содержат предложение `WHERE`.

1. **Синтаксический разбор (parsing).** Проверяется синтаксис запроса и запрос преобразовывается в дерево. После этого выполняется проверка всех объектов базы данных, на которые ссылается запрос. (Например, проверяется существование всех столбцов, на которые ссылается запрос, и определяются их идентификаторы.) После завершения процесса проверки создается окончательное дерево запроса.
2. **Компиляция запроса (query compilation).** Дерево запроса компилируется оптимизатором запросов.
3. **Оптимизация запроса (query optimization).** Оптимизатор запросов в качестве вводных данных принимает скомпилированное дерево запроса, созданное на предыдущем этапе, и исследует несколько стратегий обращения к данным, прежде чем решить, как обрабатывать данный запрос. Чтобы найти наиболее эффективный план выполнения, оптимизатор запроса сначала выполняет анализ запроса, в процессе которого он отыскивает аргументы поиска и операции

соединения. Затем оптимизатор решает, какие индексы использовать. Наконец, если запрос содержит операции соединения, оптимизатор выбирает порядок выполнения соединений и методы их обработки. (Эти этапы процесса оптимизации подробно рассматриваются в следующем разделе.)

4. **Выполнение запроса (query execution).** Созданный план выполнения сохраняется на перманентной основе и затем выполняется.

ПРИМЕЧАНИЕ

Для некоторых инструкций можно избежать этапов синтаксического разбора и оптимизации, если компоненту Database Engine известно, что существует только один выполнимый план. Такая оптимизация называется тривиальной оптимизацией плана. В качестве примера запроса, для которого применима тривиальная оптимизация плана, можно назвать простую форму инструкции `INSERT`.

Как работает оптимизация запроса

Как вы уже узнали в предыдущем разделе, этап оптимизации запроса можно разделить на следующие этапы:

- анализ запроса;
- выбор индекса;
- выбор порядка выполнения операций соединения;
- выбор метода выполнения операций соединения.

Эти этапы оптимизации запроса описываются в последующих разделах. Кроме этого, в конце данного раздела рассматривается кэширование плана.

Анализ запроса

В процессе анализа запроса оптимизатор исследует его на наличие аргументов поиска, использование оператора `OR` и существование критериев соединения, в приведенном порядке. Так как вопросы использования оператора `OR` и существования критериев соединения понятны сами по себе, будет рассмотрен только вопрос аргументов поиска.

Аргумент поиска — это часть запроса, которая ограничивает промежуточный результирующий набор запроса. Основным назначением аргументов поиска является то, что они позволяют использовать существующие индексы применительно к конкретному выражению. В качестве примеров аргументов поиска можно привести следующие выражения:

- `emp_fname = 'Moser'`;
- `salary >= 50000`;
- `emp_fname = 'Moser' AND salary >= 50000`.

Существует несколько типов выражений, которые оптимизатор не может использовать в качестве аргументов поиска. Первую группу таких выражений составляют

выражения с оператором отрицания NOT (<>). Кроме этого, если с левой стороны оператора используется выражение, то все выражение не может быть аргументом поиска.

В качестве примеров выражений, не являющихся аргументами поиска, можно привести следующие:

- NOT IN('d1', 'd2');
- emp_no <> 9031;
- budget * 0.59 > 55000.

Основным недостатком выражений, которые нельзя использовать в качестве аргументов поиска, является то, что оптимизатор не может использовать существующие индексы применительно к выражению, чтобы ускорить выполнение соответствующего запроса. Иными словами, в данном случае единственным способом доступа оптимизатора к данным будет сканирование таблицы.

Выбор индексов

Идентификация аргументов поиска позволяет оптимизатору принять решение о том, можно ли использовать один или более существующих индексов. На этом этапе оптимизации, оптимизатор исследует каждый аргумент поиска на наличие индексов, имеющих отношение к соответствующему выражению. Если индекс имеется, оптимизатор решает, использовать его или нет. Это решение в основном зависит от селективности соответствующего выражения. Под *селективностью* (selectivity) выражения имеется в виду соотношение количества строк, удовлетворяющих условию, к общему количеству строк в таблице.

Оптимизатор проверяет селективность выражения с индексированным столбцом, используя *статистические данные*, которые создаются для распределения значений в столбце. Оптимизатор запросов использует эту информацию, чтобы определить оптимальный план выполнения запроса, оценивая стоимость использования индекса для выполнения запроса.

В последующих разделах дается подробное описание селективности выражения с индексированным столбцом и статистическими данными. (Так как статистические данные существуют как для индексов, так и для столбцов, они рассматриваются отдельно в двух соответствующих разделах.)

ПРИМЕЧАНИЕ

Если активирована опция AUTO_CREATE_STATISTICS, компонент Database Engine вычисляет статистические данные (для индексов и столбцов) автоматически. (Опция AUTO_CREATE_STATISTICS рассматривается далее в этой главе.)

Селективность выражения с индексированным столбцом

Как уже упоминалось ранее, оптимизатор использует индексы для улучшения времени выполнения запроса. Когда таблица, по которой выполняется запрос, не со-

держит индексов или если оптимизатор принимает решение не использовать существующие индексы, система выбирает данные, сканируя таблицу. В процессе сканирования таблицы компонент Database Engine последовательно считывает страницы данных таблицы, чтобы найти строки, относящиеся к результирующему набору. *Доступ по индексу* — это метод доступа, при котором система баз данных читает и записывает данные на страницах, используя существующий индекс. Так как доступ по индексу значительно уменьшает количество операций ввода/вывода при чтении, он часто более эффективен, чем последовательное сканирование таблицы.

Для поиска данных компонент Database Engine использует некластеризованный индекс двумя способами. Если у вас имеется *куча* (т. е. таблицы без кластеризованного индекса), то система сначала выполняет обход некластеризованного индекса, а затем извлекает строку, используя идентификатор строки. Но в случае кластеризованной таблицы, после обхода структуры некластеризованного индекса следует обход структуры кластеризованного индекса таблицы. С другой стороны, использование кластеризованного индекса для поиска данных всегда уникально: Database Engine начинает с корня соответствующего B^+-дерева и обычно после трех-четырех операций чтения достигает уровня узлов листьев, где находятся данные. По этой причине обход структуры кластеризованного индекса почти всегда выполняется быстрее, чем обход структуры некластеризованного индекса.

Из изложенного материала можно видеть, что на вопрос, какой метод доступа является наиболее быстрым (сканирование индекса или сканирование таблицы), нет прямого ответа, поскольку все зависит от селективности и типа индекса.

Тесты, которые были выполнены автором книги, показали, что доступ сканированием таблицы часто начинает выполняться быстрее, чем доступ с использованием некластеризованного индекса, когда выбирается, по крайней мере, 10% строк. В этом случае решение оптимизатора, когда переключаться от доступа по некластеризованному индексу к сканированию таблицы, не следует исправлять. (Если имеются основания полагать, что оптимизатор переключается на сканирование таблицы преждевременно, то его решение можно изменить, используя в запросе подсказку `INDEX`, как это рассматривается далее в этой главе.)

По нескольким причинам производительность поиска с помощью кластеризованного индекса лучше, чем с использованием некластеризованного индекса. Прежде всего, при сканировании кластеризованного индекса системе не требуется покидать структуру B^+-дерева, чтобы сканировать страницы данных, т. к. они находятся в этом же индексе на уровне листьев этого дерева. Кроме этого, для некластеризованного индекса требуется больше операций ввода/вывода, чем для соответствующего кластеризованного индекса. Причиной этому является то обстоятельство, что для некластеризованного индекса после обхода B^+-дерева требуется или выполнять чтение страниц данных, или, если для столбца другой таблицы существует кластеризованный индекс, некластеризованному индексу требуется выполнить чтение B^+-дерева кластеризованного индекса.

Поэтому можно ожидать, что доступ с использованием кластеризованного индекса будет немного быстрее, чем последовательное сканирование таблицы, даже при плохой селективности (т. е. при высоком процентном отношении возвращенных

строк, по причине возврата запросом большого количества строк). Тесты, выполненные автором книги, показали, что при селективности выражения на уровне 75% или меньше доступ посредством кластеризованного индекса обычно быстрее, чем последовательное сканирование таблицы.

Статистические данные индекса

Статистические данные индекса обычно создаются при создании индекса для определенного столбца (или столбцов). Создание статистических данных для индекса означает, что компонент Database Engine создает *гистограмму* на основе до 200 значений столбца. (Следственно, создается до 199 интервалов.) В этой гистограмме отображается, среди прочего, количество точных совпадений строк для каждого интервала, среднее количество строк с разными значениями в каждом интервале, а также плотность значений.

ПРИМЕЧАНИЕ

Статистические данные индекса всегда создаются для одного столбца. В случае составного индекса (по нескольким столбцам) система создает статистические данные для первого столбца индекса.

Создать статистические данные явно можно с помощью следующих средств:

- системной процедуры `sp_createstats`;
- среды SQL Server Management Studio.

Системная процедура `sp_createstats` создает статистические данные по одиночному столбцу для всех столбцов всех пользовательских таблиц в текущей базе данных. Созданным статистическим данным присваивается имя столбца, на котором они созданы.

Для создания статистических данных с помощью среды SQL Server Management Studio разверните последовательно узел сервера, папку **Databases**, требуемую базу данных, папку **Table**, требуемую таблицу, затем щелкните правой кнопкой папку **Statistics** и в контекстном меню выберите пункт **New Statistics**. Откроется диалоговое окно **New Statistics on Table**, в котором следует указать имя создаваемого набора статистических данных. Затем нажмите кнопку **Add** и в открывшемся диалоговом окне **Select Columns** укажите столбец или столбцы таблицы, для которых следует создать набор статистических данных, и нажмите кнопку **OK** окна. Чтобы сохранить созданный набор статистических данных, нажмите кнопку **OK** диалогового окна **New Statistics on Table**.

По мере изменения данных столбца, статистические данные индекса устаревают. Устаревшие статистические данные могут в значительной мере отрицательно влиять на производительность запроса. Компонент Database Engine может автоматически обновлять статистические данные индекса, для чего следует активировать опцию `AUTO_UPDATE_STATISTICS` (т. е. присвоить ей значение `ON`). Тогда, если требуемые запросу статистические данные устарели, они будут автоматически обновлены в процессе оптимизации запроса.

Другая опция, AUTO_CREATE_STATISTICS, автоматически создает любые недостающие статистические данные, требуемые для оптимизации запроса. Обе эти опции можно активировать (или деактивировать) с помощью инструкции ALTER DATABASE или среды SQL Server Management Studio.

Статистические данные столбца

Как вы уже знаете из предыдущих разделов, компонент Database Engine создает наборы статистических данных для всех существующих индексов. Система также может создавать статистические данные и для неиндексированных столбцов. Такие статистические данные называются *статистическими данными столбца* (column statistics). Для оптимизации планов выполнения запросов статистические данные столбцов применяются совместно со статистическими данными индексов. Компонент Database Engine создает статистические данные для неиндексированного столбца, который входит в условие предложения WHERE.

Существует несколько ситуаций, в которых наличие статистических данных столбцов может помочь оптимизатору принять правильное решение. Одной из таких ситуаций будет наличие составного индекса по двум или больше столбцам. Для таких индексов система создает статистические данные только для первого столбца индекса. Наличие статистических данных столбца для второго столбца (и всех других столбцов) составного индекса может помочь оптимизатору выбрать оптимальный план выполнения.

Компонент Database Engine поддерживает два представления каталога для работы со статистическими данными столбца (эти представления можно также использовать для редактирования информации, связанной со статистическими данными индекса):

- sys.stats;
- sys.stats_columns.

Представление sys.stats содержит строку для каждого набора статистических данных таблицы или представления. Кроме столбца name, в котором указывается имя статистики, это представление каталога содержит, среди прочих, следующие два столбца:

- столбец auto_created — содержит статистические данные, созданные оптимизатором запросов;
- столбец user_created — содержит статистические данные, созданные пользователем.

Представление sys.stats_columns содержит дополнительную информацию о столбцах представления sys.stats. В частности, оно содержит по одной строке для каждого столбца, являющегося частью представления sys.stats. (Чтобы получить доступ к этой дополнительной информации, нужно соединить эти два представления.)

Выбор порядка соединения

Обычно порядок, в котором две или более соединяемые таблицы записываются в предложении FROM инструкции SELECT, не оказывает влияния на принимаемое оптимизатором решение относительно порядка их обработки.

Как мы узнаем в следующем разделе, на решение оптимизатора о том, к какой таблице обратиться первой, влияет много разных факторов. С другой стороны, порядок выбора таблиц можно явно указать, используя подсказку FORCE ORDER (которая рассматривается далее в этой главе).

Выбор метода выполнения соединения

Операция соединения является самой трудоемкой операцией обработки запроса. Компонент Database Engine поддерживает следующие методы обработки соединения, позволяя оптимизатору выбрать один из них на основе статистики для обеих таблиц:

- вложенный цикл;
- соединение слиянием;
- соединение хешированием.

Эти методы выполнения соединения рассматриваются в следующих подразделах.

Вложенный цикл

Метод вложенного цикла основан на применении "грубой силы", или полного перебора. Иными словами, для каждой строки внешней таблицы извлекается и сравнивается каждая строка внутренней таблицы. Метод вложенного цикла для обработки соединения двух таблиц показан в примере 19.1.

В примере A и B являются двумя таблицами.

Пример 19.1. Обработка двух таблиц посредством вложенного цикла

```
for each row in the outer table A do:
     read the row
     for each row in the inner table B do:
        read the row
           if A.join_column = B.join_column then
              accept the row and add it to the resulting set
           end if
     end for
end for
```

В примере 19.1 обращение к каждой строке из внешней таблицы (таблица A) вызывает обращение ко всем строкам внутренней таблицы (таблица B). Затем сравниваются значения столбцов соединения, и если значения обоих столбцов совпадают, то строка добавляется в результирующий набор запроса.

Метод вложенного цикла работает очень медленно, если для столбца соединения внутренней таблицы не существует индекса. При отсутствии такого индекса компоненту Database Engine требуется выполнить сканирование внешней таблицы один раз, а внутренней *n* раз, где *n* — количество строк внешней таблицы. По этой причине оптимизатор запросов обычно выбирает этот метод в том случае, если соединяемый столбец во внутренней таблице проиндексирован так, что для внутренней таблицы не нужно выполнять сканирование для каждой строки внешней таблицы.

Соединение слиянием

Метод соединения слиянием предоставляет экономичную альтернативу созданию индекса вложенным циклам. Строки соединяемых таблиц должны быть физически упорядочены с использованием значений столбца соединения. Затем выполняется сканирование обеих таблиц в порядке столбцов соединения, сопоставляя строки с одинаковыми значениями для столбцов соединения. Метод слияния для обработки соединения двух таблиц показан в примере 19.2.

В примере внешнюю таблицу `A` сортируем в возрастающем порядке с использованием столбца соединения.

Внутреннюю таблицу `B` также сортируем в возрастающем порядке с использованием столбца соединения.

Пример 19.2. Соединение двух таблиц методом слияния

```
for each row in the outer table A do:
   read the row
   for each row from the inner table B with a value
      less than or equal to the join column do:
         read the row
         if A.join_column = B.join_column then
             accept the row and add it to the resulting set
         end if
   end for
end for
```

Метод соединения двух таблиц слиянием имеет высокие накладные расходы, если строки в обеих таблицах не отсортированы. Но этот метод является предпочтительным, когда таблицы предварительно отсортированы. (Таблицы всегда отсортированы таким образом, когда оба столбца соединения являются первичными ключами соответствующих таблиц, поскольку компонент Database Engine по умолчанию создает кластеризованный индекс для первичного ключа таблицы.)

Соединение хешированием

Метод соединения хешированием обычно применяется при отсутствии индексов для столбцов соединения. В случае использования этого метода обе соединяемые

таблицы рассматриваются как два потока ввода: компонуемый ввод и контрольный ввод. (В качестве компонуемого ввода обычно используется наименьшая таблица.) Процесс работает следующим образом:

1. Значение соединяемого столбца из строки компонуемого ввода сохраняется в определенном сегменте хеша в зависимости от числа, возвращаемого алгоритмом хеширования.
2. После обработки всех строк из компонуемого ввода начинается обработка строк из контрольного ввода.
3. Каждое значение соединяемого столбца строки из контрольного ввода обрабатывается с использованием того же самого алгоритма хеширования.
4. Соответствующие строки извлекаются из каждого сегмента хеша и затем используются для создания результирующего набора.

ПРИМЕЧАНИЕ

Метод соединения хешированием не требует никакого индекса. Поэтому этот метод хорошо подходит для незапланированных запросов, для которых не предполагается наличие индексов. Кроме этого, если оптимизатор использует этот метод, то это может служить подсказкой о необходимости создания дополнительных индексов для одного или более соединяемых столбцов.

Кэширование планов

Компонент Database Engine использует набор кэшей для хранения данных и планов выполнения запросов. При первом выполнении запроса его скомпилированная версия сохраняется в памяти. (Эта память для хранения скомпилированных планов запроса называется *кэшем планов* — plan cache.) При повторном выполнении запроса компонент Database Engine проверяет, нет ли для него плана в кэше планов. Если такой план имеется, то повторная компиляция запроса не выполняется.

ПРИМЕЧАНИЕ

Процесс кэширования планов хранимых процедур аналогичен кэшированию планов запросов.

Редактирование планов выполнения

Существует несколько способов редактирования планов выполнения. В этом разделе мы рассмотрим два из таких способов:

- опция `optimize for ad hoc workloads`;
- инструкция `DBCC FREEPROCCACHE`.

Опция расширенного конфигурирования `optimize for ad hoc workloads` предотвращает размещение системой плана выполнения в кэш при первом выполнении соответствующего элемента. Вместо всего плана выполнения компонент Database

Engine помещает в кэш только заглушку плана. Эта заглушка содержит минимальную информацию, которая требуется системе, чтобы найти совпадения с будущими запросами. Целью этого подхода является уменьшение неуправляемого роста кэша планов. (Имейте в виду, что для плана выполнения простого запроса с парой индексированных столбцов в списке выборки `SELECT` требуется около 20 Кбайт памяти. Для планов сложных запросов может потребоваться значительно больше памяти.)

Инструкция `DBCC FREEPROCCACHE` удаляет из кэша все планы. Эта команда может быть полезной для тестирования. Иными словами, если необходимо определить, какие планы кэшируются (т. е. когда планы определенных запросов используются повторно), с помощью этой команды можно очистить кэш. (С помощью этой команды возможно также удалить из кэша определенный план, указав соответствующий параметр для идентификатора плана.)

Отображение информации о кэше планов

Информацию о кэше планов можно получить с помощью следующих динамических административных представлений:

- `sys.dm_exec_cached_plans`;
- `sys.dm_exec_query_stats`;
- `sys.dm_exec_sql_text`.

Все эти представления рассматриваются в *разд. "Динамические административные представления и оптимизатор запросов" далее в этой главе*.

Инструменты для редактирования стратегии оптимизатора

Компонент Database Engine предоставляет набор инструментов, позволяющих редактировать работу оптимизатора запросов. В этот набор, кроме прочих, входят следующие инструменты:

- инструкция `SET` (для отображения планов выполнения в виде простого текста или в формате XML);
- среда Management Studio (для графического отображения планов выполнения);
- динамические административные представления и функции;
- приложение SQL Server Profiler (подробно рассматривается в *главе 20*).

Первые три из этих инструментов рассматриваются в последующих разделах.

ПРИМЕЧАНИЕ

Почти во всех примерах этой главы используется база данных `AdventureWorks2012`. Если у вас еще нет этой базы данных, то ее можно загрузить, руководствуясь инструкциями во введении к этой книге.

Инструкция *SET*

Для понимания различных опций инструкции SET нужно знать, что существует три разных следующих формата отображения плана выполнения запроса:

- текстовый формат;
- формат XML;
- графический формат.

Для первых двух форматов применяется инструкция SET, поэтому они рассматриваются в следующих подразделах. Графическая форма отображения планов выполнения запроса обсуждается в *разд. "Среда Management Studio и графические планы выполнения" далее в этой же главе*.

Текстовый план выполнения

Термин *текстовый план выполнения* означает, что план выполнения запроса отображается построчно в текстовом формате. Таким образом, вывод текстового плана выполнения возвращается в виде строки. Зависимости между исполняющимися операциями обозначаются вертикальными черточками. Для отображения текстового плана выполнения используются следующие опции инструкции SET:

- SHOWPLAN_TEXT;
- SHOWPLAN_ALL.

Текстовый план выполнения запроса можно отобразить, активировав опцию SHOWPLAN_TEXT или SHOWPLAN_ALL (т. е. присвоив опции значение ON), перед соответствующей инструкцией SELECT. Опция SHOWPLAN_ALL задает отображение такой же самой подробной информации о выбранном плане выполнения для запроса, что и опция SHOWPLAN_TEXT, но с добавлением оценки требований ресурсов для выполнения инструкции.

Использование опции SHOWPLAN_TEXT показано в примере 19.3.

ПРИМЕЧАНИЕ

Активирование опции SHOWPLAN_TEXT отключает выполнение всех последующих инструкций языка Transact-SQL до тех пор, пока вы не деактивируете эту опцию, выполнив инструкцию SET SHOWPLAN_TEXT OFF. (Опция SHOWPLAN_XML имеет такое же свойство.)

Пример 19.3. Отображение текстового плана запроса посредством опции SHOWPLAN_TEXT

```
SET SHOWPLAN_TEXT ON;
GO
USE AdventureWorks2012;
```

```
SELECT * FROM HumanResources.Employee e JOIN Person.Address a
          ON e.BusinessEntityID = a.AddressID
          AND e.BusinessEntityID = 10;
GO
SET SHOWPLAN_TEXT OFF;
```

Результат выполнения запроса в примере 19.3 будет следующим:

```
|--Nested Loops(Inner Join)
   |--Clustered Index Seek(OBJECT:([AdventureWorks2012].[Person].
      [Address].[PK_Address_AddressID] AS [a]), SEEK:
      ([a].[AddressID]=(10)) ORDERED FORWARD)
   |--Compute Scalar(DEFINE:([e].[OrganizationLevel]=
      [AdventureWorks2012].[HumanResources].[Employee].
      [OrganizationLevel] as [e].[OrganizationLevel]))
      |--Compute Scalar(DEFINE:([e].[OrganizationLevel]=
         [AdventureWorks2012].[HumanResources].[Employee].
         [OrganizationNode] as [e].[OrganizationNode].GetLevel()))
         |--Clustered Index Seek(OBJECT:([AdventureWorks2012].
           [HumanResources].[Employee].[PK_Employee_BusinessEntityID]
            AS [e]), SEEK:([e].[BusinessEntityID]=(10))
            ORDERED FORWARD)
```

Перед тем как обсуждать план выполнения, отображенный в результате выполнения примера 19.3, нужно понимать, как его интерпретировать. Прежде всего, текст после символов `|--` располагается в одной строке. В книге он приведен в нескольких строках вследствие недостатка места в одной строке. Далее, отступ операторов (выделены жирным шрифтом), с которых начинается каждая строка, определяет порядок их выполнения: оператор с самым большим отступом выполняется первым. Если два или более операторов имеют одинаковый отступ, они выполняются в порядке сверху вниз. Теперь начнем разбор результата выполнения примера 19.3. Как можно видеть, в нем имеется три оператора: `Nested Loops`, `Compute Scalar` и `Clustered Index Seek`. (Перед всеми операторами стоит символ вертикальной черточки: `|`.) Первым выполняется оператор `Clustered Index Seek` для таблицы `Employee`. За ним исполняются операторы `Compute Scalar` для таблицы `Employee` и `Clustered Index Seek` для таблицы `Address`. В конце выполняется соединение обеих таблиц (`Employee` и `Address`), используя метод вложенного цикла.

ПРИМЕЧАНИЕ

Оператор `Compute Scalar` вычисляет выражение, выдавая в результате скалярное значение. К этому значению можно затем обращаться из любой точки запроса, как в данном примере. А оператор `Clustered Index Seek` выполняет поиск строк, используя соответствующие кластеризованные индексы таблиц.

ПРИМЕЧАНИЕ

Существует два типа обращения к индексам: *сканирование индексов* (index scan) и *поиск индекса* (index seek). При сканировании индекса обрабатываются все листья страницы дерева индексов, тогда как при поиске индекса возвращаются значения индексов (указатели) из одного или нескольких диапазонов индекса.

План выполнения XML

Термин *план выполнения XML* (XML execution plan) означает, что план выполнения запроса отображается в формате XML. (Дополнительную информацию по XML см. в *главе 26*.) Самое значительное достоинство использования планов выполнения XML состоит в том, что эти планы можно переносить с одной системы на другую, что позволяет использовать их в разных средах. (Сохранение планов выполнения в файл рассматривается несколько далее в этой главе.)

Для отображения планов выполнения в формате XML используются следующие две опции инструкции SET:

- SHOWPLAN_XML;
- STATISTICS XML.

Опция SHOWPLAN_XML задает возвращение информации в виде набора документов XML. Иными словами, если активировать эту опцию (т. е. присвоить ей значение ON), компонент Database Engine возвращает подробную информацию о выполнении инструкций в виде аккуратного документа XML, не исполняя эти инструкции в действительности. Для каждой инструкции выводится отдельный документ, содержащий текст инструкции, за которым следуют подробности его выполнения.

Использование опции SHOWPLAN_XML показано в примере 19.4.

Пример 19.4. Отображение плана выполнения посредством опции SHOWPLAN_XML

```
SET SHOWPLAN_XML ON;
GO
USE AdventureWorks2012;
SELECT * FROM HumanResources.Employee e JOIN Person.Address a
                         ON e.BusinessEntityID = a.AddressID
                                 AND e.BusinessEntityID = 10;
GO
SET SHOWPLAN_XML OFF;
```

Основное различие между опциями SHOWPLAN_XML и STATISTICS XML состоит в том, что вывод для второй создается в процессе выполнения инструкций. Поэтому результат опции STATISTICS XML включает результат опции SHOWPLAN_XML, а также дополнительную информацию времени выполнения.

Чтобы сохранить план выполнения XML в файл, в панели **Results** щелкните правой кнопкой метку **SQL Server XML Showplan** и в контекстном меню выберите пункт **Save Results As**. В открывшемся диалоговом окне **Save Grid Results** (или **Save Text Results**) в поле **Save As Type** укажите **All Files (*.*)**, а в поле **File Name** введите имя файла с расширением .sqlplan, а затем нажмите кнопку **Save**.

Чтобы открыть сохраненный план выполнения XML, выполните последовательность команд среды Management Studio **File | Open | File**. В открывшемся диалого-

вом окне **Open File** выберите в раскрывающемся списке поля **Files of Type** значение **Execution Plan Files (*.sqlplan)**, чтобы ограничить выбор файлами планов XML, затем выберите файл требуемого плана и нажмите кнопку **Open**.

ПРИМЕЧАНИЕ

Выбранный план можно также открыть двойным щелчком мыши. План открывается в среде Management Studio в графическом формате.

Другие опции инструкции *SET*

Инструкция `SET` имеет много других опций, применяемых с инструкциями блокировки, транзакции и даты и времени. В связи со статистическими данными компонент Database Engine предоставляет следующие три опции инструкции `SET`:

- `STATISTICS IO`;
- `STATISTICS TIME`;
- `STATISTICS PROFILE`.

Опция `STATISTICS IO` указывает системе отображать статистическую информацию об объеме дисковых операций, сгенерированную запросом, например, количество операций чтения и записи ввода/вывода, обработанных в запросе. Опция `STATISTICS TIME` задает отображение времени, требуемого для обработки, оптимизации и выполнения запроса.

При активировании опции `STATISTICS PROFILE` каждый выполненный запрос возвращает свой обычный результирующий набор, за которым следует дополнительный результирующий набор, отображающий профиль выполнения запроса.

Использование этой опции показано в примере 19.5.

Пример 19.5. Использование опции `STATISTICS PROFILE` в инструкции `SET`

```
SET STATISTICS PROFILE ON;
GO
USE AdventureWorks2012;
SELECT * FROM HumanResources.Employee e JOIN Person.Address a
                            ON e.BusinessEntityID = a.AddressID
                                    AND e.BusinessEntityID = 10;
GO
SET STATISTICS PROFILE OFF;
```

Результат выполнения запроса в примере 19.5, который здесь не показан по причине большого его размера, подобен результату выполнения запроса в примере 19.3 и содержит вывод инструкций `SET STATISTICS IO` и `SET SHOWPLAN_ALL`.

Среда Management Studio и графические планы выполнения

Графический план выполнения является самым лучшим способом отображения плана выполнения запроса для начинающих или если требуется просмотреть несколько разных планов в короткий промежуток времени. В этом формате для отображения операторов плана запроса используются пиктограммы.

В качестве примера отображения плана выполнения запроса в графическом формате на рис. 19.2 приводится графический план выполнения запроса из примера 19.3.

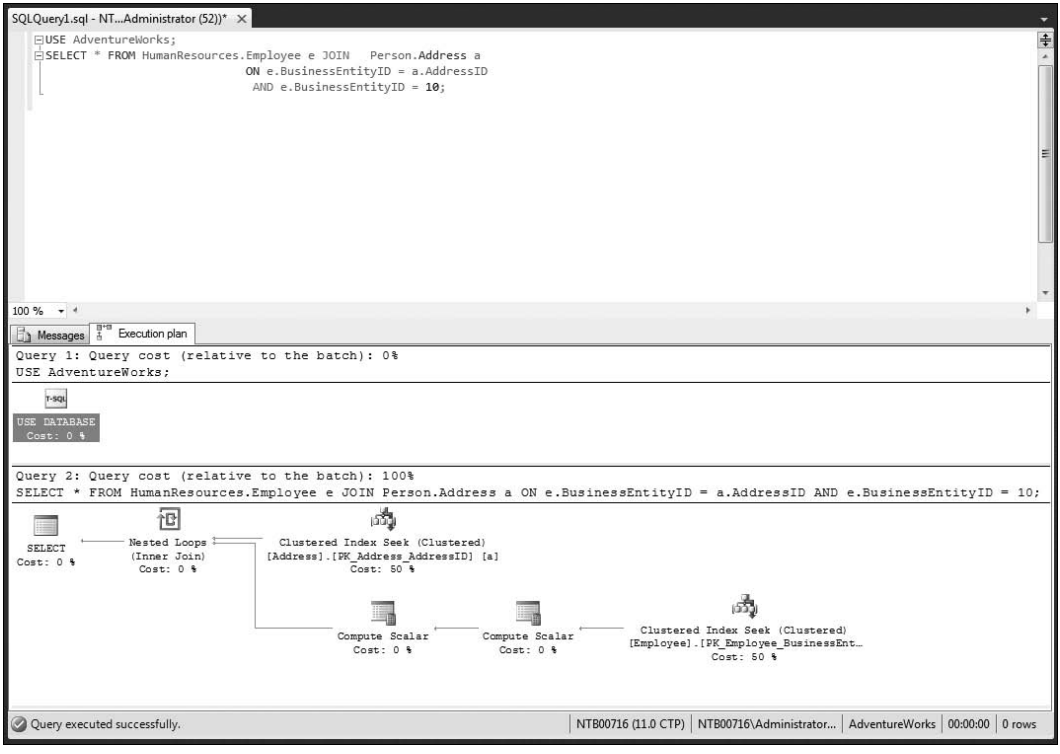


Рис. 19.2. Графический план выполнения запроса из примера 19.3

Для отображения плана выполнения в графическом формате вставьте запрос в редактор запросов Query Editor среды Management Studio, а затем на панели инструментов редактора запросов нажмите кнопку **Display Estimated Execution Plan** (показать предполагаемый план выполнения). Альтернативным способом является выполнение последовательности команд из меню среды Management Studio **Query | Display Estimated Execution Plan**.

На рис. 19.2 можно видеть, что каждый оператор плана выполнения представлен одной пиктограммой. При наведении указателя мыши на пиктограмму выводится всплывающее окно, содержащее подробную информацию о данном операторе, включая предполагаемые затраты на операции ввода/вывода и загрузку централь-

ного процессора, предполагаемое количество строк и их объем, а также затраты на выполнение оператора. Стрелки между пиктограммами представляют направление потока данных. (При наведении указателя мыши на стрелку отображается такая информация, как ожидаемое количество и размер строк.)

Для "чтения" графического плана выполнения запроса нужно следовать в направлении движения информации справа налево и сверху вниз. (Это аналогично направлению движения информации в текстовом плане выполнения.)

ПРИМЕЧАНИЕ

Толщина стрелок в графическом плане выполнения связана с количеством возвращаемых оператором строк. Чем толще стрелка, тем больше возвращается строк.

Как можно предположить по ее названию, нажатие кнопки **Display Estimated Execution Plan** отображает предполагаемый план выполнения запроса, не выполняя его в действительности. Другая кнопка, **Include Actual Execution Plan** (включить план фактического выполнения), выполняет запрос, а также отображает его план выполнения. Фактический план выполнения содержит дополнительную информацию относительно ожиданий этого плана выполнения, такую как фактическое количество обработанных строк и фактическое количество выполнений каждого оператора.

Пример планов выполнения

В этом разделе приведено несколько запросов, связанных с базой данных `AdventureWorks2012` совместно с их планами выполнения. Эти примеры демонстрируют уже рассмотренные ранее темы, позволяя увидеть работу оптимизатора запросов на практике.

В примере 19.6 в базе данных `sample` создается новая таблица `new_addresses`.

Пример 19.6. Создание таблицы `new_addresses` в базе данных `sample`

```
USE sample;
SELECT * into new_addresses
   FROM AdventureWorks2012.Person.Address;
GO
CREATE INDEX i_stateprov on new_addresses(StateProvinceID)
```

В примере 19.6 все содержимое таблицы `Address` из схемы `Person` базы данных `AdventureWorks2012` копируется в таблицу `new_addresses` базы данных `sample`. Такой подход вызван тем, что исходная таблица `Address` в базе данных `AdventureWorks2012` содержит несколько индексов, что затрудняет ее использование для демонстрации особых возможностей оптимизатора запросов. (Кроме этого, при создании новой таблицы в ней создается индекс для столбца `StateProvinceID`.)

В примере 19.7 показан запрос с высоким уровнем селективности, а также текстовый план, выбираемый оптимизатором в этом случае.

Пример 19.7. Запрос с высоким уровнем селективности

```
-- высокая селективность
SET SHOWPLAN_TEXT ON;
GO
USE sample;
SELECT * FROM new_addresses a
   WHERE a.StateProvinceID = 32;
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.7 получаем следующий текстовый план выполнения:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000]))
   |--Index Seek(OBJECT:([sample].[dbo].[new_addresses].
      [i_stateprov] AS [a]), SEEK:([a].[StateProvinceID]=(32))
      ORDERED FORWARD)
   |--RID Lookup(OBJECT:([sample].[dbo].[new_addresses] AS [a]),
         SEEK:([Bmk1000]=[Bmk1000]) LOOKUP ORDERED FORWARD)
```

ПРИМЕЧАНИЕ

План выполнения содержит оператор `Nested Loops`, хотя запрос в примере 19.7 обращается только к одной таблице. Начиная с SQL Server 2005, этот оператор всегда отображается в тех случаях, когда имеется два "соединенных" вместе оператора (как операторы `RID Lookup` и `Index Seek` в примере 19.7).

Фильтр в примере 19.7 выбирает из таблицы `new_addresses` только одну строку из общего количества 19 614 строк. Следственно, выражение в предложении `WHERE` имеет очень высокий уровень селективности (1/19614). В таком случае, как можно видеть в выводе плана выполнения запроса примера 19.7, оптимизатор использует существующий индекс для столбца `StateProvinceID`.

В примере 19.8 показан тот же самый запрос, как и в примере 19.7, но с другим фильтром.

Пример 19.8. Запрос с низкой селективностью

```
-- низкая селективность
SET SHOWPLAN_TEXT ON;
GO
USE sample;
SELECT * FROM new_addresses a
   WHERE a.StateProvinceID = 9;
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.8 получаем следующий текстовый план выполнения:

```
|--Table Scan(OBJECT:([sample].[dbo].[new_addresses] AS [a]),
   WHERE:([sample].[dbo].[new_addresses].[StateProvinceID] as [a].
  [StateProvinceID]=(9)))
```

Хотя запрос в примере 19.8 отличается от запроса примера 19.7 всего лишь значением с правой части условия в предложении `WHERE`, выбранный для него оптимизатором план выполнения отличается существенно. В данном случае существующий индекс не используется по причине низкого уровня селективности фильтра. (Соотношение количества строк, удовлетворяющих условию, к общему количеству строк таблицы составляет 4564/19612 = 0,23, или иначе 23%.)

В примере 19.9 показано использование кластеризованного индекса.

Пример 19.9. Использование кластеризованного индекса

```
SET SHOWPLAN_TEXT ON;
GO
USE AdventureWorks;
SELECT * FROM HumanResources.Employee
   WHERE HumanResources.Employee.BusinessEntityID = 10;
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.9 получаем следующий текстовый план выполнения:

```
|--Compute Scalar(DEFINE:([AdventureWorks2012].[HumanResources].
   [Employee].[OrganizationLevel]=[AdventureWorks2012].
   [HumanResources].[Employee].[OrganizationLevel]))
   |--Compute Scalar(DEFINE:([AdventureWorks2012].[HumanResources].
     [Employee].[OrganizationLevel]=[AdventureWorks2012].
     [HumanResources].[Employee].[OrganizationNode].GetLevel()))
      |--Clustered Index Seek(OBJECT:([AdventureWorks2012].
         [HumanResources].[Employee].[PK_Employee_
         BusinessEntityID]),SEEK:([AdventureWorks2012].
         [HumanResources].[Employee].[BusinessEntityID]=
         CONVERT_IMPLICIT(int,[@1],0))ORDERED FORWARD)
```

В запросе примера 19.9 используется кластеризованный индекс `PK_Employee_BusinessEntityID`. Этот кластеризованный индекс неявно создается системой, поскольку столбец `BusinessEntityID` является первичным ключом таблицы `Employee`. (Описание оператора `Compute Scalar` см. в примере 19.3.)

В примере 19.10 показано использование метода вложенного цикла.

Пример 19.10. Соединение таблиц методом вложенного цикла

```
SET SHOWPLAN_TEXT ON;
GO
USE AdventureWorks;
SELECT * FROM HumanResources.Employee e JOIN
          Person.Address a
               ON e.BusinessEntityID = a.AddressID
               AND e.BusinessEntityID = 10;
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.10 получаем следующий текстовый план выполнения:

```
   |--Nested Loops(Inner Join)
      |--Clustered Index Seek(OBJECT:([AdventureWorks].[Person].
         [Address].[PK_Address_AddressID] AS [a]),
         SEEK:([a].[AddressID]=(10)) ORDERED FORWARD)
      |--Compute Scalar(DEFINE:([e].[OrganizationLevel]=
         [AdventureWorks].[HumanResources].[Employee].
         [OrganizationLevel] as [e].[OrganizationLevel]))
         |--Compute Scalar(DEFINE:([e].[OrganizationLevel]=
           [AdventureWorks].[HumanResources].[Employee].
           [OrganizationNode] as [e].[OrganizationNode].GetLevel()))
            |--Clustered Index Seek(OBJECT:([AdventureWorks].
              [HumanResources].[Employee].
              [PK_Employee_BusinessEntityID] AS [e]), SEEK:
              ([e].[BusinessEntityID]=(10))ORDERED FORWARD)
```

В запросе примера 19.10 используется метод вложенного цикла, хотя столбцы соединения таблиц одновременно являются их первичными ключами, вследствие чего можно было бы ожидать использования метода соединения слиянием. Причина, по которой оптимизатор запросов решает применить вложенный цикл, состоит в наличии дополнительного фильтра (`e.EmployeeID = 10`), который сводит результирующий набор запроса к одной строке.

В примере 19.11 показано использование метода соединения хешированием.

Пример 19.11. Запрос с использованием метода соединения хешированием

```
SET SHOWPLAN_TEXT ON;
GO
USE AdventureWorks;
SELECT * FROM Person.Address a JOIN Person.StateProvince s
   ON a.StateProvinceID = s.StateProvinceID;
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.11 получаем следующий текстовый план выполнения:

```
|--Hash Match(Inner Join, HASH:([s].[StateProvinceID])=
   ([a]. [StateProvinceID]))
   |--Clustered Index Scan(OBJECT:([AdventureWorks].[Person].
      [StateProvince].[PK_StateProvince_StateProvinceID] AS [s]))
   |--Clustered Index Scan(OBJECT:([AdventureWorks].[Person].
      [Address].[PK_Address_AddressID] AS [a]))
```

Хотя оба столбца соединения в предложении `ON` являются первичными ключами соответствующих таблиц (`Address` и `StateProvince`), оптимизатор запросов не выбирает метод соединения слиянием. Причина этого выбора оптимизатора состоит в том, что в результирующий набор входят *все* 19 614 строк таблицы `Address`. В таком случае использование метода соединения хешированием более выгодно, чем другие два метода обработки.

Динамические административные представления и оптимизатор запросов

Существует большое число динамических административных представлений (и функций), непосредственно связанных с оптимизацией запросов. В этом разделе мы рассмотрим следующие *динамические административные представления — DMV* (Dynamic Management View):

- `sys.dm_exec_query_optimizer_info`;
- `sys.dm_exec_query_plan`;
- `sys.dm_exec_query_stats`;
- `sys.dm_exec_sql_text`;
- `sys.dm_exec_text_query_plan`;
- `sys.dm_exec_procedure_stats`;
- `sys.dm_exec_cached_plans`.

Представление *sys.dm_exec_query_optimizer_info*

Представление `sys.dm_exec_query_optimizer_info` является, возможно, самым важным динамическим административным представлением (DMV) в плане работы оптимизатора запросов, потому что оно возвращает подробную статистику о работе оптимизатора. Это представление можно использовать при настройке рабочей нагрузки, чтобы определить проблемы оптимизации запросов и возможные ее улучшения.

Представление `sys.dm_exec_query_optimizer_info` содержит три столбца: `counter`, `occurrence` и `value`. В столбце `counter` содержится имя события оптимизатора, а в столбце `occurrence` — отображается общее количество проявлений этих событий. Значение столбца `value` содержит дополнительную информацию о событиях. (Не все события возвращают значение `value`.)

С помощью этого представления можно, например, отобразить общее количество оптимизаций, прошедшее время, а также окончательные затраты, чтобы сравнить оптимизации запроса для текущей нагрузки и любые изменения, наблюдаемые в процессе настройки.

Использование представления `sys.dm_exec_query_optimizer_info` показано в примере 19.12.

Пример 19.12. Применение представления `sys.dm_exec_query_optimizer_info`

```
USE sample;
SELECT counter, occurrence, value
      FROM sys.dm_exec_query_optimizer_info
      WHERE value IS NOT NULL
      AND counter LIKE 'search 1%';
```

Этот запрос возвращает следующий результат:

counter	Occurrence	value
search 1	117	1
search 1 time	95	0.0120736842105263
search 1 tasks	117	513.982905982906

В столбце `counter` отображаются этапы процесса оптимизации. Таким образом, в примере 19.12 исследуются, сколько раз выполняется оптимизация этапа 1. (Существует три этапа оптимизации, этап 0, этап 1 и этап 2, которые задаются значениями `search 0`, `search 1` и `search 2` соответственно.)

ПРИМЕЧАНИЕ

По причине своей сложности процесс оптимизации разбивается на три этапа. На первом этапе (этап 0) рассматриваются только планы непараллельного выполнения. В случае неоптимальных затрат этапа 0 исполняется этап 1, в котором рассматриваются как непараллельные, так и параллельные планы. На этапе 2 рассматриваются только планы параллельного выполнения.

Представление *sys.dm_exec_query_plan*

Как уже упоминалось ранее, планы выполнения для пакетов и инструкций языка Transact-SQL сохраняются в кэше, что позволяет оптимизатору использовать их в любое время. Проверить содержимое кэша можно с помощью нескольких динамических административных представлений (DMV). Одним из таких представлений является представление `sys.dm_exec_query_plan`, которое возвращает все планы выполнения, находящиеся в кэше системы. (Планы выполнения отображаются в формате XML.)

ПРИМЕЧАНИЕ

Электронная документация содержит несколько полезных примеров использования этого динамического административного представления.

Каждый хранящийся в кэше план выполнения имеет однозначный идентификатор, который называется *дескриптором плана* (plan handle). Чтобы извлечь из кэша план выполнения для определенной инструкции или пакета инструкций, представлению `sys.dm_exec_query_plan` требуется его дескриптор. Этот дескриптор можно узнать с помощью представления `sys.dm_exec_query_stats`, которое рассматривается в следующем разделе.

Представление *sys.dm_exec_query_stats*

Представление `sys.dm_exec_query_stats` возвращает суммарную статистику производительности для кэшированных планов выполнения запросов. Это представление содержит одну строку для каждой инструкции запроса в кэшированном плане, а время жизни строк связано с самим планом.

Использование инструкции `sys.dm_exec_query_stats` показано в примере 19.13.

Пример 19.13. Применение представления `sys.dm_exec_query_stats`

```
SELECT ecp.objtype AS Object_Type,
(SELECT t. text FROM sys.dm_exec_sql_text(qs.sql_handle) AS t) AS
Adhoc_Batch, qs. execution_count AS Counts,
qs. total_worker_time AS Total_Worker_Time,
(qs. total_worker_time / qs. execution_count) AS Avg_Worker_Time,
(qs. total_physical_reads / qs. execution_count) AS Avg_Physical_Reads,
(qs. total_logical_writes / qs. execution_count) AS Avg_Logical_Writes,
(qs. total_logical_reads / qs. execution_count) AS Avg_Logical_Reads,
qs. total_clr_time AS Total_CLR_Time,
(qs. total_clr_time / qs. execution_count) AS Avg_CLR_Time,
qs. total_elapsed_time AS Total_Elapsed_Time,
(qs. total_elapsed_time / qs. execution_count) AS Avg_Elapsed_Time,
qs. last_execution_time AS Last_Exec_Time,
qs. creation_time AS Creation_Time
     FROM sys.dm_exec_query_stats AS qs
   JOIN sys.dm_exec_cached_plans ecp ON qs.plan_handle = ecp.plan_handle
   ORDER BY Counts DESC;
```

Выполнение запроса в примере 19.13 соединяет представления `sys.dm_exec_query_stats` и `sys.dm_exec_cached_plans` и возвращает все кэшированные планы выполнения, упорядоченные по количеству их выполнений.

Представления *sys.dm_exec_sql_text* и *sys.dm_exec_text_query_plan*

Предыдущее представление, `sys.dm_exec_query_stats`, можно использовать с несколькими другими динамическими административными представлениями (DMV) для отображения разных свойств запросов. Иными словами, каждое динамическое административное представление, для которого требуется дескриптор плана, чтобы идентифицировать запрос, "соединяется" с представлением `sys.dm_exec_query_stats`, чтобы предоставить требуемую информацию. Одним из таких представлений является представление `sys.dm_exec_sql_text`. Это представление возвращает текст пакета SQL, который определяется указанным дескриптором. Электронная документация содержит очень полезный пример, который "соединяет" представления `sys.dm_exec_sql_text` и `sys.dm_exec_query_stats`, чтобы возвратить текст запросов SQL, выполняемых в пакетах, и предоставляет статистическую информацию о них.

В отличие от представления `sys.dm_exec_sql_text`, представление `sys.dm_exec_query_plan` возвращает план выполнения пакета в формате XML. Подобно ранее рассмотренным представлениям, это представление указывается дескриптором плана. (Указываемый дескриптором план может или находиться в кэше, или может выполняться в настоящее время.)

Представление *sys.dm_exec_procedure_stats*

Это динамическое административное представление (DMV) похоже на представление `sys.dm_exec_query_stats`. Оно возвращает суммарную статистику производительности для кэшированных хранимых процедур. Это представление содержит одну строку для каждой хранимой процедуры, а время жизни строки определяется временем нахождения процедуры в кэше. При удалении хранимой процедуры из кэша, соответствующее представление удаляется из этого представления.

Представление *sys.dm_exec_cached_plans*

Это представление возвращает строку для каждого кэшированного плана запроса. Его можно использовать, чтобы найти кэшированные планы запросов, кэшированные тексты запросов, объем памяти, занимаемый кэшированными планами, а также количество повторных выполнений кэшированных планов.

Наиболее важными столбцами этого представления являются столбцы `cacheobjtype` и `usecount`. В первом столбце указывается тип кэшированного объекта, а во втором — количество использований данного кэшированного объекта с момента его создания.

Подсказки оптимизации

В большинстве случаев оптимизатор запросов выбирает самый быстрый план выполнения. Но в некоторых особых ситуациях оптимизатор, в силу определенных обстоятельств, не может найти оптимального решения. В таких случаях следует

использовать подсказки оптимизации, чтобы вынудить оптимизатор использовать определенный план выполнения с лучшей производительностью.

Подсказки оптимизации являются факультативной частью инструкции SELECT, которые указывают оптимизатору запросов, что нужно выполнять данную инструкцию определенным образом. Иными словами, посредством подсказок оптимизатору запросов запрещается искать способ выполнения данного запроса, поскольку ему дается указание, каким точно способом его выполнять.

Зачем надо использовать подсказки оптимизации

Подсказки оптимизации следует использовать только временно и только для тестирования. Иными словами, следует избегать делать их постоянной составляющей запроса. Этому есть две причины. Во-первых, если заставить оптимизатор использовать определенный индекс, а затем определить индекс, который улучшает производительность запроса, то этот запрос и его приложение не смогут извлечь пользу из нового индекса. Во-вторых, компания Microsoft постоянно прилагает усилия над улучшением оптимизатора запросов. Если запрос привязать к конкретному плану выполнения, то он не сможет воспользоваться новыми и улучшенными возможностями последующих версий SQL Server.

Существует две причины, по которым оптимизатор иногда выбирает не самый быстрый план выполнения:

- оптимизатор запросов не является безупречным;
- система не предоставляет оптимизатору требуемую информацию.

ПРИМЕЧАНИЕ

Подсказки оптимизации полезны только в том случае, если оптимизатор выбирает менее оптимальный план выполнения. Если система не предоставляет оптимизатору требуемую информацию, следует создать или модифицировать существующие статистические данные, используя опции базы данных AUTO_CREATE_STATISTICS и AUTO_ UPDATE_STATISTICS соответственно.

Типы подсказок оптимизации

Компонент Database Engine поддерживает следующие типы подсказок оптимизации:

- табличные подсказки;
- подсказки соединения;
- подсказки запросов;
- структуры планов.

Эти типы подсказок по оптимизации рассматриваются в последующих разделах.

ПРИМЕЧАНИЕ

В приводимых далее примерах демонстрируется использование подсказок оптимизации, но они не содержат никаких рекомендаций по применению их в каком-либо конкретном запросе. (В большинстве случаев, в этих показанных примерах использование подсказок приводит к результату, противоположному ожидаемому.)

Табличные подсказки

Табличные подсказки можно применять к отдельной таблице. Поддерживаются следующие подсказки:

- INDEX;
- NOEXPAND;
- FORCESEEK.

Подсказка INDEX используется для указания одного или более индексов, которые затем применяются в запросе. Эта подсказка указывается в предложении FROM запроса. Если по какой-либо причине оптимизатор решает последовательно сканировать таблицу при выполнении данного запроса, с помощью этой подсказки можно заставить запрос обращаться к индексу. Кроме этого, подсказку INDEX можно использовать, чтобы не допустить использование оптимизатором определенного индекса.

Использование подсказки INDEX показано в примерах 19.14 и 19.15.

Пример 19.14. Принуждение оптимизатора к использованию индекса

```
SET SHOWPLAN_TEXT ON;
GO
USE sample;
SELECT * FROM new_addresses a WITH (INDEX(i_stateprov))
   WHERE a.StateProvinceID = 9;
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.14 получаем следующий текстовый план выполнения:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000],
  [Expr1004]) WITH UNORDERED PREFETCH)
   |--Index Seek(OBJECT:([sample].[dbo].[new_addresses].
      [i_stateprov] AS [a]), SEEK:([a].[StateProvinceID]=
       (9))ORDERED FORWARD)
   |--RID Lookup(OBJECT:([sample].[dbo].[new_addresses]
      AS [a]), SEEK:([Bmk1000]=[Bmk1000]) LOOKUP ORDERED FORWARD)
```

Пример 19.14 идентичен примеру 19.8, но содержит подсказку INDEX, которая заставляет оптимизатор запросов использовать индекс i_stateprov. Без этой подсказ-

ки оптимизатор принимает решение сканировать таблицу, как это можно видеть в результате выполнения примера 19.8.

Другая форма этой подсказки, `INDEX(0)`, заставляет оптимизатор не использовать никакие существующие некластеризованные индексы. Использование этой формы подсказки показано в примере 19.15.

Пример 19.15. Принуждение оптимизатора не использовать некластеризованные индексы

```
SET SHOWPLAN_TEXT ON;
GO
USE AdventureWorks;
SELECT * FROM Person.Address a
   WITH(INDEX(0))
   WHERE a.StateProvinceID = 32;
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.15 получаем следующий текстовый план выполнения:

```
|--Clustered Index Scan (OBJECT:([AdventureWorks].[Person].[Address].
   [PK_Address_AddressID] AS [a]), WHERE:([AdventureWorks].[Person].
    [Address].[StateProvinceID] as [a].[StateProvinceID]=(32)))
```

ПРИМЕЧАНИЕ

Если существует кластеризованный индекс, то подсказка `INDEX(0)` принуждает к его сканированию, а подсказка `INDEX(1)` требует сканирования кластеризованного индекса или поиска по индексу. Если же кластеризованный индекс отсутствует, то подсказка `INDEX(0)` требует сканирования таблицы, а подсказка `INDEX(1)` рассматривается как ошибка.

План выполнения запроса в примере 19.15 показывает, что вследствие наличия подсказки `INDEX(0)` оптимизатор выполняет сканирование кластеризованного индекса. Без этой подсказки сам оптимизатор запросов выбрал бы сканирование некластеризованного индекса. (Это можно проверить, удалив подсказку из запроса.)

Подсказка `NOEXPAND` указывает, что не будут разворачиваться никакие индексированные представления для доступа к лежащим в его основе таблицам в то время, когда оптимизатор запросов обрабатывает запрос. Оптимизатор запросов рассматривает представление как таблицу с кластеризованным индексом. (Индексированные представления рассматриваются в *главе 11*.)

Табличная подсказка `FORSEEK` заставляет оптимизатор использовать в качестве пути доступа к данным в указанных в запросе таблицах или представлениях только операцию поиска по индексу. Эту табличную подсказку можно использовать, чтобы отменить выбранный оптимизатором план, чтобы предотвратить возникновение

проблем, вызванных неэффективным планом запроса. Например, если план содержит операторы сканирования таблицы или индекса, а в процессе выполнения запроса соответствующие таблицы требуют большого количества операций чтения, то принудительное задание операции поиска по индексу может повысить производительность запроса. Это особенно справедливо в тех случаях, когда ошибочные определение количества элементов или оценка затрат приводят к тому, что оптимизатор при компилировании плана отдает предпочтение операциям сканирования.

ПРИМЕЧАНИЕ

Подсказку FORCESEEK можно применять как с кластеризованными, так и с некластеризованными индексами.

Подсказки соединения

Подсказки соединения дают указания оптимизатору запросов касательно метода реализации соединений в запросе. Они принуждают оптимизатор или соединять таблицы в том порядке, в котором они указаны в предложении FROM инструкции SELECT, или использовать методы обработки соединения, явно указанные в инструкции. Компонент Database Engine поддерживает следующие подсказки соединения:

- FORCE ORDER;
- LOOP;
- HASH;
- MERGE.

Подсказка FORCE ORDER заставляет оптимизатор соединять таблицы в том порядке, в котором они указаны в запросе. Использование этой подсказки показано в примере 19.16.

Пример 19.16. Использование подсказки FORCE ORDER

```
SET SHOWPLAN_TEXT ON;
GO
USE AdventureWorks2012;
SELECT e.BusinessEntityID, e.LoginID, d.DepartmentID
   FROM HumanResources.Employee e, HumanResources.Department d,
      HumanResources.EmployeeDepartmentHistory h
    WHERE d.DepartmentID = h.DepartmentID
   AND h.BusinessEntityID = e.BusinessEntityID
   AND h.EndDate IS NOT NULL
   OPTION(FORCE ORDER);
GO
SET SHOWPLAN_TEXT OFF;
```

В результате выполнения примера 19.16 получаем следующий текстовый план выполнения:

```
|--Merge Join(Inner Join, MERGE:([d].[DepartmentID],[e].
   [BusinessEntityID])=([h].[DepartmentID], [h].[BusinessEntityID]),
    RESIDUAL:([AdventureWorks].[HumanResources].[Department].
   [DepartmentID] as [d].[DepartmentID]=[AdventureWorks2012].
   [HumanResources].[EmployeeDepartmentHistory].[DepartmentID]
   as [h].[DepartmentID] AND [AdventureWorks2012].[HumanResources].
   [EmployeeDepartmentHistory].[BusinessEntityID] as [h].
   [BusinessEntityID]=[AdventureWorks2012].[HumanResources].
   [Employee].[BusinessEntityID] as [e].[BusinessEntityID]))
    |--Sort(ORDER BY:([d].[DepartmentID] ASC, [e].
       [BusinessEntityID] ASC))
    |   |--Nested Loops(Inner Join)
    |       |--Index Scan(OBJECT:([AdventureWorks2012].
               [HumanResources].[Employee].[AK_Employee_LoginID]
                AS [e]))
    |       |--Clustered Index Scan(OBJECT:([AdventureWorks].
               [HumanResources].[Department].[PK_Department_
                DepartmentID] AS [d]))
    |--Sort(ORDER BY:([h].[DepartmentID] ASC, [h].[BusinessEntityID]
       ASC))
        |--Clustered Index Scan(OBJECT:([AdventureWorks2012].
           [HumanResources].[EmployeeDepartmentHistory].
           [PK_EmployeeDepartmentHistory_BusinessEntityID_
           StartDate_DepartmentID] AS [h]), WHERE:
           ([AdventureWorks2012].[HumanResources].
           [EmployeeDepartmentHistory].[EndDate]
           as [h].[EndDate] IS NOT NULL))
```

Как можно видеть по результату выполнения примера 19.16, оптимизатор выполняет операцию соединения таблиц в том порядке, в каком они указываются в запросе. Это означает, что сначала обрабатывается таблица `EmployeeDepartmentHistory`, затем таблица `Department` и, наконец, таблица `Employee`. Если же выполнить этот запрос без подсказки `FORCE ORDER`, то оптимизатор запросов будет обрабатывать эти таблицы в обратном порядке: сначала таблицу `Employee`, затем таблицу `Department`, а таблицу `EmployeeDepartmentHistory` — самой последней.

ПРИМЕЧАНИЕ

Не стоит думать, что этот новый план выполнения является более эффективным, чем выбранный оптимизатором.

Подсказки запроса `LOOP`, `MERGE` и `HASH` заставляют оптимизатор использовать метод вложенного цикла, соединение слиянием и соединение хешированием соответственно. Эти три подсказки соединения можно использовать только в том случае, когда операция соединения отвечает требованиям стандарта SQL, т. е. когда соединение явно указывается при помощи ключевого слова `JOIN` в предложении `FROM` инструкции `SELECT`.

В примере 19.17 показан запрос, в котором используется метод соединения слиянием, поскольку в инструкции SELECT явно указана подсказка с этой целью. (Подсказки HASH и LOOP используются таким же образом.)

Пример 19.17. Использование подсказки слияния MERGE

```
SET SHOWPLAN_TEXT ON;
GO
USE AdventureWorks2012;
SELECT * FROM Person.Address a JOIN Person.StateProvince s
   ON a.StateProvinceID = s.StateProvinceID
     OPTION (MERGE JOIN);
GO
SET SHOWPLAN_TEXT OFF;
```

Реализация примера 19.17 выдает следующий текстовый план выполнения:

```
|--Merge Join(Inner Join, MERGE:([s].[StateProvinceID])=([a].
  [StateProvinceID]), RESIDUAL:([AdventureWorks2012].
  [Person].[StateProvince].[StateProvinceID] as [s].
  [StateProvinceID]=[AdventureWorks2012].[Person].[Address].
  [StateProvinceID] as [a].[StateProvinceID]))
    |--Clustered Index Scan(OBJECT:([AdventureWorks].[Person].
       [StateProvince].[PK_StateProvince_StateProvinceID]
        AS [s]), ORDERED FORWARD)
    |--Nested Loops(Inner Join, OUTER REFERENCES:([a].[AddressID],
       [Expr1004]) WITH ORDERED PREFETCH)
        |--Index Scan(OBJECT:([AdventureWorks2012].[Person].
           [Address].[IX_Address_StateProvinceID] AS [a]),
            ORDERED FORWARD)
        |--Clustered Index Seek(OBJECT:([AdventureWorks2012].[Person].
           [Address].[PK_Address_AddressID] AS [a]), SEEK:([a].
           [AddressID]=[AdventureWorks2012].[Person].[Address].
           [AddressID] as [a].[AddressID]) LOOKUP ORDERED FORWARD)
```

Как можно видеть в выводе запроса примера 19.17, оптимизатор запросов должен использовать технику соединения слиянием. (Если удалить соответствующую подсказку, то оптимизатор запросов выберет технику слияния хешированием.)

Требуемая подсказка соединения может быть записана или в предложении FROM запроса или при использовании предложения OPTION в конце этого запроса. При совместном применении нескольких разных подсказок рекомендуется использовать предложение OPTION. Пример 19.18 идентичен примеру 19.16, но в нем подсказка соединения указывается в предложении FROM. Обратите внимание, что в данном случае требуется использовать ключевое слово INNER.

Пример 19.18. Использование подсказки слияния MERGE в предложении FROM

```
USE AdventureWorks2012;
SELECT * FROM Person.Address a INNER MERGE JOIN Person.StateProvince s
   ON a.StateProvinceID = s.StateProvinceID;
```

Подсказки запросов

Существует несколько подсказок запросов, которые применяются с разными целями. В этом разделе рассматриваются следующие подсказки запросов:

- FAST *n*;
- OPTIMIZE FOR;
- OPTIMIZE FOR UNKNOWN;
- USE PLAN.

Подсказка FAST *n* указывает, что запрос оптимизируется для быстрого возвращения первых *n* строк таблицы. После выдачи первых *n* строк запрос продолжает выполняться и возвращает полный результирующий набор.

ПРИМЕЧАНИЕ

Эта подсказка может быть полезной в случае очень сложного запроса, возвращающего большое количество строк и требующего много времени на выполнение. В обычном случае запросы выполняются полностью, после чего система выводит их результирующий набор. Но эта подсказка запроса заставляет систему отображать первые *n* строк сразу же после их обработки запросом.

Подсказка OPTIMIZE FOR заставляет оптимизатор запросов использовать определенное значение для локальной переменной при компилировании и оптимизации запроса. Это значение используется только в процессе оптимизации запроса, но не при его выполнении. Эту подсказку запроса можно использовать при создании структур планов, которые рассмотрены в следующем разделе.

Использование подсказки OPTIMIZE FOR показано в примере 19.19.

Пример 19.19. Использование подсказки запроса OPTIMIZE FOR

```
DECLARE @city_name nvarchar(30)
SET @city_name = 'Newark'
SELECT * FROM Person.Address
   WHERE City = @city_name
      OPTION (OPTIMIZE FOR (@city_name = 'Seattle'));
```

Хотя переменная @city_name имеет значение Newark, подсказка OPTIMIZE FOR заставляет оптимизатор при оптимизации запроса для этой переменной использовать значение Seattle.

Подсказка OPTIMIZE FOR UNKNOWN указывает оптимизатору использовать статистические данные вместо исходных значений для всех локальных переменных при компилировании и оптимизации запроса, включая параметры, созданные посредством принудительной параметризации. Выражение *принудительная параметризация* (forced parameterization) означает, что любое литеральное значение в инструкции SELECT, INSERT, UPDATE или DELETE, предоставленное в любой форме, при компилиро-

вании запроса преобразовывается в параметр. Таким образом все запросы с разным значением параметра будут вынуждены повторно использовать кэшированный план запроса, вместо того, чтобы перекомпилировать запрос при каждом новом значении параметра.

Подсказка `USE PLAN` принимает в качестве параметра план выполнения, хранящийся в виде XML-документа, и указывает компоненту Database Engine использовать данный план для выполнения запроса. Как сохранять план выполнения в виде документа XML, рассмотрено при описании параметра `SHOWPLAN_XML` инструкции `SET` в *разд. "План выполнения XML" ранее в этой главе*.

Структуры планов

Как уже упоминалось в предыдущем разделе, подсказки явно указываются в инструкции `SELECT` для оказания влияния на работу оптимизатора запросов. Но иногда изменить текст инструкции `SELECT` прямым образом не представляется возможным или не является желательным. В таком случае все равно можно повлиять на выполнение запроса, используя *структуры планов* (plan guides). Иными словами, структуры планов позволяют использовать определенную подсказку оптимизации, не изменяя синтаксис инструкции `SELECT`.

ПРИМЕЧАНИЕ

Основная цель использования структур планов состоит в том, чтобы избежать жесткого программирования подсказок в тех случаях, когда это не рекомендуется или нежелательно (например, для кода сторонних разработчиков).

Для создания структур планов применяется системная процедура `sp_create_plan_guide`. Эта процедура создает структуру плана для привязки подсказок запросов или фактических планов запросов с запросами в базе данных. Другая системная процедура, `sp_control_plan_guide`, включает, отключает или удаляет существующие структуры планов.

ПРИМЕЧАНИЕ

Инструкции языка DDL для создания и удаления структур планов не существует. Будем надеяться, что в будущих версиях SQL Server такие инструкции будут поддерживаться.

Компонент Database Engine поддерживает следующие три типа структур планов:

- **SQL** — сопоставляет запросы, выполняющиеся в контексте изолированных инструкций Transact-SQL и пакетов, не входящих ни в один объект базы данных;
- **OBJECT** — сопоставляет запросы, которые выполняются в контексте процедур и триггеров DML;
- **TEMPLATE** — сопоставляет одиночные запросы, которые параметризируются к указанной форме.

В примере 19.20 показано создание подсказки оптимизации из примера 19.17, не прибегая к модификации соответствующей инструкции SELECT.

Пример 19.20. Создание структуры плана

```
sp_create_plan_guide  @name = N'Example_19_15',
@stmt = N'SELECT * FROM Person.Address a JOIN Person.StateProvince s
   ON a.StateProvinceID = s.StateProvinceID',
@type = N'SQL',
@module_or_batch = NULL,
@params = NULL,
@hints = N'OPTION (HASH JOIN)'
```

Как можно видеть в примере 19.20, системная процедура sp_create_plan_guide принимает несколько параметров. В параметре @name указывается имя создаваемой структуры плана. Параметр @stmt содержит инструкция Transact-SQL, а параметр @type указывает тип структуры плана (SQL, OBJECT или TEMPLATE). Подсказка по оптимизации указывается в параметре @hints. Структуры планов можно также создавать с помощью среды Management Studio.

Для редактирования структуры планов используется представление каталога sys.plan_guides. Это представление содержит строку для каждой структуры плана в текущей базе данных. Наиболее важными столбцами этого представления являются столбцы plan_guid_id, name и query_text. Столбец plan_guide_id содержит однозначный идентификатор структуры плана, а столбец name определяет ее имя. В столбце query_text указывается текст запроса, для которого предназначается данная структура плана.

Резюме

Оптимизатор запросов является составной частью компонента Database Engine, которая определяет наиболее оптимальный метод выполнения запроса. Оптимизатор создает несколько планов выполнения для данного запроса и выбирает из них наименее затратный.

Процесс оптимизации запроса можно разбить на следующие этапы: анализ запроса, выбор индексов и выбор порядка выполнения соединения. В процессе анализа запроса оптимизатор исследует его на наличие аргументов поиска, использование оператора OR и наличие критериев соединения, в приведенном порядке. Идентификация аргументов поиска позволяет оптимизатору решить, использовать ли существующие индексы.

Порядок указания в предложении FROM инструкции SELECT двух или больше соединяемых таблиц не влияет на принимаемое оптимизатором решение относительно порядка их обработки. Компонент Database Engine поддерживает три различные техники обработки соединения, которые могут быть использованы оптимизатором.

То, какую технику выберет оптимизатор, зависит от имеющихся статистических данных для соединяемых таблиц.

Компонент Database Engine поддерживает разнообразные инструментальные средства для редактирования планов выполнения. Наиболее важными из этих средств являются инструменты текстового и графического представления планов, а также динамические административные представления (DMV).

Оказать воздействие на работу оптимизатора можно, используя подсказки оптимизации. Компонент Database Engine поддерживает разнообразные подсказки оптимизации, которые можно сгруппировать следующим образом: табличные подсказки, подсказки соединения и подсказки запросов.

Структуры планов позволяют влиять на процесс оптимизации запросов, не изменяя при этом конкретную инструкцию `SELECT`, как это требуется в случае использования подсказок запросов.

В следующей главе мы рассмотрим настройку производительности.

Глава 20

Настройка производительности

- **Факторы, влияющие на производительность**
- **Мониторинг производительности**
- **Выбор правильного средства мониторинга**
- **Другие средства SQL Server для настройки производительности**

Улучшение производительности системы баз данных требует принятия многих решений, таких как, например, где хранить данные и как осуществлять доступ к данным. Эта задача отличается от других задач администрирования, поскольку она состоит из нескольких разных этапов, затрагивающих все аспекты программного и аппаратного обеспечения. Если система баз данных не работает в оптимальном режиме, то системный администратор должен выполнить проверку многих аспектов всей компьютерной системы и, возможно, выполнить настройку программного обеспечения (операционной системы, системы управления базами данных, приложений баз данных) и аппаратного обеспечения.

Производительность компонента Database Engine, как и любой другой реляционной системы управления базами данных, измеряется двумя критериями:

- временем реакции;
- пропускной способностью.

Время реакции является мерилом эффективности отдельной транзакции или программы. Время реакции определяется как длительность периода времени от момента введения пользователем команды или инструкции до момента, когда система завершает выполнение этой команды или инструкции. Чтобы достичь оптимального времени реакции всей системы, время реакции почти всех существующих команд и инструкций (от 90 до 95% от их общего количества) не должно выходить за установленные пределы.

Пропускная способность измеряет общую эффективность системы в исчислении количества транзакций, которые компонент Database Engine может обработать в

течение определенного периода времени. Обычно пропускная способность измеряется в транзакциях в секунду. Таким образом, между временем реакции системы и ее пропускной способностью существует прямая связь: при понижении времени реакции системы (например, вследствие одновременного использования системы большим количеством пользователей) также понижается ее пропускная способность.

В этой главе обсуждаются вопросы производительности и инструментальные средства для настройки системы баз данных, релевантные для повседневного администрирования системы. В первой части главы обсуждаются факторы, влияющие на производительность. Затем даются некоторые рекомендации по выбору правильных инструментов для администрирования производительности. В конце главы рассматриваются средства для мониторинга работы системы баз данных.

Факторы, влияющие на производительность

Факторы, влияющие на производительность системы, можно разбить на три общие категории, такие как:

- прикладные программы баз данных;
- система баз данных;
- системные ресурсы.

Эти факторы в свою очередь могут подвергаться влиянию многих других факторов, которые рассматриваются в последующих разделах.

Приложения базы данных и производительность

Следующие факторы могут влиять на производительность приложений базы данных:

- эффективность кода приложения;
- проектирование на физическом уровне.

Эффективность кода приложения

Приложения вносят свою нагрузку на системное программное обеспечение и на компонент Database Engine. По этой причине приложения могут усугублять проблемы с производительностью при нерациональном использовании системных ресурсов. Большинство проблем с производительностью в прикладных программах связано с неправильным выбором инструкций Transact-SQL и их последовательностью в программах приложений.

Улучшить общую производительность можно, используя, среди прочих, следующие два подхода:

- использовать кластеризованные индексы;
- исключить использование предиката `NOT IN`.

Кластеризованные индексы обычно улучшают производительность. Наилучшая производительность запроса для диапазона значений (range query) достигается использованием кластеризованного индекса для столбца в фильтре. Но при выборке только нескольких строк нет существенной разницы между использованием некластеризованного и кластеризованного индекса.

Предикат `NOT IN` не является оптимизируемым, иными словами, оптимизатор запросов не может использовать его в качестве аргумента поиска (часть запроса, которая ограничивает промежуточный результирующий набор запроса). Поэтому выражения с предикатом `NOT IN` всегда вызывают сканирование таблицы.

ПРИМЕЧАНИЕ

Дополнительные советы по модифицированию кода для улучшения общей производительности см. в *главе 19*.

Проектирование на физическом уровне

В процессе проектирования базы данных выбираются конкретные системы хранения и пути доступа для файлов базы данных. На этом этапе проектирования иногда рекомендуется денормализировать некоторые таблицы базы данных, чтобы улучшить производительность различных приложений баз данных.

Денормализация таблиц означает соединение вместе двух или более нормализованных таблиц в одну с некоторой избыточностью данных. Денормализацию можно продемонстрировать на следующем примере. Возьмем две исходные нормализованные таблицы базы данных `sample`: `department` и `employee` (табл. 20.1).

Таблица 20.1. *Нормализованные таблицы* `employee` *и* `department`

emp_no	emp_fname	emp_lname	dept_no
25348	Matthew	Smith	d3
10102	Ann	Jones	d3
18316	John	Barrimore	d1
29346	James	James	d2
2581	Elke	Hansel	d2
28559	Sybill	Moser	d1
9031	Elsa	Bertoni	d2

dept_no	dept_name	location
d1	Research	Dallas
d2	Accounting	Seattle
d3	Marketing	Dallas

(Дополнительную информацию по нормализации данных см. в *главе 1*.) Данные в этих двух таблицах можно определить, используя только одну таблицу dept_emp (табл. 20.2), содержащую денормализованные данные таблиц department и employee.

Таблица 20.2. *Денормализованная таблица dept_emp*

emp_no	emp_fname	emp_lname	dept_no	dept_name	location
25348	Matthew	Smith	d3	Marketing	Dallas
10102	Ann	Jones	d3	Marketing	Dallas
18316	John	Barrimore	d1	Research	Dallas
29346	James	James	d2	Accounting	Seattle
2581	Elke	Hansel	d2	Accounting	Seattle
28559	Sybij]	Moser	D1	Research	Dallas
9031	Elsa	Bertoni	d2	Accounting	Seattle

В отличие от таблиц department и employee, которые не содержат избыточных данных, таблица dept_emp содержит много избыточных данных, поскольку два столбца этой таблицы (dept_name и location) зависят от столбца dept_no.

Денормализация данных имеет два преимущества и два недостатка. Сначала перечислим преимущества. Первое преимущество: если для получения часто требуемых запросами данных один столбец таблицы зависит от другого (как столбец dept_name таблицы dept_emp зависит от столбца dept_no), то это позволяет избежать использования операции соединения, которая имеет отрицательное влияние на производительность приложений. Второе преимущество: для денормализованных данных требуется меньшее число таблиц, чем для нормализованных.

С другой стороны, для хранения денормализованной таблицы требуется больший объем дискового пространства, а модификация данных усложняется вследствие избыточности данных.

Другим фактором в физическом проектировании базы данных, улучшающим производительность, является создание индексов. Несколько руководствующих принципов по созданию индексов изложены в *главе 10*, а в этой главе приведено несколько примеров.

Компонент Database Engine и производительность

Компонент Database Engine может оказывать существенное влияние на производительность всей системы. Наиболее важными составляющими Database Engine, которые влияют на производительность, являются следующие:

- оптимизатор запросов;
- блокировки.

Оптимизатор запросов

Оптимизатор формулирует несколько планов выполнения запроса для выборки строк данных, требуемых для обработки запроса, а затем решает, какой из этих планов использовать. Решение о выборе наиболее подходящего плана выполнения принимается на основе таких соображений, включая то, какие индексы использовать, как обращаться к таблицам и в каком порядке таблицы должны соединяться. Все эти решения могут оказать значительное влияние на производительность приложений базы данных. Оптимизатор запросов подробно рассмотрен в предыдущей главе.

Блокировки

Система баз данных использует блокировки в качестве механизма предохранения работы одного пользователя от работы другого. Поэтому блокировки используются для управления одновременным доступом к данным несколькими пользователями и для предотвращения потенциальных ошибок, возможных в случае одновременного доступа к одним и тем же данным.

Блокировка оказывает влияние на производительность системы вследствие ее гранулярности, т. е. зависит от размера блокируемого объекта и уровня блокировки. Блокировка на уровне строк обеспечивает наилучшую производительность системы, поскольку она блокирует только одну строку страницы, не затрагивая всех других, и, следственно, позволяет выполнять больше параллельных действий, чем блокировки на уровне страницы или таблицы.

Уровни изоляции влияют на длительность блокировки для инструкций `SELECT`. Применяя более низкие уровни изоляции, такие как `READ UNCOMMITTED` и `READ COMMITTED`, может быть улучшена доступность данных и, следственно, увеличено количество одновременных обращений к данным. Блокировки и уровни изоляции подробно рассмотрены в *главе 13*.

Системные ресурсы и производительность

Компонент Database Engine выполняется под управлением операционной системы Windows, которая сама использует базовые системные ресурсы. Эти ресурсы оказывают значительное влияние на производительность как операционной системы, так и системы баз данных. Производительность любой системы баз данных зависит от четырех основных системных ресурсов:

- центрального процессора (CPU — central processing unit);
- оперативной памяти;
- дисковых операций ввода/вывода;
- сетевого окружения.

Центральный процессор вместе с оперативной памятью является ключевым компонентом, определяющим быстродействие компьютера. Он также является ключом к производительности системы, поскольку он управляет другими ресурсами системы

и выполняет все приложения. Центральный процессор выполняет пользовательские процессы и взаимодействует с другими ресурсами системы. Проблемы с производительностью, связанные с центральным процессором, могут возникать, когда операционная система и пользовательские программы обращаются к нему со слишком большим количеством запросов. Обычно, чем мощнее центральный процессор компьютера, тем выше вероятность лучшей общей производительности системы.

Компонент Database Engine динамически запрашивает и по мере надобности освобождает память. Проблемы производительности, связанные с памятью, могут возникать только в тех случаях, когда ее недостаточно для выполнения требуемой работы. В таких случаях многие страницы оперативной памяти записываются в *файл подкачки* (pagefile). Понятие файла подкачки подробно рассматривается далее в этой главе. Частое обращение в файл подкачки может отрицательно сказаться на производительности системы. Поэтому, по аналогии с центральным процессором, чем больше памяти установлено на компьютере, тем выше общая производительность системы.

В плане дисковых операций ввода/вывода важность представляют два аспекта: скорость диска и скорость передачи данных. Скорость диска определяется скоростью выполнения операций записи и чтения диска. Скорость передачи данных определяется объемом данных, который можно записать на диск в единицу времени (обычно в секунду). Очевидно, что чем быстрее диск, тем больший объем данных он может обработать. Кроме этого, при одновременном использовании системы баз данных большим количеством пользователей, большое количество дисков обычно лучше, чем один диск. В этом случае доступ к данным обычно распределяется между многими дисками, повышая, таким образом, общую производительность системы.

В конфигурации "клиент-сервер" производительность системы баз данных иногда страдает при большом количестве клиентских подключений. В таком случае объем данных, который требуется передать по сети, может превышать ее пропускную способность. Чтобы избежать такого узкого места в производительности, следует принимать во внимание следующие рекомендации:

- если сервер баз данных отправляет приложению какие-либо строки, отправлять следует только лишь те строки, которые действительно требуются приложению;
- если продолжительное пользовательское приложение выполняется сугубо на стороне клиента, то его следует переместить на сторону сервера (выполняя его, например, как хранимую процедуру).

Все системные ресурсы зависят друг от друга. Это означает, что проблемы производительности с одним ресурсом могут вызвать проблемы производительности с другими. Подобным образом улучшения, связанные с одним ресурсом, могут значительно повысить производительность какого-либо другого или даже всех ресурсов. Рассмотрим два примера.

- При увеличении количества центральных процессоров нагрузка на них распределяется равномерно, что может решить проблему узкого места с дисковыми операциями ввода/вывода. С другой стороны, неэффективное использование

центрального процессора часто является результатом большой нагрузки на дисковый ввод/вывод и/или оперативную память.

- Больший объем оперативной памяти повышает шансы, что требуемая приложению страница будет найдена в памяти, и ее не придется считывать с файла подкачки с диска, что повышает уровень производительности. И напротив, чтение данных с диска вместо получения их с намного более быстродействующей памяти, значительно тормозит систему, особенно при большом количестве конкурентных процессов.

Дисковые операции ввода/вывода и операции с памятью подробно рассматриваются в следующих разделах.

Дисковые операции ввода/вывода

Основным назначением базы данных является хранение, выборка и модифицирование данных. Поэтому компонент Database Engine, подобно любой другой системе баз данных, должен выполнять большой объем дисковых операций. В отличие от других системных ресурсов, подсистема дискового привода имеет две движущиеся составляющие: дисковые пластины и головки чтения-записи. По сравнению с другими операциями компьютерной системы, вращение пластин и перемещение дисковых головок в требуемое положение занимают много времени, поэтому операции чтения с диска и записи на него являются двумя наиболее затратными операциями системы баз данных. (Например, доступ к диску намного медленнее доступа к оперативной памяти.)

Компонент Database Engine хранит данные в страницах размером в 8 Кбайт. Кэш оперативной памяти также разделен на страницы размером по 8 Кбайт. Система читает данные по страницам. Операции чтения выполняются не только для выборки данных, но также для любых инструкций изменений данных, таких как `UPDATE` и `DELETE`, поскольку система базы данных должна прочесть данные, прежде чем она сможет их модифицировать.

Если требуемая страница находится в кэше, то обращаться к диску не требуется. Такая операция дискового ввода/вывода называется *логическим вводом/выводом* или *логическим чтением*. Если же требуемая страница отсутствует в кэше, то она считывается с диска и помещается в буферный кэш. Такой тип операции дискового ввода/вывода называется *физическим вводом/выводом* или *физическим чтением*. Буферный кэш является памятью совместного использования, поскольку компонент Database Engine использует архитектуру памяти с единым адресным пространством. Поэтому к одной и той же странице могут обращаться несколько пользователей. При модифицировании данных в буферном кэше выполняется операция логической записи. Операция физической записи происходит только при сохранении данных из кэша на диск. Поэтому, прежде чем записывать страницу на диск, с ней можно выполнить несколько операций логической записи, пока она еще находится в кэше.

Компонент Database Engine предоставляет несколько средств, которые оказывают большое влияние на производительность в связи со значительным потреблением

ими ресурсов ввода/вывода. Database Engine имеет несколько следующих элементов, которые оказывают большое влияние на производительность, поскольку они существенно пользуются ресурсами ввода/вывода:

- упреждающее чтение (read ahead);
- контрольные точки.

Контрольные точки подробно рассмотрены в *главе 16*, а упреждающее чтение описано в следующем подразделе.

Упреждающее чтение

Оптимальным режимом работы системы базы данных было бы выполнение чтения данных только из памяти и никогда не ожидать завершения операции чтения с диска. Наилучшим способом выполнения этой задачи будет знать, какие следующие несколько страниц будут требоваться пользователю, и считывать эти страницы с диска в буферный пул до того, как пользовательский процесс запросит их. Этот механизм получения данных называется *упреждающим чтением* и позволяет системе оптимизировать производительность, благодаря эффективной обработке большого объема данных.

Составляющая Database Engine, называемая *Read Ahead Manager*, полностью внутренне управляет процессами упреждающего чтения, так что пользователь не имеет никакой возможности воздействовать на этот процесс. Вместо использования обычных страниц размером по 8 Кбайт, компонент Database Engine использует в качестве единицы памяти для упреждающих операций чтения блоки данных размером в 64 Кбайт. Это существенно повышает пропускную способность запросов ввода/вывода. Механизм упреждающего чтения используется системой баз данных для выполнения объемных сканирований таблиц и сканирований диапазонов индексов. Сканирования таблицы выполняются с использованием информации, которая хранится в страницах *карты распределения индекса* (index allocation map, IAM), для создания последовательного списка дисковых адресов, по которым требуется выполнить чтение. (*Страницы IAM* — это страницы размещения, содержащие информацию об экстентах, используемых таблицей или индексами.) Это позволяет системе баз данных оптимизировать свой ввод/вывод путем объемных последовательных операций чтения данных в порядке их размещения на диске. Read Ahead Manager считывает до 2 Мбайт данных за один раз. Каждый экстент считывается с помощью одной операции.

ПРИМЕЧАНИЕ

Компонент Database Engine поддерживает множественные одновременные операции упреждающего чтения для каждого файла, связанного со сканированием таблицы. Для этой возможности может быть полезным применение расслоенных наборов дисковых накопителей.

Для диапазонов индексов, чтобы определить страницы для считывания, компонент Database Engine использует информацию из промежуточного уровня индексных страниц, находящегося сразу же над уровнем листьев. Система сканирует все эти

страницы и создает список страниц листьев, которые должны быть прочитаны. В процессе этой операции распознаются смежные страницы и считываются в один прием. Когда требуется выполнить выборку большого количества страниц, компонент Database Engine планирует выполнение блока операций чтения за один раз.

Однако механизм упреждающего чтения может также иметь и отрицательное воздействие на производительность, если приходится читать слишком много страниц, излишне заполняя кэш. Единым решением этой проблемы будет создание индексов, которые в действительности необходимы.

Существует несколько разных счетчиков производительности и динамических административных представлений, которые связаны с деятельностью упреждающего чтения. Они рассматриваются далее в этой главе.

Операции с памятью

Оперативная память является важнейшим компонентом не только для выполняющихся приложений, но также и для операционной системы. Чтобы выполнить приложение, его необходимо загрузить в оперативную память, для чего выделяется определенной объем памяти. (По терминологии Microsoft, общий объем памяти, который доступен приложению, называется *адресным пространством* (address space).)

Операционная система Windows поддерживает виртуальную память. Это означает, что приложениям доступен общий объем физической оперативной памяти (или RAM), установленной на компьютере, плюс объем виртуальной оперативной памяти в виде особого файла на диске, который называется *файлом подкачки* (swap file) или страничным файлом (page file). (В операционной системе Windows файл подкачки называется *pagefile.sys*.) Когда данные перемещаются из физической памяти в виртуальную, то они помещаются в файл подкачки. Если система получает запрос обращения к данным, которые не находятся в оперативной памяти, она загружает их из виртуальной памяти и генерирует так называемую *ошибку страницы* (page fault).

ПРИМЕЧАНИЕ

Файл подкачки *pagefile.sys* следует размещать на другом физическом диске, чем тот, на котором хранятся файлы, используемые компонентом Database Engine, поскольку процесс обращения к файлу подкачки может иметь негативное влияние на дисковые операции ввода/вывода.

Только часть всего приложения загружается в оперативную память (RAM). (В частности, страницы, к которым недавно выполнялось обращение, находятся в RAM.) Когда приложению требуются данные, которых нет в оперативной памяти, операционной системе нужно переместить в RAM страницу с этими данными из файла подкачки. Этот процесс называется *подкачкой страниц по требованию* (demand paging). Чем чаще системе приходится выполнять подкачку страниц, тем хуже ее общая производительность.

ПРИМЕЧАНИЕ

Когда требуется страница, отсутствующая в RAM, то самая старая страница из адресного пространства приложения перемещается в файл подкачки, чтобы освободить место для новой страницы. Перемещения страниц всегда выполняются в пределах адресного пространства текущего приложения. Поэтому возможность замены страниц в адресном пространстве других выполняющихся приложений отсутствует.

Как уже упоминалось, ошибка страницы возникает, когда страница с запрашиваемой приложением информацией отсутствует в должном месте в оперативной памяти компьютера. Эта информация может быть или выгруженной в файл подкачки или находиться где-то в другом месте в оперативной памяти. Поэтому существует два типа ошибки страницы:

- *ошибка страницы диска* (hard page fault) — страница была выгружена из оперативной памяти в файл подкачки и ее с диска требуется загрузить в память;
- *ошибка программной страницы* (soft page fault) — страница находится в другом месте оперативной памяти.

Ошибки программной страницы потребляют только ресурсы оперативной памяти. Поэтому в плане производительности они лучше, чем ошибки страницы диска, которые вызывают необходимость выполнения операций чтения и записи диска.

ПРИМЕЧАНИЕ

Несмотря на предполагаемое из названия значение, ошибки страниц в действительности не являются ошибками, а являются нормальными событиями операционной системы Windows, поскольку операционной системе постоянно требуется выгружать из памяти страницы выполняющихся приложений, чтобы удовлетворить потребность в памяти для запускающихся приложений. Но слишком большой объем замещения страниц (особенно при ошибках страниц диска) представляет собой серьезную проблему для производительности, поскольку это может приводить к перегрузке дискового устройства и вызывать дополнительное расходование вычислительной мощности центрального процессора.

Мониторинг производительности

Все факторы, которые оказывают влияние на производительность, можно отслеживать, используя для этого различные средства. Эти средства можно сгруппировать следующим образом:

- счетчики системного монитора Performance Monitor;
- динамические административные представления (DMV) и представления каталогов;
- команды DBCC;
- системные хранимые процедуры.

В этом разделе сначала дается обзор монитора Performance Monitor, после чего описываются все средства для мониторинга производительности, связанной с че-

тырьмя факторами: центральным процессором, памятью, доступом к диску и сетевым окружением.

Обзор монитора Performance Monitor

Монитор Performance Monitor (Системный монитор) представляет собой графический инструмент операционной системы Windows, который предоставляет возможность отслеживать деятельность этой операционной системы и деятельность системы баз данных. Полезность этого средства состоит в том, что он тесно интегрирован с операционной системой Windows и вследствие этого отображает достоверную информацию, связанную с различными проблемами производительности. Монитор Performance Monitor предусмотрен для большого числа объектов производительности, каждый из которых содержит несколько счетчиков. Показания этих счетчиков можно отслеживать локально или по сети.

Монитор Performance Monitor поддерживает три различных режима представления результатов:

- *графический режим* — отображает выбранные счетчики в виде цветных линий, где ось X представляет время, а ось Y — значения счетчика; это режим отображения по умолчанию;

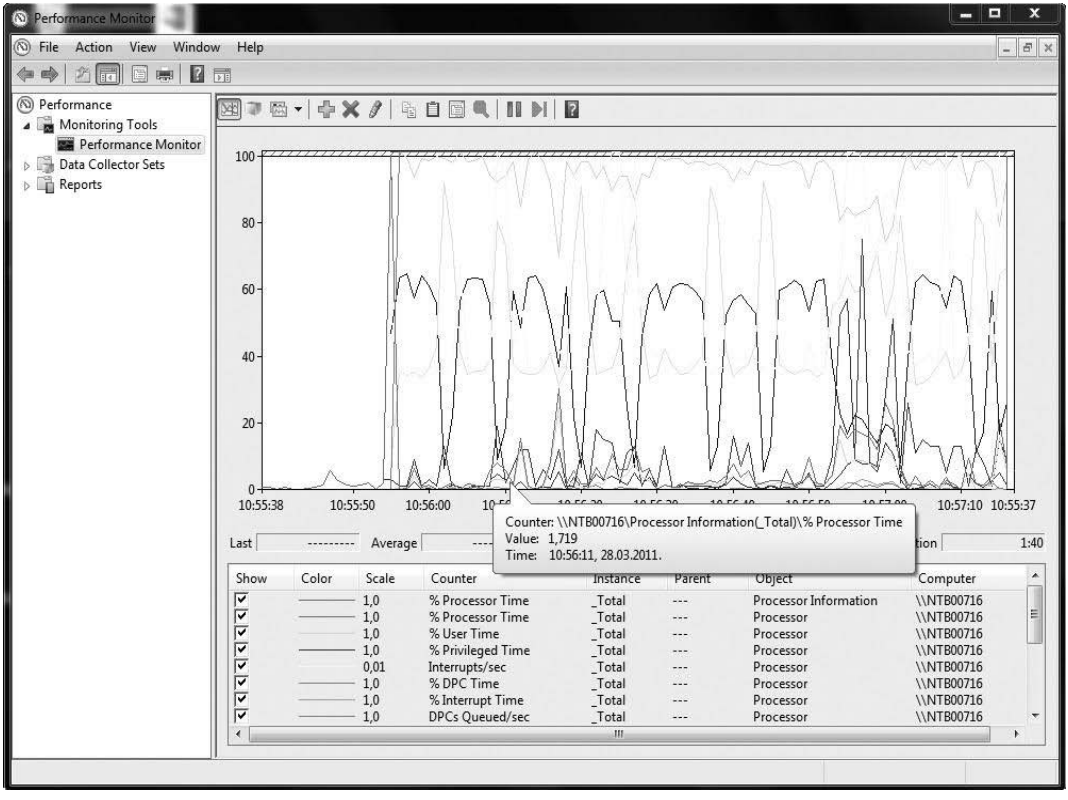


Рис. 20.1. Начальное окно монитора Performance Monitor

- *режим гистограммы* — отображает выбранные счетчики в виде цветных горизонтальных столбцов, которые представляют выборочные значения данных;
- *режим отчета* — отображает значения счетчиков в текстовом формате.

Для запуска монитора Performance Monitor щелкните по кнопке **Start** (Пуск), выберите **Control Panel | Administrative Tools | Performance Monitor** (Панель управления | Администрирование | Системный монитор). Начальное окно монитора Performance Monitor (рис. 20.1) по умолчанию содержит один счетчик: % Processor Time (Object: Processor).

Это очень важный счетчик, но для полноценного мониторинга системы нужно отображать значения и других счетчиков.

Чтобы добавить счетчик для мониторинга, щелкните значок с изображением зеленого плюса (+) на панели инструментов системного монитора, выберите объект производительности, которому принадлежит счетчик, выберите счетчик и нажмите кнопку **Add** (Добавить). Чтобы удалить счетчик, выделите его строку в нижней панели окна монитора и нажмите значок с изображением красного крестика на панели инструментов системного монитора. В следующих разделах описываются, среди прочего, наиболее важные счетчики для отслеживания работы центрального процессора, памяти, ввода/вывода и сетевого окружения.

Мониторинг центрального процессора

Этот раздел содержит два подраздела, связанных с мониторингом центрального процессора. В первом подразделе описывается несколько счетчиков системного монитора, а во втором обсуждаются представления каталогов и динамические административные представления (DMV), которые можно использовать для этих же целей.

Мониторинг центрального процессора посредством счетчиков

Следующие счетчики применяются для мониторинга центрального процессора:

- % Processor Time (Object: Processor);
- % Interrupt Time (Object: Processor);
- Interrupts/sec (Object: Processor);
- Processor Queue Length (Object: System).

Счетчик *% Processor Time* (% загруженности процессора) отображает общесистемный уровень использования центрального процессора и является основным индикатором активности процессора. Чем меньше значение этого счетчика, тем лучшее использование центрального процессора. Если значение этого счетчика постоянно выше 90, то следует попытаться уменьшить нагрузку на центральный процессор. (Использование центрального процессора с загруженностью 100% допустимо только лишь в течение коротких промежутков времени.)

Счетчик *% Interrupt Time* (% времени прерываний) отображает процент времени, которое центральный процессор затрачивает на обслуживание аппаратных преры-

ваний. Значения этого счетчика важно знать, если, по крайней мере, одно аппаратное устройство пытается получить процессорное время.

Счетчик *Interrupts/sec* (прерываний/сек) отображает, сколько раз в секунду процессор получает аппаратные прерывания для обслуживания запросов от периферийных устройств. Значение этого счетчика обычно может быть большим в окружении с высоким уровнем использования диска или высокими сетевыми требованиями.

Счетчик *Processor Queue Length* (длина очереди процессора) указывает количество потоков, ожидающих выполнения. Если значение этого счетчика выше 5, когда уровень использования процессора приближается к 100%, то это может быть признаком наличия большего количества активных потоков, чем центральный процессор в состоянии обслужить.

Мониторинг центрального процессора посредством представлений

Для мониторинга использования центрального процессора можно использовать, среди прочих, следующие представления каталога и динамические административные представления (DVM):

- `sys.sysprocesses`;
- `sys.dm_exec_requests`;
- `sys.dm_exec_query_stats`.

Представление каталога `sys.sysprocesses` может быть полезным для определения процессов, которые используют больше всего процессорного времени. Использование этого представления показано в примере 20.1.

Пример 20.1. Использование представления каталога `sys.sysprocesses`

```
USE master;
SELECT spid, dbid, uid, cpu
   FROM master.dbo.sysprocesses
     order by cpu DESC;
```

Это представление содержит информацию о процессах, выполняющихся на экземпляре сервера. Эти процессы могут быть клиентскими или системными процессами. Представление `sys.sysprocesses` принадлежит системной базе данных `master`. Наиболее важными столбцами этого представления являются столбцы `spid` (идентификатор сессии), `dbid` (идентификатор текущей базы данных), `uid` (идентификатор пользователя, выполняющего текущую команду) и `cpu` (общее время центрального процессора, затраченного на данный процесс).

Динамическое административное представление `sys.dm_exec_requests` предоставляет ту же информацию, что и представление каталога `sys.sysprocesses`, но его столбцы называются по-другому. Использование этого динамического административного представления для отображения той же информации, что и в примере 20.1, показано в примере 20.2.

Пример 20.2. Использование ДАП `sys.dm_exec_requests`

```
SELECT session_id, database_id, user_id, cpu_time, sql_handle
   FROM sys.dm_exec_requests
       order by cpu_time DESC;
```

Столбец `sql_handle` этого представления указывает на область, в которой хранится весь пакет. Объем отображаемой информации можно свести к одной инструкции, используя столбцы `statement_start_offset` и `statement_end_offset`. (Назначение других столбцов должно быть понятным по их названиям.)

ПРИМЕЧАНИЕ

Это динамическое административное представление особенно полезно, если требуется определить продолжительные запросы.

Другим динамическим административным представлением, которое можно использовать для отображения информации о кэшированных инструкциях Transact-SQL и хранимых процедурах, которые используют больше всего времени центрального процессора, является `sys.dm_exec_query_stats`. Это динамическое административное представление подробно рассмотрено в *главе 19*. Использование этого представления показано в примере 20.3.

Пример 20.3. Использование динамического административного представления `sys.dm_exec_query_stats`

```
SELECT TOP 20 SUM(total_worker_time) AS cpu_total,
     SUM(execution_count) AS exec_ct, COUNT(*) AS all_stmts, plan_handle
     FROM sys.dm_exec_query_stats
     GROUP BY plan_handle
     ORDER BY cpu_total;
```

В столбце `total_worker_time` представления `sys.dm_exec_query_stats` отображается общее время центрального процессора, использованное на выполнение кэшированных SQL-инструкций и хранимых процедур с момента их компиляции. В столбце `execution_count` отображается количество времени, которое кэшированных планы будут выполняться с момента их последней компиляции. Предложение `TOP` уменьшает количество отображаемых строк в соответствии с параметром и предложением `ORDER BY`. Это предложение подробно рассмотрено в *главе 23*.

Мониторинг памяти

Этот раздел содержит три подраздела, связанных с мониторингом памяти. В первом подразделе описывается несколько счетчиков системного монитора Performance Monitor, а во втором рассматриваются динамические административные представления, которые можно использовать для этих же целей; последний подраздел отведен рассмотрению использования команды `DBCC MEMORYSTATUS`.

Мониторинг памяти посредством счетчиков

Для мониторинга памяти используются следующие счетчики системного монитора:

- Buffer Cache Hit Ratio (Object: Memory);
- Pages/sec (Object: Memory);
- Page Faults/sec (Object: Memory).

Счетчик *Buffer Cache Hit Ratio* (процент попаданий в буферный кэш) отображает количество страниц в процентах, которые не требуется считывать с диска. Чем выше это процентное отношение, тем реже системе требуется обращаться к диску за данными, вследствие чего повышается общая производительность системы. Обратите внимание, что для этого счетчика не существует идеального значения, потому что оно зависит от конкретного приложения.

ПРИМЕЧАНИЕ

Этот счетчик отличается от большинства других счетчиков тем, что отображает не значение реального времени, а среднее значение за все дни с момента последнего запуска компонента Database Engine.

Счетчик *Pages/sec* (обмен страниц/сек) отображает объем обмена страниц, т. е. количество страниц считываемых или записываемых в секунду. Этот счетчик является важным индикатором типов ошибок, которые могут вызвать проблемы с производительностью. Если значение этого счетчика слишком большое, то следует рассмотреть установку дополнительной физической памяти.

Счетчик *Page Faults/sec* (ошибок страницы/сек) отображает среднее количество ошибок страниц в секунду. Этот счетчик включает как программные ошибки, так и ошибки страниц диска. Как уже упоминалось, ошибки страниц возникают, когда система обращается к странице памяти, которая в настоящий момент отсутствует в рабочей области физической памяти. Если запрашиваемая страница находится в списке простаивающих страниц или в настоящий момент используется другим процессом в совместном режиме, то генерируется *ошибка программной страницы* и ссылка на память определяется без физического обращения к диску. Но если запрошенная страница в настоящее время находится в файле подкачки, то генерируется *ошибка страницы диска*, и данные должны быть загружены с диска.

Мониторинг памяти с использованием динамических административных представлений

Для мониторинга памяти применяются следующие динамические административные представления:

- `sys.dm_os_memory_clerks`;
- `sys.dm_os_memory_objects`.

Представление `sys.dm_os_memory_clerks` возвращает набор всех служб памяти, которые являются активными в текущем экземпляре сервера. С помощью этого пред-

ставления можно получить распределение памяти для различных типов памяти. Использование этого представления показано в примере 20.4.

Пример 20.4. Использование представления `sys.dm_os_memory_clerks`

```
SELECT type, SUM(pages_kb)
   FROM sys.dm_os_memory_clerks
   WHERE pages_kb != 0
   GROUP BY type
    ORDER BY 2 DESC;
```

Столбец `type` представления `sys.dm_os_memory_clerks` содержит описание типа службы памяти, а в столбце `pages_kb` указывается объем памяти, выделенный с помощью распределителя одиночных страниц узла памяти.

ПРИМЕЧАНИЕ

Диспетчер памяти компонента Database Engine имеет трехуровневую иерархию. Самый нижний уровень этой иерархии содержит узлы памяти, следующий уровень содержит службы, кэши и пулы памяти. Последний уровень содержит объекты памяти, которые обычно применяются для распределения памяти.

Представление `sys.dm_os_memory_objects` возвращает объекты памяти, которые в настоящее время распределены системой баз данных. Это динамическое административное представление в основном используется для анализа использования памяти и для определения возможного недостатка памяти, как это показано в примере 20.5.

Пример 20.5. Использование динамического административного представления `sys.dm_os_memory_objects`

```
SELECT types, SUM(pages_in_bytes) AS total_memory
   FROM sys.dm_os_memory_objects
   GROUP BY type
   ORDER BY total_memory DESC;
```

Запрос, приведенный в примере 20.5, группирует все объекты памяти по их типу, а затем использует значения в столбце `pages_in_bytes` для отображения общего объема памяти в каждой группе.

Мониторинг памяти с использованием команды *DBCC MEMORYSTATUS*

Команда `DBCC MEMORYSTATUS` позволяет получить моментальный снимок текущего состояния памяти компонента Database Engine. Вывод этой команды полезен при поиске неполадок, связанных с потреблением памяти компонентом Database Engine, или определенных ошибок недостатка памяти (многие из которых автоматически записывают этот вывод в журнал ошибок).

Вывод этой команды состоит из нескольких частей, включая, среди прочих, часть `Process/System Counts`, которая содержит важную информацию об общем объеме памяти (параметр `Working set`), и фактический объем памяти, используемый компонентом Database Engine (параметр `Available Physical Memory`).

Мониторинг дисковой системы

Этот раздел состоит из двух частей, в первой из которых обсуждаются несколько счетчиков монитора Performance Monitor для наблюдения за дисковой системой, а во второй рассматриваются соответствующие динамические административные представления.

Мониторинг дисковой системы с использованием счетчиков

Для мониторинга дисковой системы применяются следующие счетчики монитора Performance Monitor:

- % Disk Time (Object: Physical Disk);
- Current Disk Queue Length (Object: Physical Disk);
- Disk Read Bytes/sec (Object: Physical Disk);
- Disk Write Bytes/sec (Object: Physical Disk);
- % Disk Time (Object: Logical Disk);
- Current Disk Queue Length (Object: Logical Disk);
- Disk Read Bytes/sec (Object: Logical Disk);
- Disk Write Bytes/sec (Object: Logical Disk).

Как можно видеть из предшествующего списка, названия счетчиков для объекта физического диска Physical Disk и для объекта логического диска Logical Disk одинаковые. (Различие между физическими и логическими объектами объяснялось в *главе 5*.) Назначение этих счетчиков для этих объектов также одинаковое, поэтому мы рассмотрим только счетчики для объекта физического диска Physical Disk.

Счетчик *% Disk Time* (% активности диска) отображает фактическое время работы диска. Он предоставляет хорошее относительное представление о занятости дисковой системы. Для определения потенциальной необходимости расширения возможностей ввода/вывода его следует использовать в течение длительного периода времени.

Счетчик *Current Disk Queue Length* (текущая длина очереди диска) отображает количество операций ввода/вывода, ожидающих получить доступ к диску. Это значение должно быть как можно меньше.

Счетчик *Disk Read Bytes/sec* (скорость чтения с диска (байт/сек)) отображает скорость передачи данных (в байтах в секунду) с жесткого диска при операциях чтения, а счетчик *Disk Write Bytes/sec* (скорость записи на диск (байт/с)) делает то же самое для операций записи.

Мониторинг дисковой системы с использованием динамических административных представлений

Для отображения информации о дисковой системе полезными могут быть следующие динамические административные представления:

- `sys.dm_os_wait_stats`;
- `sys.dm_io_virtual_file_stats`.

Представление `sys.dm_os_wait_stats` возвращает информацию об ожиданиях в выполняемых потоках. Это представление можно использовать для диагностики проблем производительности как всего сервера Database Engine, так и отдельных запросов и пакетов. Использование этого представления показано в примере 20.6.

Пример 20.6. Использование динамического административного представления `sys.dm_os_wait_stats`

```
SELECT wait_type, waiting_tasks_count, wait_time_ms
   FROM sys.dm_os_wait_stats
   ORDER BY wait_type;
```

Наиболее важными столбцами этого представления являются столбцы `wait_type` и `waiting_tasks_count`. Первый столбец содержит названия типов ожидания, а второй — количество ожиданий соответствующего типа.

Второе представление, `sys.dm_io_virtual_file_stats`, возвращает статистику ввода/вывода для файлов данных и журнала. Использование этого представления показано в примере 20.7.

Пример 20.7. Использование динамического административного представления `sys.dm_io_virtual_file_stats`

```
SELECT database_id, file_id, num_of_reads,
   num_of_bytes_read, num_of_bytes_written
   FROM sys.dm_io_virtual_file_stats (NULL, NULL);
```

Назначение столбцов представления `sys.dm_io_virtual_file_stats` должно быть понятным из их названий. Как можно видеть в примере 20.7, это представление имеет два параметра. В первом параметре, `database_id`, указывается однозначный идентификационный номер базы данных, а во втором, `file_id`, — идентификатор файла. Когда указано значение `NULL`, тогда возвращаются все базы данных, т. е. все файлы экземпляра Database Engine.

Мониторинг сетевого интерфейса

В трех частях этого раздела рассматривается мониторинг сетевого интерфейса. В первой части описываются несколько счетчиков монитора Performance Monitor, во второй обсуждаются соответствующие этим счетчикам динамические администрат-

ративные представления, а в третьей рассматривается системная процедура sp_monitor.

Мониторинг сетевого интерфейса с помощью счетчиков

Для мониторинга сетевого интерфейса используются следующие счетчики монитора Performance Monitor:

- Bytes Total/sec (Object: Network Interface);
- Bytes Received/sec (Object: Network Interface);
- Bytes Sent/sec (Object: Network Interface).

Счетчик *Bytes Total/sec* (всего байт/сек) отображает общее количество байтов, которые были приняты и отправлены по сети в секунду. Сюда входит как трафик сервера Database Engine, так и другой сетевой трафик. Предполагая, что сервер является выделенным сервером баз данных, подавляющий объем измеряемого этим счетчиком сетевого трафика должен принадлежать компоненту Database Engine. Постоянно низкое значение этого счетчика указывает на возможность наличия сетевых проблем, мешающих работе приложения.

Счетчики *Bytes Received/sec* (получено байт/сек) и *Bytes Sent/sec* (отправлено байт/сек) отображают объем данных (в байтах в секунду), которые сервер получает из сети и отправляет в сеть соответственно. Эти счетчики могут быть полезными для выяснения уровня сетевой активности сервера.

Мониторинг сетевого интерфейса с помощью динамического административного представления

Представление sys.dm_exec_connections возвращает информацию о соединениях, установленных с экземпляром компонента Database Engine и подробности каждого соединения. Использование этого представления показано в примерах 20.8 и 20.9.

Пример 20.8. Отображение сетевого протокола и механизма проверки подлинности

```
SELECT net_transport, auth_scheme
   FROM sys.dm_exec_connections
   WHERE session_id=@@SPID;
```

Запрос в примере 20.8 возвращает основную информацию о текущем соединении: сетевой транспортный протокол и механизм аутентификации. Условие в предложении WHERE сокращает вывод информации о текущей сессии. (Глобальная переменная @@spid, которая рассматривается в *главе 4*, возвращает идентификатор текущего серверного процесса.)

Пример 20.9. Отображение количества пакетов чтения и записи

```
SELECT num_reads, num_writes
   FROM sys.dm_exec_connections;
```

В примере 20.9 показано использование двух важных столбцов динамического административного представления `sys.dm_exec_connections`, а именно столбцов `num_reads` и `num_writes`. В первом столбце отображается количество прочитанных пакетов в текущем соединении, а во втором — количество записанных пакетов.

Мониторинг сетевого интерфейса с помощью системной процедуры *sp_monitoring*

Системная процедура `sp_monitoring` может быть очень полезной для мониторинга состояния сетевого интерфейса, поскольку она возвращает информацию об общем текущем количестве отправленных и принятых пакетов. Кроме того, эта системная процедура возвращает статистические данные о SQL Server, такие как количество секунд, затраченных центральным процессором на системную активность, количество секунд простоя системы, а также количество входов (или попыток входа) в систему.

Выбор подходящего инструмента для мониторинга

Выбор соответствующего инструмента для мониторинга зависит от аспектов производительности, которые требуется отслеживать, и режима мониторинга. Мониторинг может осуществляться в следующих режимах:

- в режиме реального времени;
- в отложенном режиме (например, сохраняя информацию в файл).

Мониторинг в режиме реального времени означает, что информация о проблемах производительности доступна сразу же при их возникновении. Если требуется отобразить фактические значения одного или нескольких аспектов производительности, например количество пользователей или количество попыток входа, то следует использовать динамические административные представления по причине их простоты. Более того, динамические административные представления можно использовать только для мониторинга в режиме реального времени. Поэтому для мониторинга производительности активностей в течение определенного периода времени требуется использовать такое средство, как SQL Server Profiler (см. *следующий раздел*).

Наверное, самым лучшим универсальным инструментом для мониторинга является монитор Performance Monitor, потому что он содержит большое количество опций, предоставляемых этим инструментом. Во-первых, можно выбирать требуемые аспекты производительности для мониторинга и отображать их все одновременно. А во-вторых, монитор Performance Monitor позволяет установить пороговые значения для счетчиков определенных аспектов производительности, чтобы генерировать предупреждения и извещать операторов. Таким образом, можно своевременно реагировать на возникновение каких-либо узких мест в производительности. Нако-

нец, аспекты производительности можно документировать и позже исследовать файлы с записями журналов графиков.

В следующих разделах мы рассмотрим приложение SQL Server Profiler и помощник Database Engine Tuning Advisor.

Приложение SQL Server Profiler

Приложение SQL Server Profiler представляет собой графический инструмент, который позволяет системному администратору выполнять мониторинг и записывать об активностях базы данных и сервера, таких как вход в систему, работа пользователей и исполнение приложений. Приложение SQL Server Profiler может отображать информацию о нескольких активностях сервера в реальном режиме, а также может создавать фильтры для концентрирования внимания на определенных событиях пользователя, типах команд или типах инструкций языка Transact-SQL. Помимо прочего, посредством этого приложения можно отслеживать следующие события:

- соединения, попытки соединений, ошибки соединений и отсоединения;
- использование центрального процессора пакетом;
- проблемы взаимоблокировок;
- все инструкции DML (`SELECT`, `INSERT`, `UPDATE` и `DELETE`);
- запуск и завершение работы хранимой процедуры.

Наиболее полезной особенностью приложения SQL Server Profiler является возможность захвата операций, связанных с запросами. Эти операции можно использовать в качестве ввода для помощника Database Engine Tuning Advisor, который позволяет выбирать индексы и индексированные представления для запросов. По этой причине в следующем разделе рассматриваются возможности приложения SQL Server Profiler вместе с помощником Database Engine Tuning Advisor.

Помощник Database Engine Tuning Advisor

Помощник Database Engine Tuning Advisor является частью общей системы и позволяет автоматизировать физическое проектирование баз данных. Как упоминалось ранее, помощник Database Engine Tuning Advisor тесно связан с приложением SQL Server Profiler, которое может отображать информацию о нескольких активностях сервера одновременно в режиме реального времени, а также создавать фильтры, чтобы фокусироваться на определенных событиях пользователя, типах команд или инструкциях Transact-SQL.

Специфичной особенностью приложения SQL Server Profiler, которая используется помощником Database Engine Tuning Advisor, является его способность наблюдать за выполняемыми пользователями пакетами и записывать их, а также способность предоставлять информацию о производительности, такую как использование цен-

трального процессора пакетом и соответствующие статистические данные ввода/вывода, как объясняется далее в этом разделе.

Предоставление информации помощнику Database Engine Tuning Advisor

Помощник Database Engine Tuning Advisor обычно используется вместе с приложением SQL Server Profiler для автоматизации процесса настройки системы. С помощью приложения SQL Server Profiler можно записывать в файл трассировки информацию об исследуемой рабочей нагрузке. (Вместо файла рабочей нагрузки можно использовать любой файл, содержащий набор инструкций языка Transact-SQL. В этом случае использовать приложение SQL Server Profiler не требуется.) После этого помощник Database Engine Tuning Advisor может прочитать этот файл и порекомендовать создать для данной рабочей нагрузки несколько физических объектов, таких как индексы, индексированные представления или схема секционирования.

Для демонстрации генерирования помощником Database Engine Tuning Advisor рекомендации по созданию физических объектов в примере 20.10 для базы данных `sample` создаются две новые таблицы: `orders` и `order_details`.

ПРИМЕЧАНИЕ

Если ваша база данных `sample` уже содержит таблицу `orders`, то, прежде чем выполнять этот пример, ее нужно удалить (для этого можно использовать инструкцию `DROP TABLE`).

Пример 20.10. Создание двух новых таблиц для базы данных `sample`

```
USE sample;
CREATE TABLE orders
   (orderid INTEGER NOT NULL,
   orderdate DATE,
   shippeddate DATE,
   freight money);
CREATE TABLE order_details
   (productid INTEGER NOT NULL,
   orderid INTEGER NOT NULL,
   unitprice money,
   quantity INTEGER);
```

Для демонстрации использования помощника Database Engine Tuning Advisor созданные таблицы должны содержать большое количество строк. В примерах 20.11—20.12 в таблицу `orders` вставляется 3000 строк, а в таблицу `order_details` — 30 000 строк соответственно.

Пример 20.11. Вставка новых строк в таблицу `orders`

```
-- Этот пакет в таблицу orders вставляет 3000 строк
USE sample;
declare @i int, @order_id integer
   declare @orderdate datetime
   declare @shipped_date datetime
   declare Sfreight money
   set @i = 1
   set @orderdate = getdate()
   set @shipped_date = getdate()
   set @freight = 100.00 while @i < 3001
   begin
   insert into orders (orderid, orderdate, shippeddate, freight)
      values(@i, @orderdate, @shipped_date, @freight)
   set @i = @i+1
   end
```

Пример 20.12. Вставка новых строк в таблицу `order_details`

```
-- Этот пакет в таблицу order_details вставляет 30 000 строк
-- и модифицирует некоторые из них
USE sample;
declare @i int, @j int
   set @i = 3000
   set @j = 10
   while @j > 0
   begin
   if @i > 0
      begin
      insert into order_details (productid, orderid, quantity)
      values (@i, @ j , 5)
      set @i = @i — 1
   end
   else begin
         set @j = @j — 1
         set @i = 3000
      end
   end
   go
   update order_details set quantity = 3
      where productid in (1511, 2678)
```

Запрос, показанный в примере 20.13, будет использован в качестве входного файла для приложения SQL Server Profiler. (Предполагается, что для столбцов в инструкции `SELECT` не создается никаких индексов.) Сначала нужно запустить приложение SQL Server Profiler. Для этого выберите последовательность команд меню **Пуск |**

Все программы | Microsoft SQL Server 2012 | Performance Tools | SQL Server Profiler. В меню **File** приложения выберите пункт **New Trace**. После соединения с сервером откроется диалоговое окно **Trace Properties**. Введите имя трассировки, а в поле **Save to File** выберите выходной *trc*-файл, в котором будет сохраняться информация для приложения SQL Server Profiler. Нажмите кнопку **Run**, чтобы начать процесс захвата, а затем выполните в среде Management Studio запрос из примера 20.13.

Пример 20.13. Запрос для трассировки

```
USE sample;
SELECT orders.orderid, orders.shippeddate
   FROM orders
   WHERE orders.orderid between 806 and 1600
    and not exists (SELECT order_details.orderid
      FROM order_details
      WHERE order_details.orderid = orders.orderid);
```

Наконец, остановите процесс захвата, выбрав в меню **File** команду **Stop Trace** и указав соответствующую трассировку.

Работа с помощником Database Engine Tuning Advisor

Помощник Database Engine Tuning Advisor анализирует рабочую нагрузку и выдает рекомендации по физической структуре одной или нескольких баз данных. Анализ содержит рекомендации по добавлению, удалению или модификации физических структур баз данных, таких как индексы, индексированные представления или секции. Помощник Database Engine Tuning Advisor рекомендует набор физических структур базы данных, которые оптимизируют задачи, входящие в рабочую нагрузку.

Чтобы начать работу с помощником Database Engine Tuning Advisor (рис. 20.2), выполните последовательность команд меню **Пуск | Все программы | Microsoft SQL Server 2012 | Performance Tools | Database Engine Tuning Advisor**. Другим способом является запуск приложения SQL Server Profiler и затем выполнение команды меню **Tools | Database Engine Tuning Advisor**.

В поле **Session name** введите имя сессии, для которой помощник Database Engine Tuning Advisor будет создавать рекомендации по настройке. В разделе **Workload** установите переключатель **File** или **Table**. Если вы выбрали переключатель **File**, то введите имя для файла трассировки. Если же вы выбрали переключатель **Table**, то нужно ввести имя таблицы, созданной приложением SQL Server Profiler. Используя приложение SQL Server Profiler, вы можете захватывать данные о каждой рабочей нагрузке и сохранять их в файл или в таблицу SQL Server.

ПРИМЕЧАНИЕ

Запуск приложения SQL Server Profiler может подвергнуть большой нагрузке занятый работой экземпляр компонента Database Engine.

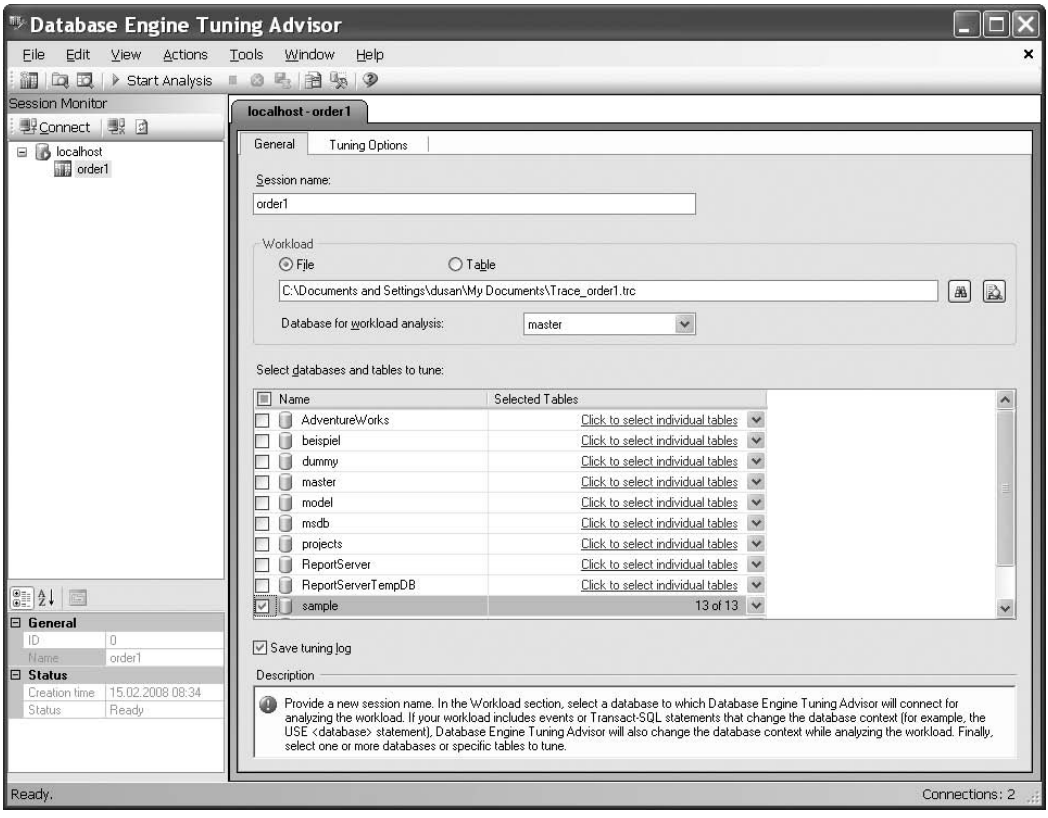


Рис. 20.2. Окно помощника Database Engine Tuning Advisor, вкладка **General**

В разделе **Select databases and tables to tune** выберите одну или несколько баз данных и/или одну или несколько таблиц, которые требуется настроить. Помощник Database Engine Tuning Advisor может выполнять настройку рабочей нагрузки, состоящей из нескольких баз данных. Это означает, что этот инструмент может рекомендовать индексы, индексированные представления и схемы секционирования для любой базы данных в рабочей нагрузке.

Для выбора опций для настройки откройте вкладку **Tuning Options** (рис. 20.3).

Большинство опций на этой вкладке разделены на три группы.

- **Physical Design Structures (PDS) to use in database** (физические структуры для применения в базе данных).

  Эти опции позволяют выбрать, какие физические структуры (индексы и/или индексированные представления) помощник Database Engine Tuning Advisor должен рекомендовать после настройки существующей рабочей нагрузки. (Установка переключателя **Evaluate utilization of existing PDS only** указывает помощнику Database Engine Tuning Advisor выполнять анализ только существующих физических структур и давать рекомендации, какие из них следует удалить.)

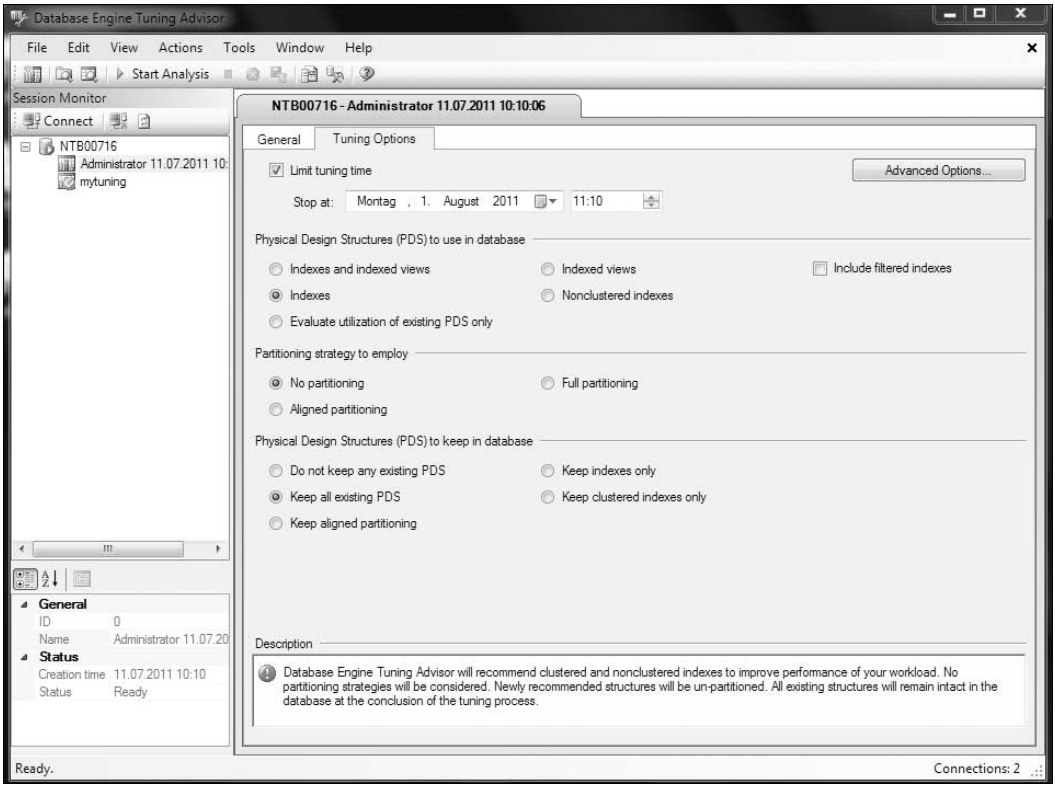


Рис. 20.3. Помощник Database Engine Tuning Advisor, вкладка **Tuning Options**

- **Partitioning strategy to employ** (стратегия секционирования для применения).

  Опции группы **Partitioning strategy to employ** позволяют выбрать, будут ли созданы рекомендации по секционированию или нет. Если выбрать рекомендации по секционированию, то можно также выбрать тип секционирования — полное или с выравниванием. (Тема секционирования подробно рассматривается в *главе 25*.)

- **Physical Design Structures (PDS) to keep in database** (физические структуры для сохранения в базе данных). Здесь предоставляется возможность выбрать структуры базы данных, которые не должны затрагиваться процессом настройки.

В случае баз данных большого объема настройка физических структур обычно требует значительного времени и ресурсов. Вместо того чтобы начинать исчерпывающий поиск возможных индексов, помощник Database Engine Tuning Advisor по умолчанию предлагает режим ограниченного использования ресурсов. Но этот режим работы также обеспечивает очень точные результаты, хотя количество настраиваемых ресурсов значительно уменьшается.

В процессе указания параметров настройки можно задать дополнительные опции настройки в диалоговом окне **Advanced Tuning Options** (рис. 20.4), которое открывается нажатием кнопки **Advanced Options**.

Рис. 20.4. Диалоговое окно **Advanced Tuning Options**

Установкой самого верхнего флажка в этом диалоговом окне можно задать максимальный объем для рекомендаций. Если планируется исчерпывающий поиск, то максимальный объем для рекомендаций следует повысить до 20 Мбайт. (В случае баз данных большого объема выбор физических структур обычно требует значительных ресурсов. Вместо того чтобы начинать поиск методом полного перебора, помощник Database Engine Tuning Advisor предлагает вам опцию ограничения пространства, используемого для настройки.)

Из всех опций для настройки индексов наибольший интерес представляет вторая опция в этом диалоговом окне, которая позволяет задать максимальное количество столбцов для каждого индекса. Одностолбцовый или составной индекс по двум столбцам можно использовать повторно для нагрузки с несколькими запросами, он требует меньше места для хранения, чем составной индекс, созданный из четырех или более столбцов. (Это применимо в том случае, когда для определенной нагрузки применяется пользовательский файл нагрузки вместо использования трассировки приложения SQL Server Profiler.) Но с другой стороны, составной индекс, созданный из четырех или более столбцов, можно использовать в качестве покрывающего индекса, чтобы разрешить доступ только к индексам для некоторых запросов в рабочей нагрузке. (Подробную информацию о покрывающих индексах см. в *главе 10*.)

После выбора требуемых опций в диалоговом окне **Advanced Tuning Options** нажмите кнопку **OK**, чтобы закрыть его. Теперь можно приступить к анализу рабочей нагрузки. Чтобы начать процесс настройки, из меню помощника выберите последовательность команд **Actions | Start Analysis**. После запуска процесса настройки для файла трассировки запроса из примера 20.13 помощник Database Engine Tuning Advisor создает рекомендации по настройке, которые можно просмотреть на вкладке **Recommendations** (рис. 20.5).

Как можно видеть, в данном случае помощник Database Engine Tuning Advisor рекомендует создать два индекса.

Рекомендации помощника Database Engine Tuning Advisor, связанные с физическими структурами, можно также просматривать, используя ряд отчетов, которые пре-

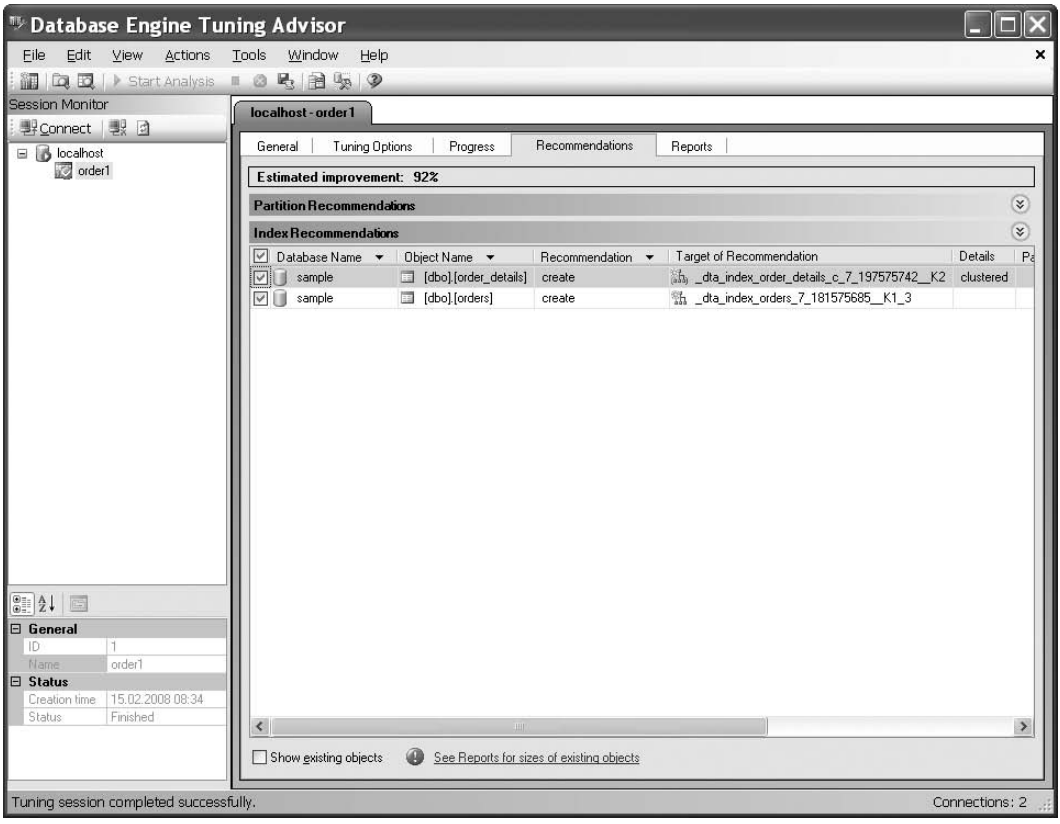


Рис. 20.5. Помощник Database Engine Tuning Advisor, вкладка **Recommendations**

доставляют информацию о некоторых представляющих значительный интерес опциях. Эти отчеты позволяют увидеть, как помощник Database Engine Tuning Advisor выполнял оценку рабочей нагрузки. Для просмотра этих отчетов после окончания процесса настройки в диалоговом окне помощника перейдите на вкладку **Reports**. Для просмотра доступны, среди прочих, следующие отчеты:

- **Index Usage Report (recommended configuration** (отчет по использованию индексов (рекомендуемая конфигурация)) — содержит информацию об ожидаемом использовании рекомендуемых индексов и их предполагаемых размерах;
- **Index Usage Report (current configuration)** (отчет по использованию индексов (текущая конфигурация)) — предоставляет ту же самую информацию, что и предшествующий отчет, но для текущей конфигурации;
- **Index Detail Report (recommended configuration)** (подробный отчет по индексам (рекомендуемая конфигурация)) — содержит информацию об именах всех рекомендуемых индексов и их типах;
- **Index Detail Report (current configuration)** (подробный отчет по индексам (текущая конфигурация)) — предоставляет ту же самую информацию, что и предшествующий отчет, но для фактической конфигурации до начала процесса настройки;

- **Table Access Report** (отчет о доступе к таблицам) — предоставляет информацию о затратах всех запросов в рабочей нагрузке (используя таблицы базы данных);
- **Workload Analysis Report** (отчет анализа рабочей нагрузки) — предоставляет информацию об относительных частотах всех инструкций по модификации данных (затраты подсчитываются относительно наиболее затратной инструкции при текущей конфигурации индексов).

Эти рекомендации можно использовать тремя способами: немедленно, по расписанию или после сохранения в файл. Чтобы применить рекомендации сейчас же, из меню **Actions** выполните команду **Apply Recommendations**, а для сохранения их в файл из этого же меню выберите команду **Save Recommendations**. (Вторая опция полезна в тех случаях, когда сценарий создается на одной, тестовой, системе, а рекомендации настройки планируется использовать на другой, рабочей, системе.) Третий пункт этого же меню **Actions** (**Evaluate Recommendations**) используется для оценки рекомендаций, выданных помощником Database Engine Tuning Advisor.

Другие средства SQL Server для настройки производительности

Сервер SQL Server предоставляет два других дополнительных инструмента, связанных с настройкой производительности:

- сборщик данных по производительности Performance Data Collector;
- регулятор ресурсов Resource Governor.

Эти инструменты рассматриваются в последующих разделах.

Сборщик Performance Data Collector

Обычно администратору базы данных очень трудно найти причину проблемы производительности, поскольку его обычно нет там, где происходит проблема и когда она происходит. Поэтому, чтобы исправить возникшую проблему, администратору сначала нужно выяснять ее происхождение.

Для оказания помощи администраторам в таких мероприятиях разработчики Microsoft предоставляют в их пользование целую инфраструктуру для сбора данных по производительности, которая называется Performance Data Collector. Этот сборщик данных устанавливается как часть экземпляра компонента Database Engine и его можно настроить для выполнения или по определенному графику, или на постоянной основе. Этот инструмент предназначен для выполнения следующих трех задач:

- сбора различных наборов данных, связанных с производительностью;
- сохранением этих данных в хранилище MDW (Management Data Warehouse — хранилище управляющих данных);

- предоставлением пользователю возможности просматривать собранные данные, используя предопределенные отчеты.

Прежде чем использовать сборщик Performance Data Collector, необходимо настроить хранилище управляющих данных (MDW). Для этого в обозревателе объектов разверните узел сервера, в нем откройте папку **Management**, щелкните правой кнопкой узел **Data Collection** и, в появившемся контекстном меню, выберите пункт **Configure Management Data Warehouse**. Откроется диалоговое окно мастера настройки хранилища MDW — **Configure Management Data Warehouse Wizard**. Этот мастер выполняет два задания: создает хранилище MDW и организует сбор данных. После выполнения этих заданий можно запускать сборщик данных Performance Data Collector и просматривать создаваемые им отчеты.

Создание хранилища MDW

Нажатие кнопки **Next** в окне приветствия мастера конфигурирования хранилища MDW открывает окно выбора задачи конфигурирования **Select Configuration Task**. В этом окне выберите опцию **Create or Upgrade a Management Data Warehouse** и нажмите кнопку **Next**. В открывшемся окне **Configure Management Data Warehouse Storage** (рис. 20.6) выберите требуемый сервер и базу данных, в которых надо разместить хранилище MDW, и после этого нажмите кнопку **Next**.

Рис. 20.6. Окно выбора сервера и базы данных для размещения хранилища MDW

В открывшемся диалоговом окне **Map Logins and Users** (рис. 20.7) присвойте для существующих имен входа и пользователей роли хранилища данных.

Рис. 20.7. Присвоение именам входа и пользователям ролей хранилища MDW

Операцию присвоения ролей необходимо выполнить явно, поскольку никто из пользователей не является членом роли хранилища MDW по умолчанию. Присвоив все требуемые роли, нажмите кнопку **Next**. В открывшемся окне завершения работы мастера проверьте правильность выполненных настроек и нажмите кнопку **Finish**.

Организация сбора данных

Настроив хранилище MDW, можно начинать сбор данных. Снова запустите мастер настройки хранилища MDW, но на этот раз в окне **Select Configuration Task** выберите опцию **Set Up Data Collection** и нажмите кнопку **Next**. В диалоговом окне **Configure Management Data Warehouse Storage** (рис. 20.8) укажите имя сервера и имя хранилища данных, созданного в предыдущем разделе, а затем укажите, где локально кэшировать собранные данные перед их отправкой в хранилище MDW.

Нажмите кнопку **Next**. В открывшемся окне завершения работы мастера нажмите кнопку **Finish**. Мастер завершит свою работу и предоставит обзор выполненных заданий.

Просмотр отчетов

После того как сборщик данных Performance Data Collector будет настроен и активирован, система начинает сбор информации о производительности и сохраняет ее в хранилище MDW. Кроме этого, будут созданы три новых отчета для просмотра собранных данных: Server Activity History, Disk Usage Summary и Query Statistics

Рис. 20.8. Диалоговое окно **Configure Management Data Warehouse Storage**

History. Для просмотра этих отчетов разверните узел сервера, в нем разверните папку **Management**, в этой папке щелкните правой кнопкой узел **Data Collection**. В появившемся контекстном меню выберите пункт **Reports**, затем выберите для просмотра один из этих трех отчетов.

Первый отчет, Server Activity History, содержит статистику производительности для системных ресурсов, рассмотренных в этой главе. Второй отчет, Disk Usage Summary, содержит начальный размер и среднее ежедневное увеличение файлов данных и журнала. Последний отчет, Query Statistics History, содержит статистические данные по выполнению запросов.

Регулятор ресурсов Resource Governor

Одна из самых трудных задач в плане настройки производительности состоит в попытке управления ресурсами при конкурирующих за эти ресурсы рабочих нагрузках на разделяемом сервере баз данных. Эту задачу можно решить посредством использования или виртуализации сервера, или нескольких экземпляров сервера. В обоих случаях для экземпляра не существует возможности для определения, используют ли другие экземпляры память и центральный процессор (или виртуальные машины). Регулятор ресурсов Resource Governor управляет такой ситуацией, позволяя одному экземпляру зарезервировать часть системных ресурсов для определенного процесса.

Обычно регулятор Resource Governor позволяет администратору баз данных определить пределы ресурсов и приоритеты для разных рабочих нагрузок. Таким образом можно добиться согласованности производительности для процессов.

Регулятор Resource Governor имеет два главных компонента:

- группы рабочих нагрузок;
- пулы ресурсов.

Когда процесс подключается к Database Engine, ему присваивается определенная классификация, на основе которой он помещается в одну из групп рабочих нагрузок. (Классификация выполняется посредством или встроенного классификатора или функции, определенной пользователем. Затем одна или несколько групп рабочих нагрузок назначаются конкретным пулам ресурсов (рис. 20.9).

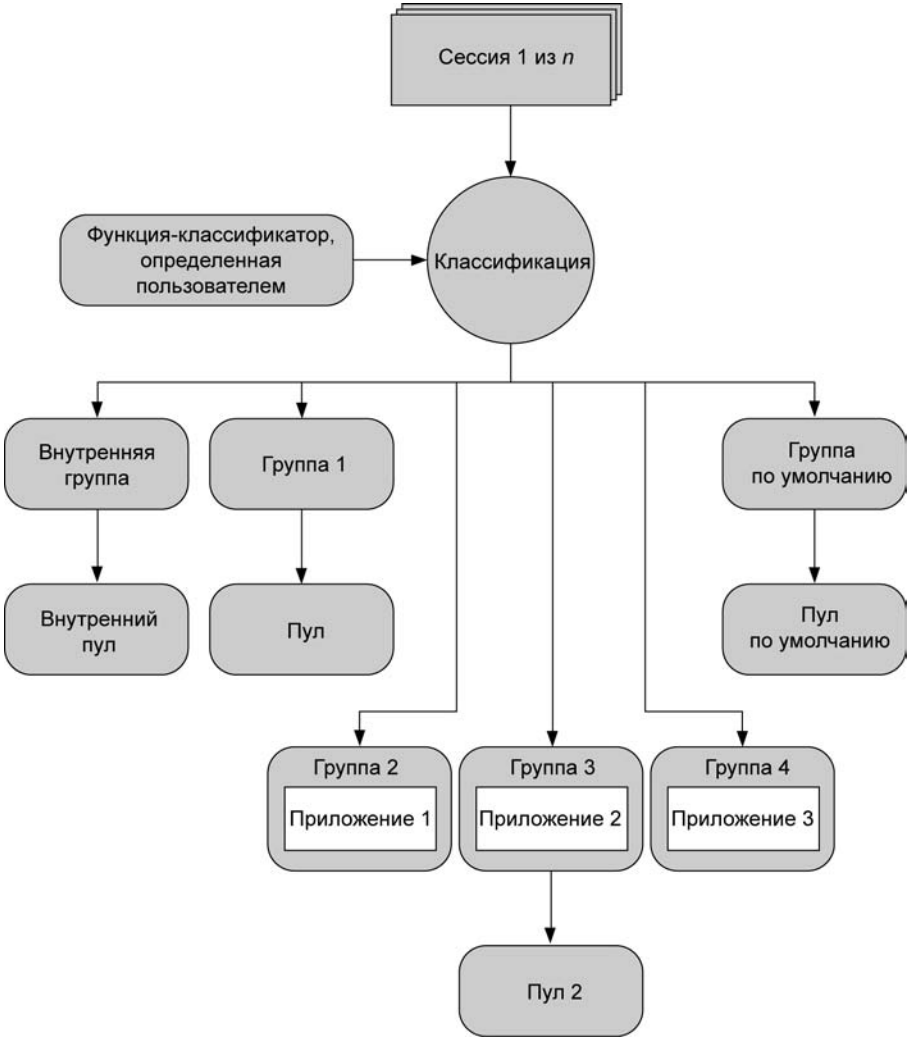


Рис. 20.9. Архитектура регулятора ресурсов Resource Governor

ПРИМЕЧАНИЕ

Регулятор Resource Governor может управлять только центральным процессором и памятью. Иными словами, он не поддерживает ресурсы ввода/вывода и сетевого окружения.

Как можно видеть на рис. 20.9, существуют две разные группы ресурсов:

- внутренняя группа;
- группа по умолчанию.

Внутренняя группа применяется для выполнения определенных системных функций, а группа по умолчанию используется для процессов, которые не имеют определенной классификации. (Классификацию для внутренней группы изменять нельзя, но можно выполнять мониторинг ее нагрузки.)

ПРИМЕЧАНИЕ

Внутренняя группа и группа по умолчанию являются предопределенными группами рабочих нагрузок. Кроме этих групп это средство позволяет задать 18 дополнительных (определяемых пользователем) групп рабочих нагрузок.

Пул ресурсов представляет выделение системных ресурсов компонента Database Engine. Каждый пул ресурсов имеет две разные части, которые задают минимальную и максимальную резервацию ресурсов. Поскольку минимальные назначения ресурсов всех пулов не могут пресекаться, их общий объем не может превышать 100% всех ресурсов системы. С другой стороны, максимальное значение пула ресурсов может быть в диапазоне от минимального значения и до 100%.

Аналогично группам рабочих нагрузок существует два предопределенных пула ресурсов: внутренний пул и пул по умолчанию. Внутренний пул содержит системные ресурсы, которые используются внутренними процессами системы. Пул по умолчанию содержит как группу нагрузок по умолчанию, так и определяемые пользователем группы.

Создание групп нагрузок и пулов ресурсов

Группы нагрузок и пулы ресурсов создаются в три шага:

1. Создаются пулы ресурсов.
2. Создаются группы рабочих нагрузок, которые назначаются пулам ресурсов.
3. Для каждой группы рабочих нагрузок определяется и регистрируется соответствующая классифицирующая функция.

ПРИМЕЧАНИЕ

Управлять регулятором ресурсов Resource Governor можно посредством среды Management Studio или с помощью инструкций Transact-SQL. В этом разделе описывается создание групп и пулов, используя среду Management Studio. Соответствующие инструкции Transact-SQL упоминаются без каких-либо объяснений.

Прежде чем создавать новый пул ресурсов, необходимо включить регулятор ресурсов Resource Governor. Для этого в обозревателе объектов разверните узел сервера, в нем разверните папку **Management**, щелкните правой кнопкой узел **Resource Governor** и в контекстном меню выберите пункт **Enable**. (Альтернативно можно использовать инструкцию Transact-SQL `ALTER RESOURCE GOVERNOR`.)

ПРИМЕЧАНИЕ

Пулы ресурсов и группы нагрузок можно создать в одном диалоговом окне, как описывается далее.

Чтобы создать новый пул ресурсов, разверните последовательно экземпляр сервера, затем папку **Management** и узел **Resource Governor**, щелкните правой кнопкой папку **Resource Pools** и в контекстном меню выберите пункт **New Resource Pool**. В результате откроется диалоговое окно **Resource Governor Properties** (рис. 20.10).

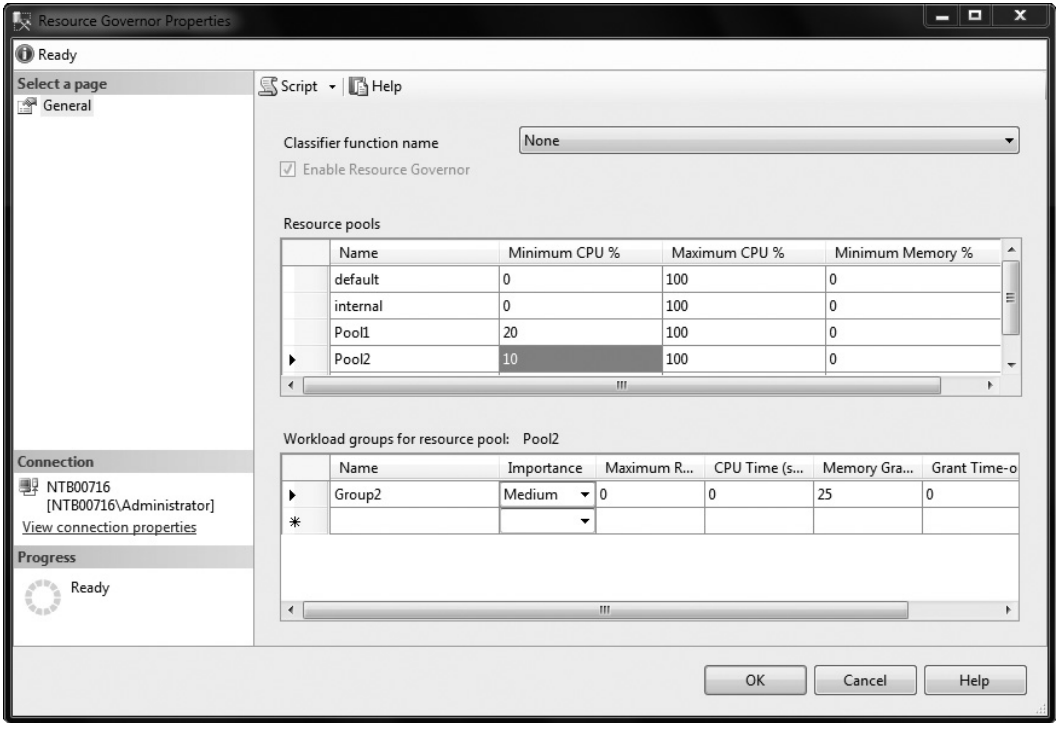


Рис. 20.10. Диалоговое окно **Resource Governor Properties**

Обычно при создании нового пула ресурсов требуется указать его имя и минимальные и максимальные значения для выделенных пределов ресурсов центрального процессора и памяти. Поэтому в разделе **Resource pools** диалогового окна свойств регулятора ресурсов выберите двойным щелчком первый столбец первой пустой строки и введите имя нового пула ресурсов, а так же укажите для него минимальные и максимальные значения использования центрального процессора и памяти.

В этом же диалоговом окне (рис. 20.10) можно задать и соответствующую группу рабочих нагрузок.

Для этого в разделе **Workload groups for resource pool** выберите двойным щелчком пустую ячейку **Name** и введите в нее имя соответствующей группы. Факультативно, для этой группы рабочих нагрузок можно указать несколько разных свойств. Выполнив все настройки, чтобы закрыть диалоговое окно, нажмите кнопку **OK**.

ПРИМЕЧАНИЕ

Для создания пула ресурсов посредством Transact-SQL используется инструкция `CREATE RESOURCE POOL`, а инструкция `CREATE WORRKLOAD GROUP` создает новую группу нагрузок.

После создания нового пула ресурсов и соответствующей группы нагрузок нужно создать функцию классификации. Это определяемая пользователем функция, посредством которой создается связь между группой рабочих нагрузок и пользователями. (Пример такой функции смотрите в описании инструкции `ALTER RESOURCE GOVERNOR` в *электронной документации*.)

Мониторинг конфигурации регулятора Resource Governor

Для мониторинга групп рабочих нагрузок и пулов ресурсов можно использовать следующие два динамические административные представления:

- `sys.dm_resource_governor_workload_groups`;
- `sys.dm_resource_governor_resource_pools`.

Динамическое представление `sys.dm_resource_governor_workload_groups` содержит информацию о группах рабочих нагрузок. Столбец `total_query_optimization_count` этого представления содержит общее число оптимизаций запросов в данной группе рабочей нагрузки. Слишком большое значение этого столбца может указывать проблемы с памятью.

Представление `sys.dm_resource_governor_resource_pools` содержит информацию о пулах ресурсов. В столбцах `total_cpu_usage_ms` этого представления указывается совокупное использование центрального процессора, а в столбце `used_memory_kb` — текущий объем используемой памяти, указывая, таким образом, потребление пулами этих двух системных ресурсов.

Резюме

Вопросы производительности можно разделить на упреждающие и реактивные категории действий. Упреждающие категории касаются всех видов активностей, которые влияют на производительность всей системы и которые будут влиять на будущие системы организации. К упреждающей категории относятся правильное проектирование базы данных и правильный выбор формы инструкций Transact-SQL в прикладных программах. Реагирующие вопросы производительности связа-

ны с мерами, которые предпринимаются после возникновения узкого места в производительности. SQL Server предоставляет разнообразные инструменты (графические средства, инструкции языка Transact-SQL и хранимые процедуры) для обзора и трассировки проблем производительности системы SQL Server.

Из всех этих инструментов наилучшими для мониторинга производительности являются системный монитор Performance Monitor и динамические административные представления, поскольку они позволяют отслеживать, отображать и выполнять трассировку узких мест в производительности и создавать соответствующие отчеты.

Со следующей главы начинается *часть IV* этой книги, в которой рассматриваются службы Analysis Services. В ней представляются общие термины и концепции, которые следует знать по этой важной теме.

Упражнения

Упражнение 20.1

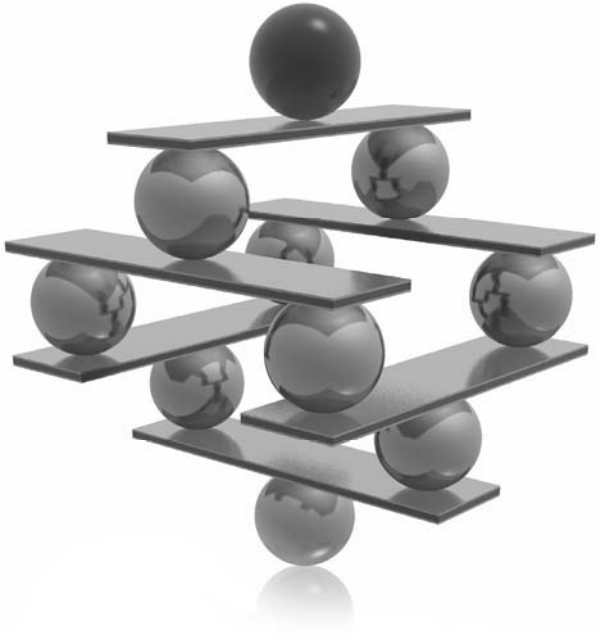
Опишите различия между приложением SQL Server Profiler и помощником Database Engine Tuning Advisor.

Упражнение 20.2

Какие имеются различия между сборщиком данных Performance Data Collector и регулятором ресурсов Resource Governor?

Часть IV

SQL Server и бизнес-аналитика



Глава 21

Введение в бизнес-аналитику

- **Оперативная обработка транзакций по сравнению с бизнес-аналитикой**
- **Хранилища данных и киоски данных**
- **Проектирование хранилищ данных**
- **Кубы и их архитектура**
- **Доступ к данным**

Целью этой главы является ознакомление с важной областью технологии баз данных: *бизнес-аналитикой* (business intelligence — BI). В первой части этой главы объясняются различия между *оперативной обработкой транзакций* (online transaction processing — OLTP), с одной стороны, и бизнес-аналитикой — с другой. *Складом данных* (data store) для процесса бизнес-аналитики может быть либо *хранилище данных* (data warehouse), либо *киоск данных* (data mart). Обсуждаются оба типа складов данных, а их различия перечисляются во второй части главы. Проектирование данных в бизнес-аналитике и необходимость создания агрегатных таблиц рассматриваются в конце этой главы.

Оперативная обработка транзакций в сравнении с бизнес-аналитикой

С самого начала своего существования реляционные системы баз данных использовались почти исключительно для хранения основных деловых данных, таких как заказы и счета, используя для этого обработку на основе транзакций. Такое акцентирование на деловых данных имеет свои преимущества и недостатки. Одним из преимуществ является то, что по сравнению с ранними системами баз данных уровень производительности современных систем баз данных значительно повысился, и сегодня многие из них могут выполнять тысячи транзакций в секунду (при условии использования соответствующего аппаратного обеспечения). Но с другой сто-

роны, фокусирование внимания на обработке транзакций мешало разработчикам баз данных видеть другое естественное их применение: для фильтрации и анализа требуемой информации изо всех существующих данных на уровне предприятия или отдела.

Оперативная обработка транзакций

Как уже упоминалось, производительность является одной из основных проблем систем, которые базируются на обработке транзакций. Эти системы называются *системами оперативной обработки транзакций* или *OLTP-системами*. Типичным примером операции, выполняемой OLTP-системой, будет снятие денег с банковского счета через банкомат. Важными особенностями OLTP-систем являются следующие:

- обработка коротких транзакций, идущих большим потоком;
- большое количество (возможно, сотни или тысячи) пользователей;
- постоянное выполнение операций чтения и записи с небольшими количествами строк;
- данные среднего размера, которые хранятся в базе данных.

Уровень производительности системы баз данных повышается при условии коротких транзакций. Это объясняется тем, что для предотвращения возможных отрицательных последствий одновременного доступа к данным транзакции используют блокировки (см. *главу 13*). Если транзакции занимают много времени, количество блокировок и их длительность возрастает, понижая уровень доступности данных для других транзакций и, следственно, снижая их производительность.

Большие OLTP-системы обычно имеют большое количество пользователей, работающих в системе одновременно. Типичным примером OLTP-системы будет система резервирования авиакомпании, которая должна почти немедленно одновременно обрабатывать тысячи запросов по продаже билетов на авиарейсы в одной стране или во всем мире. В такой системе большинство пользователей ожидают, что их требование к времени отклика будет выполнено системой и система будет доступной в течение рабочего времени (или 24 часа в день, семь дней в неделю).

Пользователи OLTP-систем непрерывно выполняют свои DML-инструкции, т. е. они одновременно и постоянно используют операции чтения и записи. Поскольку данные OLTP-систем постоянно изменяются, то эти данные очень динамичны. Все операции (или результаты операций) в базе данных обычно выполняются с небольшим объемом данных, хотя не исключено, что система базы данных должна иметь доступ ко многим строкам одной или нескольких таблиц, хранящихся в базе данных.

В последние годы объем данных, хранящихся в *оперативной базе данных* (т. е. базе данных под управлением OLTP-системы), постоянно увеличивается. В настоящее время существуют многие базы данных, в которых хранится несколько гигабайт или даже десятки гигабайт данных. Но как вы увидите далее, это сравнительно небольшие объемы данных по сравнению с объемами хранилищ данных.

Системы бизнес-аналитики

Бизнес аналитика (business intelligence (BI)) — это процесс интегрирования данных всего предприятия в один склад данных, к которому конечные пользователи могут осуществлять нерегламентированные запросы для анализа этих данных и создания отчетов. Иными словами, цель бизнес-аналитики состоит в хранении данных, доступных пользователям для принятия деловых решений на основе их анализа. Эти системы часто называются *аналитическими* или *информационными*, потому что при обращении к этим данным пользователи получают необходимую информацию для принятия наилучших решений в области бизнеса.

Цели систем бизнес-аналитики отличаются от целей OLTP-систем. Типичным примером запроса в системе бизнес-аналитики будет такой: "Для какой категории продуктов был самый высокий уровень продаж для каждого региона в третьем квартале 2011 года?" Поэтому свойства системы бизнес-аналитики существенно отличаются от свойств OLTP-системы, перечисленных в предшествующем разделе. Наиболее важными свойствами системы бизнес-аналитики являются следующие:

- периодические операции записи (загрузка данных) в результате запросов, охватывающих громадное количество строк;
- небольшое количество пользователей;
- большой объем данных, хранящихся в базе данных.

Помимо регулярной загрузки данных (обычно выполняющейся ежедневно), системы бизнес-аналитики в основном работают в режиме только чтения. Поэтому данные в такой системе являются статическими. Как будет подробно рассмотрено далее в этой главе, данные собираются из различных источников, очищаются (приводятся в согласованное состояние) и загружаются в базу данных, называемую *хранилищем данных* (или *киоском данных*). Очищенные данные обычно не подвергаются изменениям, т. е. запросы пользователей обращаются к этим данным с помощью инструкции `SELECT` для получения необходимой им информации, а операции модификации данных выполняются очень редко.

Поскольку системы бизнес-аналитики применяются для получения информации, по сравнению с OLTP-системами, количество пользователей, одновременно использующих такую систему, обычно относительно небольшое. Пользователи системы бизнес-аналитики, как правило, создают отчеты, отображающие различные аспекты, связанные с финансами предприятия или отдела, или же они выполняют сложные запросы для сравнения данных.

ПРИМЕЧАНИЕ

Другим отличием OLTP-систем и систем бизнес-аналитики, которое даже влияет на поведение пользователей, является расписание их работы, т. е. когда эти системы доступны пользователям для работы с ними. Тогда как OLTP-система может быть доступной непрерывно (если спроектирована для такого режима работы), система бизнес-аналитики доступна только после очистки данных и их загрузки в базу данных.

В отличие от систем баз данных OLTP-системы, в которых хранятся только текущие данные, системы бизнес-аналитики должны также отслеживать историю данных. Не забывайте, что системы бизнес-аналитики сравнивают данные, собранные в разные периоды времени. По этой причине хранилища данных содержат очень большие объемы данных.

Хранилища данных и киоски данных

Хранилище данных представляет собой базу данных, содержащую все данные корпорации, к которым пользователи могут иметь единообразный доступ. Это краткое определение, а объяснить идею хранилища данных намного более сложно. Предприятие обычно имеет большой объем данных, полученных в разное время и хранящихся в разных базах данных под управлением различных систем управления базами данных (СУБД). Эти СУБД не обязательно должны быть реляционными, и некоторые предприятия до сих пор используют иерархические или сетевые базы данных. Особая команда специалистов по разработке программного обеспечения исследует исходные базы данных (и файлы данных) и преобразовывает их в целевую базу данных: хранилище данных. Но так как данные в хранилище данных содержат критически важную информацию для работы предприятия, их еще нужно консолидировать. *Консолидация данных* означает, что все эквивалентные запросы, выполняемые к хранилищу данных в разное время, должны возвращать одинаковые результаты. Консолидация данных в хранилище данных осуществляется в следующие три этапа:

- сбор данных из различных источников (также называется извлечением данных);
- очистка данных (иными словами, процесс преобразования);
- обеспечение качества данных.

Данные должны быть тщательно собраны из различных источников. В процессе сборки данные извлекаются из их источников, преобразовываются в промежуточную схему и помещаются во временную рабочую область. Для извлечения данных требуются инструменты, которые извлекают только те данные, которые нужно поместить в хранилище данных.

Процесс очистки обеспечивает достоверность данных, предназначенных для помещения в целевую базу данных. В частности, в процессе очистки проверяется соответствие значений их полям (например, адресов) или использование совместимых типов данных для одних и тех же полей в различных источниках. Для реализации очистки данных требуется специальное программное обеспечение. Более ясно объяснить процесс очистки данных поможет следующий пример. Предположим, что имеется два источника данных, содержащих информацию о сотрудниках, и что в каждой из этих баз данных есть атрибут `Gender` (пол). В первой базе данных тип этого атрибута определен как `char(6)`, а его значения — как "female" (женский) и "male" (мужской). В другой базе данных этот же атрибут имеет тип `char(1)` и соответствующие значения "f" и "m". Значения в обоих источниках данных являются

корректными, но для помещения этих данных в целевую базу данных их нужно очистить, т. е. обеспечить единообразный формат значений атрибута.

Последний этап консолидации данных — обеспечение качества данных — включает собой процесс проверки данных, при котором определяются данные, к которым конечный пользователь должен получить доступ. По этой причине конечные пользователи должны принимать непосредственное участие в этом процессе. Когда процесс консолидации данных завершается, данные загружаются в хранилище данных.

ПРИМЕЧАНИЕ

Весь процесс консолидации данных называется *ETL* (extraction, transformation, loading — извлечение, преобразование, загрузка). Для поддержки пользователей в процессе ETL компания Microsoft предоставляет средство *SQL Server Integration Services* (SSIS).

По своей природе (будучи хранилищем для всех данных предприятия) хранилище данных содержит громадный объем данных. (Некоторые хранилища данных содержат десятки терабайт и даже петабайт[1] данных.) Кроме этого, поскольку эти данные должны охватывать все предприятие, реализация хранилищ данных обычно занимает много времени, объем которого зависит от размера предприятия. Вследствие этих недостатков, многие компании начинают с меньших решений, которые называются киоском данных.

Киоск данных (data mart) представляет собой склад данных, содержащий все данные на уровне отдела, что позволяет пользователям иметь доступ к данным, затрагивающим только отдельную часть их организации. Например, отдел сбыта хранит все свои данные по сбыту в своем собственном киоске данных, исследовательский отдел хранит данные по исследованиям в своем киоске данных и т. п. Таким образом, киоск данных предоставляет несколько следующих преимуществ перед хранилищем данных:

- более узкая область применения;
- более короткое время разработки и более низкая стоимость;
- более легкое обслуживание данных;
- разработка снизу вверх.

Как уже упоминалось, киоск данных содержит только ту информацию, которая требуется одной части организации, обычно одному отделу. Поэтому подготовить данные для конечного пользователя, предназначенные для использования такой небольшой организационной единицей, можно с большей легкостью.

Разработка хранилища данных занимает в среднем два года и стоит 5 млн долларов. С другой стороны, стоимость разработки киоска данных в среднем составляет 200 тыс. долларов, а разработка занимает от трех до пяти месяцев. По этим причи-

[1] 1 Пбайт (петабайт) равен 1024 Тбайт (терабайт). — *Ред.*

нам предпочтительнее разрабатывать киоск данных, особенно если это первый проект бизнес-аналитики в вашей организации.

То обстоятельство, что киоск данных содержит значительно меньшие объемы данных, чем хранилище данных, помогает сократить количество и упростить сложность задач, таких как извлечение, очистка и обеспечение качества данных. Кроме этого, спроектировать решения для отдела проще, чем для всей организации. (Дополнительную информацию по бизнес-аналитике и размерной модели *см. в следующем разделе этой главы*.)

Если вы проектируете и разрабатываете несколько киосков данных в вашей организации, то следует иметь в виду, что их все можно позже объединить в одно хранилище данных. Этот процесс проектирования снизу вверх имеет несколько преимуществ перед проектированием всего хранилища данных с самого начала. Во-первых, каждый киоск данных может содержать идентичные целевые таблицы, которые можно объединить в соответствующем хранилище данных. Во-вторых, некоторые задачи логически являются задачами на уровне предприятия, например, такие задачи, как сбор финансовой информации для бухгалтерии. Если существующие киоски данных предстоит объединить, чтобы создать хранилище данных на уровне предприятия, то потребуется наличие глобального репозитория (т. е. каталог данных, который содержит информацию обо всех данных, хранящихся как в исходных, так и целевых базах данных).

ПРИМЕЧАНИЕ

Имейте в виду, что задача создания хранилища данных путем соединения нескольких киосков данных может быть очень сложной и трудоемкой, по причине возможных значительных различий в структуре и проектировании соединяемых киосков данных. Разные подразделения предприятия могут использовать различные модели данных и различные инструкции для представления данных. По этой причине в начале этого процесса проектирования настоятельно рекомендуется создать единое представление всех данных, которые будут действительными на уровне предприятия. Нельзя позволять различным подразделениям проектировать свои данные отдельно.

Проектирование хранилищ данных

Только хорошо спланированная и спроектированная база данных позволит получить хорошую производительность. Реляционные базы данных и хранилища данных отличаются друг от друга во многих отношениях, которые требуют разных методов проектирования. Реляционные базы данных проектируются с помощью стандартной модели "сущность — отношение" (entity-relationship — ER), тогда как для проектирования хранилищ и киосков данных применяется размерная модель.

В реляционных базах данных избыточность данных устраняется посредством нормальных форм (см. *главу 1*). В процессе нормализации каждая таблица базы данных, содержащая избыточные данные, разделяется на две отдельные таблицы. Этот процесс повторяется до тех пор, пока все таблицы базы данных не будут содержать неизбыточные данные.

Тщательно нормализованные таблицы удобны для оперативной обработки транзакций (OLTP), поскольку позволяют свести длину и сложность всех транзакций к минимуму. С другой стороны, процессы бизнес-аналитики основаны на запросах, которые работают с громадными объемами данных, и не являются ни короткими, ни простыми. Поэтому тщательно нормализованные таблицы не подходят для проектирования хранилищ данных, поскольку цель систем бизнес-аналитики существенно другая: в них используется небольшое количество одновременно выполняющихся транзакций, и каждая транзакция обращается к очень большому количеству записей. (Представьте себе хранилище данных громадного объема, в котором данные хранятся в сотнях таблиц. Для выборки данных из такого хранилища большинство запросов будут соединять вместе десятки больших таблиц. Такие запросы не могут хорошо выполняться, даже при использовании параллельных процессоров и системы базы данных с самым лучшим оптимизатором запросов.)

Хранилища данных не могут использовать модель "сущность — отношение", потому что эта модель предназначена для разработки баз данных с неизбыточными данными. Для проектирования хранилищ данных используется другая логическая модель, которая называется *размерной* (dimensional model) или *пространственной моделью*.

ПРИМЕЧАНИЕ

Существует и другая важная причина, по которой модель "сущность — отношение" (ER) не подходит для проектирования хранилищ данных: использование данных хранилища не структурировано. Это означает, что запросы частично выполняются специальным образом, позволяя пользователям анализировать данные совершенно разными способами. (С другой стороны, OLTP-системы обычно имеют жестко запрограммированные приложения, и поэтому содержат запросы, которые не подвергаются частым изменениям.)

В размерном моделировании каждая конкретная модель состоит из одной таблицы, в которой хранятся факты, и нескольких других таблиц, в которых описываются измерения. Первая таблица называется *таблицей фактов* (fact table), а вторая — *таблицей измерения* (dimension table). В качестве примера данных, хранящихся в таблице фактов, можно назвать, среди прочих, продажи товара и расходы. Таблицы измерений обычно содержат такие данные, как время, учетная запись, товар и сведения о сотрудниках. Пример размерной модели хранилища данных показан на рис. 21.1.

Каждая таблица измерения обычно имеет простой первичный ключ и несколько других атрибутов, которые дают точное описание этого измерения. С другой стороны, таблица фактов имеет составной первичный ключ, состоящий из первичных ключей всех таблиц измерений (см. рис. 21.1). По этой причине первичный ключ таблицы фактов состоит из нескольких внешних ключей. (Количество измерений также задает количество внешних ключей таблицы фактов.) Как можно видеть на рис. 21.1, таблицы в размерной модели создают звездообразную структуру, вследствие чего эта модель часто называется *схемой типа "звезда"*.

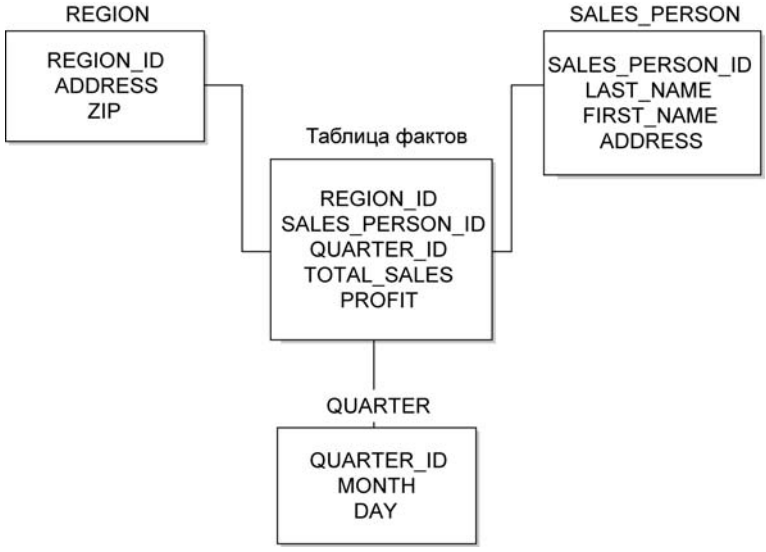


Рис. 21.1. Пример размерной модели: схема "звезда"

Другим отличием в природе данных в таблице фактов и соответствующих таблицах измерений является то, что большинство неключевых столбцов в таблице фактов являются числовыми и аддитивными, потому что такие данные могут быть использованы для выполнения необходимых вычислений. (Вспомните, что типичный запрос к хранилищу данных обрабатывает тысячи и даже миллионы строк за один раз, и единственно полезной операцией по отношению к такому громадному количеству строк будет применение агрегатной функции, такой как суммирование или нахождение максимального или среднего значения.) В качестве примера типичных столбцов таблицы фактов можно назвать такие столбцы, как `Units_of_product_sold` (количество единиц проданной продукции), `Total_sales` (общие продажи), `Profit` (прибыль) или `Dollars_cost` (цена в долларах). Числовые столбцы таблицы фактов, которые не составляют первичный ключ таблицы, называются *мерами* (measure).

С другой стороны, столбцы таблиц измерений имеют строчный тип данных и содержат текстовые описания соответствующего измерения. В качестве примера таких столбцов можно назвать столбцы `Address` (адрес), `Location` (размещение) и `Name` (имя), которые часто появляются в таблицах измерения. Имена этих столбцов обычно используются в качестве заголовков в отчетах. Другим последствием текстовой природы столбцов таблиц измерений и их использования в запросах является то, что каждая таблица измерения содержит намного больше индексов, чем соответствующая таблица фактов. (Таблица фактов обычно имеет только один однозначный индекс, составленный изо всех столбцов, создающих первичный ключ этой таблицы.) Краткое описание различий между таблицей фактов и таблицей измерений приведено в табл. 21.1.

Таблица 21.1. *Различия между таблицей фактов и таблицей измерений*

Таблица фактов	Таблица измерений
Обычно одна таблица в размерной модели	Много таблиц (12—20)
Содержит большинство строк хранилища данных	Содержит сравнительно небольшой объем данных
Имеет составной первичный ключ (состоящий изо всех первичных ключей таблиц измерений)	Первичный ключ таблицы состоит из одного столбца таблицы
Неключевые столбцы имеют числовой тип данных и являются аддитивными	Столбцы описательной природы и, поэтому, содержат строковые данные

ПРИМЕЧАНИЕ

Иногда бывает в хранилище данных необходимо иметь несколько таблиц фактов. Такая потребность возникает при наличии нескольких наборов мер, каждый из которых должен быть привязан к другой таблице фактов.

Столбцы таблиц измерений обычно весьма *денормализованы*, что означает, что многие столбцы зависят друг от друга. Денормализованная структура таблиц измерений служит одной важной цели: все имена столбцов такой таблицы используются в качестве заголовков столбцов в отчетах. Если денормализация данных в таблице измерения не является желательной, такую таблицу можно разложить на несколько подтаблиц. Такая необходимость обычно возникает, когда столбцы таблицы измерений составляют иерархии. (Например, измерение `product` может иметь такие столбцы, как `Product_id`, `Category_id` и `Subcategory_id`, которые создают три иерархических уровня с первичным ключом `Product_id` в качестве корня.) Такая структура, в которой каждый уровень базовой сущности представлен своей собственной таблицей, называется *схемой типа "снежинка"*. На рис. 21.2 показана схема измерения `product` типа "снежинка".

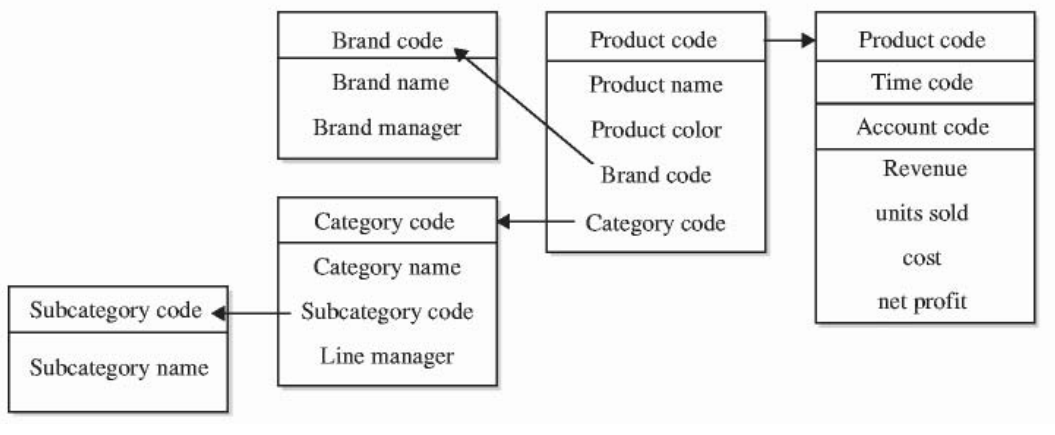


Рис. 21.2. Схема измерения `product` типа "снежинка"

Расширение схемы типа "звезда" в соответствующую схему типа "снежинка" имеет определенные преимущества (например, уменьшение используемого дискового пространства) и один основной недостаток: для получения информации из таблиц поиска схема типа "снежинка" требует большее количество операций соединения, что отрицательно сказывается на производительности. По этой причине запросы на основе схемы типа "снежинка" обычно имеют низкий уровень производительности. Поэтому проектирование с использованием схемы типа "снежинка" рекомендуется крайне редко и только в специализированных случаях.

Кубы и их архитектура

Системы бизнес-аналитики поддерживают различные типы хранилищ данных. Некоторые из этих типов хранилищ данных основаны на многомерной базе данных, которая также называется кубом. *Куб* — это подмножество данных из хранилища данных, которое можно организовать в многомерные структуры. Чтобы определить куб, сначала из схемы измерений выбирается таблица фактов и в ней определяются числовые столбцы (меры), представляющие интерес. Затем выбираются таблицы измерений, предоставляющие описания для набора данных, подлежащих анализу. Для демонстрации, рассмотрим определение куба для анализа продаж легковых автомобилей. Например, таблица фактов может содержать меры `Cars_sold`, `Total_sales` и `Costs`, а таблицы `Models`, `Quarters` и `Regions` определяют таблицы измерения. Все эти три измерения показаны в кубе на рис. 21.3.

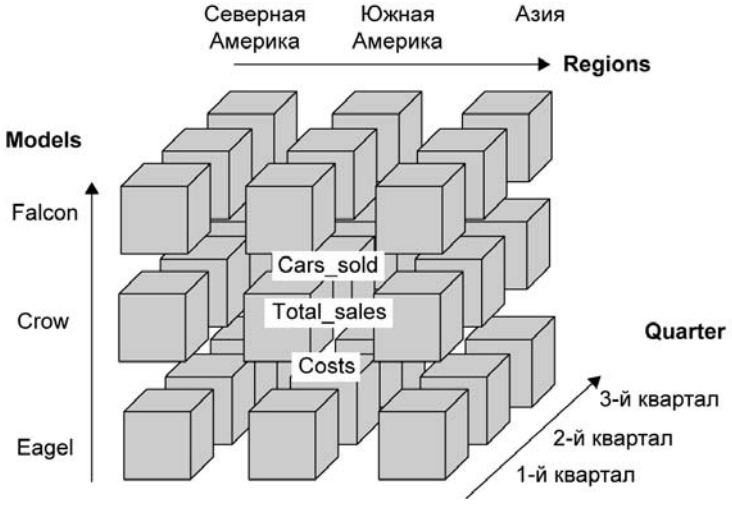


Рис. 21.3. Куб с измерениями `Models`, `Quarters` и `Regions`

Каждое измерение содержит дискретные значения, называемые *членами*. Например, измерение `Regions` может содержать следующие члены: `ALL`, `North America` (Северная Америка), `South America` (Южная Америка) и `Asia` (Азия). (Член `ALL` определяет сумму всех членов измерения.)

Кроме этого, каждое измерение куба может содержать иерархию уровней, которые позволяют пользователям задавать более подробные вопросы. Например, измерение `Regions` может содержать следующие иерархии уровней: `Country`, `Province` и `City`. Подобным образом, измерение `Quarters` может содержать иерархии уровней `Month`, `Week` и `Day`.

ПРИМЕЧАНИЕ

Кубы и многомерные базы данных управляются специальными системами, называемыми *многомерными системами управления базами данных* (МСУБД). МСУБД сервера SQL Server называется *Analysis Services*.

Физическое хранение куба описывается после рассмотрения агрегирования.

Агрегирование

В таблице фактов данные сохраняются в максимально подробной форме, чтобы их могли пользовать соответствующие отчеты. С другой стороны, как упоминалось ранее, типичный запрос к таблице фактов обрабатывает тысячи и даже миллионы строк за один раз, и единственной полезной операцией для такого громадного количества строк будет применение агрегатной функции, такой как суммирование или нахождение максимального или среднего значения. Такое различное использование данных может снизить уровень производительности нерегламентированных запросов, если они выполняются над данными низкого уровня (атомарными данными), потому что для выполнения каждой агрегатной функции потребуется трудоемкие и ресурсоемкие вычисления.

По этой причине, низкоуровневые данные в каждой таблице нужно заранее сложить и сохранить результат в промежуточных таблицах. По причине содержания "агрегатной" информации, такие таблицы называются *агрегатными таблицами* (aggregate tables), а весь процесс — *агрегированием* (aggregation).

ПРИМЕЧАНИЕ

Агрегатная строка из таблицы фактов всегда связана с одной или больше агрегатных строк таблицы измерений. Например, размерная модель на рис. 21.1 могла бы содержать такие агрегатные строки: сводные месячные продажи продавцами по регионам и сводные продажи продавцами по регионам за день.

Рассмотрим на примере, как следует агрегировать низкоуровневые данные. Конечному пользователю может потребоваться выполнить нерегламентированный запрос для отображения общего объема продаж организации за последний месяц. Для выполнения этого запроса серверу потребовалось бы суммировать все продажи за каждый день последнего месяца. Предположим, что в каждом из 500 магазинов организации каждый день выполнялось в среднем 500 транзакций продаж, и что данные хранятся на уровне транзакций. Тогда, чтобы возвратить требуемую информацию, этому запросу потребовалось бы прочитать 7 500 000 строк (500×500×30 дней)

и сложить их значения. Теперь посмотрим, что будет, если данные агрегируются в таблице, создаваемой по месячным продажам каждого магазина. Такая таблица будет иметь только 500 строк (итог за месяц для каждого из 500 магазинов), что позволит значительно повысить уровень производительности запроса.

Уровень агрегирования

Существует два экстремальных подхода к агрегированию: не выполнять агрегирования вообще и выполнять исчерпывающее агрегирование для каждой возможной комбинации запросов, которые могут потребоваться пользователям. Из предшествующего обсуждения должно быть ясно, что полное отсутствие агрегирования не подлежит рассмотрению по причине вопросов производительности. (Хранилище данных, не содержащее никаких агрегатных таблиц, скорей всего, нельзя будет вообще использовать в качестве рабочего хранилища данных.) Противоположный подход также не является приемлемым по нескольким причинам, включая следующие:

- для хранения дополнительных данных требуются громадные объемы дискового пространства;
- чрезмерно сложная поддержка агрегатных таблиц;
- исходная загрузка данных занимает слишком много времени.

Хранение дополнительных данных, которые агрегируются на каждом возможном уровне, требует дополнительных объемов дискового пространства, повышая начальные требования в шесть или больше раз (в зависимости от начального объема дискового пространства и количества запросов, которые будут требоваться пользователям). Создание таблиц для хранения агрегированных значений для всех возможных комбинаций запросов представляет непосильную задачу для системного администратора. Наконец, агрегирование при начальной загрузке данных может иметь плачевный результат, если уже сама загрузка занимает длительное время, а дополнительного времени нет.

Исходя из этого, можно сделать вывод, что к планированию и созданию агрегатных таблиц следует подходить с особой тщательностью. При определении на этапе планирования, какие данные агрегировать, следует иметь в виду следующие два главных фактора.

- Где концентрируются данные?
- Агрегирование каких данных улучшило бы производительность?

Планирование и создание агрегатных таблиц зависит от концентрации данных в столбцах основной таблицы фактов. В хранилище данных, когда в определенный день нет никакой активности, соответствующая строка не сохраняется вообще. Поэтому, если система загружает большое количество строк, по сравнению со всеми строками, которые могут быть загружены, агрегирование по этому столбцу базовой таблицы фактов чрезвычайно повышает уровень производительности. В противоположность этому, если система загружает небольшое количество строк, по сравнению со всем количеством строк, которые можно загрузить, агрегирование по этому столбцу не дает ощутимого результата.

Вот еще один пример. Из всех наименований продуктов в магазине только несколько наименований (скажем, 15%) фактически проданы за текущий день. Для размерной модели с тремя измерениями `Product`, `Store` и `Time` будет использовано только 15% комбинаций трех соответствующих первичных ключей для определенного дня и конкретного магазина. Таким образом, ежедневные данные по продаже продуктов будут *разреженными* (sparse). В противоположность этой ситуации, если в любой конкретный день были проданы все или многие названия продуктов, данные по ежедневным продажам продуктов будут *плотными* (dense).

Чтобы выяснить, какие измерения являются разреженными, а какие плотными, нужно из всех возможных комбинаций таблиц создать строки и оценить их. Обычно измерение `Time` является плотным, потому что записи делаются каждый день. Учитывая измерения `Product`, `Store` и `Time`, комбинация измерений `Store` и `Time` будет плотной, потому что для каждого дня непременно будут данные по продажам в каждом магазине. С другой стороны, комбинация измерений `Store` и `Product` будет разреженной (по ранее изложенным причинам). В данном случае, измерение `Product` обычно будет разреженным, потому что его появление в комбинации с другими измерениями является разреженным.

Выбор агрегированных значений, которые в большей мере повысили бы производительность, зависит от конечных пользователей. Поэтому, в начале работы над проектом бизнес-аналитики, следует опросить конечных пользователей, чтобы собрать информацию о том, как будут выполняться запросы по данным, сколько строк эти запросы будут извлекать и другие критерии.

Физическое хранение куба

В системах *оперативной аналитической обработки* (OLAP — online analytical processing) для хранения многомерных данных обычно используется одна из трех различных архитектур мест хранения данных:

- *реляционная*: *OLAP* (ROLAP — relational online analytical processing);
- *многомерная*: *OLAP* (MOLAP — multidimensional online analytical processing);
- *гибридная*: *OLAP* (HOLAP — hybrid online analytical processing).

В основном, разница между этими тремя архитектурами заключается в способе хранения данных на уровне листьев и наличием предварительно рассчитанных агрегированных данных. (Данные на уровне листьев являются наилучшей частицей данных, определяемых в группе мер куба. Поэтому данные на уровне листьев соответствуют данным таблицы фактов куба.)

В архитектуре ROLAP предварительно вычисленные данные не сохраняются. Вместо этого, для получения данных, требуемых для предоставления ответа на запрос, запросы обращаются к реляционной базе данных и ее таблицам. В хранилище типа MOLAP данные на уровне листьев и их агрегированные данные сохраняются в многомерном кубе.

Хотя логическое содержимое этих двух типов хранилищ идентично для одного и того же хранилища данных и аналитические инструменты ROLAP и MOLAP пред-

назначены для анализа данных посредством использования размерной модели данных, между ними есть некоторые значительные различия. Преимущества хранилищ типа ROLAP следующие:

- отсутствие дублирования данных;
- материализованные (т. е. индексированные) представления, которые можно использовать для агрегирования (суммирования).

Если данные также должны храниться в многомерной базе данных, то определенный объем данных приходится дублировать. Поэтому для хранилища типа ROLAP не требуется дополнительное пространство для копирования данных на уровне листьев. Кроме этого, в архитектуре ROLAP вычисления, относящиеся к агрегированию, можно выполнять очень быстро, если соответствующие сводные таблицы созданы, используя индексированные представления.

Но с другой стороны, архитектура MOLAP имеет несколько преимуществ над архитектурой ROLAP:

- агрегированные данные сохраняются в многомерной форме;
- время выполнения запросов обычно быстрее.

В архитектуре MOLAP многие агрегированные данные вычисляются заранее и сохраняются в многомерном кубе. Это позволяет системе не вычислять такие данные каждый раз, когда они требуются. В случае архитектуры MOLAP механизм СУБД и сама база данных обычно оптимизируются для совместной работы, поэтому скорость выполнения запросов может быть выше, чем в архитектуре ROLAP.

Хранилище архитектуры HOLAP является комбинацией хранилищ типа MOLAP и ROLAP. Заранее вычисленные данные сохраняются, как и в случае с хранилищами типа MOLAP, тогда как данные на уровне листьев остаются в реляционной базе данных. (Поэтому для запросов, использующих агрегирование, хранилище типа HOLAP идентично хранилищу типа MOLAP.) Преимущество хранилища типа HOLAP состоит в том, что данные на уровне листьев не дублируются.

Доступ к данным

Доступ к данным в хранилище данных осуществляется посредством трех основных методов:

- составления отчетов;
- оперативной аналитической обработки;
- извлечения информации из данных (data mining).

Составление отчетов является наиболее простой формой доступа к данным. *Отчет* — это всего лишь представление результата запроса в табличной форме. (Тема составления отчетов рассматривается подробно в *главе 24*.) При оперативной аналитической обработке данные исследуются интерактивно, т. е. этот способ позволяет выполнять сравнения и вычисления среди любых измерений в хранилище данных.

ПРИМЕЧАНИЕ

Язык Transact-SQL поддерживает все стандартизованные функции и конструкции, связанные с SQL/OLAP. Эта тема рассмотрена подробно в *главе 23*.

Извлечение информации из данных (data mining) применяется для исследования и анализа больших объемов данных с целью обнаружения важных закономерностей. Но обнаружение закономерностей не является единственной задачей извлечения информации из данных, поскольку с помощью этого подхода требуется преобразовать полученные данные в информацию, а информацию преобразовать в действия. Иными словами, простого анализа данных недостаточно, и результаты добычи данных еще нужно осмысленным образом применить и предпринять меры по полученным результатам. (По причине своей сложности, предмет извлечения информации из данных выходит за пределы вводного курса данной книги и поэтому в ней не рассматривается.)

Резюме

В начале работы над проектом бизнес-аналитики основным вопросом является выбор типа места для хранения данных: хранилища данных или киоска данных. Самым лучшим подходом к решению этого вопроса будет, скорей всего, начать с одного или нескольких киосков данных, которые позже можно будет объединить в хранилище данных. Большинство инструментальных средств на рынке для бизнес-аналитики поддерживают именно этот подход.

В отличие от баз данных систем для принятия решений, для проектирования которых применяется модель "сущность — отношение", для проектирования хранилищ данных лучше всего использовать размерную модель. Эти две модели значительно отличаются друг от друга. Для тех, кто уже знаком с моделью "сущность — отношение", лучшим способом изучения и использования размерной модели будет забыть все, что они знают о модели "сущность — отношение" и начать изучать размерную модель с чистого листа.

После этого вводного ознакомления с общими аспектами процесса бизнес-аналитики в следующей главе рассматривается серверная часть служб Microsoft Analysis Services.

Упражнения

Упражнение 21.1

Рассмотрите различия между оперативными и аналитическими системами.

Упражнение 21.2

Изложите, чем отличается модель "сущность — отношение" от размерной модели.

Упражнение 21.3

В начале работы над проектом хранилища данных выполняется так называемый ETL-процесс (extraction, transformation, loading), состоящий из трех этапов: извлечения, преобразования и загрузки. Объясните эти этапы.

Упражнение 21.4

Объясните разницу между таблицей фактов и соответствующими таблицами измерений.

Упражнение 21.5

Обсудите преимущества каждого из трех типов хранилищ данных: MOLAP, ROLAP и HOLAP.

Упражнение 21.6

Зачем требуется агрегировать данные, хранящиеся в таблице фактов?

Глава 22

Службы SQL Server Analysis Services

- **Терминология**
- **Разработка многомерного куба, используя средство BIDS**
- **Извлечение и доставка данных**
- **Система безопасности служб SQL Server Analysis Services**

Службы SQL Server Analysis Services (службы SSAS) являются группой сервисов, которые используются для управления данными, хранящимися в хранилищах или киосках данных. Посредством служб SSAS данные из хранилища данных можно организовать в многомерные кубы (см. *главу 21*) с агрегированными данными, чтобы получить возможность создавать сложные запросы и подробные отчеты. Основными особенностями служб SSAS являются следующие:

- удобство использования;
- поддержка различных архитектур;
- поддержка некоторых интерфейсов API.

Службы SSAS предоставляют мастеров почти для каждой задачи, выполняющейся в процессе проектирования и реализации хранилища данных. Например, мастер указания источников данных *Data Source Wizard* позволяет задать один или несколько источников данных, а мастер кубов *Cube Wizard* применяется для создания многомерных кубов для хранения агрегированных данных. Кроме этого, простота использования обеспечивается средством *Business Intelligence Development Studio (BIDS)*, с помощью которого можно разрабатывать базы данных и другие объекты хранилищ данных. Этому способствует то, что средство BIDS предоставляет один общий интерфейс для разработки проектов SSAS, а также SQL Server Integration Services (SSIS) и SQL Server Reporting Services (SSRS).

В отличие от большинства других систем хранилищ данных, службы SSAS позволяют использовать ту архитектуру, которая наиболее отвечает имеющимся требо-

ваниям. В частности, можно выбирать между тремя архитектурами хранилищ для данных (MOLAP, ROLAP и HOLAP), рассмотренных в *главе 21*.

Службы SSAS предоставляют много различных интерфейсов API для извлечения и доставки данных. Одним из этих интерфейсов является OLE DB для взаимодействия с OLAP, который позволяет обращаться к данным кубов SSAS. Несколько из этих интерфейсов API описывается в *разд. "Извлечение и доставка данных" далее в этой главе*.

В конце главы рассматриваются аспекты безопасности служб SSAS.

Терминология служб SSAS

Далее приводятся наиболее важные термины служб SSAS:

- куб (cube);
- измерение (dimension);
- член (member);
- иерархия (hierarchy);
- ячейка (cell);
- уровень (level);
- группа мер (measure group);
- секция (partition).

Куб (cube) — это многомерная структура, содержащая все или часть данных из хранилища данных. Хотя термин "куб" предполагает три измерения, многомерный куб обычно может иметь намного больше измерений. Каждый куб содержит все другие элементы, перечисленные в предыдущем списке.

Измерение (dimension) — это набор логически связанных атрибутов (хранящихся вместе в таблице измерения), которые подробно описывают меры (хранящиеся в таблице фактов). Например, `Time`, `Product` и `Customer` будут типичными измерениями, используемыми во многих приложениях бизнес-аналитики. (Эти три измерения входят в базу данных `AdventureWorksDW` и используются в примере в следующем разделе для демонстрации создания и обработки многомерного куба, применяя средство BIDS.)

ПРИМЕЧАНИЕ

Одним из важных измерений куба является измерение `Measures`, которое содержит все меры, определенные в таблице фактов.

Каждое измерение содержит дискретные значения, называемые *членами измерения* (member). Например, членами измерения `Product` могут быть `Computers`, `Disks` и `CPUs`. Каждый член может быть вычисляемым, что означает, что его значение вычисляется в процессе выполнения, используя выражение, которое задается при определении члена. (Поскольку вычисляемые члены не сохраняются на диске, то можно добавлять новых членов, не увеличивая объем соответствующего куба.)

Иерархии (hierarchy) задают группирования множественных членов в каждом измерении. Они применяются для уточнения запросов, связанных с анализом данных.

Ячейки (cell) являются частями многомерного куба, которые определяются координатами (*x*, *y* и *z* для трехмерного куба). Это означает, что ячейка является набором, содержащим членов из каждого измерения. Рассмотрим, например, трехмерный куб из *главы 21* (см. рис. 21.3), который представляет продажи легковых автомобилей для одного региона за один квартал. К этому кубу принадлежат, среди прочих, ячейки со следующими координатами:

- `1-й квартал`, `Южная Америка`, `Falcon`;
- `3-й квартал`, `Азия`, `Eagle`.

Иерархии определяются на основе их уровней. Иными словами, *уровни* (level) описывают иерархию от самого высшего уровня данных (наиболее обобщенной) до самого низшего (наиболее подробной). В следующем списке приводятся возможные уровни иерархии для измерения `Time`:

- `Quarter` (`Q1`, `Q2`, `Q3`, `Q4`);
- `Month` (`January`, `February`, ...);
- `Day` (`Day1`, `Day2`, ...).

Как уже упоминалось в *главе 21*, *меры* (measure) — это числовые значения, такие как цена или количество, которые присутствуют в таблице фактов, но не используются в ее первичном ключе. *Группа мер* (measure group) — это набор мер, которые совместно создают логическую единицу для целей деловой деятельности. Каждая группа мер создается на лету, используя соответствующую информацию из метаданных.

Для куба можно задать одну или несколько секций. *Секции* (partition) используются службами SSAS для управления и хранения данных и агрегированных данных для группы мер куба. Каждая группа мер имеет, по крайней мере, одну секцию, которая создается при определении этой группы мер. Секции являются мощным и гибким средством управления больших кубов.

Разработка многомерного куба, используя средство BIDS

Основным компонентом служб SSAS является средство управления Business Intelligence Development Studio (средство BIDS), которое предоставляет единую платформу для разработки разных приложений бизнес-аналитики. Средство BIDS основано на Visual Studio и поддерживает интегрированную платформу для разработчиков систем бизнес-аналитики.

С помощью средства BIDS можно не только создавать и управлять кубами, но также разрабатывать возможности для служб отчетов SQL Server Reporting Services (SSRS) и интеграционных служб SQL Server Integration Services (SSIS). (Службы SSRS рассмотрены в *главе 24*, но предмет служб SSIS выходит за рамки тематики этой книги.)

ПРИМЕЧАНИЕ

Пользовательский интерфейс средства Intelligence Development Studio (BIDS) подобен интерфейсу средства SQL Server Management Studio. Но эти два инструмента имеют разное назначение: средство BIDS применяется для разработки проектов бизнес-аналитики, тогда как основным назначением средства SQL Server Management Studio является работа с объектами баз данных в отношении бизнес-аналитики.

Процесс создания и обработки многомерного куба посредством средства BIDS состоит из следующих шагов:

1. Создание проекта бизнес-аналитики.
2. Определение источников данных.
3. Создание представлений источников данных.
4. Создание куба.
5. Проектирование агрегирования для хранилища.
6. Обработка куба.
7. Просмотр куба.

Все эти шаги подробно описываются в последующих разделах.

Создание проекта бизнес-аналитики

Первым шагом в разработке приложения бизнес-аналитики будет создание нового проекта в BIDS. Чтобы запустить средство BIDS, выполните последовательность команд меню **Пуск | Все программы | Microsoft SQL Server 2012 | SQL Server Business Intelligence Development Studio**. Далее выберите **File | New Project**. В панели **Installed Templates** диалогового окна **New Project** (рис. 22.1) разверните узел **Business Intelligence** и выберите в правой панели окна пункт **Analysis Services Multidimensional and Data Mining Project**.

В поля **Name** и **Location** введите имя и соответственно место расположения проекта. Для этого примера назовем проект BI_Project (рис. 22.1). Затем, чтобы создать новый проект, нажмите кнопку **OK**.

Новый проект всегда создается в новом решении (solution). Следовательно, решение является самой большой единицей управления в средстве BIDS и содержит один или несколько проектов. (В данном примере имя решения такое же, как и имя проекта.)

ПРИМЕЧАНИЕ

Если панель обозревателя решений Solution Explorer (которая позволяет просматривать и управлять объектами в решении или проекте) не открыта, выберите пункты **View | Solution Explorer**, чтобы открыть ее.

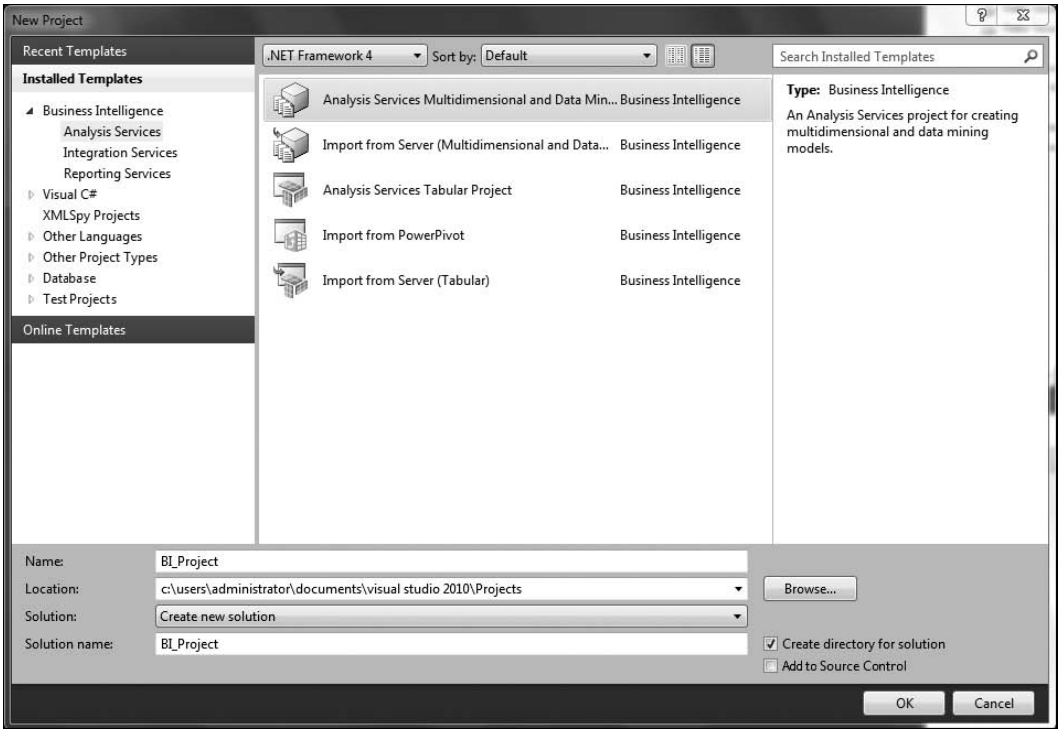


Рис. 22.1. Диалоговое окно **New Project** средства BIDS

Определение источников данных

Чтобы задать источники данных, в панели **Solution Explorer** щелкните правой кнопкой папку **Data Sources** и в появившемся контекстном меню выберите пункт **New Data Source**. Откроется мастер источников данных Data Source Wizard, который будет предоставлять инструкции по созданию источника данных. (В данном примере в качестве источника данных используется база данных примера SQL Server с именем `AdventureWorksDW`.)

Сначала на странице **Select How to Define the Connection** установите переключатель **Create a Data Source Based on an Existing or New Connection** и нажмите кнопку **New**. В открывшемся диалоговом окне **Connection Manager** (рис. 22.2) выберите в раскрывающемся списке **Provider** опцию **Native OLE DB\SQL Server Native Client 11.0**, а в раскрывающемся списке **Server name** выберите имя требуемого сервера баз данных.

(Выбор поставщика **Native OLE DB\SQL Server Native Client 11.0** позволяет подключаться к существующей базе данных экземпляра сервера Database Engine.) В этом же диалоговом окне установите переключатель **Use Windows Authentication**; здесь же установите переключатель **Select or enter a database name** и в раскрывающемся списке под ним выберите базу данных **AdventureWorksDW**. Проверьте соединение с базой данных, нажав кнопку **Test Connection**, и, при по-

ложительном результате проверки, нажмите кнопку **OK**. (Страница **Select How to Define the Connection** отображается только при первом соединении с базой данных.)

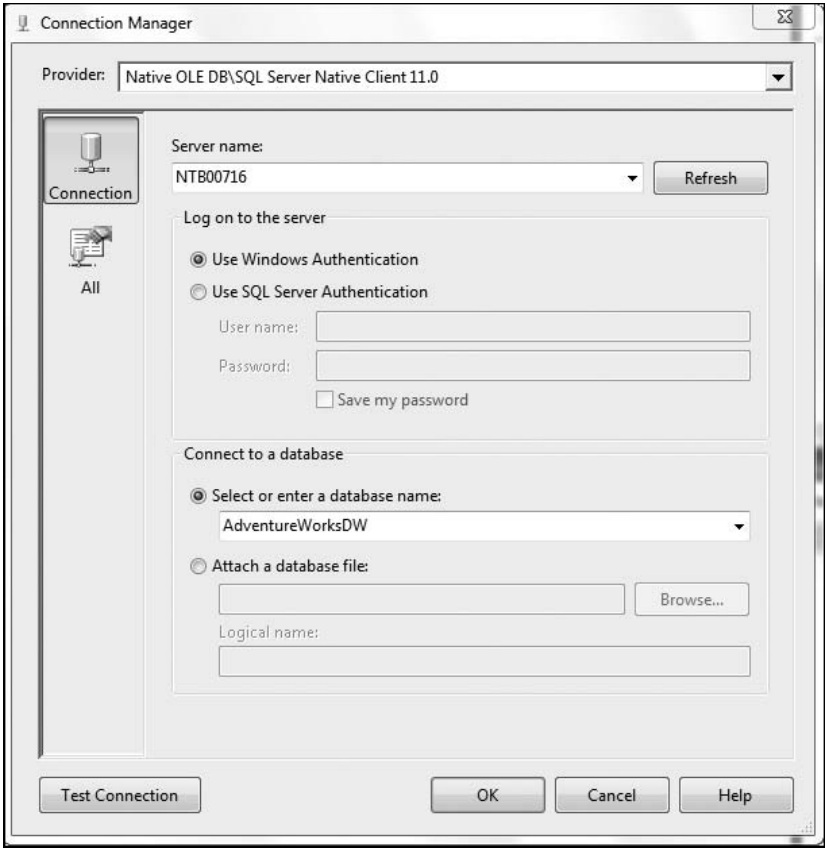


Рис. 22.2. Диалоговое окно **Connection Manager**

Откроется следующая страница мастера, **Impersonation Information**, параметры которой определяют учетную запись пользователя, применяемую службами SSAS при соединении с источником данных, используя проверку подлинности Windows. Выбор определенной настройки зависит от способа использования этого источника данных. Установите переключатель **Use a Specific Windows User Name and Password**, введите в соответствующие поля свои имя пользователя и пароль и нажмите кнопку **Next**.

На следующей странице мастера, **Completing the Wizard**, присвойте новому источнику данных имя (для этого примера это будет **BI_Source**) и нажмите кнопку **Finish**. Созданный источник данных отобразится в папке **Data Sources** панели **Solution Explorer**.

После определения источника данных нужно указать точные данные, которые требуется выбрать из него. В нашем примере это означает, что нужно выбрать таблицы

из базы данных AdventureWorksDW, которые будут использоваться для создания куба. Для этого нужно указать представления источника данных; этот процесс рассматривается в следующем разделе.

Создание представлений источников данных

Чтобы создать представления источников данных, в панели Solution Explorer щелкните правой кнопкой папку **Data Sources Views** и в контекстном меню выберите пункт **New Data Source View**. Откроется мастер Data Source View Wizard, который будет предоставлять инструкции по всем шагам, требуемым для создания представления источника данных. (В этом примере создается представление с именем BI_View, которое основывается на таблицах Customer и Project.)

Сначала на странице **Select a Data Source** выберите существующий реляционный источник данных (для этого примера выберите **BI_Source**) и нажмите кнопку **Next**. На следующей странице мастера — **Select Tables and Views** — добавьте таблицы, входящие в куб как таблицы измерений или как таблицы фактов. Чтобы добавить таблицу, в панели **Available objects** выберите ее имя и нажмите кнопку >, чтобы переместить эту таблицу в панель **Included Objects**. Для этого примера выберите в базе данных AdventureWorksDW таблицы для клиентов и продуктов (Dim Customer и Dim Product соответственно). Эти таблицы будут использоваться для создания куба измерений. Они создают набор таблиц измерения для схемы типа "звезда".

Далее, на этой же странице мастера нужно указать одну или несколько таблиц фактов, которые соответствуют выбранным таблицам измерений. (Одна таблица фактов вместе с соответствующими таблицами измерений создает схему типа "звезда".) Для этого внизу панели **Included objects** нажмите кнопку **Add Related Tables**. Таким образом мы даем задание системе найти таблицы, которые связаны с таблицами DimCustomer и DimProduct. (Чтобы найти связанные таблицы, система просматривает все зависимости "первичный ключ/внешний ключ", которые существуют в базе данных.)

Система обнаружит несколько таблиц фактов и поместит их в панель **Included objects**. Для создания схемы типа "звезда" из всех этих таблиц требуется только одна: FactInternetSales. Кроме соответствующих таблиц фактов, система также выполняет поиск и добавляет другие таблицы, созданные отдельно для уровня иерархии соответствующего измерения. Одной из таких таблиц, которую следует оставить, является таблица DimProductSubcategory, которая воплощает уровень иерархии Subcategory измерения Product. Также следует оставить таблицу Dim Date, т. к. измерение Time почти всегда является частью куба.

Таким образом, для нашей схемы типа "звезда" требуются следующие пять таблиц (как показано на рис. 22.3):

- FactInternetSales;
- DimCustomer;
- DimDate;

- `DimProduct`;
- `DimProductSubcategory`.

Удалите все другие выбранные системой таблицы из правой панели, выбрав ненужную таблицу и нажав кнопку <.

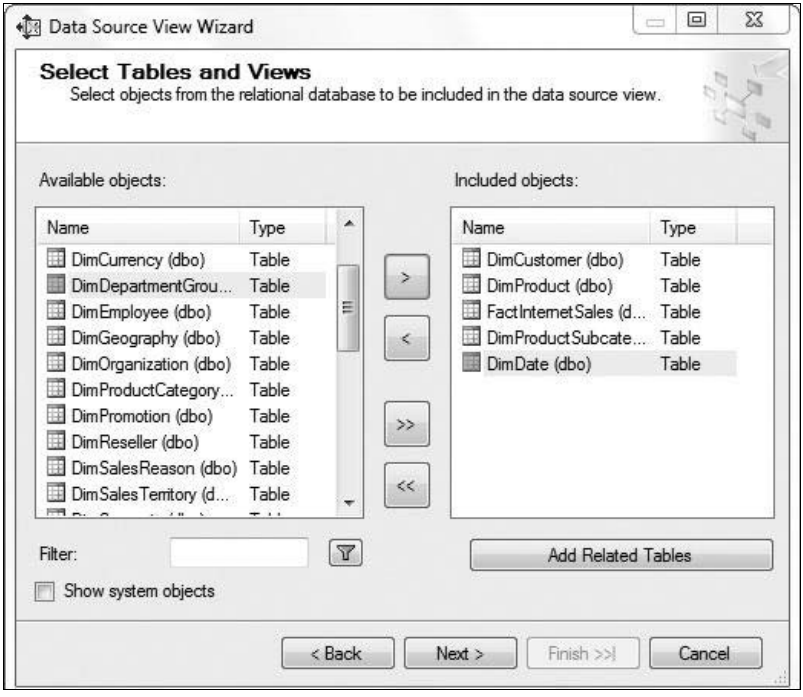


Рис. 22.3. Страница **Select Tables and Views** мастера Data Source View Wizard

Завершив отбор таблиц, нажмите кнопку **Next**. На следующей странице мастера, **Completing the Wizard**, укажите имя для нового представления источника данных (для нашего примера это будет **BI_View**) и нажмите кнопку **Finish**.

После нажатия кнопки **Next** и небольшого времени обработки в окне **Data Source View Designer** будет отображено графическое представление выбранных таблиц в определенной нами схеме данных, как это показано на рис. 22.4.

(Инструмент Data Source View Designer используется для отображения графического представления схем данных.)

ПРИМЕЧАНИЕ

Графическое представление схемы данных, показанное на рис. 22.4, было отредактировано методом перетаскивания, размещая таблицы в форме схемы типа "звезда". На рисунке видно, что таблица фактов находится посредине, а соответствующие таблицы измерений окружают ее. (Схема на рис. 22.4 в действительности имеет организацию типа "снежинка", поскольку таблица `DimProductSubcategory` представляет уровень иерархии измерения `Product`.)

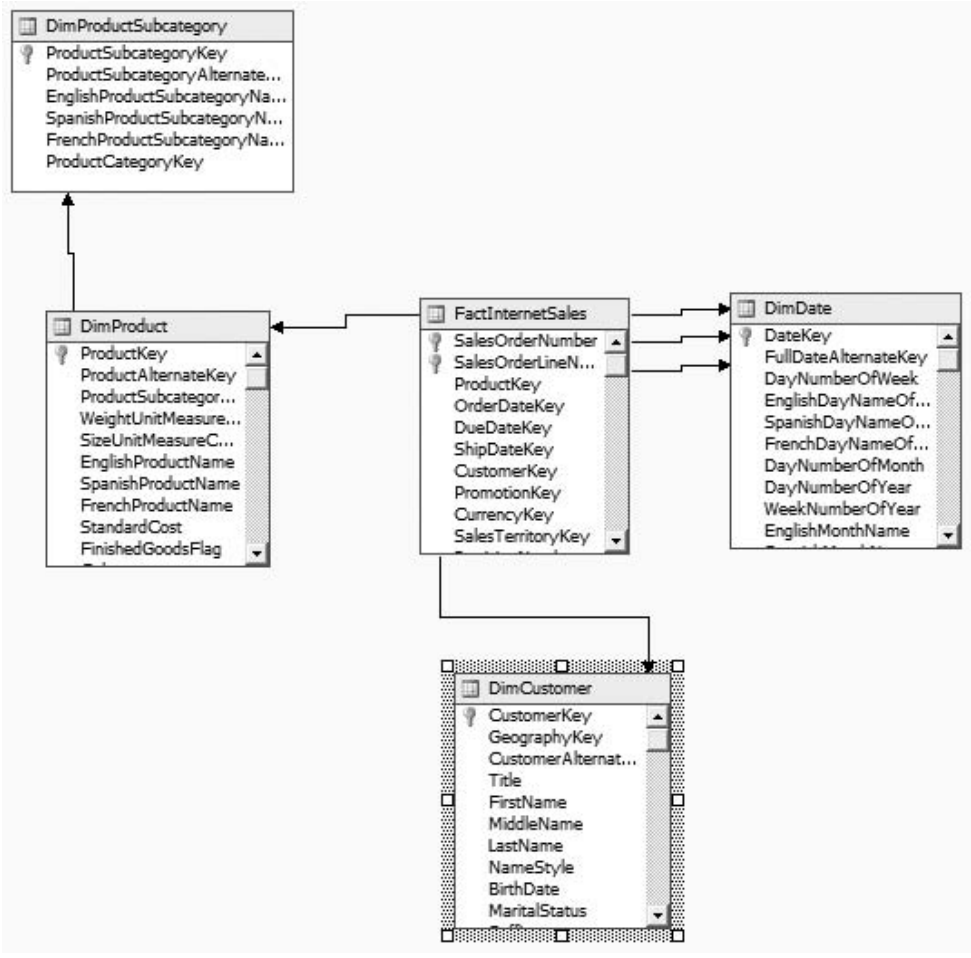


Рис. 22.4. Отображение выбранных таблиц в окне **Data Source View Designer**

Инструмент Data Source View Designer предоставляет несколько полезных функций. Для перемещения по объектам представления источника данных нужно навести указатель мыши на значок скрещенных двунаправленных стрелок в правой нижней части окна. Когда форма указателя мыши сменится на подобную этому значку, нажмите и удерживайте левую кнопку мыши. В правом нижнем углу панели просмотра схемы откроется небольшое окно навигации, в котором можно перемещаться в любую часть диаграммы. Эта функциональность особенно полезна для диаграмм с большим количеством объектов. Альтернативно, перемещаться по диаграмме можно с помощью обычных полос прокрутки (горизонтальной и вертикальной), что более подходит для диаграмм небольшого размера. Для просмотра данных таблицы щелкните требуемую таблицу правой кнопкой мыши и в контекстном меню выберите пункт **Explore Data**. Содержимое таблицы отобразится в отдельном окне.

Можно также создавать именованные запросы, которые сохраняются на постоянной основе, что позволяет обращаться к ним как к любой таблице. Чтобы создать

именованный запрос, выберите пункт меню **Data Source View**, а затем пункт **New Named Query**. Откроется диалоговое окно **Create Named Query**, в котором можно создавать любые запросы для выбранных таблиц.

Создание куба

Прежде чем создавать куб, нужно задать один или несколько источников данных и создать представление источника данных, как было описано ранее в предшествующих разделах. После этого можно приступать к созданию куба.

Для этого в обозревателе решений щелкните правой кнопкой мыши папку **Cubes** проекта **BI_Project** и в контекстном меню выберите пункт **New Cube**. Откроется страница приветствия мастера кубов. Нажмите кнопку **Next**. На странице **Select Creation Method** выберите переключатель **Use Existing Tables**, поскольку представление источника данных уже существует и его можно использовать для создания куба. Нажмите кнопку **Next**.

На странице **Select Measure Group Tables** выбираются меры из таблиц фактов. Выберите здесь таблицу **Fact Internet Sales** и нажмите кнопку **Next**. Мастер отберет все возможные меры из указанной таблицы фактов и отобразит их на странице **Select Measures**. Установите только флажок столбца `Total Product Costs` таблицы `Fact Internet Sales` только для одной меры (рис. 22.5) и нажмите кнопку **Next**.

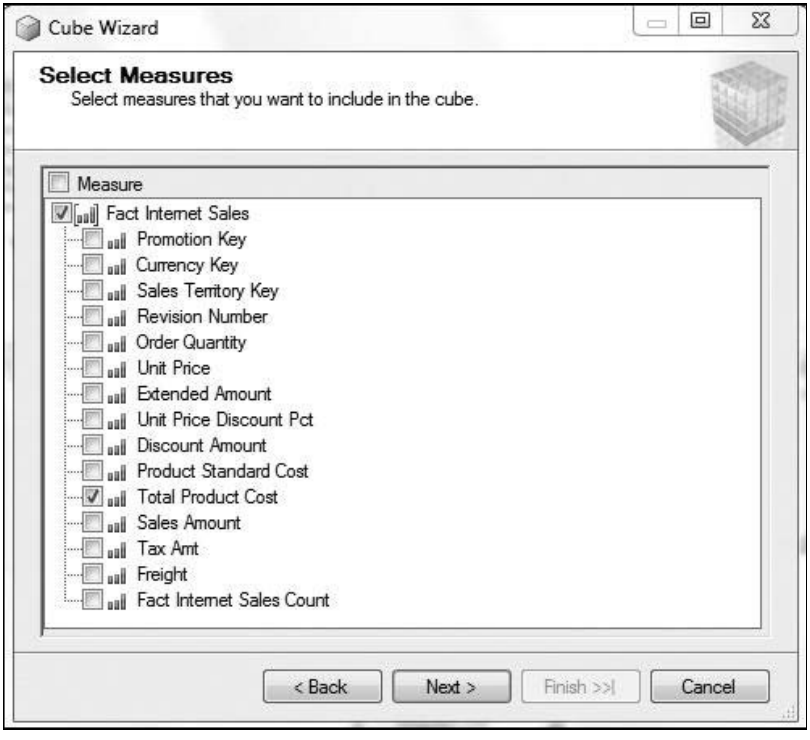


Рис. 22.5. Таблица мастера **Select Measures**

На странице **Select New Dimensions** выберите все три измерения (`Dim Date`, `Dim Product` и `Dim Customer`), которые нужно создать на основе имеющихся таблиц, и нажмите кнопку **Next**. На последней странице мастера **Completing the Wizard** отображается сводная информация по всем выбранным мерам и измерениям. Введите имя создаваемого куба (в данном примере `BI_Cube`) и, чтобы создать этот куб, нажмите кнопку **Finish**.

Проектирование агрегирования для хранилища

Как упоминалось в *главе 21*, базовые данные из таблицы фактов можно суммировать наперед и сохранять результаты в постоянных таблицах. Этот процесс называется *агрегированием* и может существенно повысить скорость выполнения запросов, поскольку сканирование миллионов строк по ходу выполнения запроса для вычисления агрегатного значения может занять очень длительное время.

Существует прямая взаимосвязь между требованиями дискового пространства для хранения агрегированных значений и процентом всех возможных агрегирований, которые вычисляются и сохраняются. Создание всех возможных агрегированных значений куба и сохранение их на диске позволяет получить наилучшее время выполнения всех запросов. Недостатком этого подхода являются значительные требования процессорных ресурсов для вычисления агрегированных значений и дискового пространства для их хранения.

С другой стороны, если не вычислять наперед и не сохранять никаких агрегаций, не будет требоваться дополнительного дискового пространства, однако выполнение запросов, содержащих агрегатные функции, будет медленным, вследствие необходимости вычисления каждого агрегатного значения в ходе выполнения запроса.

Для разработки оптимальных агрегаций службы SSAS предоставляют мастер Aggregation Design Wizard. Чтобы запустить этот мастер, сначала нужно запустить средство Cube Designer. (Средство Cube Designer применяется для редактирования разных свойств существующих кубов, включая группы мер и отдельные меры, измерения куба и взаимосвязи между измерениями.) Для этого щелкните правой кнопкой требуемый куб в обозревателе решений и в контекстном меню выберите пункт **Open** или **View Designer**. В открывшемся окне средства Cube Designer откройте вкладку **Aggregations**. В отображенной таблице (`Fact Internet Sales`) щелкните правой кнопкой ячейку в столбце **Aggregations** и в контекстном меню выберите пункт **Design Aggregations**. Будет запущен мастер Aggregation Design Wizard.

На первой странице мастера, **Review Aggregation Usage**, можно просмотреть и откорректировать параметры агрегирования. В частности, можно включить или отключить атрибуты, отображенные на этой странице. Оставьте параметры как они есть и нажмите кнопку **Next**.

На следующей странице, **Specify Object Counts**, нужно указать количество членов в каждом атрибуте. Прежде чем мастер может начать создать и сохранять выбранные агрегации, для каждого выбранного объекта куба требуется предоставить чис-

ло значений или число секций. Эту задачу можно выполнить с помощью мастера, нажав кнопку **Count**. На рис. 22.6 показана страница **Specify Object Counts** с результатами этих вычислений мастера. Нажмите кнопку **Next**.

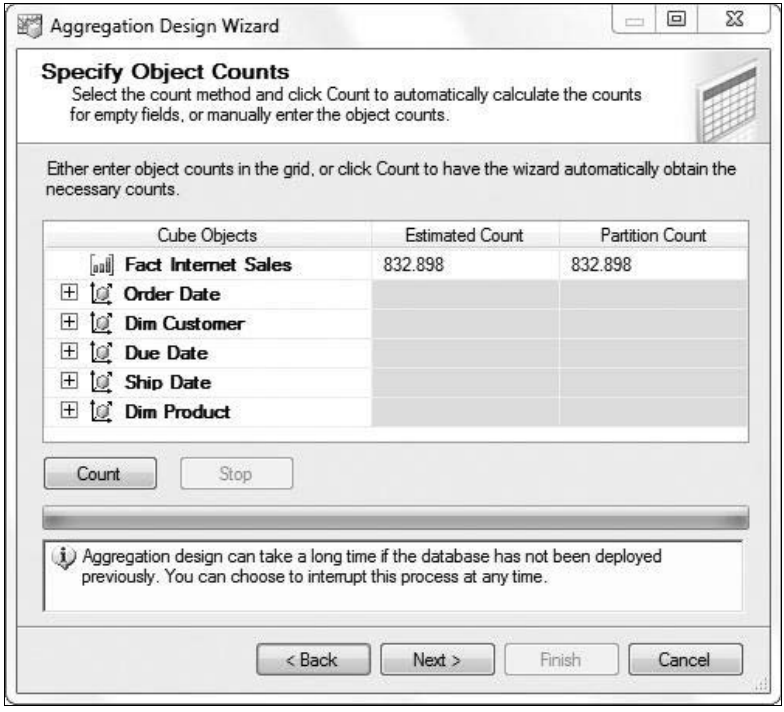


Рис. 22.6. Страница мастера **Specify Object Counts** после подсчета мастером объектов

На предпоследней странице, **Set Aggregation Options**, для указания точки, до которой нужно проектировать агрегации (или не проектировать их вообще), выберите одну из четырех следующих опций:

- **Estimated storage reaches_MB** (предполагаемый объем хранилища достигнет_Мбайт) — задает максимальный объем дискового хранилища, который следует использовать для вычисленных наперед агрегаций. Чем больше это значение, тем большее количество вычисленных наперед агрегаций будет создано;
- **Performance gain reaches_%** (прирост производительности достигнет__%) — задает прирост производительности, которую требуется получить. Чем больше процентное значение вычисленных наперед агрегаций, тем лучше будет производительность запросов;
- **I click Stop** (я нажму кнопку **Stop**) — позволяет пользователю остановить процесс проектирования в любой момент по своему желанию;
- **Do not design aggregation (0%)** (не проектировать агрегирование) — указывает не создавать вычисляемые наперед агрегации.

ПРИМЕЧАНИЕ

Обычно следует выбрать одну из первых двух опций. Автор предпочитает вторую из них, поскольку предположительно оценить требуемый объем дискового пространства для разных схем типа "звезда" и разных наборов запросов представляет очень трудную задачу. Значение между 80 и 90% будет оптимальным в большинстве случаев.

На рис. 22.7 показан результат выбора второй опции, для которой было установлено значение 80%.

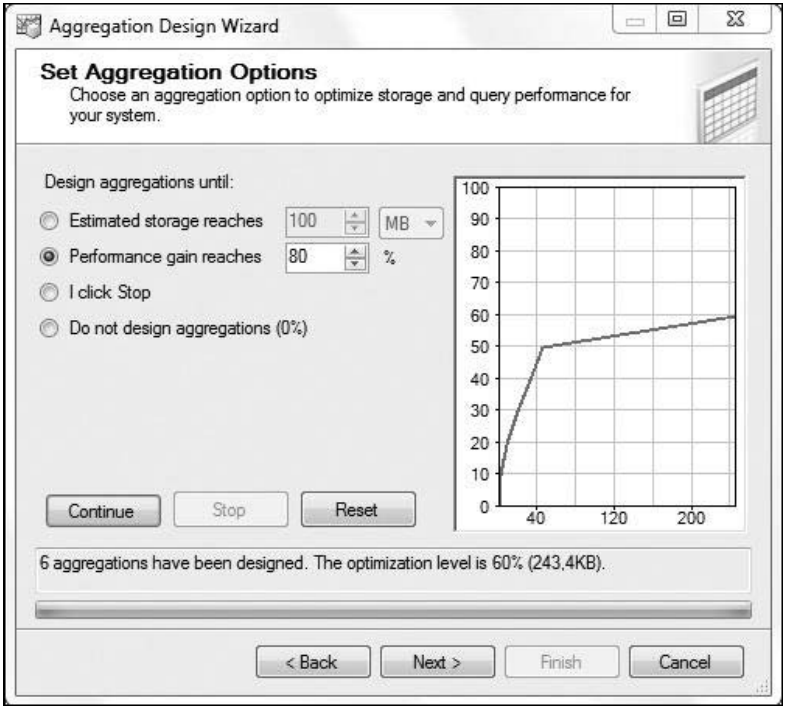


Рис. 22.7. Результат выбора проектирования агрегаций для получения 80% прироста производительности

Система создала шесть агрегаций и использует 243,4 Кбайт для их хранения.

Чтобы перейти к странице **Completing the Wizard**, нажмите кнопку **Next**. На этой странице можно выбрать метод применения полученных агрегаций: немедленно (переключатель **Deploy and process now**) или позже (переключатель **Save the aggregations but do not process them**). Выберите второй переключатель и нажмите кнопку **Finish**.

Обработка куба

Если в предшествующем разделе вы выбрали рекомендованную опцию **Save the aggregations but do not process them**, то теперь нужно выполнить обработку куба.

Куб требуется обрабатывать после его создания и после каждого его изменения. Если куб содержит большой объем данных и вычисляемых наперед агрегаций, его обработка может занимать очень длительный промежуток времени. Чтобы начать обработку куба, в папке **Cubes** обозревателя решений щелкните правой кнопкой мыши требуемый куб и в контекстном меню выберите пункт **Process**. Система начинает обработку куба, отображая ход выполнения этого процесса (рис. 22.8).

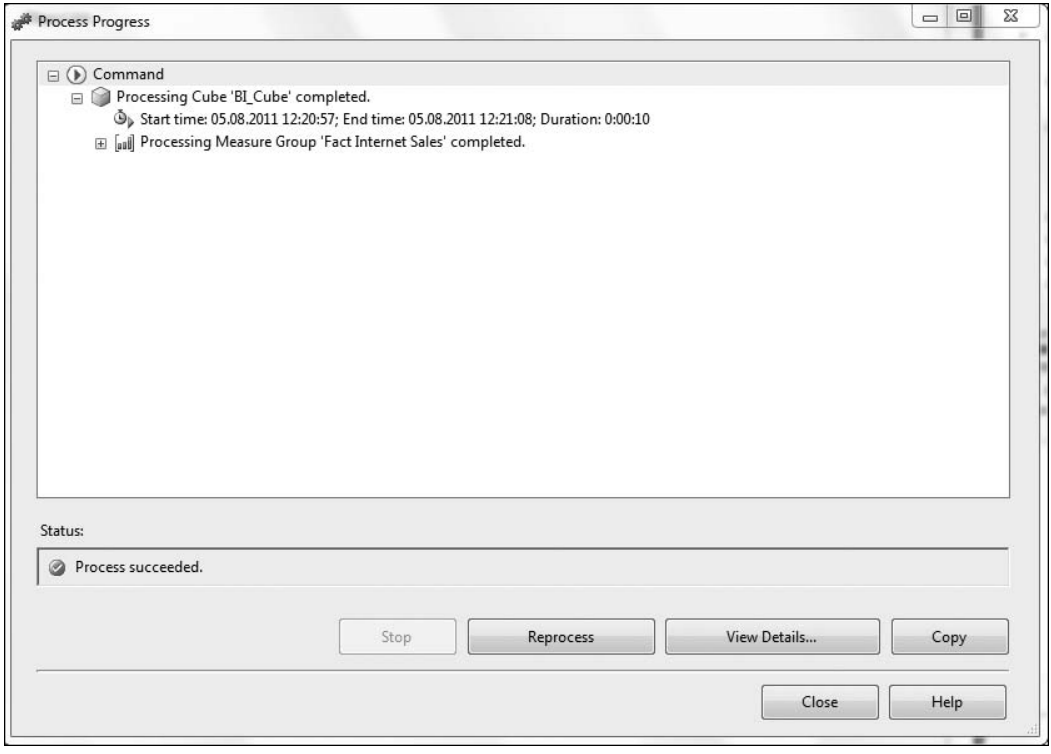


Рис. 22.8. Окно обработки куба с индикатором хода выполнения

Просмотр куба

Для просмотра куба, щелкните правой кнопкой мыши требуемый куб и в контекстном меню выберите опцию **Browse**. Откроется окно просмотра куба **Browse**. Здесь в запрос можно вставить любое измерение, щелкнув правой кнопкой имя требуемого измерения в левой панели и выбрав в контекстном меню опцию **Add to Query**. Таким же образом к запросу добавляются и меры. (Меры рекомендуется добавлять первыми.) На рис. 22.9 показано табличное представление общей стоимости продуктов для продаж типа `Internet Sales` для разных клиентов и продуктов.

Для вычисления значений мер для определенных измерений и их иерархий применяется другой подход. Предположим, например, что нам нужно предоставить клиентам с идентификаторами 11008 и 11741 информацию об общей стоимости всех

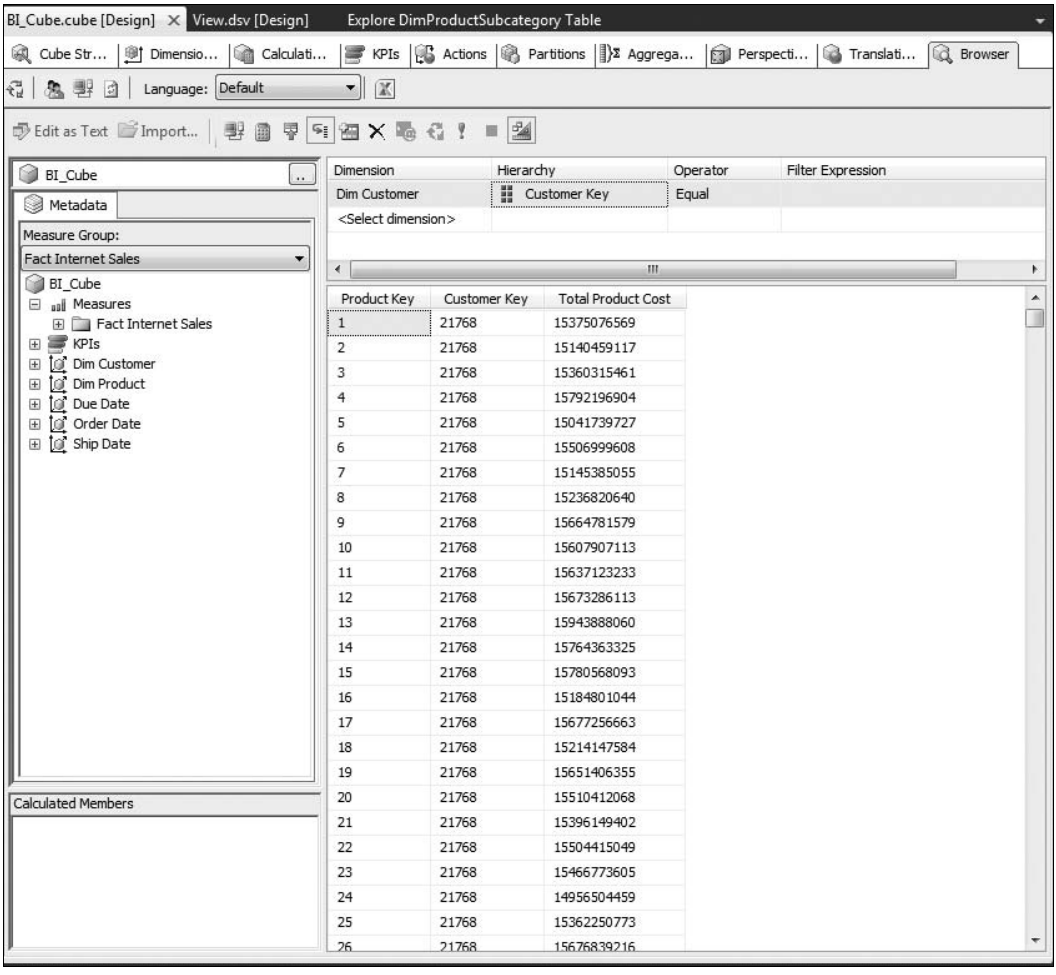


Рис. 22.9. Общая стоимость продаж типа Internet Sales для разных клиентов и продуктов

продуктов, которые они заказали в период между 1/03/2006 и 31/08/2006. Для этого мы сначала перетаскиваем меру `Total Product Cost` с левой панели в панель редактирования, а затем в расположенной выше панели выбираем значения, чтобы ограничить условия для каждого измерения (рис. 22.10).

Сначала в столбце `Dimension` выбираем таблицу `Dim Customer`, а в столбце `Hierarchy` первичный ключ этой таблицы: `Customer Key`. Затем в столбце `Operator` выбираем `Equal`, а в столбце `Filter Expression` — значения `11008` и `11741`.

Таким же образом задаются условия для таблицы измерений `Dim Product` с единственной разницей, что вместо указания только двух значений, выбирается опция `All` с помощью установки соответствующего флажка в столбце `Filter Expression`. (Корень каждого измерения представляется значением `All`.) Наконец, для таблицы измерения `Dim Date` выбирается столбец `Order Date` с соответствующим ключом. В этом случае в столбце `Operator` выбирается значение `Range (Inclusive)`, а в

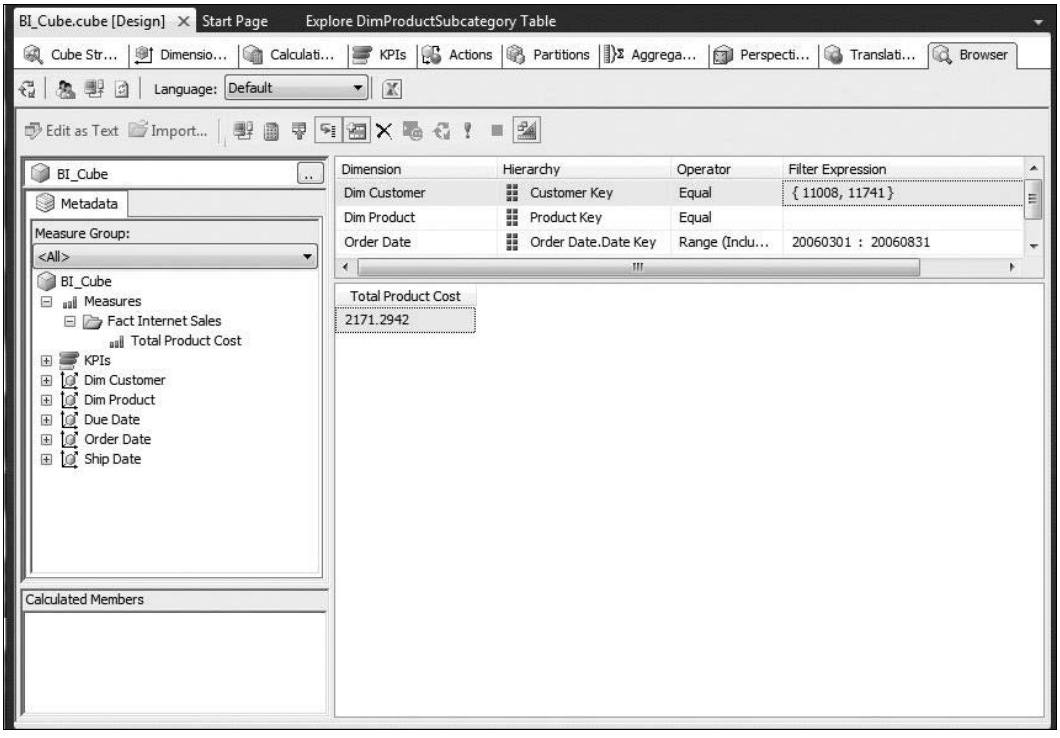


Рис. 22.10. Вычисление общей стоимости продаж продуктов определенным клиентам за определенный период времени

столбце `Filter Expression` — начальная и конечная дата требуемого периода. Общая сумма продаж по заданным фильтрам составляет 2171.2942, как это можно видеть на рис. 22.10.

Извлечение и доставка данных

Теперь, когда мы знаем, как создавать и просматривать куб с помощью средства BIDS, можем перейти к рассмотрению того, как извлекать из куба данные и доставлять их пользователям. Основной целью средства Development Studio является разработка проектов бизнес-аналитики, а не извлечение данных и доставка их пользователям. Для этой задачи существует много других интерфейсов, включая следующие:

- надстройка PowerPivot for Excel;
- язык запросов Multidimensional Expressions (MDX);
- среда Management Studio;
- инструментальный набор OLE DB for OLAP;
- инструменты сторонних разработчиков;
- поставщик данных ADOMD.NET.

Надстройка PowerPivot for Excel и язык запросов MDX рассматриваются каждый отдельно в последующих двух разделах, а остальные средства, из ранее приведенного списка, рассмотрены здесь вкратце. Этим двум средствам уделено большое внимание, чем остальным по той причине, что надстройка PowerPivot for Excel является самым важным интерфейсом для конечных пользователей, а язык запросов MDX используется во многих SSAS-решениях сторонних разработчиков.

Для просмотра куба в среде Management Studio запустите этот инструмент и подключитесь к серверу служб SSAS, на котором развернут требуемый куб. В обозревателе объектов разверните папку **Database**, а в ней разверните папку **Cube**, которая содержит все кубы, созданные для данной базы данных. Щелкните правой кнопкой мыши требуемый куб и, в появившемся контекстном меню, выберите пункт **Browse**. Интерфейс для просмотра кубов точно такой же, как и для их разработки. Это можно видеть на рис. 22.9 и 22.10. Поэтому просмотр данных куба можно осуществлять таким же способом, как описано ранее в *разд. "Просмотр куба" в этой главе*.

ПРИМЕЧАНИЕ

Самым важным аспектом, который следует принять во внимание при использовании среды Management Studio для многомерного анализа, является то, что мы не подключаемся к компоненту Database Engine. Компонент Database Engine применяется для управления реляционными данными, тогда как службы SSAS применяются для хранения и управления многомерными кубами. По этой причине подключаться следует к серверу SSAS.

Инструментальный набор OLE DB for OLAP является промышленным стандартом для обработки многомерных данных, предлагаемый компанией Microsoft. Этот набор инструментальных средств и интерфейсов расширяет возможности OLE DB, предоставляя доступ к многомерным складам данных. Инструментальный набор OLE DB for OLAP позволяет выполнять анализ данных посредством интерактивного обращения к разнообразным возможным представлениям данных. Многие независимые разработчики программного обеспечения используют спецификацию OLE DB for OLAP для реализации разных интерфейсов, позволяющих пользователям обращаться к кубам, созданным посредством служб SSAS. Кроме этого, используя OLE DB for OLAP, разработчики могут реализовывать приложения OLAP, которые могут одинаково обращаться как к реляционным, так и нереляционным данным, хранящимся в различных источниках информации, независимо от их расположения или типа.

Поставщик данных ADOMD.NET (ActiveX Data Objects Multidimensional) предназначен для взаимодействия со службами SSAS. С помощью этого интерфейса можно обращаться к данным в многомерном кубе и манипулировать ими, что предоставляет возможность разработки приложений OLAP на основе веб-технологий. Этот интерфейс использует протокол XML for Analysis для взаимодействия с источниками аналитических данных. Команды обычно отправляются на языке MDX. С помощью ADOMD.NET можно также просматривать метаданные и работать с ними.

Обращение к данным посредством PowerPivot for Excel

Надстройка PowerPivot for Excel позволяет анализировать данные, используя наиболее популярное средство для этой цели — Microsoft Excel. Это удобный для пользователя способ выполнять анализ данных, используя такие функциональности, как PivotTable, PivotChart представления и срезы (slices).

ПРИМЕЧАНИЕ

Для работы с PowerPivot на компьютер должен быть установлен офисный пакет Microsoft Office 2010. Можно также использовать SharePoint 2010, но в этой главе рассматривается только PowerPivot for Excel.

Прежде чем приступать к изучению, как работать с этим инструментом, давайте рассмотрим его преимущества, которые включают, среди прочих, следующие:

- знакомые инструменты и возможности Excel для представления данных;
- наборы данных очень большого объема можно загружать практически из любого источника;
- наличие новых аналитических возможностей, таких как Data Analysis Expressions (DAX).

Как мы вскоре увидим, те же самые источники данных, которые используются для служб SSAS, можно использовать почти таким же образом и для PowerPivot. (В рассматриваемых примерах будет использоваться куб наподобие куба, созданного в предшествующем разделе, чтобы научиться доставлять данные из куба. Далее этот куб будет использоваться в других упражнениях.)

Новый язык надстройки PowerPivot — Data Analysis Expressions (DAX) — позволяет определять специализированные вычисления в таблицах PowerPivot и Excel PivotTables. Язык DAX содержит некоторые функции, которые используются в формулах Excel, и дополнительные операции для работы с реляционными данными.

Работа с PowerPivot for Excel

Первым шагом в работе с PowerPivot for Excel будет импортировать данные с одного или нескольких источников в таблицу Excel. Запустите Excel 2010 и выберите вкладку **PowerPivot**. В ленте вкладки **PowerPivot** значок **PowerPivot Window**. Откроется окно **PowerPivot for Excel**. Наша задача заключается в том, чтобы создать куб, подобный тому, который мы создали посредством служб SSAS. Поэтому создайте и обработайте куб, следуя шагам, описанным в предшествующем разделе, и применяя схему таблиц, показанную на рис. 22.11.

Далее приводится список таблиц, которые нужно выбрать (см. рис. 22.11):

- `FactInternetSales`
- `FactResellerSales`

- DimCustomer
- DimDate
- DimProduct
- DimProductSubcategory

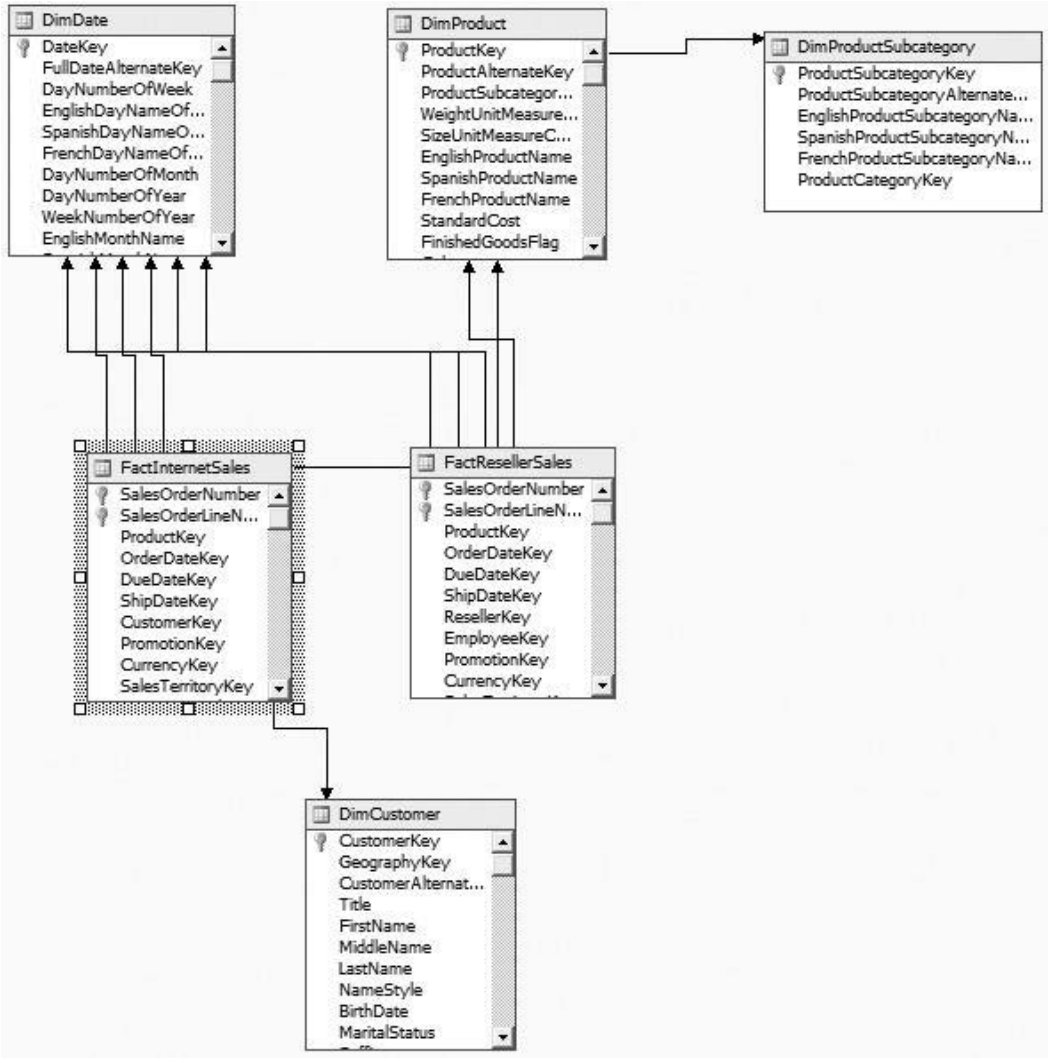


Рис. 22.11. Таблицы (в окне **Data Source View Designer**), которые нужно выбрать для создания куба

Чтобы выбрать эти таблицы, используйте тот же самый источник данных (**BI_Source**), что и для проекта BI_Cube, и создайте новое представление источника данных. Затем создайте с помощью мастера Cube Wizard новый куб и назовите его **BI_Cube2**. На странице мастера **Select Measure Group Tables** выберите следующие меры с обеих таблиц фактов (`FactInternetSales` и `FactResellerSales`): `Sales Amount`, `Total Product Costs` и `Freight`.

Следующей задачей после создания куба будет задача подключения к нему. Для этого в разделе **Get External Data** ленты окна **PowerPivot for Excel** нажмите значок **From Other Sources**. Откроется мастер Table Import Wizard на странице **Connect to a Data Source** (рис. 22.12). На ней выберите опцию **Microsoft Analysis Services** и нажмите кнопку **Next**.

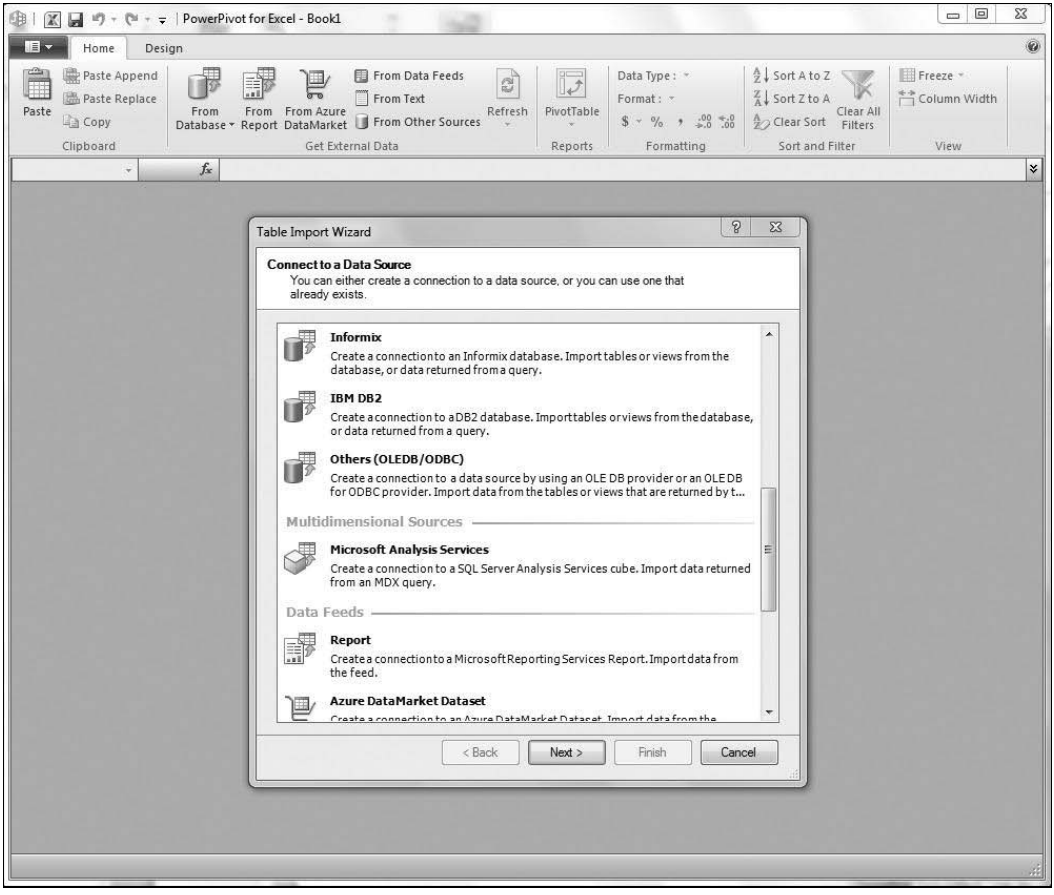


Рис. 22.12. Мастер импорта таблиц, страница **Connect to a Data Source**

ПРИМЕЧАНИЕ

В качестве источника данных можно выбрать базу данных Microsoft (Access или SQL Server), базу данных стороннего разработчика (Oracle, Teradata и т. п.), а также источники данных, отличные от баз данных.

На следующей странице мастера, **Connect to Microsoft SQL Server Analysis Services** (рис. 22.13), нужно указать информацию о кубе, к которому нужно подключиться. В текстовое поле **Friendly connection name** введите имя соединения (например, PowerPivot_Project), в поле **Server or File Name** введите имя сервера, а также выберите способ проверки подлинности для входа на сервер.

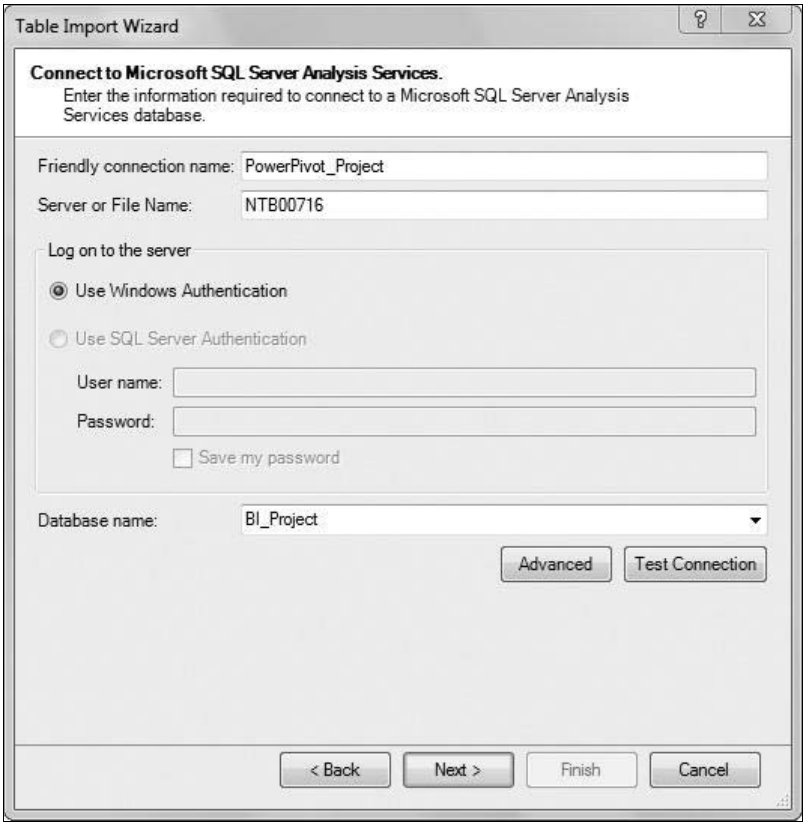


Рис. 22.13. Страница мастера **Connect to Microsoft SQL Server Analysis Services**

Наконец, в поле **Database name** введите имя существующего куба (в данном случае это будет BI_Cube2). (Из списка имен существующих кубов можно также выбрать имя куба, который предоставляется для конкретного сервера.) Нажмите кнопку **Next**.

В следующем окне **Choose How to Import the Data** установите переключатель **Write a query that will specify the data to import** и нажмите кнопку **Next**. В следующем окне **Specify SQL Query** для выборки данных для импортирования из источника данных имеется возможность задать запрос MDX. Этот запрос можно ввести в поле **SQL Statement** вручную или вставить, скопировав его, а затем нажать кнопку **Validate**, чтобы проверить его правильность. Альтернативно, этот запрос можно создать с помощью графического средства Query Designer. (Создание запросов на языке MDX рассматривается в следующем разделе.)

Чтобы создать запрос графически, нажмите кнопку **Design**. Откроется окно графического разработчика запросов мастера Table Import Wizard (рис. 22.14).

Здесь можно выбрать требуемые для вычислений меры и поля из существующих таблиц фактов и таблиц измерений соответственно. Перетащите из таблицы фактов `Fact Reseller Sales` меру `Sales Amount`, а из таблиц измерений `Due Date` и `Dim Product` меры `Data Key` и `Product Key` соответственно. Нажмите кнопку **OK**.

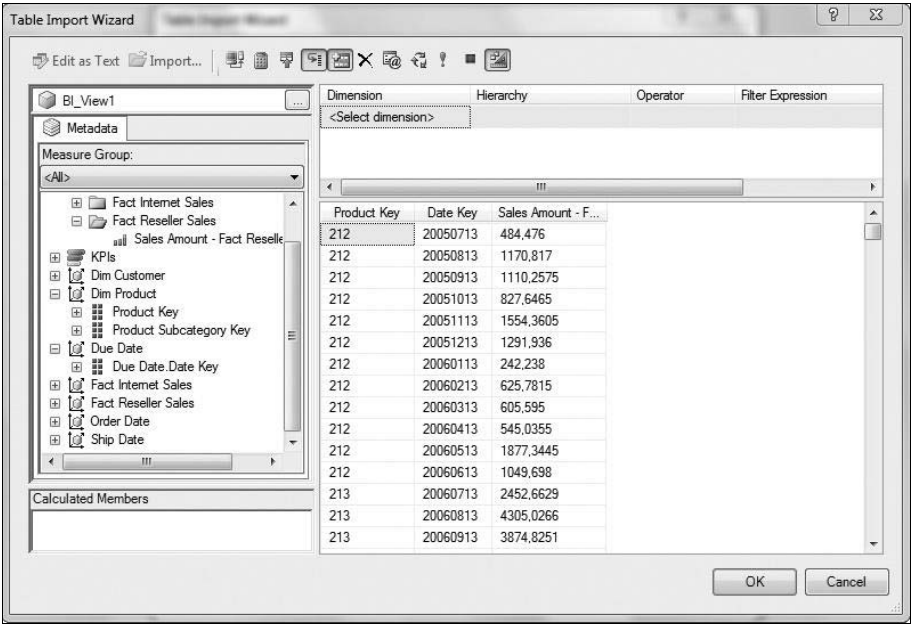


Рис. 22.14. Графический разработчик запросов мастера Table Import Wizard

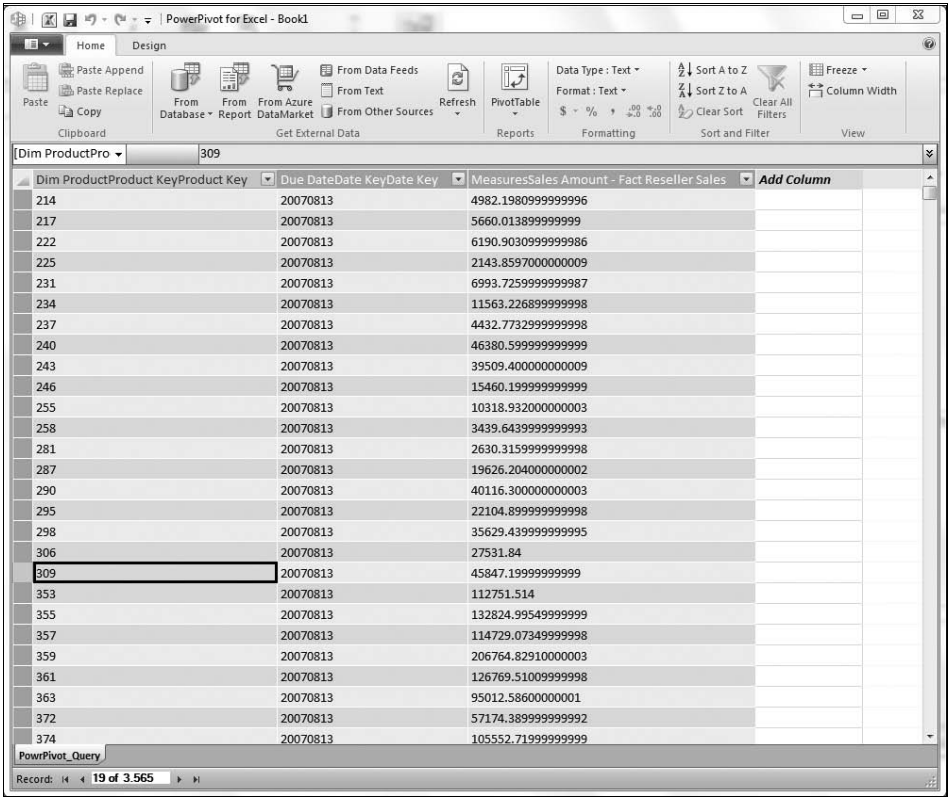


Рис. 22.15. Выбранные запросом данные вставляются в таблицу PowerPivot for Excel

Система возвратится назад к окну **Specify SQL Query**, но в котором уже будет отображен текст созданного запроса. Чтобы начать импорт данных, нажмите кнопку **Finish**. По завершению процесса импорта в поле сведений сообщается о его успешном (или неуспешном) выполнении.

Нажатие кнопки **Close** закрывает окно импорта данных и помещает выбранные запросом данные в таблицу PowerPivot for Excel (рис. 22.15).

Теперь эти данные можно представлять, используя одну из форм презентации Excel. В этом примере мы рассмотрим, как создавать перекрестные табличные данные (сводные таблицы) для представления данных в Excel. В разделе **Reports** ленты окна **PivotTable for Excel** нажмите значок **PivotTable**. Откроется диалоговое окно **Create PivotTable**. Установите в нем переключатель **New Worksheet** и нажмите кнопку **OK**. В окне Excel появится новый лист (рис. 22.16).

В правой панели, которая называется **PowerPivot Field List**, можно выбирать столбцы для представления в листе. (Если панель **PowerPivot Field List** не отображается, то нажмите на ленте значок **Field List**, чтобы отобразить ее.) В этом примере выбраны (установлены флажки) мера и два столбца из таблицы измерений. Теперь данные можно представлять в листе, перетаскивая и помещая их в область **Row Labels**, **Column Labels** или **Values**. На рис. 22.16 ключевой столбец таблицы

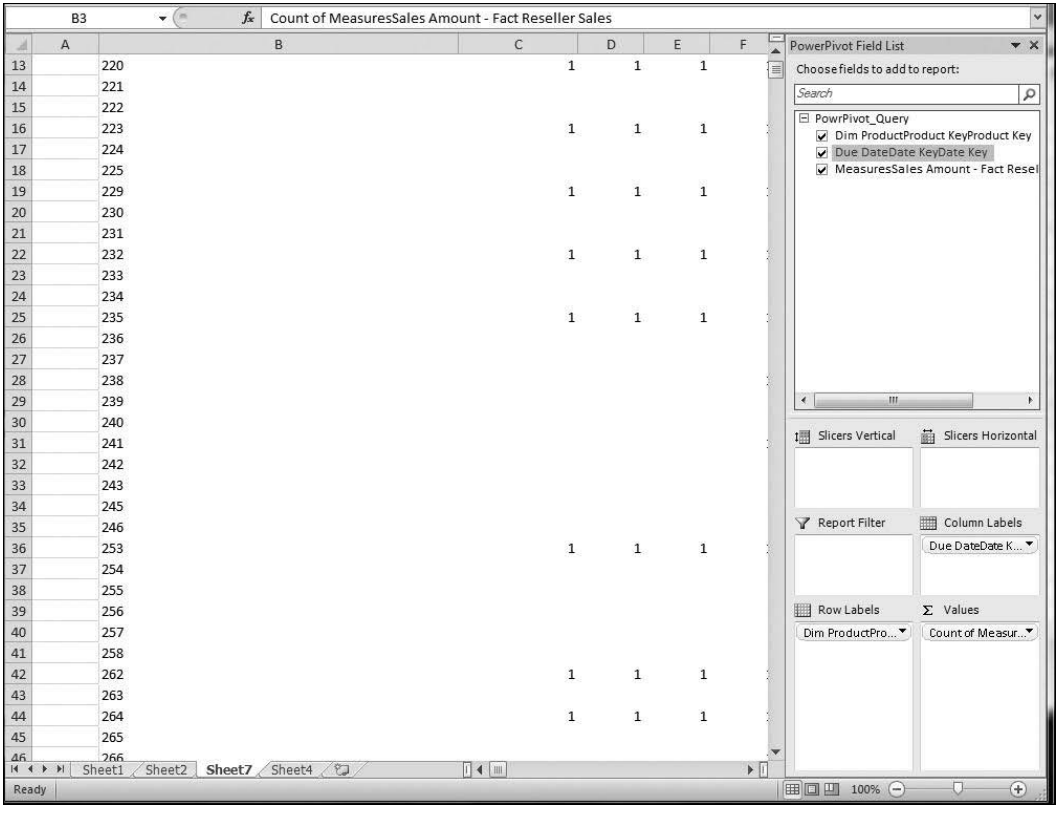


Рис. 22.16. Представление данных в листе Excel

измерения `Dim Product` помещен в область **Row Labels**, а ключевой столбец таблицы измерений `Dim Date` помещен в область **Column Labels**. (Мера помещена в область **Values**.)

ПРИМЕЧАНИЕ

Представленные на рис. 22.16 данные показывают вычисления, выполненные посредством агрегатной функции `COUNT`. Чтобы изменить тип вычислений (например, на суммирование значений), щелкните правой кнопкой в пустом месте области `Values` и в контекстном меню выберите требуемую функцию (например, `SUM`).

Обращение к данным посредством многомерных выражений

Язык Multidimensional Expressions (MDX) применяется для обращения к многомерным данным, хранящимся в кубах OLAP. (С помощью этого языка можно также создавать кубы.) В языке MDX инструкция `SELECT` задает результирующий набор, который содержит подмножество многомерных данных, выбранных из куба. Чтобы задать результирующий набор, запрос MDX должен содержать перечисленную далее информацию.

- Одну или несколько осей, применяемых для указания результирующего набора. В запросе MDX можно указать до 128 таких осей. Первая ось задается предложением `ON COLUMNS`, а вторая — предложением `ON ROWS`. В случае использования больше чем двух осей, применяется альтернативный синтаксис: первая ось указывается предложением `ON AXIS(0)`, вторая — предложением `ON AXIS(1)` и т. д.
- Набор членов или записей, включаемых в каждую ось запроса MDX. Это указывается в списке `SELECT`.
- Имя куба, который устанавливает контекст запроса MDX, указывается в предложении `FROM` запроса.
- Набор чисел или записей для включения в ось среза, которые указываются в предложении `WHERE` (см. примеры 22.1 и 22.2).

ПРИМЕЧАНИЕ

Семантическое значение предложения `WHERE` в SQL не такое, как в языке MDX. В SQL оно означает фильтрацию строк по заданному критерию. А в языке MDX предложение `WHERE` означает выполнение среза многомерного запроса. Хотя эти понятия в некоторой мере сходны, они не являются эквивалентными.

Синтаксис языка объясняется на примере 22.1. Запросы MDX можно выполнять непосредственно в среде Management Studio. Для разработки и выполнения запросов и сценариев на языке MDX применяется редактор запросов MDX Query Editor. Сценарий вводится или копируется в панель редактирования запроса, а для его

выполнения нажимается клавиша <F5> или кнопка **Execute** на панели инструментов редактора запросов.

Пример 22.1. Выборка общей стоимости продуктов для каждого клиента к оплате на 1 марта 2007 г.

```
SELECT [Measures].MEMBERS ON COLUMNS,
        [Dim Customer].[Customer Key].MEMBERS ON ROWS
FROM BI_Cube
WHERE ([Due Date].[Date Key].[20070301])
```

В примере 22.1 осуществляется выборка данных из куба `BI_Cube`. Список `SELECT` первой оси запроса выбирает всех членов измерения `Measures`. Иными словами, он выбирает значения столбца `Total Product Costs`, т. к. это единственная мера в этом кубе. Вторая ось запроса отображает всех членов столбца `Customer Key` измерения `Dim Customer`.

В предложении `FROM` указывается, что источником данных в этом случае является куб `BI_Cube`. Предложение `WHERE` выполняет срез по измерению `Due Data` в соответствии с ключевыми значениями, используя одно значение даты — `2007/03/01`.

В примере 22.2 показан еще один запрос MDX.

Пример 22.2. Вычисление общей стоимости продукта номер 7 для клиента номер 11741 к уплате в марте 2007 г.

```
SELECT [Measures].MEMBERS ON COLUMNS
   FROM BI_Cube
WHERE ({[Due Date].[Date Key].[20070301]:[Due Date].[Date key].[20070331]},
   [Dim Customer].[Customer Key].[11741],
   [Dim Product].[Product Key].[7])
```

Список `SELECT` в примере 22.2 содержит только членов измерения `Measures`. Поэтому запрос отображает значение столбца `Total Product Costs`. Предложение `WHERE` в примере 22.2 более сложное, чем в примере 22.1. Во-первых, в нем выполняется три операции среза, которые разделяются запятыми. Для среза используется только один член измерения `Customer` и один член измерения `Product`, тогда как из измерения `Due Date` вырезаются даты с 2007/03/01 по 2007/03/31. (Как можно видеть в запросе для указания диапазона дат используется знак двоеточия ":").

ПРИМЕЧАНИЕ

Язык MDX очень сложный. В этом разделе дано только очень краткое его описание. Дополнительную информацию по этому языку смотрите в *электронной документации*. Кроме этого, весьма рекомендуется статья *MDX for Everyone* автора Моша Пасумански (Mosha Pasumansky), одного из основателей языка MDX. Эту статью можно прочитать по адресу **www.mosha.com/msolap/articles/MDXForEveryone.htm**.

Безопасность служб Analysis Services SQL Server

Аспекты безопасности служб SQL Server Analysis Services такие же, как и аспекты безопасности компонента Database Engine. Это означает, что службы SSAS поддерживают те же самые возможности — авторизацию и проверку подлинности, — что и компонент Database Engine, но в более сжатом виде.

В процессе авторизации определяются пользователи, которые имеют санкционированный доступ к службам SSAS. Этот вопрос тесно связан с авторизацией в операционной системе. Иными словами, функции авторизации пользователей служб SSAS основаны на правах доступа, предоставляемых пользователю операционной системой Windows.

Количество пользователей, которые могут выполнять функции администрирования служб SSAS, можно ограничить. Можно также указывать конечных пользователей, которые могут обращаться к данным и определять типы операций, которые они могут выполнять. Кроме этого, можно также управлять доступом пользователей к разным уровням данным, таким как уровень куба, измерения и ячейки куба. Такое управление реализуется посредством применения ролей.

Как уже упоминалось, роль базы данных определяет группу пользователей базы данных, которые могут обращаться к определенным объектам базы данных. Каждая роль определяется на уровне базы данных служб SSAS, а затем назначается кубам. После этого членство в этой роли присваивается отдельным пользователям и другим ролям.

Чтобы создать роль, щелкните правой кнопкой мыши папку **Roles** в панели обозревателя решений и выберите в контекстном меню пункт **New Role**. В открывшемся окне **New Role** на странице **General** измените имя по умолчанию роли на требуемое имя роли. В этом же окне (рис. 22.17) на его разных вкладках можно настроить разные параметры этой роли.

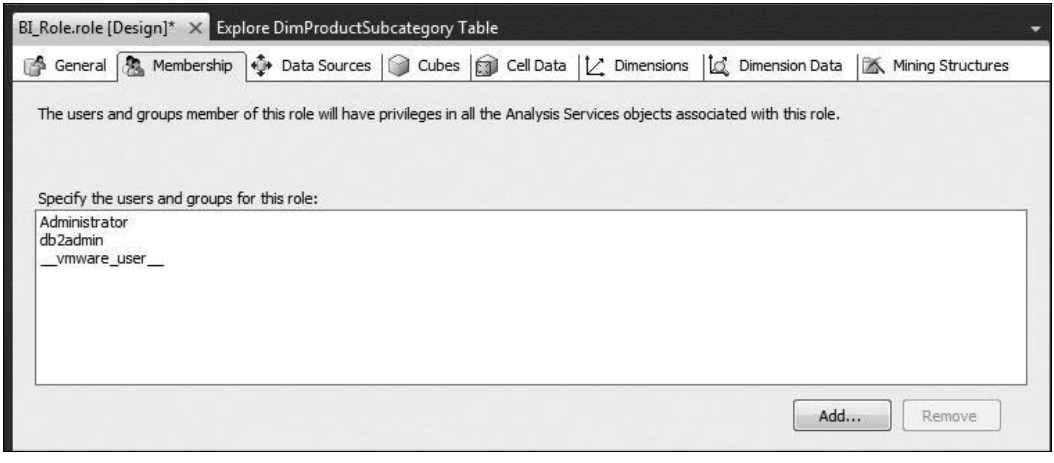


Рис. 22.17. Окно для создания новой роли для служб SSAS и настройки ее параметров

На вкладке **Membership** можно указать всех пользователей, которым присвоено членство в этой роли. (На рис. 22.17 показано три пользователя, добавленных в членство роли `BI_Role`.) На вкладках **Data Sources** и **Cubes** указываются источники данных, т. е. кубы, которые могут использовать члены роли. На вкладке **Cell Data** можно присвоить разрешения для определенной ячейки. На вкладках **Dimensions** и **Dimensions Data** указываются измерения (данные измерений), к которым члены роли могут иметь доступ.

Резюме

Посредством служб SQL Server Analysis Services корпорация Microsoft предоставляет набор инструментальных средств для выполнения начального и среднего уровня анализа данных, находящихся в хранилищах данных. Основным компонентом служб SSAS является средство BIDS (Business Intelligence Development Studio), которое основано на Visual Studio и предоставляет пользователям легкий и удобный способ проектировать и разрабатывать хранилища данных и киоски данных.

Службы SSAS предоставляют мастеров почти для каждой задачи, выполняющейся в процессе проектирования и реализации хранилища данных. Например, мастер источников данных Data Source Wizard позволяет задать один или несколько источников данных, а мастер кубов Cube Wizard применяется для создания многомерных кубов для хранения агрегированных данных. Для импортирования таблиц применяется мастер Table Import Wizard, а мастер Aggregation Design Wizard применяется для проектирования и создания оптимальных агрегаций.

Доставку аналитической информации пользователям можно осуществлять с помощью нескольких различных интерфейсов, наиболее важными из которых являются среда SQL Server Management Studio, язык MDX и инструментальный набор OLE DB for OLAP.

В следующей главе мы рассмотрим расширения SQL/OLAP для языка Transact-SQL.

Упражнения

Упражнение 22.1

Используя среду Management Studio и куб `BI_Cube2`, определите объем продаж по всем продуктам для клиента номер 11111.

Упражнение 22.2

Используя среду Management Studio и куб `BI_Cube2`, определите общий объем продаж продукта номер 14 всем клиентам.

Глава 23

Бизнес-аналитика и Transact-SQL

- **Конструкция окна**
- **Расширения предложения *GROUP BY***
- **Функции запросов OLAP**
- **Стандартные и нестандартные аналитические функции**

Вы когда-либо пытались составить запрос Transact-SQL, который вычисляет относительное изменение в значениях последних двух кварталов? Или запрос для вычисления нарастающих сумм или скользящих агрегаций? Если вы когда-либо пытались решить эти задачи, то должны знать, насколько это трудно. Но теперь решать такие задачи самому нет необходимости. В стандарте SQL:1999 был принят набор функций оперативной аналитической обработки (OLAP), позволяющий с легкостью выполнять такие вычисления, а также многие другие вычисления, которые раньше поддавались реализации с очень большой сложностью. Эта часть стандарта SQL называется SQL/OLAP и содержит все функции и операторы, применяемые для анализа данных.

Использование функций OLAP дает пользователям несколько преимуществ, включая следующие:

- пользователи, обладающие стандартными знаниями языка SQL, могут с легкостью указать требуемые им вычисления;
- системы баз данных, такие как Database Engine, могут выполнять эти вычисления намного более эффективно;
- поскольку для этих функций существует стандартная спецификация, для поставщиков инструментальных средств и приложений их использование обходится намного экономнее;
- почти все аналитические функции, предложенные в стандарте SQL:1999, реализуются в промышленных системах баз данных одинаково. Поэтому запросы

SQL/OLAP можно переносить с одной системы на другую, не требуя никаких изменений в коде.

Компонент Database Engine предоставляет большое количество расширений для инструкции `SELECT`, которые можно использовать в основном для аналитических операций. Определение некоторых из этих расширений соответствует стандарту SQL:1999, а некоторых — нет. В следующих далее разделах рассматриваются как стандартные, так и нестандартные функции и операторы SQL/OLAP.

Наиболее важным расширением языка Transact-SQL в плане анализа данных является конструкция окна, которая рассматривается в следующем разделе.

Конструкция окна

Окно (относящееся к SQL/OLAP) определяет секционированный набор строк, к которому применяется функция. Количество принадлежащих окну строк определяется динамически в зависимости от требований пользователя. Конструкция окна задается посредством предложения `OVER`.

Стандартизованная конструкция окна имеет три основные части:

- секционирование;
- упорядочение;
- группирование агрегаций.

ПРИМЕЧАНИЕ

На данном этапе компонент Database Engine не поддерживает группирование агрегаций, поэтому эта возможность здесь не обсуждается.

Прежде чем погружаться в конструкцию окна и ее составляющие, взглянем на таблицу, которая будет использоваться в примерах. В примере 23.1 создается таблица `project_dept` (содержимое которой приведено в табл. 23.1), которая используется в этой главе для демонстрации расширений Transact-SQL, касающихся SQL/OLAP.

Содержимое таблицы `project_dept`

dept_name	emp_cnt	budget	date_month
Research	5	50000	01.01.2007
Research	10	70000	02.01.2007
Research	5	65000	07.01.2007
Accounting	5	10000	07.01.2007
Accounting	10	40000	02.01.2007
Accounting	6	30000	01.01.2007
Accounting	6	40000	02.01.2008

(окончание)

dept_name	emp_cnt	budget	date_month
Marketing	6	100000	01.01.2008
Marketing	10	180000	02.01.2008
Marketing	3	100000	07.01.2008
Marketing	NULL	120000	01.01.2008

Пример 23.1. Создание таблицы project_dept

```
USE sample;
CREATE TABLE project_dept
      (dept_name CHAR(20) NOT NULL,
      emp_cnt INT,
      budget FLOAT,
      date_month DATE);
```

Таблица project_dept содержит информацию о нескольких отделах и количестве работающих в каждом из них сотрудников в разные периоды времени, а также о бюджетах проектов, контролируемых каждым отделом. В примере 23.2 приводится листинг запроса для вставки в таблицу project_dept строк, перечисленных в табл. 23.1.

Пример 23.2. Запрос для заполнения таблицы project_dept

```
USE sample;
INSERT INTO project_dept VALUES
   ('Research', 5, 50000, '01.01.2007');
INSERT INTO project_dept VALUES
   ('Research', 10, 70000, '02.01.2007');
INSERT INTO project_dept VALUES
   ('Research', 5, 65000, '07.01.2007');
   INSERT INTO project_dept VALUES
   ('Accounting', 5, 10000, '07.01.2007');
INSERT INTO project_dept VALUES
   ('Accounting', 10, 40000, '02.01.2007');
INSERT INTO project_dept VALUES
   ('Accounting', 6, 30000, '01.01.2007');
INSERT INTO project_dept VALUES
   ('Accounting', 6, 40000, '02.01.2008');
INSERT INTO project_dept VALUES
   ('Marketing', 6, 100000, '01.01.2008');
INSERT INTO project_dept VALUES
   ('Marketing', 10, 180000, '02.01.2008');
```

```
INSERT INTO project_dept VALUES
   ('Marketing', 3, 100000, '07.01.2008');
INSERT INTO project_dept VALUES
   ('Marketing', NULL, 120000, '01.01.2008');
```

Секционирование

Посредством секционирования (partitioning) результирующий набор запроса можно разбить на группы таким образом, что каждая строка секции будет отображаться отдельно. Если секционирование не задано, то весь набор строк составляет одну секцию. Хотя секционирование может выглядеть подобно группированию посредством использования предложения GROUP BY, но это не то же самое. Предложение GROUP BY сворачивает строки секции в одну строку, тогда как секционирование в конструкции окна просто организовывает строки в группу, не сворачивая их при этом. В следующих двух примерах показывается разница между секционированием с помощью конструкции окна и группированием, используя предложение GROUP BY. Предположим, что требуется вычислить несколько разных агрегатных значений касательно сотрудников каждого отдела. В примере 23.3 показано использование предложения OVER совместно с предложением PARTITION BY для создания секций. В частности, используя конструкцию окна, создаются разделы согласно значениям столбца dept_name и вычисляется общее и среднее значения для отделов Accounting и Research.

Пример 23.3. Создание секций и выполнение по ним агрегатных вычислений

```
USE sample;
SELECT dept_name, budget,
   SUM(emp_cnt) OVER(PARTITION BY dept_name) AS emp_cnt_sum,
   AVG(budget) OVER(PARTITION BY dept_name) AS budget_avg
      FROM project_dept
      WHERE dept_name IN ('Accounting', 'Research');
```

Этот запрос возвращает следующий результат:

dept_name	budget	Emp_cnt_sum	budget_avg
Accounting	10000	27	30000
Accounting	40000	27	30000
Accounting	30000	27	30000
Accounting	40000	27	30000
Research	50000	20	61666.6666666667
Research	70000	20	61666.6666666667
Research	65000	20	61666.6666666667

В примере 23.3 посредством предложения OVER определяется конструкция окна, а внутри его параметр PARTITION BY разделяет результирующий набор на секции.

(Обе секции в примере 23.3 группируются, используя значения в столбце dept_name.) Наконец, к полученным секциям применяется агрегатная функция. (Запрос в примере 23.3 вычисляет два агрегатных значения — сумму значений столбца emp_cnt и среднее значение бюджетов.) Опять же, как можно видеть в результирующем наборе запроса примера, секционирование организовывает строки в группы, но не сворачивает их.

В примере 23.4 показан подобный запрос, но с использованием предложения GROUP BY. Этот запрос группирует значения столбца dept_name для отделов Accounting и Research и вычисляется общее и среднее значения для этих двух групп.

Пример 23.4. Группирование посредством использования предложения GROUP BY

```
USE sample;
SELECT dept_name, SUM(emp_cnt) AS cnt, AVG(budget) AS budget_avg
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Research')
   GROUP BY dept_name;
```

Результат выполнения этого запроса будет следующим:

Dept_name	cnt	budget_avg
Accounting	27	30000
Research	20	61666.6666666667

Как уже упоминалось, при использовании предложения GROUP BY каждая группа сворачивается в одну строку.

ПРИМЕЧАНИЕ

Между предложениями OVER и GROUP BY есть еще одна значительная разница. Как можно видеть в примере 23.3, при использовании предложения OVER соответствующий список выборки SELECT может содержать имя любого столбца таблицы. Это очевидно, поскольку секционирование организовывает строки в группы, не сворачивая их в одну. Если же в список выборки SELECT в примере 23.4 добавить столбец budget, то запрос возвратит ошибку.

Упорядочение и кадрирование

Упорядочение внутри конструкции окна подобно упорядочению в запросе. Во-первых, требуемый порядок строк в результирующем наборе задается с помощью предложения ORDER BY. Во-вторых, оно включает список ключей сортировки с указанием порядка сортировки — возрастающего или убывающего. Наиболее важное различие заключается в том, что упорядочение в окне применяется только *внутри* каждой секции.

В отличие от SQL Server 2008, SQL Server 2012 поддерживает упорядочение внутри конструкции окна для агрегатных функций. Иными словами, теперь предложение OVER для агрегатных функций может также содержать предложение ORDER BY. Такое применение предложения ORDER BY показано в примере 23.5. В частности, посредством конструкции окна строки таблицы project_dept секционируются, используя значения столбца dept_name, и строки каждой секции сортируются, используя значения столбца budget.

Пример 23.5. Использование предложения ORDER BY в предложении OVER

```
USE sample;
SELECT dept_name, budget, emp_cnt,
      SUM(budget) OVER(PARTITION BY dept_name ORDER BY budget)
                                                  AS sum_dept
      FROM project_dept;
```

Запрос в примере 23.5, который обычно называется нарастающим агрегированием, а в данном случае — нарастающим суммированием, использует предложение ORDER BY для указания упорядочения внутри определенной секции. Эту функциональность можно расширить с помощью кадрирования. *Кадрирование* (framing) означает, что результат можно сузить еще больше, используя две граничные точки, которые ограничивают набор строк определенным подмножеством, как показано в примере 23.6.

Пример 23.6. Кадрирование результирующего набора

```
USE sample;
SELECT dept_name, budget, emp_cnt,
   SUM(budget) OVER(PARTITION BY dept_name ORDER BY budget
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      AS sum_dept
   FROM project_dept;
```

В примере 23.6 используется два предложения: UNBONDED PRECEDING и CURRENT ROW для задания граничных точек выбранных строк. Это означает, что на основе порядка значений столбца budget выбираемое подмножество строк не имеет нижней граничной точки, а верхней — является текущая строка. Результирующий набор содержит все 11 строк.

Кроме границ кадра, использованных в примере 23.6, можно применить и другие границы. В частности, предложение UNBOUNDED FOLLOWING означает, что заданный кадр не имеет верхней граничной точки. Кроме этого, обе граничные точки можно указать посредством смещение от текущей строки. Иными словами, можно указать *n* строк до текущей строки и *n* строк после нее с помощью предложений n PRECEDING и n FOLLOWING соответственно. Таким образом, следующий кадр задает все три строки — текущую, предшествующую и следующую:

```
   ROWS BETWEEN 1 PRECEDING and 1 FOLLOWING
```

В SQL Server 2012 также вводится две новые функции, связанные с кадрированием: LEAD и LAG. Функция LEAD может вычислять выражение по следующим далее строкам (т. е. строкам, которые будут идти после текущей строки). Иными словами, функция LEAD возвращает значение следующей по порядку *n*-й строки. Эта функция имеет три параметра. В первом параметре указывается имя столбца для вычисления следующей строки, во втором указывается смещение следующей строки относительно текущей, а последний содержит значение, возвращаемое, когда смещение указывает на строку, находящуюся вне пределов секции. (Семантика функции LAG подобна: она возвращает значение предшествующей *n*-й строки.)

В примере 23.6 используется предложение ROWS для ограничения строк внутри секции, указывая физическое количество строк, предшествующих и следующих за текущей строкой. Альтернативно, SQL Server 2012 поддерживает предложение RANGE, которое логически ограничивает количество строк в секции. Иными словами, при использовании предложения ROWS указывается точное количество строк на основе определенного кадра. С другой стороны, предложение RANGE не указывает точного количества строк, поскольку заданный кадр может также содержать дубликаты.

Используя несколько столбцов из таблицы, в запросе можно создавать разные схемы секционирования, как это показано в примере 23.7. Здесь посредством конструкции окна создается две секции для отделов Accounting и Research: используя значения столбца budget для одной секции, а для второй — значения столбца dept_name. Для первой секции вычисляются суммы, а для второй — средние значения.

Пример 23.7. Запрос для создания двух секций по двум столбцам

```
USE sample;
SELECT dept_name, CAST(budget AS INT) AS budget,
   SUM(emp_cnt) OVER(PARTITION BY budget) AS emp_cnt_sum,
   AVG(budget) OVER(PARTITION BY dept_name) AS budget_avg
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Research');
```

Этот запрос возвращает следующий результат:

dept_name	Budget	emp_cnt_sum	budget_avg
Accounting	10000	5	30000
Accounting	30000	6	30000
Accounting	40000	16	30000
Accounting	40000	16	30000
Research	50000	5	61666.6666666667
Research	65000	5	61666.6666666667
Research	70000	10	61666.6666666667

Запрос в примере 23.7 имеет две разные схемы сегментирования: одна по значениям столбца budget, а вторая по значениям столбца dept_name. Первая секция исполь-

зуется для вычисления количества сотрудников по отношению к отделам с одинаковыми бюджетами, а вторая — для вычисления среднего значения бюджетов отделов, сгруппированных по их именам.

В примере 23.8 показано использование выражения NEXT VALUE FOR инструкции CREATE SEQUENCE для управления порядком создания значений, используя предложение OVER. (Описание инструкции CREATE SEQUENCE см. в *главе 6*.)

Пример 23.8. Управление порядком генерирования значений

```
USE sample;
CREATE SEQUENCE Seq START WITH 1 INCREMENT BY 1;
GO
CREATE TABLE T1 (col1 CHAR(10), col2 CHAR(10));
GO
INSERT INTO dbo.T1(col1, col2)
   SELECT NEXT VALUE FOR Seq OVER(ORDER BY dept_name ASC), budget
   FROM (SELECT dept_name, budget
      FROM project_dept
      ORDER BY budget, dept_name DESC
      OFFSET 0 ROWS FETCH FIRST 5 ROWS ONLY) AS D;
```

Содержимое создаваемой этим запросом таблицы T1 будет таким:

col1	col2
1	10000
2	30000
3	40000
4	40000
5	50000

Первые две инструкции запроса создают последовательность Seq и вспомогательную таблицу T1. Следующая за ними инструкция INSERT с помощью подзапроса отфильтровывает пять отделов с самыми большими бюджетами и генерирует для них значения последовательности. Для этого используется комбинация параметров OFFSET/FETCH, которая рассматривается в *главе 6*. (Несколько других примеров с использованием комбинации OFFSET/FETCH приведено в одноименном разделе далее в этой главе.)

Расширения предложения *GROUP BY*

В языке Transact-SQL предложение GROUP BY расширяется следующими операторами и функциями:

- оператором CUBE;
- оператором ROLLUP;

- функциями группирования;
- наборами группирования.

Эти операторы и функции рассматриваются в последующих разделах.

Оператор *CUBE*

В этом разделе рассматриваются различия между выполнением группирования посредством использования самого предложения GROUP BY и выполнением группирования, используя это предложение в комбинации с операторами CUBE и ROLLUP. Основная разница заключается в том, что предложение GROUP BY группирует выбранный набор строк для получения набора сводных строк по значениям одного или нескольких столбцов или выражений. Операторы CUBE и ROLLUP предоставляют дополнительные сводные строки для сгруппированных данных. Эти сводные строки также называются *многомерными сводками*.

Эти различия показаны в следующих двух примерах. В примере 23.9 предложение GROUP BY применяется к строкам таблицы project_dept согласно двум критериям: dept_name и emp_cnt. В частности, строки таблицы project_dept со значениями Accounting и Research для столбца dept_name группируются по столбцам dept_name и emp_cnt.

Пример 23.9. Группирование строк по значениям столбцов с помощью предложения GROUP BY

```
USE sample;
SELECT dept_name, emp_cnt, SUM(budget) sum_of_budgets
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Research')
   GROUP BY dept_name, emp_cnt;
```

Результат выполнения этого запроса будет следующим:

dept_name	emp_cnt	Sum_of_budgets
Accounting	5	10000
Research	5	115000
Accounting	6	70000
Accounting	10	40000
Research	10	70000

Разница, создаваемая с использованием дополнительного оператора CUBE, приведена в примере 23.10 и его результирующем наборе. В частности, строки таблицы project_dept со значениями Accounting и Research для столбца dept_name группируются по столбцам dept_name и emp_cnt, а также дополнительно отображаются все возможные сводные строки.

Пример 23.10. Группирование с использованием оператора CUBE

```
USE sample;
SELECT dept_name, emp_cnt, SUM(budget) sum_of_budgets
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Research')
   GROUP BY CUBE (dept_name, emp_cnt);
```

Этот запрос возвращает следующий результат:

dept_name	emp_cnt	sum_of_budgets
Accounting	5	10000
Research	5	115000
NULL	5	125000
Accounting	6	70000
NULL	6	70000
Accounting	10	40000
Research	10	70000
NULL	10	110000
Accounting	NULL	120000
Research	NULL	185000
NULL	NULL	305000

Основным отличием между этими двумя примерами является то, что результирующий набор запроса из примера 23.9 содержит только значения, относящиеся к группам, тогда как результирующий набор примера 23.10 дополнительно содержит все возможные сводные строки. (Так как оператор CUBE выбирает все возможные комбинации групп и сводных строк, количество строк всегда будет одинаковым, независимо от порядка столбцов в предложении GROUP BY.) В качестве заполнителя для значений в ненужных столбцах сводных строк используется значение NULL. Например, следующая строка в результирующем наборе:

```
NULL NULL 305000
```

содержит общую сумму (т. е. сумму всех бюджетов всех проектов в таблице), тогда как строка:

```
NULL 5    125000
```

содержит сумму всех бюджетов всех проектов, в которых участвует ровно пять сотрудников.

ПРИМЕЧАНИЕ

Синтаксис оператора CUBE в примере 23.10 соответствует стандартизованному синтаксису этого оператора. Благодаря обратной совместимости, компонент Database Engine также поддерживает синтаксис старого типа:

```
USE sample;
SELECT dept_name, emp_cnt, SUM(budget) sum_of_budgets
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Research')
   GROUP BY dept_name, emp_cnt
   WITH CUBE;
```

Оператор *ROLLUP*

В отличие от оператора CUBE, который возвращает все возможные комбинации групп и сводных строк, при использовании оператора ROLLUP иерархия группы определяется порядком, в котором указаны группируемые столбцы. Использование оператора ROLLUP показано в примере 23.11. В частности, строки таблицы project_dept со значениями Accounting и Research для столбца dept_name группируются по столбцам dept_name и emp_cnt, а также дополнительно отображаются все возможные сводные строки для столбца dept_name.

Пример 23.11. Использование оператора ROLLUP

```
USE sample;
SELECT dept_name, emp_cnt, SUM(budget) sum_of_budgets
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Research')
   GROUP BY ROLLUP (dept_name, emp_cnt);
```

Этот запрос возвращает следующие результирующие наборы:

dept_name	emp_cnt	sum_of_budgets
Accounting	5	10000
Accounting	6	70000

dept_name	emp_cnt	sum_of_budgets
Accounting	10	40000
Accounting	NULL	120000
Research	5	115000
Research	10	70000
Research	NULL	185000
NULL	NULL	305000

Как можно видеть в результирующем наборе запроса примера 23.11, выбрано меньшее количество строк, чем в примере с оператором CUBE. Это объясняется тем, что сводные строки используются только для первого столбца в предложении GROUP BY ROLLUP.

ПРИМЕЧАНИЕ

В примере 23.11 используется стандартизованный синтаксис оператора ROLLUP. Синтаксис старого типа для этого оператора подобен синтаксису для оператора CUBE, который показан в примечании в конце предыдущего раздела.

Функции группирования

Как уже упоминалось, в качестве заполнителя значений в ненужных столбцах результирующих наборов операторов CUBE и ROLLUP используется значение NULL. В таком случае не представляется возможным различить значение NULL, используемое в этом контексте, от значения NULL, применяемого обычным образом. Язык Transact-SQL поддерживает две стандартные функции группирования, которые позволяют разрешить эту проблему неопределенности, а именно функции:

- GROUPING;
- GROUPING_ID.

Эти две функции подробно рассматриваются в следующих подразделах.

Функция *GROUPING*

Функция GROUPING возвращает значение 1, если значение NULL в результирующем наборе относится к операторам CUBE или ROLLUP, и значение 0, если оно представляет группу значений NULL. Использование функции GROUPING демонстрируется в примере 23.12. В частности, эта функция уточняет, какие значения в результирующем наборе инструкции SELECT обозначают сводные строки

Пример 23.12. Использование функции GROUPING для обозначения сводных строк

```
USE sample;
SELECT dept_name, emp_cnt, SUM(budget) sum_b, GROUPING(emp_cnt) gr
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Marketing')
   GROUP BY ROLLUP (dept_name, emp_cnt);
```

Этот запрос возвращает следующий результат:

dept_name	emp_cnt	sum_b	gr
Accounting	5	10000	0
Accounting	6	70000	0
Accounting	10	40000	0
Accounting	NULL	120000	1
Marketing	NULL	120000	0
Marketing	3	100000	0
Marketing	6	100000	0

```
Marketing          10              180000          0
Marketing          NULL            500000          1
NULL               NULL            620000          1
```

В столбце группирования gr можно видеть, что он содержит значения 0 и 1. Значение 1 указывает, что соответствующее значение NULL в столбце emp_cnt обозначает суммарное значение, тогда как значение 0 указывает, что значение NULL в этом столбце обозначает само себя.

Функция *GROUPING_ID*

Функция GROUPING_ID вычисляет уровень группирования. Эту функцию можно использовать только в списке выборки столбцов инструкции SELECT, предложении HAVING или в предложении ORDER BY, когда применяется предложение GROUP BY.

Использование функции GROUPING_ID показано в примере 23.13.

Пример 23.13. Использование функции GROUPING_ID

```
USE sample;
SELECT dept_name, YEAR(date_month) , SUM(budget),
GROUPING_ID (dept_name, YEAR(date_month)) AS gr_dept
   FROM project_dept
   GROUP BY ROLLUP (dept_name, YEAR(date_month));
```

Результат выполнения этого запроса будет следующим:

dept_name	date_month	budget	gr_dept
Accounting	2007	80000	0
Accounting	2008	40000	0
Accounting	NULL	120000	1
Marketing	2008	500000	0
Marketing	NULL	500000	1
Research	2007	185000	0
Research	NULL	185000	1
NULL	NULL	805000	3

Функция GROUPING_ID подобна функции GROUPING, но она особенно полезна для определения суммирования нескольких столбцов, как это показано в примере 23.13. Эта функция возвращает целое число, полученное при преобразовании в двоичный формат, представляет собой последовательность значений 1 и 0, которые отражают суммирование каждого столбца, возвращаемое функцией в параметре. Например, значение 3 в столбце gr_dept последней строки результирующего набора означает, что суммированию подверглись оба столбца — dept_name и date_month. (Двоичным эквивалентом десятичного числа 3 будет 11_2.)

Наборы группирования

Наборы группирования являются расширением предложения GROUP BY, которое позволяет определять несколько групп в одном запросе. Для реализации наборов группирования применяется оператор GROUPING SETS. Использование этого оператора показано в примере 23.14. В этом примере вычисляется сумма бюджетов отделов Accounting и Research, сначала используя комбинацию значений столбцов dept_name и emp_cnt, а затем используя значения одного столбца dept_name.

Пример 23.14. Использование оператора GROUPING SETS

```
USE sample;
SELECT dept_name, emp_cnt, SUM(budget) sum_budgets
   FROM project_dept
   WHERE dept_name IN ('Accounting', 'Research')
   GROUP BY GROUPING SETS ((dept_name, emp_cnt),(dept_name));
```

Этот запрос возвращает следующий результат:

dept_name	emp_cnt	sum_budgets
Accounting	5	10000
Accounting	6	70000
Accounting	10	40000
Accounting	NULL	120000
Research	5	115000
Research	10	70000
Research	NULL	185000

Как можно судить по результирующему набору примера 23.14, в запросе используется два разных группирования для вычисления сумм бюджетов: первое с использованием значений столбцов dept_name и emp_count, а второе с помощью значений одного столбца dept_name. Первые три строки результата содержат суммы бюджетов для трех разных группирований первых двух столбцов: Accounting, 5; Accounting, 6; и Accounting, 10. В четвертом столбце отображается сумма бюджетов для всех отделов Accounting. Последние три строки содержат подобные результаты для отдела Research.

Вы можете использовать последовательности наборов группирования, чтобы заменить операторы ROLLUP и CUBE. Например, следующая последовательность наборов группирования:

```
GROUP BY GROUPING SETS ((dept_name, emp_cnt), (dept_name), ())
```

эквивалентна следующему предложению ROLLUP:

```
GROUP BY ROLLUP (dept_name, emp_cnt)
```

А эта последовательность наборов группирования:

```
GROUP BY GROUPING SETS ((dept_name, emp_cnt),
                        (emp_cnt, dept_name), (dept_name), ())
```

эквивалентна следующему предложению CUBE:

```
GROUP BY CUBE (dept_name, emp_cnt)
```

Функции запросов OLAP

Язык Transact-SQL поддерживает следующие две группы функций, которые классифицируются, как функции запросов OLAP:

- ранжирующие функции;
- статистические агрегатные функции.

Эти категории функций подробно рассматриваются в следующих подразделах.

ПРИМЕЧАНИЕ

Рассматриваемая ранее функция GROUPING также принадлежит к категории функций OLAP.

Ранжирующие функции

Ранжирующие функции возвращают ранжирующее значение для каждой строки в секции. Язык Transact-SQL поддерживает следующие ранжирующие функции:

- RANK;
- DENSE_RANK;
- ROW_NUMBER.

Использование функции RANK показано в примере 23.15. В данном запросе определяются все отделы с бюджетом, не превышающим 3000, и результаты выводятся в нисходящем порядке.

Пример 23.15. Использование ранжирующей функции RANK

```
USE sample;
SELECT RANK() OVER(ORDER BY budget DESC) AS rank_budget,
                                dept_name, emp_cnt, budget
   FROM project_dept
   WHERE budget <= 30000;
```

Результат выполнения этого запроса:

rank_budget	dept_name	emp_cnt	budget
1	Accounting	6	30000
2	Accounting	5	10000

В примере 23.15 функция RANK возвращает число (в первом столбце результирующего набора), указывающее ранг строки в секции. Посредством предложения OVER строки результирующего набора сортируются по столбцу budget в нисходящем порядке. (В данном примере предложение PARTITION BY опущено. По этой причине весь результирующий набор состоит только из одной секции.)

ПРИМЕЧАНИЕ

Функция RANK использует логическое агрегирование. Иными словами, если две или большее количество строк результирующего набора имеют одинаковое значение в столбце, по которому выполняется упорядочение, их ранг будет одинаковым. Ранг последующей строки будет равным количеству предшествующих рангов плюс один. Поэтому возвращаемая последовательность рангов может быть с разрывами (т. е. после ранга 3 может следовать, например, ранг 5), если две или большее количество строк имеют одинаковый ранг.

В примере 23.16 показано использование двух других ранжирующих функций — DENSE_RANK и ROW_NUMBER. Данный запрос выбирает все отделы, бюджеты которых не превышают 40 000, и отображает без промежутков в ранжировании ранг и последовательный номер каждой строки результирующего набора.

Пример 23.16. Использование ранжирующих функций DENSE_RANK и ROW_NUMBER

```
USE sample;
SELECT DENSE_RANK() OVER(ORDER BY budget DESC) AS dense_rank,
      ROW_NUMBER() OVER(ORDER BY budget DESC) AS row_number,
      dept_name, emp_cnt, budget
   FROM project_dept
   WHERE budget <= 40000;
```

Этот запрос возвращает следующий результат:

dense_rank	row_number	dept_name	emp_cnt	budget
1	1	Accounting	10	40000
1	2	Accounting	6	40000
2	3	Accounting	6	30000
3	4	Accounting	5	10000

Первые два столбца в результирующем наборе примера 23.16 содержат значения, возвращаемые функциями DENSE_RANK и ROW_NUMBER соответственно. Вывод функции DENSE_RANK подобен выводу функции RANK (см. пример 23.15), но без промежутков в ранжировании в случае, если две или больше строки имеют одинаковый ранг.

Назначение функции ROW_NUMBER должно быть очевидным: она возвращает последовательные номера строк результирующего набора, начиная с номера 1 для первой строки.

В последних двух примерах показано использование предложения OVER для определения упорядочения результирующего набора. Как уже упоминалось, это предложение можно также использовать, чтобы разделить на группы (секции) результирующий набор, созданный предложением FROM, а затем применить агрегатную или ранжирующую функцию к каждой секции по отдельности.

В примере 23.17 показано применение функции RANK в секциях. В частности, используя конструкцию окна, строки таблицы project_dept секционируются в соответствии со значениями столбца date_month, а строки в каждой секции сортируются и отображаются в порядке возрастания.

Пример 23.17. Использование функции RANK в секциях

```
USE sample;
SELECT date_month, dept_name, emp_cnt, budget,
      RANK() OVER(PARTITION BY date_month ORDER BY emp_cnt desc) AS rank
   FROM project_dept;
```

Результат выполнения этого запроса:

date_month	dept_name	emp_cnt	budget	rank
2007-01-01	Accounting	6	30000	1
2007-01-01	Research	5	50000	2
2007-02-01	Research	10	70000	1
2007-02-01	Accounting	10	40000	1
2007-07-01	Research	5	65000	1
2007-07-01	Accounting	5	10000	1
2008-01-01	Marketing	6	100000	1
2008-01-01	Marketing	NULL	120000	2
2008-02-01	Marketing	10	180000	1
2008-02-01	Accounting	6	40000	2
2008-07-01	Marketing	3	100000	1

Результирующий набор примера 23.17 сначала секционируется (разделяется) на восемь групп в соответствии со значениями столбца date_month, после чего к каждой секции применяется функция RANK. По сравнению с запросом примера 23.5 запрос этого примера работает должным образом, в то время как запрос примера 23.5 выдает ошибку, хотя в обоих запросах используется одна и та же конструкция окна. Как упоминалось ранее, в настоящее время язык Transact-SQL поддерживает предложение OVER для агрегатных функций только с предложением PARTITION BY, но в случае ранжирующих функций система поддерживает общий стандартный синтаксис SQL с предложениями PARTITION BY и ORDER BY.

Статистические агрегатные функции

Статистические агрегатные функции были представлены в *главе 6*. Существует четыре таких функции:

- VAR — вычисляет статистическую дисперсию всех значений в столбце или выражении;
- VARP — вычисляет статистическую дисперсию для заполнения всех значений столбца или выражения;
- STDEV AS rank — вычисляет статистическое стандартное отклонение совокупности всех значений столбца или выражения. (Статистическое стандартное отклонение вычисляется как квадратный корень соответствующей статистической дисперсии);
- STDEVP — вычисляет статистическое стандартное отклонение совокупности всех значений столбца или выражения.

Статистические агрегатные функции можно использовать как с конструкцией окна, так и без него. В примере 23.18 показано использование функций VAR и STDEV с конструкцией окна для вычисления статистической дисперсии и стандартного отклонения бюджетов в разделах, созданных, используя значения столбца dept_name.

Пример 23.18. Использование функций VAR и STDEV с конструкцией окна

```
USE sample;
SELECT dept_name, budget,
   VAR(budget) OVER(PARTITION BY dept_name) AS budget_var,
   STDEV(budget) OVER(PARTITION BY dept_name) AS budget_stdev
      FROM project_dept
      WHERE dept_name in ('Accounting', 'Research');
```

Этот запрос возвращает следующий результат:

dept_name	budget	budget_var	budget_stdev
Accounting	10000	200000000	14142.135623731
Accounting	40000	200000000	14142.135623731
Accounting	30000	200000000	14142.135623731
Accounting	40000	200000000	14142.135623731
Research	50000	108333333,333333	10408.3299973306
Research	70000	108333333,333333	10408.3299973306
Research	65000	108333333,333333	10408.3299973306

Стандартные и нестандартные аналитические функции

Компонент Database Engine поддерживает следующие стандартные и нестандартные функции OLAP:

- TOP;
- OFFSET/FETCH;
- NTILE;
- PIVOT и UNPIVOT.

Вторая функция OFFSET/FETCH определена в стандарте SQL, тогда как остальные три являются расширениями языка Transact-SQL. Эти аналитические операторы и функции рассматриваются в последующих разделах.

Предложение *TOP*

Предложение TOP задает первые *n* строк результата запроса, которые нужно возвратить. Это предложение всегда следует использовать с предложением ORDER BY, поскольку результат такого запроса всегда хорошо определен, и его можно использовать в табличных выражениях. (Табличное выражение определяет экземпляр сгруппированного табличного результата.) Запрос с предложением TOP, но без предложения ORDER BY является недетерминированным, означая, что множественные выполнения запроса по одним и тем же данным не всегда будут возвращать один и тот же результирующий набор. В примере 23.19 показано применение этого выражения для выборки четырех проектов, которые имеют самые большие бюджеты.

Пример 23.19. Использование предложения TOP в запросе

```
USE sample;
SELECT TOP (4) dept_name, budget
   FROM project_dept
   ORDER BY budget DESC;
```

Результат выполнения этого запроса:

dept_name	budget
Marketing	180000
Marketing	120000
Marketing	100000
Marketing	100000

Как можно судить по примеру 23.19, предложение TOP является частью списка выборки столбцов в инструкции SELECT и указывается в начале этого списка.

ПРИМЕЧАНИЕ

Входное значение предложения TOP следует заключить в круглые скобки, поскольку система поддерживает в качестве ввода любое независимое выражение.

Предложение TOP является нестандартным решением языка Transact-SQL, которое применяется для отображения ранжирования *n* верхних строк таблицы. Запрос, эквивалентный запросу примера 23.19, можно создать с помощью конструкции окна и функции RANK. Такой запрос, который выбирает четыре проекта с наибольшими бюджетами, показан в примере 23.20.

Пример 23.20. Реализация результатов предложения TOP посредством функции RANK

```
USE sample;
SELECT dept_name, budget
   FROM (SELECT dept_name, budget,
         RANK() OVER (ORDER BY budget DESC) AS rank_budget
      FROM project_dept) part_dept
      WHERE rank_budget <= 4;
```

Предложение TOP также можно использовать с дополнительной опцией PERCENT. В этом случае запрос возвращает *n* процентов строк результирующего набора. А дополнительная опция WITH TIES задает возвращение дополнительных строк, если значения их столбца (или столбцов) ORDER BY такие же, как и у последней строки, входящей в отображаемый набор. Применение параметров PERCENT и WITH TIES для выборки верхних 25% строк для отделов с наименьшим количеством сотрудников показано в примере 13.21.

Пример 23.21. Использование опций PERCENT и WITH TIES

```
USE sample;
SELECT TOP (25) PERCENT WITH TIES emp_cnt, budget
   FROM project_dept
ORDER BY emp_cnt ASC;
```

Этот запрос возвращает следующий результат:

emp_cnt	Budget
NULL	120000
3	100000
5	50000
5	65000
5	10000

Результат примера 23.21 содержит пять строк, поскольку существует три проекта, в которых занято по пять сотрудников.

Предложение TOP можно также применять с инструкциями UPDATE, DELETE и INSERT. Использование этого предложения с инструкцией UPDATE для выборки трех проектов с наибольшими бюджетами и уменьшения этих бюджетов на 10% показано в примере 23.22.

Пример 23.22. Использование предложения TOP с инструкцией UPDATE

```
USE sample;
UPDATE TOP (3) project_dept
   SET budget = budget * 0.9
   WHERE budget in (SELECT TOP (3) budget
   FROM project_dept
   ORDER BY budget desc);
```

В примере 23.23 показано использование предложения TOP с инструкцией DELETE для удаления четырех проектов с наименьшими бюджетами.

Пример 23.23. Применение инструкции TOP с инструкцией DELETE

```
USE sample;
DELETE TOP (4)
   FROM project_dept
   WHERE budget IN
      (SELECT TOP (4) budget FROM project_dept
         ORDER BY budget ASC);
```

В примере 23.23 предложение TOP сначала применяется в подзапросе для выборки четырех проектов с наименьшими бюджетами, а затем в инструкции DELETE для удаления этих проектов.

Комбинация предложений *OFFSET* и *FETCH*

В *главе 6* рассмотрено использование предложений OFFSET и FETCH для реализации на стороне сервера разбиения результатов запроса на страницы. Но это только одно из многих возможных применений комбинации этих параметров. В общем, комбинация OFFSET/FETCH позволяет отфильтровать несколько строк согласно указанному порядку. Кроме этого, можно указать количество строк результирующего набора, которые следует пропустить, и количество строк, которые следует возвратить. По этой причине комбинация предложений OFFSET/FETCH подобна предложению TOP. Но между ними также есть определенные различия.

- Комбинация предложений OFFSET/FETCH является стандартным способом для фильтрации данных, тогда как предложение TOP является расширением языка Transact-SQL. По этой причине существует возможность, что в будущем предложение TOP будет заменено этой комбинацией.

- Комбинация предложений OFFSET/FETCH более гибкая, чем предложение TOP в том отношении, что предложение OFFSET позволяет пропускать строки. (Компонент Database Engine не разрешает использовать предложение FETCH самостоятельно, без предложения OFFSET. Иными словами, даже если нет надобности пропускать строки, все равно необходимо использовать предложение OFFSET, присвоив ему значение 0.)
- Предложение TOP более гибкое, чем комбинация OFFSET/FETCH в том отношении, что его можно использовать в DML-инструкциях INSERT, UPDATE и DELETE (см. примеры 23.22 и 23.23).

В примерах 23.24 и 23.25 показано использование комбинации предложений OFFSET/FETCH с ранжирующей функцией ROW_NUMBER(). (Прежде чем выполнять эти примеры, нужно вновь заполнить таблицу project_table. Для этого нужно сначала удалить из нее все строки, а затем выполнить запрос из примера 23.2.)

Пример 23.24. Использование комбинации OFFSET/FETCH с функцией ROW_NUMBER()

```
USE sample;
SELECT date_month, budget, ROW_NUMBER()
   OVER (ORDER BY date_month DESC, budget DESC) as row_no
   FROM project_dept
   ORDER BY date_month DESC, budget DESC
   OFFSET 5 ROWS FETCH NEXT 4 ROWS ONLY;
```

Результат выполнения этого запроса:

date_month	Budget	row_no
2007-07-01	65000	6
2007-07-01	10000	7
2007-02-01	70000	8
2007-02-01	40000	9

Запрос из примера 23.24 отображает строки столбцов date_month и budget таблицы project_dept. (Первые пять строк результирующего набора пропускаются и выводятся только следующие четыре.) Кроме этого, в столбце row_no выводится номер соответствующей строки. Первая возвращаемая строка начинается с номера 6, поскольку номера присваиваются строкам в общем результирующем наборе, *до* его фильтрации с помощью комбинации предложений OFFSET/FETCH. (Комбинация OFFSET/FETCH является частью предложения ORDER BY и поэтому выполняется после списка выборки столбцов в инструкции SELECT, которая содержит функцию ROW_NUMBER. Иными словами, значения функции ROW_NUMBER определяются до применения комбинации OFFSET/FETCH.)

Если требуется, чтобы нумерация возвращаемых строк начиналась с 1, нужно модифицировать инструкцию SELECT, как это показано в примере 23.25.

Пример 23.25. Модификация инструкции SELECT для начала нумерации строк с 1

```
USE sample;
SELECT *, ROW_NUMBER()
   OVER (ORDER BY date_month DESC, budget DESC) as row_no
      FROM (SELECT date_month, budget
      FROM project_dept
      ORDER BY date_month DESC, budget DESC
      OFFSET 5 ROWS FETCH NEXT 4 ROWS ONLY) c;
```

Результат выполнения запроса примера 23.25 идентичен результату запроса примера 23.24, с тем отличием, что в нем нумерация строк начинается с 1. Причиной этого является то, что в примере 23.25 запрос с комбинацией предложений OFFSET/FETCH написан в виде табличного выражения внутри внешнего запроса с функцией ROW_NUMBER() в списке выборки столбцов инструкции SELECT. Таким образом, значения функции ROW_NUMBER() определяются до применения комбинации предложений OFFSET/FETCH.

Функция *NTILE*

Функция NTILE является ранжирующей функцией. Она распределяет строки секции в указанное количество групп. Для каждой строки функция NTILE возвращает номер группы, к которой она принадлежит. По этой причине данная функция обычно используется для организации строк в группы.

ПРИМЕЧАНИЕ

Функция NTILE разделяет данные только на основе подсчета количества значений.

Использование функции NTILE показано в примере 23.26.

Пример 23.26. Применение функции NTILE

```
USE sample;
SELECT dept_name, budget,
      CASE NTILE(3) OVER (ORDER BY budget ASC)
         WHEN 1 THEN 'Low'
         WHEN 2 THEN 'Medium'
         WHEN 3 THEN 'High'
      END AS groups
   FROM project_dept;
```

Этот запрос возвращает следующий результат:

dept_name	budget	groups
Accounting	10000	Low
Accounting	30000	Low
Accounting	40000	Low
Accounting	40000	Low
Research	50000	Medium
Research	65000	Medium
Research	70000	Medium
Marketing	100000	Medium
Marketing	100000	High
Marketing	120000	High
Marketing	180000	High

Сведение данных

Сведение данных заключается в преобразовании данных из строчной формы в столбцовую. Кроме этого, некоторые значения из исходной таблицы могут быть агрегированными перед созданием целевой таблицы.

Для сведения данных применяются два следующих оператора:

- PIVOT;
- UNPIVOT.

Эти операторы подробно рассматриваются далее в следующих подразделах.

Оператор *PIVOT*

Оператор PIVOT является нестандартным реляционным оператором, который поддерживается в языке Transact-SQL. С помощью этого оператора выражение, возвращающее табличное значение, можно преобразовывать в другую таблицу. Оператор PIVOT преобразовывает такое выражение, превращая однозначные значения одного столбца выражения в несколько столбцов выхода, и агрегирует все остальные значения столбца, которые требуется включить в вывод.

Для демонстрации работы оператора PIVOT используем таблицу project_dept_pivot, которая создается на основе таблицы project_dept, созданной в начале этой главы. Новая таблица содержит столбец budget из исходной таблицы и два дополнительных столбца: month и year. Столбец year таблицы project_dept_pivot содержит значения 2007 и 2008, которые содержатся в столбце date_month таблицы project_dept. Кроме этого, столбцы month таблицы project_dept_pivot (January, February, July) содержат суммарные значения бюджетов, соответствующие этим месяцам в таблице project_dept.

Запрос для создания таблицы project_dept_pivot показан в примере 23.27.

Пример 23.27. Создание таблицы `project_dept_pivot`

```
USE sample;
SELECT budget, month(date_month) as month, year(date_month) as year
   INTO project_dept_pivot
   FROM project_dept;
```

Содержимое созданной таблицы `project_dept_pivot` показано в табл. 23.2.

Содержимое таблицы `project_dept_pivot`

Budget	Month	Year
50000	1	2007
70000	2	2007
65000	7	2007
10000	7	2007
40000	2	2007
30000	1	2007
40000	2	2008
100000	1	2008
180000	2	2008
100000	7	2008
120000	1	2008

Допустим, что нам нужно для каждого года возвратить одну строку, для каждого месяца — один столбец, а для пресечений года и месяца — возвратить соответствующее суммарное значение бюджета. Требуемый результат показан в табл. 23.3.

Суммарные бюджеты для каждого месяца года

Year	January	February	July
2007	80000	110000	75000
2008	220000	220000	100000

В примере 23.28 показано решение этой задачи посредством стандартных средств SQL.

Пример 23.28. Создание сводной таблицы с помощью стандартного SQL

```
USE sample;
SELECT year,
   SUM(CASE WHEN month = 1 THEN budget END) AS January,
```

```
   SUM(CASE WHEN month = 2 THEN budget END) AS February,
   SUM(CASE WHEN month = 7 THEN budget END) AS July
      FROM project_dept_pivot
      GROUP BY year;
```

Процесс сведения данных можно разделить на три этапа:

- *Группирование данных*. Для каждого отдельного строчного элемента в результирующем наборе создается одна строка. В примере 23.28 строчным элементом является столбец `year`, и он входит в предложение `GROUP BY` инструкции `SELECT`.
- *Манипулирование данными*. Распределяем значения, которые будут агрегироваться, в столбцы целевой таблицы. В примере 23.28 столбцами целевой таблицы являются все отдельные значения столбца `month`. Этот шаг реализуется, применяя выражение `CASE` для каждого отдельного значения столбца `month`: `1` (January), `2` (February) и `7` (July).
- *Агрегирование данными*. Агрегируем значения данных в каждом столбце целевой таблицы. В примере 23.28 этот шаг выполняется посредством использования функции `SUM`.

Запрос в примере 23.29 дает тот же результат, что и запрос в примере 23.28, но посредством использования оператора `PIVOT`.

Пример 23.29. Создание сводной таблицы с помощью оператора `PIVOT`

```
USE sample;
SELECT year, [1] as January, [2] as February, [7] July FROM
   (SELECT budget, year, month from project_dept_pivot) p2
   PIVOT (SUM(budget) FOR month IN ([1],[2],[7])) AS P;
```

Инструкция `SELECT` в примере 23.29 содержит внутренний запрос, вложенный в предложение `FROM` внешнего запроса. Предложение `PIVOT` является частью внутреннего запроса. Оно начинается с задания агрегатной функции `SUM` (которая суммирует бюджеты). Во второй части задается столбец (`month`), по которому будет выполняться сведение, и значения этого столбца будут использоваться в качестве заголовков столбцов результирующего набора — первый, второй и седьмой месяцы года. Значение для конкретного столбца строки вычисляется, используя заданную агрегатную функцию по строкам, значения которых соответствуют заголовку столбца.

Наиболее важным преимуществом использования оператора `PIVOT` по сравнению со стандартным решением является его простота в тех случаях, когда целевая таблица содержит большое количество столбцов. В таком случае стандартное решение требует большого объема кода, поскольку для каждого столбца целевой таблицы необходимо указать отдельное выражение `CASE`.

Оператор *UNPIVOT*

Оператор UNPIVOT выполняет действие, обратное действию оператора PIVOT, превращая столбцы в строки. Использование этого оператора показано в примере 23.30.

Пример 23.30. Использование оператора UNPIVOT

```
USE sample;
CREATE TABLE project_dept_pvt (year int, January float,
                               February float, July float);
INSERT INTO project_dept_pvt VALUES (2007, 80000, 110000, 75000);
INSERT INTO project_dept_pvt VALUES (2008, 50000, 80000, 30000);
--UNPIVOT the table
SELECT year, month, budget
FROM
   (SELECT year, January, February, July
      FROM project_dept_pvt) p
         UNPIVOT (budget FOR month IN (January, February, July)
      )AS unpvt;
```

Результат выполнения этого запроса:

year	month	budget
2007	January	80000
2007	February	110000
2007	July	75000
2008	January	50000
2008	February	80000
2008	July	30000

Для демонстрации работы оператора UNPIVOT в примере 23.30 используется таблица project_dept_pvt. Сначала оператор UNPIVOT обрабатывает имя столбца, который содержит нормализованные значения (в данном случае, это столбец budget). Далее, посредством параметра FOR определяется имя целевого столбца (month). Наконец, в параметре IN указываются выбранные значения имени целевого столбца.

ПРИМЕЧАНИЕ

Оператор UNPIVOT не является точной противоположностью оператора PIVOT, поскольку значения NULL в преобразовываемой таблице нельзя использовать в качестве значений столбцов в выводе.

Резюме

Расширения SQL/OLAP языка Transact-SQL поддерживают возможности анализа данных. Компонент Database Engine поддерживает четыре основные составляющие расширений SQL/OLAP:

- конструкция окна;
- расширения предложения `GROUP BY`;
- функции запросов OLAP;
- стандартные и нестандартные аналитические функции.

Конструкция окна является наиболее важным расширением. Совместно с ранжирующими и агрегатными функциями она позволяет с легкостью вычислять аналитические функции, такие как накопительные и скользящие агрегатные значения, а также ранжирование. Компонент Database Engine поддерживает несколько расширений предложения `GROUP BY`, представленных в стандарте SQL. Это такие расширения, как операторы `CUBE`, `ROLLUP` и `GROUPING SETS`, а также функции группирования `GROUPING` и `GROUPING_ID`.

Наиболее важными аналитическими функциями запросов являются ранжирующие функции: `RANK`, `DENSE_RANK` и `ROW_NUMBER`. Язык Transact-SQL поддерживает несколько нестандартных аналитических функций и операторов: `TOP`, `NTILE`, `PIVOT` и `UNPIVOT`, а также стандартную комбинацию предложений `OFFSET/FETCH`.

В следующей главе рассматриваются службы отчетности Reporting Services, которые являются компонентом бизнес-аналитики сервера SQL Server.

Упражнения

Упражнение 23.1

Вычислите среднее количество сотрудников в отделе `Accounting`. Решите эту задачу, используя:

- конструкцию окна;
- предложение `GROUP BY`.

Упражнение 23.2

Используя конструкцию окна, определите отдел с наибольшим бюджетом в 2007 и в 2008 гг.

Упражнение 23.3

Определите общее число сотрудников в соответствии с комбинацией значений в отделах и суммах бюджетов. Нужно также отобразить все возможные суммарные строки.

Упражнение 23.4

Решите предыдущее упражнение 23.3, используя оператор `ROLLUP`. В чем заключается разница между этим результатом и результатом упражнения 23.3?

Упражнение 23.5

Используя функцию `RANK`, определите три отдела с наибольшим числом сотрудников.

Упражнение 23.6

Решите упражнение 23.5, используя предложения `TOP`.

Упражнение 23.7

Вычислите ранги всех отделов за 2008 г. по числу сотрудников. Также отобразите значения для функций `DENSE_RANK` и `ROW_NUMBER`.

Глава 24

Службы отчетности SQL Server Reporting Services

- **Введение в информационные отчеты**
- **Архитектура служб отчетности SQL Server Reporting Services**
- **Настройка служб отчетности SQL Server Reporting Services**
- **Создание отчетов**
- **Управление отчетами**

В этой главе рассматривается предоставляемый корпорацией Microsoft инструмент корпоративной отчетности, называемый *SQL Server Reporting Services (SSRS)*. После предоставления вводной информации об общей структуре отчета, в главе рассматриваются основные компоненты служб отчетности Reporting Services, а также описывается настройка установленного экземпляра этих служб. Затем рассматривается создание отчетов, используя средство BIDS (Business Intelligence Development Studio). Каждый шаг процесса создания отчетов показан на примерах: один без использования параметров, а другой с использованием. Далее объясняется обработка отчетов, а в завершении демонстрируются разные способы доставки сконструированного и развернутого отчета.

Введение в информационные отчеты

Отчет представляет собой один из интерфейсов, посредством которого пользователи могут взаимодействовать с системой баз данных. С помощью отчетов данные визуализируются и представляются пользователям. Данные можно представлять во многих разных форматах.

Обычно информационные отчеты имеют ряд свойств.

- *Отчет можно использовать только для отображения данных.* В отличие от интерфейсов на основе форм, которые можно использовать как для чтения, так и

для изменения данных, отчет является интерфейсом только для чтения. Посредством отчетов данные можно просматривать статически или динамически.

- *Генератор отчетов поддерживает большое количество разных макетов и файловых форматов.* В отчете данные представляются в предварительно отформатированном виде. Иными словами, с помощью макета отчета можно скомпоновать элементы, соответствующие результирующим значениям запроса отчета. (Отчет обычно можно спроектировать в графической или табличной форме.)
- *Отчет всегда основан на соответствующей инструкции* `SELECT`. Поскольку отчеты можно использовать только для отображения данных, каждый отчет основан на инструкции `SELECT`, которая извлекает данные. Разница между результатом операции извлечения и соответствующим отчетом состоит в том, что во втором случае для отображения данных используются разные стили и форматы, тогда как результирующий набор каждой инструкции `SELECT` имеет статическую форму, устанавливаемую системой.
- *В отчете можно использовать параметры, которые являются частью определенного запроса и чьи значения устанавливаются во время выполнения.* С помощью используемых в запросах параметров отчетам можно придать больше гибкости. Значения этих параметров передаются запросу пользователем или прикладной программой.
- *Всегда существует один или больше источников вводных данных для отчета.* Каждый отчет содержит запрос, который извлекает данные. Эти данные сохраняются в одном или нескольких источниках, которыми обычно являются базы данных, но также могут быть и файлы.

Что касается структуры, каждый отчет имеет следующие два набора инструкций, которые совместно определяют содержимое отчета.

- *Набор инструкций описания данных*. Определяет источники данных и набор данных. (Источники и наборы данных подробно рассматриваются в *разд. "Планирование источников данных и наборов данных" далее в этой главе*.) Содержимое набора данных можно определить, используя средство разработки запросов Query Designer. (Средство Query Designer представляет собой графический инструмент, с помощью которого можно создавать запросы для использования в отчетах. Этот инструмент особенно полезен для пользователей, которые не знают синтаксиса инструкции `SELECT` и нуждаются в помощи для создания запроса отчета.)
- *Набор инструкций макета отчета*. Позволяет представлять выбранные данные пользователям. В частности, можно задавать соответствие значений столбцов полям, а также формат и размещение заголовков и номеров страниц.

С этими общими знаниями об отчетах можно теперь приступить к рассмотрению архитектуры служб отчетности SQL Server Reporting Service.

Архитектура служб отчетности SQL Server Reporting Services

Службы отчетности SQL Server Reporting Services (SSRS) представляют собой систему программного обеспечения для генерирования отчетов. С помощью этой системы можно разрабатывать и обслуживать любые типы отчетов.

Службы SSRS содержат три основных компонента, которые представляют уровень сервера, уровень метаданных и уровень приложений соответственно:

- служба Windows Reporting Services;
- каталог отчетов;
- диспетчер отчетов.

Эти компоненты обсуждаются в последующих разделах, после короткого введения в язык определения отчетов *RDL* (Report Definition Language).

В процессе сбора информации, связанной с определением данных и макета отчета, службы SSRS сохраняют ее, используя язык RDL. Основанный на языке XML, язык RDL применяется исключительно для хранения определений и макетов отчетов. Язык RDL имеет открытую схему, что означает, что разработчики могут расширять этот язык дополнительными XML-атрибутами и элементами. (Описания языка XML в общем и его элементов и атрибутов в частности см. в *главе 26*.) Язык RDL обычно используется в Visual Studio, хотя его также можно использовать и программным образом.

Типичный файл RDL содержит три основных раздела. В первом разделе определяется стиль страниц, во втором задаются определения полей, а в третьем определяются параметры.

ПРИМЕЧАНИЕ

Для разработки отчетов использование языка RDL не является обязательным. Этот язык представляет важность только в тех случаях, когда нужно создавать свои файлы RDL.

Служба Windows Reporting Services

Служба Reporting Services является компонентом Windows, которая содержит два важных компонента в плане сервера отчетов. Первый из них — веб-служба Reporting Services, представляет собой приложение, используемое для реализации веб-сайта диспетчера отчетов Report Manager, а второй — собственно служба Windows — Reporting Services, применяется в качестве программируемого интерфейса для отчетов. (Диспетчер отчетов Report Manager рассматривается далее в этой главе.)

ПРИМЕЧАНИЕ

Обе службы, Windows служба Reporting Services и веб-служба Reporting Services, работают совместно и составляют единый экземпляр сервера отчетов.

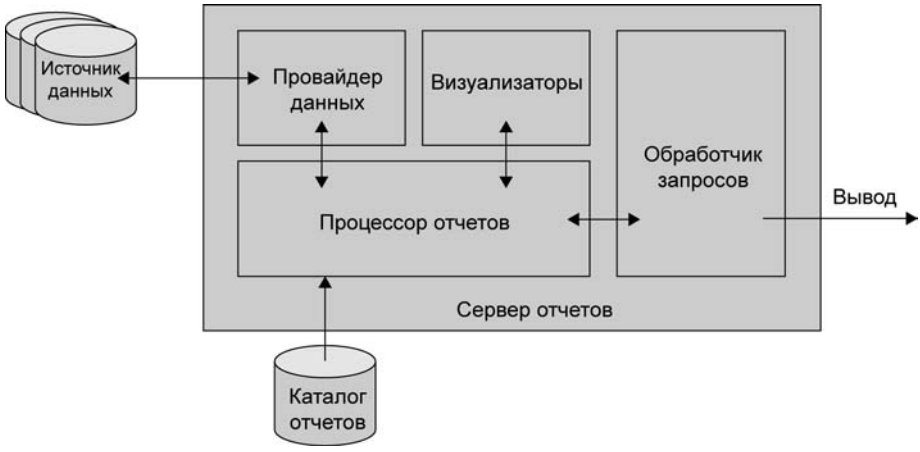


Рис. 24.1. Компоненты службы Windows — Reporting Services

На рис. 24.1 показана блок-диаграмма Windows службы Reporting Services.

Как можно видеть на рис. 24.1, Windows служба Reporting Services содержит следующие компоненты:

- процессор отчетов;
- поставщик (провайдер) данных;
- визуализаторы (рендереры, от англ. *Renderer*);
- обработчик запросов.

Процессор (обработчик) отчетов управляет выполнением отчета. Он извлекает определение отчета, которое составлено на языке RDL, и определяет, что требуется для отчета. Кроме этого, он управляет работой других компонентов, участвующих в создании отчетов. Процессор отчетов также извлекает данные из источников данных. Для этого он выбирает поставщика данных, который знает, как получить информацию из источника данных. Задача поставщика данных заключается в подключении к источнику данных, получению информации для отчета и в возвращении ее процессору отчетов в виде соответствующих наборов данных.

После получения данных от поставщика процессор отчетов может начинать работать над макетом отчета. Чтобы создать макет, процессору нужно знать формат отчета (например, HTML или PDF). Для создания требуемого формата применяются визуализаторы.

Обработчик запросов получает запросы на отчеты и отправляет их процессору отчетов, а также доставляет готовые отчеты. (Разные способы доставки отчетов рассматриваются *далее в этой главе*.)

Каталог отчетов

Каталог отчетов содержит две базы данных, в которых хранятся определения всех существующих отчетов, принадлежащих к определенной службе. Хранящаяся ин-

формация включает имена отчетов, описания, сведения о соединении источника данных, учетные данные, параметры и свойства выполнения. В каталоге отчетов также хранятся параметры безопасности и информация об организации и доставке данных.

Службы SSRS используют постоянную и временную базы данных сервера Report Server для хранения постоянных и временных данных соответственно. Эти базы данных создаются вместе и привязываются по имени. По умолчанию этим базам данных присваиваются имена `reportserver` и `reportservertempdb` соответственно. Первая база данных используется для хранения каталога отчетов, а вторая используется в качестве временного хранилища кэшированных отчетов и рабочих таблиц, генерируемых отчетами.

Диспетчер отчетов Report Manager

Диспетчер отчетов Report Manager представляет собой инструмент на основе веб-технологий для доступа к отчетам и управления ими посредством браузера. В этом разделе приведено только перечисление заданий, которые можно выполнять с помощью этого инструмента. Описания выполнения определенных заданий рассматриваются в последующих разделах этой главы.

Диспетчер отчетов Report Manager можно использовать для выполнения ряда задач.

- Просматривать и распечатывать отчеты, подписываться на отчеты и выполнять их поиск. Создание подписок на отчеты с помощью диспетчера отчетов рассматривается в *разд. "Стандартные подписки" далее в этой главе*.
- Определять проблемы безопасности, связанные с доступом к элементам и операциям.
- Осуществлять настройку параметров отчета и особенности его выполнения.
- Создавать модели отчетов, подключающиеся к источникам данных.
- Создавать разделяемые источники данных, чтобы сделать соединения с источниками данных более управляемыми. (Разделяемые источники данных подробно рассматриваются в *разд. "Организация источников данных и наборов данных" далее в этой главе*.)
- Создавать управляемые данными подписки, которые отправляют отчеты большому списку подписчиков. (Управляемые данными подписки рассматриваются в одноименном разделе *далее в этой главе*.)
- Запускать построитель отчетов Report Builder для создания отчетов, которые можно сохранять и выполнять на сервере отчетов.

В следующем разделе мы рассмотрим настройку установленного экземпляра служб SQL Server Reporting Services.

Настройка служб отчетности SQL Server Reporting Services

Для настройки установленного экземпляра служб Reporting Services применяется диспетчер *Reporting Services Configuration Manager (RSCM)*. Как уже упоминалось в *главе 2*, где рассматривалась установка служб SSRS, при установке этих служб имеется выбор установить и настроить сервер отчетов или же только установить его без настройки.

Если при установке была выбрана опция установки без настройки (**Install, but do not configure the report server** — см. рис. 2.9), прежде чем использовать сервер отчетов, его нужно настроить с помощью диспетчера RSCM. Если сервер отчетов был установлен, используя опцию установки с настройками по умолчанию, то примененные во время установки настройки можно проверить или изменить с помощью диспетчера RSCM.

ПРИМЕЧАНИЕ

Диспетчер RSCM устанавливается автоматически при установке служб SSRS, и с его помощью можно выполнять настройку как локальных, так и удаленных серверов отчетов.

Чтобы запустить диспетчер RSCM, выполните последовательность команд меню **Пуск | Все программы | Microsoft SQL Server 2012 | Configuration Tools | Reporting Services Configuration Manager**. В открывшемся диалоговом окне **Reporting Services Configuration Connection** можно выбрать экземпляр сервера отчетов, который требуется настроить. Для этого в поле **Server Name** введите имя компьютера, на котором установлен требуемый экземпляр сервера отчетов. (Если же экземпляр сервера отчетов установлен на удаленном компьютере, то нажмите кнопку **Find**, чтобы найти этот экземпляр в сети и подключиться к нему.) В раскрывающемся списке **Report Server Instance** выберите экземпляр служб Reporting Services, который требуется настроить, и нажмите кнопку **Connect**. После подключения к серверу отчетов можно будет выполнять целый ряд задач.

- *Настраивать служебную учетную запись сервера Report Server*. Исходные настройки этой учетной записи можно изменить, используя диспетчер RSCM.
- *Задавать и настраивать адреса URL*. Службы Reporting Services и диспетчер Report Manager являются приложения ASP.NET, доступ к которым осуществляется посредством использования адресов URL. Для каждого из этих приложений можно задать один или несколько адресов URL.
- *Создавать и настраивать базу данных сервера отчетов.* На этой странице можно создать или изменить базу данных сервера отчетов или обновить учетные данные соединения с базой данных.
- *Задавать настройки e-mail*. Можно указать сервер SMPT и учетную запись e-mail, чтобы использовать возможности e-mail сервера отчетов.

- *Настраивать учетную запись для необслуживаемого выполнения*. Эта учетная запись применяется для удаленных подключений при запланированных операциях или когда отсутствуют учетные данные пользователя.
- *Создавать резервную копию ключей шифрования*. Такая необходимость может возникнуть при переносе системы сервера отчетов на другой компьютер.

Теперь, когда мы ознакомились с компонентами служб отчетности SQL Server Reporting Services и их настройкой, можно приступать к изучению, как создавать, развертывать и доставлять отчеты.

Создание отчетов

Для создания отчетов службы Reporting Services предоставляют следующие два инструмента:

- *средство Business Intelligence Development Studio (BIDS)*. Этот инструмент используется на стадии разработки. Он тесно интегрирован с Visual Studio и позволяет разрабатывать и тестировать отчеты, прежде чем начинать применять их;
- *построитель отчетов Report Builder*. Этот автономный инструмент позволяет пользователям создавать специализированные отчеты, не имея никакой информации о структуре используемой базы данных и не обладая навыками создания SQL-запросов.

ПРИМЕЧАНИЕ

В этой книге рассматривается только первая опция создания отчетов — посредством использования средства BIDS. Причиной этого является то обстоятельство, что построитель отчетов Reports Builder в основном используется не для создания новых отчетов, а для редактирования уже существующих. Дополнительную информацию по построителю отчетов можно найти в *электронной документации*.

Чтобы запустить средство BIDS, выполните последовательность команд меню **Пуск | Все программы | Microsoft SQL Server 2012 | SQL Server Business Intelligence Development Studio**. Первым шагом в процессе создания отчета будет создание нового проекта, к которому будет принадлежать этот отчет. Для этого выберите последовательность команд меню **File | New | Project**. В открывшемся диалоговом окне **New Project** выберите папку **Business Intelligence**. Введите в поля **Name** и **Location** имя и место расположения проекта соответственно. В данном примере проект называется Project1 (рис. 24.2).

В средней области диалогового окна можно найти два шаблона проектов для отчетов. Шаблон "Report Server Project" позволяет создать пустой отчет, предоставляя пользователю выполнять всю прочую работу по созданию отчета. А шаблон "Report Server Project Wizard" запускает мастера отчетов, который предоставляет инструкции по созданию отчета.

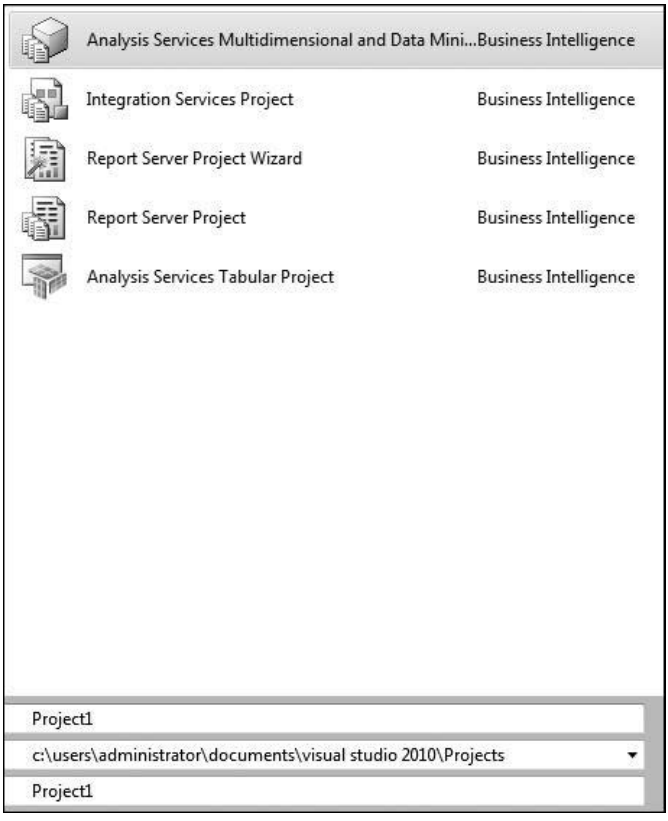


Рис. 24.2. Часть диалогового окна **New Project** для создания нового проекта

ПРИМЕЧАНИЕ

Обычно отчеты служб Reporting Services создаются, используя конструктор отчетов Report Designer, который представляет собой набор графических инструментов для проектирования запросов и отчетов. В панели **Report Data** этого конструктора можно организовать данные, используемые в отчете, а с помощью представлений с **Design** и **Preview** можно проектировать отчеты в интерактивном режиме. В состав конструктора Report Designer также входит конструктор запросов Query Designer, упомянутый ранее в этой главе, с помощью которого можно указать данные для выборки, а также диалоговое окно **Expression**, в котором можно задать данные для использования в макете отчета. Конструктор Report Designer доступен через средство BIDS.

В следующем разделе мы создадим новый отчет, используя мастер Report Server Project Wizard. Поэтому в диалоговом окне **New Project** (см. рис. 24.2) выберите опцию **Report Server Project Wizard** и нажмите кнопку **OK**. Откроется страница приветствия этого мастера.

Создание отчетов с помощью мастера Report Server Project Wizard

На странице приветствия мастера Report Server Project Wizard представляются основные шаги, через которые он проведет пользователя при создании отчета:

1. Выбор источника данных для получения данных.
2. Конструирование запроса для работы с выбранным источником данных.
3. Выбор типа отчета.
4. Указание базового макета отчета.
5. Выбор форматирования отчета.

Эти шаги (и другие, которых нет в списке на странице приветствия) описываются в последующих разделах, после чего следует краткое руководство по просмотру результирующего набора и развертыванию отчета.

Но сначала нам нужно научиться организовывать источники данных и наборы данных.

Организация источников данных и наборов данных

Прежде чем создавать отчет, для него нужно подготовить источники данных. Эти источники данных применяются для создания соответствующих таблиц и/или представлений, с помощью которых будет извлекаться определенный результирующий набор. По этой причине источники данных подготавливаются в среде SQL Server Management Studio, которая обладает возможностями языка Transact-SQL, а не с помощью служб SSRS.

На стадии планирования требуется работать как с источниками данных, так и с наборами данных. Далее поясняется различие между этими двумя понятиями, после чего приводятся описания применения их обоих.

Источники данных по сравнению с наборами данных. Самое основное различие между источниками данных и наборами данных состоит в том, что источники данных не включаются в отчет. Источник данных только предоставляет информацию для отчета. Эта информация может находиться в файле или в базе данных. Задача служб SSRS заключается в том, чтобы сгенерировать наборы данных с указанных источников данных, используя для этого набор инструкций. Этот набор инструкций включает, среди прочего, информацию о типе источника данных, имя базы данных (если источник данных находится в базе данных) или путь к файлу, а также, факультативно, информацию для подключения к источнику данных.

При выполнении отчета службы SSRS используют эту информацию для создания нового формата, который называется *набором данных* (dataset). Таким образом, набор данных является всего лишь абстракцией лежащих в его основе источников данных и используется в качестве непосредственного ввода для соответствующего отчета.

Использование источников данных. Чтобы включить данные в отчет, сначала нужно создать подключение данных (синоним для источников данных). Подключе-

ние данных содержит тип источника данных, информацию о подключении и имя входа для подключения. (Создание подключений данных рассматривается в *разд. "Выбор источника данных" далее в этой главе*.)

Существует два типа источников данных: *встроенные* и *разделяемые*. Встроенный источник данных определяется в отчете и используется только этим отчетом. Иными словами, для того чтобы модифицировать встроенный источник данных, необходимо изменить свойства этого источника данных с помощью диспетчера отчетов Report Manager. Разделяемый источник данных определяется независимо от отчета и может использоваться несколькими отчетами. Разделяемые источники данных полезны в тех случаях, когда источники данных используются часто.

ПРИМЕЧАНИЕ

Рекомендуется как можно больше использовать разделяемые источники данных. Они делают управление отчетами более легким, а доступ к отчетам более безопасным.

Использование наборов данных. Как уже упоминалось, каждый набор данных является абстракцией соответствующих источников данных и определяет поля из источника данных, которые планируется применить в отчете. Все наборы данных, создаваемые для определения отчета, отображаются в окне **Datasets**.

Начиная с версии SQL Server 2008 R2, службы SSRS поддерживают разделяемые наборы данных. Проще говоря, *разделяемый набор данных* — это набор данных, который позволяет нескольким отчетам использовать один запрос для предоставления непротиворечивых данных. Разделяемые наборы данных тесно связаны с разделяемыми источниками данных. Иными словами, для генерации разделяемых наборов данных используются разделяемые источники данных.

Запрос по разделяемому набору данных может содержать параметры. Разделяемый набор данных можно настроить, чтобы кэшировать результаты для определенных комбинаций параметров при первом использовании или по указанному расписанию.

Для создания разделяемого набора данных можно использовать средства BIDS или построитель отчетов Report Builder. Чтобы создать разделяемый набор данных с помощью средства BIDS, откройте окно обозревателя объектов, щелкните правой кнопкой папку **Shared Datasets** и в контекстном меню выберите пункт **Add New Dataset**. В открывшемся диалоговом окне **Dataset Properties** выберите требуемый источник данных и в следующем окне введите (или скопируйте и вставьте) инструкцию `SELECT`.

Выбор источника данных

Информация о подключении к базе данных-источнику содержится в источнике данных. Чтобы выбрать источник данных, нажмите кнопку **Next** на странице приветствия мастера Report Server Project Wizard. На открывшейся странице **Select Data Source** в поле **Name** введите имя нового источника данных. (Для этого примера назовем источник данных **Source1**.)

В раскрывающемся списке **Type** на этой странице можно выбрать один из нескольких различных типов источников данных. Службы SSRS могут создавать отчеты для разных реляционных баз данных (SQL Server, Teradata и Oracle, среди прочих) или многомерных баз данных (Analysis Services). Можно также использовать источники данных типа OLE DB, ODBC и XML. Выбрав тип источника данных, нажмите кнопку **Edit**. Откроется диалоговое окно **Connection Properties** (рис. 24.3).

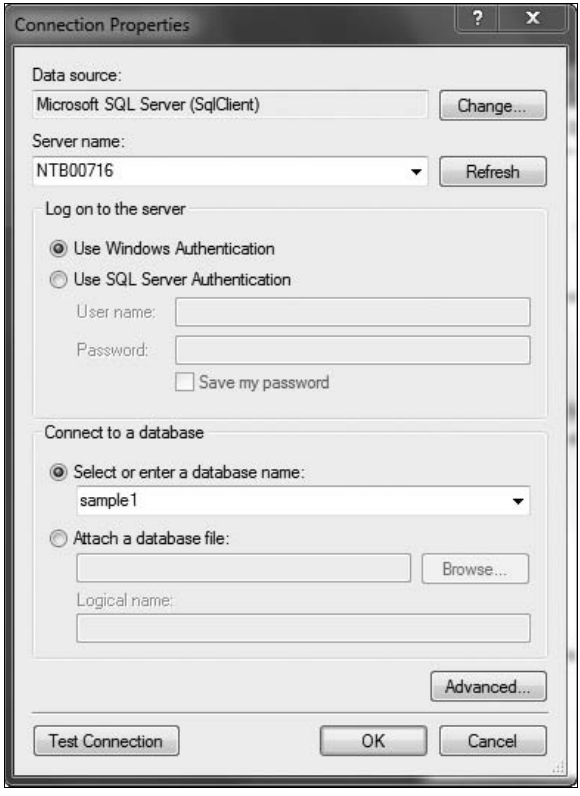


Рис. 24.3. Диалоговое окно **Connection Properties**

В поле **Server name** введите `localhost` или имя вашего установленного сервера баз данных. Потом задайте или проверку подлинности Windows, или проверку подлинности SQL Server, выбрав соответствующий переключатель. Затем установите переключатель **Select or enter a database name** и в раскрывающемся списке этого переключателя в качестве источника данных выберите одну из баз данных. Проверьте соединение с базой данных, нажав кнопку **Test Connection**, и, при положительном результате проверки, нажмите кнопку **OK**. Это возвратит вас обратно на страницу мастера **Select the Data Source**. Нажмите кнопку **Next** для продолжения работы мастера.

Проектирование запроса

Следующим шагом будет проектирование запроса, который можно выполнять по отношению к выбранному источнику данных. На странице мастера **Design the**

Query в окне **Query string** можно ввести (или вставить) уже существующий запрос или же воспользоваться компонентом Query Builder и создать новый запрос.

ПРИМЕЧАНИЕ

Компонент Query Builder соответствует подобному компоненту приложения Access, с помощью которого можно конструировать запросы, даже не зная языка SQL. Этот компонент обычно называется *QBE* (Query by Example, запрос по образцу).

Для нашего первого отчета используйте запрос, показанный в примере 24.1.

Пример 24.1. Запрос для создания отчета

```
USE sample;
SELECT dept_name, emp_lname, emp_fname, job, enter_date
      FROM department d JOIN employee e ON d.dept_no = e.dept_no
                        JOIN works_on w ON w.emp_no = e.emp_no
      WHERE YEAR(enter_date) = 2007
      ORDER BY dept_name;
```

Этот запрос выбирает данные для сотрудников, которые начали работать в 2007 году, с дополнительной сортировкой полученных данных по отделам. Вставив запрос в поле **Query string**, нажмите кнопку **Next**.

ПРИМЕЧАНИЕ

Службы SSRS проверяют имена таблиц и столбцов, используемых в запросе. Если система обнаружит какие-либо ошибки синтаксиса, то она выводит соответствующее сообщение в новом окне.

Выбор типа отчета

Следующим шагом в создании отчета будет выбор его типа. Имеется возможность выбрать один из следующих типов отчета:

- *табличный* (tabular) — создает отчет в табличной форме. Столбцы таблицы отчета соответствуют столбцам, указанным в списке инструкции `SELECT`, а количество строк таблицы соответствует количеству строк, возвращенных в результирующем наборе запроса;
- *матричный* (matrix) — создает отчет матричного типа, который подобен табличному, но предоставляет функциональность перекрестных табличных данных. В отличие от отчета табличного типа с его статичным набором столбцов, отчет матричного типа может быть динамическим.

ПРИМЕЧАНИЕ

Для запросов, содержащих агрегатные функции, такие как `AVG` или `SUM`, всегда следует использовать отчеты матричного типа.

Запрос в примере 24.1 не содержит никаких агрегатных функций, поэтому выбираем табличный тип запроса и нажимаем кнопку **Next**.

Создание таблицы

Страница мастера **Design the Table** (рис. 24.4) позволяет разместить столбцы в требуемом месте отчета.

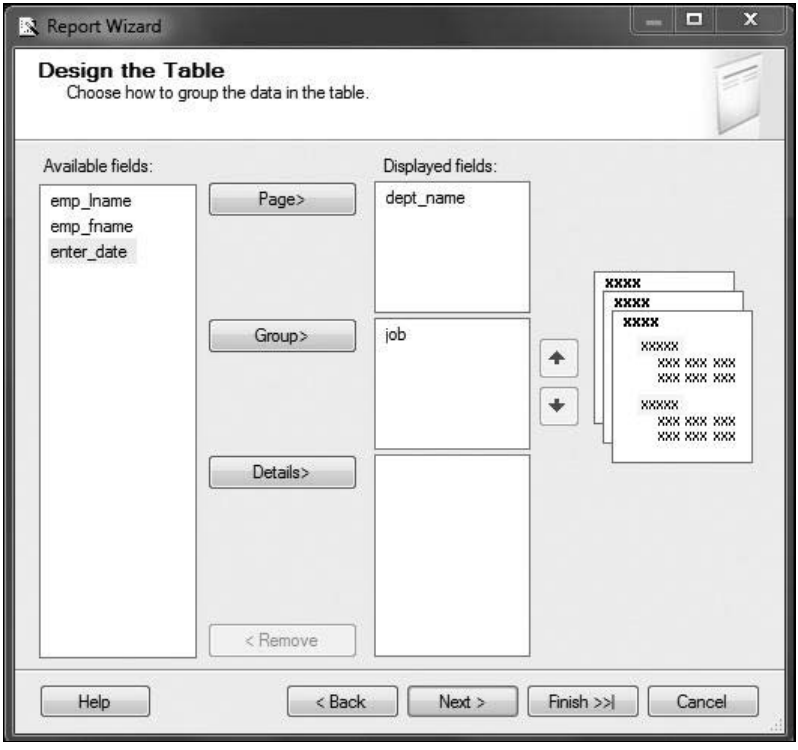


Рис. 24.4. Мастер отчетов, страница **Design the Table**

Страница **Design the Table** содержит две следующие группы полей:

- **Available fields** (доступные поля);
- **Displayed fields** (отображаемые поля).

Также эта страница имеет три представления:

- **Page** (страница);
- **Group** (сгруппировать)
- **Details** (подробности)

Доступные поля — это столбцы из списка выбора в инструкции `SELECT` вашего запроса. Каждый из этих столбцов можно поместить в одно из представлений группы отображаемых полей (**Page**, **Group** или **Details**). Для этого нужно выбрать требуе-

мое поле, а затем нажать соответствующую кнопку **Page**, **Group** или **Details**. *Отображаемое поле* — это доступное поле, помещенное в одно из представлений.

В представлении страницы перечисляются все столбцы, которые отображаются на уровне страницы, а в представлении группы перечисляются столбцы, которые используются для группирования результирующего набора запроса. В представлении подробностей перечисляются столбцы, которые отображаются в разделе подробностей таблицы. На рис. 24.4 показана страница **Design the Table** с проектированием табличного представления отчета по результирующему набору запроса примера 24.1.

Здесь столбец `dept_name` будет отображаться на уровне страницы, а выбранные строки будут группироваться в столбце `job`. После распределения столбцов по группам (представлениям) таблицы нажмите кнопку **Next**.

ПРИМЕЧАНИЕ

Порядок размещения столбцов в представлениях может быть важным, особенно для представления **Group**. Чтобы переместить столбец вверх или вниз по представлениям, выберите требуемый столбец и нажмите кнопку со стрелкой вверх или вниз справа от области **Displayed fields**.

Выбор макета отчета

Следующим шагом будет выбор макета отчета (размещение полей, текста и графики в отчете) из нескольких предлагаемых на странице **Choose the Table Layout** следующих опций:

- **Stepped** (ступенчатый);
- **Block** (блочный);
- **Include subtotals** (включить подитоги);
- **Enable drilldown** (разрешить детализацию).

Ступенчатый тип отчета (установлен флажок **Stepped**) содержит по одному столбцу для каждого поля набора данных, где группирующие поля размещаются в виде заголовков в столбцах слева от столбцов полей подробностей, с каждым группирующим полем следующего уровня на уровень ниже предыдущего. Нижних колонтитулов в данном примере не создается. Если бы они были в таком макете, например промежуточные итоги, они бы помещались в соответствующие столбцы заголовков на соответствующих уровнях.

Для отчета блочного типа (установлен флажок **Block**) для каждого поля результирующего набора данных также создается отдельный столбец, и группирующие поля также размещаются в столбцах в виде заголовков, но все такие поля для одной группы помещаются в одну строку. Нижние колонтитулы для этого типа отчета создаются только в том случае, если активирована опция включения подитогов (установлен флажок **Include subtotals**).

Активирование опции углубленной детализации (флажок **Enable drilldown**) создает отчет в виде иерархии групп столбцов, уровни которой можно сворачивать и разво-

рачивать подобно уровням иерархии, например объектов сервера Database Engine в обозревателе объектов. (Опция **Enable drilldown** применима только с отчетами ступенчатого типа.)

Для продолжения создания нашего примера отчета установите переключатель **Stepped** и флажок **Include subtotals** и нажмите кнопку **Next**.

Выбор формата отчета

Следующим шагом будет выбор формата отчета. На странице **Choose the Table Style** можно выбрать шаблон форматирования шрифтов, цвета и типа границы отчета. Из нескольких разных доступных стилей выберите **Bold** и нажмите кнопку **Next**.

Выбор места развертывания и завершение работы мастера

При первом создании отчета после выбора стиля отчета есть еще один промежуточный шаг — выбор места развертывания на странице **Choose the Deployment Location**. Здесь нужно выбрать адрес URL виртуальной папки сервера отчетов и папку для развертывания отчетов. Если сервер отчетов работает в собственном режиме, используйте путь к серверу отчетов, на котором развертывается проект (например, **http://localhost/ReportServer**). Если же сервер отчетов работает в режиме интеграции с SharePoint, укажите URL сайта SharePoint, на котором развертывается проект (например, **http://localhost**). Указав требуемые пути, нажмите кнопку **Next**.

ПРИМЕЧАНИЕ

Службы SSRS поддерживают два режима развертывания экземпляров сервера отчетов: собственный режим и режим интеграции с SharePoint. В собственном режиме (который применяется по умолчанию) сервер отчетов является автономным сервером приложений, который предоставляет все функциональности по просмотру, управлению, обработке и доставке отчетов. В режиме интеграции с SharePoint сервер отчетов становится частью развертывания веб-приложения SharePoint. Пользователи SharePoint могут сохранять отчеты в библиотеках SharePoint и обращаться к ним из тех же сайтов SharePoint, которые они используют для обращения к другим деловым документам.

Последним шагом в создании отчета является присвоение ему имени. Кроме этого, можно просмотреть суммарную информацию об отчете, в которой задокументированы все шаги по его созданию. Чтобы завершить работу мастера и сохранить созданный отчет, нажмите кнопку **Finish**.

Просмотр результирующего набора

После завершения работы мастера создания отчетов созданный отчет будет представлен в центральной панели средства BIDS в окне **Report Designer**. Это окно имеет две вкладки, позволяющие просматривать отчет в разных форматах. Если окно **Report Designer** не отображается, его можно открыть, выполнив последова-

тельность команд меню **View | Designer**. В частности, отчет можно просматривать в следующих представлениях:

- **Design** (макет);
- **Preview** (предпросмотр).

На вкладке **Design** можно просматривать и изменять макет отчета. В режиме **Design** отчет содержит такие разделы, как тело, страница, заголовок (верхний колонтитул) и нижний колонтитул. Объектами макета отчета можно манипулировать, используя окно панели элементов **Toolbox** и окно свойств **Properties**. Чтобы открыть эти окна, выберите соответствующие пункты в меню **View**. В панели элементов **Toolbox** можно выбирать элементы для их размещения в одном из разделов. В режиме макета каждый элемент отчета обладает свойствами, которыми можно управлять с помощью окна **Properties** (рис. 24.5).

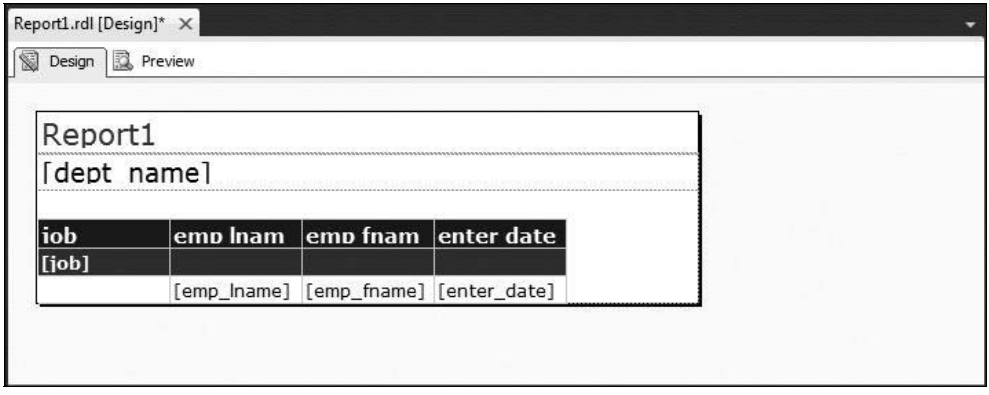


Рис. 24.5. Отображение отчета в режиме макета, вкладка **Design**

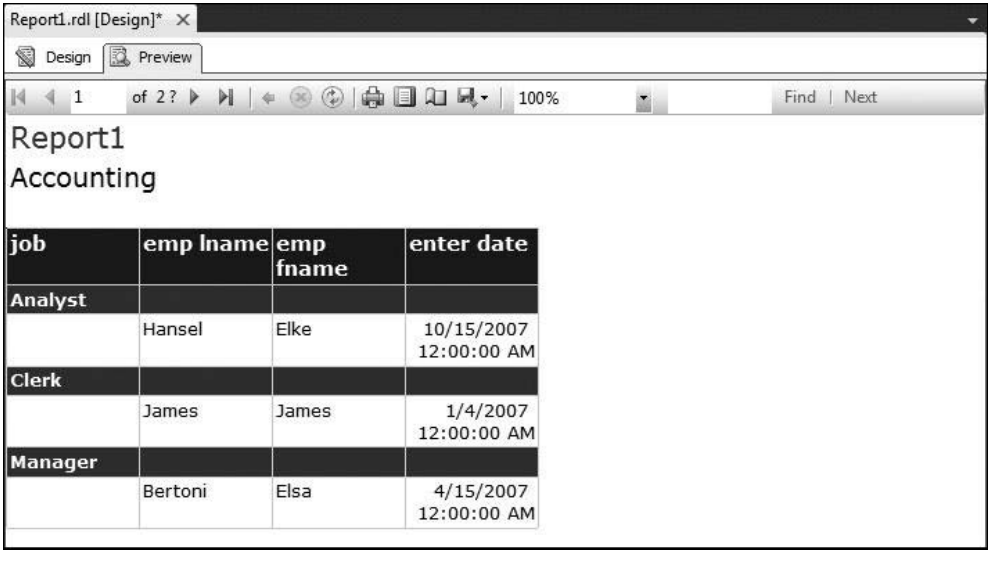


Рис. 24.6. Созданный отчет в режиме предпросмотра

На вкладке **Preview** можно выполнить предпросмотр отчета, т. е. посмотреть, как он будет выглядеть в рабочем виде. При выборе этой вкладки отчет отображается автоматически, с использованием заданных свойств. На рис. 24.6 в режиме предпросмотра показан отчет, созданный нами после выполнения всех предыдущих шагов.

Развертывание отчета

Прежде чем отчет можно использовать или распространять, его необходимо развернуть. Чтобы развернуть отчет, щелкните его имя в обозревателе решений правой кнопкой и в контекстном меню выберите пункт **Deploy**. Процесс развертывания выполняется в несколько этапов, которые отображаются в панели **Output**, как это показано далее.

```
--- Build started: Project: Project1, Configuration: Debug ---
Build complete -- 0 errors, 0 warnings
--- Deploy started: Project: Project1, Configuration: Debug---
Deploying to http://localhost/ReportServer
Deploying data source '/Project1/sample'.
Deploying report '/Project1/Report1'.
Deploy complete -- 0 errors, 0 warnings
=== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ===
=== Deploy: 1 succeeded, 0 failed, 0 skipped ===
```

Создание параметризованных отчетов

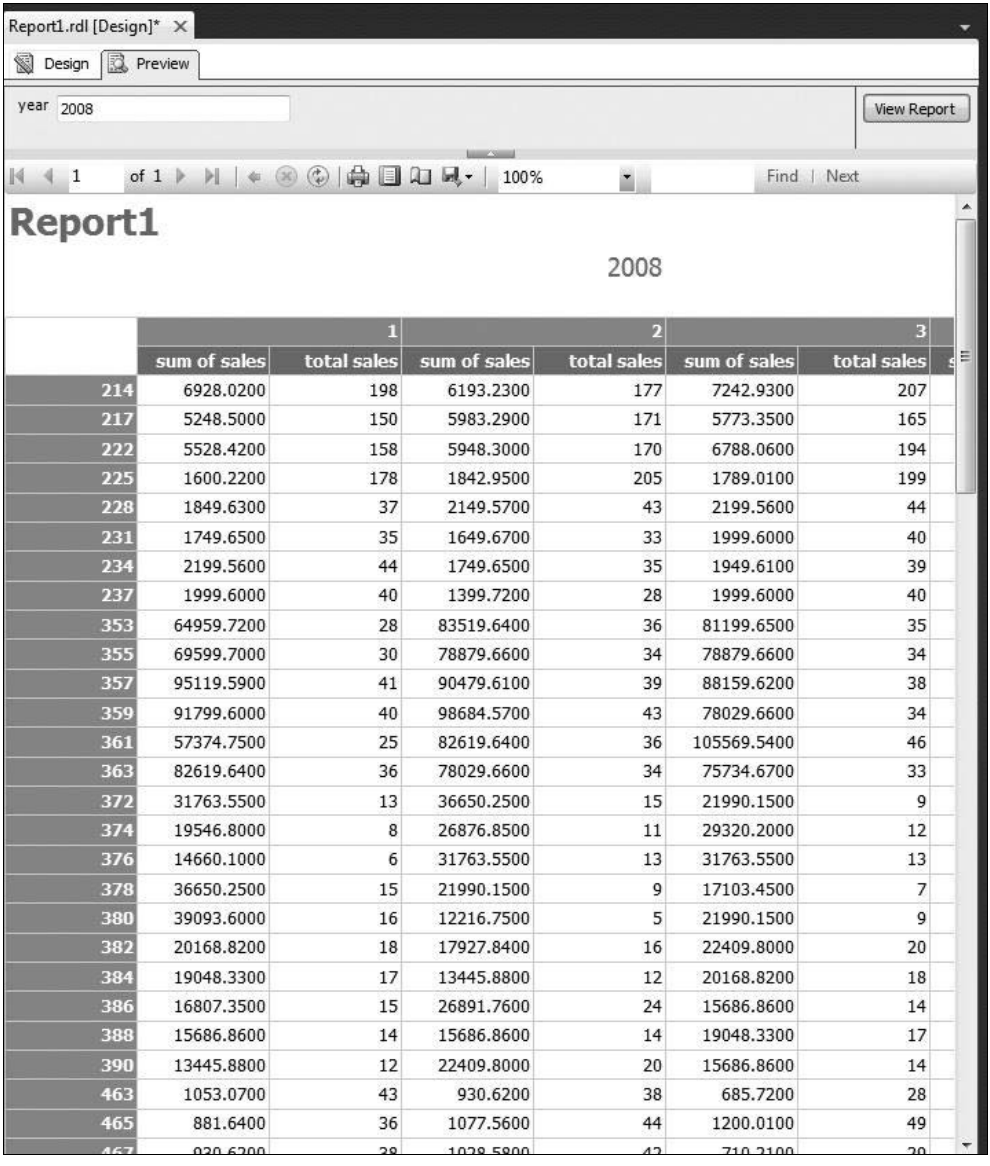
Параметризованный отчет — это такой отчет, в который передаются вводные параметры для его обработки. Эти параметры используются при выполнении запроса, выбирающего конкретные данные для отчета. Чтобы успешно разрабатывать или выполнять развертывание параметризованного отчета, необходимо понимать, как выбор параметров влияет на отчет.

В службах SQL Server Reporting Services параметры применяются для фильтрации данных. Параметры указываются, используя синтаксис для переменных (например, `@year`). Если в запросе указывается параметр, для него должно быть предоставлено значение, чтобы выполнить инструкцию `SELECT` или хранимую процедуру, которые извлекают данные для отчета.

Для параметра можно задать значение по умолчанию. Если для всех параметров заданы значения по умолчанию, то при выполнении отчета он сразу же будет содержать выбранные данные. Если же хоть для одного параметра не указано значение по умолчанию, то отчет отобразит данные только после того, как пользователь введет все требуемые значения параметров.

Когда отчет выполняется в браузере, значение параметра отображается в поле вверху отчета. А когда отчет выполняется в режиме предпросмотра (вкладка **Preview** панели **Report Designer**), значение параметра отображается в соответствующем поле.

Создание параметризованного отчета будет продемонстрировано на примере. Но в этом примере описываются только шаги, которые отличаются от шагов, уже рассмотренных для создания отчета в примере 24.1. Для данного примера используется запрос, показанный в примере 24.2, который выбирает данные из базы данных `AventureWorkwsDW`. По этой причине будет необходимо выбрать и определить новый источник данных. Этот источник данных задается идентично заданию источника данных `Source1` в предыдущем примере, с той разницей, что вместо базы данных `sample` выбирается база данных `AdventureWorksDW`.



	1		2		3	
	sum of sales	total sales	sum of sales	total sales	sum of sales	total sales
214	6928.0200	198	6193.2300	177	7242.9300	207
217	5248.5000	150	5983.2900	171	5773.3500	165
222	5528.4200	158	5948.3000	170	6788.0600	194
225	1600.2200	178	1842.9500	205	1789.0100	199
228	1849.6300	37	2149.5700	43	2199.5600	44
231	1749.6500	35	1649.6700	33	1999.6000	40
234	2199.5600	44	1749.6500	35	1949.6100	39
237	1999.6000	40	1399.7200	28	1999.6000	40
353	64959.7200	28	83519.6400	36	81199.6500	35
355	69599.7000	30	78879.6600	34	78879.6600	34
357	95119.5900	41	90479.6100	39	88159.6200	38
359	91799.6000	40	98684.5700	43	78029.6600	34
361	57374.7500	25	82619.6400	36	105569.5400	46
363	82619.6400	36	78029.6600	34	75734.6700	33
372	31763.5500	13	36650.2500	15	21990.1500	9
374	19546.8000	8	26876.8500	11	29320.2000	12
376	14660.1000	6	31763.5500	13	31763.5500	13
378	36650.2500	15	21990.1500	9	17103.4500	7
380	39093.6000	16	12216.7500	5	21990.1500	9
382	20168.8200	18	17927.8400	16	22409.8000	20
384	19048.3300	17	13445.8800	12	20168.8200	18
386	16807.3500	15	26891.7600	24	15686.8600	14
388	15686.8600	14	15686.8600	14	19048.3300	17
390	13445.8800	12	22409.8000	20	15686.8600	14
463	1053.0700	43	930.6200	38	685.7200	28
465	881.6400	36	1077.5600	44	1200.0100	49

Рис. 24.7. Часть полученного отчета в режиме предпросмотра

Пример 24.2. Запрос для выборки данных для параметризованного запроса

```
SELECT t.MonthNumberOfYear AS month, t.CalendarYear AS year,
   p.ProductKey AS product_id, SUM(f.UnitPrice) AS sum_of_sales,
   COUNT(f.UnitPrice) AS total_sales
   FROM DimDate t, DimProduct p, FactInternetSales f
   WHERE t.DateKey = f.OrderDateKey AND p.ProductKey = f.ProductKey
         AND CalendarYear = @year
   GROUP BY t.CalendarYear, t.MonthNumberOfYear, p.ProductKey
   ORDER BY 1;
```

Запрос в примере 24.2 вычисляет количество и сумму продаж продуктов. Он также группирует строки согласно списку имен столбцов, указанных в предложении `GROUP BY`. Выражение:

```
CalendarYear = @year
```

в предложении `WHERE` этого запроса указывает, что во вводном параметре `@year` требуется предоставить значение календарного года, для которого нужно получить данные.

Этот отчет является типичным примером отчета матричного типа. Значения столбца `year` указываются в представлении `Page`, столбца `month` — в представлении `Columns`, а столбца `product_id` — в представлении `Rows`. В представлении `Details` отображаются агрегатные значения `SUM` и `COUNT`.

Для запуска отчета в режиме предпросмотра в поле **year** введите значение параметра `CalendarYear` (например, `2008`) и нажмите кнопку **View Report**. На рис. 24.7 показана часть полученного отчета.

Управление отчетами

Обращение к отчетам и их доставка может осуществляться двумя способами:

- по требованию;
- по подписке.

Эти методы описываются в следующих разделах, после чего обсуждаются опции доставки отчетов.

Отчеты по требованию

Как уже упоминалось, прежде чем отчет можно было бы использовать или распространять, его необходимо развернуть. Развернуть отчет можно по требованию. Доступ по требованию позволяет пользователям выбирать отчет в средстве просмотра отчетов. Просматривать отчеты можно в диспетчере отчетов Report Manager или в обычном браузере. В этом разделе рассматривается просмотр отчетов по требованию в браузере.

Доступ к отчетам по требованию обеспечивает веб-служба Reporting Services. Все отчеты организованы в иерархическом пространстве имен и доступ к ним осуществляется через виртуальные каталоги. Если были указаны адреса URL по умолчанию, доступ к веб-службе Reporting Services можно получить по адресу URL, который задает в качестве хоста имя компьютера или localhost: **http://localhost/ReportServer**. Виртуальным каталогом по умолчанию для диспетчера отчетов Report Manager является **http://localhost/Reports/**. (Оба эти значения по умолчанию можно изменить.)

Для просмотра отчета по требованию выберите нужный отчет в соответствующей иерархии папок. В данном случае для доставки отчета сервер отчетов создает временным моментальный снимок, который удаляется после доставки отчета.

Существует несколько способов выполнения отчета по требованию. В одном из таких способов указывается, что отчет запрашивает соответствующий источник данных каждый раз, когда пользователь запускает отчет. В этом случае при каждом выполнении отчета создается новый экземпляр отчета.

Если требуется повысить производительность, следуют пользоваться кэшированными отчетами. (Кэшированные отчеты рассматриваются в *этой главе далее*.)

Отчеты по подписке

Для предоставления отчетов по требованию, каждый раз, когда необходимо просмотреть результаты отчета, этот отчет нужно выбрать. В противоположность этому, доступ к отчетам, основанный на подписке, позволяет автоматически генерировать отчеты и доставлять их по назначению.

Службы SQL Server Reporting Services поддерживают два вида подписки на отчеты:

- стандартные подписки;
- подписки, управляемые данными.

Эти типы подписки рассматриваются в следующих подразделах.

Стандартные подписки

Стандартная подписка обычно состоит из конкретных параметров для параметризованных отчетов, а также из опций для представления и доставки отчета. Управлять стандартными подписками можно посредством разных инструментов. Обычно для этого используется диспетчер отчетов Report Manager.

Чтобы создать новую подписку, откройте диспетчер отчетов и найдите отчет, для которого нужно создать подписку. Наведите указатель мыши на отчет и нажмите на стрелку вниз. В выпадающем меню выберите пункт **Subscribe**. В результате откроется страница отчета **New Subscription**.

Здесь первым шагом нужно будет указать способ доставки, который можно выбрать между доставкой по e-mail или с помощью разделяемого файла. Если выбран метод разделяемого файла, необходимо предоставить имя файла, его путь, формат представления и информацию о безопасности.

Далее нужно настроить расписание. Расписание можно задать только для определенного отчета или же использовать общее расписание для нескольких подписок. (Для отчетов с параметрами нужно определить значения, присваиваемые параметрам при запуске подписки.)

Подписки, управляемые данными

Подписки, управляемые данными, доставляют отчеты списку пользователей, которые определяются в процессе выполнения отчета. Этот тип подписки отличается от стандартной подписки способом получения информации о подписке: некоторые параметры из источника данных предоставляются в процессе выполнения, а другие являются статическими (они предоставляются из определения подписки). Статические аспекты управляемой данными подписки включают такие, как доставляемый отчет, расширение доставки, информация о подключении к внешнему источнику данных, который содержит данные о подписчике, и запрос. Динамические параметры подписки находятся в наборе строк, выдаваемых запросом, включая список подписчиков и специфические для подписчиков предпочтения расширений доставки или значения параметров.

ПРИМЕЧАНИЕ

Для реализации подписок, управляемых данными, требуется SQL Server Enterprise Edition.

Далее, в следующем разделе, мы рассмотрим обработку отчетов.

Параметры доставки отчетов

Обработка отчета начинается с публикации определения отчета, которая включает запрос, информацию о макете и код. Обработка отчета и данных совместно создают набор данных с информацией о макете, которая сохраняется в промежуточном формате. После завершения обработки отчет компилируется в виде сборки CLR и отправляется на сервер отчетов.

При развертывании отчета на сервере отчетов он по умолчанию настраивается для выполнения по требованию. Использование отчетов по требованию рассмотрено в *разд. "Отчеты по требованию" ранее в этой главе*. В этом же разделе рассматриваются две следующие дополнительные возможности, которые вы можете использовать для выполнения отчетов:

- кэширование отчетов;
- выполнение моментальных снимков.

Кэширование отчетов

Кэширование отчета означает, что отчет генерируется только для первого пользователя, который его открывает, а затем он извлекается из кэша всеми последующими пользователи, которые работают с тем же отчетом. Сервер отчетов может кэши-

ровать копию обработанного отчета и возвращать эту копию, когда пользователь открывает отчет. Единственное, что указывает пользователю на то, что отчет получен из кэша, — это дата и время выполнения отчета. Если же дата и время нетекущие, а также отчет не является моментальным снимком, тогда этот отчет был получен из кэша.

Как вы уже, наверное, догадались, кэширование отчета сокращает время для его получения, поэтому кэширование рекомендуется применять с часто используемыми или большими отчетами. При перезапуске сервера все кэшированные экземпляры отчетов восстанавливаются при запуске веб-службы Reporting Services.

Содержимое кэша является непостоянным и может изменяться при добавлении новых отчетов и удалении старых. Для более предсказуемой стратегии кэширования следует использовать моментальные снимки, которые рассматриваются в следующем разделе.

Выполнение моментальных снимков

Основной недостаток кэшированных отчетов состоит в том, что для первого использования определенного отчета нужно ожидать, пока система создаст его. Более удобным для пользователя подходом было бы автоматическое создание отчета системой, чтобы даже при первом использовании отчета не нужно было бы ожидать его создания. Реализация такого подхода возможна посредством выполнения моментальных снимков.

Выполнение моментального снимка отчета позволяет создавать кэшированные отчеты, содержащие данные, захваченные в определенный момент времени. Преимущество моментального снимка состоит в том, что пользователю не нужно ожидать выполнения отчета, поскольку данные уже были получены из источника данных отчета и помещены во временную базу данных сервера отчетов. Таким образом, отчет составляется в течение очень короткого времени. Недостатком моментальных снимков отчетов является то, что при достаточно большом периоде времени между созданием и просмотром моментального снимка отчета, содержащиеся в нем данные могут устареть.

Основное различие между моментальными снимками отчетов и кэшированными отчетами состоит в том, что кэшированные отчеты создаются в результате действий пользователя, а моментальные снимки создаются системой автоматически.

Резюме

Службы SQL Server Reporting Services — это корпоративное средство для работы с отчетами, которое основано на SQL Server. Для создания отчетов можно использовать мастер Report Server Project Wizard или построитель отчетов Report Builder. Определение отчета, которое состоит из необходимого запроса, информации о макете и кода, сохраняется с помощью языка определения отчетов RDL (Report Definition Language), основанного на языке XML. Службы SSRS преобразовывают определение отчета в один из стандартных форматов, таких как HTML или PDF.

Отчеты могут предоставляться по требованию или по подписке. При выполнении отчета по требованию новый экземпляр отчета создается при каждом выполнении. Отчеты по подписке могут быть стандартными или управляемыми данными. Отчеты на основе стандартной подписки обычно состоят из конкретных параметров, а также из опций для представления и доставки отчетов. Подписка, управляемая данными, доставляет отчеты списку пользователей, который определяется при исполнении отчета.

Для сокращения времени доставки отчета можно использовать кэширование отчетов или их моментальные снимки, тем самым повышая производительность. Основное различие между моментальными снимками отчетов и кэшированными отчетами состоит в том, что кэшированные отчеты создаются в результате действий пользователя, а моментальные снимки создаются системой автоматически.

В следующей главе рассматриваются методы оптимизации для бизнес-аналитики.

Упражнения

Упражнение 24.1

Выполните выборку табельных номеров и имен всех сотрудников с должностью `clerk`. На основе этого запроса создайте отчет матричного типа. Просмотрите отчет, используя диспетчер отчетов Report Manager.

Упражнение 24.2

Создайте запрос для выборки из базы данных `sample` имен и бюджетов проектов, в которых участвуют сотрудники отдела `Research`, чей табельный номер больше, чем `25000`. На основе этого запроса создайте отчет табличного типа. Просмотрите этот отчет в браузере.

Глава 25

Методы оптимизации для реляционной оперативной аналитической обработки

- **Секционирование данных**
- **Оптимизация запроса схемы типа "звезда"**
- **Индексы columnstore**

В этой главе рассматривается несколько методов оптимизации, применяемых в *реляционной оперативной аналитической обработке* (*Relational Online Analytical Processing — ROLAP*). Слово "реляционной" в определении обработки указывает на то, что эти методы можно применять только для *реляционного* хранения многомерных данных. В первой части этой главы обсуждаются обстоятельства, когда будет разумным сохранять все экземпляры сущности в одной таблице, а когда будет лучше, по причинам производительности, секционировать данные таблицы. После общего введения в секционирование данных и типы секционирования, поддерживаемые компонентом Database Engine, подробно рассматриваются шаги, которые необходимо выполнять для выполнения секционирования таблиц. Далее представляется несколько методов секционирования, которые могут помочь повысить уровень производительности системы, а после них приводится список важных рекомендаций по секционированию данных и индексов.

Во второй части этой главы объясняется метод, называемый оптимизацией запроса схемы типа "звезда". Предоставляется два примера для демонстрации случаев, в которых оптимизатор запросов применяет этот метод вместо обычных методов обработки соединений. Также объясняется роль битовой фильтрации.

В последней части главы обсуждается использование колоночных индексов (columnstore index). Рассматривается создание таких индексов и их использование для улучшения производительности группы запросов по данным хранилища данных. Также обсуждаются ограничения реализации этого метода работы с хранилищами данных в SQL Server 2012.

Секционирование данных

Самым легким и наиболее естественным способом проектирования сущности будет использовать одну таблицу. Кроме этого, если все экземпляры сущности принадлежат к таблице, тогда нет необходимости решать, где физически хранить ее строки, поскольку это делается автоматически системой баз данных. По этой причине, если пользователь не хочет выполнять какие-либо административные работы, связанные с хранением данных таблицы, он может этого не делать.

С другой стороны, одной из наиболее распространенных причин низкой производительности реляционных систем баз данных является конкуренция за данные, которые находятся на одном устройстве ввода/вывода. Это особенно справедливо в случае таблиц с очень большим количеством строк, порядка миллионов. В этом случае в системах с множественными центральными процессорами секционирование таблиц может повысить уровень производительности благодаря применению параллельных операций.

Используя секционирование данных, большие таблицы (и индексы тоже) можно разделить на меньшие части, которые легче поддаются управлению. Это позволяет выполнять в параллельном режиме многие операции, такие как загрузка данных и обработка запросов.

Секционирование также улучшает доступность всей таблицы. Поместив каждую секцию на отдельный диск, можно иметь доступ к одной части данных, даже если один или несколько дисков становятся недоступными. В таком случае все данные в доступных секциях можно использовать для операций чтения и записи. То же самое справедливо и для операций по техническому обслуживанию.

В случае секционированной таблицы, оптимизатор запросов может определить, где условия поиска в запросе относятся к строкам лишь в определенных секциях и, следственно, это может ограничить область поиска только этими секциями. Таким образом можно значительно повысить уровень производительности, поскольку оптимизатору запросов требуется анализировать только часть данных из всей сегментированной таблицы.

ПРИМЕЧАНИЕ

Секционирование данных повышает производительность только в случае таблиц очень большого объема, содержащих минимум несколько сотен тысяч строк.

Как компонент Database Engine секционирует данные

Таблицу можно секционировать, используя любой из ее столбцов. Такой столбец называется *ключом секционирования* (partition key). (Кроме того, является возможным использовать в качестве ключа секционирования группу столбцов.) Значения ключа секционирования используются для секционирования строк таблицы в разные файловые группы.

Двумя другими важными понятиями в плане секционирования являются схема секционирования и функция секционирования. *Схема секционирования* (partition schema) сопоставляет строки таблицы с одной или несколькими файловыми группами, а *функция секционирования* (partition function) определяет способ, которым это сопоставление выполняется. Иными словами, функция секционирования определяет алгоритм, который используется для направления строк в их физическое место хранения.

Компонент Database Engine поддерживает только один тип секционирования, называемый *секционированием по диапазонам* (range partitioning). Этот метод секционирования разделяет строки таблицы на секции на основе значения ключа секционирования. Таким образом, применение секционирования по диапазонам позволяет всегда знать, в какой конкретной секции будет храниться определенная строка.

ПРИМЕЧАНИЕ

Кроме секционирования по диапазонам существует несколько других типов секционирования, одним из которых является *хеш-секционирование*. В отличие от секционирования по диапазонам хеш-секционирование помещает последовательные строки в секции, применяя функцию хеширования к ключу секционирования. Компонент Database Engine хеш-секционирование не поддерживает.

Последовательность шагов для создания секционированных таблиц методом секционирования по диапазонам рассматривается в следующем разделе.

Шаги для создания секционированных таблиц

Прежде чем приступать к секционированию таблиц базы данных, необходимо выполнить следующие шаги:

1. Установить цели секционирования.
2. Определить ключ секционирования и количество секций.
3. Создать файловую группу для каждой секции.
4. Создать функцию секционирования и схему секционирования.
5. Создать секционированные индексы (факультативно).

Все эти шаги рассматриваются в последующих разделах.

Установление целей секционирования

Цели секционирования зависят от типа приложений, которые обращаются к таблице, подлежащей секционированию. Существует несколько следующих различных целей секционирования, каждая из которых в отдельности может быть достаточной для секционирования таблицы:

- увеличение производительности отдельных запросов;
- уменьшение конкуренции в доступе к данным;
- улучшение доступности данных.

Если основной целью секционирования является улучшение производительности отдельных запросов, тогда все строки таблицы следует распределить равномерно. Таким образом, системе баз данных не нужно будет ожидать получения данных из секции с бóльшим количеством строк, чем остальные секции. Кроме этого, если эти запросы обращаются к данным посредством сканирования значительных частей таблицы, тогда секционировать нужно только строки таблицы. Секционирование соответствующего индекса в таком случае только повысит накладные расходы.

Секционирование данных может понизить конкуренцию, когда несколько запросов выполняют сканирование индекса, для возвращения всего лишь нескольких строк таблицы. В таком случае таблицу и индекс следует секционировать, используя схему секционирования, которая позволяет каждому запросу устранить ненужные части со своего сканирования. Чтобы добиться этой цели, сначала следует выяснить, какие запросы обращаются к определенным частям таблицы, после чего секционировать строки таблицы таким образом, чтобы разные запросы обращались к различным частям таблицы.

Секционирование повышает уровень доступности базы данных. Помещая каждую секцию в свою собственную файловую группу, а каждую файловую группу на свой собственный диск мы повышаем уровень доступности данных, поскольку в случае отказа одного диска недоступными становятся только данные в этой секции. Пока системный администратор принимает меры по исправлению сбоя диска, доступ к другим секциям таблицы остается возможным.

Определение ключа секционирования и количества секций

Таблица может быть секционирована с помощью любого ее столбца. Значения ключа секционирования используются для секционирования строк таблицы в разные файловые группы. Чтобы получить самый лучший уровень производительности, каждую секцию следует хранить в отдельной файловой группе, а каждую файловую группу помещать на отдельный физический диск. Распределение данных по нескольким дисковым устройствам позволяет сбалансировать ввод/вывод и улучшить производительность, доступность и обслуживание запросов.

Для секционирования данных таблицы следует использовать столбец, который не подвержен частым изменениям. Если для этого использовать столбец, данные которого изменяются часто, любая операция обновления этого столбца может вынудить систему переместить модифицированные строки из одной секции в другую, на что может потребоваться значительное время.

Создание файловой группы для каждой секции

Для улучшения производительности, повышения доступности данных и облегчения технического обслуживания данные таблицы разносятся по разным файловым группам. Количество используемых файловых групп в основном зависит от доступного оборудования. В случае наличия нескольких центральных процессоров, данные следует секционировать таким образом, чтобы каждый центральный процессор обращался к одному дисковому устройству. Если компоненту Database

Engine предоставить возможность обрабатывать несколько секций одновременно, то время выполнения приложения будет значительно сокращено.

Каждая секция данных должна сопоставляться с файловой группой. Файловую группу можно создать с помощью инструкции `CREATE DATABASE` или `ALTER DATABASE`. В примере 25.1 приведен запрос для создания базы данных `test_partitioned`, содержащей одну основную и две вспомогательные файловые группы.

ПРИМЕЧАНИЕ

Прежде чем выполнять запрос примера 25.1, в нем необходимо откорректировать расположение файлов с расширением *mdf* и *ndf* в соответствии с файловой системой вашего компьютера.

Пример 25.1. Создание базы данных, содержащей файловые группы

```
USE master;
CREATE DATABASE test_partitioned
   ON PRIMARY
   (NAME='MyDB_Primary',
   FILENAME=
         'd:\mssql\PT_Test_Partitioned_Range_df.mdf',
      SIZE=2000,
      MAXSIZE=5000,
      FILEGROWTH=1),
FILEGROUP MyDB_FG1
      (NAME = 'FirstFileGroup',
      FILENAME =
         'd:\mssql\MyDB_FG1.ndf',
      SIZE = 1000MB,
      MAXSIZE=2500,
      FILEGROWTH=1),
FILEGROUP MyDB_FG2
   (NAME = 'SecondFileGroup',
   FILENAME =
   'f:\mssql\MyDB_FG2.ndf',
   SIZE = 1000MB,
   MAXSIZE=2500,
   FILEGROWTH=1);
```

Запрос, приведенный в примере 25.1, создает базу данных `test_partitioned`, которая содержит основную файловую группу `MyDB_Primary` и две вспомогательные файловые группы: `MyDB_FG1` и `MyDB_FG2`. Файловая группа `MyDB_FG1` хранится на диске D:, а файловая группа `MyDB_FG2` — на диске F:.

Для добавления файловых групп к существующей базе данных используется инструкция `ALTER DATABASE`. В примере 25.2 показано применение этой инструкции для создания новой файловой группы `MyDB_FG3` в базе данных `test_partitioned`.

Пример 25.2. Создание новой файловой группы в существующей базе данных

```
USE master;
ALTER DATABASE test_partitioned
   ADD FILEGROUP MyDB_FG3
GO
ALTER DATABASE test_partitioned
DD FILE
   (NAME = 'ThirdFileGroup',
   FILENAME =
   'G:\mssql\MyDB_FG3.ndf',
   SIZE = 1000MB,
   MAXSIZE=2500,
   FILEGROWTH=1)
TO FILEGROUP MyDB_FG3;
```

В примере 25.2 инструкция `ALTER DATABASE` создает в базе данных `test_partitioned` новую файловую группу `MyDB_FG3`. Вторая инструкция `ALTER DATABASE` добавляет в созданную файловую группу новый файл. Обратите внимание, что имя файловой группы, в которую добавляется новый файл, задается в опции `TO FILEGROUP`.

Создание функции секционирования и схемы секционирования

Следующим шагом после создания файловых групп будет создание функции секционирования, используя для этого инструкцию `CREATE PARTITION FUNCTION`. Синтаксис этой инструкции выглядит таким образом:

```
CREATE PARTITION FUNCTION function_name(param_type)
   AS RANGE [LEFT | RIGHT]
      FOR VALUES ([boundary_value [,...n ]])
```

Здесь параметр *function_name* определяет имя функции секционирования, а параметр *param_type* задает тип данных ключа секционирования. В параметре *boundary_value* указывается одно или несколько граничных значений для каждой секции секционированной таблицы или индекса, который использует функцию секционирования.

Инструкция `CREATE PARTITION FUNCTION` поддерживает две формы опции `RANGE`: `RANGE LEFT` и `RANGE RIGHT`. Опция `RANGE LEFT` задает верхнюю границу первой секции в качестве граничного условия. Соответственно, опция `RANGE RIGHT` задает в качестве граничного условия нижнюю границу в последней секции. Если опция не указана, то по умолчанию принимается `RANGE LEFT`.

Прежде чем определять функцию секционирования, необходимо указать таблицу, которая будет секционироваться. В примерах данной главы для этой цели используется таблица `orders`, которая создается в примере 25.3. Прежде чем выполнять запрос, приведенный в примере 25.3, проверьте, чтобы база данных `sample` не содержала бы таблицу `orders`. Если такая таблица уже имеется, то ее следует удалить.

Пример 25.3. Создание таблицы `orders` в базе данных `sample`

```
USE sample;
CREATE TABLE orders
   (orderid INTEGER NOT NULL,
   orderdate DATE,
   shippeddate DATE,
   freight money);
GO declare @i int, @order_id integer
      declare @orderdate datetime
      declare @shipped_date datetime
      declare @freight money
      set @i = 1
      set @orderdate = getdate()
      set @shipped_date = getdate()
      set @freight = 100.00
      while @i < 1000001
      begin
      insert into orders (orderid, orderdate, shippeddate, freight)
         values( @i, @orderdate, @shipped_date, @freight)
      set @i = @i+1
      end
```

В примере 25.3 таблица `orders` создается инструкцией `CREATE TABLE`, а последующий пакет операторов загружает в эту таблицу один миллион строк.

В примере 25.4 показано определение функции секционирования для таблицы `orders`.

Пример 25.4. Определение функции секционирования для таблицы `orders`

```
USE test_partitioned;
CREATE PARTITION FUNCTION myRangePF1 (int)
   AS RANGE LEFT FOR VALUES (500000);
```

Функция секционирования `myRangePF1` задает две секции и граничное значение `500 000`. Это означает, что все значения ключа секционирования меньшие или равные чем `500 000` будут помещены в первую секцию, а все значения большие чем `500 000` будут помещены во вторую секцию. (Обратите внимание, что граничное значение связано со значениями ключа секционирования, которым в данном случае является столбец `orderid` таблицы `orders`. Как мы увидим далее, имя ключа секционирования указывается в соответствующей инструкции `CREATE TABLE`.)

Созданная функция секционирования будет бесполезной, если ее не связать с конкретными файловыми группами. Как упоминалось ранее в этой главе, эту привязку можно осуществить посредством схемы секционирования, используя для указания связи между функцией секционирования и соответствующими файловыми группа-

ми инструкцию CREATE PARTITION SCHEME. В примере 25.5 показано создание схемы секционирования для функции секционирования из примера 25.4.

Пример 25.5. Создание схемы секционирования для функции секционирования

```
USE test_partitioned;
CREATE PARTITION SCHEME myRangePS1
   AS PARTITION myRangePF1
   TO (MyDB_FG1, MyDB_FG2);
```

В примере 25.5 создается схема секционирования, называемая myRangePS1. Согласно этой схеме, все значения слева от граничного значения (т. е. все значения меньшие, чем 500 000) будут помещены в файловую группу MyDB_FG1. Подобным образом, все значения справа от граничного значения будут помещены в файловую группу MyDB_FG2.

ПРИМЕЧАНИЕ

При определении схемы секционирования для каждой секции в обязательном порядке необходимо указать файловую группу, даже если в одной файловой группе будет храниться несколько секций.

Создание секционированной таблицы только немного отличается от создания обычной таблицы. Как можно предполагать, инструкция CREATE TABLE должна содержать имя схемы секционирования и имя столбца таблицы, используемого в качестве ключа секционирования. В примере 25.6 показана расширенная форма инструкции CREATE TABLE, которая применяется для секционирования таблицы orders.

Пример 25.6. Секционирование таблицы orders

```
USE test_partitioned;
CREATE TABLE orders
   (orderid INTEGER NOT NULL,
   orderdate DATETIME,
   shippeddate DATETIME,
   freight money)
ON myRangePS1 (orderid);
```

Предложение ON в конце инструкции CREATE TABLE указывает уже определенную (в примере 25.5) схему секционирования. С помощью этой схемы столбец orderid таблицы связывается с функцией секционирования, в которой определен тип данных (INT) ключа секционирования (см. пример 25.4).

Создание секционированных индексов

При секционировании данных таблицы можно также секционировать индексы, связанные с этой таблицей. Для секционирования индексов можно использовать суще-

ствующую схему секционирования для этой таблицы или другую схему. Когда для секционирования индексов и таблицы используется одна и та же функция секционирования и те же самые столбцы секционирования (в одинаковом порядке), тогда говорят, что таблица и индекс совмещены. Когда таблица и индексы совмещены, система баз данных может перемещать секции секционированных таблиц с большой эффективностью, поскольку для секционирования обоих объектов базы данных применяется один и тот же алгоритм. По этой причине в большинстве практических случаев рекомендуется использовать совмещенные индексы.

В примере 25.7 показано создание кластеризованного индекса для таблицы `orders`. Этот индекс совмещен с таблицей `orders`, поскольку для его секционирования применяется та же схема секционирования, что и для таблицы `orders`.

Пример 25.7. Создание совмещенного индекса

```
USE test_partitioned;
CREATE UNIQUE CLUSTERED INDEX CI_orders
   ON orders(orderid)
   ON myRangePS1(orderid) ;
```

Как вы можете видеть из примере 25.7, создание секционированного индекса для таблицы `orders` осуществляется с использованием расширенной формы инструкции `CREATE INDEX`. Эта форма инструкции `CREATE INDEX` содержит дополнительное предложение `ON`, в котором указывается схема секционирования. Если индекс нужно совместить с таблицей, то следует указать ту же самую схему секционирования, что и для соответствующей таблицы. Первое предложение `ON` является частью стандартного синтаксиса инструкции `CREATE INDEX` и указывает столбцы для индексирования.

Методы секционирования для повышения производительности системы

Имеется возможность значительно повысить уровень производительности системы, применив следующие методы секционирования:

- совместное размещение таблиц;
- использование операций поиска, поддерживающих секционирование;
- параллельное выполнение запросов.

Совместное размещение таблиц

Кроме секционирования таблицы вместе с соответствующими индексами, компонент Database Engine также поддерживает секционирование двух таблиц с использованием одной и той же функции секционирования. Такая форма секционирования означает, что строки обеих таблиц с одинаковыми значениями ключей секционирования хранятся вместе в определенном месте на диске. Эта концепция секционирования данных называется *совместным размещением* (collocation).

Допустим, что кроме таблицы orders (см. пример 25.3) имеется также таблица order_details, которая содержит любое количество строк для каждого уникального значения orderid в таблице orders. Если выполнить секционирование обеих таблиц, используя одну и ту же функцию секционирования по столбцам соединения orders.orderid и order_details.orderid, строки обеих таблиц с одинаковыми значениями столбцов orderid будут вместе сохранены на диске. Предположим, что в таблице orders имеется уникальный заказ с идентификационным номером 49031, для которого в таблице order_details содержится пять соответствующих строк. В случае применения совместного размещения, все шесть строк будут сохранены на диске вместе, рядом друг с другом. Та же самая процедура будет применена ко всем строкам этих таблиц с одинаковыми значениями столбцов orderid.

Этот метод позволяет получить значительное улучшение производительности, когда при обращении к нескольким таблицам соединяемые данные хранятся все в одном месте. В таком случае системе не требуется перемещать данные между разными секциями.

Использование операций поиска, поддерживающих секционирование

Обработчик запросов видит внутреннее представление секционированной таблицы в виде составного (многостолбцового) индекса, в котором внутренний столбец является главным. (Этот столбец, который называется partitionedID, представляет собой скрытый вычисляемый столбец, который используется внутренне системой для представления идентификатора секции, содержащей конкретную строку.)

Например, предположим, что имеется таблица tab с тремя столбцами: col1, col2 и col3. (Столбец col1 используется для секционирования таблицы, а столбец col2 имеет кластеризованный индекс.) Компонент Database Engine внутренне рассматривает такую таблицу, как не секционированную таблицу со схемой tab (partitionedID, col1, col2, col3) и с кластеризованным индексом по составному ключу (partitionedID, col2). Это позволяет оптимизатору запросов выполнять операции поиска по вычисляемому столбцу partitionedID в любой секционированной таблице или индексе. Таким образом можно повысить производительность значительного количества запросов в секционированных таблицах, вследствие предварительного исключения секции.

Параллельное выполнение запросов

В версиях сервера до SQL Server 2008 при запросах по нескольким секциям на каждую секцию выделяется один поток. Иными словами, одновременное обращение к одной секции нескольких потоков невозможно, вследствие чего работа над одной секцией в параллельном режиме также невозможна. (О параллельных запросах см. в *главе 15*.) Это может вызвать проблемы с производительностью на системах с несколькими центральными процессорами, если количество секций в таблице меньше, чем количество процессоров. В таком случае в обработке запроса будут принимать участие не все процессоры.

Начиная с версии сервера SQL Server 2008, компонент Database Engine поддерживает две стратегии выполнения запросов для планов параллельных запросов по секционированным объектам.

- *Стратегия одного потока на секцию*. Для выполнения параллельного плана запроса, который обращается к нескольким секциям, оптимизатор запросов выделяет по одному потоку на каждую секцию. Одна секция не разделяется между несколькими потоками, но несколько секций могут обрабатываться в параллельном режиме.
- *Стратегия нескольких потоков на секцию*. Оптимизатор запросов выделяет несколько потоков на секцию, независимо от количества секций, с которыми нужно работать. Иными словами, все имеющиеся потоки стартуют с первой секции и начинают ее сканирование. Когда поток достигает конца секции, он переходит на следующую секцию и начинает сканировать ее. Для перехода на другую секцию поток не ожидает, пока остальные потоки завершат свою работу над текущей секцией.

Выбор одной из этих двух стратегий оптимизатором запросов зависит от среды выполнения. Если запросы связаны с вводом/выводом и обращаются к бóльшему числу секций, чем уровень параллелизма, оптимизатор выбирает стратегию одного потока на секцию. Стратегия нескольких потоков на секцию выбирается в следующих случаях:

- секции равномерно распределены на нескольких дисках;
- запросы обращаются к меньшему количеству секций, чем число имеющихся потоков;
- существует значительная разница между размерами секций в пределах одной таблицы.

Руководство по секционированию таблиц и индексов

Для успешного секционирования таблиц и индексов рекомендуются следовать перечисленным далее общим правилам.

- Не нужно секционировать все таблицы, а только те, к которым наиболее часто выполняется обращение.
- Возможность секционирования таблицы следует рассматривать только в случае таблиц очень большого размера, т. е. таблиц, содержащих, по крайней мере, несколько сотен тысяч строк.
- Для улучшения производительности следует использовать секционированные индексы, чтобы понизить конкуренцию между сеансами.
- Следует сохранять равновесие между количеством секций и количеством процессоров в системе. Если невозможно достичь соотношения 1:1 между количеством секций и числом процессоров, то следует задавать такое количество секций, которое будет кратным количеству процессоров. (Иными словами, если

компьютер оснащен четырьмя процессорами, количество задаваемых секций должно делиться на четыре.)

- Для секционирования данных таблицы не следует использовать столбец, который подвержен частым изменениям. Если секционировать таблицу по столбцу, данные которого изменяются часто, любая операция обновления этого столбца может вынудить систему переместить модифицированные строки из одной секции в другую, на что может потребоваться значительное время.
- Для оптимальной производительности, таблицы следует секционировать для повышения уровня параллелизма, но их индексы секционировать не нужно. Индексы следует помещать в отдельную файловую группу.

Оптимизация запроса схемы типа "звезда"

Как уже упоминалось в *главе 21*, схема типа "звезда" является общей формой структурирования данных в хранилище данных. Такая схема обычно имеет одну таблицу фактов, которая соединена с несколькими таблицами измерений. Таблица фактов может иметь 100 миллионов строк и более, тогда как таблицы измерений, по сравнению с размером соответствующей таблицы фактов, довольно небольшие. В запросах для поддержки принятия решений обычно несколько таблиц измерений соединяются с соответствующей таблицей фактов. Общепринятым способом выполнения таких запросов оптимизатором является соединение каждой из используемых в запросе таблиц измерений с таблицей фактов, используя зависимость первичный ключ/внешний ключ. Хотя этот способ является самым лучшим в случае большого числа запросов, уровень производительности можно значительно повысить, если оптимизатор запросов будет использовать специальные способы выполнения для определенной группы запросов. Одним из таких специальных способов является оптимизация запроса схемы типа "звезда".

Но прежде чем приступить к исследованию этого нового метода, взглянем, как оптимизатор запросов выполняет запрос, приведенный в примере 25.8.

Пример 25.8. Запрос для демонстрации обычного выполнения оптимизатором

```
USE AdventureWorksDW;
SELECT ProductAlternateKey
   FROM FactInternetSales f JOIN DimDate t ON f.OrderDateKey = t.DateKey
   JOIN DimProduct d ON d.ProductKey = f.ProductKey
   WHERE CalendarYear BETWEEN 2003 AND 2004
   AND ProductAlternateKey LIKE 'BK%'
   GROUP BY ProductAlternateKey, CalendarYear;
```

План выполнения этого запроса показан на рис. 25.1.

Как можно видеть в плане выполнения на рис. 25.1, запрос сначала соединяет таблицу фактов `FactInternetSales` с таблицей измерений `DimDate`, используя связь ме-

жду первичным ключом в таблице измерений (DateKey) и внешним ключом в таблице фактов (DateKey). После этого вторая таблица измерений DimProduct соединяется с таблицей фактов подобным образом. В конце запроса оба временных результата соединяются, используя метод соединения хешированием.

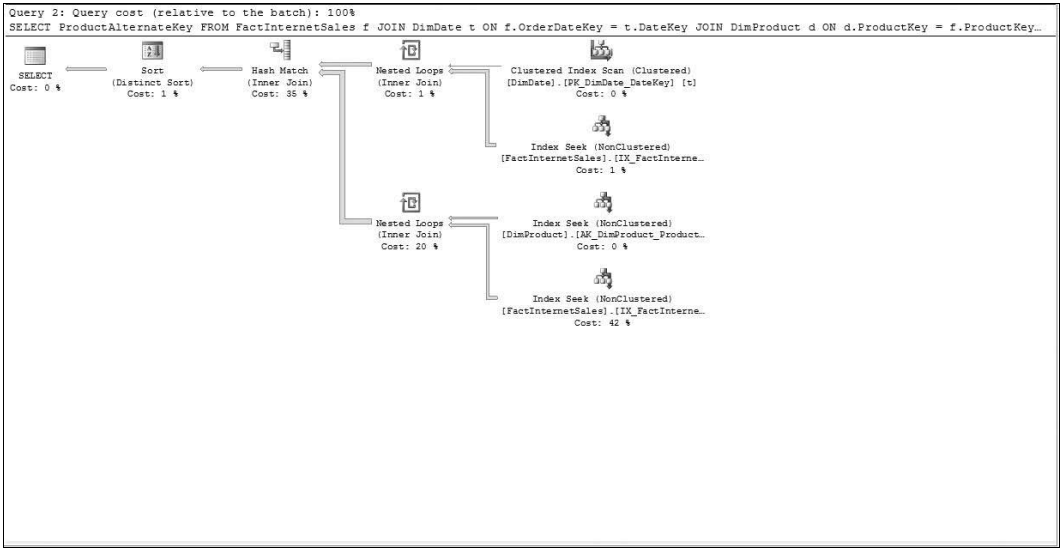


Рис. 25.1. План выполнения запроса примера 25.8

Теперь рассмотрим применение метода оптимизации запроса схемы типа "звезда" на примере запроса в примере 25.9.

Пример 25.9. Запрос оптимизации схемы типа "звезда"

```
USE  AdventureWorksDWMod;
GO
SELECT F.ProductKey, F.CurrencyKey, D1.CurrencyName, D2.EndDate
FROM dbo.FactInternetSales AS F
JOIN dbo.DimCurrency AS D1 ON F.CurrencyKey = D1.CurrencyKey
JOIN dbo.DimProduct D2 ON F.ProductKey = D2.ProductKey
WHERE D1.CurrencyKey <= 12 AND D2.ListPrice > 50
OPTION (MAXDOP 32);
```

Оптимизатор запросов использует метод оптимизации запроса схемы типа "звезда" только в тех случаях, когда размер таблицы фактов очень большой по сравнению с соответствующими таблицами измерений. Чтобы обеспечить применение оптимизатором запросов метода оптимизации запроса схемы типа "звезда", таблица фактов FactInternetSales базы данных AdventureWorksDW была существенным образом расширена. Исходная таблица содержит приблизительно 64 000 строк, но для этого примера в нее были добавлены еще 500 000 строк, генерируя случайные значения для столбцов ProductKey, SalesOrderNumber и SalesOrderLineNo. План выпол-

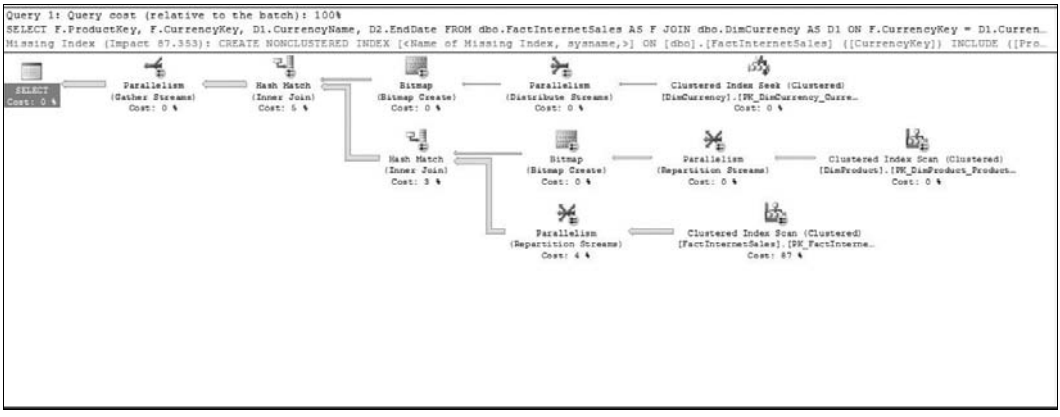


Рис. 25.2. План выполнения запроса примера 25.9

нения запроса в примере 25.9 для модифицированной таким образом таблицы фактов показан на рис. 25.2.

Оптимизатор запросов обнаруживает, что для данного запроса можно применить метод оптимизации запроса схемы типа "звезда", и выполняет оценку использования битовых фильтров. (Битовый фильтр представляет собой набор значений, размером от небольшого до среднего, который применяется для фильтрации данных. Битовые фильтры всегда хранятся в памяти.)

Как можно видеть в плане исполнения запроса на рис. 25.2, сначала сканируется таблица фактов, используя для этого соответствующий кластеризованный индекс. Затем к обеим таблицам применяются битовые фильтры. (Задача битовых фильтров заключается в том, чтобы отфильтровать строки из таблицы фактов.) Для этого был использован метод соединения хешированием. В конце выполнения значительно уменьшенные в размерах наборы строк из обоих потоков соединяются вместе.

ПРИМЕЧАНИЕ

Не путайте битовые фильтры с битовыми индексами! Битовые индексы представляют собой персистентные структуры, которые используются в бизнес-аналитике в качестве альтернативы структур B^+-деревьев. Компонент Database Engine не поддерживает битовые индексы.

Колончатые индексы

Как вы уже знаете из *главы 10*, доступ посредством индексов означает использование индексов для обращения ко всем строкам, которые удовлетворяют условию данного запроса. Это общий подход, в котором количество возвращаемых столбцов не имеет значения. Иными словами, возвращаются значения всех столбцов выбранных строк, даже если требуются значения только одного или двух столбцов. Причиной этому является то обстоятельство, что компонент Database Engine, как и все другие реляционные системы баз данных, сохраняет строки таблиц в страницах

данных. Этот традиционный подход к хранению данных называется *построчным хранением* (row store).

Новый подход к хранению данных обещает улучшить производительность в случаях, когда нужно возвратить значения только нескольких столбцов таблицы. Этот подход называется *постолбцовым хранением* (column store) и реализуется в SQL Server 2012 посредством использования *колоночного индекса* (columnstore index). При постолбцовом хранении данные группируются и сохраняются *по одному столбцу*. Обработчик запросов системы баз данных, которая поддерживает постолбцовое хранение, может воспользоваться таким размещением данных и значительно сократить время выполнения запросов, которые возвращают только некоторые из столбцов таблицы.

Работа с колоночными индексами

Колоночные индексы (columnstore index) можно создавать, используя следующие методы:

- инструкции языка Transact-SQL;
- среду SQL Server Management Studio.

Оба эти метода рассматриваются в последующих разделах.

Создание колоночных индексов посредством Transact-SQL

Для создания колоночных индексов применяется хорошо известная инструкция языка Transact-SQL `CREATE INDEX` с небольшими модификациями. В примере 25.10 показано создание такого индекса для таблицы `FactInternetSales` базы данных `AdventureWorksDW`.

Пример 25.10. Создание колоночного индекса

```
USE AdventureWorksDW;
GO
CREATE NONCLUSTERED COLUMNSTORE INDEX cs_index1
   ON FactInternetSales (OrderDateKey, ShipDateKey, UnitPrice);
```

Как можно видеть в этом запросе, для создания колоночного индекса стандартная инструкция `CREATE INDEX` расширяется предложением `COLUMNSTORE`. Но обратите внимание на то обстоятельство, что из многих опций для стандартных индексов можно использовать для колоночных индексов только две: `MAXDOP` и `DROP_EXISTING`. (Опция `MAXDOP` задает максимальный уровень параллелизма, а опция `DROP_EXISTING` применяется для перестроения индекса.)

Инструкция `CREATE INDEX` в примере 25.10 создает колоночный индекс для трех столбцов таблицы `FactInternetSales`: `OrderDateKey`, `ShipDateKey` и `UnitPrice`. Это означает, что значения для каждого из этих трех столбцов будут сгруппированы и сохранены отдельно.

В примере 25.11 приведен простой аналитический запрос по столбцу `UnitPrice` таблицы `FactInternetSales`.

Пример 25.11. Простой аналитический запрос

```
USE AdventureWorksDW;
GO
SET STATISTICS TIME ON;
GO
SELECT AVG (UnitPrice)
   FROM FactInternetSales;
```

При выполнении этого запроса с отсутствующим колоночным индексом `cs_index1` (см. пример 25.10) возвратится следующее сообщение:

```
Table 'FactInternetSales'. Scan count 5, logical reads 25411,
physical reads 2, read-ahead reads 23230, lob logical reads 0,
lob physical reads 0, lob read-ahead reads 0.
SQL Server Execution Times: CPU time = 280 ms, elapsed time = 9181 ms.
```

А при выполнении этого запроса, когда для таблицы `FactInternetSales` был создан колоночный индекс `cs_index1`, произойдет существенное повышение уровня производительности, как по времени выполнения, так и по числу операций логического считывания:

```
Table 'FactInternetSales'. Scan count 1, logical reads 779,
physical reads 1, read-ahead reads 2882, lob logical reads 0,
lob physical reads 0, lob read-ahead reads 0.
SQL Server Execution Times: CPU time = 109 ms, elapsed time = 155 ms.
```

Создание колоночных индексов посредством среды SQL Server Management Studio

Для создания колоночного индекса в среде SQL Server Management Studio разверните в обозревателе объектов дерево таблицы, для которой создается индекс, щелкните правой кнопкой папку **Indexes** и из контекстного меню выберите **New Index | Nonclustered Columnstore Index**. В открывшемся мастере нажмите кнопку **Add**, установите флажки для столбцов, которые будут использоваться для постолбцового хранения, и нажмите кнопку **OK**.

Преимущества и недостатки колоночных индексов

Как метод повышения производительности, колоночные индексы предоставляют значительное увеличение производительности только для группы запросов. Достоинства и недостатки этого метода рассматриваются в следующих двух подразделах.

Достоинства колоночных индексов

Перечислим некоторые достоинства колоночных индексов.

- *Система извлекает только требуемые столбцы.* Чем меньше количество извлекаемых столбцов, тем меньше требуется операций ввода/вывода. Например, если извлекается только 10% длины каждой строки, использование колоночных индексов может значительно уменьшить объем операций ввода/вывода, поскольку переносить с диска в память нужно только небольшую часть данных. (Это особенно справедливо для хранилищ данных, таблицы которых обычно имеют миллионы строк.)
- *Оптимальное сжатие значений.* При хранении данных построчно сжатие данных не является оптимальным. Причиной этому являются разные формы столбцов таблицы: некоторые из них содержат числовые данные, а другие строчные или временные данные. Большинство алгоритмов сжатия основаны на использовании сходства групп значений. Когда данные сохраняются построчно, возможность использования схожести значений ограничена вследствие их разных типов. В противоположность этому, при постолбцовом хранении сохраняются данные одного типа. Обычно при этом наблюдается повторение и сходство данных сохраняемого столбца, что обеспечивает более эффективную работу алгоритмов сжатия.
- *Значительное ускорение времени выполнения запросов с особыми характеристиками.* Как показано в примере 25.11, запрос по нескольким столбцам, для которых создан колоночный индекс, выполняется намного быстрее, чем обычный запрос по всем столбцам строки. (Очевидно, что производительность понижается с увеличением количества столбцов в списке выборки столбцов инструкции `SELECT`.)
- *Отсутствие ограничений на количество ключевых столбцов.* Понятие ключевых столбцов применимо только к построчному хранению. Поэтому ограничение на количество ключевых столбцов для индекса не относится к колоночным индексам. Кроме этого, если базовая таблица имеет кластеризованный индекс, некластеризованный колоночный индекс должен содержать все столбцы ключа кластеризации. В противном случае он будет добавлен в колоночный индекс автоматически.
- *Колоночные индексы можно применять с секционированными таблицами.* При этом не требуется никоим образом изменять синтаксис для секционирования таблицы. Колоночный индекс для секционированной таблицы должен быть выровнен по секциям с базовой таблицей. Таким образом, некластеризованный колоночный индекс для секционированной таблицы можно создать только в том случае, если столбец секционирования является одним из столбцов колоночного индекса.

Недостатки колоночных индексов

Поддержка колоночных индексов была впервые предоставлена в SQL Server 2012. Поэтому нельзя ожидать их оптимальной реализации.

Далее приводится список некоторых ограничений при использовании колоночных индексов.

- *Таблица с колоночным индексом доступна только для чтения.* Таблицу с колоночным индексом нельзя обновлять. (Этот вопрос не является существенным для систем хранилищ данных, поскольку в них операции обновления выполняются нечасто.) В электронной документации рассматривается несколько способов обхода этого ограничения.
- *Колоночные индексы поддерживают не все типы данных.* Сервер SQL Server 2012 поддерживает создание колоночных индексов для распространенных деловых типов данных, таких как `CHAR`, `VARCHAR`, `INT`, `DECIMAL` и `FLOAT`. Колоночные индексы не могут содержать, среди прочих, следующие типы данных: `BINARY`, `VARBINARY`, `VARCHAR(max)` и `SQL_VARIANT`.
- *Существует несколько ограничений для кластеризованных и некластеризованных колоночных индексов.* В настоящее время для таблицы невозможно создать больше одного некластеризованного колоночного индекса, а кластеризованные колоночные индексы в SQL Server 2012 не поддерживаются вообще. Ожидается, что в будущих версиях SQL Server оба эти ограничения будут устранены.

Резюме

Компонент Database Engine поддерживает секционирование по диапазонам данных и индексов, которое для приложений является совершенно прозрачным. Секционирование по диапазонам разделяет строки таблицы на секции на основе значения ключа секционирования. Иными словами, данные разделяются, используя значения ключа секционирования.

Для секционирования данных нужно выполнить следующие шаги:

1. Установить цели секционирования.
2. Определить ключ секционирования и количество секций.
3. Создать файловую группу для каждой секции.
4. Создать функцию секционирования и схему секционирования.
5. Создать секционированные индексы (если необходимо).

Для улучшения производительности, повышения доступности данных и облегчения технического обслуживания данные таблицы разносятся по разным файловым группам.

Функция секционирования используется для разнесения строк таблицы или индекса по секциям на основе значений заданного столбца. Для создания функции секционирования используется инструкция `CREATE PARTITION FUNCTION`. Для привязки функции секционирования к конкретной файловой группе применяется схема секционирования. При секционировании данных таблицы можно также секционировать индексы, связанные с этой таблицей. Для секционирования индексов можно использовать существующую схему секционирования для этой таблицы или другую схему.

Оптимизация запросов схемы типа "звезда" является методом оптимизации на основе индексов, который поддерживает оптимальное использование индексов таблиц фактов очень большого размера. Этот метод оптимизации предоставляет следующие преимущества:

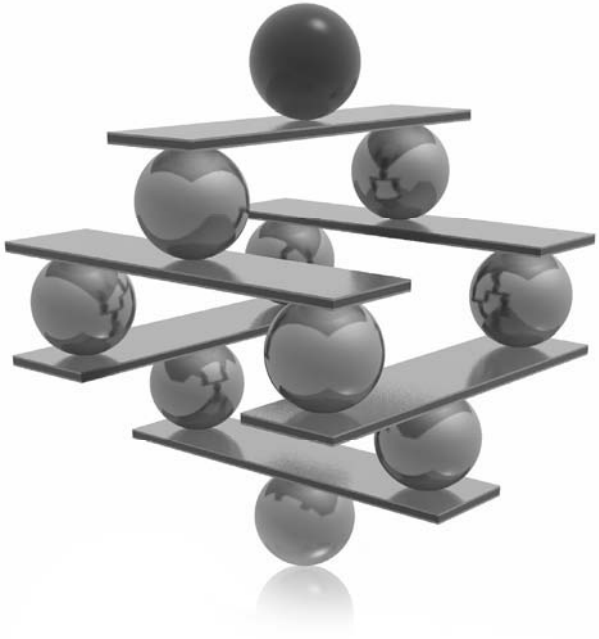
- значительное повышение уровня производительности при высокой и средней селективности запросов к схеме типа "звезда";
- отсутствие дополнительных расходов по хранению; эта система не создает никаких новых индексов, а использует битовые фильтры.

Сервер SQL Server 2012 поддерживает колоночные индексы, что предоставляет новый способ для хранения значений столбцов таблиц. Вследствие многих ограничений в текущей версии, использование этого очень обещающего метода следует ограничивать. Иными словами, если этот подход планируется использовать с рабочими базами данных, использование колоночных индексов нужно сначала всесторонне проверить на тестовых базах данных.

Итак, эта глава является последней главой *части IV* книги, посвященной бизнес-аналитике. Следующая глава начинает заключительную часть книги, в которой представлено введение в язык XML.

Часть V

За пределами реляционных данных



Глава 26

SQL Server и XML

- **Основные понятия XML**
- **Языки схем**
- **Хранение XML-документов в SQL Server**
- **Представление данных**
- **Запрашивание данных**

Эта глава состоит из четырех основных частей. В первой части рассматривается расширяемый язык разметки *XML* (Extensible Markup Language), который становится все более важным форматом для хранения данных. В этой части также описываются требования к правильно сконструированному документу, а также объясняются основные концепты XML. Во второй части представляются два средства описания схем: язык определения типа документа *DTD* (Document Type Definition) и язык XML Schema.

В третьей части рассматривается язык XML в отношении к системам баз данных в общем и в отношении к компоненту Database Engine в частности. После этого обсуждается наиболее важная форма хранения данных, используя тип данных `XML`. В четвертой, последней, части главы описывается извлечение XML-документов, используя системные хранимые процедуры, после чего следует обсуждение представления реляционных данных в XML. В конце этой части вкратце рассматривается язык XQuery и методы XQuery, поддерживаемые в SQL Server.

Основные концепции XML

Язык XML является HTML-подобным языком и применяется для обмена данными и цифрового представления данных. Как HTML, так и XML является языком разметки, что означает, что в обоих этих языках применяются теги для представления логической структуры данных, и оба эти языка представляют важность для функционирования Всемирной сети. Но в отличие от языка HTML, который имеет фик-

сированное количество тегов, каждый из которых имеет свое собственное значение, набор тегов XML не установлен наперед, а его теги не имеют установленных семантических значений. Краткое описание языка HTML приводится в *разд. "Родственные XML языки" далее в этой главе*.

В этом разделе сначала рассматриваются требования к правильно сконструированным XML-документам, после чего представляются три основные компонента языка XML: элементы, атрибуты и пространства имен. Далее дается краткое введение во Всемирную сеть, после чего следует обсуждение языков, родственных языку XML.

Требования к правильно сформированному документу XML

В следующем списке перечислены требования, которым должен отвечать правильно сформированный XML-документ, образец которого показан в примере 26.1:

- наличие корневого элемента (`PersonList` в примере 26.1);
- каждый открывающий тег имеет соответствующий закрывающий тег;
- правильное вложение элементов документа;
- атрибут должен иметь значение, которое берется в кавычки.

Пример 26.1. Правильно сформированный XML-документ

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonList Type="Employee">
   <Title> Value="Employee List"></Title>
   <Contents>
      <Employee>
         <Name>Ann Jones</Name>
         <No>10102</No>
         <Deptno>d3</Deptno>
         <Address>
            <City>Dallas</City>
            <Street>Main St</Street>
         </Address>
      </Employee>
      <Employee>
         <Name>John Barrimore</Name>
         <No>18316</No>
         <Deptno>d1</Deptno>
         <Address>
            <City>Seattle</City>
            <Street>Abbey Rd</Street>
         </Address>
      </Employee>
   </Contents>
</PersonList>
```

Как показано в примере 26.1, XML-документ обычно состоит из трех частей:

- необязательная первая строка, в которой указывается версия XML, на основе которой составлен данный документ (в примере 26.1 это версия 1.0);
- необязательная внешняя схема (обычно задается с использованием языка DTD или языка XML Schema; см. *разд. "Языки описания схем" далее в этой главе*);
- корневой элемент — элемент, содержащий все остальные элементы документа.

Наиболее важным компонентом XML-документов являются элементы, которые рассматриваются в следующем разделе.

ПРИМЕЧАНИЕ

Кроме XML-элементов, атрибутов и пространств имен, которые подробно рассматриваются в последующих разделах, XML-документы также содержат другие компоненты, такие как примечания и инструкции по обработке данных. Эти компоненты не рассматриваются в этой книге, поскольку они выходят за рамки ее тематики.

Элементы языка XML

Язык XML применяется для цифрового представления документов. Чтобы представить документ, необходимо знать его структуру. Например, если рассматривать в качестве документа книгу, то ее сначала можно разбить на главы (с заголовками). Каждая глава содержит несколько разделов (со своими заголовками и соответствующими рисунками), а каждый раздел состоит из абзацев.

Все части XML-документа, входящие в его логическую структуру (такие как главы, разделы и абзацы в случае книги), называются *элементами*. Таким образом, в языке XML каждый элемент представляет компонент документа. В примере 26.1 `PersonList`, `Title` и `Employee` являются образцами элементов XML. Кроме этого, каждый элемент может содержать другие элементы. Части элемента, которые не входят в логическую структуру документа, называются *символьными данными*. Например, слова и предложения книги можно рассматривать, как символьные данные.

Все элементы документа создают иерархию элементов, которая называется *древовидной структурой документа*. Каждая такая структура имеет элемент верхнего уровня, который содержит все остальные элементы. Этот элемент называется *корневым элементом*. Элементы, которые не содержат вложенных элементов, называются *листьями*.

ПРИМЕЧАНИЕ

В отличие от HTML, где действительные теги определены спецификацией языка, имена тегов в языке XML выбираются программистом.

Элементы XML, непосредственно вложенные внутри других элементов, называются *дочерними элементами элемента*, в который они вложены. Например, в приме-

ре 26.1, элементы `Name`, `No` и `Address` являются дочерними элементами элемента `Employee`, который в свою очередь является дочерним элементом элемента `Contents`, а тот — дочерним элементом корневого элемента `PersonList`.

Любой элемент может содержать дополнительную информацию, которая называется *атрибутом* и описывает свойства этого элемента. Атрибуты применяются совместно с элементами для представления объектов. Общий синтаксис элемента с атрибутом и их значениями выглядит следующим образом:

```
<el_name attr_name="attr_value">el_value</el_name>
```

В представленной далее строке кода из примера 26.1:

```
<PersonList Type="Employee">,
```

`Type` является именем атрибута элемента `PersonList`, а `Employee` является значением этого атрибута.

Атрибуты подробно рассматриваются в *следующем разделе*.

Атрибуты XML

Атрибуты используются для представления данных. Так как элементы также могут использоваться с этой же целью, то возникает вопрос, а нужны ли атрибуты вообще, поскольку почти все, что можно делать с помощью атрибутов, можно делать с помощью элементов (или субэлементов). Но следующие задачи выполнимы только с помощью атрибутов:

- определение однозначного значения;
- принудительное обеспечение ограничения ссылочной целостности ограниченного типа.

ПРИМЕЧАНИЕ

Общего правила по определению данных не существует. Наилучшим практическим правилом является использование атрибутов для представления общих свойств элементов, а также применение субэлементов для представления конкретных свойств элементов.

Атрибут можно задать как атрибут в виде идентификатора. В рамках XML-документа значение атрибута типа `ID` должно быть уникальным. Поэтому для определения уникального значения можно использовать атрибут типа `ID`.

Атрибут типа `IDREF` должен ссылаться на действительный идентификатор, объявленный в этом же документе. Иными словами, значение атрибута типа `IDREF` в документе должно встречаться как значение соответствующего атрибута типа `ID`.

Атрибут типа `IDREFS` задает список разделенных пробелами строк, на которые ссылаются значения атрибута типа `ID`. Например, следующая строка является фрагментом атрибута `IDREFS` разметки XML:

```
<Department Members="10102 18316 "/>
```

(Для этого примера предполагается, что атрибут `No` элемента `Employee` является атрибутом типа `ID`, а атрибут `Members` элемента `Department` является атрибутом типа `IDRESFS`.)

С некоторыми отличиями, пары `ID/IDREF` и `ID/IDREFS` соответствуют связям "первичный ключ/внешний ключ" реляционной модели. В XML-документе значения разных атрибутов типа `ID` должны быть однозначными. Например, значения атрибутов `CustomerID` и `SalesOrderID` XML-документа должны быть разными.

ПРИМЕЧАНИЕ

`ID`, `IDREF` и `IDREFS` являются типами данных языка DTD, который рассматривается *далее в этой главе*.

Пространства имен XML

При использовании языка XML создается словарь терминов, соответствующих домену, в котором моделируются данные. В такой ситуации разные словари для различных доменов могут вызывать конфликты имен, когда эти домены требуется применять совместно в одном XML-документе. (Это обычно происходит при попытке объединить информацию, полученную из разных доменов.) Например, элемент `article` в одном домене может ссылаться на научные статьи, тогда как элемент с таким же именем в другом домене может означать предмет товара. Эта проблема решается с помощью пространств имен XML.

Обычно имя каждого тега XML требуется писать в виде *`namespace:name`*, где *`namespace`* указывает пространство имен XML, а *`name`* — имя тега XML.

Пространство имен всегда представляется глобальным уникальным идентификатором *URI* (Uniform Resource Identifier — унифицированный идентификатор ресурса), который обычно является адресом URL, но также может быть и абстрактным идентификатором.

В примере 26.2 показано использование двух пространств имен.

Пример 26.2. Использование пространств имен в XML

```
<Faculty xmlns="http://www.fh-rosenheim.de/informatik"
                 xmlns:lib="http://www.fh-rosenheim.de/library">
   <Name>Book</Name>
   <Feature>
   <lib:Title>Introduction to Database Systems</lib:Title>
   <lib:Author>A. Finkelstein</lib:Author>
   </Feature>
</Faculty>
```

Для определения пространств имен используется атрибут `xmlns`. В примере 26.2 задается два пространства имен. Первое из них является *пространством имен по умолчанию*, поскольку оно задано только посредством ключевого слова `xmlns`. По-

этому, это просто краткая форма для пространства имен **http://www.fh-rosenheim.de/informatik**. Второе пространство имен задано в виде `xmlns:lib`. Префикс `lib` является сокращением для **http://www.fh-rosenheim.de/library**.

Поэтому теги для этого пространства имен должны иметь префикс `lib:`. Теги без этого префикса относятся к пространству имен по умолчанию. (В примере 26.2 ко второму пространству имен относятся два тега: `Title` и `Author`.)

ПРИМЕЧАНИЕ

Одним из часто применяемых префиксов является префикс `xsd`. Как можно видеть в примере 26.8, этот префикс служит сокращением для ссылки на спецификацию XML Schema: **http://www.w3.org/2001/XMLSchema**.

XML и Всемирная паутина

Интернет стал доминирующим средством коммуникации вследствие его использования миллиардами людей для разнообразных видов деятельности, включая деловую, социальную, политическую, управленческую и образовательную деятельности. В основном, интернет-технология состоит из следующих четырех составляющих:

- веб-сервера;
- веб-браузера;
- языка HTML;
- протокола HTTP.

Веб-сервер отправляет страницы (обычно типа HTML) в Интернет. Веб-браузер получает страницы и отображает их на экране компьютера. Примером браузера может являться Internet Explorer корпорации Microsoft.

Отправляемые в Интернет документы создаются, используя язык разметки HTML. Этот язык позволяет форматировать данные, которые отображаются в веб-браузере. Одной из причин, по которым Интернет получил такое широкое распространение, является простота языка HTML. Но этот язык имеет один большой недостаток: он может передать сведения только о том, как должны выглядеть данные. Иными словами, этот язык используется только для описания расположения данных.

Протокол *HTTP* (Hypertext Transfer Protocol) "соединяет" веб-браузер с веб-сервером и используется для отправки запросов страниц и доставки запрошенных страниц. Если страница содержит гиперссылку, этот протокол применяется для подключения к серверу, на который указывает эта ссылка.

Родственные XML-языки

Язык XML является родственным двум другим языкам разметки:

- SGML;
- HTML.

Стандартный язык обобщенной разметки *SGML* (Standard General Markup Language) применяется для обмена большими и сложными документами. Этот язык используется во многих областях, для которых требуются сложные документы, например в области обслуживания самолетов. Как мы вскоре увидим, язык XML является облегченной версией языка SGML, т. е. это упрощенное подмножество языка SGML, которое в основном используется для Интернета.

Язык *HTML* (Hypertext Markup Language) является самым важным языком разметки, используемым в Интернете. Любой HTML-документ является SGML-документом, который имеет фиксированное определение типа документа. (Фиксированные типы документов рассматриваются в следующем разделе.) Таким образом, язык HTML — это всего лишь экземпляр языка SGML.

Документ HTML представляет собой текстовый файл, состоящий из набора тегов, заключенных в угловые скобки. Наиболее важными тегами являются теги гиперссылок, которые применяются для ссылки на документы, управляемые веб-сервером. Перекрестный набор таких ссылок и соответствующих документов и составляет то, что называется Интернетом.

Язык HTML обладает двумя важными особенностями:

- он используется только для форматирования документов;
- он не является расширяемым языком.

Язык HTML является языком разметки, который используется для описания того, как должны выглядеть данные. (С другой стороны, этот язык предоставляет гораздо больше возможностей, чем простой язык форматирования, такой как, например, язык LaTeX, поскольку его элементы являются обобщенными и описательными.)

Язык HTML содержит фиксированное число элементов, вследствие чего он не очень подходит для определенных типов документов.

Языки схем

В отличие от языка HTML, состоящего из набора фиксированных правил, которым нужно следовать при создании HTML-документа, язык XML не имеет таких правил, поскольку он предназначен для использования во многих различных областях применения. Поэтому язык XML содержит составляющие, которые используются для определения структуры или схемы документов. Наиболее важными средствами описания схем для XML-документов являются следующие:

- язык DTD;
- язык XML Schema.

Эти средства рассматриваются в последующих разделах.

Язык DTD

Язык определения типа документа *DTD* (Document Type Definition) — это набор правил для структурирования XML-документов. DTD можно указать как часть

XML-документа или же в виде адреса URL в XML-документе, указывающего место хранения DTD. Документ, который согласован со связанным DTD, называется *действительным документом*.

ПРИМЕЧАНИЕ

В языке XML нет требования, чтобы документы имели соответствующие DTD, но есть требование, чтобы документы были правильно сконструированными (как рассматривалось ранее в этой главе).

В примере 26.3 показан DTD для XML-документа из примера 26.1.

Пример 26.3. DTD для XML-документа из примера 26.1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE PersonList SYSTEM "C:\tmp\Unbenannt4.dtd">
<!ELEMENT EmployeeList (Title, Contents)>
<!ELEMENT Title EMPTY>
<!ELEMENT Contents (Employee*)>
<!ELEMENT Employee (Name, No, Deptno, Address)>
<!ELEMENT Name (Fname, Lname)>
<!ELEMENT Fname (#PCDATA)>
<!ELEMENT Lname (#PCDATA)>
<!ELEMENT No (#PCDATA)>
<!ELEMENT Deptno (#PCDATA)>
<!ELEMENT Address (City, Street) >
<!ELEMENT City (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ATTLIST EmployeeList Type CDATA #IMPLIED
                       Date CDATA #IMPLIED>
<!ATTLIST Title Value CDATA #REQUIRED>
```

ПРИМЕЧАНИЕ

Оба документа (XML-документ из примера 26.1 и DTD из примера 26.3) нужно связать друг с другом. Эту связь можно создать в XML-редакторе или вручную. В последнем случае нужно будет добавить соответствующую информацию в XML-документ.

DTD содержит несколько общих компонентов: имя (в примере 26.3 это `EmployeeList`) и набор инструкций `ELEMENT` и `ATTLIST`. Имя DTD должно соответствовать имени тега корневого элемента XML-документа (см. пример 26.1), для проверки правильности структуры которого применяется данный DTD. Кроме этого, XML-документ нужно связать с соответствующим файлом DTD.

Объявления типов элементов должны начинаться с инструкции `ELEMENT`, за которым следует имя определяемого элемента. (Каждый элемент действительного XML-документа должен соответствовать типу элемента, объявленного в DTD.) Кроме

этого, порядок элементов XML-документа должен отвечать порядку их указания в DTD. В примере 26.3 первая инструкция `ELEMENT` указывает, что элемент `EmployeeList` состоит из элементов `Title` и `Contents`, в указанном порядке. Элементы, которые не содержат никаких субэлементов, должны объявляться как текстовые, т. е. типа `#PCDATA`. (Элемент `Title` является таким типом элемента.)

Символ `*` (звездочка) в определении элемента `Contents` указывает на наличие элементов (от нуля и больше) типа `Employee`. (Кроме символа `*` в определениях используются символы `?` и `+`. Символ `?` указывает на наличие не более одного элемента, а символ `+` означает наличие по крайней мере одного элемента.)

Атрибуты для конкретных элементов объявляются, используя инструкцию `ATTLIST`. В частности, объявление атрибута начинается со строки `<!ATTLIST`, сразу же после которой следует имя атрибута и его тип данных. В примере 26.3 элементу `EmployeeList` разрешается иметь атрибуты `Type` и `Date`, тогда как элементу `Title` разрешается иметь только атрибут `Value`. (Все другие элементы атрибутов не имеют.)

Ключевое слово `#IMPLIED` означает, что соответствующий атрибут является необязательным, а ключевое слово `#REQUIRED` означает, что атрибут является обязательным.

ПРИМЕЧАНИЕ

Кроме типа данных `#PCDATA` язык DTD поддерживает несколько других типов данных, таких как `ID`, `IDREF` и `IDREFS`. Эти типы данных рассматриваются *ранее в этой главе*.

Кроме определения структуры документа, его форматирование также представляет важность. Для решения этой задачи язык XML использует расширяемый язык стилей *XSL* (Extensible Stylesheet Language), который позволяет описывать, каким образом следует форматировать и отображать данные документа.

ПРИМЕЧАНИЕ

Стиль документа описывается в отдельном модуле. Поэтому документ без такого модуля будет использовать форматирование браузера по умолчанию.

Язык XML Schema

Язык XML Schema является стандартизованным языком определения данных для XML-документов. Он позволяет определить набор базовых типов, которые поддерживаются как типы XML. Язык XML Schema содержит большое число продвинутых возможностей и поэтому является значительно более сложным, чем язык DTD.

ПРИМЕЧАНИЕ

По причине его сложности в этой книге язык XML Schema рассматривается только вкратце. Пример языка XML Schema приводится *далее в этой главе* (см. пример 26.8).

Основными особенностями языка XML Schema являются следующие:

- в нем используется такой же синтаксис, как и для XML-документов; по этой причине схемы сами являются правильно сформированными XML-документами;
- этот язык интегрирован в механизм пространства имен; хотя для пространства имен может существовать несколько документов определения схемы, документ определения схемы определяет тип только в одном пространстве имен;
- язык XML Schema предоставляет набор базовых типов, таким же образом, как и SQL предоставляет `CHAR`, `INTEGER` и другие базовые типы данных;
- в нем также поддерживаются ограничения первичного и внешнего ключей для обеспечения целостности данных.

Хранение XML-документов в SQL Server

Как упоминалось в *главе 1*, реляционная модель данных лучше всего подходит для использования со структурированными данными с определенной схемой. С другой стороны, для полуструктурированных данных нужно знать, как их моделировать. В таком случае язык XML будет хорошим выбором, поскольку он является платформно-независимой моделью, что обеспечивает переносимость полуструктурированных данных.

Целью всех современных систем баз данных, включая компонент Database Server, является хранение любого типа данных. Поэтому между реляционными базами данных и XML-документами существует тесная связь. Перед тем, как фокусироваться на хранении XML-документов в Database Engine в частности, рассмотрим вкратце общие методы хранения XML-документов в реляционных базах данных в общем. В реляционных базах данных XML-документы хранятся в трех общих форматах:

- как исходные ("raw") документы;
- разложенные на реляционные столбцы;
- использующие внутренний формат.

При сохранении XML-документа в виде объекта типа LOB (Large Object, большой объект) сохраняется точная копия данных. В этом случае XML-документы сохраняются в их исходном, так называемом "сыром" (raw), виде, т. е. в форме строк символов. Использование исходной формы данных позволяет очень просто сохранять документы. Такой документ легко поддается извлечению, если извлекать весь документ. Для извлечения частей документа нужно создавать специальные индексы.

Разложить XML-документ на отдельные столбцы одной или нескольких таблиц можно, используя его схему. В этом случае иерархическая структура документа сохраняется, тогда как порядок других элементов игнорируется. (Как уже упоминалось, реляционная модель не поддерживает упорядочение столбцов таблицы, тогда как элементы XML-документа упорядочены.) Хранение XML-документов в разло-

женном виде значительно облегчает индексирование элемента, если он помещен в отдельный столбец.

ПРИМЕЧАНИЕ

Разложение XML-документа на отдельные столбцы также называется "измельчением" (shredding).

Внутренний формат хранения (native storage) означает, что XML-документ сохранятся в разобранном (parsed) виде. Иными словами, для хранения документа используется внутреннее представление (например, XML Infoset), при котором сохраняется XML-содержимое данных. (Стандартизированная спецификация XML Infoset (XML Information Set) предоставляет набор единиц информации для использования в других спецификациях, которым нужно обращаться к информации в документе XML.)

Использование внутреннего формата хранения делает задачу запрашивания информации легкой, основанной на структуре XML-документа. Но с другой стороны, восстановление исходной формы XML-документа является довольно трудным делом, поскольку восстановленное содержимое может быть не совсем точной копией документа. (Обычно подробная информация о важных пробелах, порядке атрибутов и префиксов пространств имен в XML-документах не сохраняется.)

ПРИМЕЧАНИЕ

В последующем материале этой главы термин XML имеет два значения. Первое означает язык XML, а второе тип данных `XML` в компоненте Database Engine. Чтобы избежать путаницы, для обозначения языка применяется фраза "язык XML", а для обозначения данных — фраза "тип данных `XML`". Кроме этого, фраза "столбец `XML`" означает столбец, содержащий данные типа `XML`.

SQL Server поддерживает все три формата хранения XML-документов, рассмотренных ранее в этом разделе.

- *Формат исходного (raw) документа.* Для хранения XML-документов в исходном формате в Database Engine применяются типы данных `VARCHAR(MAX)` и `VARBINARY(MAX)`. Вследствие его сложности, этот подход в данной книге подробно не рассматривается.
- *Разложение (декомпозиция) на реляционные столбцы.* Компонент Database Engine может разлагать XML-документы на составные столбцы таблиц, используя системную процедуру `sp_xml_preparedocument`. Эта процедура выполняет синтаксический разбор документа и представляет его узлы в виде дерева. (Более подробно эта системная процедура рассмотрена в *разд. "Хранение XML-документов, используя разложение" далее в этой главе*.)
- *Внутренний формат.* Используя тип данных XML, документы XML можно сохранять во внутреннем формате в базе данных под управлением Database Engine. (Системы баз данных, такие как Database Engine, которые сохраняют XML-

документы в полностью разобранной форме, называются системами баз данных с внутренним XML.)

Хранение XML-документов в последних двух форматах подробно рассматривается в следующих разделах. Поскольку использование типа данных XML является наиболее важной формой хранения XML-документов, внутренний формат рассматривается первым.

Хранение XML-документов, используя тип данных *XML*

Тип данных XML является базовым типом данных в Transact-SQL, что значит, что этот тип данных можно использовать таким же образом, как и стандартные типы данных, такие как INTEGER или CHARACTER. С другой стороны, тип данных XML имеет некоторые ограничения, поскольку при определении столбца XML нельзя использовать предложения UNIQUE, PRIMARY KEY или FOREIGN KEY.

В основном, тип данных XML можно использовать для определения следующих объектов:

- столбцов таблицы;
- переменных;
- входных или выходных параметров (в хранимых процедурах или определяемых пользователем функциях).

ПРИМЕЧАНИЕ

В этом разделе рассматривается использование типа данных XML для объявления столбца таблицы. Подобным образом с использованием этого типа осуществляется объявление переменных и параметров.

В примере 26.4 показано использование XML типа данных для объявления столбца таблицы.

Пример 26.4. Объявление столбца таблицы XML типа данных

```
USE sample;
CREATE TABLE xmltab (id INTEGER NOT NULL PRIMARY KEY, xml_column XML);
```

В примере 26.4 инструкция CREATE TABLE создает таблицу с двумя столбцами: id и xml_column. Столбец id используется для однозначного идентифицирования строк таблицы. Столбец xml_column является столбцом XML, который будет использоваться в последующих примерах для демонстрации сохранения, индексирования и извлечения XML-документов.

Как уже упоминалось ранее, XML-документы можно сохранять во внутреннем формате в столбце типа данных XML. В примере 26.5 показано сохранение XML-документа, используя инструкцию INSERT.

Пример 26.5. Сохранение XML-документа

```
USE sample;
INSERT INTO xmltab VALUES (1,
'<?xml version="1.0"?>
<PersonList Type="Employee">
   <Title> Value="Employee List"></Title>
   <Contents>
      <Employee>
         <Name>Ann Jones</Name>
         <No>10102</No>
         <Deptno>d3</Deptno>
         <Address>
            <City>Dallas</City>
            <Street>Main St</Street>
         </Address>
      </Employee>
      <Employee>
         <Name>John Barrimore</Name>
         <No>18316</No>
         <Deptno>d1</Deptno>
         <Address>
            <City>Seattle</City>
            <Street>Abbey Rd</Street>
      </Address>
      </Employee>
   </Contents>
</PersonList>');
```

В примере 26.5 инструкция `INSERT` вставляет два значения: значение идентификатора и XML-документ. (Вставляемый в данном примере документ это тот же самый документ, что показан в начале этой главы в примере 26.1.) Перед тем как сохранять XML-документ, выполняется его синтаксический анализ при помощи синтаксического анализатора XML, который проверяет, правильно ли сформирован данный XML-документ. Например, если бы в документе была опущена последняя строка (`</PersonList>`), синтаксический анализатор XML выдал бы следующее сообщение об ошибке:

```
Msg 9400, Level 16, State 1, Line 3
XML parsing: line 24, character 0, unexpected end of input
```

(непредвиденное завершение ввода)

При просмотре содержимого таблицы `xmltab` посредством инструкции `SELECT`, SQL Server Management Studio отображает XML-документ с помощью редактора XML. (Для отображения всего документа в редакторе щелкните соответствующее значение в результирующем наборе.)

Индексирование столбца *XML*

Database Engine сохраняет значения XML в виде большого двоичного объекта. При отсутствии индекса в таких объектах во время выполнения запроса осуществляется декомпозиция, что может занять достаточно много времени. Поэтому индексирование столбцов XML помогает повысить уровень производительности запросов.

ПРИМЕЧАНИЕ

Если вы хотите создать произвольные виды XML-индексов, то соответствующая таблица должна включать явное определение первичного ключа (см. пример 26.4).

Система поддерживает создание одного первичного XML-индекса и три типа вторичных XML-индексов. При создании первичного XML-индекса индексируются все теги, значения и пути в экземплярах XML столбца типа XML. Первичный XML-индекс используется запросами для возвращения скалярных значений или поддеревьев XML.

В примере 26.6 показано создание первичного индекса XML.

Пример 26.6. Создание первичного индекса XML

```
USE sample;
GO
CREATE PRIMARY XML INDEX i_xmlcolumn ON xmltab(xml_column);
```

Как вы можете видеть в примере 26.6, создание первичного индекса XML подобно созданию обычного индекса, которое было рассмотрено в *главе 10*. С помощью первичного XML-индекса создается соответствующая внутренняя реляционная форма XML-экземпляра. Это позволяет избежать генерирования внутренней формы при каждом выполнении запроса или обновлении.

Чтобы повысить уровень производительности поиска еще больше, можно создать вторичные XML-индексы. Обязательным условием для создания вторичных XML-индексов является наличие первичного XML-индекса. Можно создавать три типа вторичных XML-индексов, используя следующие ключевые слова:

- FOR PATH — создает вторичный XML-индекс по структуре документа;
- FOR VALUE — создает вторичный XML-индекс по значениям элементов и атрибутов столбца XML;
- FOR PROPERTY — создает вторичный XML-индекс для поиска по свойствам.

Далее представлено несколько рекомендаций по созданию вторичных XML-индексов.

- Если ваши запросы существенно используют выражения путей в столбцах XML, то, скорее всего, индекс PATH повысит их скорость. Наиболее общим способом будет использовать метод exist() для столбцов XML в предложении WHERE. (Метод exist() рассматривается *далее в этой главе*.)

- Использование индекса PROPERTY может быть полезным для запросов, которые извлекают множественные значения из отдельных экземпляров XML, используя для этого выражения пути.
- Если запросы используются для выборки значений из экземпляров XML, не имея информации об именах элементов или атрибутов, содержащих эти значения, может быть полезным создать индекс VALUE.

В примере 26.7 показано создание индекса PATH. Синтаксис для создания всех других вторичных XML-индексов аналогичен.

Пример 26.7. Создание индекса PATH

```
USE sample;
GO
CREATE XML INDEX i_xmlcolumn_path ON xmltab(xml_column)
                        USING XML INDEX i_xmlcolumn FOR PATH;
```

В примере 26.7 для создания вторичного индекса PATH используется ключевое слово FOR PATH. Предложение USING необходимо указывать для определения любого вторичного XML-индекса.

По сравнению с обычными индексами, XML-индексы имеют некоторые ограничения:

- XML-индексы не могут быть составными;
- не существует кластеризованных XML-индексов.

ПРИМЕЧАНИЕ

Причины для создания XML-индексов иные, чем причины для создания обычных индексов. XML-индексы повышают производительность запросов по XML-документам, тогда как обычные индексы повышают производительность запросов SQL.

SQL Server также поддерживает инструкции ALTER INDEX и DROP INDEX для XML-индексов. Инструкция ALTER INDEX позволяет изменять структуру имеющегося XML-индекса, а инструкция DROP INDEX применяется для удаления XML-индекса.

Для XML-индексов также существует представление каталога, называемое sys.xml_indexes. Это представление возвращает строку для каждого имеющегося XML-индекса. Наиболее важными столбцами этого представления являются столбцы using_xml_index_id и secondary_type. В первом столбце указывается степень индекса (первичный или вторичный), а во втором — тип вторичного индекса (P для PATH, V для VALUE и R для PROPERTY). Это представление также наследует столбцы представления каталога sys.indexes.

Типизированные в сравнении с нетипизированными XML

Как уже упоминалось, XML-документ может быть правильно сформированным и действительным. (Проверить действительность можно только для правильно сфор-

мированного документа.) О документе XML, который соответствует одной или нескольким схемам, говорят, что он является *действительным согласно схеме* (schema-valid) и называется *документом экземпляра* (instance document) схем. Схемы XML применяются для выполнения более точного контроля типов при компиляции запросов.

Столбцы, переменные и параметры с типом данных `XML` могут быть типизированными (typed), т. е. могут соответствовать одной или нескольким схемам, или нетипизированными (untyped). Иными словами, когда типизированный экземпляр XML присваивается столбцу, переменной или параметру с типом данных `XML`, то система проверяет экземпляр.

В следующем разделе рассматривается использование схем XML, после чего более подробно рассматривается предмет типизированных экземпляров XML.

Схемы XML и SQL Server

Схема XML задает набор типов данных, которые существуют в определенном пространстве имен. Для реализации XML-схем применяется язык XML Schema. (Язык Transact-SQL не поддерживает язык DTD для определения XML-схем.) В Database Engine для импорта компонентов схемы в базу данных применяется инструкция `CREATE XML SCHEMA COLLECTION`. Использование этой инструкции показано в примере 26.8.

Пример 26.8. Импортирование компонентов схемы в базу данных

```
USE sample;
CREATE XML SCHEMA COLLECTION EmployeeSchema AS
   N'<?xml version="1.0" encoding="UTF-16"?>
   <xsd:schema elementFormDefault="unqualifled"
      attributeFormDefault="unqualified"
      xmlns:xsd="http://www.wS.org/2001/XMLSchema" >
      <xsd:element name="employees">
         <xsd:complexType mixed="false">
         <xsd:sequence>
            <xsd:element name="fname" type="xsd:string"/>
            <xsd:element name="lname" type="xsd:string"/>
            <xsd:element name="department" type="xsd:string"/>
            <xsd:element name="salary" type="xsd:integer"/>
            <xsd:element name="comments" type="xsd:string"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>';
```

В примере 26.8 представлено использование инструкции `CREATE XML SCHEMA COLLECTION` для создания каталога схемы `EmployeeSchema` в виде объекта базы дан-

ных. Схема XML в примере 26.8 содержит атрибуты (элементы) сведений о сотрудниках, такие как фамилия, имя и оклад. Подробное обсуждение языка XML Schema выходит за рамки этой книги.

Обычно, коллекция схем XML имеет имя, которое может быть уточнено, используя имя реляционной схемы (например, `dbo.EmployeeSchema`). Коллекция схем состоит из одной или нескольких схем, которые определяют типы в одном или нескольких пространствах имен XML. Если в схеме XML опущен атрибут `targetNamespace`, то эта схема не имеет связанного с ней пространства имен. Коллекция схем XML может иметь максимум одну такую схему.

Компонент Database Engine также поддерживает инструкции `ALTER SML SCHEMA COLLECTION` и `DROP XML SCHEMA COLLECTION`. Первая инструкция позволяет добавлять новые схемы в существующую коллекцию схем XML, а вторая удаляет всю коллекцию схем.

Типизированные XML-столбцы, переменные и параметры

Для каждого типизированного XML-столбца, переменной или параметра должна быть указана соответствующая схема. Для этого имя коллекции схем, которая была создана с помощью инструкции `CREATE XML SCHEMA COLLECTION`, должно быть указано внутри пары скобок после имени экземпляра, как это показано в примере 26.9.

Пример 26.9. Задание XML-схемы для столбца `xml_person`

```
USE sample;
CREATE TABLE xml_persontab (id INTEGER, xml_person
                                                XML(EmployeeSchema));
```

В примере 26.9 столбец `xml_person` связан с коллекцией схем XML `EmployeeSchema` (см. пример 26.8). Это означает, что для проверки действительности содержимого этого столбца применяются все спецификации из определенных схем. Иными словами, при вводе нового значения в столбец типа `XML` (или изменения существующего значения) проверяются все определенные в схеме ограничения.

Спецификация коллекции схем XML для типизированного экземпляра XML может быть расширена двумя ключевыми словами:

- `DOCUMENT`;
- `CONTENT`.

Ключевое слово `DOCUMENT` указывает, что столбец `XML` может содержать только XML-документы, а ключевое слово `CONTENT`, которое является значением по умолчанию, указывает, что столбец `XML` может содержать как документы, так и фрагменты документов. Вспомним, что XML-документ должен иметь один корневой элемент, тогда как XML-фрагмент является конструкцией XML, не имеющей корневого элемента.

Использование ключевого слова `DOCUMENT` показано в примере 26.10.

Пример 26.10. Использование ключевого слова DOCUMENT

```
USE sample;
CREATE TABLE xml_persontab_doc (id INTEGER,
                     xml_person XML(DOCUMENT EmployeeSchema));
```

Компонент Database Engine поддерживает несколько представлений каталога для схем XML, наиболее важными из которых являются следующие:

- sys.xml_schema_attributes — возвращает строку для каждого компонента схемы XML, который является атрибутом;
- sys.xml_schema_elements — возвращает строку для каждого компонента схемы XML, который является элементом;
- sys.xml_schema_components — возвращает строку для каждого компонента схемы XML.

Хранение XML-документов с использованием декомпозиции

Системная процедура sp_xml_preparedocument считывает вводимый XML-текст, выполняет его синтаксический анализ и представляет проанализированный документ в виде дерева с различными узлами: элементами, атрибутами, текстом и комментариями.

Применение системной процедуры sp_xml_preparedocument показано в примере 26.11.

Пример 26.11. Использование системной процедуры sp_xml_preparedocument

```
USE sample;
DECLARE @hdoc INT
DECLARE @doc VARCHAR(1000)
SET @doc ='<ROOT>
   <Employee>
      <Name>Ann Jones</Name>
      <No>10102</No>
      <Deptno>d3</Deptno>
      <Address>Dallas</Address>
   </Employee>
   <Employee>
      <Name>John Barrimore</Name>
      <No>18316</No>
      <Deptno>d1</Deptno>
      <Address>Seattle</Address>
   </Employee>
</ROOT>'
EXEC sp_xml_preparedocument @hdoc OUTPUT, @doc
```

В примере 26.11 XML-документ сохраняется в виде строки в переменной `@doc`. Эта строка дробится системной процедурой `sp_xml_preparedocument`. Процедура возвращает дескриптор (`@hdoc`), который затем может быть использован для доступа к вновь созданному представлению XML-документа.

ПРИМЕЧАНИЕ

Дескриптор `@hdoc` будет использован в первом примере следующего раздела (пример 26.12) для извлечения данных из XML-документа.

Системная процедура `sp_xml_removedocument` удаляет внутреннее представление XML-документа, указанное дескриптором документа, и делает этот дескриптор недействительным.

Представление XML-данных, сохраненных в реляционной форме, рассматривается в следующем разделе.

Представление данных

Компонент Database Engine позволяет представлять данные следующими способами:

- представлять XML-документы и фрагменты как реляционные данные;
- представлять реляционные данные как XML-документы.

Эти два способа представления данных рассматриваются в следующих разделах.

Представление XML-документов в качестве реляционных данных

Посредством языка Transact-SQL из XML-документов или фрагментов можно сгенерировать набор строк, по которым можно выполнять запросы данных. Для этого применяется набор средств OpenXML, позволяющий использовать инструкции языка Transact-SQL для извлечения данных из XML-документов, как будто бы эти данные находились в реляционной таблице. Интернациональный стандарт OpenXML для документов поддается реализации многими разными приложениями на многих различных платформах.

В примере 26.12 показано, как вы можете запросить XML-документ из примера 26.11 посредством использования OpenXML.

ПРИМЕЧАНИЕ

Код в примере 26.12 должен быть добавлен к коду примера 26.11 и затем выполнен этот объединенный код.

Пример 26.12. Выборка данных из XML-документа посредством OpenXML

```
SELECT * FROM OPENXML (Shdoc, '/ROOT/Employee', 1)
   WITH (name VARCHAR(20) 'Name',
      no INT 'No',
      deptno VARCHAR(6) 'Deptno',
      address VARCHAR(50) 'Address');
```

Результат выполнения этого запроса:

Name	No	deptno	address
Ann Jones	10102	d3	Dallas
John Barrimore	18316	d1	Seattle

Представление реляционных данных в качестве XML-документов

Как вы уже знаете из *главы 6*, инструкция SELECT извлекает и данные из одной или нескольких таблиц и отображает соответствующий результирующий набор. По умолчанию этот результирующий набор инструкции SELECT отображается в виде таблицы. Применив в инструкции SELECT предложение FOR XML, результирующий набор запроса можно отобразить в виде XML-документа или фрагмента. В этом предложении можно задать один из следующих четырех режимов:

- RAW;
- AUTO;
- EXPLICIT;
- PATH.

ПРИМЕЧАНИЕ

Предложение FOR XML указывается в конце инструкции SELECT.

Эти режимы рассматриваются в последующих далее разделах. Кроме этого, в последнем разделе обсуждаются директивы.

Режим *RAW*

Опция FOR XML RAW преобразовывает каждую строку результирующего набора в XML-элемент с идентификатором <row>. Каждое значение столбца преобразовывается в атрибут XML-элемента, где имя атрибута такое же, как и имя соответствующего столбца. Это справедливо только для столбцов, значения которых не равны NULL.

В примере 26.13 показано использование опции FOR XML RAW, заданной для соединения таблиц employee и works_on базы данных sample.

Пример 26.13. Использование опции FOR XML RAW

```
USE sample;
SELECT employee.emp_no, emp_lname, works_on.job
FROM employee, works_on
WHERE employee.emp_no <= 10000
AND employee.emp_no = works_on.emp_no
FOR XML RAW;
```

В результате выполнения запроса, приведенного в примере 26.13, отображается следующий фрагмент XML-документа:

```
<row emp_no="2581" emp_lname="Hansel " job="Analyst        " />
<row emp_no="9031" emp_lname="Bertoni " job="Manager       " />
<row emp_no="9031" emp_lname="Bertoni " job="Clerk         " />
```

Если бы в запросе примера 26.13 отсутствовала бы опция FOR XML RAW, то результат его выполнения был бы следующим:

emp_no	emp_lname	job
2581	Hansel	Analyst
9031	Bertoni	Manager
9031	Bertoni	Clerk

Как можно видеть по обоим результирующим наборам, первый запрос возвращает один XML-элемент для каждой выбранной строки.

Режим *AUTO*

Режим AUTO возвращает результирующий набор запроса в виде простого вложенного XML-дерева. Каждая таблица в предложении FROM, из которой в списке выборки столбцов указан по крайней мере один столбец, представляется в виде XML-элемента. Столбцы в списке выборки отображаются в соответствующие атрибуты элементов.

Использование режима AUTO показано в примере 26.14.

Пример 26.14. Запрос для отображения XML-документа в режиме AUTO

```
USE sample;
SELECT employee.emp_no, emp_lname, works_on.job
FROM employee, works_on
WHERE employee.emp_no <= 10000
AND employee,emp_no = works_on.emp_no
FOR XML AUTO;
```

Этот запрос возвращает следующий результат:

```
<employee emp_no="9031"   emp_lname="Bertoni       ">
   <works_on job="Manager      "    />
   <works_on job="Clerk         "   />
</employee>
<employee emp_no="2581"   emp_lname="Hansel        ">
   <works_on job="Analyst      "    />
</employee>
```

ПРИМЕЧАНИЕ

Результат выполнения запроса, показанного в примере 26.14, не является действительным XML-документом. Как упоминалось ранее, действительный XML-документ должен иметь корневой элемент.

Результат выполнения запроса примера 16.14 значительно отличается от результата предыдущего примера, хотя инструкция SELECT в обоих запросах одинаковая (за исключением указания в ней режима AUTO, вместо режима RAW). Как можно видеть из примера 26.14, результирующий набор этого запроса отображается в виде иерархии таблиц employee и works_on. Эта иерархия основана на связи первичного и внешнего ключей обеих таблиц. По этой причине сначала отображаются данные из таблицы employee, а соответствующие данные из таблицы works_on отображаются после них, на низшем уровне иерархии.

Уровень вложения элементов результирующего XML-документа зависит от порядка таблиц, определяемых столбцами, указанными в списке выборки инструкции SELECT. Поэтому порядок указания столбцов в списке выборки имеет значение. По этой причине, в примере 26.14 значения столбца emp_no таблицы employee формируют верхний элемент результирующего XML-фрагмента, а значения столбца job таблицы works_on формируют субэлемент, вложенный в верхний элемент.

Режим *EXPLICIT*

Как можно видеть из примера 26.14, результирующий набор в режиме AUTO отображается в виде простого вложенного XML-дерева. Использование режима AUTO в запросах полезно только в тех случаях, когда нужно создать простую иерархию, поскольку этот режим не предоставляет достаточного уровня управления над формой XML-документа, создаваемого по результатам запроса.

Если нужно задать расширенную форму результирующего набора, то можно использовать опцию FOR XML EXPLICIT. При использовании этой опции результирующий набор отображается в виде универсальной таблицы, содержащей всю информацию о результирующем XML-дереве. Данные в этой таблице секционируются по вертикальным группам, каждая из которых становится XML-элементом в результирующем наборе.

Использование режима EXPLICIT показано в примере 26.15.

Пример 26.15. Запрос для отображения XML-документа в режиме EXPLICIT

```
USE sample;
SELECT 1 AS tag, NULL as parent,
emp_lname AS [employee!1!emp_lname],
NULL AS [works_on!2!job]
FROM employee
UNION
SELECT 2, 1, emp_lname, works_on.job
FROM employee, works_on
WHERE employee.emp_no <= 10000
AND employee.emp_no = works_on.emp_no
ORDER BY [employee!1!emp_lname]
FOR XML EXPLICIT;
```

Результат выполнения этого запроса:

```
<employee emp_lname="Barrimore    " />
<employee emp_lname="Bertoni      ">
   <works_on job="Clerk      " />
   <works_on job="Manager    " />
</employee>
<employee emp_lname="Hansel  ">
   <works_on job="Analyst       " />
</employee>
<employee emp_lname="James   "      />
<employee emp_lname="Jones   "      />
<employee emp_lname="Moser   "      />
<employee emp_lname="Smith   "      />
```

Как можно видеть в инструкции SELECT примера 26.15, опция FOR XML EXPLICIT требует два дополнительных столбца метаданных: tag и parent. Эти два столбца используются для определения отношения "первичный ключ/внешний ключ" в XML-дереве. В столбце tag сохраняется номер тега текущего элемента, а в столбце parent — номер тега родительского элемента. (Родительской является таблица с первичным ключом.) Если родительским тегом является NULL, эта строка помещается сразу же под корневым элементом.

ПРИМЕЧАНИЕ

Режим EXPLICIT не следует использовать по причине его сложности. Вместо него следует использовать режим PATH, который рассматривается следующим.

Режим *PATH*

Все три рассмотренные ранее опции FOR XML обладают разными недостатками и ограничениями. Опция FOR XML RAW поддерживает только один уровень вложенности, а опция FOR XML AUTO требует, чтобы все выбираемые из одной таблицы столб-

цы были на одном уровне. Кроме этого, обе эти опции не позволяют смешивать элементы и атрибуты в одном и том же XML-документе. С другой стороны, опция FOR XML EXPLICIT позволяет смешивать элементы и атрибуты, но, как можно видеть из предыдущего примера, имеет сложный синтаксис.

Опция FOR XML PATH позволяет с большой легкостью реализовать почти все запросы, для которых требуется режим EXPLICIT. В режиме PATH имена или псевдонимы столбцов рассматриваются как расширения XPath, которые указывают, какие значения преобразовываются в XML. (Выражение XPath состоит из последовательности узлов, возможно, разделенных знаком наклонной черты /. Для каждой наклонной черты в результирующем документе система создает уровень иерархии.)

Использование режима PATH показано в примере 26.16.

Пример 26.16. Запрос для отображения XML-документа в режиме PATH

```
USE sample;
SELECT d.dept_name "SDepartment",
   emp_ f name "EmpName/First",
   emp_lname "EmpName/Last"
   FROM Employee e, department d
   WHERE e.dept_no = d.dept_no
   AND d.dept_no = 'd1'
   FOR XML PATH;
```

Результат выполнения этого запроса:

```
<row Department="Research                    ">
   <EmpName>
      <First>John                 </First>
      <Last>Barrimore            </Last>
   </EmpName> </row>
<row Department="Research                    ">
   <EmpName>
      <First>Sybill               </First>
      <Last>Moser                </Last>
   </EmpName>
</row>
```

В режиме PATH все имена столбцов используются в качестве пути в создании XML-документа. Столбец, содержащий имена отделов, начинается с символа @. Это означает, что атрибут Department добавляется в элемент <row>. Все другие столбцы содержат наклонную черту в имени столбца, что означает иерархию. По этой причине, в результирующем XML-документе дочерний элемент <EmpName> будет находиться под элементом <row>, а элементы <First> и <Last> — на следующем подуровне.

Директивы

Компонент Database Engine поддерживает несколько разных директив, которые позволяют получать разные результаты при представлении XML-документов и фрагментов. Некоторые из этих директив приведены в последующем списке:

- TYPE;
- ELEMENTS (с XSINIL);
- ROOT.

Эти директивы рассмотрены далее.

Директива *TYPE*

Компонент Database Engine позволяет сохранять результат реляционного запроса как XML-документ или фрагмент типа данных XML, используя для этого директиву TYPE. Когда указывается директива TYPE, запрос, содержащий опцию FOR XML, возвращает результирующий набор, состоящий из одной строки и одного столбца. (Это директива является общей директивой, т. е. ее можно использовать во всех четырех режимах.) В примере 26.17 показано использование директивы TYPE с режимом AUTO.

Пример 26.17. Использование директивы TYPE с режимом AUTO

```
USE sample;
DECLARE @x xml;
SET @x = (SELECT * FROM department
                   FOR XML AUTO, TYPE);
SELECT @x;
```

Этот запрос возвращает следующий результат:

```
<department dept_no="d1" dept_name="Research   " location="Dallas " />
<department dept_no="d2" dept_name="Accounting " location="Seattle "/>
<department dept_no="d3" dept_name="Marketing  " location="Dallas " />
```

В примере 26.17 сначала объявляется локальная переменная @x типа данных XML, которой присваивается результат выполнения инструкции SELECT. Последняя инструкция SELECT в пакете отображает содержимое этой переменной.

Директива *ELEMENTS*

Как вы уже знаете из *главы 3*, компонент Database Engine использует значения NULL для обозначения неизвестных (или отсутствующих значений). В отличие от реляционной модели, XML не поддерживает значения NULL, и эти значения отсутствуют в результирующих наборах запросов с параметром FOR XML.

Компонент Database Engine позволяет отображать отсутствующие значения в XML-документе, используя директиву ELEMENTS с параметром XSINIL. Обычно директива ELEMENTS конструирует соответствующий XML-документ таким образом, чтобы

значение каждого столбца отображается в элемент. По умолчанию, если значение столбца равно NULL, элемент не добавляется. Указывая с этой директивой дополнительную опцию XSINIL, можно задать создание элемента также и для значений NULL. В таком случае для каждого значения NULL столбца возвращается элемент, чей атрибут SXINIL имеет значение TRUE.

Директива *ROOT*

Обычно запросы с опцией FOR XML создают XML-фрагменты, т. е. XML-документы без корневого элемента. Это может быть проблемой, если интерфейс API принимает в качестве ввода только XML-документы. Компонент Database Engine позволяет добавить в XML-фрагмент корневой элемент с использованием директивы ROOT. Указывая в запросе FOR XML директиву ROOT, можно задать добавление к результирующему набору XML одного элемента верхнего уровня. (Имя корневого элемента указывается в аргументе директивы.)

Использование директивы ROOT показано в примере 26.18.

Пример 26.18. Запрос с использованием директивы ROOT

```
USE sample;
SELECT * FROM department
FOR XML AUTO, ROOT ('AllDepartments');
```

Результат выполнения этого запроса:

```
<AllDepartments>
   <department dept_no="d1 "dept_name="Research  " location="Dallas " />
   <department dept_no="d2 "dept_name="Accounting" location="Seattle "/>
   <department dept_no="d3 "dept_name="Marketing " location="Dallas " />
</AllDepartments>
```

Запрос в примере 26.18 возвращает XML-фрагмент, состоящий из всех строк таблицы department. Директива ROOT запроса при помощи параметра AllDepartments добавляет в результирующий набор корневую спецификацию в качестве имени корня.

Запрашивание данных из XML-документов

Для запроса данных из XML-документов можно использовать два стандартных языка:

- язык запросов XPath;
- язык запросов XQuery.

Простой язык запросов *XPath* применяется для доступа к элементам и атрибутам XML-документа. Язык *XQuery* — это сложный язык запросов, частью которого является язык XPath. Язык запросов XQuery поддерживает так называемые *FLWOR*-выражения (произносится как "flower"), т. е. предложения FOR, LET, WHERE, ORDER BY и

RETURN. (Подробное рассмотрение этих языков запросов выходит за пределы тематики этой книги. В этом разделе дается только краткое ознакомление с ними.)

Компонент Database Engine, используя язык запросов XQuery, предоставляет пять следующих методов для запроса данных из XML-документов:

- метод query() — принимает в качестве ввода инструкцию XQuery и возвращает экземпляр типа данных XML;
- метод exist() — принимает в качестве ввода инструкцию XQuery и возвращает значение 0, 1 или NULL, в зависимости от результата выполнения запроса;
- метод value() — принимает в качестве ввода инструкцию XQuery и возвращает одно скалярное значение;
- метод nodes() — этот метод применяется для разложения экземпляра типа данных XML на реляционные данные. Метод позволяет задавать узлы для отображения в новых строках;
- метод modify() — применяется для вставки, изменения и удаления XML-документов.

В примере 26.19 показано использование метода query() в запросе.

Пример 26.19. Запрос с использованием метода query()

```
USE sample;
SELECT xml_column.query('/PersonList/Title')
   FROM xmltab
   FOR XML AUTO, TYPE;
```

Результат выполнения этого запроса:

```
<xmltab>
   <Title> Value="Employee List"&gt;</Title>
</xmltab>
```

Список выборки столбцов инструкции SELECT в запросе примера 26.19 содержит метод query(), который применяется к экземпляру столбца xml_column. Параметром этого метода является выражение на языке запросов XPath. (Обратите внимание, что методы применяются с использованием разделителя "точка".) Абсолютное выражение '/PersonList/Title' извлекает значения элемента Title. (Вспомним, что выражение на языке XPath являются действительными инструкциями языка XQuery, поскольку язык XPath является подмножеством языка XQuery.)

В примере 26.20 показано использование метода exist() в запросе.

Пример 26.20. Запрос с использованием метода exist()

```
SELECT xml_column.exist('/PersonList/Title/@Value=EmployeeList') AS a
FROM xmltab
FOR XML AUTO, TYPE;
```

Результат выполнения этого запроса:

```
<xmltab a="1" />
```

Как уже упоминалось ранее, метод `exist()` принимает в качестве ввода инструкцию языка XQuery и возвращает значение `0`, `1` или `NULL`, в зависимости от результата выполнения запроса. Если запрос возвращает пустую последовательность, то метод возвращает значение `0`. Для последовательности, содержащей, по крайней мере, один элемент, возвращается значение `1`, а если значение столбца `NULL`, то также возвращается `NULL`.

Список выборки столбцов инструкции `SELECT` в примере 26.20 содержит метод `exist()` с выражением `XPath`, которое проверяет, равно ли значение атрибута `Value` XML-документа выражению `EmployeeList`. (В языке XPath символ `@` применяется для обозначения атрибута XML.) Результат проверки является положительным, поэтому запрос возвращает значение `1`.

Резюме

Формат XML применяется для обмена и архивации данных. Документ XML содержит несколько тегов, которые определяются составителем этого документа. Все части XML-документа, входящие в его логическую структуру, называются элементами. Элементы вместе со своими субэлементами создают иерархическую структуру. Любой элемент может содержать дополнительную информацию, которая называется атрибутом. Кроме элементов и атрибутов, XML-документ может также содержать пространства имен, инструкции по обработке и комментарии.

Стандартизованный язык схем XML Schema применяется для определения схем XML-документов. Документ XML, который отвечает требованиям соответствующей схемы, является действительным по схеме документом. (Также существует другой простой язык схем DTD, который не стандартизован.) Язык XML Schema содержит инструкции описания данных для XML, во многом подобно тому, как язык DDL содержит инструкции описания данных для SQL.

Компонент Database Engine предоставляет полную поддержку для хранения и представления XML-документов и запроса данных из них. Самой важной особенностью этой поддержки является наличие типа данных `XML`, который позволяет системе баз данных сохранять XML-документы в виде объектов первого класса.

Значения типа данных `XML` могут быть действительными согласно схеме, если с этим типом связана одна или несколько схем. Определить точные типы элементов и атрибутов можно только в том случае, если соответствующий XML-документ содержит типы, заданные XML-схемами. Определения схем задаются с помощью инструкции `CREATE XML SCHEMA COLLECTION`.

Язык XML также поддерживает несколько методов, которые можно использовать для запроса данных из XML-документов. Эти методы принимают в качестве параметров выражения языков запросов XPath и XQuery.

В следующей главе мы рассмотрим пространственные данные.

Глава 27

Пространственные данные

- **Введение**
- **Работа с данными пространственного типа**
- **Отображение информации о пространственных данных**
- **Новые возможности SQL Server 2012 для работы с пространственными данными**

Эта глава состоит из четырех частей. Во вводной части описываются наиболее важные общие понятия пространственных данных, которые необходимо понимать для того, чтобы работать с этими данными. Кроме различных пространственных моделей и форматов, в этой части представляются оба типа пространственных данных, поддерживаемых SQL Server: `GEOMETRY` и `GEOGRAPHY`, а также подробно описываются несколько подтипов этих корневых типов данных.

Во второй части главы приводится несколько примеров, демонстрирующих способы применения пространственных данных. В этих примерах рассматривается применение различных методов для типов данных `GEOMETRY` и `GEOGRAPHY`, а также описывается создание и использование пространственного индекса.

Третья часть главы отводится рассмотрению использования среды SQL Server Management Studio для отображения или вывода пространственных данных в графическом формате. Предоставляются примеры для отображения обоих типов пространственных данных.

В последней части главы обсуждаются новые возможности, введенные в SQL Server 2012. Рассматриваются новые подтипы типов данных `GEOMETRY` и `GEOGRAPHY`, а также новые пространственные индексы и хранимые системные процедуры.

Введение

В течение последних нескольких лет существенно возросла потребность бизнеса во включении пространственных данных в собираемую информацию и управление

этими данными посредством систем управления базами данных. Самым основным способствующим фактором роста этой необходимости является быстрое распространение географических сервисов и устройств, таких как Virtual Earth корпорации Microsoft и недорогих *GPS*-устройств (Global Positioning System — глобальная система навигации и определения местоположения).

Обычно, поддержка пространственных данных поставщиком базы данных помогает пользователям принимать более качественные решения во многих сценариях, таких как:

- *анализ в области недвижимости* (например, "найти подходящий дом на расстоянии 500 метров от общеобразовательной школы");
- *информация для потребителей* (например, "найти торгово-развлекательные центры, ближайшие к данному почтовому индексу");
- *анализ рынка* (например, "определить географические регионы продаж и выяснить, существует ли надобность в новом филиале").

Как вы уже знаете из *главы 5*, посредством инструкции `CREATE TYPE` пользователи могут определять свои типы данных. Эти типы данных реализуются с помощью общеязыковой среды выполнения (CLR), которая рассмотрена в *главе 8*. Разработчики SQL Server также использовали эту среду для реализации двух новых, пространственных типов данных: `GEOMETRY` и `GEOGRAPHY`. Эти два типа данных обсуждаются после краткого рассмотрения различных моделей представления пространственных данных.

Модели для представления пространственных данных

Как правило, для представления пространственных данных применяются следующие две группы моделей:

- геодезические пространственные модели;
- плоские пространственные модели.

Планеты являются сложными объектами, которые можно представить посредством сплющенной сферы (называемой *сфероидом*). Глобус является хорошим приближенным представлением планеты Земля (да и других планет тоже), где местоположения на поверхности описываются, используя широту (параллели) и долготу (меридианы). (Широта определяет местоположение на поверхности Земли к северу или югу от экватора, а долгота определяет местоположение справа или слева от выбранного меридиана.) Модели, в которых применяются эти измерения, называются *геодезическими моделями*. Так как эти модели предоставляют хорошее приближение сфероидов, они обеспечивают наиболее точный способ представления пространственных данных.

В плоских (или планарных) пространственных моделях для представления Земли используются двумерные карты. В этом случае сфероид сплющивается и проецируется на плоскости. Этот процесс преобразования в плоскость вносит определенные

искажения в форму и размеры проецируемых (географических) объектов. Плоские пространственные модели лучше всего подходят для небольших участков поверхности, поскольку чем больше представляемый таким способом участок поверхности, тем больше вносится искажений.

Как будет продемонстрировано в следующих двух разделах, тип данных `GEOMETRY` основан на плоской пространственной модели, тогда как тип данных `GEOGRAPHY` — на геодезической.

Тип данных *GEOMETRY*

Термин *геометрический объект* (geometry) был введен консорциумом *OGC* (Open Geospatial Consortium — открытый ГИС-консорциум) для представления пространственных свойств, таких как точки и линии. Поэтому геометрический объект представляет данные в двумерном пространстве в виде точек, прямых и многоугольников, используя одну из существующих плоских пространственных моделей.

Геометрический объект можно представить себе, как тип данных с несколькими подтипами, как это показано на рис. 27.1.

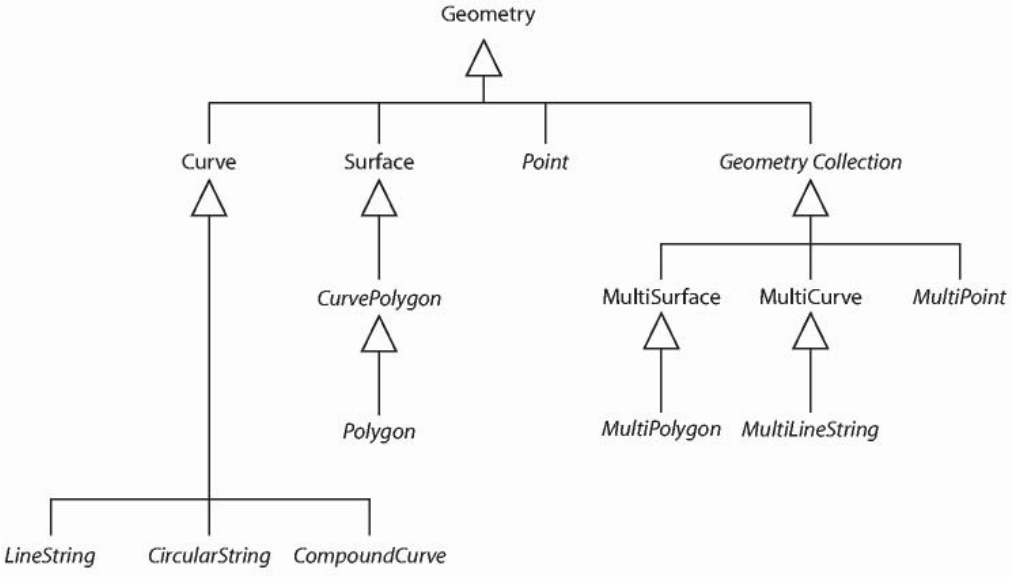


Рис. 27.1. Иерархия типа с корневым типом `GEOMETRY`

Подклассы разделяются на две категории: базовые геометрические подклассы и подклассы однородных коллекций. Базовые геометрические подклассы включают, среди прочих, подклассы `Point`, `LineString` и `Polygon`, а однородные коллекции включают подклассы `MultiPoint`, `MultiLineString` и `MultiPolygon`. Как можно судить по их именам, однородные коллекции являются коллекциями базовых геометрических подклассов. В дополнение к разделяемым свойствам базовых геометрических подклассов, однородные коллекции имеют свои собственные свойства.

Типы, отображенные на рис. 27.1 курсивом, являются экземплируемыми (instantiable), т. е. они имеют экземпляры класса (instances). В SQL Server все экземплируемые типы реализуются, как определяемые пользователем типы. Далее приводится список и краткое описание экземплируемых типов.

- **`Point`**. Экземпляр типа `Point` (точка) является двумерным геометрическим объектом, имеющим одинаковые значения координат X и Y. Таким образом, он имеет нулевую (`NULL`) границу. Факультативно, экземпляр типа `Point` может иметь две дополнительные координаты: уровень (координата Z) и меру (координата M). Экземпляры типа `Point` обычно используются для построения сложных пространственных типов.
- **`MultiPoint`**. Экземпляр типа `Multipoint` (многоточечный) является коллекцией из нулевого или большего количества точек. Точки в многоточечном типе не обязательно должны быть разными.
- **`LineString`**. Экземпляр типа `LineString` (ломаная линия) представляет собой геометрический объект, который имеет длину и сохраняется в виде последовательности точек, определяющих линейный путь. Следовательно, ломаная линия определяется набором точек, которые задают ее контрольные точки. Результирующая ломаная линия определяется линейной интерполяцией между контрольными точками. Ломаная линия называется *простой* (simple), если она не пересекает сама себя. Ломаная линия, у которой начальная и конечная точки совпадают, называется *замкнутой* (closed). Замкнутая простая ломаная линия называется *кольцом* (ring).
- **`MultiLineString`**. Экземпляр типа `MultiLineString` (набор ломаных линий) представляет собой коллекцию с нулевым или большим количеством ломаных линий.
- **`Polygon`**. Экземпляр типа `Polygon` (многоугольник) представляет собой двумерную геометрическую фигуру с поверхностью. Многоугольник хранится в виде последовательности точек, определяющих его внешнее ограничивающее кольцо и внутренние кольца, число которых может быть от нуля и больше. Внешнее и любые внутренние кольца задают границы многоугольника, а пространство между кольцами задает его внутреннюю часть.
- **`MultiPolygon`**. Экземпляр типа `MultiPolygon` представляет собой коллекцию многоугольников, числом от нуля и более.
- **`GeometryCollection`**. Коллекция геометрических экземпляров `GeometryCollection` содержит от нуля и более геометрических объектов. Иными словами, экземпляр этого типа может содержать экземпляры любого подтипа типа данных `GEOMETRY`.

ПРИМЕЧАНИЕ

Как можно видеть на рис. 27.1, существует три других подтипа, для которых можно создавать экземпляры: `CircularString`, `CompoundCurve` и `CurvePolygon`. Эти три подтипа подробно рассматриваются в *разд. "Новые возможности SQL Server 2012 для работы с пространственными данными" далее в этой главе*.

Тип данных *GEOGRAPHY*

Тогда как тип данных GEOMETRY использует для хранения данных координаты X и Y, тип данных GEOGRAPHY применяет для этой цели координаты широты и долготы системы GPS. (Долгота представляет горизонтальный угол в диапазоне от –180° до +180°, а широта — вертикальный угол в диапазоне от –90° до +90°.)

В отличие от типа данных GEOMETRY, для типа данных GEOGRAPHY требуется указать конкретную пространственную систему координат посредством соответствующего целочисленного идентификатора *SRID* (Spatial Reference ID — идентификатор системы пространственных координат). Список идентификаторов SRID и соответствующие им пространственные системы координат, поддерживаемые в SQL Server, можно узнать с помощью представления каталога `sys.spatial_reference_systems`. (Это представление каталога рассматривается *далее в этой главе*.)

ПРИМЕЧАНИЕ

Все экземплируемые типы (см. рис. 27.1) для типа данных GEOMETRY являются таковыми и для типа данных GEOGRAPHY.

Различия между типами данных *GEOMETRY* и *GEOGRAPHY*

Как уже упоминалось ранее, тип данных GEOMETRY применяется в плоских пространственных моделях, а тип данных GEOGRAPHY используется в геодезических моделях. Основная разница между этими двумя группами моделей состоит в том, что для типа данных GEOMETRY расстояния и площади указываются в тех же самых единицах измерения, которые используются для указания координат экземпляров. (Таким образом, расстояние между точками (0,0) и (3,4) всегда будет 5 единиц.)

Ситуация иная с типом данных GEOGRAPHY, для которого применяются эллипсоидальные координаты, которые выражаются в углах широты и долготы.

Кроме этого, на тип данных GEOGRAPHY накладываются определенные ограничения. Например, каждый экземпляр типа данных GEOGRAPHY должен вмещаться внутри одной полусферы.

Внешние форматы данных

Сервер SQL Server поддерживает три внешних формата данных, которые можно использовать для представления данных в независимой от реализации форме:

- *WKT* (Well-Known Text — известный текстовый формат) — применяется для представления систем пространственных координат пространственных структур и преобразований между системами пространственных координат;
- *WKB* (Well-Known Binary — известный двоичный формат) — является двоичным эквивалентом формата WKT;

- *GML* (Geography Markup Language — язык географической разметки) является грамматикой языка разметки XML, определенной консорциумом OGC для выражения географических свойств. Это открытый формат обмена данными для выполнения географических транзакций в Интернете.

Эти три внешних формата данных также являются частью стандарта SQL/MM, как это рассматривается в *разд. "Новые подтипы дуг окружностей" далее в этой главе*.

ПРИМЕЧАНИЕ

Все примеры в этой главе ссылаются на формат WKT, поскольку это самый легкий для чтения формат.

В следующих примерах демонстрируется синтаксис WKT для некоторых избранных типов:

- `POINT(3,4)` — значения 3 и 4 указывают координаты X и Y соответственно;
- `LINESTRING(0 0, 3 4)` — первые два значения представляют координаты X и Y начальной точки, а последние два значения — координаты X и Y конечной точки линии;
- `POLYGON(300 0, 150 0, 150 150, 300 150, 300 0)` — каждая пара чисел представляет точку на границе многоугольника. Конечная точка задаваемого многоугольника такая же, как и начальная.

Работа с данными пространственного типа

Как уже упоминалось, SQL Server поддерживает два разных типа данных в плане пространственных данных: `GEOMETRY` и `GEOGRAPHY`. Это определяемые пользователем типы данных, которые реализуются разработчиками приложений SQL Server, используя среду CLR. Каждый из этих типов данных имеет несколько подтипов, которые могут быть экземплируемыми или неэкземплируемыми. Для экземплируемых подтипов можно создавать экземпляры и работать с ними. Эти экземпляры можно сохранять как значения столбцов таблицы, а также в качестве значений переменных или параметров. Соответственно, для неэкземплируемых подтипов создавать экземпляры нельзя. В иерархии классов корневой класс обычно является неэкземплируемым, тогда как классы листьев дерева иерархии почти всегда являются экземплируемыми. (Корневой неэкземплируемый класс называется *абстрактным классом*.) В следующих двух разделах описывается, как можно использовать эти два типа данных для создания пространственных данных и выполнения запросов по ним. После этого будут представлены пространственные индексы.

Работа с типом данных *GEOMETRY*

Использование типа данных GEOMETRY продемонстрируем на примере 27.1, в котором создается таблица рынков безалкогольных напитков для определенного города (или области).

Пример 27.1. Использование типа GEOMETRY для создания таблицы рынков

```
USE sample;
CREATE TABLE beverage_markets
   (id INTEGER IDENTITY(1,1),
   name VARCHAR(25),
   shape GEOMETRY);
   INSERT INTO beverage_markets
      VALUES ('Coke1, GEOMETRY::STGeomFromText
      ('POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))', 0));
   INSERT INTO beverage_markets
      VALUES ('Pepsi', GEOMETRY::STGeomFromText
      ('POLYGON ((300 0, 150 0, 150 150, 300 150, 300 0))', 0));
   INSERT INTO beverage_markets
      VALUES ('7UP', GEOMETRY::STGeomFromText
      ('POLYGON ((300 0, 150 0, 150 150, 300 150, 300 0))', 0));
   INSERT INTO beverage_markets
      VALUES ('Almdudler', GEOMETRY::STGeomFromText
      ('POINT (50 0)', 0));
```

Создаваемая в примере 27.1 таблица содержит три столбца. Значения первого столбца (id) генерируются системой, поскольку в определении этого столбца указано свойство IDENTITY. Второй столбец (name) используется для хранения имени напитков. А третий столбец (shape) содержит информацию о форме области рынка, в котором покупатели наибольшее предпочтение отдают данному напитку. Первые три инструкции INSERT создают три области, в которых определенный напиток является наиболее предпочитаемым. Все эти три области являются многоугольниками. Четвертая инструкция INSERT вставляет точку, потому что специфический напиток (Almdudler) можно купить только в одном месте.

ПРИМЕЧАНИЕ

В примере 27.1 в определении типа данных POINT (и POLYGON также) можно видеть дополнительный, последний, параметр. В этом параметре указывается идентификатор SRID. Это обязательный параметр, значение которого по умолчанию для типа данных GEOMETRY равно 0.

В примере 27.1 вводится первый метод для работы с типом данных GEOMETRY: STGeomFromText(). Этот статический метод применяется для вставки координат геометрических фигур, таких как многоугольники и точки. Иными словами, он возвращает экземпляр типа данных GEOMETRY в формате WKT.

ПРИМЕЧАНИЕ

Обычно тип может иметь две разные группы методов: статические методы и методы экземпляров класса. Статические методы всегда применяются со всем типом (т. е. классом), тогда как методы экземпляра класса применяются с определенными

экземплярами класса. Методы каждой группы вызываются по-разному. Для вызова статических методов между именем типа и именем метода используется знак "::" (например, GEOMETRY::STGeomFromText; см. пример 27.1), тогда как для вызова методов экземпляров применяется разделительная точка (например, @g.STContains; см. пример 27.2).

Кроме метода STGeomFromText(), SQL Server поддерживает три других подобных статических метода:

- STPointFromText() — возвращает представление WKT экземпляра типа данных POINT;
- STLineFromText() — возвращает представление WKT экземпляра типа данных LINESTRING, дополненного значениями высоты и меры;
- STPolyFromText() — возвращает представление WKT экземпляра типа данных MULTIPOLYGON, дополненного значениями высоты и меры.

Запросы по пространственным данным выполняются так же, как и по реляционным данным. В следующем примере показана выборка информации из содержимого столбца shape таблицы beverage_markets. В частности, запрос в примере 27.2 определяет, находится ли магазин, продающий напиток Almdudler, в области, в которой предпочитаемым напитком является Coke.

ПРИМЕЧАНИЕ

Сервер SQL Server поддерживает большое количество методов, которые можно применять с экземплярами типа данных GEOMETRY. В последующих примерах приведены только некоторые наиболее важные из этих методов. Дополнительную информацию по другим методам экземпляров см. в *электронной документации*.

Пример 27.2. Определение нахождения объекта в пределах определенной области методом STContains()

```
DECLARE @g geometry;
DECLARE @h geometry;
SELECT @h = shape FROM beverage_markets WHERE name ='Almdudler';
SELECT @g = shape FROM beverage_markets WHERE name = 'Coke';
SELECT @g.STContains(@h);
```

Этот запрос возвращает результат 0, что означает отсутствие указанного магазина в указанной области.

Метод STContains() возвращает 1, если один экземпляр типа данных GEOMETRY содержит другой экземпляр этого же типа, который указывается в параметре метода. Результат выполнения запроса, приведенного в примере 27.2, означает, что в области, где предпочитаемым напитком является напиток Coke, нет ни одного магазина, продающего напиток Almdudler.

В запросе примера 27.3 показано использование метода `STLength()`.

Пример 27.3. Определить длину и представление WKT столбца `shape` для магазина, продающего напиток `Almdudler`

```
SELECT id, shape.ToString() AS wkt, shape.STLength() AS length
   FROM beverage_markets
   WHERE name = 'Almdudler';
```

Результат выполнения этого запроса:

Id	wkt	Length
4	POINT (500)	0

В примере 27.3 метод `STLength()` возвращает общую длину элементов типа данных `GEOMETRY`. (Возвращаемый результат равен `0`, поскольку значение является точкой.) Метод `ToString()` возвращает строку, содержащую логическое представление текущего экземпляра. Как можно видеть в результате выполнения запроса примера 27.3, этот метод используется для загрузки всех свойств данной точки и отображения их в формате WKT.

В запросе примера 27.4 показано использование метода `STIntersects()`.

Пример 27.4. Определение, пересекается ли область, в которой продается `Coke`, с областью, в которой продается `Pepsi`

```
USE sample;
DECLARE @g geometry;
DECLARE @h geometry;
SELECT @h = shape FROM beverage_markets WHERE name = 'Coke';
SELECT @g = shape FROM beverage_markets WHERE name = 'Pepsi';
SELECT @g.STIntersects(@h);
```

В результате выполнения запроса примера 27.4 возвращается значение `1` (`TRUE`), что означает, что эти две геометрические фигуры, представляющие области продаж, пересекаются.

В отличие от примера 27.3, в котором тип данных `GEOMETRY` объявляется для столбца таблицы, в примере 27.4 с этим типом данных объявляются переменные `@g` и `@h`. (Как уже упоминалось, с типом данных `GEOMETRY` можно объявлять столбцы таблиц, переменные и параметры процедур.) Метод `STIntersects()` возвращает значение `1`, если один экземпляр геометрической фигуры пересекает другой экземпляр геометрической фигуры. В примере 27.4 этот метод применяется к обеим областям, объявленным в переменных, чтобы определить, пересекаются ли эти области.

В запросе примера 27.5 показано использование метода `STIntersection()`.

Пример 27.5. Применение метода STIntersection()

```
USE   sample;
DECLARE @poly1 GEOMETRY = 'POLYGON ((1 1, 1 4, 4 4, 4 1, 1 1))';
DECLARE @poly2 GEOMETRY = 'POLYGON ((2 2, 2 6, 6 6, 6 2, 2 2))';
DECLARE Sresult GEOMETRY;
SELECT Sresult = @poly1.STIntersection(@poly2);
SELECT Sresult.STAsText();
```

Выполнение этого запроса дает следующий результат (значения округлены):

```
POLYGON ((22, 42, 44, 24, 22))
```

Метод STIntersection() возвращает объект, представляющий точки, где экземпляр типа данных GEOMETRY пересекается с другим экземпляром этого типа. Поэтому запрос, приведенный в примере 27.5, возвращает прямоугольник, в котором пересекаются многоугольник, объявленный в переменной @poly1, и многоугольник, объявленный в переменной @poly2. Метод STAsText() возвращает представление WKT экземпляра типа данных GEOMETRY, что является результатом выполнения примера.

ПРИМЕЧАНИЕ

Разница между методами STIntersects() и STIntersection() заключается в том, что первый метод проверяет пересечение двух геометрических объектов, тогда как второй возвращает объект пересечения.

Работа с типом данных *GEOGRAPHY*

С типом данных GEOGRAPHY работают таким же образом, как и с типом данных GEOMETRY. Это означает, что те же самые методы (статические и методы экземпляра), которые можно использовать с типом данных GEOMETRY, применимы и с типом данных GEOGRAPHY. Поэтому работа с этим типом данных показана только на одном примере 27.6.

Пример 27.6. Демонстрация работы с типом данных GEOGRAPHY

```
USE AdventureWorks;
SELECT SpatialLocation, City
   FROM Person.Address
   WHERE City = 'Dallas';
```

Этот запрос возвращает следующий результат:

SpatialLocation	City
0xE6100000010C4DD260393369404026C0A31BF73458C0	Dallas
0xE6100000010C10A810D1886240403A0F0653663158C0	Dallas
0xE6100000010C4346160AA26440406340F0E64F3 B58C0	Dallas

```
0xE6100000010C107E16DAAD6540403DA892EAD52C58C0            Dallas
0xE6100000010C8044A1422D5F4040F66D784F983758C0            Dallas
0xE6100000010C8E345943826A4040839B00B8E03358C0            Dallas
0xE6100000010CAA5BBD5FAB69404087866D198D3C58C0            Dallas
```

Таблица Address базы данных AdventureWorks содержит столбец SpatialLocation типа данных GEOGRAPHY. Запрос, приведенный в примере 27.6, отображает географическое расположение всех сотрудников, живущих в городе Dallas. Как можно видеть в результатах выполнения этого запроса, столбец SpatialLocation содержит шестнадцатеричные представления долготы и широты места проживания каждого сотрудника. (Далее в этой главе, в примере 27.10, будет показано отображение результатов этого запроса с помощью среды SQL Server Management Studio.)

Работа с пространственными индексами

Как вы уже знаете из *главы 10*, индексирование обычно применяется с целью предоставления более быстрого доступа к данным. Поэтому для ускорения операций выборки пространственных данных требуются пространственные индексы.

Пространственный индекс определяется по столбцу таблицы типа данных GEOMETRY или GEOGRAPHY. В сервере SQL Server для построения этих индексов применяются B-деревья, что означает, что эти индексы представляют два измерения в линейном порядке B-деревьев.

Поэтому, прежде чем закладывать данные в пространственный индекс, система разделяет пространство в иерархическом единообразном порядке. Процесс создания индекса разбивает пространство на четырехуровневую решеточную иерархию.

Для создания пространственного индекса используется инструкция CREATE SPATIAL INDEX. В общем, эта инструкция подобна стандартной инструкции CREATE INDEX, но содержит дополнительные опции и предложения, некоторые из которых описываются далее.

- GEOMETRY_GRID — это предложение задает используемую схему тесселяции решетки геометрического объекта. (Процесс тесселяции выполняется после считывания данных для пространственного объекта. В этом процессе объект вставляется в решеточную иерархию, связывая его с набором ячеек решетки, с которыми он соприкасается.) Обратите внимание, предложение GEOMETRY_GRID можно указывать только для столбца с типом данных GEOMETRY.
- BOUNDING_BOX — эта опция задает набор из четырех числовых значений, определяющих четыре координаты ограничивающего прямоугольника: X_{min} и Y_{min} координаты нижнего левого угла и X_{max} и Y_{max} координаты верхнего правого угла. Этот параметр применим только к предложению GEOMETRY_GRID.
- GEOGRAPHY_GRID — это предложение задает схему тесселяции решетки географического объекта. Это предложение можно применять только для столбцов с типом данных GEOGRAPHY.

В примере 27.7 показано создание пространственного индекса для столбца shape таблицы beverage_markets.

Пример 27.7. Создание пространственного индекса

```
USE sample;
GO
ALTER TABLE beverage_markets
   ADD CONSTRAINT prim_key PRIMARY KEY(id);
   GO
   CREATE SPATIAL INDEX i_spatial_shape
      ON beverage_markets(shape)
      USING GEOMETRY_GRID
      WITH (BOUNDING_BOX = (xmin=0, ymin=0, xmax=500, ymax=200),
         GRIDS = (LOW, LOW, MEDIUM, HIGH),
         PAD_INDEX = ON);
```

Пространственный индекс можно создать только в том случае, если для таблицы со столбцом пространственных данных явно определен первичный ключ. По этой причине, в первой инструкции ALTER TABLE запроса примера 27.7 определяется это ограничение.

В следующей инструкции CREATE SPATIAL INDEX создается пространственный индекс, используя для этого предложение GEOMETRY_GRID. Параметр BOUNDING BOX задает границы, в пределах которых будет помещен экземпляр столбца shape. В параметре GRIDS указывается плотность решетки на каждом уровне схемы тесселяции. (Описание параметра PAD_INDEX см. в *главе 10*.)

ПРИМЕЧАНИЕ

В версии SQL Server 2012 вводятся дополнительные пространственные индексы, которые описываются в *разд. "Новые возможности SQL Server 2012 для работы с пространственными данными" далее в этой главе*.

SQL Server поддерживает, среди прочих, три следующих представления каталога, связанных с пространственными данными:

- sys.spatial_indexes;
- sys.spatial_index_tessellations;
- sys.spatial_reference_systems.

Представление каталога sys.spatial_indexes отображает информацию основного индекса пространственных индексов (пример 27.8). С помощью представления каталога sys.spatial_index_tesselations можно отобразить информацию о схеме тесселяции и параметры всех существующих пространственных индексов. Представление каталога sys.spatial_reference_systems перечисляет все пространственные системы координат, поддерживаемые сервером SQL Server. Основными столбцами

этого представления являются столбцы `spatial_reference_id` и `well_known_text`. Первый столбец содержит однозначный идентификатор пространственной системы координат (SRID), а второй — ее описание.

В примере 27.8 показано использование представления каталога `sys.spatial_indexes`.

Пример 27.8. Использование представления каталога `sys.spatial_indexes`

```
USE sample;
SELECT object_id, name, type_desc
   FROM sys.spatial_indexes;
```

Выполнение этого запроса дает следующий результат:

object_id	Name	type_desc
914102297	i_spatial_shape	SPATIAL

Представление каталога в примере 27.8 отображает информацию о существующих пространственных индексах (созданных в примере 27.7).

ПРИМЕЧАНИЕ

Далее в *этой главе в разд. "Новые возможности SQL Server 2012 для работы с пространственными данными"* рассматриваются две новые системные процедуры, введенные в SQL Server 2012.

Отображение информации о пространственных данных

Разработчики Microsoft расширили функциональность среды SQL Server Management Studio возможностью отображения пространственных данных в графическом формате. Эта функциональность будет продемонстрирована на двух gjcktle.ob[примерах. В примере 27.9 используется тип данных `GEOMETRY`, а пример 27.10 основан на типе данных `GEOGRAPHY`.

Пример 27.9. Графическое отображение пространственных данных типа `GEOMETRY`

```
USE sample;
DECLARE @rectangle1 GEOMETRY = 'POLYGON((1 1, 1 4, 4 4, 4 1, 1 1))';
DECLARE @line GEOMETRY = 'LINESTRING (0 2, 4 4)';
SELECT @rectangle1
UNION ALL
SELECT @line
```

Для отображения в среде SQL Server Management Studio пространственных данных результата выполнения запроса в панели отображения результатов выполнения запроса выберите вкладку **Spatial Results**, которая находится справа от вкладки **Results**. В случае выполнения запроса, приведенного в примере 27.9, выводится прямоугольник, отображающий содержимое переменной `@rectangle1`, а также прямая согласно переменной `@line` (рис. 27.2).

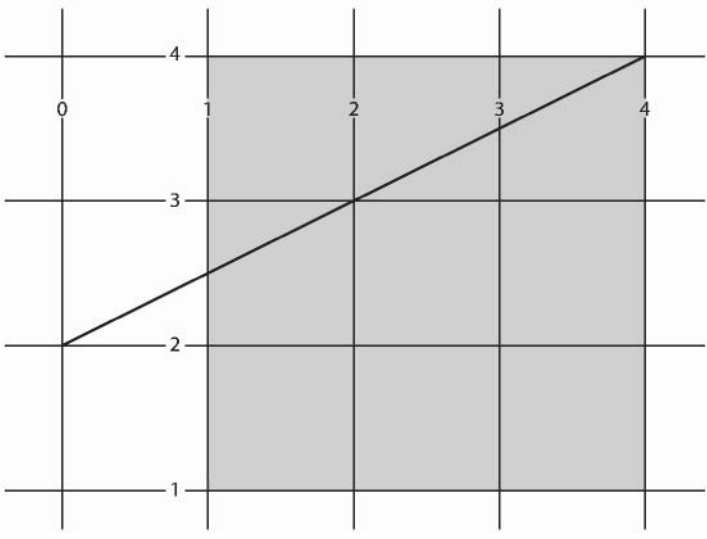


Рис. 27.2. Графическое отображение в среде SQL Server Management Studio результатов выполнения запроса из примера 27.9

ПРИМЕЧАНИЕ

Для отображения в среде SQL Server Management Studio нескольких объектов типа данных `GEOMETRY`, их нужно возвратить в виде нескольких строк одной таблицы. По этой причине в примере 27.9 используется две инструкции `SELECT`, результаты которых соединяются посредством предложения `UNION ALL`. (В противном случае одновременно будет отображаться только одна точка.)

В примере 27.10 приводится запрос для демонстрации графического отображения результирующего экземпляра типа данных `GEOGRAPHY`.

Пример 27.10. Графическое отображение пространственных данных типа `GEOGRAPHY`

```
USE AdventureWorks;
SELECT SpatialLocation, City
   FROM Person.Address
   WHERE City = 'Dallas';
```

Запрос в примере 27.10 такой же, что и в примере 27.6. Для просмотра графического варианта результатов выполнения запроса (рис. 27.3) в панели результатов снова откройте вкладку **Spatial Results**.

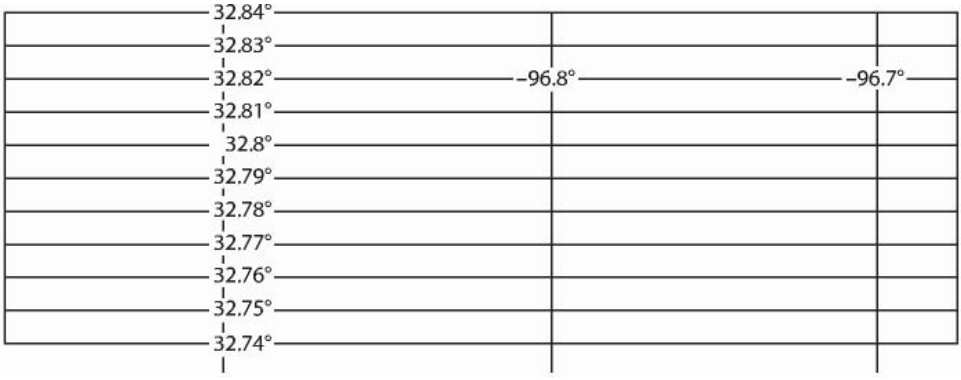


Рис. 27.3. Графическое отображение в среде SQL Server Management Studio результатов выполнения запроса из примера 27.10

ПРИМЕЧАНИЕ

При наведении указателя мыши на одну из точек, отображаемых в графическом варианте результатов (см. рис. 27.3), среда SQL Server Management Studio выводит адрес, связанный с этой точкой.

Новые возможности SQL Server 2012 для работы с пространственными данными

В SQL Server 2012 вводятся следующие улучшения для пространственных типов:

- новые подтипы дуг окружностей;
- новые пространственные индексы;
- новые системные хранимые процедуры, связанные с пространственными данными.

Все эти особенности описываются далее в следующих разделах.

Новые подтипы дуг окружностей

Дуги окружностей основаны на стандарте ANSI SQL/MM. Все стандарты ANSI, связанные с SQL, разделены на части по областям SQL. В стандарте SQL/MM описана третья часть, связанная с пространственными данными. Вообще, дуга представляет собой замкнутый отрезок кривой в двумерной плоскости. Следственно, дуга окружности является отрезком окружности круга. Дуги окружности можно задавать самостоятельно или совместно с отрезком прямой. Кроме этого, их можно использовать как основание для нового типа многоугольника, который содержит один или более дуговых компонентов.

Дуги окружности поддерживаются типами данных `GEOMETRY` и `GEOGRAPHY` и их можно определять в форматах WKT, WKB и GML.

Существуют следующие три новых типа дуг окружности:

- CircularString;
- CompoundCurve;
- CurvePolygon.

Эти типы дуг окружности рассматриваются далее в следующих подразделах.

Тип *CircularString*

На диаграмме иерархии типа GEOMETRY, приведенной на рис. 27.1, можно видеть, что тип CircularString является прямым подтипом типа Curve. По этой причине объекты типа CircularString являются базовым подтипом кривой.

Для определения объекта типа CircularString требуется задать, по крайней мере, три точки. Первая точка указывает начало дуги, вторая — ее конец, а третья должна быть в каком-либо месте дуги. Экземпляры типа CircularString можно соединять вместе, где последняя точка предыдущей кривой становится первой точкой следующей. В примере 27.11 с использованием переменной @g показано определение объекта типа CircularString.

Пример 27.11. Определение объекта типа CircularString

```
DECLARE @g GEOMETRY;
SET @g =
   GEOMETRY::STGeomFromText ('CIRCULARSTRING(0 -12.5, 0 0, 0 12.5)',0);
```

Тип *CompoundCurve*

Тип CompoundCurve позволяет определять новые составные кривые, состоящие или только из экземпляров типа CircularString, или из экземпляров типа CircularString и LineString. Концевая точка каждого предыдущего компонента связывается с начальной точкой следующего.

В примере 27.12 показано создание составной кривой, состоящей из нескольких разных объектов типа CircularString.

Пример 27.12. Определение объекта типа CompoundCurve

```
DECLARE @g GEOGRAPHY;
SET @g = GEOGRAPHY::STGeomFromText('
COMPOUNDCURVE(CIRCULARSTRING(0 -23.43778, 0 0, 0 23.43778),
CIRCULARSTRING(0 23.43778, -45 23.43778, -90 23.43778),
CIRCULARSTRING(-90 23.43778, -90 0, -90 -23.43778),
CIRCULARSTRING(-90 -23.43778, -45 -23.43778, 0 -23.43778) )' ,4326);
```

В примере 27.12 создается экземпляр типа данных GEOGRAPHY, который присваивается переменной @g. Данная переменная состоит из экземпляра типа CompoundCurve,

который в свою очередь состоит из экземпляров типа CircularString. Обратите внимание, что последним аргументом метода STGeomFromText() является значение 4326. Это значение по умолчанию идентификатора SRID для типа данных GEOGRAPHY и соответствует пространственной системе координат WGS 82. (Идентификатор SRID объясняется *ранее в этой главе*, в примечании после примера 27.1.)

Тип *CurvePolygon*

Объект типа CurvePolygon состоит из объектов типа LineString и CircularString, а также объектов типа CompoundCurves. Как можно видеть на диаграмме иерархии типа GEOMETRY, показанной на рис. 27.1, тип CurvePolygon является прямым подтипом типа Surfface и супертипом типа Polygon. Внутри данного кольца первая точка предыдущего компонента объекта CurvePolygon должна быть идентична последней точке следующего компонента.

Новые пространственные индексы

В SQL Server 2012 для типов данных GEOMETRY и GEOGRAPHY был введен новый пространственный индекс auto_grid_index. (Функциональность этого индекса для обоих типов подобна, поэтому рассматривается только индекс auto_grid_index для типа данных GEOMETRY.)

Индекс *auto_grid_index* для типа данных *GEOMETRY*

Применяемая в новом индексе auto_grid_index стратегия для определения правильного баланса между производительностью и эффективностью отличается от стратегии, рассмотренной *ранее в этой главе*. В этом индексе используется восемь уровней тесселяции для лучшего аппроксимирования объектов разных размеров. (В ранее рассмотренном пространственном индексе с "ручным" заданием решетки используется только четыре указываемые пользователем уровня тесселяции.)

В примере 27.13 показано создание автоматического индекса auto_grid_index для геометрического объекта.

Пример 27.13. Создание индекса auto_grid_index

```
CREATE SPATIAL INDEX auto_grid_index
   ON beverage_markets(shape)
   USING GEOMETRY_AUTO_GRID
      WITH (BOUNDING_BOX = (xmin=0, ymin=0, xmax=500, ymax=200 ),
   CELLS_PER_OBJECT = 32, DATA_COMPRESSION = page);
```

Создание индекса auto_grid_index указывается в предложении GEOMETRY_AUTO_GRID. В предложении CELLS_PER_OBJECT задается максимальное количество ячеек тесселяции, которое может использоваться для одного пространственного объекта в индексе. В предложении DATA_COMPRESSION указывается, включено ли сжатие данных

для данного пространственного индекса. (Все другие предложения инструкции CREATE SPACIAL INDEX объясняются сразу же после примера 27.7.)

Новые системные хранимые процедуры, касающиеся пространственных данных

В SQL Server 2012 были введены следующие системные хранимые процедуры:

- sp_help_spatial_geometry_index;
- sp_help_spatial_geography_index.

ПРИМЕЧАНИЕ

Поскольку эти обе процедуры имеют сходный синтаксис и функциональность, то в этой книге рассматривается только первая из них.

Системная хранимая процедура sp_help_spatial_geometry_index возвращает имена и значения для указанного набора свойств о пространственном индексе геометрического объекта. Результаты возвращаются в табличном формате. Можно задать возврат основного набора свойств или всех свойств индекса. Применение этой системной процедуры показано в примере 27.14.

Пример 27.14. Использование процедуры sp_help_spatial_geometry_index

```
DECLARE @query geometry
='POLYGON((-90.0 -180.0, -90.0 180.0, 90.0 180.0,
                              90.0 -180.0, -90.0 -180.0))' ;
EXEC sp_help_spatial_geometry_index 'beverage_markets',
                                 'auto_grid_index', 0, @query;
```

В примере 27.14 системная процедура sp_help_spatial_geometry_index отображает свойства пространственного индекса auto_grid_index, созданного в предыдущем примере 27.13.

Резюме

Сервер SQL Server поддерживает два пространственных типа данных: GEOGRAPHY и GEOMETRY. Тип данных GEOMETRY применяется в плоских пространственных моделях, а тип данных GEOGRAPHY — в геодезических моделях. Для работы с этими типами данных предоставляется набор соответствующих методов.

В частности, для обоих типов данных корпорация Microsoft реализовала методы, обусловленные консорциумом OGC (Open GIS Consortium), которые можно использовать для извлечения пространственных данных из столбцов таблиц.

Среда SQL Server Management Studio обеспечивает хорошую поддержку для типов данных GEOMETRY и GEOGRAPHY. В частности, эта среда позволяет отображать возвращаемые запросами пространственные данные в графическом формате.

В SQL Server 2012 вводятся следующие улучшения для пространственных типов:

- новые подтипы дуг окружностей: `CircularString`, `CompoundCurve` и `CurvePolygon`;
- пространственный индекс `auto_grid_index`;
- системные хранимые процедуры `sp_help_spatial_geometry_index` и `sp_help_spatial_geography_index`.

В следующей, и последней, главе этой книги рассматривается функциональность полнотекстового поиска SQL Server.

Глава 28

Полнотекстовый поиск в SQL Server

- **Введение**
- **Индексирование полнотекстовых данных**
- **Полнотекстовые запросы**
- **Диагностирование проблем с полнотекстовыми данными**
- **Новые возможности SQL Server 2012 по полнотекстовому поиску**

Эта глава состоит из пяти частей. Во вводной части описываются общие понятия, относящиеся к полнотекстовым данным, о которых нужно быть осведомленным, прежде чем начинать работать с такими данными. В этой части рассматриваются лексемы (token), средства для деления текста на слова (word breaker) и списки стоп-слов (stop list) и описывается их роль в полнотекстовом поиске. Также в ней дается введение в разные операции, которые можно выполнять с лексемами, и объясняется принцип работы полнотекстового поиска в SQL Server.

Во второй части описываются общие шаги, необходимые для создания полнотекстового индекса, и демонстрируется применение этих шагов, сначала используя язык Transact-SQL, а потом среду SQL Server Management Studio.

Третья часть отведена под рассмотрение полнотекстовых запросов. В ней описываются два предиката и две строчные функции, которые можно использовать для полнотекстового поиска. Для этих предикатов и функций дается несколько примеров, чтобы показать, как с их помощью можно решать конкретные проблемы, связанные с расширенными операциями.

В четвертой части главы обсуждаются динамические административные представления (Dynamic Management View — DMV), которые можно использовать для поиска и устранения проблем с полнотекстовыми индексами. Два из этих динамических представлений (DMV) можно использовать на этапе индексирования, а одно — в процессе поиска.

В последней части главы обсуждаются новые полнотекстовые возможности, реализованные в SQL Server 2012, которые включают возможность поиска по расширенным свойствам и усовершенствованную функциональность предложения NEAR в предикате CONTAINS и в строчной функции CONTAINSTABLE.

Введение

Компонент SQL Server, называемый *Full-Test Search* (FTS), позволяет выполнять поиск данных в текстовых документах. Такие данные обычно находятся в неструктурированном виде, что означает, что они содержат нерегулярности и неоднозначности, которые затрудняют понимание таких данных при использовании традиционных компьютерных программ. По этой причине для неструктурированных данных не существует предопределенной модели данных.

Компонент FTS сохраняет данные из текстовых документов таким же образом, как и текстовые данные сохраняются в реляционной модели. Это означает, что такие данные хранятся в столбцах таблиц, которые имеют тип данных CHAR, VARCHAR, NCHAR, NVARCHAR, XML, VARBINARY и VARBINARY(max). (Тип данных VARBINARY(max) может применяться как со свойством FILESTREAM, так и без него.)

ПРИМЕЧАНИЕ

Когда столбец данных двоичного типа (например, VARBINARY(max)) содержит документ с поддерживаемым расширением файла документа, компонент FTS интерпретирует двоичные данные с помощью фильтра. Этот фильтр, который реализует интерфейс IFilter, извлекает текстовую информацию из документа и предоставляет ее для индексирования. (Интерфейс IFilter рассмотрен в *следующем разделе*.)

Прежде чем обсуждать извлечение из документов, хранящихся в них данных, рассмотрим, как такие данные требуется подготовить для запроса поиска информации. В последующих разделах представлены общие концепции, которые связаны с полнотекстовыми данными и которые полезно знать, прежде чем приступать к индексированию данных в текстовых документах для их подготовки к поиску информации.

Лексемы, делители текста на слова и списки стоп-слов

Полнотекстовый запрос позволяет выполнять поиск слов в текстовых документах. Базовая единица такого запроса называется *лексемой*[1] (token). В европейских языках этот термин обычно имеет то же самое значение, что и *слово*. Но в восточных языках (например, в китайском, японском и т. п.) четкого понятия слова не существует, поскольку в этих языках значимые строки (т. е. последовательности симво-

[1] Лексема (от греч. *léxis* — слово, выражение) — это единица лексического уровня языка, его лексики, и представляет собой слово во всей совокупности его форм и значений. В полнотекстовом поиске — это слово или символьная строка, распознаваемая средством деления текста на слова. — *Ред.*

лов) не разделяются пробелами. В таком случае компоненту FTS требуется принимать решения относительно границ между двумя лексемами. (В этой главе термины *лексема* и *слово* используются взаимозаменяемо.)

Делители текста на слова и фильтры IFilter

Как мы увидим немного далее в этой главе, прежде чем можно запустить процесс запроса, компоненту FTS требуется проиндексировать данные в документах. А прежде чем выполнять индексирование, компонент FTS определяет границы лексем в тексте документа, используя для этого ориентированные на конкретный язык компоненты, которые называются специальными *делителями текста на слова* (word breakers). Для краткости, в последующем тексте данной главы эти средства будут называться просто *делителями*. Таким образом, основной задачей делителей является разделить содержимое текста на лексемы и решить, каким образом эти лексемы следует сохранить в полнотекстовом индексе. Делитель, предназначенный для английского языка, делит текст на слова по границам пробелов и знаков пунктуации.

Делители для языков иных, чем английский, индексируют составные слова и извлекают из них все составляющие слова. Рассмотрим, например, делитель для немецкого языка, в котором слово может состоять из нескольких других слов. Например, немецкое слово *Wortzusammensetzung* означает "сложное слово" и состоит из трех разных слов. Задачей делителя для немецкого языка является, среди прочего, выполнять анализ таких составных слов и извлекать из них составляющие слова или символы.

Индексирование документов в столбце типа данных `VARBINARY`, `VARBINARY(max)` или `XML` требует дополнительной обработки, для выполнения которой требуется применения фильтра. Этот фильтр извлекает текстовую информацию из документа и отправляет текст для обработки делителю того языка, который связан с данным столбцом. В компоненте FTS такой фильтр называется *IFilter*.

Выбор конкретного фильтра IFilter зависит от типа данных столбца таблицы, в котором хранятся данные. Это означает, что для столбцов с типом данных `CHAR`, `NCHAR`, `VARCHAR` и `NVARCHAR` применяется текстовый фильтр IFilter. Подобным образом для столбцов с типом данных `XML` компонент FTS применяет XML-фильтр IFilter. (Пользователь изменить выбор фильтра не может.)

Для столбцов с типом данных `VARBINARY` компонент FTS применяет фильтр IFilter, который соответствует расширению файла документа. Иными словами, для документа Word это будет расширение *docx*, а для презентации PowerPoint — расширение *pptx*.

Списки стоп-слов

После того как делитель выполнит свою работу, к тексту применяется список стоп-слов (stop list). *Список стоп-слов* содержит набор определенных стоп-слов. *Стоп-словами* (*stop words* или *noise words*) называются слова, которые игнорируются при

полнотекстовом поиске, вследствие их низкой значимости (релевантности[1]) в данном тексте. (В качестве примера русских стоп-слов можно назвать, среди прочих, такие слова, как *и*, *а*, *ну* и т. п.) Практический эффект игнорирования этих слов заключается в том, что текст становится короче, а для хранения соответствующего полнотекстового индекса требуется меньше места.

ПРИМЕЧАНИЕ

Начиная с версии SQL Server 2008, списки стоп-слов хранятся в базе данных, к которой они принадлежат. В базе данных можно хранить несколько списков стоп-слов.

Списки стоп-слов можно создавать с помощью среды SQL Server Management Studio. Существующий список стоп-слов можно редактировать, добавляя в него новые слова, удаляя существующие отдельные или все стоп-слова или удаляя весь список.

Операции с лексемами

Поскольку полнотекстовый поиск работает с неструктурированными данными, то для него требуются определенные специальные операции. Эти операции можно разделить на три следующие группы:

- расширенные операции над словами;
- операции задания параметров соответствия;
- операции определения близости.

Эти три типа операций рассматриваются в следующих далее подразделах.

ПРИМЕЧАНИЕ

Указанные полнотекстовые операции можно комбинировать, используя логические операторы `AND`, `OR` и `NOT`.

Расширенные операции над словами

В *главе 6* мы научились, как извлекать информацию из реляционных таблиц, используя оператор `LIKE`, который сравнивает значения столбца с указанным шаблоном. Эта операция очень ограничена, поскольку она может находить только точные совпадения с указанным шаблоном. Компонент FTS имеет возможность выполнять поиск не только слов, в точности совпадающих с шаблоном, но и родственных слов. Например, при поиске слова *person* можно ожидать, что найдутся такие родственные слова как *persons* и *personal*. При полнотекстовом поиске должен также выполняться поиск более специфических родственных слов, таких как *man* и *woman*, а также синонимов, таких как *individual*.

---

[1] Релевантность (лат. *relevo* — поднимать, облегчать) в информационном поиске — семантическое соответствие поискового запроса и поискового образа документа. — *Ред.*

Другой тип расширенного поиска исправляет ошибки в термине запроса и в тексте, в котором выполняется поиск. Ошибки при вводе термина поиска являются распространенным явлением, поэтому при полнотекстовом поиске требуется исправлять распространенные ошибки, например неправильное *heirarchy* вместо правильного *hierarchy*.

Операции задания параметров соответствия

Параметры соответствия определяют, какие факторы имеют значение при решении, соответствует ли слово в тексте, примененному шаблону поиска. Распространенным заданием параметра соответствия является задание чувствительности к регистру при поиске. Например, если требуется найти определенное слово только тогда, когда оно начинается с заглавной буквы, и игнорировать это же слово, начинающееся с прописной буквы, то слово поиска можно ввести с начальной заглавной буквы и затем указать, что поиск должен быть чувствителен к регистру. Другим примером задания параметра соответствия будет использование *групповых символов* (wildcards). Например, в слове, которое используется для поиска, можно применить символ звездочки (*) для представления любого символа в данной позиции слова. Тогда для слова поиска *lim** будут возвращены совпадения *limb*, *lime*, *limo* и *limp*. Можно также задать такой параметр соответствия, который будет игнорировать или не игнорировать диакритические знаки[1]. Диакритические знаки добавляются в буквы не как самостоятельные обозначения звуков, а для изменения или уточнения их звучания. Использование диакритических знаков в английском языке можно продемонстрировать такими словами, как *naïf* и *café*.

Операции определения близости

Если при поиске нескольких слов требуется возвратить результаты только для тех их экземпляров, которые в данном тексте встречаются рядом друг от друга, можно применить *операцию определения близости* (proximity operation). Например, если нужно узнать, связаны ли в данном тексте слова *Microsoft* и *management*, можно указать, чтобы компонент FTS возвращал только те совпадения с этими словами, в которых они встречаются в указанной близости друг от друга. Для указания таких операций определения близости компонент FTS применяет предложение NEAR (использование которого продемонстрировано в примере 28.10 *далее в этой главе*).

ПРИМЕЧАНИЕ

Как рассматривается в *разд. "Полнотекстовые запросы" далее в этой главе*, компонент FTS поддерживает все три типа только что рассмотренных расширенных операций полнотекстового поиска.

[1] Диакритический знак (от греч. *diakritikos* — различительный) — лингвистический знак при букве, указывающий на то, что она читается иначе, чем без него. Ставится над буквой, ниже буквы или пересекая ее. Исключение составляет буква "i". В современном русском языке диакритическим знаком являются две точки над "е", а именно "ё". — *Ред.*

Коэффициент релевантности

Реляционные запросы всегда возвращают точные ответы, чего нельзя сказать о полнотекстовом поиске. Результирующий набор полнотекстового запроса может быть субъективным, а упорядочение его элементов зависеть от системы, используемой для поиска.

Компонент FTS присваивает каждому проиндексированному термину *коэффициент релевантности* (relevance score). Этот коэффициент основан на оценке важности, или иначе значимости концепта, представляемого термином. (Значимость измеряется количеством вхождений термина в текст документа.) Почти все инструменты полнотекстового поиска используют различные алгоритмы для определения коэффициента релевантности для каждого запроса.

Как работает компонент FTS сервера SQL Server

Прежде чем выполнять запрос по текстовому документу, его необходимо проиндексировать. Это требуется по той причине, что эффективный поиск лексем текстового документа можно выполнять только в том случае, если они проиндексированы. Компонент FTS создает полнотекстовый индекс в процессе, называемым *заполнением* (population), который заполняет индекс словами и их местонахождением в документе. Полнотекстовые индексы хранятся в каталогах. Для базы данных можно задать один или больше каталогов, но каждый каталог всегда принадлежит к определенной базе данных.

Полнотекстовые индексы активируются по отдельности для каждого каталога. Каталог можно заполнить одним из двух способов.

- *Полным заполнением*. Все данные каталога удаляются и каталог заполняется сначала, создавая элементы индекса для всех строк всех таблиц, охватываемых каталогом. Полное заполнение обычно применяется при первом заполнении каталога.
- *Инкрементальным заполнением*. Корректируются только элементы индекса для тех строк, которые уже были изменены со времени последнего заполнения. Преимуществом инкрементального заполнения является скорость: этот способ заполнения минимизирует время заполнения индекса, поскольку число сохраняемых элементов в этом случае будет значительно меньше, чем при полном заполнении.

ПРИМЕЧАНИЕ

Система всегда выполняет полное заполнение индекса при модифицировании структуры таблицы, включая изменение определения любого столбца, индекса или полнотекстового индекса.

Компонент FTS заполняет полнотекстовые индексы автоматически. Это означает, что подобно обычным индексам SQL Server, полнотекстовые индексы можно автоматически обновлять при изменении (добавлении, обновлении или удалении) дан-

ных соответствующих таблиц. Кроме этого, полнотекстовый каталог можно заполнять или сразу же или по расписанию. В обоих случаях можно применять полное или инкрементальное заполнение.

Поскольку полнотекстовые индексы могут занять значительный объем дискового пространства, важно уделить внимание планированию их размещения в полнотекстовые каталоги. При помещении таблицы в каталог следует рассмотреть несколько рекомендаций.

- Минимизируйте размер полнотекстового индекса насколько это возможно. Одним из способов для этого в качестве ключа следует выбрать самый короткий столбец.
- При индексировании большой таблицы выделите ей свой собственный полнотекстовый каталог.

С этими знаниями основ полнотекстового поиска можно приступить к рассмотрению создания полнотекстовых индексов с помощью SQL Server FTS.

Индексирование полнотекстовых данных

Для индексирования текстовых документов, хранящихся в столбце таблицы, нужно выполнить следующие шаги:

1. Обеспечить, что в таблице есть столбец, в котором запрещены значения `NULL`, и что для этого столбца указано свойство `UNIQUE`.
2. Разрешить полнотекстовое индексирование базы данных, к которой принадлежит данная таблица.
3. Создать полнотекстовый каталог для хранения полнотекстовых индексов.
4. Создать полнотекстовый индекс по столбцу, для которого требуется полнотекстовое индексирование.

Все эти операции можно выполнить или посредством инструкций языка Transact-SQL, или же с помощью среды SQL Server Management Studio. Оба эти способа описываются в последующих далее разделах.

Полнотекстовое индексирование посредством Transact-SQL

Чтобы подробно исследовать, как выполнять перечисленные в предыдущем разделе задачи посредством языка Transact-SQL, сначала в базе данных `sample` нужно создать таблицу `product`, как это показано в примере 28.1. (Если же ваша база данных `sample` уже содержит таблицу с именем `product`, то прежде чем выполнять запрос примера 28.1, эту таблицу с помощью инструкции `DROP TABLE` нужно удалить.) Таблица `product` требуется для демонстрации использования полнотекстовых запросов в следующем разделе.

Пример 28.1. Создание таблицы product в базе данных sample

```
USE sample;
CREATE TABLE product
   (product_id CHAR(5) NOT NULL,
    product_name VARCHAR (30) NOT NULL,
    description VARCHAR(900) NOT NULL);
```

Таблица product, созданная с помощью приведенного в примере 28.1 запроса, отвечает всем требованиям для создания полнотекстового индекса. Прежде всего, она содержит столбец, в котором не допускаются значения NULL, а именно столбец product_id. Кроме того, столбец description этой таблицы имеет тип данных VARCHAR, вследствие чего он поддается полнотекстовому поиску.

Создание однозначного индекса

Запрос в примере 28.2 создает однозначный индекс по столбцу product_id и вставляет две строки данных в таблицу product. Эти строки требуются для демонстрации выполнения полнотекстовых запросов.

Пример 28.2. Создание индекса и вставка текстовых данных в таблицу

```
USE sample;
CREATE UNIQUE INDEX ind_description
   ON dbo.product(product_id);
GO
INSERT INTO product VALUES (1, 'MS Application Center ', 'This
Microsoft product simplifies the deployment and management of Windows
DNA solutions within farms of servers. Application Center makes it
easy to configure and manage high-availability server arrays ');
INSERT INTO product VALUES (2, 'MS Commerce Server ', 'Commerce Server
is the fastest way to build an effective Microsoft online business.
It provides all of the personalization, user and product management,
marketing, closed loop analysis, and electronic ordering
infrastructure necessary for both business-to-business and business-
to-consumer e-commerce.');
```

Разрешение полнотекстового индексирования для базы данных

Для разрешения (или запрещения) полнотекстового индексирования для базы данных используется системная хранимая процедура sp_fulltext_dtabase. Эта хранимая процедура имеет следующий синтаксис:

```
[EXEC] sp_full_text_database [@action=] 'enable' | 'disable'
```

Аргумент enable разрешает полнотекстовое индексирование для текущей базы данных, а аргумент disable удаляет из файловой системы все полнотекстовые каталоги для текущей базы данных. В примере 28.3 показано, как разрешить полнотекстовое индексирование для базы данных sample.

Пример 28.3. Разрешение полнотекстового индексирования для базы данных sample

```
USE sample;
EXEC sp_fulltext_database 'enable';
```

Альтернативно полнотекстовое индексирование базы данных можно разрешить с помощью среды SQL Server Management Studio. Для этого с помощью правой кнопки мыши щелкните в обозревателе объектов требуемую базу данных, выберите в контекстном меню пункт **Properties**, а в открывшемся окне свойств базы данных выберите вкладку **Files**. Установите на этой вкладке флажок **Use full-text indexing** и нажмите кнопку **OK**.

Создание полнотекстового каталога

Для создания полнотекстового каталога можно использовать инструкцию `CREATE FULLTEXT CATALOG` или же системную процедуру `sp_system_catalog`.

ПРИМЕЧАНИЕ

В будущих версиях SQL Server системная процедура `sp_fulltext_catalog` будет удалена. Поэтому в этом разделе рассматривается только использование инструкции `CREATE FULLTEXT CATALOG`.

Инструкция `CREATE FILLTEXT CATALOG` создает полнотекстовый каталог для базы данных. Один полнотекстовый каталог может иметь несколько полнотекстовых индексов, но полнотекстовый индекс может быть частью только одного полнотекстового каталога. Каждая база данных может содержать от нуля до нескольких полнотекстовых каталогов.

ПРИМЕЧАНИЕ

Начиная с SQL Server 2008, полнотекстовые каталоги являются частью базы данных, к которой они принадлежат. Это позволяет одновременно пересоздавать все индексы в каталоге.

В примере 28.4 показано, как создать полнотекстовый каталог для базы данных `sample`.

Пример 28.4. Создание полнотекстового каталога для базы данных sample

```
USE sample;
CREATE FULLTEXT CATALOG sample_catalog
   WITH ACCENT_SENSITIVITY = OFF
   AS DEFAULT;
```

В примере 28.4 для базы данных `sample` создается полнотекстовый каталог `sample_catalog`. Предложение `AS DEFAULT` задает этот каталог в качестве каталога по

умолчанию для базы данных sample. Это означает, что каждый полнотекстовый индекс, при создании которого явно не указывается каталог для его хранения, будет храниться в этом каталоге.

ПРИМЕЧАНИЕ

Не используйте каталог по умолчанию для хранения полнотекстовых индексов большой таблицы! С точки зрения повышения производительности, для такой таблицы следует создавать свой собственный полнотекстовый каталог.

В примере 28.4 предложение ACCENT_SENSITIVITY устанавливает чувствительность или нечувствительность каталога к диакритическим знакам. По умолчанию устанавливается чувствительность к диакритическим знакам (значение ON).

ПРИМЕЧАНИЕ

Кроме только что рассмотренных предложений, инструкция CREATE FULLTEXT CATALOG поддерживает другие предложения, такие как ONFILEGROUP, IN PATH и AUTHORIZATION. Описание этих предложений см. в *электронной документации*.

Язык Transact-SQL также поддерживает инструкции ALTER FULLTEXT CATALOG и DROP FULLTEXT CATALOG. Первая инструкция применяется для изменения свойств существующего каталога. Самым важным предложением этой инструкции является предложение REBUILD, которое указывает системе вновь создать весь каталог. При создании каталога вновь существующий каталог удаляется и вместо него создается полностью новый каталог. Инструкция DROP FULLTEXT CATALOG полностью удаляет полнотекстовый каталог. Прежде чем удалять полнотекстовый каталог, необходимо удалить все полнотекстовые индексы, связанные с ним.

Создание полнотекстового индекса

Инструкция CREATE FULLTEXT INDEX создает полнотекстовый индекс для указанной таблицы. Для таблицы разрешается иметь только один полнотекстовый индекс. Использование этой инструкции показано в примере 28.5.

Пример 28.5. Создание полнотекстового индекса для таблицы sample

```
USE sample;
CREATE FULLTEXT INDEX
   ON dbo.product(description)
   KEY INDEX ind_description
   ON sample_catalog;
```

В примере 28.5 полнотекстовый индекс создается для столбца description таблицы product. Как можно видеть в этом примере, синтаксис инструкции для создания полнотекстового индекса подобен синтаксису инструкции CREATE INDEX для создания обычного индекса. Дополнительное предложение KEY INDEX задает имя одно-

значного, не содержащего значений NULL индекса, который необходим для создания полнотекстового индекса (см. пример 28.2). Во втором предложении ON указывается имя каталога для хранения полнотекстового индекса. (В данном примере это предложение ON можно опустить, поскольку для базы данных sample используется каталог по умолчанию sample_catalog.)

Эта инструкция имеет два важных предложения: LANGUAGE и TYPE COLUMN. Предложение LANGUAGE содержит параметр, которому можно присвоить строковое, целочисленное или шестнадцатеричное значение, задающее соответствующий идентификатор языка LCID (locale identifier — идентификатор региона). Если этот параметр не задан, то используется язык по умолчанию компонента Database Engine. (Язык, соответствующий данному идентификатору LCID, можно узнать с помощью представления каталога sys.fulltext_language.)

ПРИМЕЧАНИЕ

Компонент FTS поддерживает полнотекстовые операции со многими различными языками. Информацию о поддерживаемых языках можно получить посредством представления каталога sys.syslanguages.

Предложение TYPE COLUMN требуется в том случае, когда в столбце с полнотекстовым индексом хранятся двоичные данные (например, типа VARBINARY(max)). В предложении TYPE COLUMN указывается имя другого столбца в таблице, в котором хранится расширение файла для двоичных данных. Например, двоичными данными может быть файл с расширением docx. Сервер SQL Server использует указанный в предложении TYPE COLUMN столбец для установления соотношения между двоичными данными и соответствующей системой программного обеспечения.

Язык Transact-SQL также поддерживает инструкции ALTER FULLTEXT INDEX и DROP FULLTEXT INDEX. Первая инструкция используется для изменения свойств существующего полнотекстового индекса. Наиболее важными предложениями этой инструкции являются предложения ADD и DROP. Эти предложения указывают системе добавить или удалить заданные столбцы соответственно. (Предложение DROP следует использовать только со столбцами, для которых разрешено полнотекстовое индексирование.) Инструкция DROP FULLTEXT INDEX применяется для удаления указанного индекса.

Полнотекстовое индексирование с помощью среды SQL Server Management Studio

Как упоминалось ранее, операции по индексированию текстовых документов, хранящихся в столбце таблицы, можно также выполнять с помощью среды SQL Server Management Studio. Для этого в обозревателе объектов последовательно разверните узел сервера, папку **Databases**, требуемую базу данных и папку **Tables**. Щелкните правой кнопкой таблицу, для которой требуется создать полнотекстовый индекс, и в появившемся контекстном меню выберите пункт **Full-Text Index**, а затем пункт

Define Full-Text Index. В результате будет запущен мастер полнотекстового индексирования *Full-Text Indexing Wizard*. На начальной странице мастера нажмите кнопку **Next**. Откроется страница выбора индекса **Select an Index** (рис. 28.1).

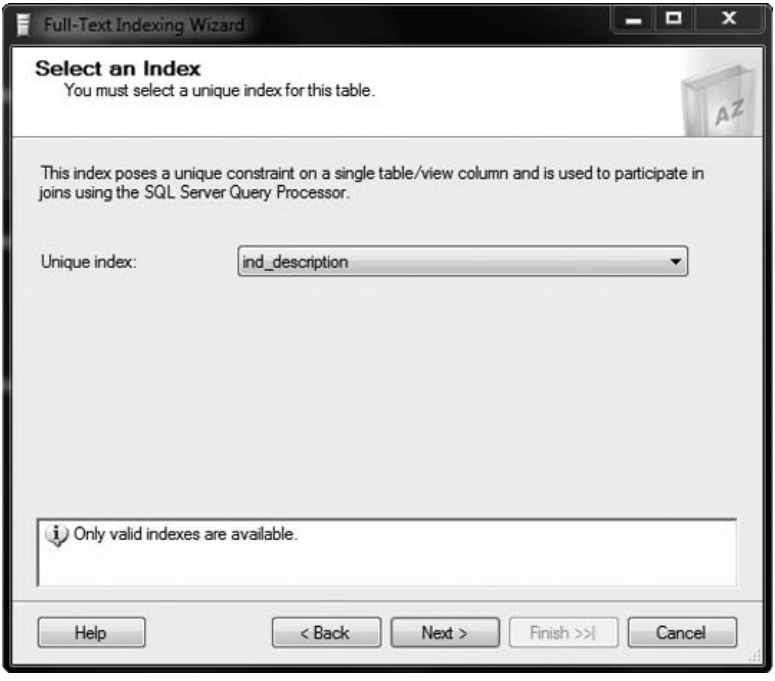


Рис. 28.1. Страница выбора индекса **Select an Index**

Создайте однозначный, не содержащий значений `NULL` индекс, который будет использоваться как исходная точка для полнотекстового индексирования (см. пример 28.2), а затем нажмите кнопку **Next**.

ПРИМЕЧАНИЕ

Если в таблице нет столбца с такими свойствами, то будет выведено сообщение о необходимости определения однозначного столбца для данной таблицы: "A unique column must be defined on this table/view". В таком случае модифицируйте структуру таблицы, чтобы она отвечала требуемым условиям.

На следующей странице, **Select Table Columns** (рис. 28.2), мастер выбирает из таблицы и отображает все столбцы с текстовыми данными и данными изображений.

Установите флажки для столбцов, которые требуется сделать доступными для полнотекстового поиска, и нажмите кнопку **Next**. (Обратите внимание, что если выбрать несколько столбцов, то компонент FTS создает один составной полнотекстовый индекс для всех выбранных столбцов.)

На следующей странице мастера **Select Change Tracking** (рис. 28.3) можно выбрать способ отслеживания изменений в таблице (выбрав соответствующий переключа-

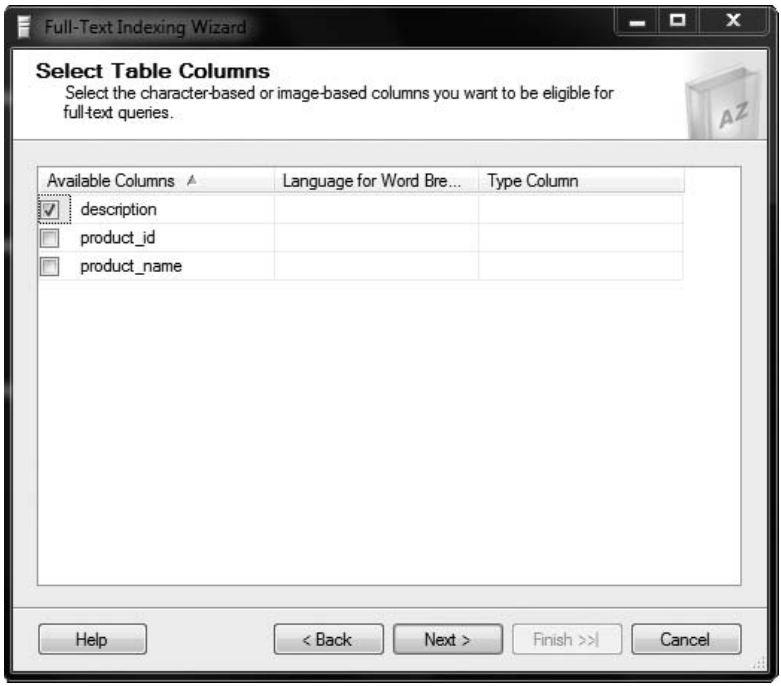


Рис. 28.2. Страница **Select Table Columns**

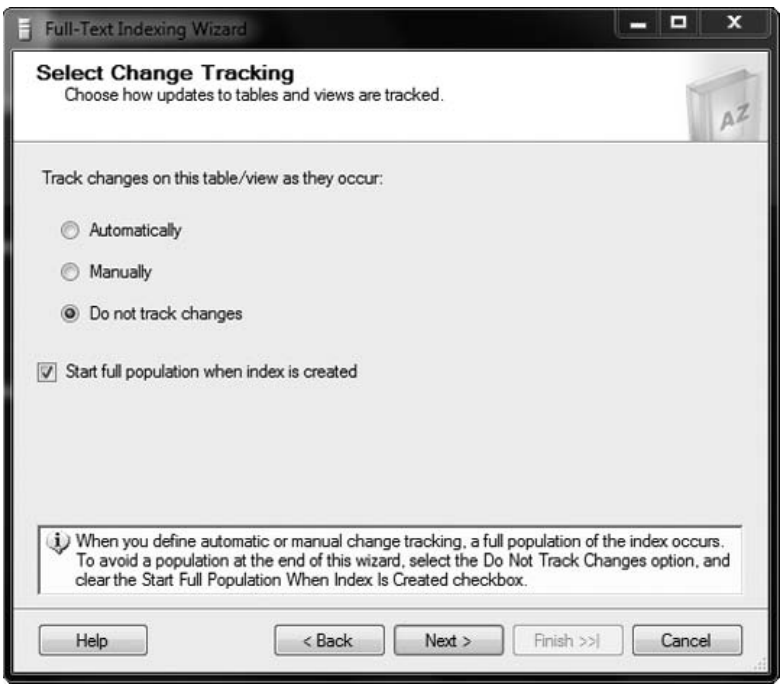


Рис. 28.3. Страница мастера **Select Change Tracking**

тель), а также задать вариант полного заполнения индекса при его создании, установив флажок.

При выборе автоматического (**Automatically**) или ручного (**Manually**) режима отслеживания изменений таблицы выполняется полное заполнение индекса. Чтобы избежать выполнения полного заполнения индекса по завершению работы мастера, задайте опцию не отслеживать изменения, выбрав переключатель **Do not track changes**, а также сбросьте флажок **Start full population when index is created**. Задав все требуемые параметры, нажмите кнопку **Next**. На следующей странице мастера, **Select Catalog, Index Filegroup, and Stoplist** (рис. 28.4), можно выбрать существующий каталог полнотекстового индекса или создать новый. Осуществив требуемые действия, нажмите кнопку **Next**.

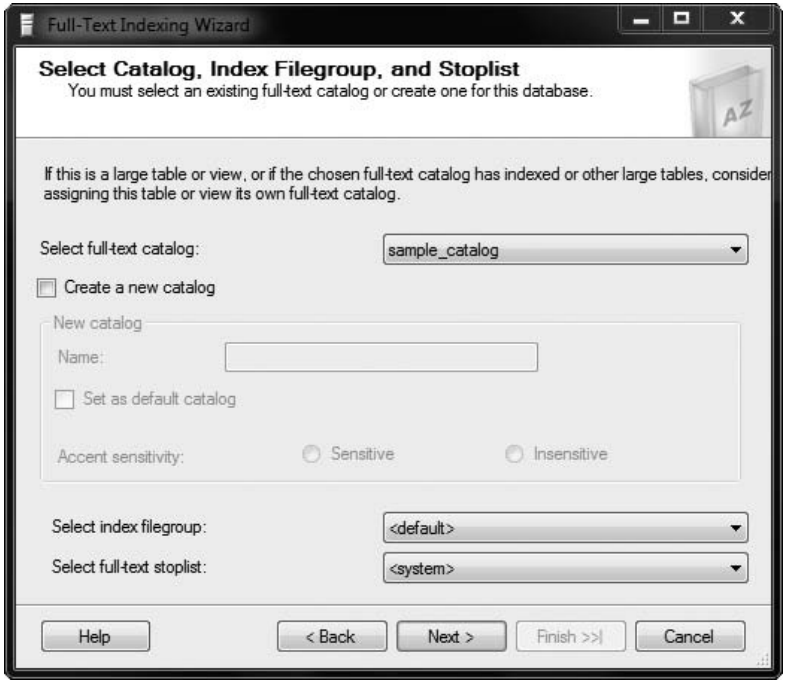


Рис. 28.4. Страница мастера
Select Catalog, Index Filegroup, and Stoplist

Выполнение операций на следующей странице **Define Population Schedules** не является обязательным. В нашем примере пропускаем эту страницу и нажимаем кнопку **Next**. Откроется страница для итогового просмотра выполненных операций и установленных параметров. Чтобы завершить процесс создания полнотекстового индекса, нажмите кнопку **Finish**.

Полнотекстовые запросы

Компонент FTS для выполнения запросов по тексту с использованием полнотекстового индекса поддерживает два следующих предиката и две функции:

- предикат FREETEXT;
- предикат CONTAINS;
- функцию FREETEXTTABLE;
- функцию CONTAINSTABLE.

Эти предикаты и функции рассматриваются в последующих разделах.

Предикат *FREETEXT*

Предикат FREETEXT используется для поиска значений в столбцах с полнотекстовым индексом, которые соответствуют значению условия поиска. Кроме того, этот предикат можно использовать для поиска всех слов, связанных со словом (или словами) запроса поиска.

В примере 28.6 демонстрируется использование предиката FREETEXT для поиска определенной строки.

Пример 28.6. Поиск строки с помощью предиката FREETEXT

```
USE sample;
SELECT product_id, product_name
   FROM product
   WHERE FREETEXT(description, 'manage');
```

Выполнение этого запроса дает следующий результат:

product_id	product_name
1	MS Application Center

Как можно видеть в примере 28.6, предикат FREETEXT имеет два аргумента. В первом аргументе указывается столбец, для которого выполнен полнотекстовый индекс, а во втором — строка, поиск которой выполняется в документах, хранящихся в указанном столбце (или столбцах).

В примере 28.7 предикат FREETEXT используется для поиска в тексте строк, подобных строке поиска.

Пример 28.7. Поиск слов, подобных словам поиска

```
USE sample;
SELECT product_id, product_name
   FROM product
   WHERE FREETEXT(description, 'fast solution');
```

Этот запрос возвращает следующий результат:

product_id	product_name
1	MS Application Center

В примере 28.7 демонстрируется две концепции: использование неявного логического оператора OR и поиск в тексте слов, которые похожи на слова поиска, но не совпадают с ними в точности. (Эта операция принадлежит к категории расширенных операций со словами.) Хотя может выглядеть так, что второй аргумент предиката FREETEXT в примере 28.7 задает поиск в тексте строки fast solution, в действительности он задает поиск одной из двух разных строк — fast и solution (или их обеих). (Наличие в строке поиска нескольких строк неявно задает применение логического оператора OR.) Кроме этого, ни одна из двух строк таблицы не содержит ни слово fast, ни слово solution. Но, как уже упоминалось, предикат FREETEXT выполняет поиск слов, подобных словам поиска. Поэтому компонент FTS находит слово solutions в первом документе и слово fastest во втором, считая эти слова родственными словам solution и fast.

Предикат *CONTAINS*

Предикат CONTAINS используется для поиска в столбцах, содержащих текстовые данные, точных и приблизительных совпадений. Он также позволяет выполнять поиск определенных слов и фраз, поиск слов в заданной близости друг от друга, а также частотный поиск. *Частотный поиск* (weighted search или frequency search) основывается на частоте вхождения слов поиска в документы, в которых выполняется поиск. Частотный поиск часто применяется в поисковых системах. Этот тип поиска возвращает число вхождений слов поиска в документ.

ПРИМЕЧАНИЕ

Предикат CONTAINS предоставляет достаточно высокий уровень функциональности в плане полнотекстового поиска, чем предикат FREETEXT.

В примере 28.8 показано использование предиката CONTAINS для поиска с применением группового символа подстановки. Эта операция принадлежит к категории операций поиска с заданием параметров соответствия.

Пример 28.8. Использование предиката CONTAINS с использованием группового символа

```
USE sample;
SELECT product_id, product_name FROM product
   WHERE CONTAINS(description, ' "config*" ');
```

Выполнение этого запроса дает следующий результат:

product_id	product_name
1	MS Application Center

Предикат CONTAINS в данном примере выполняет поиск в столбце description документа любого слова, начинающегося с config. Символ * является групповым символом предиката CONTAINS и обозначает любую последовательность символов от нуля и более.

Следственно, этот символ имеет такое же значение, как и символ % предиката LIKE. (Как можно видеть в примере 28.8, шаблон поиска, содержащий групповой символ, необходимо заключать в дополнительные двойные кавычки.)

Предикат CONTAINS позволяет комбинировать несколько операций, используя логические операторы AND, OR и NOT. Использование логического оператора AND в предикате CONTAINS показано в примере 28.9.

Пример 28.9. Использование логического оператора AND в предикате CONTAINS

```
USE sample;
SELECT product_id, product_name
   FROM product
   WHERE CONTAINS(description, ' "manage*" AND "market*" ');
```

Этот запрос приводит к следующему результату:

product_id	product_name
2	MS Commerce Server

Запрос в примере 28.9 выбирает все строки, которые содержат слово, начинающееся с manage, и слово, начинающееся с market.

В примере 28.10 показано использование опции NEAR для поиска слов, расположенных в тексте вблизи друг от друга. Эта операция относится к категории операций определения близости.

Пример 28.10. Использование оператора NEAR в предикате CONTAINS

```
USE sample;
SELECT product_id, product_name FROM product
   WHERE CONTAINS(description, 'Microsoft NEAR management');
```

Выполнение этого запроса дает следующий результат:

product_id	Product_name
1	MS Application Center
2	MS Commerce Server

Как можно видеть по результатам выполнения запроса примера 28.10, компонент FTS возвращает обе строки, поскольку слова Microsoft и management встречаются сравнительно близко друг от друга в обоих документах (см. также пример 28.17).

ПРИМЕЧАНИЕ

Компонент FTS возвращает идентификаторы строк, а также их коэффициент релевантности. Система использует коэффициент релевантности для упорядочения элементов результирующего набора. По результатам выполнения запроса примера 28.10 можно видеть, что коэффициент релевантности второй строки выше, чем первой. Это очевидно, т. к. слова `Microsoft` и `management` расположены ближе друг к другу во второй строке, чем в первой.

В примере 28.11 показано использование в предикате `CONTAINS` предложения `FORMSOF` со спецификацией `INFLECTIONAL`.

Пример 28.11. Использование предложения FORMSOF

```
USE sample;
SELECT product_id, product_name FROM product
   WHERE CONTAINS(description,'FORMSOF (INFLECTIONAL, provide)');
```

Этот запрос дает следующий результат:

product_id	product_name
2	MS Commerce Server

Спецификация `INFLECTIONAL` задает возвращение различных форм строки поиска — множественное и единственное число существительных, сравнительные формы прилагательных и разные времена и лица глаголов. Соответственно, инструкция `SELECT` в примере 28.11 возвращает слово `provides`, как форму глагола `provide`.

Функция *FREETEXTTABLE*

Функция `FREETEXTTABLE` возвращает таблицу, которая содержит одну или больше строк для тех столбцов, которые содержат значения, совпадающие со словами в строках поиска. Аналогично предикату `FREETEXT`, функция `FREETEXTTABLE` также возвращает строки, содержащие слова не совсем точно совпадающие со словами условия поиска. Функция `FREETEXTTABLE` использует такое же условие поиска, что и предикат `FREETEXT`, но обладает дополнительной функциональностью: запросы, использующие эту функцию, указывают коэффициент релевантности для каждой строки, что позволяет отображать первые *n* совпадений. (Предикат `FREETEXT` также вычисляет коэффициент релевантности для всех строк результирующего набора, но не использует этот коэффициент для упорядочения элементов отображаемого набора.)

ПРИМЕЧАНИЕ

Функцию `FREETEXTTABLE` можно использовать в предложении `FROM` инструкции `SELECT` как обычное имя таблицы.

Использование функции `FREETEXTTABLE` показано в примере 28.12.

Пример 28.12. Использование функции FREETEXTTABLE в запросе

```
USE sample;
SELECT pr.product_id, pr.product_name
   FROM product pr, FREETEXTTABLE(product, description,
                                  'fast solution1,1) ftt
     WHERE pr.product_id = ftt.[key];
```

Этот запрос возвращает следующий результат:

product_id	product_name
1	MS Application Center

Запрос в примере 28.12 соединяет две таблицы: таблицу product и таблицу, возвращаемую функцией FREETEXTTABLE. Как можно видеть в примере 28.12, функция FREETEXTTABLE принимает четыре параметра. В первом параметре указывается имя таблицы (product в данном примере), во втором — имя столбца таблицы, в котором выполняется полнотекстовый поиск (description в этом примере), в третьем — строка поиска, а в последнем задается целое число, ограничивающее отображаемый результирующий набор первыми *n* строками, упорядоченными по значению столбца rank. (Возвращаемая функцией FREETEXTTABLE таблица содержит два неявных столбца, называющиеся key и rank, к которым можно обращаться в запросе для получения соответствующих значений строк и коэффициента релевантности соответственно.)

ПРИМЕЧАНИЕ

Пример 28.12 похож на пример 28.7, но эти примеры возвращают разные результаты по причине различий в последнем параметре функции FREETEXTTABLE.

Функция *CONTAINSTABLE*

Подобно функции FREETEXTTABLE, функция CONTAINSTABLE возвращает таблицу, содержащую нулевое или большее количество строк. Функция CONTAINSTABLE используется для поиска в столбцах, содержащих текстовые данные, точные и приблизительные совпадения. Кроме этого, она также позволяет выполнять поиск определенных слов и фраз, поиск слов в заданной близости друг от друга, а также частотный поиск. (В функции CONTAINSTABLE используются такие же условия поиска, как и в предикате CONTAINS.)

ПРИМЕЧАНИЕ

Синтаксис функции CONTAINSTABLE аналогичен синтаксису функции FREETEXTTABLE. Также функция CONTAINSTABLE принимает те же самые четыре параметра, описанные ранее для функции FREETEXTTABLE. Кроме того, возвращаемая функцией CONTAINSTABLE таблица имеет два неявных столбца: key и rank, имеющих то же само значение.

В примере 28.13 показано использование функции CONTAINSTABLE.

Пример 28.13. Использование функции CONTAINSTABLE

```
USE sample;
SELECT pr,product_id, pr,product_name
   FROM product pr INNER JOIN
   CONTAINSTABLE(product, description,
                                 'Microsoft NEAR management',1) ct
   ON pr,product_id = ct.[key]
   ORDER BY ct.rank DESC;
```

Этот запрос дает следующий результат:

product_id	product_name
1	MS Application Center

Запрос в примере 28.13 соединяет две таблицы: таблицу product и таблицу, возвращаемую функцией CONTAINSTABLE. Этот пример подобен примеру 28.10, но возвращает другой результат по причине различного последнего параметра функции CONTAINSTABLE, который задает количество отображаемых строк результирующего набора, упорядоченных по значению коэффициента релевантности. В примере 28.13 этот параметр имеет значение 1, поэтому отображается только одна строка результирующего набора. Кроме этого, в предложении ORDER BY указан параметр DESC, вследствие чего выводится строка с самым последним значением коэффициента релевантности.

Функцию CONTAINSTABLE можно использовать для поиска взвешенных совпадений, как показано в примере 28.14.

Пример 28.14. Поиск взвешенных совпадений

```
SELECT pr,product_id, pr,product_name
   FROM product pr,
   CONTAINSTABLE (product, description,
        'ISABOUT (Microsoft WEIGHT(.4), management WEIGHT(.2))') ct
   WHERE pr.product_id = ct.[key]
   ORDER BY ct.rank DESC;
```

Выполнение этого запроса дает следующий результат:

product_id	product_name
2	MS Commerce Server
1	MS Application Center

Предложение ISABOUT функции CONTAINSTABLE выполняет поиск в полнотекстовом столбце группы из одного или нескольких взвешенных слов поиска. Таким обра-

зом, запрос в примере 28.14 выполняет поиск всех слов `Microsoft` и `management` и присваивает им разный вес, используя для этого предложение `WEIGHT` с параметром, указывающим данный вес. (Значение веса должно быть в диапазоне от 0 до 1.)

Поиск и устранение проблем с полнотекстовыми данными

Проблемы с полнотекстовыми данными на основе вызывающих их причин можно разделить на две группы. Проблемы первой группы возникают в процессе полнотекстового индексирования, а второй группы — в процессе выполнения запросов. Для поиска и устранения проблем, связанных с полнотекстовыми данными, можно использовать следующие динамические административные представления (DMV):

- `sys.dm_fts_index_keywords`;
- `sys.dm_fts_index_keywords_by_document`;
- `sys.dm_fts_parser`.

ПРИМЕЧАНИЕ

Два первых динамических административных представления (DMV) поддерживают диагностику проблем, связанных с полнотекстовой индексацией, а последнее представление — диагностику проблем, возникающих в процессе выполнения запросов.

Динамическое административное представление `sys.dm_fts_index_keywords` возвращает информацию о содержимом полнотекстового индекса для указанной таблицы. Использование этого динамического представления показано в примере 28.15.

Пример 28.15. Использование динамического административного представления (DMV) `sys.dm_fts_index_keywords`

```
USE sample;
SELECT keyword, display_term, document_count
   FROM sys.dm_fts_index_keywords (db_id('sample'),
                                   object_id('product'));
```

Запрос в примере 28.15 отображает информацию о полнотекстовом индексе таблицы `product`. В столбце `keyword` представления отображается шестнадцатеричное представление ключевого слова, хранящегося в полнотекстовом индексе. Столбец `display_column` содержит ключевое слово в удобном для восприятия людьми формате, а столбец `document_count` содержит значение числа документов или строк, содержащих текущий термин.

Второе динамическое административное представление (DMV) `sys.dm_fts_index_keywords_by_documents` имеет подобную функциональность, но разбивает ключевые слова по документам. Поэтому это представление помогает выполнять диагностику

проблем в документе на более глубоком уровне. Это представление возвращает те же самые столбцы, что и первое, а также два следующих дополнительных столбца:

- `document_id` — содержит идентификатор столбца или строки, из которой был создан полнотекстовый индекс текущего ключевого слова;
- `occurrence_count` — содержит число вхождений текущего ключевого слова в документе или строке, указанной идентификатором `document_id`.

Динамическое административное представление `sys.dm_fts_parser` возвращает внутреннюю информацию в процессе поиска. Иными словами, система отображает интерпретацию стадии поиска для данного запроса. Использование представления `sys.dm_fts_parser` показано в примере 28.16.

Пример 28.16. Использование динамического административного представления (DMV) `sys.dm_fts_parser`

```
USE sample;
SELECT keyword, occurrence, special_term, display_term
   FROM sys.dm_fts_parser (' "The Microsoft business analysis" ',
                                                    1033, 0, 0);
```

Выполнение этого запроса возвращает следующий результат:

Keyword	occurrence	special_term	display_term
0x007400680065	1	Noise Word	the
0x006D006900630072006F0073006F00660074	2	Exact Match	Microsoft
0x006200750073006900 6E006500730073	3	Exact Match	business
0x0061006E0061006C0079007300690073	4	Exact Match	analysis

В столбце `keyword` представления отображается шестнадцатеричное представление ключевого слова, хранящегося в полнотекстовом индексе. В столбце `occurrence` указывается порядок вхождения каждого термина в результат разбора текста на слова. В столбце `special_term` отображается информация о характеристиках термина, отображаемого в столбце `display_term`, которые используются делителем.

Новые возможности SQL Server 2012 по полнотекстовому поиску

В сервере SQL Server 2012 вводится два усовершенствования полнотекстового поиска.

- Настройка поиска с учетом близости.
- Поиск в расширенных свойствах.

Эти усовершенствования описываются далее в следующих разделах.

Настройка поиска с учетом близости

Как показано в примере 28.10, компонент FTS использует предложения NEAR для нахождения лексем, которые расположены сравнительно близко друг от друга в документе. В предшествующих версиях SQL Server для этого предложения нельзя было задавать расстояние между лексемами.

В SQL Server 2012 поиск с учетом близости лексем с использованием расширенного предложения NEAR для предиката CONTAINS или строчной функции CONTAINSTABLE можно настроить. Эта факультативная возможность позволяет указать максимальное количество нецелевых слов между первым и последним словом поиска, чтобы считать их совпадением. Настройка поиска с учетом близости посредством предложения NEAR также позволяет указать, что слова и фразы считаются совпадением только в том случае, если они расположены в тексте в указанном в условии поиска порядке.

В примере 28.17 показано выполнение настроенного поиска с учетом близости с помощью функции CONTAINSTABLE.

Пример 28.17. Выполнение настроенного поиска с учетом близости

```
USE sample;
SELECT pr.product_id, pr.product_name
   FROM product pr INNER JOIN
   CONTAINSTABLE (product, description,
              '(user NEAR analysis)') AS ct
   ON pr.product_id = ct.[key]
   ORDER BY ct.rank DESC;

SELECT pr, product_id, pr, product_name
   FROM product pr INNER JOIN
   CONTAINSTABLE (product, description,
           ' NEAR((user, analysis), 3)') AS ct
   ON pr, product_id = ct.[key]
   ORDER BY ct.rank DESC;
```

В примере 28.17 в первой инструкции SELECT демонстрируются функциональные возможности предложения NEAR предыдущих версий SQL Server. В этом предложении задается выборка документов, в которых встречаются оба слова поиска, но не указывается какое-либо расстояние между этими словами. (Предложение вычисляет расстояние между словами, но только для того, чтобы определить ранг каждого документа в результирующем наборе.)

Во второй инструкции SELECT приведены расширенные функциональные возможности предложения NEAR. Теперь в этом предложении используется параметр (значение которого в данном примере равно 3) для указания расстояния между словами. (Расстояние в 3 слова означает, что между целевыми словами в тексте может быть не более чем 3 нецелевых слов.) Поэтому эти две инструкции SELECT возвра-

щают разные результаты: первая отображает информацию о втором документе, а вторая возвращает пустой результирующий набор. Второй документ содержит оба слова, но расстояние между ними больше, чем три нецелевых слова.

Свойства расширенного поиска

Сервер SQL Server 2012 позволяет выполнять поиск текстовых данных не только по содержимому, но также и по расширенным свойствам. Поиск по расширенным свойствам возможен только в случае наличия конкретного фильтра IFilter. (Фильтры IFilter подробно описываются в *начале этой главы*.)

Первым шагом поиска по расширенным свойствам будет создание списка соответствующих свойств. Для создания такого списка используется инструкция `CREATE SEARCH PROPERTY LIST`. В примере 28.18 показано создание списка свойств для столбца `description` таблицы `product`.

Пример 28.18. Создание списка свойств

```
USE sample;
CREATE SEARCH PROPERTY LIST Sample_Properties;
GO
ALTER SEARCH PROPERTY LIST Sample_Properties
      ADD 'MS Tools'
   WITH (PROPERTY_SET_GUID = 'F29F85EO-4FF9-1068-AB91-08002B27B3D9',
                                                PROPERTY_INT_ID = 1);
```

Первая инструкция в примере 28.18 создает пустой список свойств поиска, называющийся `Sample_Properties`. Этот список свойств поиска используется для указания одного или больше расширенных свойств, которые требуется включить в полнотекстовый индекс. Для добавления расширенного свойства поиска (в данном примере это `MS Tools`) в список свойств используется предложение `ADD`. (Добавление свойства означает, что это свойство регистрируется для списка свойств поиска.)

В параметре `PROPERTY_SET_GUID` указывается глобально уникальный идентификатор GUID (Globally Unique Identifier) набора свойств, к которому принадлежит данное свойство. В параметре `PROPERTY_INT_ID` указывается целое число, которое однозначно идентифицирует свойство в его наборе свойств.

Созданный список свойств поиска можно назначить существующему полнотекстовому индексу, используя инструкцию `ALTER FULLTEXT INDEX`, как это показано в примере 28.19.

Пример 28.19. Присвоение списка свойств поиска полнотекстовому индексу

```
USE sample;
ALTER FULLTEXT INDEX ON dbo.product
   SET SEARCH PROPERTY LIST Sample_Properties
   WITH NO POPULATION;
```

```
GO
   ALTER FULLTEXT INDEX ON dbo.product
   START FULL POPULATION;
```

ПРИМЕЧАНИЕ

При исполнении запроса в примере 28.19 система выдаст предупреждающее сообщение. Не обращайте на это сообщение внимания. Это предупреждение связано с первой инструкцией примера. Вторая инструкция приводит полнотекстовый индекс в согласованное состояние.

В примере 28.19 первая инструкция добавляет список `Sample_Properties` в существующий полнотекстовый индекс. (Индекс при этом не заполняется.) Вторая инструкция заполняет индекс, используя полный режим заполнения. (Полный и инкрементальный режимы заполнения индекса рассмотрены *ранее в этой главе.*)

Просмотреть имена существующих списков свойств поиска можно с помощью представления каталога `sys.registered_search_property_lists`, как это показано в примере 28.20.

Пример 28.20. Просмотр существующих списков свойств поиска

```
USE sample;
SELECT name FROM
   sys.registered_search_property_lists;
```

Резюме

Компонент полнотекстового поиска Full-Test Search (FTS) позволяет выполнять поиск по всему тексту документа, сохраненному в столбце текстового типа реляционной таблицы. Перед началом поискового запроса вы должны создать полнотекстовый индекс для столбца(ов) и затем заполнить его.

Компонент FTS поддерживает следующие типы запросов:

- поиск слов или фраз;
- поиск слов, находящихся в тексте вблизи друг от друга;
- поиск различных форм глаголов, существительных и прилагательных;
- поиск слова, которому присвоен больший вес, чем другому слову.

Компонент FTS поддерживает два предиката: `FREETEXT` и `CONTAINS`. Предикат `CONTAINS` предоставляет существенно больший уровень функциональности в плане полнотекстового поиска, чем предикат `FREETEXT`. Кроме этого, поддерживаются две строчные функции: `FREETEXTTABLE` и `CONTAINSTABLE`. Возможности функции `FREETEXTTABLE` соответствуют возможностям предиката `FREETEXT`, а функции `CONTAINSTABLE` — предиката `CONTAINS`.

Предметный указатель

A

ACID 372
Address space 559
Aggregate tables 601
Aggregation 601
Alias data type 130
Analysis Services 601
Anonymous subscription 502
Article 501
Authentication mode 42
Availability group 468
Availability replica 468

B

Backup 435
Backup set 451
Batch 237
BIDS 665
Books Online 45
Bulk-logged recovery model 460
Business Intelligence (BI) 591, 593
Business Intelligence Development Studio (BIDS) 607, 609, 671
◊ запуск 671

C

Candidate key 120
Category 429
CDC 396
Cell 609
Certification Authority (CA) 329
Check constraint 122
CLR 251, 405
Collocation 697
Column statistics 523
Column store 703
Columnstore Index 689, 703
Common Table Expression (CTE) 208
Concurrency 369
Concurrency update 361
Condition 429
Configure Distribution Wizard 511
Contained databases 138
Covering index 287
Cube 608
Cube Wizard 607

D

DAC-подключение 424, 506
Data Analysis Expressions (DAX) 624
Data mart 11, 591, 595
Data mining 605
Data Source Wizard 607, 611
Data store 591
Data warehouse 11, 591
Database Engine Tuning Advisor (DTA) 299, 571, 572, 574
Database Mail 479
DBCC 427
DDL 26, 111, 129
Deadlock 384
Demand paging 559
Dimension 608
Dimension table 597
Dimensional model 597
Distribution Agent 503
DML 26, 129, 219
Domain 130
DPAPI 326
DTD 711, 717
Dynamic Help 75
Dynamic Management Functions (DMF) 272
Dynamic Management Views (DMV) 272, 537

E

Entity-relationship (ER) 30, 596
ETL 595
Exception 242
Exclusive lock 370, 378
Extensible Key Management (EKM) 331

F

Facet 429
Fact table 597
FLWOR 736
Forced parameterization 547
Foreign key 72, 123
Framing 640
Frequency search 774
Full recovery model 460
Full-Test Search (FTS) 760
Full-Text Indexing Wizard 770

G

Geometry 741
GML 744
GPS 740

H

HADR 435
Hard page fault 560
Hierarchy 608
HTML 717
HTTP 716
Hybrid online analytical processing (HOLAP) 603

I

IAM (index allocation map) 558
IFilter 760, 761
Index entry 282
Instance document 726
Integrity constraints 119

L

Level 609
Lock escalation 381
Log Reader 503

M

Maintenance Plan Wizard 435, 470
Managed targets 429
Matrix 676
Measure 598, 609
Measure group 609
Member 608
Merge replication 507
Microsoft Cluster Service 467
MSSQLSERVICE 477
Multidimensional Expressions (MDX) 13, 630
Multidimensional online analytical processing (MOLAP) 603

N

Namespace 130
New Publication Wizard 513
New Subscription Wizard 514
Noise words 761
NTFS 116

O

Object Explorer 56, 58
OGC 741
Online analytical processing (OLAP) 603
Online Release Notes 45
Online transaction processing (OLTP) 229, 591
◊ OLTP-система 592

P

Page chain 284
Page fault 559
Pagefile 556
Parent table 124
Parsing 518
Partition 609
Partition function 691
Partition key 690
Partition schema 691
Partitioning 638
Performance Data Collector 579
Performance Monitor 561
Plan cache 526
Plan guides 548
Plan handle 539
Policy 429
Policy-based management 428

Population 764
PowerPivot for Excel 624
Proximity operation 763
Publication 501
Pull subscription 502
Push subscription 502

Q

Query by Example (QBE) 676
Query compilation 518
Query Designer 666
Query Editor 56
Query execution 519
Query optimization 518
Query optimizer 517

R

RAID 464
RAID 0 465
RAID 1 465
RAID 5 466
Range partitioning 12, 691
Read Ahead Manager 558
Recovery 435
Referenced table 124
Referencing table 124
Referential integrity 124
Registered Servers 56, 58
Relational Online Analytical Processing (ROLAP) 603, 689
Relevance score 764
Report Builder 669, 671
Report Definition Language (RDL) 667
Report Manager 669
Reporting Services Configuration Manager (RSCM) 670
Resource Governor 582
Role switching 469
Routine 237
Row store 703
RSA 326

S

Savepoint 374
Schema 129
Schema-valid 726
Selectivity 520
SGML 717
Shared lock 370, 378
Snapshot Agent 503
Soft page fault 560
Solution Explorer 56
SQL 19, 26
SQL Intellisense 76
SQL Server:
◊ версии 37
◊ отладка 77
◊ планирование установки 42
◊ установка 46
SQL Server Analysis Services (SSAS) 607
SQL Server Integration Services (SSIS) 595, 607
SQL Server Profiler 571
SQL Server Reporting Services (SSRS) 607, 665, 667
SQL/MM 753
SRID 743
Standby server 463
Stop words 761
Stored procedure 129
Synonym 129
System availability 435

T

Table Import Wizard 627
Tabular 676
Target set 429
Token 760
Transaction log 376
◊ after image 376
◊ before image 376
◊ undo activity 376
Transact-SQL 26
Trigger 129
Two-phase commit 498

U

Uncontained objects 273
Update lock 378
URI 715
User Defined Function (UDF) 237, 256

V

View 128

W

Weighted search 774
Wildcards 763
Witness server 466

Well-Known Binary (WKB) 743
Well-Known Text (WKT) 743
Word breakers 761

X

XML 711
XML execution plan 530
XML Schema 711, 719, 726
XML-документ:
◊ действительный согласно схеме 726
XPath 736
XQuery 711, 736
XSL 719
XSN 389

А

Абстрактный класс 744
Автономные базы данных 138
Авторизация 23, 324
Агент:
◊ Distribution Agent 503
◊ Log Reader 503
◊ Snapshot Agent 503
Агрегатная таблица 601
Агрегатная функция 96, 165
◊ AVG 168
◊ COUNT 169
◊ COUNT_BIG 169, 170
◊ MAX 166
◊ MIN 166
◊ SUM 168
◊ обычная 165
◊ статистическая 170
Агрегирование 601, 617
◊ уровень 602
Адресное пространство 559
Алгоритм шифрования RSA 326
Аналитические системы 593
Аргумент поиска 519
Аспект 429
Асимметричные ключи 328
Атрибут 30
◊ XML 714
◊ атомарный 30
◊ многозначный 31
◊ однозначный 30
◊ составной 31
◊ элемента 714
Аутентификация 22, 323, 325
◊ SQL Server 325
◊ Windows 325

Б

База данных 20
◊ distribution 502
◊ master 414
◊ model 414
◊ tempdb 414
◊ автономная 138
◊ безопасность 22
◊ владелец 112
◊ зеркальная 466
◊ изменение 132
◊ имя 112
◊ оперативная 592
◊ отсоединение 116
◊ по умолчанию 333
◊ присоединение 116
◊ создание 112
◊ создание моментального снимка 115
Бизнес-аналитика 591, 593
◊ свойства 593
Блок инструкций 238
Блокировка 377
◊ гранулярность 380
◊ длительность 377
◊ монопольная 370, 378
◊ настройка 382
◊ немонопольная 370
◊ обновления 378
◊ подсказки 382

Блокировка *(прод.)*:
◊ порог укрупнения 381
◊ разделяемая 378
◊ режимы 378
◊ с намерением 379
◊ с обеспечением разделяемого доступа 370
◊ укрупнение 381

В

Ввод/вывод:
◊ логический 557
◊ физический 557
Версии SQL Server 37
Взаимоблокировка 384
Владелец базы данных 112
Вложенный запрос 185
Внешнее соединение 198
Внешний запрос 185
Внешний ключ 123
Восстановление 435
Временная таблица 189
Временные таблицы 117
Время реакции 551
Вторичное размещение 469
Выборка 150
Выражение:
◊ CASE 183
◊ селективность 520
Вычисляемый столбец 297
◊ детерминированный 298

Г

Геодезические модели 740
Геометрический объект 741
Гистограмма 522
Главный сервисный ключ:
◊ базы данных 326
◊ системы 326
Глобальные переменные 107
Группа:
◊ мер 609
◊ обеспечения доступности 468
Групповые символы 763

Д

Двухфазная фиксация 498
Действительный документ 718
Декартово произведение 197
Делители текста на слова 761
Денормализация таблиц 553
Дерево B+ 282
Дескриптор плана 539
Диакритический знак 763
Динамические административные представления (ДАП) 272, 537
Динамические административные функции (ДАФ) 272
Доверительное соединение 325, 423
Документ экземпляра 726
Домен 130
Доступ по индексу 521
Доступность системы 435
Доступные поля 677
Дочерние элементы элемента 713
Древовидная структура документа 713

Е

Естественное соединение 192

Ж

Журналы транзакций 376
◊ исходные образы записей 376
◊ операция отмены записей 376
◊ операция повторного выполнения действий 376
◊ порядковый номер записи 376
◊ преобразованные образы записей 376

З

Заметки о версии в сети 45
Заполнение 764
◊ инкрементальное 764
◊ полное 764
Запрос покрывающий 287
Запуск и останов серверов 63
Зарегистрированные серверы 56, 58
Зарезервированные ключевые слова 88
Защищаемые объекты 324, 350
Зеркалирование 465
Зеркальное отображение базы данных 466
Значение NULL 107

И

Иерархия 608
Избыточность данных 27

Извлечение информации из данных 604, 605
Издатель 500
Измерение 608
Индекс 281
◊ изменение 292
◊ кластеризованный 283
◊ колоночный 689
◊ некластеризованный 284
◊ переименование 294
◊ покрывающий 287, 296
◊ простой 287
◊ создание 286
◊ составной 287
◊ страницы 282
◊ удаление 294
◊ фильтруемый 319
◊ фрагментация 290
◊ элемент 282
Индексированные представления 286, 301
Инструкция:
◊ ALTER APPLICATION ROLE 347
◊ ALTER ASYMMETRIC KEY 328
◊ ALTER DATABASE 132, 133, 693
◊ ALTER FULLTEXT CATALOG 768
◊ ALTER FUNCTION 395
◊ ALTER INDEX 292
◊ ALTER PROCEDURE 251
◊ ALTER ROLE 349
◊ ALTER SERVER ROLE 343
◊ ALTER TABLE 140, 381
◊ ALTER USER 340
◊ ALTER VIEW 305
◊ BACKUP DATABASE 442
◊ BACKUP LOG 444
◊ BEGIN DISTRIBUTED TRANSACTION 373
◊ BEGIN TRANSACTION 373
◊ COMMIT WORK 373
◊ CREATE APPLICATION ROLE 346
◊ CREATE ASYMMETRIC KEY 328
◊ CREATE CERTIFICATE 329
◊ CREATE DATABASE 112
◊ CREATE FULLTEXT CATALOG 767
◊ CREATE FULLTEXT INDEX 768
◊ CREATE INDEX 286
◊ CREATE LOGIN 333
◊ CREATE PARTITION FUNCTION 694
◊ CREATE PARTITION SCHEME 696
◊ CREATE ROLE 348
◊ CREATE SEARCH PROPERTY LIST 782
◊ CREATE SERVER ROLE 343
◊ CREATE SYNONYM 129
◊ CREATE TABLE 116
◊ CREATE TRIGGER 129, 394
◊ CREATE TYPE 131
◊ CREATE USER 339
◊ CREATE VIEW 302
◊ CROSS APPLY 261
◊ DBCC FREEPROCCACHE 527
◊ DECLARE 241
◊ DELETE 313
◊ DENY 355
◊ DROP ASYMMETRIC KEY 328
◊ DROP FULLTEXT CATALOG 768
◊ DROP INDEX 290, 294
◊ DROP ROLE 349
◊ DROP SERVER ROLE 343
◊ DROP TRIGGER 395
◊ GO 14
◊ GOTO 242
◊ GRANT 350
◊ IF 238
◊ INSERT 219, 308
  ▫ вставка нескольких строк 222
  ▫ вставка одной строки 220
◊ MERGE 229
◊ OUTER APPLY 261
◊ RAISEERROR 242
◊ RESTORE DATABASE 452
◊ RESTORE FILELISTONLY 451
◊ RESTORE HEADERONLY 451
◊ RESTORE LABELONLY 451
◊ RESTORE VERIFYONLY 452
◊ RETURN 242
◊ REVOKE 356
◊ ROOLBACK WORK 374
◊ SAVE TRANSATION 374
◊ SELECT 149
◊ THROW 243
◊ TRUNCATE TABLE 228
◊ TRY 242
◊ UPDATE 224, 311
◊ USE 14
◊ WAITFOR 242
◊ WHILE 239
◊ атомарная 337
Информационная схема 274
Информационные системы 593
Исключение 242

Источники данных:
◊ встроенные 674
◊ использование 673
◊ разделяемые 674

К

Кадрирование 640
Карта распределения индекса 558
Категория 429
Киоск данных 11, 591, 593, 595
Кластеризованная таблица 284
Кластеризованный индекс 283
Клиентские компоненты 20
Ключ:
◊ внешний 72, 123
◊ секционирования 690
Колоночный индекс 689
Команда:
◊ DBCC CHECKALLOC 428
◊ DBCC CHECKCATALOG 428
◊ DBCC CHECKDB 428
◊ DBCC CHECKTABLE 428
Консолидация данных 594
Контроль по четности 466
Контрольная точка 450
Координатор 498
Корневой каталог 41
Корневой элемент 713
Коэффициент релевантности 764
Куб 600, 608
◊ обработка 619
◊ просмотр 620
◊ создание 616
◊ физическое хранение 603
Курсор 251
Куча 285, 521
Кэш планов 526
Кэширование отчета 685

Л

Лексема 760
Листья 713
Литерал 86
Логином 325
Логическая независимость данных 21
Логические операторы 153
Логический ввод/вывод 557
Логическое чтение 557
Локальные переменные 240

М

Мастер:
◊ Configure Distribution Wizard 511
◊ New Publication Wizard 513
◊ New Subscription Wizard 514
◊ плана обслуживания 470
Мера 598, 609
Многомерные сводки 643
Многомерные системы управления базами данных (МСУБД) 601
Модель:
◊ восстановления:
▫ полная 460
▫ простая 461
▫ с неполным протоколированием 460
◊ данных 23
◊ репликации 508
◊ "сущность — отношение" 30, 596
Моментальный снимок базы данных 115
◊ создание 115
Мониторинг:
◊ дисковой системы 567
◊ памяти 564
◊ производительности 560
◊ сетевого интерфейса 568
◊ центрального процессора 562

Н

Набор:
◊ данных 673
▫ использование 674
▫ разделяемый 674
◊ значений 28
◊ резервного копирования 451
Настройка производительности 551
Незапротоколированные операции 449
Некластеризованные индексы 284
Неограниченные объекты 273
Нерекурсивные запросы 208
Нормализация данных 27

О

Обобщенное табличное выражение (ОТВ) 208
Обозреватель:
◊ объектов 56, 58
◊ решений 56, 77
Обратное проектирование 71
Общеязыковая среда выполнения 405

Общие интерфейсы 269
Объекты баз данных:
◊ логические 111
◊ создание 111
◊ физические 111
Ограничение:
◊ проверочное 122
◊ целостности 119
  ▫ на уровне столбца 122
Одновременный конкурентный доступ 369
Оперативная аналитическая обработка 603
◊ гибридная 603
◊ многомерная 603
◊ реляционная 603
Оперативная обработка транзакций 591
Оператор 182
◊ BETWEEN 158
◊ CREATE SEQUENCE 176
◊ CUBE 643
◊ IN 157
◊ JOIN 190
◊ LIKE 161
◊ ROLLUP 645
◊ UNION 179
◊ обмена 422
◊ соединения 190
Операция определения близости 763
Опорный запрос 210
Определяемая пользователем функция (ОПФ) 237, 256
◊ скалярная 257
◊ табличная 257
Оптимизатор запросов 21, 282, 517, 555
Отказоустойчивая кластеризация 467
Отладка SQL Server 77
Отношение 31
Отображаемое поле 678
Отслеживание изменений 324, 361
Отчет 604
◊ матричный 676
◊ параметризованный 681
◊ табличный 676
Ошибка:
◊ программной страницы 560
◊ страницы 559
◊ страницы диска 560

П

Пакет 77, 237
Параллельные обновления 361
Параметризованный отчет 681
Параметры процедуры 246
Первичное размещение 469
Первичный ключ 26, 121
Переключение ролей 469
Переменные локальные 240
Перехват изменения данных 396
План:
◊ выполнения XML 530
◊ выполнения запроса 77, 517
◊ исполнения запроса 21
Планирование установки 42
Подзапрос 185
◊ с многоуровневым вложением 187
◊ преимущества 205
Подкачка страниц по требованию 559
Подключение к серверу 56, 62
Подписка:
◊ анонимная 502
◊ по запросу 502
◊ принудительная 502
Подписчик 500
Подпрограмма 237
Подсказки блокировок 382
Подстановочные символы 161
Покрывающий запрос 287
Покрывающий индекс 287, 296
Политика 429
◊ применение управления 429
Полнотекстовые запросы 773
Полусоединение 202
Пользовательские интерфейсы 20
Постолбцовое хранение 703
Построчное хранение 703
Потенциальный ключ 26, 120
Предикат:
◊ CONTAINS 774
◊ FREETEXT 773
Предложение:
◊ GROUP BY 164
◊ HAVING 171
◊ ORDER BY 172
◊ WHERE 151
Представления 128, 301
◊ изменение 305
◊ индексированные 301, 314
  ▫ создание 314
◊ каталога 269
◊ материализованные 314
◊ редактирование 307
◊ создание 302

Прикладная программа баз данных 20
Принудительная параметризация 547
Принципалы 271, 335
Проверочное ограничение 122
Проект 77
Проектирование базы данных 27
Проекция 150
Прозрачное шифрование данных 331
Производная таблица 206
Пропускная способность 551
Простой индекс 287
Пространственная модель 597
Пространство имен 130
◊ по умолчанию 715
Псевдоним:
◊ таблицы 150
◊ типа данных 130
Публикация 501

Р

Размерная модель 597
Разрешения 324
Распространитель 500
Расслоение дисков 465
Расширенное управление ключами 331
Регистрация серверов 61
Редактор запросов 56, 73
Режим обеспечения доступности 469
Режимы аутентификации 42
Резервное копирование 435
◊ выполнение 440
◊ динамическое 437
◊ журнала транзакций 438
◊ методы 437
◊ полное 437
◊ разностное 438
◊ статическое 437
◊ файлов или файловых групп 440
Резервный сервер 463
Результирующий набор 149
Рекурсивные запросы 210
Рекурсивный член 210
Релевантность, коэффициент 764
Реляционная оперативная аналитическая обработка 689
Реплика обеспечения доступности 468
◊ первичная 468
Репликация:
◊ данных 497
◊ модели 508
◊ моментальных снимков 506
◊ слиянием 507
◊ транзакций 504
▫ одноранговая 505
◊ управление репликацией 511
Роли приложений 346
Роль базы данных 341

С

Самосоединение 201
Свойство IDENTITY 174
Связанный подзапрос 203
Секции 609
Секционирование 638
◊ данных 690
◊ по диапазонам 12, 691
Селективность 295
Сервер:
◊ баз данных 20
◊ зеркальный 466
◊ подписки 500
◊ публикаций 500
◊ распространения 500
◊ следящий 466
Сертификаты 329
Символьные данные 713
Симметричные ключи 327
Синоним 129
Система:
◊ баз данных 19
◊ оперативной обработки транзакций 592
Системная хранимая процедура 276
◊ sp_fulltext_dtabase 766
Системные базовые таблицы 267
Системные ресурсы 555
Системные функции 103, 277
Скалярные операторы 106
Скалярные функции 97
Сканирование таблицы 282
Склад данных 591
Совместное размещение 697
Соединение:
◊ преимущества 206
◊ типа "звезда" 202
Создание:
◊ индексированного представления 314
◊ индексов 286
◊ новой группы серверов 62
◊ полнотекстового индекса 768
◊ полнотекстового каталога 767

Составной индекс 287
Специализированные интерфейсы 275
Список:
◊ выбора 150
◊ стоп-слов 761
Ссылочная целостность 124
◊ проблемы 125
Статистические данные 517
◊ индекса 522
◊ столбца 523
Статья 501
Столбцы соединения 191, 192
Стоп-слова 761
Страница 416
◊ IAM 558
◊ заголовок 417
◊ индексов 282
◊ переполнения строк 420
◊ пространство для данных 418
◊ свойства 417
◊ таблица смещения строк 418
Строковые функции 100
Структуры планов 548
СУБД 22
Сущность 30
Сфероид 740
Схема 129, 335
◊ секционирования 691
◊ типа "звезда" 597
◊ типа "снежинка" 599

Т

Таблица:
◊ агрегатная 601
◊ временная 117
◊ дочерняя 124
◊ изменение 140
◊ измерений 597
◊ истинности 108
◊ родительская 124
◊ ссылающаяся 124
◊ ссылочная 124
◊ фактов 597
Табличные выражения 206
Тета-соединение 201
Транзакция 371
◊ неподтвержденная (незафиксированная) 450
◊ неявная 371
◊ подтвержденная (зафиксированная) 450
◊ распределенная 373, 498
◊ свойства 372
◊ точка сохранения 374
◊ уровень изоляции 385
◊ явная 371
Тривиальная оптимизация плана 519
Триггер 129, 393
◊ DDL:
▫ применение 402
▫ уровня базы 403
▫ уровня сервера 404
◊ DML:
▫ изменение структуры 395
▫ применение 396
▫ создание 394
◊ среда CLR 405

У

Укрупнение:
◊ блокировок 381
◊ множественными серверами 62
◊ на основе политик 428
◊ параллелизмом 22
Управляемые объекты 429
Упреждающее чтение 558
Уровни 609
Условие 429
◊ точки останова 79
Установка SQL Server 46

Ф

Файл подкачки 556, 559
Фантомы 386
Физическая независимость данных 21
Физический ввод/вывод 557
Физическое чтение 557
Фильтруемые индексы 319
Функции:
◊ даты 100
◊ метаданных 105
◊ свойств 277
Функциональная зависимость 27
◊ многозначная 28
◊ тривиальная 28
Функция:
◊ CONTAINSTABLE 777
◊ DecryptByAsymKey 328
◊ EncryptByAsymKey 328
◊ EXISTS 204
◊ FREETEXTTABLE 776

Функция *(прод.)*:
◊ objectproperty 316
◊ секционирования 691

Х

Хеш-секционирование 691
Хранилище данных 11, 40, 593, 594
◊ обращение к данным 604
◊ проектирование 596
Хранимая процедура 129, 245
◊ создание 246

Ц

Целостность:
◊ данных 21
◊ ограничение 119
◊ ссылочная 124
Центр сертификации 329
Цепочка страниц 284

Ч

Частотный поиск 774
Число попаданий 79
Числовые функции 98
Член измерения 600, 608
Чтение:
◊ логическое 557
◊ физическое 557

Ш

Шифрование 326
◊ данных 324
◊ на уровне столбцов 331

Э

Экземпляр 41, 62
◊ именованный 41
Экстент 381, 416
Электронная документация 45
Элемент индекса 282
Элементы документа 713
Этапы обработки запроса 518
◊ выполнение запроса 519
◊ компиляция запроса 518
◊ оптизация запроса 518
◊ синтаксический разбор 518

Я

Язык:
◊ DDL 26
◊ DML 26, 219
◊ SQL 26
◊ Transact-SQL 26
Ячейки 609