Михаил Фленов

bhv®

Transact-SQL

- Подробное описание языка
- Большое количество примеров
- Готовые решения типичных задач
- SQL для обслуживания 1С:Предприятия

Наиболее полное руководство

+

В ПОДЛИННИКЕ®

Михаил Фленов

Transact-SQL

Санкт-Петербург

«БХВ-Петербург»

2006

УДК 681.3.068
ББК 32.973.26-018.1
Ф69

Фленов М. Е.

Ф69 Transact-SQL. — СПб.: БХВ-Петербург, 2006. — 576 с.: ил.

ISBN 5-94157-790-7

Подробно рассмотрено использование языка Transact-SQL для администрирования и манипуляции данными СУБД Microsoft SQL Server. Материал сопровождается большим количеством практических примеров, написанных автором. Уделено внимание вопросам применения Transact-SQL при совместном использовании 1С и Microsoft SQL Server. На прилагаемом к книге компакт-диске размещены примеры запросов, тестовая база данных, а также дополнительная документация и статьи автора, посвященные базам данных.

Для программистов и администраторов СУБД

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Ирина Иноземцева*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Наталия Першакова*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 21.02.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-790-7

© Фленов М. Е., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Оглавление

Предисловие **1**

Благодарности 2
Для кого эта книга 3
Введение в SQL 4
Работа с запросами 7
Именование 7
CyD SQL Factory 9
Query Analyzer 12

Глава 1. Управление базой данных **17**

1.1. Создание и удаление базы данных 18
1.1.1. Файловые группы 28
1.1.2. Подключение базы данных 33
1.1.3. Сопоставление 34
1.2. Создание таблиц 35
1.2.1. Оператор *CREATE TABLE* 39
1.2.2. Автоматическое увеличение 43
1.2.3. Значения по умолчанию 47
1.2.4. Ограничения 49
1.2.5. Первичный ключ 58
1.2.6. Внешний ключ 60
1.2.7. Индексы 69
1.2.8. Опции индексов 80
1.2.9. Вычисляемые поля 82
1.2.10. Создание временных таблиц 85
1.2.11. GUID-поля 87
1.3. Редактирование параметров базы данных 88
1.3.1. Изменение размера файла 89
1.3.2. Добавление и удаление файла 93
1.3.3. Добавление и удаление файловых групп 94
1.3.4. Переименование базы данных 95
1.3.5. Изменение свойств базы данных 95

1.4. Редактирование таблиц 98
1.4.1. Добавление новых полей 100
1.4.2. Удаление полей 101
1.4.3. Изменение ограничений 101
1.4.4. Изменение поля 103
1.5. Обеспечение целостности данных 104
1.5.1. Ограничение *DEFAULT* 107
1.5.2. Ограничение *CHECK* 108
1.5.3. Ключи 109
1.5.4. Уникальность 110
1.5.5. Отключение ограничений 110
1.5.6. Правила и объекты значений по умолчанию 111
1.6. Именование 114
1.7. Резюме 115

Глава 2. Работа с данными 121

2.1. Оператор *SELECT* 122
2.2. Выборка данных 124
2.2.1. Полный путь 125
2.2.2. Ограничение вывода строк 127
2.2.3. Псевдонимы полей 128
2.3. Ограничение выборки 129
2.4. Булевы операторы 133
2.5. Улучшенный поиск 136
2.6. Вставка в таблицу 139
2.7. Шаблоны строк 140
2.8. Работа с несколькими таблицами 142
2.9. Объединение в стиле Microsoft 149
2.10. Простейшие расчеты 151
2.11. Сортировка 155
2.12. Группировка 156
2.13. Объединение запросов 160
2.14. Подзапросы 162
2.15. Операторы работы с подзапросами 169
2.15.1. Оператор *EXISTS* 169
2.15.2. Операторы *ANY*, *SOME* и *ALL* 170
2.16. Добавление записей 172
2.17. Изменение данных 178
2.18. Удаление данных 183
2.19. Транзакции 187
2.20. Переменные 196
2.21. Конвертирование типов 200
2.22. Работа с датами и временем 203
2.22.1. Преобразование дат 203

2.22.2. Функции для работы с датами ... 205
2.22.3. Замечания по работе с датами ... 209
2.23. Ход выполнения запроса ... 210
2.23.1. Условный оператор *IF* ... 210
2.23.2. Условный оператор *CASE* ... 214
2.23.3. Оператор цикла *WHILE* ... 216
2.23.4. Прерывание работы сценария ... 218
2.23.5. Подмена ... 219
2.23.6. Ожидание ... 220
2.24. Работа с GUID-полями ... 221
2.25. Функции работы со строками ... 227
2.25.1. Функция *SUBSTRING* ... 227
2.25.2. Функция *LEFT* ... 228
2.25.3. Функция *LEN* ... 229
2.25.4. Функция *LOWER* ... 229
2.25.5. Функция *UPPER* ... 229
2.25.6. Функции *LTRIM* и *RTRIM* ... 230
2.25.7. Функция *PATINDEX* ... 231
2.25.8. Функция *REPLACE* ... 231
2.25.9. Функция *REPLICATE* ... 232
2.25.10. Функция *REVERSE* ... 233
2.25.11. Функция *SPACE* ... 234
2.25.12. Функция *STR* ... 234
2.25.13. Функция *STUFF* ... 235
2.26. Математические функции ... 236
2.26.1. Знаки ... 236
2.26.2. Округление ... 237
2.26.3. Сложная математика ... 238
2.26.4. Случайное значение ... 239
2.26.5. Тригонометрические функции ... 239
2.26.6. Степень ... 240
2.27. Связь "многие-ко-многим" ... 241

Глава 3. Программирование на сервере ... 245

3.1. Представления ... 246
3.1.1. Создание представления ... 246
3.1.2. Редактирование представления ... 252
3.1.3. Удаление представления ... 253
3.1.4. Изменение содержимого представления ... 253
3.1.5. Удаление строк из представления ... 254
3.1.6. Опции представления ... 254
3.2. Хранимые процедуры ... 255
3.2.1. Создание хранимых процедур ... 257
3.2.2. Выполнение процедур ... 259

3.2.3. Удаление процедур ... 259
3.2.4. Использование параметров ... 260
3.2.5. Преимущества хранимых процедур ... 261
3.2.6. Практика создания и использования процедур ... 261
3.2.7. Изменение процедур ... 264
3.2.8. Использование процедур при вставке данных ... 266
3.2.9. Опции ... 266
3.3. Хранимые функции ... 267
3.3.1. Создание функции ... 268
3.3.2. Скалярные функции ... 269
3.3.3. Использование функций ... 271
3.3.4. Функция, возвращающая таблицу ... 272
3.3.5. Многооператорная функция, возвращающая таблицу ... 274
3.3.6. Опции функций ... 276
3.3.7. Изменение функций ... 277
3.3.8. Удаление функций ... 279
3.4. Триггеры ... 279
3.4.1. Создание триггера ... 280
3.4.2. Откат изменений в триггере ... 281
3.4.3. Изменение триггера ... 283
3.4.4. Удаление триггера ... 285
3.4.5. Как работают триггеры? ... 285
3.4.6. Триггер *INSTEAD OF* ... 290
3.4.7. Дополнительные сведения о триггерах ... 293
3.4.8. Практика использования триггеров ... 295
3.5. SQL Server Agent ... 298
3.5.1. Добавление задания ... 300
3.5.2. Управление операторами ... 302
3.5.3. Добавление шага ... 306
3.5.4. Запуск задания ... 312
3.5.5. Информация о задании ... 315
3.5.6. Управление заданиями ... 319
3.5.7. Управление шагами ... 320
3.5.8. Эффективное использование заданий ... 322
3.6. Планировщик заданий ... 323
3.6.1. Добавление плана выполнения ... 324
3.6.2. Обновление планировщика ... 328
3.6.3. Удаление планировщика ... 329
3.6.4. Информация о планировщике ... 329
3.7. Оповещения ... 329
3.7.1. Создание сообщения ... 330
3.7.2. Создание оповещения ... 331
3.7.3. Создание уведомления ... 336

Глава 4. Дополнительные возможности Transact-SQL 339
4.1. Свойства сервера 339
4.1.1. Ограничение выводимых строк 340
4.1.2. Управление неявными транзакциями 341
4.1.3. Управление блокировками 342
4.1.4. Управление датой 346
4.1.5. Объединение с *NULL* 347
4.1.6. Запрет на подсчет строк 348
4.1.7. Закрытие курсора 348
4.1.8. План выполнения 348
4.1.9. Соответствие ANSI 349
4.2. Информация о системе 351
4.2.1. Информация о базе данных 351
4.2.2. Имя пользователя 354
4.2.3. Имя приложения 354
4.2.4. Информация об объекте 355
4.2.5. Информация о журнале транзакций 358
4.2.6. Свойство *IDENTITY* 359
4.2.7. Информационные процедуры 360
4.2.8. Пользовательские параметры конфигурации 363
4.3. Обработка ошибок 365
4.3.1. Глобальная переменная *@@ERROR* 366
4.3.2. Генерирование сообщений 367
4.3.3. Создание собственных сообщений 369
4.3.4. Резюме 370
4.4. Поддержка XML 370
4.5. Типы данных, определенные пользователем 372
4.6. Поддержка индексов 373
4.7. Работа со статистикой 380
4.8. Управление пользователями 387
4.8.1. Управление пользователями сервера 387
4.8.2. Управление пользователями базы данных 390
4.8.3. Роли 392
4.8.4. Создание и удаление ролей 394
4.8.5. Управление ролями 394
4.9. Права доступа 396
4.9.1. Разрешение доступа 396
4.9.2. Запрещение доступа 399
4.9.3. Отмена прав доступа 401
4.9.4. Информация о правах доступа 403
4.10. Резервное копирование и восстановление 405
4.10.1. Стратегия резервного копирования 406
4.10.2. Стратегия восстановления 409

4.10.3. Резервное копирование ... 410
4.10.4. Восстановление данных ... 422
4.10.5. Замечания по резервному копированию ... 435
4.11. Уменьшение базы данных ... 437
4.12. Отключение базы данных ... 439

Глава 5. Сложные запросы ... 443

5.1. Распределенные запросы ... 443
5.1.1. Динамическое создание подключений ... 444
5.1.2. Создание связанного сервера ... 448
5.1.3. Код на связанном сервере ... 452
5.2. Оптимизация запросов ... 453
5.2.1. Работа с планом выполнения ... 454
5.2.2. Отображение профиля ... 461
5.2.3. Генерация плана выполнения ... 462
5.3. Расширенные процедуры ... 464
5.3.1. Обращение к системе ... 464
5.3.2. Информация об учетной записи ... 466
5.3.3. Список групп ... 468
5.3.4. Информация о сервере ... 468
5.3.5. Доступ к серверу ... 469
5.3.6. Доступ к журналу ... 469
5.4. Внешнее выполнение ... 471
5.5. Домашняя бухгалтерия ... 476
5.5.1. Создание тестовой базы ... 476
5.5.2. Выборка данных о затратах ... 482
5.5.3. Простые отчеты ... 483
5.5.4. Многомерные отчеты ... 486
5.6. Типы данных *TEXT* и *IMAGE* ... 490
5.6.1. Чтение больших объемов данных ... 493
5.6.2. Обновление данных ... 494
5.7. Курсоры ... 497
5.7.1. Объявление курсора ... 499
5.7.2. Открытие курсора ... 501
5.7.3. Выборка записей из курсора ... 501
5.7.4. Закрытие курсора ... 505
5.7.5. Изменение данных в курсоре ... 506
5.8. Полнотекстовый поиск ... 509
5.8.1. Включение поиска ... 511
5.8.2. Создание каталога ... 511
5.8.3. Регистрация таблиц ... 512
5.8.4. Регистрация полей ... 513
5.8.5. Информация о каталоге ... 515
5.8.6. Использование поиска ... 517

Глава 6. Transact-SQL и 1С 523

6.1. Конфигурирование 524
6.2. Обслуживание базы данных 530
6.2.1. Настройка базы данных 530
6.2.2. Резервное копирование 531
6.2.3. Восстановление данных 534
6.2.4. Задания 536
6.3. Выборка данных 541

Заключение 545

ПРИЛОЖЕНИЯ 547

Приложение 1. Типы данных в SQL Server 2000 549

Числа 549
Числа с плавающей точкой 549
Денежные типы 550
Дата и время 550
Строки 550
Бинарные данные 551
Другие типы данных 551

Приложение 2. Описание компакт-диска 553

Предметный указатель 555

Предисловие

Уже долгое время язык запросов SQL (Structured Query Language, структурированный язык запросов) является стандартом доступа к базам данных. Не имеет значения, какой язык программирования вы используете, я больше чем уверен, что доступ к данным на сервере баз данных происходит с помощью запросов SQL. Исключением могут быть только локальные таблицы типа DBF или Paradox. В них к данным можно обращаться благодаря драйверу через прямой доступ. Но и в этом случае драйвер способен поддерживать запросы, значительно расширяющие возможности работы с данными.

При работе с клиент-серверными или *n*-уровневыми системами доступ обязательно происходит именно через SQL-запросы. Более удобного и мощного средства пока не придумали. Даже там, где доступ, как вам кажется, идет напрямую, используется SQL, просто среда разработки прячет от нас запросы.

Хорошее знание баз данных и умение писать эффективные запросы позволит вам создавать действительно быстрые и полезные приложения. Помимо этого, программистам очень часто приходится выполнять какие-либо одноразовые задания, и SQL позволяет сделать все быстро и качественно.

Язык запросов был стандартизирован еще в 1992 году. За это время его возможности немного устарели, но не потеряли своей актуальности. В конце 90-х годов предпринимались попытки принять обновленный стандарт, но война между различными производителями баз данных не позволила достичь компромисса. В связи с этим язык SQL получил два вида расширений: Transact-SQL или T-SQL (поддерживается Microsoft) и PL/SQL (яркий представитель — Oracle). Каждый из этих производителей максимально придерживается стандарта SQL 1992 года, и все запросы на этом языке будут выполняться корректно. Но для предоставления пользователю новых возможностей добавлены новые команды, которые объединены под именами Transact-SQL и PL/SQL и поддерживаются на разных базах данных.

Рассмотреть абсолютно все команды и возможности этих стандартов нереально. Поэтому мы ограничимся стандартом 1992 года и расширением Transact-SQL, т. к. серверы, созданные компанией Microsoft, получили в нашей стране достаточно широкое распространение и продолжают завоевывать сердца разработчиков. Рассматривать всю спецификацию SQL тоже не имеет смысла, потому что большая ее часть относится к разработчикам серверов баз данных (какими должны быть поля, их типы, размерность и т. д.). Мы же будем рассматривать стандарт с точки зрения программистов конечных приложений, которые уже используют язык SQL, а не реализуют его в своих программах.

Если у вас возникли вопросы или пожелания по книге или языку SQL, жду ваших писем и отзывов по адресу **horrific@vr-online.ru**. Все мои последние работы основываются на вопросах и предложениях читателей, с которыми я регулярно общаюсь на форуме сайта **www.vr-online.ru**. Если у вас появятся какие-то вопросы, то милости прошу на этот форум. Я постараюсь ответить по мере возможности и жду любых комментариев по поводу этой книги. Ваши замечания позволят мне сделать эту и последующие работы лучше.

Благодарности

Работа над книгой — достаточно тяжелый труд, который отнимает очень много сил. Каждую свою работу я стремлюсь сделать лучше предыдущей, а это уже затраты не только сил, но и времени. Если бы не моя семья, то выход этой книги задержался бы, наверное, на месяц. Лето 2005 года, когда писалась эта книга, выдалось для меня достаточно тяжелым, потому что за три месяца к нам по очереди приезжали гости и родственники из разных городов. В это время дом превращался в сумасшедший, и работать приходилось по ночам. Если бы не помощь семьи, то книга точно была бы задержана.

Перед сдачей книги из-за отсутствия времени я иногда срывался на своих родных (жену, родителей). Сейчас, оглядываясь на прошлое, хочется попросить прощения за мои срывы.

Пять лет назад я изучал MS SQL Server в МГТУ им. Баумана в Москве, где курс читал Гилев Алексей Вячеславович. Хочется поблагодарить его за то, что дал мне основные знания программирования на языке Transact-SQL. До того момента я работал только с классическим ANSI SQL, а этот курс помог мне расширить знания.

Отдельное спасибо редакции "БХВ-Петербург" за то, что помогают в издании моих работ. Редакторам и корректорам за то, что выискивают мои "баги" и делают мой технический русский немного более литературным и читаемым.

Как всегда, благодарю своих друзей из редакции журнала "Геймлэнд", всех друзей по сайту **www.vr-online.ru**. С каждой новой книгой и с каждым годом количество друзей растет, и чтобы поблагодарить всех, понадобится целая книга, а ведь каждый помогает мне своими знаниями, поддержкой и просто советом. Поэтому, если я кого-то упустил, то хочу попросить прощения.

Для кого эта книга

Вполне логичный вопрос — кому будет полезна эта книга? Конечно же, администраторам и программистам. С программистами все ясно, они должны знать язык, с помощью которого можно получать данные от сервера. Но зачем это нужно администратору?

Начинающие администраторы для управления сервером SQL очень часто используют специальную утилиту Enterprise Manager, которая предоставляет визуальный интерфейс и удобство в администрировании сервером. Визуальность — это хорошо, но сценарии лучше. Я сам в этом убедился, когда нужно было тиражировать схожие настройки базы данных на несколько серверов. Сначала я копировал базу с помощью резервного копирования и восстановления на новый сервер, а затем чистил новую базу, убирая ненужные данные. Это долгий и не очень удобный процесс.

Чтобы ускорить тиражирование, я написал один сценарий, который последовательно выполнял все необходимые действия — создание базы данных, процедур, функций и индексов. Этот сценарий выполняется намного быстрее, потому что не надо копировать избыточные данные и чистить таблицы. После этого я сохраняю на диске все сценарии создания базы данных и изменения ее настроек. Это позволяет быстро создать новую базу данных.

Язык SQL необходимо знать и для тестирования производительности сервера. Оптимизация работы сервера входит в обязанности администратора, а значит, он должен уметь выполнять запросы, анализировать скорость их работы и повышать их производительность. Конечно же, оптимизировать код сценария должен программист, но скорость можно повысить и с помощью оптимизации базы данных, и это должен делать администратор. Например, если администратор увидит с помощью программы мониторинга сервера, что какой-то запрос выполняется достаточно часто, то он должен проанализировать его текст и выяснить, какие поля чаще всего используются для сравнения и, если необходимо, добавить соответствующие индексы. Это может в несколько раз увеличить производительность.

Введение в SQL

Сначала я хотел выделить отдельный раздел для вводной информации о языке запросов, но потом решил сократить эту информацию до минимума, а максимум внимания уделить практике.

С помощью SQL-запросов можно создавать реляционные базы данных. Этот язык стал стандартом, поэтому если вы хотите работать с базами данных, то должны знать его как свои пять пальцев.

Язык SQL определяется Американским национальным институтом стандартов (American National Standards Institute, ANSI) и Международной организацией по стандартизации (International Organization for Standardization, ISO). Но некоторые производители баз данных вносят изменения и дополнения в этот язык. Изменения незначительны, и основа остается совместимой со стандартом, а вот дополнений в разных базах данных может быть очень много. В Transact-SQL, который мы будем рассматривать в данной книге, очень много возможностей, которые не описаны стандартом SQL-92.

Что такое реляционная база данных? Это таблица, в которой в качестве столбцов выступают поля данных (поля определяют класс значений в соответствующей колонке таблицы), а каждая строка хранит данные. Строки очень часто в технической литературе называют записями. Здесь действует эффект разных языков. В английском языке строку таблицы в базе данных принято называть словом *record* (запись), потому что слово *line* (строка) имеет немного другой, более узкий смысл.

Давайте рассмотрим реляционную базу данных на примере табл. П1. Заголовок таблицы — это имена полей (колонок). Каждая строка — запись с данными. Реляционная база данных состоит из таких таблиц. На первый взгляд, все слишком просто, но эта простота позволяет решить практически любую задачу. Я, по крайней мере, еще не встречался с ситуацией, в которой реляционная база была бы бессильна.

Таблица П1. *Пример таблицы реляционной базы данных*

Фамилия	Имя	Отчество	Пол
Иванов	Иван	Иванович	М
Петрова	Мария	Анатольевна	Ж
Сидоров	Сергей	Петрович	М
...			

В каждой таблице должно быть одно уникальное поле, которое однозначно будет идентифицировать строку. Это поле называется ключевым. Эти поля очень часто используются для связывания таблиц. Но даже если у вас таблица не связана, ключевое поле все равно обязательно. Допустим, что вам необходимо изменить имя записи, где в поле `Фамилия` находится значение `Иванов`. Для этого нужно написать запрос, который будет реализовывать следующую логику: "Изменить поле `Имя` на значение `XXXX`, где поле `Фамилия` содержит значение `Иванов`". Именно такую логику будет использовать база данных. Но что произойдет, если в таблице есть две записи, в которых поле `Фамилия` содержит одно и то же значение? В этом случае будут обновлены все строки, в которых в поле `Фамилия` содержится `Иванов`.

Но обновление всех записей не всегда является необходимым. Чтобы однозначно идентифицировать строку, лучше использовать ключевые поля. База данных сама следит, чтобы данные в ключевом поле были уникальными для каждой строки. Для упрощения создания ключевых полей в большинстве баз данных есть специальный тип данных (например, счетчик), который может автоматически увеличивать значение для каждой новой строки, чем и достигается уникальность. В табл. П2 приводится пример таблицы базы данных, где первое поле — это автоматически увеличиваемое число.

Таблица П2. *Пример таблицы реляционной базы данных с ключевым полем*

Ключ	Фамилия	Имя	Отчество	Пол
1	Иванов	Иван	Иванович	М
2	Петрова	Мария	Анатольевна	Ж
3	Сидоров	Сергей	Петрович	М
4	Иванов	Иван	Иванович	М
...				

В табл. П2 я добавил еще одну запись, в которой все поля идентичны строке с номером 1. Это реальная ситуация, ведь вполне вероятно наличие двух человек с одними и теми же фамилией, именем и отчеством, но разной датой рождения. Теперь, чтобы обновить запись первого Иванова, необходимо написать запрос со следующей логикой: "Изменить поле `Имя` на значение `XXXX`, где ключевое поле содержит значение 1". Благодаря уникальному ключевому полю мы четко определяем запись, которую необходимо обновить.

Имена полей в базе данных также должны быть уникальными, но в этом случае не обязательно числовыми. Их можно называть как угодно, лишь бы уни-

кальность соблюдалась в пределах таблицы, а имя отражало содержащиеся данные.

Язык запросов SQL может быть двух типов: интерактивный и вложенный. Интерактивный вариант — это отдельный язык, он сам выполняет запросы и сразу показывает результат работы. Вложенный SQL — это язык, используемый в составе другого, например, C++ или Delphi. Разницы в синтаксисе практически нет, отличие только в том, как использовать запросы. В этой книге мы будем подразумевать, что у нас интерактивный SQL. Интерактивный SQL ближе к стандартному, а во вложенном очень часто встречаются отклонения и дополнения (например, в описании параметров, которые передаются запросу).

В стандартном SQL различаются только два типа данных: строки и числа, но некоторые производители добавляют свои типы, и MS SQL Server, который будет рассматриваться в книге, не исключение. Базы данных MS SQL Server поддерживают множество различных типов, что упрощает разработку, но нам необходимо научиться правильно отражать эти данные.

Числа в SQL делятся на два типа: целые (`INTEGER` или `INT`) и дробные (`DECIMAL` или `DEC`). Во многих базах данных числа делятся и по размеру используемого в базе пространства, но стандарт этого уже не требует. Строки в стандарте ограничены 254 символами, но в реальности размер намного больше.

Целочисленные типы указываются как есть, без каких-либо кавычек. Строковые типы, даты, глобально уникальные идентификаторы оформляются в одинарных кавычках, например:

```
'Это строка'
```

Если необходимо указать, что строка должна быть закодирована Unicode-символами, то необходимо перед строкой указать букву N, например:

```
N'Это пример Unicode строки'
```

Постепенно мы познакомимся с типами данных более подробно, а список поддерживаемых сервером MS SQL типов можно увидеть в *приложении 1*.

Если исходить из стандарта, то каждый запрос должен заканчиваться символом точки с запятой. Например:

```
Текст запроса;
```

Мы же в примерах этот символ будем очень часто опускать, потому что он не является обязательным для MS SQL Server. Я явно буду указывать на те случаи, когда символ просто необходим.

Работа с запросами

Для работы с первыми двумя главами вам понадобится база данных, которая поддерживает стандарт SQL 1992 года. Это практически единственное требование, но не все базы данных четко следуют стандарту. Желательно, чтобы база данных, которую вы выберете, поддерживала его максимально полно.

Лучшим вариантом будет MS SQL Server, потому что он максимально полно соответствует стандарту, но можно использовать и MS Access, хотя тут и есть отличия при установке связей между таблицами, и поддерживается только SQL, а не Transact-SQL.

Начиная с *главы 3*, мы будем углубляться в расширение языка SQL под названием Transact-SQL, и вот тут уже сервер баз данных MS SQL Server окажется незаменимым, потому что только он поддерживает это расширение.

Для вашего удобства на компакт-диске, прилагающемся к книге, в каталогах ChapterX (где X — номер главы) расположены файлы с текстом запросов и комментариями. Эти файлы избавят вас от необходимости набирать коды запросов на клавиатуре.

Все возможности языка SQL мы будем обсуждать с одновременным рассмотрением примеров. Сначала я буду приводить общий вид команды, а потом постепенно будем изучать возможности команды на примерах, максимально приближенных к реальным. Это позволит вам на практике увидеть, как использовать те или иные возможности, и иметь готовый багаж запросов для решения наиболее часто встречающихся задач.

Именование

Для того чтобы вам проще было понимать материал книги, вы должны знать, каким образом я присваиваю имена объектам базы данных. К каждому имени объекта я добавляю префикс, который будет указывать тип. Это позволит быстро определить по имени тип любого объекта. В *приложении 1* показаны типы полей, поддерживаемые в MS SQL Server. Например, если объект имеет тип `varchar`, то к имени объекта я добавляю префикс `vc`. После этого идет имя, отражающее суть объекта. Если этот объект предназначен для хранения номера телефона, то полное название поля будет `vcTelephon`.

В базах есть определенные поля, которые должны управляться по-другому. Например, для именования ключевого поля можно использовать следующий вид: `idИмяТаблицы`. Таким образом, по ключу можно понять, к какой таблице он относится, а по имени таблицы узнать имя ключа. В некоторых простых примерах для первичного ключа мы будем использовать просто имя `id`,

но при рассмотрении сложных связей я буду следовать правилу именования `idИмяТаблицы`.

Когда ключ используется для связи с другой таблицей, то после `id` нужно ставить имя таблицы, с которой происходит связь. Помимо этого, можно отделить идентификатор `id` знаком подчеркивания, чтобы сразу было видно, что это не основной ключ, а связь. Таким образом, по имени поля можно узнать, для связи с какой таблицей он предназначен.

Отдельного разговора заслуживают триггеры. Триггер — это процедура, которая вызывается в ответ на определенные действия над данными: вставку, изменение или удаление (операции `insert`, `update` или `delete`). Это как бы обработчики событий, в которых можно повлиять на обработку данных. Вот тут для именования наилучшим способом будет следующий:

```
ИмяТаблицы_СмыслТриггера_Операции
```

Порядок может быть любым, лишь бы вам было удобно, но желательно, чтобы все три составляющие присутствовали. Допустим, что у вас должен быть триггер на таблице `Person`, который при добавлении новых записей или изменении существующих должен проверять существование дублирующих записей. Имя такого триггера может быть следующим:

```
Person_CheckDouble_iu
```

По этому имени можно сразу понять, для какой таблицы он предназначен, что делает и когда выполняется. В MS SQL Server триггеры привязаны к базам данных и таблицам, а в Oracle все триггеры находятся в одном хранилище, вне зависимости от таблиц, и здесь вы реально оцените выгоду от такого метода именования.

К именам процедур и функций я не предъявляю особых требований, но рекомендую выделять их из общей массы. Какой способ выберете вы, зависит от личных предпочтений. В данной книге будут использоваться различные способы именования, без жесткой привязки к определенному правилу, чтобы показать вам все способы именования, но определенную закономерность легко проследить.

Язык SQL нечувствителен к регистру, а значит, если написать какой-то оператор в верхнем регистре и в нижнем, результат будет одинаковым. Например, слова `SELECT` и `select` для сервера являются идентичными.

Все операторы SQL я пишу в верхнем регистре. Это не является обязательным, так что можно все писать в нижнем регистре. Большинство интерпретаторов SQL-команд нечувствительны к регистру. Благодаря верхнему регистру вы сразу можете выделить операторы языка SQL из общей массы имен полей, таблиц и т. д.

CyD SQL Factory

Прежде чем мы начнем рассматривать сам язык, давайте посмотрим, как можно выполнять запросы в MS SQL Server и MS Access, чтобы вы могли на практике проверять все, что описывается в книге. Для работы с запросами я рекомендую использовать программу CyD SQL Factory, которую можно найти на прилагаемом компакт-диске в каталоге Programs. Программа не требует установки, просто скопируйте ее на свой локальный жесткий диск и используйте. Посмотрим, как это делать. Сначала нужно создать новый проект. Для этого выбираем меню **File | New project** (Файл | Новый проект). Программа создаст клиентское окно, которое разделено на две части. В левой половине находится текстовый редактор для ввода текста запросов, в правой половине можно увидеть схему базы данных (рис. П1). Мы пока не указали базу данных, которую хотим использовать, поэтому схема пуста.

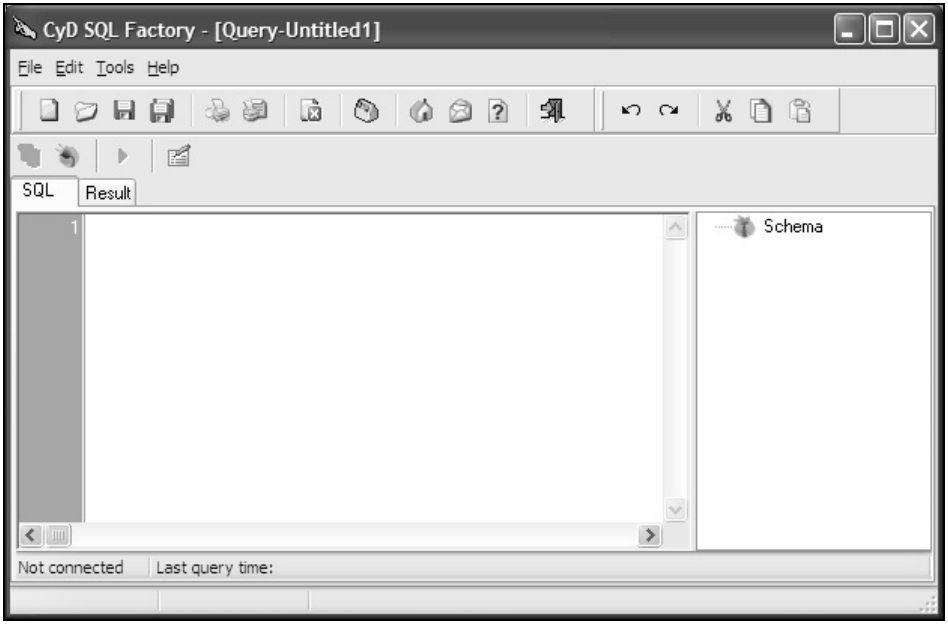


Рис. П1. Главное окно программы SQL Factory

Клиентское окно программы также разделено на две закладки: **SQL** и **Result** (Результат). На закладке **SQL** мы пишем запросы, а на закладке **Result** можно увидеть результат выполнения.

Для создания соединения с сервером нажмите кнопку **Connect to server** (Соединиться с сервером) в панели задач клиентского окна. Перед вами откроется окно, как на рис. П2.

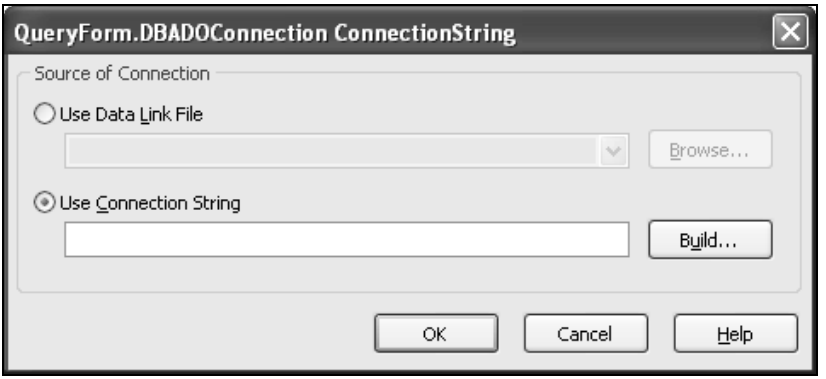


Рис. П2. Окно создания соединения с сервером

Соединение можно настроить, используя файл подключения (**Use Data Link File**) или строку подключения (**Use Connection String**). Рассмотрим второй вариант, как более простой и универсальный. Напротив строки **Use Connection String** нажмите кнопку **Build**, и на экране появится окно настройки строки соединения. На первой закладке нужно выбрать драйвер, который будет использоваться для подключения. Для MS Access нужно выбрать Microsoft Jet 4.0 OLE DB Provider (версия может отличаться), а для MS SQL Server лучше всего использовать Microsoft OLE DB Provider for SQL Server.

Теперь переходим на вкладку **Connection** (Соединение). Ее содержимое зависит от выбранного драйвера. В случае использования MS Access на этой вкладке будет только поле для указания пути к MDB-файлу базы данных. Для MS SQL Server это окно содержит больше настроек (рис. П3):

- **Select or enter a server name** (Выберите или введите имя сервера) — здесь нужно указать имя SQL-сервера, с которым вы хотите работать;
- **Enter information to log on to the server** (Введите информацию для входа на сервер) — здесь можно выбрать один из переключателей:
  - **Use Windows NT Integrated security** (Использовать интегрированную в Windows NT безопасность) — если вы вошли в ОС под учетной записью, которая имеет право доступа к SQL-серверу, то можно выбрать этот пункт. Например, SQL-сервер установлен локально, и вы входите в систему как администратор. Локальный администратор по умолчанию имеет право доступа к SQL-серверу, и можно выбирать этот переключатель;
  - **Use a specific user name and password** (Использовать указанные имя пользователя и пароль). В этом случае пароль будет указываться в соответствующих полях под этим переключателем;

- **Select the database on the server** (Выберите базу данных на сервере) — здесь можно выбрать из списка интересующую вас базу данных.

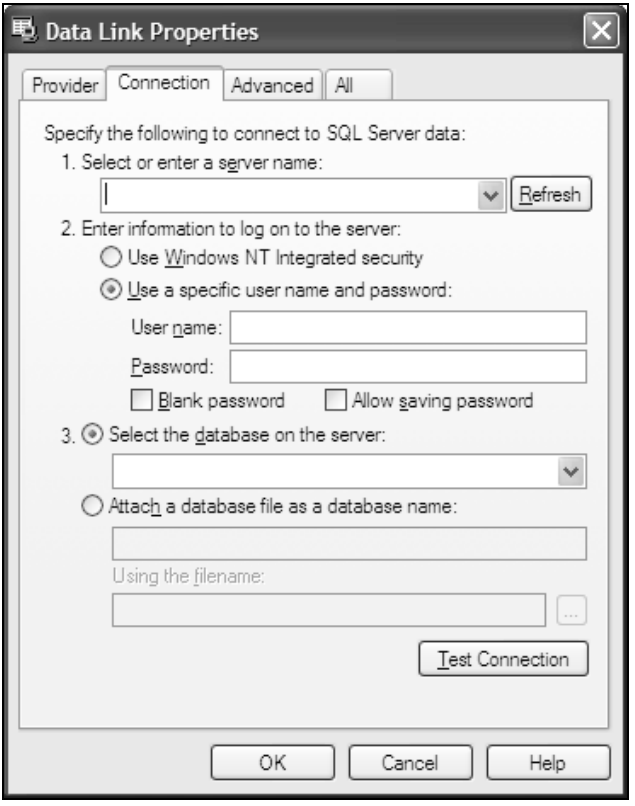


Рис. П3. Вкладка **Connection** при подключении к MS SQL Server

Если вы явно указывали имя пользователя и пароль, то обязательно установите флажок **Allow saving password** (Позволять сохранять пароль). Укажите необходимые данные и нажмите кнопку **OK** для сохранения изменений. Программа попытается соединиться с сервером, и если все прошло удачно, в строке состояния появится надпись **Connected** (Подключен), а в дереве схемы появятся существующие таблицы выбранной базы данных (рис. П4).

Теперь в редакторе можно писать запросы и выполнять их. Попробуйте сейчас выполнить запрос:

```
SELECT * FROM ИмяТаблицы
```

Параметр `ИмяТаблицы` нужно заменить на имя, уже существующее в базе данных. Можете выбрать любое имя из списка, отображаемого в схеме. Чтобы

выполнить запрос, нажмите клавишу <F5>. Окно переключится на вкладку **Result**, где будет отображена сетка с данными выбранной таблицы.

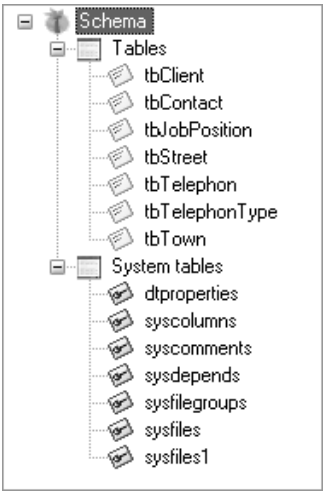


Рис. П4. Результат отображения схемы базы данных

Если у вас в редакторе большой запрос, но вы хотите выполнить только часть, выделите необходимый отрывок запроса и также нажмите клавишу <F5>.

Если вы выполнили запрос, который изменяет структуру базы данных (например, создает таблицу), то необходимо обновить схему, чтобы увидеть изменения. Для этого щелкните правой кнопкой в дереве схемы и в контекстном меню выберите пункт **Update Schema** (Обновить схему). С помощью этого же меню можно просмотреть любую таблицу. Выделите таблицу в схеме и затем в контекстном меню выберите пункт **Show All Table Data** (Отобразить все данные из таблицы).

Я рекомендую вам установить SQL Server с программой CyD SQL Factory и тестировать на своем сервере все запросы, описываемые в книге. Я уже не раз говорил, что практика является наилучшим учителем. Когда вы сами выполните все действия и увидите результат, то лучше поймете описываемый материал и запомните его. Я всегда это рекомендую своим читателям.

Query Analyzer

Программа CyD SQL Factory удобна тем, что способна работать с любыми базами данных, для которых есть Windows-драйвер, но ее возможности ограничены. Если вы будете работать с MS SQL Server, то более эффективным

решением будет использовать утилиту Query Analyzer, которая входит в поставку данного сервера. Давайте рассмотрим основные возможности работы с программой.

Запустите Query Analyzer, и откроется окно, как на рис. П5. Здесь необходимо задать параметры подключения к серверу. В выпадающем списке SQL Server необходимо задать имя сервера, к которому вы хотите подключиться. Если щелкнуть на кнопке справа от списка, то появится окно выбора доступных в сети серверов, но это окно далеко не всегда показывает все серверы, поэтому вы должны уметь вводить имя вручную.

Рис. П5. Окно создания соединения с сервером

Если у вас SQL-сервер установлен на локальном компьютере, то достаточно ввести точку или написать сетевое имя компьютера (можно вместо имени указать IP-адрес). Если сервер был установлен как экземпляр по умолчанию, то этого достаточно. Если вы устанавливали именованный экземпляр SQL Server, то вы должны написать имя в формате:

```
Имя компьютера\имя экземпляра
```

У меня компьютер называется notebook, а имя установленного экземпляра MS SQL Server — `flenov`, поэтому адрес подключения будет выглядеть так:

```
notebook\flenov
```

Чуть ниже необходимо выбрать параметры авторизации. Если вашей учетной записи, под которой вы сейчас работаете, разрешено подключаться к серверу SQL, то можно выбрать **Windows Authentication** (Аутентификация Windows), иначе нужно выбрать **SQL Server Authentication** (Аутентификация SQL Server) и указать явное имя и пароль, которым разрешено подключение к серверу.

Если установить флажок **Start SQL Server if it is stopped** (Запустить SQL Server, если он остановлен), то программа может запустить сервер баз данных, если он в данный момент не запущен. Это очень удобно, если вы не установили MS SQL Server для автоматического запуска. У меня SQL-сервер работает на ноутбуке, и я программирую в нем не каждый день, поэтому, чтобы запуск сервера не отнимал лишнее время при загрузке ОС, я отключил автозагрузку MS SQL Server, и эта опция очень сильно помогает мне.

Если вы указали параметры подключения верно, то после нажатия кнопки **OK** перед вами появится окно программы для работы с SQL-запросами (рис. П6). Это окно разделено вертикально на две части. В верхней части окна мы пишем SQL-запросы, а в нижней отображается результат работы.

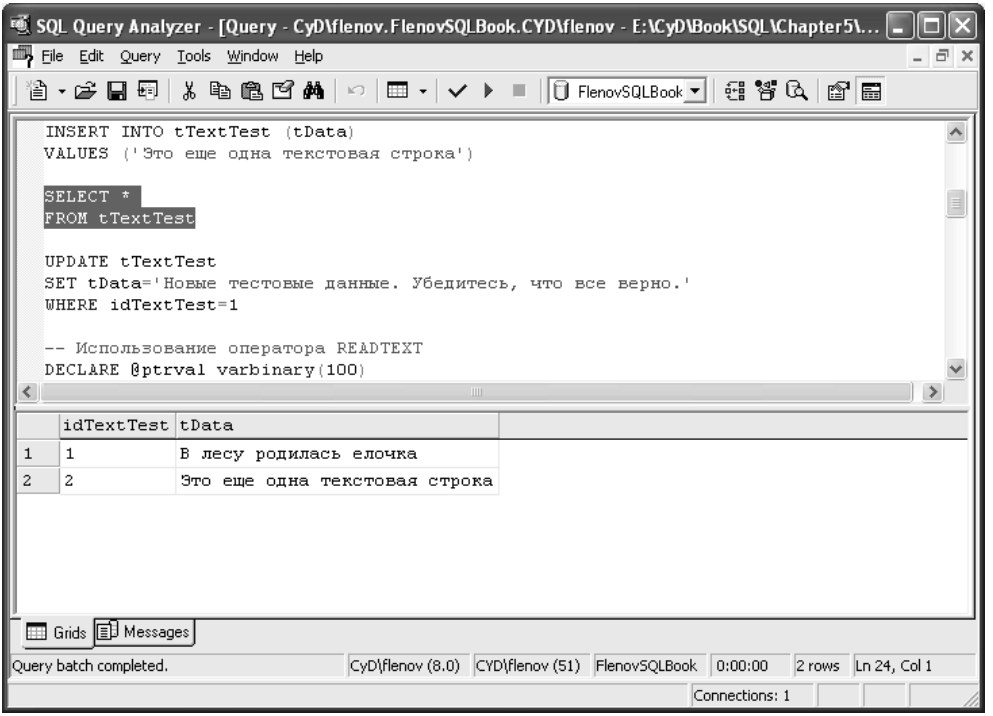


Рис. П6. Окно работы с SQL-запросами

Чтобы Query Analyzer выполнил запрос, нажмите клавишу <F5>. При этом будет выполнен весь код, набранный в окне запроса. Если необходимо выполнить только часть, то перед нажатием <F5> выделите необходимую часть запроса.

Если вы набрали сразу несколько запросов, то при запуске на выполнение сервер будет выполнять их как одно целое. Иногда это может привести

к ошибкам. Например, если первый запрос создает объект (таблицу, процедуру, функцию и т. п.), а второй обращается к объекту. Если оба запроса будут выполняться как одно целое, то сервер не сможет найти создаваемый объект. Чтобы решить эту проблему, необходимо написать между запросами оператор `GO`:

```
Код создания объекта
GO
Код обращения к объекту
```

Теперь два запроса будут отправлены серверу по отдельности, и они будут выполнены корректно.

Иногда нам придется переключаться между различными базами данных. Для этого в Query Analyzer можно использовать выпадающий список в панели инструментов или набрать команду:

```
USE Имя базы
```

Команды `USE` и `GO` не относятся к операторам Transact-SQL, а выполняются в клиентской программе, поэтому могут не действовать в других программах работы с запросами.

На этом мы пока оставим Query Analyzer, а его дополнительные возможности будем осваивать по мере надобности. А теперь уже пора перейти к знакомству непосредственно с операторами ANSI SQL и Transact-SQL.

Глава 1

Управление базой данных

Большинство авторов начинает рассматривать SQL с операторов получения данных из базы. Но у нас еще нет баз данных и неоткуда брать данные. Поэтому мы начнем с создания базы данных и таблиц. Когда у нас будет готова тестовая база, мы заполним ее данными и тогда уже научимся работать с ними.

Итак, в этой главе нам предстоит узнать:

- как создавать базу данных с помощью SQL-запросов;
- как изменять параметры базы данных;
- как создавать таблицы;
- как изменять параметры таблицы.

Я рекомендую выполнить все сценарии из каталога Chapter1 на компакт-диске, прилагаемом к книге. Это позволит вам иметь готовую структуру базы данных, на которой можно будет тестировать запросы из следующих глав.

Большинство описываемых в этой главе операторов SQL будут работать на большинстве баз данных. Но есть некоторые тонкости. Например, в MS Access нельзя создавать базу данных, потому что в этом случае база данных — файл, который создается с помощью одноименной программы. В других базах данных операторы создания баз и таблиц имеют точно такой же синтаксис, но могут быть отличия в поддерживаемых параметрах из-за большего или меньшего количества возможностей.

Операторы по описанию объектов базы данных выделяют в отдельный язык (подъязык SQL) — DDL (Data Definition Language, язык определения данных). Именно он будет рассматриваться в этой главе, ведь нам предстоит научиться описывать данные.

1.1. Создание и удаление базы данных

Информация о каждой базе данных в SQL Server хранится в таблице `sysdatabases` базы данных master. Поэтому желательно (но не обязательно) использовать базу данных master во время создания базы. К тому же, после изменения любой пользовательской базы данных нужно создавать резервную копию базы данных master. О резервном копировании и восстановлении мы поговорим в *разд. 4.10*.

Создание базы данных — это процесс указания имени, размера и расположения файлов.

В Transact-SQL для создания базы данных есть команда `CREATE DATABASE`. Эта команда может выполняться только с сервером SQL Server. При использовании базы данных MS Access команда недоступна, потому что базой данных является файл с расширением mdb, который создается в программе Access, и к которому мы подключены.

Сервер MS SQL может содержать несколько баз данных. Вы можете подключиться к любой из них (системной или тестовой, которые присутствуют в стандартной поставке) и создать новую базу данных, но желательно подключиться к базе данных master.

Синтаксис команды создания базы данных показан в листинге 1.1.

Листинг 1.1. Создание базы данных

```
CREATE DATABASE имя
[ ON
    [ < filespec > [ ,...n ] ]
    [ , < filegroup > [ ,...n ] ]
]
[ LOG ON { < filespec > [ ,...n ] } ]
[ COLLATE имя_раскладки ]
[ FOR LOAD | FOR ATTACH ]

< filespec > ::=
[ PRIMARY ]
( [ NAME = логическое_имя_файла , ]
    FILENAME = 'имя_файла_в_ОС'
    [ , SIZE = размер ]
```

```
    [ , MAXSIZE = { максимальный_размер | UNLIMITED } ]
    [ , FILEGROWTH = увеличение ] ) [ ,...n ]

< filegroup > ::=
FILEGROUP файловая_группа < filespec > [ ,...n ]
```

Давайте разберем этот код подробно. Первая строчка содержит имя команды и параметр:

```
CREATE DATABASE имя
```

В качестве параметра выступает имя создаваемой базы данных.

Если вы посмотрите на остальные параметры команды, то заметите, что все они находятся в квадратных скобках. Обычно в квадратных скобках указываются необязательные параметры. Получается, что самый простой вариант команды создания базы выглядит так:

```
CREATE DATABASE Имя
```

Имена, состоящие из нескольких слов, необходимо заключать в квадратные скобки:

```
CREATE DATABASE [Тестовая база]
```

Очень интересной является следующая строчка:

```
[ FOR LOAD | FOR ATTACH ]
```

Здесь в квадратных скобках указано два значения, разделенных вертикальной чертой. Это значит, что они являются необязательными, а вертикальная черта соответствует слову "или", т. е. в запросе можно будет указывать или `FOR LOAD`, или `FOR ATTACH`, или вообще ничего. Оба параметра одновременно указывать нельзя.

В угловых скобках указываются имена секций. Например, в описании оператора `CREATE DATABASE` есть два указания на `< filespec >`. Эта секция может идти после ключевого слова `ON` и после `LOG ON`. Описание самой секции идет после:

```
< filespec > ::=
```

Если вы имеете опыт программирования на одном из высокоуровневых языков, то в секциях вы уже, наверное, увидели аналогию с процедурами. Название секции `< filespec >` аналогично имени процедуры, а после `< filespec > ::=` идет сам код процедуры.

Следующая интересная строчка:

```
[ < filespec > [ ,...n ] ]
```

Здесь `< filespec >` — описание файла, а `[ ,...n ]` указывает на то, что возможно несколько описаний.

С помощью круглых скобок параметры объединяются в группу, например:

```
( [ NAME = логическое_имя_файла , ]
    FILENAME = 'имя_файла_в_ОС'
    [ , SIZE = размер ]
    [ , MAXSIZE = { максимальный_размер | UNLIMITED } ]
    [ , FILEGROWTH = увеличение ] ) [ ,...n ]
```

В данном случае в группу объединены параметры `NAME`, `FILENAME`, `SIZE`, `MAXSIZE` и `FILEGROWTH`, т. к. все они описывают файл. Обязательным является только параметр `FILENAME`. После круглых скобок идет `[ ,...n ]`, это означает, что может быть несколько описаний файлов (для каждого файла базы данных свое описание).

Параметр `FILENAME` интересен еще и тем, что его значение задается с помощью знака равенства, после которого идет текст в одинарных кавычках:

```
FILENAME = 'имя_файла_в_ОС'
```

Кавычки в данном случае обязательны. По наличию кавычек достаточно просто определить тип параметра. Если они присутствуют, то параметр строковый, иначе числовой. Например, параметр `SIZE` не содержит кавычек, а значит, он числовой:

```
SIZE = Размер
```

На первый взгляд, общий вид оператора `CREATE DATABASE` достаточно сложен, но здесь не так уж и много параметров. Давайте рассмотрим основные, а потом увидим их действие на практике.

- `PRIMARY`. Этот параметр указывает файл в основной файловой группе. Эта файловая группа содержит все системные базы данных. Она также содержит все объекты, не назначенные другим файловым группам. Каждая база данных содержит один основной файл данных. Основной файл — это стартовая точка базы данных. Кроме того, он указывает на ее местонахождение. Для основного файла рекомендуется расширение mdf. Если вы не укажете этот параметр, первый файл списка описания будет использован как основной.
- `FILENAME`. Этот параметр указывает имя и путь к файлу в операционной системе. Путь должен указывать папку на сервере, где установлен SQL Server. Нельзя использовать сетевые диски с других компьютеров.
- `SIZE`. Этот параметр указывает размер файла данных или журнала. Вы можете указать размер в мегабайтах (значение по умолчанию) или в кило-

байтах. Минимальный размер — 512 Кбайт для обоих файлов: журнала транзакций и файла данных. Размер, указанный для основного файла базы данных, должен быть больше или равен размеру основного файла базы данных model. Мы уже говорили, что база model копируется во все новые базы данных, поэтому размер новой не может быть меньше размера model, иначе копирование станет невозможным. Когда вы добавляете новый файл базы данных или журнала без указания размера, то сервер использует значение размера по умолчанию 1 Мбайт.

- `MAXSIZE`. Этот параметр указывает максимальный размер, до которого файл может увеличиваться. Вы можете указать размер в мегабайтах (значение по умолчанию) или в килобайтах. Если вы не укажете максимальный размер, файл будет увеличиваться, пока диск не будет заполнен целиком.
- `FILEGROWTH`. Этот параметр указывает размер приращения файла. Значение этого параметра для файла не может превышать значение `MAXSIZE`. Значение 0 указывает на запрет увеличения. Значение может быть указано в мегабайтах (по умолчанию), килобайтах или процентах. Значение по умолчанию, если этот параметр не указан, — 10%, а минимальный размер — 64 Кбайт. Указанный размер округляется до ближайшего числа, кратного 64 Кбайт.
- `COLLATE`. Этот параметр указывает значение по умолчанию для сопоставления в базе данных. Сопоставления (кодировка или раскладка) включают правила, контролирующие использование символов для языка и алфавита.

Во время создания базы данных очень важно понимать, как SQL Server хранит данные, чтобы вы могли сосчитать и указать размер дискового пространства для размещения базы данных. Во время создания баз учитывайте следующее.

- Все базы данных имеют основной файл данных, определяемый именем файла с расширением mdf, и один или более файлов журнала, определяемый именем файла с расширением ldf. База данных может также иметь вторичные файлы данных, которые определяются по имени файла с расширением ndf. Файлы могут объединяться в группы, о чем мы поговорим в *разд. 1.1.1*.
- Физические файлы имеют двойное именование — имя ОС и имя, которое вы можете использовать в операторах Transact-SQL (логическое имя, которое указывается в параметре `NAME`).
- Когда вы создаете базу данных, в нее копируется содержимое базы данных model, которая включает в себя системные таблицы и может содер-

жать пользовательские таблицы. Минимальный размер создаваемой базы данных должен быть равен или больше размера базы данных model.

- Сервер SQL хранит, читает и записывает данные блоками по 8 Кбайт, эти блоки называются страницами. База данных может хранить 128 страниц на мегабайт. Все страницы хранятся в пространстве. Пространство — это 8 последовательных страниц, или 64 Кбайт. Поэтому база данных имеет 16 пространств в мегабайте.

Страницы и пространства — это основа структуры базы данных SQL Server. Сервер MS SQL использует различные типы страниц, некоторые следят за выделенным пространством, а некоторые содержат пользовательские данные и индексы. Страницы, которые отслеживают выделенное пространство, содержат плотно сжатую информацию. Это позволяет серверу MS SQL эффективно помещать их в память, чтобы облегчить просмотр.

Сервер SQL использует два типа пространств.

- Пространства, которые хранят страницы от двух и более объектов, называются смешанными. Каждая таблица начинается как смешанное пространство. Вы используете смешанное пространство, главным образом, для страниц, которые хранят пространство и содержат маленькие объекты.
- Пространства, которые хранят все 8 страниц, выделенных одному объекту, называются однородными пространствами. Они используются, когда таблице или индексу требуется более 64 Кбайт.

Первое пространство для каждого файла является смешанным и содержит страницы заголовка файла, а потом выделяется по три страницы непосредственно для данных. Сервер выделяет эти смешанные пространства, когда вы создаете основной файл данных, и использует эти страницы для своих внутренних задач. Страница заголовка файла содержит атрибуты файла, такие как имя базы данных, которая хранится в файле, файловая группа, минимальный размер, размер приращения. Это первая страница в каждом файле (Страница 0).

Страница свободного пространства (PFS) — это выделенная страница, содержащая информацию о свободном пространстве, доступном в файле. Эта информация хранится в странице 1. Каждая такая страница может простираться на 8000 смежных страниц, что приблизительно составляет 64 Мбайт данных.

Журнал транзакций захватывает всю необходимую информацию о происходящих на сервере изменениях для восстановления базы данных в момент возникновения системной ошибки и для обеспечения целостности данных. О журнале мы будем говорить чуть позже, а сейчас необходимо понимать, что журнал — это отдельный файл, который требует дискового пространства.

Теперь рассмотрим, как можно удалять созданную базу данных:

```
DROP DATABASE имя
```

Нельзя удалять:

- базу данных, которая открыта для чтения или записи любым пользователем, поэтому при удалении вы также не должны быть к ней подключены. Лучше всего подключиться к базе данных master;
- базу данных, которая опубликовала любую свою таблицу как часть репликации SQL Server;
- системную базу данных.

Рассмотрим простейший пример создания и удаления базы данных. Для создания базы данных достаточно написать:

```
CREATE DATABASE имя
```

Все остальные параметры являются необязательными. Попробуем создать базу данных с именем `TestDatabase` и удалить. Сначала создадим базу:

```
CREATE DATABASE TestDatabase
```

И тут же ее удалим:

```
DROP DATABASE TestDatabase
```

Эти команды нужно выполнять по отдельности. Одновременно выполнять нельзя, иначе сервер вернет сообщение об ошибке. Рекомендую тестировать примеры. После этого тестовые базы данных можно удалять.

Теперь посмотрим, какие еще возможности дает нам команда создания базы данных. Имя базы данных ограничено 128 символами, если вы явно указываете логическое имя файла журнала. Я считаю, что этого вполне достаточно. Если логическое имя журнала не задано, то размер сокращается до 123 символов. Это связано с тем, что логическое имя журнала также ограничено 128 символами, и если оно не указано, то в качестве имени используется имя базы плюс суффикс `_log`. Самое интересное, что суффикс занимает четыре символа, а 128 – 4 = 124. Почему Microsoft ограничивает длину имени числом 123, для меня остается загадкой. Быть может, составители документации разучились считать?

Рассмотрим основные правила, которым должны подчиняться имена баз данных и объектов в ней.

- Первый символ должен быть буквой a—z, A—Z.
- После первого символа может идти буква, цифра или символы _, @ или #.
- Идентификаторы, начинающиеся с символов, имеют специальное назначение:
  - идентификаторы, начинающиеся с символа @, являются локальными переменными или параметрами;

- идентификаторы, начинающиеся с символа #, являются временными таблицами или процедурами;
- идентификаторы, начинающиеся с символа ##, являются глобальными временными объектами.

Эти правила относятся ко всем именам — базы данных, таблиц, полей и т. д. С некоторыми из них мы еще будем не раз встречаться на практике, например, при создании временных объектов.

Чтобы указать дополнительные параметры, нужно после оператора `CREATE DATABASE` написать `ON` и далее в круглых скобках перечислить параметры:

```
CREATE DATABASE TestDatabase ON
(
 Параметры
)
```

Если какие-то параметры не заданы, то их значения берутся по умолчанию. Пока мы не будем обсуждать возможные параметры, потому что они одинаковы для базы данных и журнала транзакций, и нужно рассмотреть, как описывается журнал.

Журнал транзакций — это файл, в котором сохраняется информация об активности в отношении базы данных. Например, вы выдаете запрос на обновление данных. В общем виде сервер выполняет следующие действия:

- ❒ если явно не указано, то сервер автоматически создает новую транзакцию;
- ❒ запрос сохраняется в журнале;
- ❒ сервер начинает выполнять обновление;
- ❒ если все изменения прошли успешно, то транзакция помечается как завершенная, и данные считаются обновленными;
- ❒ если произошел сбой сервера или ошибка обработки данных, то транзакция не завершается. В этом случае все изменения, сделанные незавершенной транзакцией, отменяются, как будто ничего не произошло. Если во время транзакции отключилось питание и данные не успели обновиться, то при следующей загрузке сервер увидит в журнале незавершенную транзакцию и попытается завершить.

Получается, что журнал обеспечивает целостность данных. Посмотрим на файловую систему NTSF. Это в своем роде тоже база данных. Если какой-либо файл поврежден и не читается, то он только нарушает целостность файловой системы и его необходимо удалить. Программа сканирования scandisk проверяет все транзакции с файловой системой и быстро находит незавершенные операции, отменяя их или завершая.

Вы должны уделять внимание не только корректности и оптимальности создания базы данных, но и журналу транзакций.

Теперь давайте рассмотрим, какие параметры можно указывать при создании файлов базы данных и журнала транзакций:

- `NAME` — логическое имя, которое будет отображаться в SQL-сервере;
- `FILE NAME` — физическое расположение и имя файла;
- `SIZE` — начальный размер файла. Это начальное значение, которое может увеличиваться по мере надобности;
- `MAXSIZE` — максимальный размер файла. Чтобы файл не имел ограничений, необходимо указать в этом параметре `UNLIMITED` вместо реального значения. Но я рекомендую указывать такое значение, которое не сможет переполнить весь жесткий диск;
- `FILEGROWTH` — приращение, т. е. на сколько должен увеличиться размер файла, если текущего размера недостаточно. Приращение можно указывать как в реальных значениях (мегабайты), так и в процентном отношении к текущему размеру. Приращение должно быть достаточным, чтобы эта операция выполнялась как можно реже.

Давайте теперь рассмотрим несколько примеров, которые позволят закрепить все вышесказанное. В листинге 1.2 описываются параметры файла данных и параметры журнала. Если какой-то параметр не указан явно, то его значение берется по умолчанию.

Листинг 1.2. Создание базы данных с описанием параметров файлов

```
CREATE DATABASE TestDatabase
ON
(
  NAME = TestDatabase_data,
  FILENAME = 'c:\data\test.mdf',
  SIZE = 10MB,
  MAXSIZE = 100GB,
  FILEGROWTH = 5MB
 )
LOG ON
(
  NAME = 'TestDatabase_log',
  FILENAME = 'c:\data\test.ldf',
```

```
  SIZE = 5MB,
  MAXSIZE = 10GB,
  FILEGROWTH = 5%
)
```

Посмотрим, что делает этот сценарий. В первой строчке мы указываем, что необходимо создать базу данных с именем `TestDatabase`. Затем идет ключевое слово `ON` и в круглых скобках перечисляются параметры файла базы данных. Мы указываем пять параметров: логическое имя (`NAME`), физическое расположение файла данных (`FILENAME`), начальный размер (`SIZE`) 10 Мбайт, максимальный размер (`MAXSIZE`) 1000 Гбайт и приращение (`FILEGROWTH`) 5 Мбайт. Этого достаточно только для тех баз данных, где добавление новых записей происходит редко.

Для часто изменяемых баз я рекомендую указывать большое приращение для предотвращения частого автоувеличения. Это облегчает решение административных задач, уменьшает фрагментацию файла и избавляет сервер от лишнего расширения файлов базы. Приращение относится как к файлу базы данных, так и к файлу журнала.

Если база данных не изменяется в размере (не добавляются новые данные), но очень часто происходит изменение уже существующих данных, то приращение для файла данных можно сделать небольшим (5—10 Мбайт, в зависимости от размера одной записи таблицы). При этом файл журнала должен иметь большее приращение, которое лучше всего указать в процентах от существующего размера. Исходя из практики, я бы порекомендовал значение 10%, но в зависимости от задачи значение можно скорректировать.

Если данные базы используются только как справочник и изменяются очень редко, то размер приращения базы и журнала можно сделать минимальным. Запросы на выборку данных не сохраняются в журнале транзакций, поэтому можно ограничиться увеличением в 1 Мбайт. Но при этом значение приращения желательно сделать больше, чем потребуется для хранения 100 строк данных. Размер строки можно примерно рассчитать по размеру всех полей самой большой таблицы.

Если вы используете автоматическое увеличение, то лучше всего указать максимальный размер. Это позволит вам предотвратить заполнение базой данных всего жесткого диска.

Приращение для файла данных можно указывать и в процентах, например:

```
CREATE DATABASE TestDatabase
ON
(
```

```
  NAME = TestDatabase_data,
  FILENAME = 'c:\data\test.mdf',
  SIZE = 10MB,
  MAXSIZE = 100GB,
  FILEGROWTH = 10%
 )
```

Вернемся к нашему примеру из листинга 1.2. После задания параметров файла данных стоит ключевое слово `LOG ON`, после которого идет описание параметров файла журнала транзакций. Тут все примерно то же самое, только приращение указано в процентах (можно и в мегабайтах). Это очень удобно, потому что журнал пополняется не только при добавлении данных, но и при изменении. Указание приращения в процентах позволяет уменьшить количество операций по увеличению размера файла.

С помощью команды создания базы данных можно подключать уже существующие файлы. Не путайте с восстановлением данных, когда база восстанавливается из резервной копии. Подключение — это создание базы данных на основе существующего MDF-файла, а не резервной копии. Файл журнала в этом случае желателен, но не обязателен. Более подробно о подключении и отключении базы можно узнать в *разд. 1.1.2* и *4.12*.

Когда создается новая база данных, она дублирует базу данных model. Любые опции и настройки в базе данных model копируются в новую базу данных. Если вам необходимо создать несколько схожих баз, или все ваши базы данных должны обладать схожими свойствами, то можно внести необходимые изменения в model.

В одной из баз данных, которую мне пришлось однажды оптимизировать, я нашел очень интересное решение. В базе данных model была реализована процедура создания резервной копии и восстановления. Таким образом, все новые базы получали в свое распоряжение готовый код, который достаточно просто подкорректировать, после чего он будет готов использованию.

Несмотря на удобство использования model, я не рекомендую перегружать его возможностями, особенно таблицами с данными. Если эти таблицы в новой базе не найдут своего применения, то они лягут на нее лишним грузом, да и время создания базы неоправданно увеличится.

Напоследок сделаем одно замечание. При создании базы данных нельзя указывать только параметры журнала без описания файла данных. Это значит, что следующий запрос завершится ошибкой:

```
CREATE DATABASE TestDatabase2
LOG ON
(
```

```
   NAME = 'TestDatabase_log',
   FILENAME = 'c:\data\log',
   SIZE = 5MB,
   MAXSIZE = 10GB,
   FILEGROWTH = 5%
)
```

В этом случае указание файла данных является необходимым. Как минимум, вы должны указать для файла его логическое и физическое имя. Остальные параметры описания файла не обязательны, и для них будут взяты значения по умолчанию:

```
CREATE DATABASE TestDatabase2
ON
(
   NAME = 'TestDatabase',
   FILENAME = 'c:\data\data',
)
LOG ON
(
   NAME = 'TestDatabase_log',
   FILENAME = 'c:\data\log',
   SIZE = 5MB,
   MAXSIZE = 10GB,
   FILEGROWTH = 5%
)
```

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter1\create_database.sql на компакт-диске, прилагаемом к книге.

1.1.1. Файловые группы

Если ваш сервер содержит несколько дисков, вы можете расположить определенные объекты и файлы на индивидуальном диске, группируя ваши файлы базы данных в файловые группы. Файловая группа — это несколько файлов, сгруппированных под определенным именем. Сервер SQL содержит одну файловую группу, которая по умолчанию называется `PRIMARY`. Вы можете создавать дополнительные файловые группы во время и после создания базы данных.

С помощью файловой группы вы можете располагать определенные объекты в определенных файлах, которые могут находиться на разных дисках, но сервером SQL будут восприниматься как одно целое пространство для хранения данных. Системный администратор может резервировать и восстанавливать индивидуально файлы или файловые группы, что очень удобно при больших размерах базы данных для эффективной стратегии резервирования и копирования.

Распределение файлов на несколько физических дисков позволяет повысить производительность базы данных. У каждого диска свой контроллер, и каждый из них может параллельно производить операции чтения/записи.

Использование файловых групп — это новая и очень мощная технология дизайна баз данных, которую реализовала компания Microsoft. Уже на этапе проектирования вы должны четко представлять себе структуру вашей базы, данные, которые она будет хранить, а также транзакции и запросы к ней. Это позволит вам оптимально сгруппировать файлы и индексы.

Файлы журнала не являются частью файловых групп, и управление ими происходит отдельно от файлов данных. В группы можно объединять только файлы данных.

Сервер SQL поддерживает два типа файловых групп:

- основная файловая группа, которая содержит системные таблицы и основные файлы данных;
- файловые группы, определенные пользователем, которые содержат любые файловые группы, указанные с использованием ключевого слова `FILEGROUP`.

Когда вы создаете базу данных, основная файловая группа автоматически становится группой по умолчанию. Файловая группа по умолчанию получает все новые таблицы, индексы и файлы, для которых файловая группа не указана. Если ваша база данных содержит более чем одну файловую группу, рекомендуется изменить группу по умолчанию так, чтобы ею была группа, определенная вами. Это позволит изолировать основную файловую группу, которая содержит системные таблицы, и обезопасить от разрушения из-за пользовательских таблиц за счет того, что пользовательские данные будут попадать в пользовательскую файловую группу.

Если файловая группа по умолчанию становится основной, установите корректный размер. Если в файловой группе закончится свободное пространство, вы не сможете добавлять какую-либо информацию в системные таблицы. Если пользовательская файловая группа не сможет увеличиваться в размере, то добавлять информацию можно будет только в пользовательские таблицы, расположенные в этой группе.

Итак, одна база данных в MS SQL Server может состоять из нескольких файлов данных, файловых групп или журналов. Это очень удобно, когда необходимо распределить нагрузку между несколькими жесткими дисками. Давайте рассмотрим эту возможность и увидим, какие преимущества она дает и как их использовать.

По умолчанию на сервере уже существует файловая группа `PRIMARY`. Все создаваемые базы данных попадают в нее, в том числе и файлы баз данных, которые мы уже создали в этой главе. Рекомендуется, чтобы в этой файловой группе были все системные таблицы SQL-сервера.

Вы можете создать новые файловые группы и в них размещать пользовательские таблицы. Основная файловая группа может быть расположена на одном физическом диске, а группа с пользовательскими таблицами — на другом. Таким образом, два диска могут читать данные параллельно. Только это уже позволит повысить производительность, ведь при обращении к пользовательским данным очень часто необходимо обратиться и к системным таблицам. Это происходит незаметно для пользователя, но если чтение и запись будут происходить параллельно, то сервер сможет обрабатывать больше запросов со стороны пользователя. В наше время жесткий диск является наиболее слабым местом (это единственное устройство, не считая съемных носителей, которое остается механическим, а не электронным).

Помимо этого, сервер позволяет производить резервное копирование определенных файловых групп. Допустим, что у вас в базе данных две таблицы очень большого размера, но одна изменяется часто, а вторая нет. Резервировать обе таблицы с одинаковой частотой нет смысла. Если поместить их в разные файловые группы, то группу с часто изменяемой таблицей можно резервировать хоть каждый час, а таблицу с редко изменяемыми данными можно резервировать по мере изменения данных.

Чтобы еще больше оптимизировать процесс резервирования, можно в одну группу файлов поместить часто изменяемые таблицы и резервировать эту группу с большей частотой, чем ту группу, в которой находятся редко изменяемые таблицы. Более подробно о резервировании мы поговорим в *разд. 4.10*.

Итак, создадим базу данных, в которой будет две файловые группы: обязательная `PRIMARY` (в нее поместим два файла) и пользовательская `FILEGR1` (в ней разместим три файла). Код создания можно увидеть в листинге 1.3.

Листинг 1.3. Создание базы данных с файловыми группами

```
CREATE DATABASE TestDatabase
ON PRIMARY
(
```

```
  NAME = tdb1_dat,
  FILENAME = 'c:\data\tdb1_data.mdf',
  SIZE = 10MB,
  MAXSIZE = 100MB,
  FILEGROWTH = 10MB
 ),
(
  NAME = tdb2_dat,
  FILENAME = 'd:\data\tb2_data.ndf',
  SIZE = 10MB,
  MAXSIZE = 100MB,
  FILEGROWTH = 10MB
 ),
FILEGROUP DBGroup1
(
  NAME = tdb_group1_dat,
  FILENAME = 'e:\data\tdb_gr1_data.ndf',
  SIZE = 10MB,
  FILEGROWTH = 5%
 ),
(
  NAME = tdb_group2_dat,
  FILENAME = 'f:\data\tdb_gr2_data.ndf',
  SIZE = 10MB,
  FILEGROWTH = 5%
 ),
(
  NAME = tdb_group3_dat,
  FILENAME = 'g:\data\tdb_gr3_data.ndf',
  SIZE = 10MB,
  FILEGROWTH = 5%
)
```

Перед описанием первого файла мы явно указываем, что он будет создан в основной файловой группе (`ON PRIMARY`). После этого в круглых скобках через запятую идет описание двух файлов. Потом создается новая файловая группа `DBGroup1` с помощью оператора `FILEGROUP`:

```
FILEGROUP DBGroup1
```

И теперь идет описание файлов для этой файловой группы. Обратите внимание, что каждый файл находится на своем жестком диске. Это позволит добиться параллельного чтения и записи в каждый файл. На рис. 1.1 вы видите окно свойств созданной базы данных с несколькими файлами (окно программы Enterprise Manager). В последней колонке списка показана файловая группа файла данных. Из рисунка видно, что все файлы находятся на одном диске, но тому есть причина. Просто у меня в компьютере нет необходимого количества разделов, потому что это не сервер, а простой ноутбук. Конечно, на рабочих серверах лучше устанавливать несколько дисков и распределять между ними нагрузку.

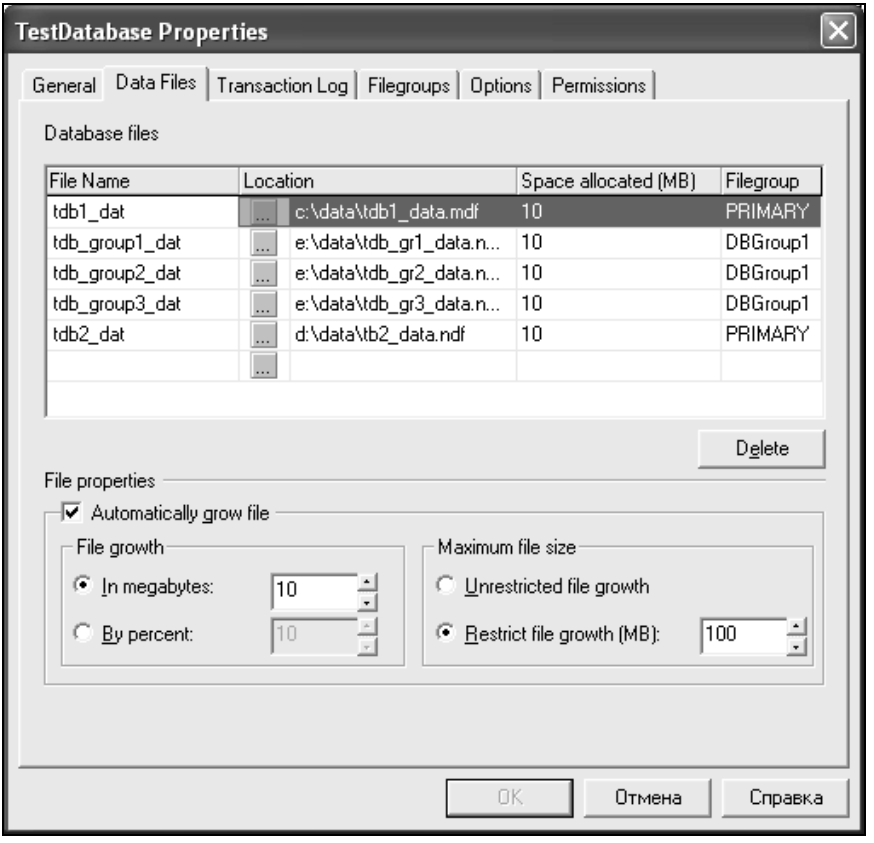


Рис. 1.1. Свойства базы данных из пяти файлов и двух файловых групп

Напоминаю, что желательно изменить файловую группу, установленную по умолчанию. Если ваша база данных имеет много файловых групп для разных типов данных, назначьте одну пользовательскую в качестве группы по умолчанию.

1.1.2. Подключение базы данных

Когда подключение удобнее и даже выгоднее резервирования? Недавно я столкнулся с такой ситуацией, что настройки репликации не попадают в резервную копию. Это оказалось серьезной проблемой. У фирмы, где я работаю, есть два офиса, которые находятся в разных концах города и не соединены через Интернет. Чтобы производить репликацию, базу данных удаленного офиса нужно перевозить в главный офис, но с помощью резервной копии это сделать не удается: база данных восстанавливается, как будто никаких настроек репликации не было.

Первое, что приходит в голову после такого разочарования, — таскать жесткий диск или весь системный блок для репликации в основной офис. Это, конечно же, не очень удачный выход. И вот однажды мы попытались отключить базу данных на удаленном сервере, привезти в центральный офис и подключить. Настройки репликации сохранились!

Для отключения базы данных используется системная процедура `sp_detach_db`. С процедурами мы пока еще не знакомы, это тема *главы 3*, поэтому пока просто выполните следующую команду для отключения:

```
EXEC sp_detach_db 'TestDatabase', 'true'
```

Про отключение баз данных мы поговорим в *разд. 4.12*. Там же процедура `sp_detach_db` будет рассмотрена более подробно.

Посмотрим, как можно подключить файл базы данных к серверу с помощью оператора `CREATE DATABASE`:

```
CREATE DATABASE Archive
ON PRIMARY (FILENAME = 'c:\data\filename.mdf')
FOR ATTACH
```

В первой строке стоит оператор `CREATE DATABASE` и имя подключаемой базы данных. Ключевое слово `ON PRIMARY` означает создание в основной файловой группе. После этого в круглых скобках указан путь к существующему файлу данных. И в последней строке написано `FOR ATTACH`, т. е. для подключения.

Обратите внимание, что имя подключаемой базы отличается от имени базы, которую мы отключали. Раньше имя было `TestDatabase`, а после подключения оно превратилось в `Archive`. Таким образом, мы смогли переименовать уже существующую базу данных. С помощью оператора `ALTER DATABASE`, который используется для редактирования параметров базы *(см. разд. 1.3)*, переименование невозможно.

1.1.3. Сопоставление

У нас остался еще один параметр, который мы не рассмотрели, — это `COLLATE`, который устанавливает в базе данных свойство `Collation` (сопоставление). Что это такое? У начинающих администраторов он вызывает страх, а у опытных — уважение. С помощью `Collation` можно указать раскладку (кодировку), которая будет использоваться по умолчанию для конкретной базы данных. В MS SQL Server существуют три способа задать раскладку (кодировку).

- Для каждого поля в отдельности. Вы можете указать кодировку конкретного поля таблицы. Если у поля не указана кодировка, то будет взято значение, указанное для всей базы данных.
- Для базы данных в целом. Указанная кодировка для базы данных будет использоваться по умолчанию для всех полей таблиц, если не указано иное. Если при создании базы параметр `COLLATE` не задан, то будет использоваться значение по умолчанию, указанное для сервера баз данных в целом.
- Глобальная кодировка. Это значение задается во время установки MS SQL Server и по умолчанию устанавливается в соответствии с локальными настройками ОС сервера.

Самый простой вызов команды:

```
CREATE DATABASE TestDatabase1
COLLATE Cyrillic_General_CI_AS
```

Описание кодировки ставится после описания файлов базы данных и журнала. То есть если вы описывали файлы, то параметр `COLLATE` должен быть в самом конце, как показано в листинге 1.4.

Листинг 1.4. Создание базы данных с описанием параметров файлов и кодировки

```
CREATE DATABASE TestDatabase2
ON
(
  NAME = TestDatabase_data,
  FILENAME = 'c:\data\test.mdf',
  SIZE = 10MB,
  MAXSIZE = 100GB,
  FILEGROWTH = 5MB
)
```

```
LOG ON
(
  NAME = 'TestDatabase_log',
  FILENAME = 'c:\data\test.ldf',
  SIZE = 5MB,
  MAXSIZE = 10GB,
  FILEGROWTH = 5%
)
COLLATE Cyrillic_General_CI_AS
```

Чтобы определить возможные значения раскладки, можно выполнить следующий запрос:

```
SELECT *
FROM ::fn_helpcollations()
```

Сейчас мы не будем останавливаться на том, что здесь происходит, потому что запросы `SELECT` — это тема *главы 2*. Замечу только, что в результате на экране вы увидите таблицу из двух колонок: `name` (имя) и `description` (описание).

Определенная раскладка может быть назначена только текстовым полям. Это вполне логично, ведь числовые поля и даты содержат только числа, которые для любого национального языка одинаковы.

1.2. Создание таблиц

В основном мы будем рассматривать различные возможности базы данных в естественном порядке — создание, удаление и редактирование. Но иногда имеет смысл отступить от этой последовательности, как сейчас. Мы пока умеем только создавать и удалять базу данных, а теперь перейдем к рассмотрению создания таблиц. Редактирование баз обсудим чуть позже *(см. разд. 1.3)*.

Когда мы создали базу данных, можно переходить к созданию таблиц, которые непосредственно будут содержать данные. Если сравнить базу данных с файловой системой, то ее можно рассматривать как папку, а таблицы — как файлы в этой папке. Только базы данных не поддерживают вложенности, т. е. в одной базе не может быть другой базы данных.

База данных сама по себе не несет никакой пользы, как и папка без файлов, потому что ни база, ни папка не могут хранить данные самостоятельно. Данные хранятся в таблицах (файлах), и именно их мы научимся создавать в этом разделе.

Что такое таблица в базе данных MS SQL Server? Данный сервер работает с реляционными таблицами, которые представляют собой двухмерный массив данных, в котором колонка определяет значение, а строки — это данные. Например, таблица базы данных схожа с классическими таблицами (списками), которые мы строим в программе Excel или на бумаге в рефератах и курсовых работах.

Рассмотрим пример таблицы. Допустим, что нам надо хранить в таблице список жителей дома. Не задумываясь о базах данных, а просто на бумаге для решения поставленной задачи нарисуем табл. 1.1.

Таблица 1.1. *Список жителей дома*

Фамилия	**Имя**	**Отчество**	**Дата рождения**	**Адрес**
Иванов	Иван	Иванович	1.1.1950	Шаболовка 37
Петров	Петр	Петрович	4.7.1967	Королева 12
...	...	...	...	...

В *предисловии* мы уже кратко говорили об уникальности, а сейчас нам предстоит обсудить это более подробно. Исходя из реляционной модели, все строки в базе данных должны быть уникальными. Это значит, что в списке не может быть двух записей для одного и того же жителя дома. Некоторые могут сказать, что это не проблема, потому что очень мала вероятность того, что в одном доме будут жить два человека с одними и теми же данными в полях `Фамилия`, `Имя`, `Отчество` и `Дата рождения`. Вероятность маленькая, но она же есть! А ведь существует еще вероятность ошибки пользователя программы, который может дважды ввести данные одного и того же человека. Вот это уже действительно станет проблемой.

Чтобы понять, какие проблемы могут возникнуть при появлении в таблице одинаковых записей, представим, что у нас в списке 10 человек с одинаковыми фамилией, именем, отчеством и датой рождения (адрес не учитываем). Такой пример вполне реален. Если взять всех жителей России, то количество людей с фамилией Иванов будет не просто большим, а огромным. Совпадение остальных параметров также возможно.

Теперь представим, что у одной из записей фамилия введена неверно, и запись случайно совпадает с 10 другими записями. Какую из 10 одинаковых строк должен изменить сервер? Любую, потому что записи одинаковые? Это не очень корректный выход, поэтому в стандарте SQL-92 сказано, что в реляционной таблице не может быть двух абсолютно одинаковых записей, хотя

многие базы данных позволяют нарушить это ограничение, и MS SQL Server не является исключением.

Чтобы обеспечить уникальность, было введено понятие *ключевое поле*. Это одно или несколько полей, которые обеспечивают уникальность строк в таблице. В нашем случае ключевыми можно сделать фамилию, имя, отчество и дату рождения. Тогда при попытке добавить новую строку, в которой значения этих полей будут сопадать с любой из уже существующих строк таблицы, база данных возвратит сообщение об ошибке и не сможет добавить такую строку. Получается, что с помощью ключевых полей база данных обеспечивает уникальность, и по ним гарантированно можно найти одну и только одну запись. То есть, запросив из базы данных строку с определенными фамилией, именем, отчеством и датой рождения, мы получим только одну запись.

Некоторые базы данных позволяют создавать таблицы без ключевых полей, но я не рекомендую так поступать без особой надобности. Всегда создавайте ключевое поле. В MS SQL Server для этого можно использовать автоматически увеличиваемое поле, которое не требует дополнительных расходов на сопровождение.

В настоящее время Microsoft не рекомендует использовать в качестве ключа автоматически увеличиваемые поля. Да, эти поля достаточно просты в сопровождении, но не всегда эффективны, потому что автоматически увеличиваемое поле нельзя редактировать. Сейчас рекомендуется для ключа создавать поле типа GUID (Global Unique Identifier, глобально уникальный идентификатор). Значения полей этого типа можно редактировать и достаточно просто сгенерировать самостоятельно.

Пессимисты считают, что GUID не могут быть идеальным решением проблемы ключевого поля, потому что значение генерируется, и может возникнуть ситуация, когда будут сгенерированы два одинаковых значения. Оптимисты утверждают, что даже если все компьютеры в мире будут одновременно генерировать GUID, вероятность получения двух одинаковых значений стремится к нулю.

В оптимистическое утверждение мне верится с трудом, ведь количество возможных GUID ограничено, и после генерации хотя бы трети этих значений вероятность совпадения увеличивается в два раза. Но к пессимистам меня тоже отнести нельзя, потому что возможных вариантов GUID настолько много, что хватит даже для очень большой базы данных, с которой вы, скорее всего, никогда не встретитесь в своей практике. Поэтому я использую GUID-поля (и пока ни разу не сталкивался с конфликтами GUID), о которых мы еще поговорим в *разд. 2.24*. А в этом разделе мы будем, в основном, созда-

вать таблицы с ключевым полем в виде автоматически увеличиваемого поля, что намного удобнее при изучении языка SQL.

Теперь посмотрим, как сервер хранит данные в строке. Иногда возникает необходимость хотя бы приблизительно рассчитать размер строки, а лучше всегда точно знать его. Это позволит вам более эффективно управлять размером базы данных.

Строка данных состоит из строки заголовка и порции данных. Заголовок содержит 4 байта информации о поле строки данных (указатель на расположение конца фиксированной порции данных строки и размер переменной, используемой в строке).

Поле может содержать следующие элементы:

- данные фиксированной длины (`char`, `int` и т. д.) — помещаются в страницу перед данными переменной длины;
- нулевые блоки (`NULL`) — это набор байтов переменной длины;
- переменные блоки (`varchar`) — содержат два байта, описывающих длину колонки. Дополнительные два байта на колонку указывают на конец каждой колонки переменной длины;
- данные переменной длины (`text`, `image`) — хранятся в странице после переменных блоков.

Старайтесь делать строки компактными (минимизировать общий размер всех полей), чтобы поместить больше строк в страницу. Это позволит уменьшить количество операций ввода-вывода и улучшить работу буфера. Для того чтобы строка была компактной, выделяйте каждому полю только необходимый объем памяти. Например, для хранения фамилии вполне достаточно 20 символов строки переменной длины типа `varchar`, и не имеет смысла выделять поле типа `char` фиксированной длины в 50 символов.

Данные типа `char` имеют фиксированную длину, и в странице данных на них всегда отводится определенный объем памяти (в нашем случае 50 байт). Данные типа `varchar` будут занимать ровно столько памяти, сколько необходимо. Например, вы объявили поле типа `varchar` размером 50 символов, но записали в него только 10. В странице данных на диске будет выделено (помимо заголовка) только 10 байт и ни байта больше.

Данные переменной длины (`text`, `ntext` и `image`) могут храниться как одна коллекция страниц или данные в строке. Рассмотрим эти типы:

- `text` — может хранить 2 147 483 647 символов. Текстовые данные в кодировке, отличной от Unicode, не могут использоваться в качестве переменных или параметров в хранимых на сервере процедурах;

- `ntext` — может хранить 1 073 741 823 символа в кодировке Unicode, где каждый символ определяется двумя байтами;
- `image` — может хранить 2 147 483 647 байт бинарных данных.

Так как данные типов `text`, `ntext` и `image` обычно довольно объемистые, SQL Server хранит их вне строк. В строке содержится только шестнадцатибитный указатель на данные. Если данные этих типов хранят небольшой объем информации, то вы можете включить опцию `text in row`. Вы можете также указать предел от 24 до 7000 байтов. Эту опцию для таблицы можно включить, используя системную процедуру `sp_tableoption`.

В следующем примере с помощью процедуры `sp_tableoption` устанавливается опция `text in row` и указывается, что в строке будет храниться до 1000 символов:

```
EXEC sp_tableoption N'TestDatabase', 'text in row', '1000'
```

Вам пока может быть не очень понятна эта команда, потому что хранимые процедуры мы еще не изучали. Но в *главе 3* мы рассмотрим эту тему более подробно.

При создании таблицы вы должны указать ее имя, имена колонок и типы данных колонок. Имена колонок должны быть уникальными в каждой таблице, но вы можете использовать одно и то же имя в разных таблицах одной базы данных.

MS SQL Server позволяет создавать до 2 биллионов таблиц в базе данных, каждая таблица может содержать до 1024 колонок, и общий размер полей в таблице может достигать 8060 байтов на строку (без учета `text`, `ntext` и `image`). Это действительно очень много, и я пока не слышал, чтобы кому-то требовалось больше. Если вам все же не хватит количества полей, то можно создать две таблицы и связать их "один-к-одному". Для этого используются внешние ключи, о которых мы будем говорить в *разд. 1.2.6*.

1.2.1. Оператор *CREATE TABLE*

С теорией таблиц мы закончили, теперь попробуем эту таблицу рассмотреть с точки зрения базы данных. Имена колонок в списке баз данных называются именами полей. Нам необходимо только создать таблицу в базе данных с такими полями, и все будет готово для хранения списка.

Общий вид оператора `CREATE TABLE` показан в листинге 1.5.

Листинг 1.5. Общий вид оператора CREATE TABLE

```
CREATE TABLE
    [ database_name.[ owner ] . | owner. ] table_name
    ( { < column_definition >
        | column_name AS computed_column_expression
        | < table_constraint > ::= [ CONSTRAINT constraint_name ] }

            | [ { PRIMARY KEY | UNIQUE } [ ,...n ]
    )

[ ON { filegroup | DEFAULT } ]
[ TEXTIMAGE_ON { filegroup | DEFAULT } ]

< column_definition > ::= { column_name data_type }
    [ COLLATE < collation_name > ]
    [ [ DEFAULT constant_expression ]
        | [ IDENTITY [ ( seed , increment ) [ NOT FOR REPLICATION ] ] ]
    ]
    [ ROWGUIDCOL]
    [ < column_constraint > ] [ ...n ]

< column_constraint > ::= [ CONSTRAINT constraint_name ]
    { [ NULL | NOT NULL ]
        | [ { PRIMARY KEY | UNIQUE }
            [ CLUSTERED | NONCLUSTERED ]
            [ WITH FILLFACTOR = fillfactor ]
            [ON {filegroup | DEFAULT} ] ]
        ]
        | [ [ FOREIGN KEY ]
            REFERENCES ref_table [ ( ref_column ) ]
            [ ON DELETE { CASCADE | NO ACTION } ]
            [ ON UPDATE { CASCADE | NO ACTION } ]
            [ NOT FOR REPLICATION ]
        ]
        | CHECK [ NOT FOR REPLICATION ]
        ( logical_expression )
    }
```

```
< table_constraint > ::= [ CONSTRAINT constraint_name ]
    { [ { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
        { ( column [ ASC | DESC ] [ ,...n ] ) }
        [ WITH FILLFACTOR = fillfactor ]
        [ ON { filegroup | DEFAULT } ]
    ]
    | FOREIGN KEY
        [ ( column [ ,...n ] ) ]
        REFERENCES ref_table [ ( ref_column [ ,...n ] ) ]
        [ ON DELETE { CASCADE | NO ACTION } ]
        [ ON UPDATE { CASCADE | NO ACTION } ]
        [ NOT FOR REPLICATION ]
    | CHECK [ NOT FOR REPLICATION ]
        ( search_conditions )
    }
```

Оператор `CREATE TABLE` довольно сложен, потому что включает в себя не только создание самой таблицы, но и описание полей, ограничений, первичных и внешних ключей. Поэтому рассмотрим оператор `CREATE TABLE` по частям.

Прежде всего создадим базу данных, в которой будет проходить тестирование SQL-запросов. Все предыдущие тестовые базы данных можно удалить и создать новую, чистую базу данных. Для этого я создал базу данных с именем `TestDatabase` и параметрами по умолчанию.

Теперь посмотрим, как создать таблицу и удалить ее. Подключитесь к своей тестовой базе данных (`TestDatabase`) и выполните следующее действие:

```
CREATE TABLE TestTable
(
 id int
)
```

Невозможно создавать пустую таблицу, без описания полей. Хотя бы одно поле должно быть. В данном примере создается таблица с именем `TestTable`. После имени в скобках указываются имена и типы полей в формате:

```
имя тип параметры
```

Обязательными являются только имя и тип, как в примере выше, а параметры могут отсутствовать. В этом случае поле не будет обладать никакими дополнительными параметрами.

Если полей несколько, то они перечисляются в скобках через запятую. В данном случае описано только одно поле с именем `id` типа `int` и без параметров. Какие могут быть типы данных? Стандарт SQL описывает только основные типы: строка, число и т. д., но в MS SQL Server их намного больше. Все типы данных, которые поддерживает SQL Server 2000, перечислены в *приложении 1*.

Для удаления таблицы используется оператор:

```
DROP TABLE имя
```

где `имя` — это имя удаляемой таблицы. При этом вы должны быть подключены к серверу, таблицу которого вы удаляете, т. е. удаление происходит в текущей базе данных.

Следующий запрос показывает, как удалить таблицу `TestTable`, созданную нами ранее:

```
DROP TABLE TestTable
```

Если в создаваемой таблице несколько полей, то их необходимо указывать в круглых скобках через запятую. Например:

```
CREATE TABLE TestTable
(
 id int,
 vcName varchar(30),
 vcLastName varchar(30),
 vcSurName varchar(30),
 cPol char(1),
 dBirthDay datetime
)
```

В качестве дополнительных параметров можно указывать, например, явную кодировку (Collation) для поля. В следующем примере создается таблица с двумя текстовыми полями. В первом поле для хранения фамилии кодировка не указана, поэтому для него будет взято значение из базы данных, а у второго поля явно указываем латинскую кодировку:

```
CREATE TABLE TestTable
(
   Famil varchar(10),
   Name varchar(10) collate latin1_general_cs_as
)
```

При создании таблицы можно указать файловую группу, в которой должны храниться данные этой таблицы. Имя группы указывается после ключевого

слова `ON` за объявлением всех полей. В следующем примере создается таблица, данные которой будут помещены в группу с именем `group1`:

```
CREATE TABLE TestTable
(
   Famil varchar(10),
   Name varchar(10) collate latin1_general_cs_as
) ON group1
```

Если файловая группа не указана, то таблица создается в группе по умолчанию. Если вы создали пользовательскую группу, но она не является группой по умолчанию, то обязательно указывайте ее явно.

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter1\create_table.sql на компакт-диске, прилагаемом к книге.

1.2.2. Автоматическое увеличение

Очень часто требуется какое-либо поле-счетчик, значение которого будет автоматически увеличиваться без вмешательства пользователя. Например, таким образом часто создаются ключевые поля, и об этом методе мы говорили в *разд. 1.2*. Опытные программисты, работающие с базами данных уже не первый день, с помощью таких полей добиваются уникальности данных. Например, очень часто можно видеть таблицы, в которых первое поле называется `id` (я люблю именовать их как `Key1` или `idИмяТаблицы`), автоматически увеличивающее при добавлении новых строк. Таким образом, можно сказать, что это поле всегда уникально.

Вы можете использовать свойство `Identity` (тождество) для создания колонки, которая содержит сгенерированное системой следующее значение, определяющее каждую строку, вставленную в таблицу. Тождественная колонка может использоваться не только для хранения значения первичного ключа, но и для определения строки, которая была вставлена в таблицу последней, ведь каждая новая строка будет получать число, которое больше любого существующего.

Когда SQL Server автоматически предоставляет значение ключа, уменьшаются затраты, потому что облегчается программирование, первичный ключ получается короче и уменьшаются пользовательские транзакции.

Автоматически увеличиваемое поле имеет следующие ограничения:

- в таблице может быть только одна колонка `Identity`, да я и не вижу смысла в создании двух автоматически увеличиваемых полей;

- эта колонка должна использоваться с целочисленными типами данных, дробные типы запрещены;
- значение в этом поле не может обновляться. В MS SQL Server есть способ обойти такое ограничение (мы его увидим далее), но его применение может нарушить целостность данных, поэтому необходимо использовать его аккуратно и только в крайних случаях;
- не разрешаются нулевые значения. Колонка всегда должна быть заполнена, и при этом значение заносится только сервером, вы не можете на него повлиять.

При разработке запросов вы можете использовать ключевое слово `IDENTITYCOL` в месте с именем колонки в запросе. Это позволяет вам ссылаться на колонку в таблице со свойством `Identity`, не зная имени колонки. Так как в таблице может быть только одна колонка со свойством `Identity`, сервер сам сможет найти нужное поле.

Вы можете управлять свойством `Identity` несколькими способами.

- Тождественное свойство не обеспечивает уникальность. Для уникальности автоматически увеличиваемого поля создавайте уникальный индекс. Да, он создается по умолчанию, но удалять его не стоит, иначе сервер не сможет гарантировать уникальность.
- Автоматическое увеличение можно устанавливать на поля следующих типов: `tinyint`, `smallint`, `int`, `bigint`, `decimal(n,0)` или `numeric(n,0)`. Обратите внимание, что у типов `decimal` и `numeric` в качестве значения точности (количество знаков после запятой) следует указать 0, т. е. дробной части не должно быть. Вы должны также учитывать, что количество возможных записей в таблице зависит от выбранного типа. Это значит, если выбрать тип `tinyint` (может принимать значения от 0 до 255), то в таблице может быть только 256 записей (если отсчет начать с нуля). Дальше увеличение будет невозможно, а значит, новая запись не может быть добавлена. Выбирайте тот тип, которого будет достаточно.

В современных базах данных для обеспечения уникальности рекомендуется использовать глобально уникальные идентификаторы GUID (Globally Unique Identifier). Это действительно более мощный способ обеспечения целостности, но не очень удобный с практической точки зрения, поэтому программисты старой закалки продолжают использовать автоматическое увеличение.

Некоторые базы данных создают GUID-идентификаторы для каждой строки, даже если пользователь не требовал ничего подобного. При этом идентификатор прячется, и для его отображения необходимо включать специальную опцию или использовать специальную команду в SQL-запросе. В MS SQL

Server идентификаторы GUID создаются автоматически, если таблица участвует в репликации.

В таблице может быть только одна колонка с автоматически увеличиваемым значением. Чтобы задать такой тип поля, необходимо написать:

```
IDENTITY [ ( seed , increment ) [ NOT FOR REPLICATION ] ]
```

После слова `IDENTITY` нужно в круглых скобках указать два параметра — начальное значение (будет присвоено первой строке в таблице) и приращение. По умолчанию обоим параметрам присваивается значение 1. Следующий пример показывает создание таблицы с автоматически увеличиваемым полем `id`:

```
CREATE TABLE TestTable
(
 id int IDENTITY(100, 2)
)
```

В качестве начального значения выбрано число 100. При добавлении каждой последующей строки в поле будет записываться число, на 2 большее предыдущего, т. е. 102, 104, 106, 108 и т. д.

Для чего нужен ключ `NOT FOR REPLICATION` (не для репликации)? Допустим, что у вас есть два сервера баз данных, которые расположены в разных офисах и не объединены каналом связи. Необходимо иметь какую-то возможность объединения баз данных в одно целое. В MS SQL Server есть удобный инструмент решения данной проблемы — репликация. Репликация — это процесс слияния двух баз данных.

Но тут существует одна проблема. Допустим, у вас есть две базы данных: DB1 и DB2. В базе данных DB1 может быть добавлена новая строка с определенным идентификатором, который был назначен автоматически. Во время репликации эта строка должна попасть в базу данных DB2, но при вставке средствами репликации в поле `id` будет сгенерировано другое значение. Строки получатся неодинаковыми, и нарушится целостность данных.

Чтобы решить проблему, для поля с автоматически увеличиваемым значением можно указать ключ `NOT FOR REPLICATION`. В этом случае при вставке новой строки из DB1 в DB2 во время репликации новый `id` не будет генерироваться, а будет использоваться тот, который был указан в DB1.

Но это не решает проблему полностью, потому что идентификатор строки из DB1 может конфликтовать с уже существующей строкой в DB2, что недопустимо. Есть очень простой способ справиться с этим. В DB1 для поля `id` задаем начальное значение 1, а для DB2 задаем начальное значение 100 000 000.

Теперь новые строки в DB1 будут иметь номера 1, 2, 3, а в DB2 на 100 000 000 больше, и пересечение в идентификаторах произойдет нескоро, а точнее, когда в DB1 будет добавлено 100 000 000 строк.

В общей базе данных у вас будут строки с идентификаторами из обеих таблиц, и по номеру вы можете легко определить, из какой базы данных он взят. Если поле `id` более 100 000 000, то, очевидно, перед вами DB2.

Необходимо сделать одно замечание (хотя оно и выходит за рамки книги) — при настройке репликации убедитесь, что обе базы данных имеют одинаковую структуру, а в мастере укажите, что структуру базы данных реплицировать не надо. Если вы этого не сделаете, то сразу после выполнения мастера структура базы данных DB1 будет скопирована в DB2, и все настройки идентификаторов и начальных значений уничтожатся.

Итак, рассмотрим пример создания таблицы с автоматически увеличиваемым идентификатором и не участвующим в репликации:

```
CREATE TABLE TestTable1
(
 id int IDENTITY(100, 2) NOT FOR REPLICATION
)
```

Если вы используете автоматически увеличиваемое поле только для того, чтобы создать первичный ключ, то это не совсем корректное решение для базы данных, с которой работает множество человек. Дело в том, что в языках программирования строка добавляется на стороне клиента. В этот момент в добавляемой строке значение автоматически увеличиваемого поля будет нулевым, и осмысленное значение появится в этом поле только после завершения ввода данных и выполнения оператора `INSERT` (мы его рассмотрим в *главе 2*). Очень часто это неудобно, и в таком случае лучшим вариантом будет использование глобально уникальных идентификаторов GUID.

Если после ключевого слова `IDENTITY` ничего не указано, то подразумевается, что начальное значение и увеличение равны 1. Рассмотрим запрос:

```
CREATE TABLE TestTable2
(
 id int IDENTITY
)
```

Он идентичен следующему:

```
CREATE TABLE TestTable2
(
 id int IDENTITY(1,1)
)
```

Начальное значение и приращение, равные единице, встречаются, наверное, в 90% случаев, и именно поэтому единица используется в качестве значения по умолчанию для `IDENTITY`-полей.

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter1\identity.sql на компакт-диске, прилагаемом к книге.

1.2.3. Значения по умолчанию

Бывают случаи, когда необходимо упростить ввод данных со стороны пользователя или просто заносить в какое-либо поле значение без вмешательства пользователя. Например, вы можете вести таблицу финансовых расходов (в программе домашней бухгалтерии). Для этого нужны поля `дата`, `название` и `сумма`. Если дата не указана, то в поле для хранения даты должна записываться текущая дата. Это очень удобно и позволяет пользователю не вводить дату расходов, сделанных в тот же день, что и занесение информации в таблицу. Указание даты будет необходимо, только если расходы, планируемые на будущий период или сделанные ранее, не занесены в таблицу.

Для задания значения по умолчанию используется ключевое слово `DEFAULT`, после которого идет нужное значение в том же формате, что и поле. Иными словами, если поле числовое, то значение по умолчанию должно быть числом. Если поле строковое, то значение должно быть строкой, заключенной в одинарные кавычки.

Значения по умолчанию не могут быть назначены полям типа `timestamp` и автоматически увеличиваемым числовым полям. В качестве значения можно указывать конкретное значение, функцию SQL-сервера, значения `NULL` или функции SQL-92. В табл. 1.2 перечислены функции SQL-92, но кроме них можно указывать и функции SQL-сервера, которых очень много, и рассмотреть их в одной таблице или главе просто невозможно.

Таблица 1.2. *Функции SQL-92*

Функция	Возвращаемое значение
`CURRENT_TIMESTAMP`	Текущая дата и время
`CURRENT_USER`	Имя пользователя, добавившего строку
`SESSION_USER`	Имя пользователя, добавившего строку
`SYSTEM_USER`	Имя пользователя, добавившего строку
`USER`	Имя пользователя, добавившего строку

Теперь создадим таблицу, содержащую три поля:

- `id` — целое число со значением по умолчанию, равным 1;
- `dDate` — поле типа дата со значением по умолчанию, равным текущей дате;
- `vcName` — поле строкового типа со значением по умолчанию, равным `'M'`.

Все это в виде сценария будет выглядеть следующим образом:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1,
 dDate datetime DEFAULT (getdate()),
 vcName varchar(50) DEFAULT 'M'
)
```

Для поля `dDate` в качестве значения по умолчанию указывается функция SQL-сервера `getdate()`. Она возвращает текущую дату. К функциям сервера мы еще вернемся чуть позже и не раз, поэтому пока не будем заострять на этом особое внимание. Если сейчас что-то непонятно, все встанет на свои места в *главе 3*.

Если в вашей программе в качестве ключа или для других целей используется глобально уникальный идентификатор GUID, то его значение пользователь уж точно не сможет создать вручную, по крайней мере, сделать это правильно. Поэтому вполне логично как-нибудь генерировать его. Например, создадим таблицу из двух полей: идентификатора GUID и названия.

```
CREATE TABLE TestTable
(
 guid uniqueidentifier DEFAULT NEWID(),
 vcName varchar(50),
)
```

Первое поле `guid` в качестве значения по умолчанию будет получать результат выполнения функции `NEWID()`, которая как раз и генерирует глобально уникальный идентификатор на стороне сервера в тех случаях, если пользователь не указал значение или клиентская программа не сгенерировала его самостоятельно.

Значения по умолчанию позволяют упростить пользователям жизнь, и не стоит пренебрегать ими. Например, у вас есть таблица с информацией о жителях города, в которой хранятся фамилия, имя, отчество, пол и дата рождения. В поле для хранения пола должно быть написано `M` или `Ж`, а так как примерно половина из жителей будет мужчинами, можно упростить пользователю ввод

данных. Если пол не указан, то по умолчанию будет в таблице указываться пол `M`. Таким образом, пол необходимо будет указывать только для женщин.

С другой стороны, из-за значений по умолчанию могут возникнуть ошибки, которые очень сложно определить и исправить. Рассмотрим все тот же пример с полом. А что, если пользователь случайно не укажет пол для женщины? В этом случае будет указана буква `M`, и найти запись с неправильным полом будет сложно. Если бы не было никакого значения, то достаточно было бы найти все записи в таблице с нулевым значением в поле `Пол` и визуально (по имени и фамилии) определить пол и исправить.

Напоследок необходимо отметить, что значение по умолчанию будет подставляться только при добавлении строки. Если в уже существующей строке таблицы полю со значением по умолчанию установить нулевое значение, то поле станет равным `NULL`.

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter1\create_table_defaults.sql на компакт-диске, прилагаемом к книге.

1.2.4. Ограничения

Задача любого администратора и программиста — совместно обеспечить целостность и корректность данных. Например, если пользователь не укажет в одной из строк в списке жителей дома фамилию, то корректность данных нарушится. Человека очень сложно найти, если неизвестна его фамилия.

А если в наш список добавить поле для хранения пола? Что можно записать в такое поле? Конечно же, достаточно одной буквы "М" или "Ж", другого пола пока не придумали. А если пользователь случайно введет букву "Ь"? Это вполне реальная ситуация. Посмотрите на клавиатуру, и вы увидите, что на ней две буквы "М": одна латинская, другая русская. На клавише с латинской буквой "М" находится "Ь". Начинающие пользователи нередко путаются и, если не проверят результат своего ввода, то могут нарушить корректность данных.

Нередко бывают ситуации, когда вместо русской буквы "М" пользователь вводит латинскую. В этом случае в базе получается три пола: "М" русская, "М" латинская и "Ж". Конечно, программно эта ситуация решается довольно просто, достаточно только при определении мужчин выбирать из базы все записи с буквой "М" в любой раскладке, но качество такой базы будет далеко от идеала.

Все эти проблемы известны уже давно, и для их решения придумали ограничения — универсальное средство, с помощью которого задаются правила, которым должны удовлетворять данные при записи в поле. Если записываемое значение не удовлетворяет ограничениям, назначенным полю, то запись завершается ошибкой. Таким образом, сервер сам будет контролировать целостность данных, вводимых пользователем.

Самое простейшее ограничение — это разрешение или запрещение введения нулевых значений (`NULL`). Тут нужно отметить, что `NULL` и пустая строка — это совершенно разные вещи. Об этом мы еще поговорим в *главе 2*. Итак, для таблицы, в которой хранятся фамилия, имя, отчество, пол и дата рождения, ненулевыми должны быть фамилия, имя и пол. Эти параметры есть у каждого жителя нашей страны, поэтому в базе данных обязательно должно быть что-то указано, иначе будет сгенерирована ошибка и сервер не позволит записать такую строку.

А вот с отчеством и датой рождения бывают проблемы, поэтому отсутствие этих параметров не будет считаться ошибкой.

Следующий пример показывает, как создаются поля, в которых запрещаются пустые значения:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 dDate datetime DEFAULT (getdate()) NULL,
 vcName varchar(50) NOT NULL
)
```

В данном случае указано, что поля `id` и `vcName` не могут содержать нулевые значения. Пользователь обязательно должен задать хоть что-нибудь, иначе изменения не будут приняты и сервер сгенерирует ошибку. А вот поле `dDate` может содержать значение `NULL`.

Что некорректно в этом примере? Впрочем, некорректного ничего нет, но, на первый взгляд, не все логично. Для поля `dDate` мы разрешили нулевое значение, но это бессмысленно. Дело в том, что у этого поля есть значение по умолчанию, т. е. если будет произведена попытка записать в поле нулевое значение, оно будет автоматически заменено на значение по умолчанию. Получается, что указывать `NULL` бессмысленно, если для поля установлено значение по умолчанию, и нулевого значения просто не будет.

Но теперь вспомним, когда генерируется значение по умолчанию — только при вставке строки, а не при обновлении. Вы уловили ход мысли? Если полю назначено значение по умолчанию и при этом запрещено нулевое значение,

то ноль нельзя будет установить даже у существующей строки. Именно поэтому я сказал, что не все логично только на первый взгляд. Никогда не забывайте, в какой момент устанавливается значение по умолчанию.

Ограничения `NULL` и `NOT NULL` являются нежесткими, и некоторые специалисты даже не относят их к ограничениям, хотя по своей сути они таковыми и являются. Более жесткие ограничения задаются оператором `CHECK`. Рассмотрим этот оператор на примере. Допустим, что нам нужно создать список хозяев квартир. Для этого нам понадобится ключевое поле (`id`), имя хозяина (`vcName`) и номер квартиры (`iApartment`). Для квартиры вполне логичным будет сделать ограничение ввода от 1 до 1000. Квартир с отрицательными номерами и нулевыми значениями не бывает, да и более 1000 вряд ли встретится. Поэтому логично будет запретить ввод явно некорректных данных с помощью ограничения. Итак, рассмотрим следующий запрос создания таблицы:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 vcName varchar(50) NOT NULL,
 iApartment int CHECK
    (iApartment>0 and iApartment<1000)
)
```

После указания имени и типа поля `iApartment` идет ключевое слово `CHECK`, затем в круглых скобках перечисляются ограничения для данного поля. В нашем примере ограничение задано так:

```
iApartment>0 and iApartment<1000
```

Это значит, что значение в поле `iApartment` должно быть более 0 и в то же время меньше 1000. Объединение происходит с помощью ключевого слова `and`, т. е. "и". Это значит, что оба условия должны быть выполнены. Бывают случаи, когда достаточно, чтобы хотя бы одно из условий было выполнено, тогда их объединяют оператором `or`, т. е. "или". Например, нужно, чтобы в поле заносилось значение или меньше нуля, или больше 1000. Значения в промежутке от 1 до 1000 указывать нельзя. В этом случае сравнение выглядело бы так:

```
iApartment<0 or iApartment>1000
```

Попробуем вставить в таблицу квартир данные. Для этого выполним три запроса. Первый будет вставлять корректные данные:

```
INSERT INTO TestTable(vcName, iApartment)
VALUES('Корректная запись', 10)
```

Не будем вдаваться в подробности работы запроса, это нам предстоит узнать в *главе 2*. Операция пройдет успешно, и никаких ошибок не появится. Следующий запрос будет пытаться вставить запись с номером квартиры 10000, что не соответствует ограничению:

```
INSERT INTO TestTable(vcName, iApartment)
VALUES('Ошибочная запись', 10000)
```

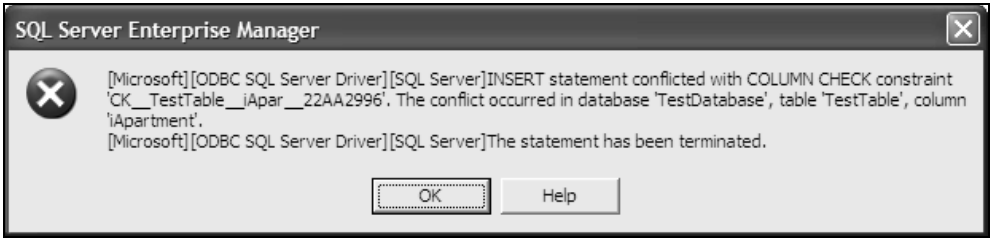


Рис. 1.2. Сообщение о несоответствии строки ограничению

В ответ на это должно появиться окно с сообщением об ошибке (рис. 1.2):

INSERT statement conflicted with COLUMN CHECK constraint 'CK__TestTable__iApar__22AA2996'. The conflict occurred in database 'TestDatabase', table 'TestTable', column 'iApartment'.

The statement has been terminated.

Оператор INSERT конфликтует с COLUMN CHECK ограничением 'CK__TestTable__iApar__22AA2996'. Конфликт обнаружен в базе данных 'TestDatabase', таблица 'TestTable', колонка 'iApartment'.

Выполнение было прервано.

Эта ошибка может быть отображена и в окне результата выполнения запроса, а не в отдельном диалоговом окне. Смысл сообщения в том, что произошел конфликт вводимых данных и ограничения.

Если с помощью Enterprise Manager открыть свойства таблицы `TestDatabase` и в появившемся окне на вкладке **Check Constraints** (Проверить ограничения) (рис. 1.3) в выпадающем списке **Selected constraint** (Выбранное ограничение) выбрать ограничение, которое мы увидели в сообщении (*CK__TestTable__iApar__22AA2996*), то в окне **Constraint expression** (Выражение ограничения) появится описание ограничения.

Напоследок выполним запрос, который покажет содержимое таблицы:

```
SELECT * FROM TestTable
```

Вы должны увидеть только одну строку, которую мы создали первой с корректными значениями. Некорректная строка в таблицу не попадет.

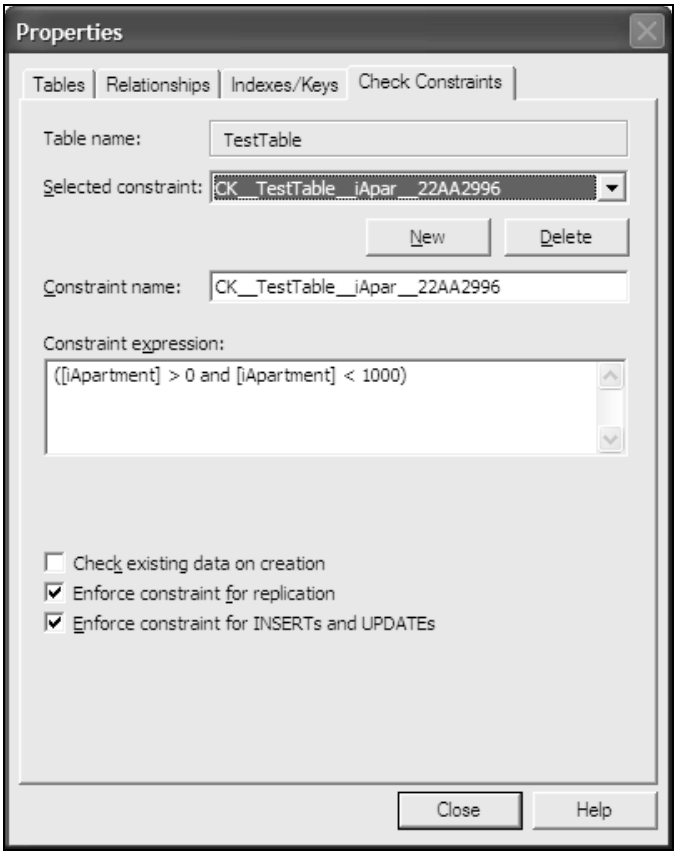


Рис. 1.3. Окно свойств таблицы

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter1\create_table_check1.sql на компакт-диске, прилагаемом к книге.

Самое неприятное в этом примере — это имя ограничения, которое было назначено автоматически, — *CK__TestTable__iApar__22AA2996*. С такими именами очень неудобно работать, намного приятнее видеть читаемые имена и со смыслом. Чтобы задать имя ограничению, необходимо использовать конструкцию:

```
CONSTRAINT имя CHECK (ограничение)
```

Внимание

Имена ограничений внутри базы данных должны быть уникальными. Это значит, что вы не можете создать два ограничения с одним и тем же именем не только для одной и той же таблицы, но и для разных таблиц одной базы данных.

Такие ограничения можно располагать внутри описания поля (как параметры), а можно вынести все в один блок после описания полей. В этом случае ограничение должно выглядеть следующим образом:

```
CONSTRAINT Имя ограничения CHECK (ограничение)
```

Такое ограничение желательно описывать после всех полей и через запятую, как и поля таблицы. Преимущество такого метода в том, что мы сами задаем имя ограничения, и с ним будет удобнее работать в дальнейшем.

Давайте удалим старую таблицу, а создадим новую с еще одним полем — `dDate` типа `datetime`. Для этого поля поставим ограничение, позволяющее вводить только даты, которые меньше текущей:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 vcName varchar(50) NOT NULL,
 iApartment int,
 dDate datetime,
 CONSTRAINT check_iApartment CHECK
    (iApartment>0 and iApartment<1000),
 CONSTRAINT check_dDate CHECK
    (dDate<getdate())
)
```

Как видите, в этом примере ограничения описываются через запятую, так же, как и поля. После ключевого слова `CONSTRAINT` указывается имя ограничения. Из своей практики хочу порекомендовать называть их `check_имя`, где `имя` — имя поля, которое проверяется ограничением. После этого указывается само ограничение точно так же, как и в предыдущем примере.

Обратите внимание, что для ограничения ввода в поле `dDate` мы использовали функцию `getdate()`. Как и при описании значений по умолчанию, в ограничениях также могут использоваться функции.

При создании ограничения можно использовать многие операторы сравнения языка SQL. Есть очень удобный оператор `IN`. С его помощью задаются возможные значения для поля. Например, вам нужно в таблице ограничить ввод данных в поле, содержащее информацию о поле человека. В этом случае можно разрешить ввод букв "М" и "Ж" следующим образом:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 vcName varchar(50) NOT NULL,
```

```
 cPol char(1),
 CONSTRAINT check_cPol CHECK
    (cPol IN ('M', 'Ж'))
)
```

В данном случае ограничение выглядит следующим образом: `cPol IN ('M', 'Ж')`. Оператор `IN` означает, что поле может принимать любые значения, перечисленные в круглых скобках. В нашем случае указано две строки: `'M'` и `'Ж'`. Другие буквы вносить в поле нельзя. То же самое можно написать по-другому:

```
(cPol = 'M' or cPol = 'Ж')
```

Но такое ограничение окажется не очень удобным для чтения. Оператор `IN` в этом случае намного красивее и читабельнее.

Очень эффективной возможностью является использование в ограничении оператора `LIKE`. Например, вы хотите, чтобы поле для хранения телефонного номера содержало номер в формате `(XXX) XXX-XX-XX`, где `X` — любая цифра. Для реализации примера такого ограничения создадим новую таблицу с полем `vcPhonenumber`:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 vcPhonenumber varchar(50) NOT NULL
 CONSTRAINT check_vcPhonenumber CHECK
 (vcPhonenumber LIKE
    '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]')
)
```

В данном случае ограничение с именем `check_vcPhonenumber` при проверке использует оператор `LIKE`. С этим оператором мы познакомимся поближе в *главе 2*, когда будем изучать работу оператора `SELECT`, но я решил привести здесь этот пример, чтобы вы заранее знали о его мощи. Если классический знак равенства производит жесткое сравнение, то `LIKE` позволяет сравнивать строки с определенным шаблоном. В данном случае шаблоном является:

```
([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]
```

В квадратных скобках мы указываем возможный диапазон значений символа. В данном случае диапазон от 0 до 9. Если заменить все `[0-9]` на `X`, то мы получим искомый шаблон `(XXX) XXX-XX-XX`. Следующий пример запроса добавляет в таблицу номер телефона (085) 880-08-00:

```
INSERT INTO TestTable(vcPhonenumber)
VALUES('(085) 880-08-00')
```

Операция пройдет успешно. Но если убрать пробел или любой из символов `(` или `-`, или заменить какую-либо цифру буквой, то операция завершится неудачей:

```
INSERT INTO TestTable(vcPhonenumber)
VALUES('(085)8800800')
```

В ответ на это мы увидим сообщение:

INSERT statement conflicted with COLUMN CHECK constraint 'check_vcPhonenumber'. The conflict occurred in database 'TestDatabase', table 'TestTable', column 'vcPhonenumber'.

The statement has been terminated.

Оператор INSERT конфликтует с ограничением COLUMN CHECK с именем check_vcPhonenumber. Конфликт появился в базе данных TestDatabase, таблице TestTable, колонке vcPhoneNumber.

Выполнение оператора прервано.

Следующий пример задает шаблон для ввода даты:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 vcDate varchar(50) NOT NULL
 CONSTRAINT check_vcDate CHECK
 (vcDate LIKE
    '[0-3][0-9].[0-1][0-9].[0-9][0-9][0-9][0-9]')
)
```

Такой шаблон бывает удобным, если дату нужно хранить в текстовом виде. Да, есть специализированный тип `datetime`, но все же бывают разные задачи, и может пригодиться текстовый тип. Шаблон позволяет ограничить действия пользователя, дабы уменьшить вероятность ошибки, но не решает задачу полностью. Давайте рассмотрим его по частям.

Первая часть (до точки) определяет число месяца. Первая цифра числа может принимать значения от 0 до 3, а вторая цифра от 0 до 9. Это значит, что пользователь может ввести числа от 01 до 39. Числа от 32 до 39 заранее являются некорректными. При указании месяца пользователь также может ошибиться, потому что это значение мы ограничили диапазоном от 01 до 19.

Иногда возникает необходимость в том, чтобы определенное поле или сочетание полей были в базе данных уникальными. Это можно проверять с помощью запросов самостоятельно, а можно доверить проверку серверу.

Для создания ограничения уникальности используется ограничение `UNIQUE`, которое выглядит следующим образом:

```
CONSTRAINT Имя ограничения UNIQUE (Поле или список полей)
```

Следующий пример создает ограничение уникальности для ключевого поля `idName`, ведь недаром оно ключевое и должно быть уникальным:

```
CREATE TABLE Names
(
 idName int,
 vcName varchar(50),
 vcLastName varchar(50),
 vcSurName varchar(50),
 dBirthDay datetime,
 CONSTRAINT cn_unique UNIQUE (idName)
)
```

Недавно мне пришлось разрабатывать базу данных, в которой формировалась отчетность из списка клиентов организации. Когда я начал работать с базой данных, то оказалось, что в базе есть дубликаты записей для клиентов. В старой базе данных не было проверок на двойственность записей, только неэффективные проверки в пользовательской программе.

Самая простая проверка на двойников проходила выявлением записей, где полностью совпадали фамилия, имя, отчество и дата рождения клиента. Вероятность совпадения всех этих параметров внутри одного города стремится к нулю. Таким образом, необходимо было всего лишь добавить ограничение, при котором сочетание этих полей было уникальным:

```
CREATE TABLE Names
(
 idName int ,
 vcName varchar(50),
 vcLastName varchar(50),
 vcSurName varchar(50),
 dBirthDay datetime,
 CONSTRAINT cn_unique UNIQUE (vcName, vcLastName,
    vcSurName, dBirthDay)
)
```

В данном примере создается таблица из полей: идентификатора, имени, фамилии, отчества и даты рождения. Затем создается ограничение, при этом оно имеет тип не `CHECK`, а `UNIQUE`. В скобках указываются поля, которые

должны быть уникальными. В данном примере я перечислил поля: фамилия, имя, отчество и дата рождения. Таким образом, база данных не позволит создать дубликат записи.

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter1\create_table_check2.sql на компакт-диске, прилагаемом к книге.

1.2.5. Первичный ключ

Мы уже достаточно много говорили про ключевые поля, но ни разу их не использовали. Самое интересное, что все работало. Это преимущество, а, может, недостаток базы данных Microsoft SQL Server и MS Access. В таблицах Paradox такой трюк не пройдет, и без наличия ключевого поля таблица будет доступна только для чтения.

В какой-то степени ключи являются ограничениями, и их можно было рассматривать вместе с оператором `CHECK`, потому что объявление происходит схожим образом и даже используется оператор `CONSTRAINT`. Давайте рассмотрим пример. Для этого создадим таблицу из двух полей: `guid` и `vcName`. При этом поле `guid` устанавливается как первичный ключ:

```
CREATE TABLE Globally_Unique_Data
(
 guid uniqueidentifier DEFAULT NEWID(),
 vcName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY (Guid)
)
```

Самое интересное здесь — строка `CONSTRAINT`. Как мы знаем, после этого ключевого слова идет название ограничения, и объявление ключа не является исключением. Я рекомендую именовать первичный ключ так: `PK_имя`, где `имя` — это имя поля, которое должно стать главным ключом. Сокращение PK происходит от Primary Key (первичный ключ).

После этого вместо ключевого слова `CHECK`, которое мы использовали в ограничениях, стоит оператор `PRIMARY KEY`. Именно он указывает на то, что нам необходима не проверка, а первичный ключ. В скобках указываются одно или несколько полей, которые будут составлять ключ.

Помните, что в ключевом поле не может быть одинакового значения у двух строк, в этом ограничение первичного ключа идентично ограничению уникальности. Это значит, что если сделать поле для хранения фамилии первичным ключом, то в такую таблицу нельзя будет записать двух Ивановых с раз-

ными именами. Это нарушает ограничение первичного ключа. Именно поэтому ключи являются ограничениями и объявляются так же, как и ограничение `CHECK`. Но это верно только для первичных ключей и вторичных, обладающих уникальностью.

В данном примере в качестве первичного ключа выступает поле типа `uniqueidentifier` (GUID). Значение по умолчанию для этого поля — результат выполнения серверной процедуры `NEWID`.

Внимание

Только один первичный ключ может быть создан для таблицы.

Для простоты примеров в качестве ключа желательно использовать числовой тип, и если позволяет база данных, то будет лучше, если он будет типа `autoincrement` (автоматически увеличивающееся/уменьшающееся число). В MS SQL Server таким полем является `IDENTITY`, а в MS Access это поле типа "счетчик".

Следующий пример показывает, как создать таблицу товаров, в которой в качестве первичного ключа выступает целочисленное поле с автоматическим увеличением:

```
CREATE TABLE Товары
(
  id int IDENTITY(1, 1),
  товар varchar(50),
  Цена money,
  Количество numeric(10, 2),
  CONSTRAINT PK_id PRIMARY KEY (id)
)
```

Именно такой тип ключа мы будем использовать чаще всего, потому что в ключевом поле будут храниться легкие для восприятия числа, и с ними проще и нагляднее работать.

Первичный ключ может состоять более чем из одной колонки. Следующий пример создает таблицу, в которой поля `id` и `Товар` образуют первичный ключ, а значит, будет создан индекс уникальности на оба поля:

```
CREATE TABLE Товары1
(
  id int IDENTITY(1, 1),
  Товар varchar(50),
  Цена money,
```

```
  Количество numeric(10, 2),
  CONSTRAINT PK_id PRIMARY KEY
         (id, [Название товара])
)
```

Очень часто программисты создают базу данных с ключевым полем в виде целого числа, но при этом в задаче четко сказано, что определенные поля должны быть уникальными. А почему не создать сразу первичный ключ из тех полей, которые должны быть уникальны, и не надо будет создавать отдельные решения для данной проблемы.

Единственный недостаток первичного ключа, состоящего из нескольких колонок, — проблемы создания связей. Тут приходится выкручиваться различными методами, но проблема все же решаема. Достаточно только ввести поле типа `uniqueidentifier` и производить связь по нему. Да, в этом случае у нас получаются уникальными первичный ключ и поле типа `uniqueidentifier`, но эта избыточность в результате не будет больше, чем та же таблица, где первичный ключ `uniqueidentifier`, а на поля, которые должны быть уникальными, установлено ограничение уникальности. Что выбрать? Зависит от конкретной задачи и от того, с чем вам удобнее работать.

1.2.6. Внешний ключ

Внешний ключ также является ограничением `CONSTRAINT` и отображает связь между двумя таблицами. Допустим, что у вас есть две таблицы:

- `Names` — содержит имена людей и состоит из полей идентификатора (ключевое поле) и имени;
- `Phones` — содержит номера телефонов и состоит из трех полей: идентификатор (ключевое поле), внешний ключ для связи с таблицей `Names` и строковое поле для хранения номера телефона.

У одного человека может быть несколько телефонов, поэтому мы разнесли данные в разные таблицы. На рис. 1.4 показана связь между двумя таблицами. Если вы уже работали со связанными таблицами, то для вас рисунка будет достаточно. Если вы слышите о связях впервые, то обсудим эту тему подробнее.

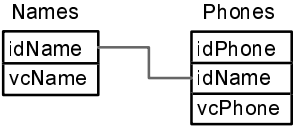


Рис. 1.4. Связь между таблицами

В качестве примера рассмотрим таблицу для трех человек. В табл. 1.3 показано содержимое таблицы `Names`. Здесь всего три строки, и у каждой свой уникальный главный ключ. Для уникальности, когда будем создавать таблицу, сделаем ключ автоматически увеличиваемым полем.

Таблица 1.3. *Содержимое таблицы* `Names`

Главный ключ	Фамилия
1	Петров
2	Иванов
3	Сидоров

Таблица 1.4. *Содержимое таблицы* `Phones`

Главный ключ	Внешний ключ	Номер телефона
1	1	678689687
2	1	2324234
3	2	324234
4	3	32432423
5	3	2
6	3	12312312

В таблице `Phones` (табл. 1.4) содержатся пять номеров телефонов. В поле `Главный ключ` находится уникальный главный ключ, которой можно сделать автоматически увеличиваемым. Внешний ключ — это связь с главным ключом таблицы `Names`. Как работает эта связь? У Петрова в таблице `Names` в качестве главного ключа стоит число 1. В таблице `Phones` во внешнем ключе ищем число 1 и получаем номера телефонов Петрова. То же самое и с остальными записями. Связь между строками двух таблиц можно увидеть на рис. 1.5.

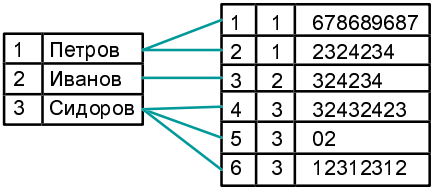


Рис. 1.5. Связь между строками таблиц

Такое хранение данных очень удобно. Если бы не было возможности создавать связанные таблицы, то в таблице `Names` пришлось бы вводить все номера телефонов в одно поле. Это неудобно с точки зрения использования, поддержки и поиска данных.

Можно создать в таблице несколько полей `Names`, но возникает вопрос — сколько. У кого-то только один телефон, а у меня, например, их три, не считая рабочих. Большое количество полей приводит к избыточности данных.

Можно для каждого телефона в таблице `Names` заводить отдельную строку с фамилией, но это приемлемо только для такого простого примера, когда нужно вводить лишь фамилию и нетрудно внести несколько записей для Петрова с несколькими номерами телефонов. А если полей будет 10 или 20?

Итак, создание двух таблиц, связанных внешним ключом, можно увидеть в листинге 1.6.

Листинг 1.6. Создание таблиц, связанных внешним ключом

```
CREATE TABLE Names
(
 idName int IDENTITY(1,1),
 vcName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY (idName),
)

CREATE TABLE Phones
(
 idPhone int IDENTITY(1,1),
 idName int,
 vcPhone varchar(10),
 CONSTRAINT PK_idPhone PRIMARY KEY (idPhone),
 CONSTRAINT FK_idName FOREIGN KEY (idName)
   REFERENCES Names (idName)
)
```

Внимательно изучите содержимое листинга. Он достаточно интересен, потому что использует некоторые операторы, которые мы уже рассмотрели, и дополнительный пример не помешает. Для обеих таблиц создается ключевое поле, которое стоит первым, имеет тип `int` и автоматически увеличивается, начиная с 1, с шагом 1. Ключевое поле делается главным ключом с помощью ограничения `CONSTRAINT`.

В описании таблицы `Phones` последняя строка содержит новое для нас объявление, а именно — объявление внешнего ключа с помощью оператора `FOREIGN KEY`. Как видите, это тоже ограничение, и чуть позже вы увидите, почему. В скобках указывается поле таблицы, которое должно быть связано с другой таблицей. После этого идет ключевое слово `REFERENCES` (ссылка), имя таблицы, с которой должна быть связь (`Names`), и в скобках имя поля (`idName`). Таким образом, мы навели связь, которая отображена на рис. 1.4.

Внимание

Внешний ключ может ссылаться только на первичный ключ другой таблицы или на ограничение уникальности. Это значит, что после ключевого слова `REFERENCES` должно идти имя таблицы, и в скобках можно указывать только первичный ключ или поле с ограничением `UNIQUE`. Другие поля указывать нельзя.

Теперь можно наполнять таблицы данными. Следующие команды добавляют три фамилии, которые мы видели в табл. 1.3:

```
INSERT INTO Names(vcName)
VALUES('Петров')

INSERT INTO Names(vcName)
VALUES('Иванов')

INSERT INTO Names(vcName)
VALUES('Сидоров')
```

Если вы уже работали с SQL, то сможете добавить записи и для таблицы телефонов. Я опущу эти команды, а вы можете увидеть их в файле \Chapter1\foreign_keys.sql на компакт-диске, прилагающемся к книге.

Наша задача — выяснить, в чем заключаются ограничительные действия внешнего ключа. Мы указали явную связь между двумя полями в разных таблицах. Если попытаться добавить в таблицу телефонов запись с таким идентификатором в поле `idName`, который не существует в одноименном поле (имя можно было сделать и другим) таблицы с фамилиями, то произойдет ошибка. Это нарушит связь между двумя таблицами, а ограничение внешнего ключа не позволит существовать записям без связи.

Ограничение действует и при изменении или удалении записей. Например, если попытаться удалить строку с фамилией Петров, то произойдет ошибка ограничения внешнего ключа. Нельзя удалять записи, для которых существуют внешне связанные строки. Для начала нужно удалить все телефоны для

данной записи, и только после этого будет возможно удаление самой строки с фамилией Петров.

Во время создания внешнего ключа можно указать `ON DELETE CASCADE` или `ON UPDATE CASCADE`. В этом случае, если удалить запись о Петрове из таблицы `Names` или изменить идентификатор, то все записи в таблице `Phones`, связанные со строкой Петрова, будут автоматически обновлены. НИКОГДА не делайте этого! Все должно удаляться или изменяться вручную. Если пользователь случайно удалит запись из таблицы `Names`, то удалятся и соответствующие номера телефонов. Какой тогда смысл создавать внешний ключ, если половина его ограничительных возможностей исчезает?! Все необходимо делать только вручную, а идентификаторы изменять не рекомендуется вообще никогда.

Удаление самих таблиц также должно начинаться с подчиненной таблицы, т. е. с `Phones`, и только потом можно удалить главную таблицу `Names`.

Напоследок покажу, каким образом можно получить соответствие имен и телефонов из двух таблиц:

```
SELECT vcName, vcPhone
FROM Names, Phones
WHERE Names.idName=Phones.idName
```

Более подробно о подобных запросах мы поговорим в *главе 2*. Сейчас же я привел пример только для того, чтобы вы увидели мощь связанных таблиц.

Таблица может содержать до 253 внешних ключей, что вполне достаточно даже для построения самых сложных баз данных. Лично мне приходилось работать с базами данных, где количество внешних ключей не превышало 7 на одну таблицу. Если больше, то, скорее всего, база данных спроектирована неверно, хотя бывают и исключения.

Сама таблица также может иметь максимум 253 внешних ключа. Внешние ключи в таблице встречаются реже, в основном, не более 3. Чаще всего в таблице может быть много ссылок на другие таблицы.

Внешний ключ может ссылаться на ту же таблицу, в которой он создается. Например, у вас есть таблица должностей в организации, как показано в табл. 1.5. Таблица состоит из трех полей: первичный ключ, внешний ключ и название должности. В любой организации может быть множество должностей, но вполне логичным будет в одной таблице отобразить их названия и структуру подчинения. Для этого внешний ключ нужно связать с первичным ключом таблицы должностей.

Таблица 1.5. *Таблица с внутренней связью*

Главный ключ	Внешний ключ	Должность
1	NULL	Генеральный директор
2	1	Коммерческий директор
3	1	Директор по общим вопросам
4	2	Начальник отдела снабжения
5	2	Начальник отдела сбыта
6	3	Начальник отдела кадров

В результате мы получаем, что у генерального директора внешний ключ нулевой, т. е. эта должность стоит во главе всех остальных. У коммерческого директора и директора по общим вопросам внешний ключ указывает на строку генерального директора. Это значит, что эти две должности подчиняются непосредственно генеральному директору. И так далее.

Посмотрим, как можно оформить все это в виде SQL-запроса:

```
CREATE TABLE Positions
(
 idPosition int IDENTITY(1,1),
 idParentPosition int,
 vcName varchar(30),
 CONSTRAINT PK_idPosition PRIMARY KEY (idPosition),
 CONSTRAINT FK_idParentPosition FOREIGN KEY (idParentPosition)
   REFERENCES Positions (idPosition)
)
```

Как видите, внешний ключ просто ссылается на ту же таблицу, которую мы создаем. В файле \Chapter1\foreign_keys_to_self.sql на компакт-диске, прилагаемом к книге, можно увидеть пример создания этой таблицы, наполнения его данными и отображения должностей с учетом их подчинения. В *главе 2* мы рассмотрим возможность работы с такими таблицами более подробно.

Один-к-одному

Пока что мы рассмотрели классическую связь, когда одной строке основной таблицы данных соответствует несколько строк из связанной таблицы. Такая связь называется "один-ко-многим". Но существуют и другие связи, и сейчас мы рассмотрим связь "один-к-одному", при которой одна запись основной таблицы связана с одной записью другой. Чтобы это реализовать, достаточно связать первичные ключи обеих таблиц.

Следующий пример создает две таблицы, у которых создана связь между первичными ключами:

```
CREATE TABLE Names
(
 idName uniqueidentifier DEFAULT NEWID(),
 vcName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY (idName)
)

CREATE TABLE Phones
(
 idPhone uniqueidentifier DEFAULT NEWID(),
 vcPhone varchar(10),
 CONSTRAINT PK_idPhone PRIMARY KEY (idPhone),
 CONSTRAINT FK_idPhone FOREIGN KEY (idPhone)
   REFERENCES Names (idName)
)
```

Внешний ключ нужен только в одной из таблиц. Так как связь идет "один-к-одному", то не имеет значения, в какой таблице создать его.

Многие-ко-многим

Самая сложная связь — "многие-ко-многим", когда много записей из одной таблицы соответствует многим записям из другой таблицы. Чтобы такое реализовать, двух таблиц мало, необходимы три таблицы.

Для начала нужно понять, когда может использоваться связь "многие-ко-многим". Допустим, что у вас есть две таблицы: список жителей дома и список номеров телефона. В одной квартире может быть более одного номера, а значит, одной фамилии может принадлежать два телефона. Получается связь "один-ко-многим". С другой стороны, в одной квартире может быть две семьи (коммунальная квартира или просто квартиросъемщик, который пользуется телефоном владельца), а значит, связь между телефоном и жителем тоже "один-ко-многим". И самый сложный вариант — в коммунальной квартире находятся два телефона. В этом случае обоими номерами пользуются несколько жителей в квартире. Вот и получается, что "много" семей может пользоваться "многими" телефонами (связь "многие-ко-многим").

Как реализовать связь "многие-ко-многим"? На первый взгляд, в реляционной модели это невозможно. Лет 10 назад я долго искал разные варианты и в результате просто создавал одну таблицу, которая переполнена избыточными

данными. Но однажды мне досталась одна задача, благодаря которой уже из условия на поверхность вышло отличное решение — нужно создать две таблицы жителей квартир и телефонов и реализовать в них только первичный ключ. Внешние ключи в этой таблице не нужны. А вот связь между таблицами должна быть реализована через третью, связующую таблицу. На первый взгляд, сложно и непонятно, но один раз разобравшись с этим методом, вы увидите всю мощь такого решения.

В табл. 1.6 и 1.7 приведены примеры таблиц фамилий и телефонов соответственно. А в табл. 1.8 показана связующая таблица.

Таблица 1.6. *Таблица фамилий*

Ключ	Имя
1	Иванов
2	Петров
3	Сидоров

Таблица 1.7. *Таблица телефонов*

Ключ	Имя
1	68768678
2	658756785
3	567575677

Таблица 1.8. *Связующая таблица*

Ключ	Связь с именем	Связь с телефоном
1	1	1
2	1	2
3	2	1
4	2	3
5	3	3

Давайте теперь посмотрим, какая будет логика поиска данных при связи "многие-ко-многим". Допустим, что нам нужно найти номера всех телефонов, которые принадлежат Иванову. У Иванова первичный ключ равен 1.

Находим в связующей таблице все записи, у которых поле `Связь с именем` равно 1. Это будут записи 1 и 2. В этих записях в поле `Связь с телефоном` находятся идентификаторы 1 и 2 соответственно, а значит, Иванову принадлежат номера из таблицы телефонов, которые расположены в строках 1 и 2.

Теперь решим обратную задачу — определим, кто имеет доступ к номеру телефона 567575677. Этот номер в таблице телефонов имеет ключ 3. Ищем все записи в связующей таблице, где значение поля `Связь с телефоном` равно 3. Это записи с номерами 4 и 5, которые в поле `Связь с именем` содержат значения 2 и 3 соответственно. Если теперь посмотреть на таблицу фамилий, то вы увидите под номерами 2 и 3 Петрова и Сидорова. Значит, именно эти два жителя пользуются телефоном с номером 567575677.

Просмотрите все три таблицы и убедитесь, что вы поняли, какие номера телефонов принадлежат каким жителям и наоборот. Если вы увидите эту связь, то поймете, что она очень проста, и сможете быстро реализовать ее в своих проектах.

```
CREATE TABLE Names
(
 idName uniqueidentifier DEFAULT NEWID(),
 vcName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY (idName)
)

CREATE TABLE Phones
(
 idPhone uniqueidentifier DEFAULT NEWID(),
 vcPhone varchar(10),
 CONSTRAINT PK_idPhone PRIMARY KEY (idPhone)
)

CREATE TABLE LinkTable
(
 idLinkTable uniqueidentifier DEFAULT NEWID(),
 idName uniqueidentifier,
 idPhone uniqueidentifier,

 CONSTRAINT PK_idLinkTable PRIMARY KEY (idLinkTable),

 CONSTRAINT FK_idPhone FOREIGN KEY (idPhone)
```

```
   REFERENCES Phones (idPhone),
 CONSTRAINT FK_idName FOREIGN KEY (idName)
   REFERENCES Names (idName)
)
```

У связующей таблицы два внешних ключа, которые связываются с таблицами имен и телефонов, и один первичный ключ, который обеспечивает уникальность записей.

В качестве первичного ключа я выбрал GUID-поле, потому что для решения именно этой задачи он удобнее. Дело в том, что нам нужно вставлять записи в две таблицы и в обоих случаях нужно указывать один и тот же ключ. Значение GUID можно сгенерировать, а потом использовать при вставке данных в обе таблицы.

Вы можете использовать в качестве ключа и автоматически увеличиваемое поле, но в этом случае задачу решать немного сложнее, точнее сказать, неудобно. Например, добавляя номер телефона, необходимо сначала вставить соответствующую строку в таблицу, потом найти ее, определить ключ, который был назначен строке, и после этого уже наводить связь.

На данном этапе мы ограничиваемся только созданием таблиц, а в *разд. 2.8* мы вернемся к этой теме и научимся работать со связанными таблицами. Работа со связями "один-к-одному" и "один-ко-многим" отличается несильно, потому что в этой схеме участвуют только две таблицы. Связь "многие-ко-многим" немного сложнее из-за связующей таблицы, поэтому мы ее рассмотрим отдельно в *разд. 2.27*.

1.2.7. Индексы

Для ограничений `UNIQUE` и `PRIMARY KEY` автоматически создается индекс, который упрощает поиск необходимых данных. Что такое индекс? Если говорить простыми словами, то это способ отсортировать данные по определенной колонке. В отсортированном списке намного легче производить поиск необходимых данных.

Мы уже знаем, как хранятся данные, теперь обсудим, как получить к ним доступ. Для начала нужно разобраться с таким понятием, как *куча*. Куча — это коллекция страниц данных, содержащих строки для таблицы.

- Каждая страница данных содержит 8 Кбайт информации. Группа из 8 последовательных страниц называется пространством.
- Строки данных не хранятся в каком-либо определенном порядке, и нет определенного порядка для последовательности страниц.

- Страницы данных не связаны в связанные списки.
- Когда строка вставляется в страницу и страница переполняется, то страница разделяется.

Сервер SQL получает доступ к данным одним из следующих способов:

- сканируя все страницы таблицы — сканирование таблицы. Когда SQL Server выполняет сканирование таблицы, он:
  - идет с начала таблицы;
  - последовательно сканирует страницы через все строки таблицы;
  - выделяет строку, которая соответствует запросу.
- используя индексы. Когда SQL Server использует индексы, он:
  - пересекает структуру дерева индексов для поиска строк, соответствующих запросу;
  - выделяет только необходимые строки, соответствующие критериям запроса.

Первым делом SQL Server выясняет, какие индексы существуют. Оптимизатор запроса (компонент, предназначенный для генерирования оптимального плана для запроса) определяет, что использовать — сканировать таблицу или индексы. Индексы предпочтительнее. Когда вы принимаете решение, нужно ли создавать индексы, рассмотрите два фактора, чтобы выяснить, окажутся ли индексы более эффективными, чем сканирование таблицы: природа данных и природа запросов к таблице.

Индексы ускоряют доступ к данным. Например, без индекса вам понадобится перелистать постранично всю книгу для поиска интересующей вас информации.

Сервер SQL использует индексы для указания на расположение строки в странице данных вместо просмотра всех страниц таблицы. Учитывайте следующие факты и рекомендации:

- индексы обычно увеличивают скорость выполнения запросов связанных таблиц и выполнение сортировки и группировки;
- индексы принуждают делать строки уникальными, если включена уникальность;
- индексы создаются в порядке возрастания или убывания.

Индексы очень полезны, но они занимают место на диске и требуют дополнительных расходов. Индексы могут создать и проблемы:

- когда вы изменяете данные в индексной колонке, сервер SQL обновляет связанные индексы;

- накладные расходы на поддержку индексов довольно велики. Поэтому не создавайте индексы, которые не будете часто использовать;
- индексы на колонки, содержащие большое количество дублирующихся данных, могут оказаться неэффективными. Например, какая польза от индекса на колонку, содержащую пол человека, ведь там можно записать только "М" или "Ж".

Индексы бывают кластерными (`CLUSTERED`) и некластерными (`NONCLUSTERED`). В кластерном индексе строки физически сортируются на диске в соответствии с индексируемым полем. Я думаю, что не надо объяснять, почему кластерный индекс может быть только один на таблицу? Нельзя же одновременно физически отсортировать данные по двум ключам.

При некластерном индексе строки могут на диске храниться в любом порядке, а сортировка осуществляется с помощью определенной таблицы или дерева индекса. В SQL-сервере используется принцип дерева.

Конечно, это все из области администрирования SQL-сервера, но вы должны понимать, что такое индекс и для чего он нужен.

Все первичные ключи, начиная с SQL Server 2000, по умолчанию создаются кластерными. Ограничения `UNIQUE` по умолчанию создаются некластерными.

Чтобы лучше понять разницу между кластерным и некластерным индексами, посмотрим на рис. 1.6, где показана примерная схема индекса. Каждый прямоугольник — это страница данных из 8 Кбайт. Вверху находится корень дерева, а внизу листья дерева.

Допустим, что нам нужно найти имя Анатолий. Из корня мы узнаем, что для поиска сведений об имени нужно спуститься влево вниз. Здесь также будет информация о том, что нужно спуститься еще влево вниз, и тогда мы окажемся в листовом блоке.

В некластерном индексе в листовом блоке находится ссылка на строку с данными, где можно найти имя Анатолий, а в кластерном индексе данные находятся непосредственно в листовом блоке.

Кластерный индекс иногда ускоряет работу базы данных, а иногда он просто удобнее. Однажды я разрабатывал программу на Delphi + MS SQL Server, которая состояла из двух частей. Серверная часть программы сохраняла в базе данных строки, полученные с производственного оборудования, а клиентская их читала. Когда сервер сохранял данные в базе, программа направляла программе-клиенту сообщение по сети. Получив это сообщение, клиент должен был прочитать сохраненную строку и вывести пользователю на экран.

Чтобы упростить чтение на клиентской стороне и обойтись без выполнения лишних запросов на выборку данных, я использовал готовый метод компо-

нентов доступа к базам данных Delphi — переход на последнюю строку. Этот метод очень удобен тем, что быстро позволяет получить последнюю строку. Все прекрасно работало, но почему-то в определенные моменты переход блокировался на определенной строке, и какое-то время новые строки не отображались.

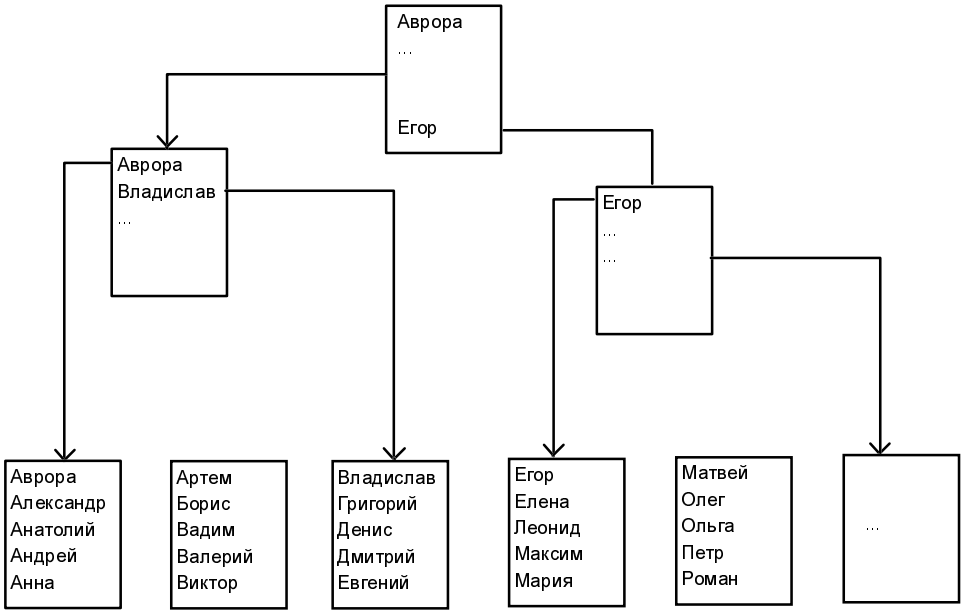


Рис. 1.6. Схема древовидного индекса

Проблема крылась как раз в некластерном индексе. Когда базе данных не хватало места, она выделяла блок памяти на диске в произвольном месте. Если это место оказывалось в конце базы, то переходы происходили удачно. Если находилось свободное место в уже существующих блоках памяти, но не заполненных до конца (в любом месте базы данных, но не в конце), то переход не осуществлялся, потому что физически строка оказывалась не в конце. Проблему решило использование кластерного индекса.

Следующий пример показывает, как можно создать некластерный индекс:

```
CREATE TABLE Names
(
 idName int IDENTITY(1,1),
 vcName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY NONCLUSTERED (idName),
)
```

В MS SQL Server 2000 первичный ключ делается кластерным, и явное указание оператора `CLUSTERED` нужно только для MS SQL Server 7. Учитывайте эту разницу при обработке первичного ключа разными версиями MS SQL Server во время разработки собственных приложений. А лучше ни на кого не надеяться, а всегда явно указывать тип необходимого ключа.

В таблице может быть создано 249 некластерных индексов, и только один индекс может быть кластерным, т. к. физически можно упорядочить записи только по одному полю.

Как мы уже знаем, кластерным может быть и ограничение уникальности. Для ограничения уникальности создается индекс, а свойство кластерный/некластерный относится как раз к индексу. Например:

```
CREATE TABLE Names
(
 idName int ,
 vcName varchar(50),
 vcLastName varchar(50),
 vcSurName varchar(50),
 dBirthDay datetime,
 CONSTRAINT cn_unique UNIQUE CLUSTERED(vcName, vcLastName,
    vcSurName, dBirthDay)
)
```

В данном примере создается только индекс для полей с ограничением уникальности. При этом главного ключа нет. А если его создать:

```
CREATE TABLE Names
(
 idName int ,
 vcName varchar(50),
 vcLastName varchar(50),
 vcSurName varchar(50),
 dBirthDay datetime,
 CONSTRAINT pk_idName PRIMARY KEY (idName),
 CONSTRAINT cn_unique UNIQUE CLUSTERED(vcName, vcLastName,
    vcSurName, dBirthDay)
)
```

В данном случае создается первичный ключ, и по умолчанию он должен быть кластерным. Но поскольку кластерным создается ограничение уникальности `cn_unique`, то первичный ключ автоматически получит некластерный индекс.

А если явно указать, что мы хотим первичный ключ и ограничение уникальности сделать кластерными? В этом случае произойдет ошибка: *"Cannot add more than one clustered index for constraints on table 'Names'"* ("Не могу добавлять более чем один кластерный индекс для ограничения на таблицу Names"). Я же говорил, что кластерным может быть только один индекс.

Чтобы увидеть, что делает сервер после создания таблицы, выполните команду:

```
sp_help Names
```

Команда `sp_help` (точнее, это процедура SQL-сервера) отображает подробную информацию об указанной таблице, в данном случае `Names`. Эту команду лучше всего выполнять в программе Query Analyzer, которая поставляется вместе с MS SQL Server, потому что ее результат состоит из нескольких таблиц, а Query Analyzer умеет их отображать в удобочитаемом виде.

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	idName	int	no	4	10	0	no	(n/a)	(n/a)	NULL
2	vcName	varchar	no	50			yes	no	no	Cyrillic_General_CI_
3	vcLastName	varchar	no	50			yes	no	no	Cyrillic_General_CI_
4	vcSurName	varchar	no	50			yes	no	no	Cyrillic_General_CI_
5	dBirthDay	datetime	no	8			yes	(n/a)	(n/a)	NULL

	Identity	Seed	Increment	Not For Replication
1	No identity column defined.	NULL	NULL	NULL

	RowGuidCol
1	No rowguidcol column defined.

	Data_located_on_filegroup
1	PRIMARY

	index_name	index_description	index_keys
1	cn_unique	clustered, unique, unique k...	vcName, vcLastName, vcSurNa...
2	pk_idName	nonclustered, unique, prima...	idName

	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constra
1	UNIQUE (clustered)	cn_unique	(n/a)	(n/a)	(n/a)	(n/a)	vcName,
2	PRIMARY KEY (non-clustered)	pk_idName	(n/a)	(n/a)	(n/a)	(n/a)	idName

Рис. 1.7. Подробная информация о таблице `Names`

На рис. 1.7 показан результат выполнения команды `sp_help` для таблицы `Names`. Обратите внимание на последние две строки. Здесь отображаются список ограничений и их имена. Чуть выше показаны две строки индексов для этих ограничений. Первая строка соответствует ограничению `cn_unique`, а во второй колонке видно, что этот индекс кластерный (`clustered`). Вторая строка — это индекс `pk_idNames`, а во второй колонке указано, что индекс некластерный (`nonclustered`).

Ваше бизнес-окружение, характеристики данных и способы их использования определяют колонки, которые должны быть проиндексированы. Эффективность применения индекса напрямую зависит от процента возвра-

щаемых запросом строк: это число должно быть очень маленьким или очень большим.

Создавайте индексы на следующие поля:

- первичный ключ, такой индекс создается автоматически;
- внешний ключ или поле, которое часто используется для связи таблиц. На внешние ключи индексы автоматически не создаются, но если в связанных таблицах находится много строк, то индекс реально может повысить производительность. Если в основной таблице много строк, а в связанной не более 100, можно обойтись и без индекса;
- поле, используемое для поиска ряда значений;
- поле, по которому сортируются данные;
- поля, которые группируются во время агрегации (оператор `GROUP BY`);
- поле, которое часто используется в запросах `SELECT`.

Не стоит создавать индексы на такие поля:

- редко используемые в запросе;
- содержащие несколько уникальных значений, например, колонки, содержащие только значение "пол". Такой индекс будет лишь замедлять работу системы;
- объявленные как `text`, `ntext` или `image` типы. Колонки с этими типами данных не могут быть проиндексированы.

Вы должны создавать только самые необходимые индексы, потому что каждый лишний индекс может серьезно ударить по производительности во время добавления новых записей. Это становится особенно заметным при массовой загрузке данных.

Попробуем разобраться, какой индекс нужно создавать — кластерный или нет. Для того чтобы сделать правильный выбор, нужно понимать, как будет использоваться ваша таблица.

Кластерный индекс на сортируемую колонку полезен при работе с таблицами, которые часто используются для отчетов, группировки для агрегации или поиска ряда значений.

Сервер SQL использует значение кластерного индекса в качестве идентификатора строки внутри каждого некластерного индекса. Кластерный индекс может повторяться много раз в структуре вашей таблицы.

Рекомендуется делать кластерный индекс минимальным по размеру. Для предотвращения больших кластерных индексов:

- ограничьте количество колонок в кластерном индексе;

- уменьшите среднее количество символов за счет использования типа данных `varchar` вместо `char`, а лучше ограничьтесь числовыми типами данных или уникальным идентификатором `guid`;
- старайтесь использовать наиболее компактный тип данных.

Когда вы определяете плотность ваших данных, помните, что она связана с определенными элементами данных. Плотность может изменяться. Что значит плотность? Рассмотрим ее на примере таблицы работников, которая содержит даты рождения. Допустим, что в фирме работает 100 человек в возрасте от 23 до 30 лет и 10 человек в возрасте старше 30. В диапазоне от 23 до 30 получается высокая плотность, потому что здесь находится очень много записей.

Так как данные распределяются неравномерно, оптимизатор запросов не обязательно будет использовать индексы. Он может:

- выполнить сканирование таблицы для поля с большой плотностью или для поля, значение которого может вызвать возврат большого количества строк;
- если в проиндексированном поле много раз встречается имя "Вася", то по нему может быть использовано сканирование. При этом для редкого значения, например, имени "Аврора", будет использоваться индекс.

Когда вы определяете плотность данных, вы должны также учитывать и разброс данных. Разброс определяет количество данных в определенных диапазонах. Если индексированная колонка имеет мало уникальных значений, получение данных может оказаться медленным. Например, если у вас есть таблица с полем, отсортированным по фамилии, то данные могут быть неравномерно разбросаны по алфавиту.

Сервер сам по себе не может ничего знать о разбросе данных. Для этого ему необходимо собрать определенную статистику о полях и содержащихся в таблице значениях. Имея статистику, сервер сможет принять более эффективное решение о том, надо ли использовать индекс, или сканирование таблицы будет выполняться быстрее. О статистике мы еще не раз будем говорить в *главе 4*.

Создание индексов

Теперь посмотрим, как создавать индексы вручную. До этого момента мы использовали индексы для первичного ключа и уникального поля, созданные сервером SQL автоматически при создании ограничений `PRIMARY KEY` или `UNIQUE`, но иногда требуется создать индекс на поле без этих ограничений.

Для создания индекса на произвольное поле используется оператор `CREATE INDEX`, а для удаления — оператор `DROP INDEX`. Вы должны быть владельцем базы данных или администратором, чтобы выполнять эти операторы.

Информация об индексах хранится в системной таблице `sysindexes`. В *главе 2* мы научимся работать с таблицами и просматривать их содержимое. Просто ради интереса попробуйте просмотреть системную таблицу `sysindexes`. Только не вздумайте ее изменять вручную, системные таблицы можно только просматривать.

Самым эффективным окажется индекс на поле небольшого размера. При создании кластерного индекса все существующие некластерные индексы перестраиваются, поэтому желательно в первую очередь создавать кластерный индекс.

В общем виде команда создания индекса выглядит следующим образом:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON { table | view } ( column [ ASC | DESC ] [ ,...n ] )
[ WITH < index_option > [ ,...n] ]
[ ON filegroup ]
```

Для удобства я разбиваю эту команду на строчки. В первой строчке указываются ключевые слова `CREATE` и `INDEX`, между которыми можно поставить `UNIQUE`, чтобы индекс был уникальным, и `CLUSTERED` или `NONCLUSTERED`, чтобы сделать индекс кластерным или некластерным соответственно. После `INDEX` указывается имя индекса.

Имя должно быть информативным. Я рекомендую использовать такой формат: "`I_CL_Имя`". Первая буква `I` указывает на то, что это индекс. Затем я ставлю `CL` или `UCL`, т. е. кластерный или некластерный индекс соответственно. И в самом конце перечисляются имена полей, которые индексируются. В данном случае только одно поле `vcName`.

Во второй строчке я пишу ключевое слово `ON`, за которым идет имя таблицы и в скобках имена индексируемых полей.

Следующий пример создает кластерный индекс на колонку `vcName`:

```
CREATE CLUSTERED INDEX I_CL_vcName
ON TestTable(vcName)
```

После имени колонки нужно указать направление сортировки индекса. Направление задается ключевыми словами `ASC` (возрастание) или `DESC` (убывание). Следующий пример создает некластерный индекс по убыванию:

```
CREATE NONCLUSTERED INDEX I_UCL_vcName
ON TestTable(vcName DESC)
```

Теперь поговорим об удалении индексов. С помощью оператора DROP INDEX можно удалять только созданные вами индексы. Нельзя использовать этот оператор для удаления индекса, который был автоматически создан на ограничения PRIMARY KEY или UNIQUE. Вы должны удалить ограничение, прежде чем удалять индекс. Нельзя удалять индексы системных таблиц.

Если удалить кластерный индекс, то все некластерные индексы будут автоматически перестроены.

В общем виде команда удаления индекса выглядит так:

```
DROP INDEX 'table.index | view.index' [ ,...n ]
```

В следующем примере удаляется созданный нами ранее индекс:

```
DROP INDEX TestTable.I_CL_vcName
```

Ранее мы уже создавали индекс уникальности, но делали это только на этапе создания таблицы. Если она уже существует, то индекс уникальности можно добавить с помощью оператора CREATE UNIQUE INDEX.

Уникальный индекс гарантирует, что все данные в колонке с таким индексом уникальны и не содержат повторяющихся значений. Сервер SQL автоматически создает индекс, когда создается ограничение PRIMARY KEY или UNIQUE.

Сервер SQL проверяет дубликаты каждый раз, когда вы выполняете операторы INSERT или UPDATE. Если дубликат существует, то сервер отклоняет ваши операторы и возвращает сообщение об ошибке.

Если повторяющиеся значения существуют, когда вы создаете уникальный индекс, операция CREATE INDEX отклоняется. Сервер возвращает сообщение об ошибке при обнаружении первого дубликата, но могут существовать и еще дубликаты. Используйте следующий простой сценарий для любых таблиц, чтобы найти дублирующиеся значения в колонке.

```
SELECT индексная колонка, COUNT(индексная колонка)
FROM имя таблицы
GROUP BY индексная колонка
HAVING COUNT (индексная колонка)>1
ORDER BY индексная колонка
```

Снова мы забегаем вперед, потому что запросы SELECT — это тема следующей главы. Если вы не работали с SQL, то этот запрос еще непонятен вам, но вернитесь к нему после прочтения *главы 2*, и все встанет на свои места.

Составные индексы

Составные индексы в качестве ключевого значения используют более одной колонки. Создавайте составные индексы, если для поиска в качестве ключа

чаще всего используются два или более полей, и если запрос ссылается на все поля в составном индексе.

Хорошим примером является телефонный справочник, организованный по фамилиям. Вместе с фамилией для поиска регулярно используется имя, потому что в справочнике присутствует много записей с одной фамилией, но с разными именами, и вполне логично создать индекс из фамилии и имени одновременно.

Вы можете объединять до 16 колонок в составной индекс. Сумма длин всех колонок составного индекса должна быть менее 900 байт. При этом все поля должны принадлежать одной таблице.

Объявляйте сначала уникальные колонки. Первые колонки, описанные в операторе `CREATE INDEX`, имеют высший приоритет при сортировке. При поиске данных в таблице ваш запрос должен ссылаться на первую колонку индекса, иначе индекс будет проигнорирован.

Индекс на поля `Фамилия` и `Имя` это не то же самое, что индекс на поля `Имя` и `Фамилия`. Эти индексы имеют разный порядок полей. Например, для первого случая сортировка будет следующей:

```
Фамилия              Имя
----------------------------------------
Иванов               Андрей
Иванов               Сергей
Петров               Андрей
Петров               Василий
```

Те же самые поля, но с индексом `Имя` и `Фамилия`, будут отсортированы следующим образом:

```
Фамилия              Имя
----------------------------------------
Иванов               Андрей
Петров               Андрей
Петров               Василий
Иванов               Сергей
```

В данном случае главным является имя, и оно сортируется первым.

Составной индекс позволяет повысить производительность запросов и уменьшить количество индексов на таблицу. Производительность повышается за счет того, что сервер для поиска необходимых данных сканирует только один индекс.

Следующий пример создает некластерный составной индекс для таблицы телефонного справочника. Обратите внимание, что поле `Фамилия` описывается

первым, потому что фамилия чаще всего является основой при выборке данных из таблицы:

```
CREATE UNIQUE NONCLUSTERED INDEX I_UCL_Фамилия_Имя
ON [Телефонный справочник] (Фамилия, Имя)
```

Так как индекс уникальный, в таблицу нельзя будет записать двух людей с фамилией и именем Иванов Андрей.

Сервер SQL предлагает опции, которые могут ускорить создание индекса, а также увеличить производительность индексов.

1.2.8. Опции индексов

Опции индексов — это отдельный и очень интересный разговор, потому что с помощью опций можно повысить производительность индекса.

Опция *FILLFACTOR*

Вы можете использовать опцию `FILLFACTOR` для оптимизации производительности операций вставки и обновления в таблице, которые содержат кластерный или некластерный индекс.

Информация об индексе может содержать множество страниц с данными. Как мы уже говорили в начале этой главы, каждая страница состоит из 8 Кбайт. Когда индексная страница заполнится, сервер создаст новую страницу, а ту страницу, в которую нужно вставить данные, разделит пополам. Например, у сервера есть страница индекса для хранения 5 записей, каждая из которых указывает на следующие имена:

```
Аврора
Александр
Анатолий
Андрей
Артем
```

Допустим, что нам нужно добавить в эту страницу имя Анна. Так как в текущую страницу имя уже не помещается, она будет разделена на две:

1. Будет содержать имена Аврора, Александр и Анатолий.
2. Будет содержать Андрей и Артем.

Исходя из правил сортировки, имя Анна должно быть добавлено после имени Андрей, поэтому оно будет добавлено во вторую страницу. Так как страница может содержать до 5 записей, в каждой из страниц останется еще по две свободных строки, и сервер впоследствии сможет их использовать, когда необходимо будет добавлять новую запись.

Для разделения страниц и отведения памяти для новой страницы серверу необходимо время. Используйте опцию `FILLFACTOR` для определения процента свободного пространства на листве индексной страницы для оптимизации разделения страницы. Чем больше свободного пространства в листовых страницах, тем реже придется впоследствии делить страницы индекса. Но при этом дерево индекса будет слишком большим, и его обход будет отнимать лишнее время.

Опция `FILLFACTOR` доступна, только когда вы создаете или перестраиваете индекс. Сервер не может динамически поддерживать указанный процент свободного пространства на индексной странице.

Значение `FILLFACTOR`, которое вы указали таблице, зависит от того, как часто данные добавляются или изменяются. Если данные изменяются редко или вообще не изменяются, то можно оставить минимум свободного места в страницах, но если данные пополняются часто, и при этом вставка может происходить в середину индекса, то лучше оставить больше свободного места. Индексы на автоматически увеличиваемые поля будут только добавлять страницы в самый конец листовых данных. Новая запись не может содержать значение, меньшее существующего, значит, данные не будут вставляться в середину индекса.

Значения опции `FILLFACTOR` изменяются от 1 до 100 (это число в процентах). Значение по умолчанию равно 0. Это означает, что страница заполняется до 100%. Вы же не можете напрямую указать значение 0. Если нужен ноль, то не указывайте эту опцию.

Используйте опцию `FILLFACTOR` для таблиц, в которые часто вставляются данные, или часто изменяется индексное значение. Пример использования оператора:

```
CREATE INDEX I_vcName
ON [Телефонный справочник](vcName)
 WITH FILLFACTOR=70
```

Опция *PAD_INDEX*

Опция `PAD_INDEX` указывает процент, до которого заполняются нелистовые индексные страницы. Вы можете использовать опцию `PAD_INDEX` только тогда, когда указана опция `FILLFACTOR`, потому что процентное значение `PAD_INDEX` зависит от процента, указанного в `FILLFACTOR`.

Сервер SQL использует процент, который вы указали в опции `FILLFACTOR` для листового и нелистового уровней индекса. По умолчанию сервер всегда ос-

тавляет необходимое пространство для размещения в списке одной строки максимального размера индекса для каждой нелистовой страницы.

Указывая значение PAD_INDEX, вы должны учитывать, что количество элементов в нелистовой странице индекса не может быть меньше двух, не обращая внимания на значение FILLFACTOR.

Следующий пример создает индекс с использованием описанных параметров:

```
CREATE INDEX I_vcName
ON [Телефонный справочник](vcName)
 WITH PAD_INDEX, FILLFACTOR=70
```

Вы должны помнить, что индексы удаляются автоматически вместе с удалением таблицы и чистить их вручную ненужно.

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter1\index.sql на компакт-диске, прилагаемом к книге.

1.2.9. Вычисляемые поля

Для иллюстрации вычисляемых полей я всегда привожу пример с покупками в магазине, потому что он самый наглядный. Допустим, что у нас есть количество товара и его цена, а нужно рассчитать сумму денег, затраченных на каждый товар. Для решения задачи нужно создать таблицу из четырех полей: название, цена, количество, сумма. Давайте удалим старую таблицу TestTable и создадим новую, которая позволит решить поставленную задачу:

```
CREATE TABLE TestTable
(
  vcName varchar(50),
  iCost money,
  iNumber float,
  iSumm AS (iCost * iNumber)
)
```

В данном примере у нас создается таблица из 4 полей:

- vcName — наименование товара. Для этого поля мы указали тип varchar и ограничили ввод 50 символами;
- iCost — цена товара. Для хранения цены лучше всего подходит специализированный тип money;

- `iNumber` — количество товара. Здесь можно использовать целочисленный тип, но я выбрал число с плавающей точкой, т. е. дробный тип (`float`). Это сделано для того, чтобы можно было указывать количество товара не только в штуках, но и в килограммах, если число окажется дробным;
- `iSumm` — самое интересное кроется именно в этой колонке, потому что у нее нет типа. Вместо этого стоит ключевое слово `AS` и формула, по которой должно рассчитываться значение. В данном случае в качестве формулы указано умножение значения в колонке `iCost` на `iNumber`.

Давайте наполним таблицу строками значений и посмотрим на результат. Для этого необходимо выполнить код из листинга 1.7.

Листинг 1.7. Наполнение таблицы значениями

```
INSERT INTO TestTable(vcName, iCost, iNumber)
VALUES('Хлеб', 7.5, 2)

INSERT INTO TestTable(vcName, iCost, iNumber)
VALUES('Молоко', 17, 3)

INSERT INTO TestTable(vcName, iCost, iNumber)
VALUES('Печенье', 46, 0.5)

INSERT INTO TestTable(vcName, iCost, iNumber)
VALUES('Морковь', 12, 0.8)

INSERT INTO TestTable(vcName, iCost, iNumber)
VALUES('Картофель', 15, 5)

SELECT * FROM TestTable
```

В этом листинге в таблицу добавляется пять строк, а в последней строчке запрашиваются все строки из таблицы. Результат показан на рис. 1.8.

В листинге 1.7 во время заполнения таблицы значениями я ничего не записывал в поле `iSumm`. Тем не менее, в нем есть значения, и они являются результатом умножения цены товара на его количество.

Особое внимание нужно уделить строке 4, где показана цена моркови. Я в таблицу заносил значение 0.8, но сервер показал, что там находится число 0.80000000000000004. Почему? Это погрешность процессора Intel при хранении вещественных чисел. Если вы работали с такими языками про-

граммирования, как C++, то должны знать, что целое число 1 никогда не будет равно вещественному 1.0, потому что процессор может воспринять вещественную единицу как число 1.00000000000000001, которое уже отличается от целого числа 1. Но если округлить вещественное число, то результат будет равным 1.

	vcName	iCost	iNumber	iSumm
1	Хлеб	7.5000	2.0	15.0
2	Молоко	17.0000	3.0	51.0
3	Печенье	46.0000	0.5	23.0
4	Морковь	12.0000	0.80000000000000004	9.6000000000000014
5	Картофель	15.0000	5.0	75.0

Grids Messages

Рис. 1.8. Результат работы примера из листинга 1.7

	vcName	iCost	iNumber	iSumm
1	Хлеб	7.5000	2.00	15.000000
2	Молоко	17.0000	3.00	51.000000
3	Печенье	46.0000	.50	23.000000
4	Морковь	12.0000	.80	9.600000
5	Картофель	15.0000	5.00	75.000000

Grids Messages

Рис. 1.9. Результат работы с вычисляемыми полями

В большинстве языков программирования и в базах данных эта погрешность не убирается на уровне отображения, хотя это достаточно легко. Например, SQL-сервер мог бы обрезать последний символ.

Учитывайте эту особенность в своих проектах. Чтобы погрешность не мешала, лучше всего вместо типа `float` использовать тип `numeric` и явно указывать необходимую точность. Если бы мы поле `iNumber` объявили как `numeric(10, 2)`, то результат на экране был бы таким, как показано на рис. 1.9.

На этот раз на экране нет никакой погрешности, потому что она может быть только в последнем разряде вещественного числа, а мы используем первые два разряда. Единственный недостаток — программа при отображении вещественного числа не выводит ноль перед точкой, т. е. вместо числа "0.8" отображается ".8". Но это уже недочет отображения, а не ошибка и не погрешность, ведь расчеты все верны.

1.2.10. Создание временных таблиц

Возможности SQL не безграничны, и тем более не безгранично воображение программистов. Иногда очень сложно соорудить запрос, который выполнял бы в базе данных необходимые действия за один проход, намного проще выполнить одно действие, а затем на основании полученного результата выполнить следующее действие. Для хранения результата можно использовать временные таблицы, которые автоматически будут уничтожаться после выполнения запроса.

В MS SQL Server вы можете создавать локальные и глобальные временные таблицы. Локальные временные таблицы будут видны только вашему сеансу, а глобальные таблицы видны всем сеансам. При этом и те, и другие таблицы уничтожаются после завершения сеанса.

Чтобы таблица стала локально видимой, перед именем нужно поставить символ #. В следующем примере создается временная таблица `#TestTable` с одним полем — идентификатором, который является первичным ключом:

```
CREATE TABLE #TestTable
 (
  id INT PRIMARY KEY
 )
```

Таблица будет существовать только во время выполнения одного сеанса, и работать с ней сможете только вы. Например, вы написали большой сценарий, выполняющий определенные действия (например, процедуру, хранящуюся на сервере). Если в этом сценарии создать временную таблицу, то она будет существовать, только пока выполняется сценарий. После этого она автоматически уничтожится.

Попробуйте создать таблицу. Теперь попробуйте получить все данные из таблицы с помощью запроса:

```
SELECT * FROM #TestTable
```

В результате должна появиться пустая таблица (мы только создали таблицу, но не наполняли ее данными) из одной колонки с именем `id`.

Закройте соединение с базой данных или просто перезапустите программу, которую вы используете для отладки запросов. Снова выполните запрос выборки данных из временной таблицы. В ответ должно появиться сообщение об ошибке: *"Invalid object name '#TestTable'"* ("Неправильное имя объекта #TestTable").

Давайте снова создадим временную таблицу. Таблица будет создана, а если бы такой объект уже существовал, то мы увидели бы сообщение об ошибке.

Проверьте существование временной таблицы с помощью запроса выборки данных. Ошибок не должно быть. Откройте новое соединение или еще одну копию программы отладки запросов и выполните выборку данных из временной таблицы в этой копии программы. В ответ должно снова появиться сообщение о несуществовании объекта, потому что в другой программе создается новый сеанс, и вы не можете видеть локальные таблицы других пользователей.

Глобально видимые таблицы работают также, но в момент существования видны всем подключенным к серверу клиентам. Имена таких таблиц начинаются с двух символов ##. Следующий пример создает глобально временную таблицу:

```
CREATE TABLE ##TestTable
 (
  id INT PRIMARY KEY
 )
```

Такая таблица будет существовать, пока существует сеанс с базой данных, создавшей таблицу, и все это время любой пользователь может работать с глобально временной таблицей.

Во временных таблицах нельзя использовать внешние ключи. Видимо, это связано с тем, что из-за внешнего ключа усложняется удаление таблиц, и нужно соблюдать определенную последовательность действий. Сервер SQL компании Microsoft не может себе позволить такую роскошь, да, наверное, и другие тоже.

Несмотря на то, что мы обсуждаем тему временных таблиц, я настоятельно рекомендую вам не использовать их в своих проектах. Очень часто программисты используют временные таблицы только из-за того, что не могут объединить некоторые действия в один запрос. Цель моей книги — научить вас не просто использовать SQL, а делать это эффективно.

Единственный случай, когда я использую временные таблицы, — это выполнение следующих условий:

- таблица интенсивно пополняется или обновляется;
- отчет создается довольно долго, и все это время обрабатываемые данные должны быть статичными.

Чтобы в течение создания отчета данные были неизменными, их можно записать во временную таблицу и потом уже формировать выходные данные. Есть еще один способ решения задачи — блокировка. Можно заблокировать таблицу для изменений, но это не очень удобно, когда происходит интенсивное изменение и добавление данных. После блокировки таблицы пользователи

уже не смогут производить модификаций, что не всегда приемлемо. В такой задаче лучше подумать о возможном использовании временных таблиц.

Возможно, есть и другие задачи, в которых временные таблицы могут пригодиться, но я в своей практике с ними не встречался.

1.2.11. GUID-поля

Используйте данные типа `uniqueidentifier` (уникальный идентификатор) и функцию `NEWID()` в тех случаях, когда данные собираются из нескольких таблиц в одну большую таблицу, и нужно обеспечить уникальность всех записей. Поле типа `uniqueidentifier` гарантирует уникальность, и разработчики рекомендуют использовать именно этот тип для первичного ключа.

Тип данных `uniqueidentifier` используется для хранения глобально уникального идентификатора (GUID) в виде 16-байтной бинарной строки. Функция `NEWID()` создает новое число уникального идентификатора, которое хранится в поле типа `uniqueidentifier`.

Для данных типа `uniqueidentifier` новый ID для вставляемой строки не генерируется автоматически (в отличие от свойства `Identity`), его нужно генерировать самостоятельно с помощью функции `NEWID()`. Но если указать функцию `NEWID()` в качестве значения по умолчанию, то работа поля типа `uniqueidentifier` будет идентична `Identity`:

```
CREATE TABLE TestTable
(
 guid uniqueidentifier DEFAULT NEWID(),
 vcFirstName varchar(50),
 vcLastName varchar(50),
)
```

Но у таблиц с первичным ключом типа `uniqueidentifier` есть один большой недостаток — нельзя определить, какое поле было добавлено последним, если GUID генерируется сервером. Чтобы найти новую запись после добавления строки при первичном ключе `Identity`, приходится искать строку с максимальным первичным ключом. С ключом типа `uniqueidentifier` такой трюк не пройдет. Единственный выход — использовать следующую логику:

- сгенерировать GUID-идентификатор и сохранить его в переменной;
- добавить строку с сгенерированным GUID;
- найти строку.

Так как мы сами генерировали GIUD-значение, то без проблем сможем после добавления записи найти соответствующую строку в базе данных. О работе

с GUID-полями мы еще поговорим в *разд. 2.24*. Сейчас же нас интересует создание таблиц с уникальными идентификаторами.

Давайте посмотрим, как можно создать связанную таблицу. Следующий пример создает главную таблицу, у которой поле `idName` является первичным ключом и имеет тип `uniqueidentifier`:

```
CREATE TABLE Names
(
 idName uniqueidentifier DEFAULT NEWID(),
 vcName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY (idName),
)
```

Теперь, чтобы создать связанную таблицу, необходимо, чтобы поле внешнего ключа также имело тип `uniqueidentifier`:

```
CREATE TABLE Phones
(
 idPhone uniqueidentifier DEFAULT NEWID(),
 idName uniqueidentifier,
 vcPhone varchar(10),
 CONSTRAINT PK_idPhone PRIMARY KEY (idPhone),
 CONSTRAINT FK_idName FOREIGN KEY (idName)
   REFERENCES Names (idName)
)
```

В этой таблице уже не имеет значения, какого типа первичный ключ. Главное, чтобы тип поля внешнего ключа совпадал с типом первичного ключа основной таблицы.

1.3. Редактирование параметров базы данных

Изменения в настройках базы данных происходят нечасто, но иногда все-таки случаются. В этом разделе мы научимся вносить изменения в базу данных с помощью SQL. Для изменения используется оператор `ALTER DATABASE`. В общем виде этот оператор можно увидеть в листинге 1.8.

Листинг 1.8. Общий вид оператора `ALTER DATABASE`

```
ALTER DATABASE database
{ ADD FILE < filespec > [ ,...n ] [ TO FILEGROUP filegroup_name ]
| ADD LOG FILE < filespec > [ ,...n ]
```

```
| REMOVE FILE logical_file_name
| ADD FILEGROUP filegroup_name
| REMOVE FILEGROUP filegroup_name
| MODIFY FILE < filespec >
| MODIFY NAME = new_dbname
| MODIFY FILEGROUP filegroup_name {filegroup_property | NAME =
new_filegroup_name }

| SET < optionspec > [ ,...n ] [ WITH < termination > ]
| COLLATE < collation_name >
}
```

В первой строке параметр `database` — это имя базы данных, которую надо изменить. Далее построчно идут параметры или свойства базы данных, которые можно, а иногда и нужно, изменить. Давайте обсуждать оператор `ALTER DATABASE` с одновременным рассмотрением примеров, потому что возможностей у него столько же, сколько и у оператора создания базы данных `CREATE DATABASE`.

1.3.1. Изменение размера файла

Когда объем данных увеличивается или растет активность их изменения, вам может понадобиться увеличить размер файлов данных или журнала. Для увеличения размера файлов можно использовать программу управления базой данных SQL Server Enterprise Manager, которая обладает визуальным интерфейсом, или воспользоваться оператором `ALTER DATABASE`.

Вы можете контролировать размер базы данных следующими способами:

- сконфигурировать файлы базы данных и журнала на автоматическое приращение;
- увеличивать или уменьшать текущее значение максимального размера файлов базы данных и журнала вручную;
- вручную добавлять дополнительные файлы журнала или данных.

Вы можете установить автоматическое приращение с помощью оператора `ALTER DATABASE`. В этом случае администратору не придется вручную увеличивать размер базы данных, но будет сложнее контролировать размер пространства, используемого сервером MS SQL.

Увеличение лучше всего контролировать самостоятельно. Да, это отнимет лишнее время и потребует некоторых усилий, но контроль никогда не бывает

лишним. Спокойно спит только тот администратор, у которого все находится под контролем.

Если вы не сконфигурировали существующие файлы на автоматическое приращение, вы все еще можете увеличить размер. Если в качестве значения для автоматического увеличения указан ноль, значит, автоматическое увеличение не происходит.

Вы можете создать вторичные файлы базы данных для расширения размера базы данных. Но лучше использовать дополнительные файлы базы данных, чтобы расположить файлы на отдельном физическом диске или выделить объект базы данных из общей массы.

Следующий пример увеличивает текущий размер журнала базы данных NewDB:

```
ALTER DATABASE NewDB
MODIFY FILE
(
  NAME='NewDB',
  SIZE = 100mb
)
```

Теперь попробуем добавить дополнительный файл. Да, и это возможно. Допустим, что имеется свободное пространство на жестком диске, но у вас есть дисковый массив, которым можно управлять "на лету". Прямо во время работы вставляем новый диск, создаем на нем файл для базы данных и больше ни о чем не заботимся. Следующий пример показывает, как можно добавить файл к базе данных Sample.

```
ALTER DATABASE Sample1
ADD FILE
(
  NAME='SampleData2',
  FILENAME='c:\Temp\SampleData2.mdf',
  SIZE = 15mb,
)
```

Вот тут необходимо сделать одно замечание: если для переноса данных с одного сервера на другой вы захотите воспользоваться функциями отключения и подключения базы данных, то не забудьте скопировать в новое место все файлы.

Допустим, что вам необходимо произвести массовую загрузку данных. База данных сейчас занимает 10 Мбайт, а приращение, указанное в настройках,

составляет 1 Мбайт. Если вы будете загружать еще 10 Мбайт данных, то во время загрузки произойдет 10 увеличений файла данных по одному мегабайту. Это отнимет дополнительное время, поэтому вполне логичным будет увеличить размер файла вручную.

Для ручного увеличения размера файла до 200 Мбайт необходимо выполнить следующую команду:

```
ALTER DATABASE testdatabase
MODIFY FILE
(
 NAME = testdatabase,
 SIZE = 200MB
)
```

Вторая строка содержит команду `MODIFY FILE`, что означает изменение файла. В скобках необходимо указать имя файла и новый размер. Новый размер обязательно должен быть больше текущего, потому что команда позволяет только увеличивать файл, но не уменьшать.

Если вы хотите узнать текущий размер файлов или не помните имена, выполните команду:

```
EXEC sp_helpdb имя
```

где `имя` — имя базы данных. Например, следующая команда отображает информацию о базе данных FlenovSQLBook, в которую входит информация и о файлах:

```
EXEC sp_helpdb FlenovSQLBook
```

На рис. 1.10 показан результат выполнения команды

```
EXEC sp_helpdb TestDatabase
```

	name	db_size	owner	dbid	created	status	compatibility_level
1	TestDatabase	20.75 MB	NOTEBOOK\Administrator	7	Apr 25 2005	Status=ONLINE, Updateabi...	80

	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	TestDatabase	1	C:\Program Files\Microso...	PRIMARY	20480 KB	Unlimited	10%	data only
2	TestDatabase_log	2	C:\Program Files\Microso...	NULL	768 KB	Unlimited	10%	log only

Рис. 1.10. Информация о состоянии базы данных

Уменьшение файлов данных и журнала — это отдельная тема. Итак, если вы создали базу данных с файлом в 1 Мбайт и потом увеличили ее размер до 5 Мбайт, то вернуть изначальный размер командой `ALTER DATABASE` невозможно.

Это значит, что следующий запрос вернет сообщение об ошибке, если размер файла FlenovSQLBook больше 2 Мбайт:

```
ALTER DATABASE FlenovSQLBook
MODIFY FILE
(
 NAME = FlenovSQLBook,
 SIZE = 1MB
)
```

С помощью команды `ALTER DATABASE` можно изменять и размер файла журнала, только в параметре `NAME` нужно указать имя файла журнала, который следует увеличить:

```
ALTER DATABASE TestDatabase
MODIFY FILE
(
 NAME = testdatabase_log,
 SIZE = 2MB
)
```

Чтобы добиться оптимальной производительности сервера и уменьшить темпы роста файлов (это отнимает лишнее время и может затормозить работу сервера):

- выделяйте достаточно начального пространства, чтобы избежать частого автоматического приращения;
- указывайте максимальный размер файлов данных, если у вас есть несколько баз данных;
- устанавливайте размер приращения файлов данных и журнала так, чтобы не было частого автоматического увеличения.

При увеличении базы данных или повышении активности изменений у вас может возникнуть необходимость в расширении журнала транзакций. Внимательно следите, чтобы журнал транзакций не оказался слишком маленьким. Регулярное наблюдение за журналом позволит вам определить оптимальный размер увеличения журнала и не допустить его переполнения. Если ваш журнал транзакций не может быть увеличен, то SQL Server не сможет записывать транзакции и не позволит делать изменения в базе данных.

Вы можете следить за журналом транзакций с помощью SQL Server Enterprise Manager, оператора `DBCC SQLPERF (LOGSPACE)` (подробнее о команде рассказывается в *разд. 4.2.5*) или системного монитора Windows.

Если данные изменяются достаточно часто, то необходимо выделить максимально возможное (но разумное) пространство и следить, чтобы всегда было достаточно свободного места в журнале даже для выполнения самой массовой операции обновления данных. Если база используется редко и изменения происходят лишь время от времени, то журнал можно сделать небольшим. Помните, что журнал транзакций пополняется только во время добавления, изменения и удаления данных. Во время выборки журнал не используется.

Перечислю некоторые ситуации, повышающие активность журнала транзакций:

- загрузка информации в таблицу, которая содержит индексы. Сервер SQL записывает все вставки и изменения индексов. При загрузке таблиц без индексов SQL Server записывает в журнал только расширение пространства;
- транзакции, которые выполняют много изменений (`INSERT`, `UPDATE` и `DELETE`) в таблице с помощью одной транзакции;
- добавление или изменение данных типа `Image` или `Text`.

1.3.2. Добавление и удаление файла

Когда вы используете автоматическое увеличение и база данных состоит из нескольких файлов, SQL Server применяет стратегию пропорционального заполнения всех файлов каждой файловой группы. Тут есть и положительные, и отрицательные моменты. Положительные видны сразу, ведь если диски подключены к разным контроллерам, то мы добиваемся параллельности, а жесткие диски являются самым слабым звеном. Но нежелательная ситуация может возникнуть при автоматическом увеличении файлов. Чаще всего размеры дисков разные, да и свободное пространство на них распределяется неравномерно (если на один диск установить пару игр, то свободного места будет меньше, чем на другом). Теперь один диск заполнится раньше, чем другой. Нет, база данных в этом случае останется доступной, и запись будет вестись только на один диск, но произойдет резкое падение производительности во время записи.

Допустим, что на вашем основном диске, где расположена база данных, заканчивается свободное место. В этом случае, если есть другие логические разделы или жесткие диски, можно добавить к серверу новый файл и расположить его на этом диске. Добавление чем-то похоже на определение файла при создании базы данных:

```
ALTER DATABASE TestDatabase
ADD FILE
(
```

```
 NAME = dbFile2,
 FILENAME = 'c:\Data\dbfile2.ndf',
 SIZE = 5MB,
 FILEGROWTH = 5MB
)
```

Здесь изменяется уже существующая база данных, на которой мы тестировали не один пример, — `TestDatabase`. Далее идет оператор `ADD FILE`, добавляющий новый файл. После этого ключевого слова в круглых скобках перечисляются параметры файла точно так же, как при описании файла во время создания базы данных.

В качестве расширения файла выбрано ndf. Оно рекомендуется для всех вторичных файлов, но может быть и другим. Напоминаю, что для основного файла рекомендуется указывать расширение mdf.

Для удаления файла из базы данных совместно с оператором `ALTER DATABASE` используется оператор `REMOVE FILE`. Если вы создали файл и поместили в него таблицу, то файл уже нельзя удалить, пока он не станет пустым, т. е. пока из файла не будут удалены все таблицы.

Нам в данной книге лишний файл не нужен, поэтому давайте удалим то, что создали. Для этого напишем оператор `REMOVE FILE` и имя удаляемого файла:

```
ALTER DATABASE TestDatabase
REMOVE FILE dbFile2
```

1.3.3. Добавление и удаление файловых групп

Для добавления файловой группы используется оператор `ADD FILEGROUP`. Например, следующий сценарий добавляет группу с именем `fgNewGroup`:

```
ALTER DATABASE TestDatabase
ADD FILEGROUP fgNewGroup
```

Теперь посмотрим, как можно добавить файл в эту группу:

```
ALTER DATABASE TestDatabase
ADD FILE
(
 NAME = dbFile2,
 FILENAME = 'c:\Data\dbfile2.ndf',
 SIZE = 5MB,
 FILEGROWTH = 5MB
) TO FILEGROUP fgNewGroup
```

Принцип такой же, как и у добавления файла, но после скобок указывается `TO FILEGROUP` и имя группы, в которую добавляется файл.

Файловую группу нельзя удалять, если в ней есть файл. Сначала необходимо удалить этот файл, а потом уже удалять группу. Для удаления группы используется оператор `REMOVE FILEGROUP`, например:

```
ALTER DATABASE TestDatabase
REMOVE FILEGROUP fgNewGroup
```

1.3.4. Переименование базы данных

Иногда возникает необходимость переименования базы данных. В моей практике это приходилось делать очень редко, но все же я затрону этот вопрос. Переименовать можно с помощью оператора `MODIFY NAME`. Например, следующий сценарий изменяет имя базы данных `TestDatabase` на `MyDatabase`:

```
ALTER DATABASE TestDatabase
MODIFY NAME = MyDatabase
```

При этом вы не должны быть подключены к этой базе данных, лучше всего, если подключение будет к базе данных master. Если к базе данных, которую необходимо переименовать, будет подключен хоть один пользователь, то переименование не сможет быть выполнено.

Если вы попробовали выполнить этот сценарий, то верните ей старое имя `TestDatabase`, потому что в дальнейшем при тестировании сценариев мы будем ссылаться на него.

1.3.5. Изменение свойств базы данных

У базы данных существует множество свойств, которые мы не задавали во время создания, но которые можно изменить уже у существующей базы. К таким свойствам относятся уровень доступа, модель восстановления и т. д. Давайте рассмотрим, что и как можно изменять.

Для изменения свойства используется оператор `SET`. Команда будет выглядеть следующим образом:

```
ALTER DATABASE Имя_базы
SET имя_свойства
```

После `ALTER DATABASE` указывается имя базы данных, свойства которой нужно изменить, а после оператора `SET` нужно указать имя свойства.

Перечислим имена свойств, которые нужно подставить вместо параметра `имя_свойства`:

- `SINGLE_USER` — перевести базу данных в однопользовательский режим. Только один пользователь сможет работать с базой;

- RESTRICTED_USER — разрешить подключаться к базе данных только пользователям ролей db_owner, dbcreator или sysadmin;
- MULTI_USER — установить нормальный многопользовательский режим, при котором действуют все права (используется по умолчанию);
- OFFLINE — отключить базу данных, подключения будут невозможны. Команда должна выполняться, когда к базе данных нет активных подключений. Вы при этом должны быть подключены к базе данных master;
- ONLINE — вернуть базу данных в активное состояние;
- READ_ONLY — перевести базу данных в режим только для чтения, теперь изменение данных будет невозможно;
- READ_WRITE — вернуть базе данных полный доступ на запись и чтение;
- CURSOR_CLOSE_ON_COMMIT ON — закрыть все открытые курсоры по завершении транзакции (принятии или откате). Если ON заменить на OFF, то при нормальном завершении транзакции (принятии изменений) курсоры остаются открытыми. При откате все курсоры, кроме INSENSITIVE и STATIC, закрываются;
- RECOVERY FULL — использовать полную модель восстановления;
- BULK_LOGGED — установить модель восстановления BULK_LOGGED;
- SIMPLE — установить простую модель восстановления.

Это основные параметры, которые можно изменить. Более подробно о моделях восстановления можно узнать из файла \Doc\BackupRestore.pdf на компакт-диске, прилагаемом к книге.

Теперь рассмотрим примеры использования этих свойств.

Разрешить доступ только одному пользователю:

```
ALTER DATABASE TestDatabase
SET SINGLE_USER
```

Разрешить доступ только пользователям ролей db_owner, dbcreator или sysadmin:

```
ALTER DATABASE TestDatabase
SET RESTRICTED_USER
```

Возвратить нормальный многопользовательский режим:

```
ALTER DATABASE TestDatabase
SET MULTI_USER
```

Перевести базу данных в режим off-line, т. е. доступ будет запрещен всем пользователям:

```
ALTER DATABASE TestDatabase
SET OFFLINE
```

Возобновить доступ к базе данных:

```
ALTER DATABASE TestDatabase
SET ONLINE
```

Перевести базу данных в режим только для чтения, любые изменения будут отклонены:

```
ALTER DATABASE TestDatabase
SET READ_ONLY
```

Вернуть базе данных полный доступ на запись и чтение:

```
ALTER DATABASE TestDatabase
SET READ_WRITE
```

По завершении транзакции (принятии или откате) закрыть все открытые курсоры:

```
ALTER DATABASE TestDatabase
SET CURSOR_CLOSE_ON_COMMIT ON
```

Установить полную модель восстановления:

```
ALTER DATABASE TestDatabase
SET RECOVERY FULL
```

Установить модель восстановления `BULK_LOGGED`:

```
ALTER DATABASE TestDatabase
SET BULK_LOGGED
```

Установить простую модель восстановления:

```
ALTER DATABASE TestDatabase
SET SIMPLE
```

И последнее, что нам предстоит узнать, — это возможность изменения раскладки (кодировки) по умолчанию для базы данных. Для этого выполняется команда:

```
ALTER DATABASE Имя_базы
COLLATE имя_кодировки
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter1\alter_database.sql на компакт-диске, прилагаемом к книге.

1.4. Редактирование таблиц

До настоящего момента, если нам нужно было внести изменения в таблицу, мы удаляли ее и создавали с новыми параметрами. Это не выход, потому что если таблица содержит данные, то очищать содержимое ради изменения структуры не эффективно (если не сказать больше, но мы промолчим).

Для изменения параметров таблицы используется оператор `ALTER TABLE`. Рассмотрим его по частям. Синтаксис оператора показан в листинге 1.9.

Листинг 1.9. Синтаксис оператора `ALTER TABLE`

```
ALTER TABLE table
{ [ ALTER COLUMN column_name
    { new_data_type [ ( precision [ , scale ] ) ]
        [ COLLATE < collation_name > ]
        [ NULL | NOT NULL ]
        | {ADD | DROP } ROWGUIDCOL }
    ]
    | ADD
        { [ < column_definition > ]
        |  column_name AS computed_column_expression
        } [ ,...n ]
    | [ WITH CHECK | WITH NOCHECK ] ADD
        { < table_constraint > } [ ,...n ]
    | DROP
        { [ CONSTRAINT ] constraint_name
            | COLUMN column } [ ,...n ]
    | { CHECK | NOCHECK } CONSTRAINT
        { ALL | constraint_name [ ,...n ] }
    | { ENABLE | DISABLE } TRIGGER
        { ALL | trigger_name [ ,...n ] }
}

< column_definition > ::=
    { column_name data_type }
    [ [ DEFAULT constant_expression ] [ WITH VALUES ]
    | [ IDENTITY [ (seed , increment ) [ NOT FOR REPLICATION ] ] ]
        ]
    [ ROWGUIDCOL ]
```

```
    [ COLLATE < collation_name > ]
    [ < column_constraint > ] [ ...n ]

< column_constraint > ::=
    [ CONSTRAINT constraint_name ]
    { [ NULL | NOT NULL ]
        | [ { PRIMARY KEY | UNIQUE }
            [ CLUSTERED | NONCLUSTERED ]
            [ WITH FILLFACTOR = fillfactor ]
            [ ON { filegroup | DEFAULT } ]
            ]
        | [ [ FOREIGN KEY ]
            REFERENCES ref_table [ ( ref_column ) ]
            [ ON DELETE { CASCADE | NO ACTION } ]
            [ ON UPDATE { CASCADE | NO ACTION } ]
            [ NOT FOR REPLICATION ]
            ]
        | CHECK [ NOT FOR REPLICATION ]
            ( logical_expression )
    }

< table_constraint > ::=
    [ CONSTRAINT constraint_name ]
    { [ { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
        { ( column [ ,...n ] ) }
        [ WITH FILLFACTOR = fillfactor ]
        [ ON {filegroup | DEFAULT } ]
        ]
        |    FOREIGN KEY
            [ ( column [ ,...n ] ) ]
            REFERENCES ref_table [ ( ref_column [ ,...n ] ) ]
            [ ON DELETE { CASCADE | NO ACTION } ]
            [ ON UPDATE { CASCADE | NO ACTION } ]
            [ NOT FOR REPLICATION ]
        | DEFAULT constant_expression
            [ FOR column ] [ WITH VALUES ]
        |    CHECK [ NOT FOR REPLICATION ]
            ( search_conditions )
    }
```

Прежде чем рассматривать возможности по изменению существующей таблицы, создадим простую таблицу для хранения информации о жителях дома. Назовем ее `Peoples`. Так как в таблице должно быть хотя бы одно поле, создадим ключевое поле. Я всегда называю поле первичного ключа `idИмяТаблицы`, т. е. для таблицы `Peoples` первичный ключ должен называться `idPeoples`. Для простоты дадим этому полю целочисленный тип `int`:

```
CREATE TABLE Peoples
(
 idPeoples int
)
```

Теперь посмотрим, как нарастить возможности таблицы. Мы уже поняли, что для этого нужно использовать оператор `ALTER TABLE` и указать имя изменяемой таблицы. Примерно так же мы изменяли базу данных.

1.4.1. Добавление новых полей

Новое поле добавляется с помощью оператора `ADD`, где через запятую перечисляются добавляемые поля, как при создании таблицы данных. Следующий пример добавляет в таблицу поля для хранения фамилии, имени и отчества:

```
ALTER TABLE Peoples
ADD
 vcFamil VARCHAR(30) NOT NULL default '',
 vcName VARCHAR(30) NOT NULL default '',
 vcSurName VARCHAR(30) NULL
```

Фамилия и имя обязательно присутствуют у человека, а вот отчества может и не быть (ведь у многих народов просто нет такого понятия, а в нашей базе могут оказаться люди самых разных национальностей). Поэтому мы должны отразить в таблице то, что фамилия и имя не могут содержать нулевые значения, а поле для хранения отчества может содержать нулевые значения.

Если попытаться добавить поле, в котором нулевые значения недопустимы, и не указать значение по умолчанию, то поле не будет добавлено. Вместо этого вы увидите сообщение об ошибке:

ALTER TABLE only allows columns to be added that can contain nulls or have a DEFAULT definition specified. Column 'vcFamil' cannot be added to table 'Peoples' because it does not allow nulls and does not specify a DEFAULT definition.

ALTER TABLE позволяет добавлять колонки, только если они могут содержать нулевые значения, или имеют объявления значения по умолчанию (DEFAULT). Колонка "vcFamil" не может быть добавлена в таблицу Peoples, потому что не позволяет нулевых значений и не имеет значения по умолчанию.

Почему такое ограничение? Дело в том, что если добавить новое поле в таблицу, уже содержащую строки, то в существующих строках будет установлено в новой колонке нулевое значение или значение по умолчанию. Если поле не поддерживает нулевых значений и не указано значение по умолчанию, что тогда вставлять в новое поле для существующих строк? Нулевое значение нельзя, а какое другое значение указать, сервер не знает. Вот из-за этого конфликта и произошло такое ограничение.

При добавлении фамилии и имени мы указываем значение по умолчанию — пустую строку (" — между кавычками ничего нет, даже пробела). Это лишний раз доказывает, что пустая строка и нулевое значение — разные вещи, и сервер воспринимает их по-разному. А для отчества у нас разрешены нулевые значения.

Конечно же, в данном случае теряется смысл запрета нулевых значений для фамилии и имени, если есть значение по умолчанию. Если пользователь не укажет фамилию, то база данных воспримет это без ошибок и просто установит пустую строку, что, в принципе, является ошибкой. Это серьезный недостаток, но с ним можно справиться, если установить дополнительное ограничение на таблицу (`CHECK`), которое запретит пустые строки. Ограничение срабатывает только при добавлении строк, а на существующие не влияет, поэтому тут никаких конфликтов не возникнет. Об изменении ограничений мы поговорим в *разд. 1.4.3*.

1.4.2. Удаление полей

Для удаления полей нужно использовать оператор `DROP COLUMN`, после которого указывается имя удаляемой колонки. Например, следующий сценарий удаляет из таблицы колонку `vcSurName`:

```
ALTER TABLE Peoples
DROP COLUMN vcSurName
```

Я думаю, что все понятно и без дополнительных комментариев. После `ALTER TABLE` просто пишем оператор `DROP COLUMN` и имя удаляемой колонки.

1.4.3. Изменение ограничений

Ограничения можно добавлять не только на этапе создания таблицы, но и во время работы с ней. Давайте добавим поле для хранения даты, а потом отдельной командой добавим ограничение. Добавление поля:

```
ALTER TABLE Peoples
ADD dDate datetime NULL
```

Устанавливаем ограничение, чтобы дата не была больше текущей:

```
ALTER TABLE Peoples WITH NOCHECK
ADD CONSTRAINT check_dDate CHECK
   (dDate < getdate())
```

Как видите, для добавления ограничения используется оператор `ADD CONSTRAINT`. После этого указывается имя ограничения и само ограничение `CHECK`. Для удаления используется оператор `DROP CONSTRAINT`, которому нужно указать имя ограничения:

```
ALTER TABLE Peoples
DROP CONSTRAINT check_dDate
```

Далеко не всегда нужно удалять ограничения. Бывают случаи, когда можно обойтись простым отключением. Например, на поле даты таблицы `Peoples` действует ограничение на ввод даты, которое мы рассматривали ранее, т. е. невозможно ввести дату более текущей. При необходимости ограничение можно обойти, но следует действовать аккуратно.

Итак, давайте вернем ограничение даты:

```
ALTER TABLE Peoples WITH NOCHECK
ADD CONSTRAINT check_dDate CHECK
   (dDate < getdate())
```

Попробуем добавить строку с датой 10.10.2010. Я думаю, что когда вы будете читать эту книгу, этот год еще не наступит.

```
INSERT INTO Peoples(idPeoples, vcFamil, vcName, dDate)
VALUES(1, 'Иванов', 'Иван', '10.10.2010')
```

В ответ на это мы получим сообщение о попытке нарушить ограничение, и строка не будет добавлена. Давайте временно отключим ограничение (в данном случае это `check_dDate`):

```
ALTER TABLE Peoples
NOCHECK CONSTRAINT check_dDate
```

Теперь еще раз попытайтесь добавить строку с датой больше текущей. Все должно пройти успешно.

Снова включаем ограничение. Для этого используется оператор `CHECK CONSTRAINT`, которому нужно указать имя ограничения:

```
ALTER TABLE Peoples
CHECK CONSTRAINT check_dDate
```

Попробуйте выполнить пример. Несмотря на то, что при включении ограничения в таблице окажется ошибочная запись, никакой ошибки не произойдет.

Дело в том, что ограничение срабатывает только в момент добавления записи. Если запись уже существует, то ошибки не будет. Вы должны учитывать это и вручную проверять корректность данных при включении ограничения.

Вспомним, что в таблице есть поля для хранения фамилии и имени, для которых запрещены нулевые значения, но нам пришлось установить значение по умолчанию. Это серьезный недостаток. Давайте устраним его с помощью ограничения `CHECK`, которое запретит ввод пустых строк в эти два поля:

```
ALTER TABLE Peoples
ADD CONSTRAINT check_vcFamil
  CHECK (vcFamil <> '')
```

Теперь попробуйте добавить строку, где в поле `vcFamil` будет пустая строка:

```
INSERT INTO Peoples (vcFamil)
VALUES ('')
```

В ответ на это сервер вернет нам ошибку ограничения `check_vcFamil`.

В данном случае мы явно пытаемся прописать в поле `vcFamil` пустую строку. А что, если это поле вообще не указать при добавлении новой записи? В следующем примере добавляется новая строка, но при этом мы заполняем только поле `vcName`:

```
INSERT INTO Peoples (vcName)
VALUES ('')
```

И все равно сработает ограничение `check_vcFamil`. Дело в том, что для поля `vcFamil` установлено значение по умолчанию — пустая строка. Если при добавлении новой записи значение не указано, будет взято значение по умолчанию, а т. к. ограничение сработало, значит, проверка ограничений происходит после того, как база данных установит для полей значения по умолчанию. Таким образом, мы реализовали необходимую защиту от нулевых и пустых значений. Конечно, это стоило нам некоторых дополнительных усилий, и появились лишние затраты сервера на установку значения по умолчанию и вызов проверки ограничения `CHECK`, но эти затраты не так уж и велики. Да, необходимость так выкручиваться — это минус для MS SQL Server, но на него можно закрыть глаза.

1.4.4. Изменение поля

Зачем изменять уже существующее поле? На первый взгляд незачем, но в реальности оказывается, что это действительно нужно. Например, вы указали, что поле должно иметь размер 20 символов, но в определенный момент оказалось, что этого недостаточно и необходимо расширить размер поля.

Для этого используется оператор `ALTER COLUMN`. Но тут есть свои подводные камни. Попробуйте изменить размер имени или фамилии в созданной нами таблице `Peoples`:

```
ALTER TABLE Peoples
ALTER COLUMN vcName NVARCHAR(40) NOT NULL
```

Я уверен, что вы увидите сообщение об ошибке типа:

The object 'DF__Peoples__vcName__72C60C4A' is dependent on column 'vcName'.

ALTER TABLE ALTER COLUMN vcName failed because one or more objects access this column

В первой строке говорится, что объект `DF__Peoples__vcName__72C60C4A` ссылается на колонку `vcName`. Во второй строке сообщения нам говорят, что команда не выполнена, потому что один или более объектов ссылаются на выбранное поле.

Выполним команду `sp_help peoples`, чтобы узнать, кто и зачем ссылается на выбранную таблицу:

```
Exec sp_help peoples
```

В результате мы увидим несколько таблиц с параметрами таблицы `Peoples`. В самой последней можно увидеть указанное имя объекта в колонке `constraint_name`. Вот в чем проблема — на поле ссылается ограничение, которое проверяет наличие пустого значения, ведь мы указали для поля `vcName` свойство `NOT NULL`. Сначала необходимо удалить это ограничение, и только потом можно будет изменить размер поля:

```
ALTER TABLE Peoples
DROP CONSTRAINT DF__Peoples__vcName__72C60C4A

ALTER TABLE Peoples
ALTER COLUMN vcName NVARCHAR(40) NOT NULL
```

На этом закончим рассмотрение оператора `ALTER TABLE`. У него также есть возможность включения и отключения триггеров, но мы эту тему пока пропустим, потому что о триггерах еще абсолютно ничего не знаем.

1.5. Обеспечение целостности данных

Во время проектирования базы данных вы должны заботиться о целостности данных. Правильная структура таблиц позволяет защитить данные от нарушения связей и внесения неверных значений. Вы должны определить

наилучший путь обеспечения целостности данных. Целостность данных основывается на стойкости и точности данных, которые хранит база данных.

Существуют различные типы целостности данных.

- Целостность полей указывает набор значений данных, которые являются правильными для поля, и определяет, возможно ли использование нулевого значения. Например, поле для хранения пола человека может содержать одно из двух значений — `М` или `Ж`. Во-первых, этого достаточно, во-вторых, других значений пола просто не бывает, и мы должны запретить ввод других букв в данное поле. Целостность полей часто всего (и лучше) обеспечивается с помощью ограничения `CHECK`, формата (с помощью шаблона) или диапазона возможных значений для поля.
- Целостность таблицы требует, чтобы все строки в таблице имели уникальный идентификатор, называемый первичным ключом. Может ли первичный ключ изменяться, или может ли строка удаляться, зависит от уровня целостности. Например, в некоторых случаях можно разрешить удаление записей, но чаще всего оно должно быть запрещено. Нежелательно терять данные, потому что мы впоследствии не сможем узнать историю изменений в таблице.
- Целостность ссылок подразумевает отношения между первичным ключом (таблицы, на которую ссылаются) и внешним ключом (таблицы, которая ссылается на другую) всегда защищенными. Строка основной таблицы, на которую ссылаются, не может быть удалена, и первичный ключ не может быть изменен, если внешний ключ ссылается на строку, пока не будет уничтожена связь. Иначе связь нарушится, и восстановление ее станет проблематичным. Вы можете назначить отношения внутри таблицы или между несколькими отдельными таблицами с помощью встроенных в SQL Server средств, не надеясь на возможности языка программирования, который вы используете для доступа к данным. Конечно же, связь между таблицами можно навести и без внешних ключей, но в этом случае сервер не гарантирует целостность. Вся ответственность ложится на программиста.

Операторы, необходимые для реализации всех уровней целостности, нам уже знакомы. Я специально вынес рассмотрение теории обеспечения целостности за рамки описания языковых средств. Знание операторов упростит понимание теоретических основ.

Как мы можем гарантировать целостность данных? Для этого существуют два способа: описанная целостность данных и предшествующая целостность данных. Пока эти понятия неясны, но после того, как вы увидите, как реализуется тот или иной способ, все встанет на свои места.

Описанная целостность данных — вы объявляете критерии, которые данные должны содержать как часть описания объекта, и после этого SQL Server автоматически гарантирует, что данные соответствуют критериям. Уже можно догадаться, что такая целостность обеспечивается с помощью ограничений `CHECK`, `DEFAULT` и внешнего ключа.

Описанная целостность является частью объявления базы данных и объявляется с помощью ограничений, которые вы можете назначить колонкам и таблицам напрямую.

Предшествующая целостность данных — это программа, которая определяет критерии, которым должны соответствовать данные. Этот метод обеспечивается с помощью процедур и триггеров (о них мы поговорим в *главе 3*), которые могут выполняться на сервере, или с помощью программного кода в клиентском приложении.

Вы должны минимизировать использование этого метода для упрощения бизнес-логики и ошибок, но иногда без триггера невозможно гарантировать, что таблицы будут содержать нужные или разрешенные значения.

Ограничение — это основной метод обеспечения целостности данных. В этой секции указывается тип ограничения, какой тип данных и для какого ограничения следует использовать и как определить ограничения.

Ограничение — метод ANSI-стандарта для обеспечения целостности данных. Каждый тип целостности данных (доменный, табличный и ссылочный) реализуется отдельным типом ограничений. Ограничения гарантируют правильность данных, введенных в поле, и корректность отношений между таблицами.

Как мы уже знаем, ограничения могут создаваться во время создания таблицы (`CREATE TABLE`) или редактирования (`ALTER TABLE`). Если ограничение назначается отдельному полю, оно называется ограничением уровня поля. Если ограничение ссылается на несколько полей, оно называется ограничением уровня таблицы, даже если оно ссылается не на все колонки таблицы.

Имена ограничений должны быть уникальными для базы данных. Если не указывать имена, то сервер сгенерирует значение самостоятельно, но как мы видели в *разд. 1.2.4*, имя получится слишком сложным и не будет отражать суть происходящей проверки. Поэтому я рекомендую всегда указывать имена самостоятельно и так, чтобы они отражали суть происходящей проверки и содержимого таблицы, в которой выполняется проверка.

Теперь рассмотрим уже знакомые нам операторы ограничений с точки зрения обеспечения целостности данных. В некоторых случаях мы узнаем что-то новое, а в других просто закрепим пройденный материал.

1.5.1. Ограничение *DEFAULT*

Ограничение DEFAULT помещает значение в колонку, когда оно не было указано в операторе INSERT. Оно относится только к оператору добавления записи (INSERT) и не срабатывает во время изменения полей (оператор UPDATE). Таким образом, данное ограничение не гарантирует, что поле содержит значение. Пользователь может добавить строку и потом с помощью оператора UPDATE обнулить содержимое поле со значением по умолчанию.

Таким образом, DEFAULT является самым простым и быстрым по скорости выполнения методом обеспечения целостности, но не является гарантом. Необходимы дополнительные средства, например, ограничение на диапазон вводимых значений или триггер. В листинге 1.10 помимо значения DEFAULT мы создаем ограничение, которое не позволяет записывать в поле нулевые значения, что защитит нас от возможности записи в поле NULL даже при обновлении данных.

Листинг 1.10. Создание таблицы с ограничением DEFAULT и CHECK

```
-- Создание таблицы
CREATE TABLE TestTable
(
 iID int DEFAULT 1,
 CONSTRAINT check_iID CHECK (iID is NOT NULL)
)

-- Добавление записи с числом 10 в колонке iID
INSERT INTO TestTable
VALUES (10)

-- Обновление существующих записей, в поле iID
-- записывается нулевое значение
UPDATE TestTable
SET iID=NULL
```

В данном примере мы устанавливаем сразу два ограничения на поле iID таблицы TestTable. Первое DEFAULT устанавливает значение по умолчанию, если во время добавления записи для поля iID не было указано значение. Вторая проверка CHECK не позволит сделать поле нулевым с помощью операции обновления записей.

После этого в листинге показаны примеры добавления и обновления записи. Во время обновления мы пытаемся записать в поле значение NULL. В ответ на это сервер сгенерирует ошибку и сообщит, что сработало ограничение check_iID.

1.5.2. Ограничение *CHECK*

Ограничение CHECK указывает допустимые значения для данных, которые пользователь может ввести в определенную колонку. Следующий пример добавляет ограничение, чтобы гарантировать, что дата рождения попадает в указанный диапазон:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 vcName varchar(50) NOT NULL,
 dBirthDate datetime,
 CONSTRAINT CK_birthdate
  CHECK (dBirthDate > '01-01-1900'
         AND dBirthDate<getdate())
)
```

В данном примере создается таблица для хранения имен и дат рождений dBirthDate. Год рождения не может быть меньше 1900. Людей, которым более 105 лет, на Земле осталось не так много, и вероятность того, что такие люди попали в нашу базу, стремится к нулю.

Как мы уже знаем, ограничение CHECK срабатывает не только при вставке данных, но и при обновлении.

При создании ограничения можно ссылаться и на другие таблицы. Например, в следующей таблице, помимо даты рождения человека, будет храниться дата выдачи паспорта — dDocDate. Вполне логично, что дата выдачи паспорта должна быть больше даты рождения и меньше текущей. Паспорт не может быть выдан до рождения, поэтому в следующей таблице мы гарантируем целостность поля даты рождения:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1 NOT NULL,
 vcName varchar(50) NOT NULL,
 dBirthDate datetime,
 dDocDate datetime,
 CONSTRAINT CK_birthdate
```

```
  CHECK (dDocDate>dBirthDate
        AND dBirthDate<getdate())
)
```

Мы еще не обсуждали запросы выборки данных и подзапросы, но необходимо заметить, что в ограничении CHECK нельзя использовать подзапросы.

1.5.3. Ключи

Ограничение PRIMARY KEY определяет первичный ключ таблицы, который уникально идентифицирует строку. Это гарантирует целостность таблицы. Когда мы изучали оператор PRIMARY KEY, то уже рассматривали примеры, и здесь добавить нечего. Давайте только суммируем все вышесказанное и перечислим свойства первичного ключа:

- в таблице может быть только один первичный ключ, но этот ключ может состоять из нескольких полей;
- поля первичного ключа не могут содержать нулевое значение;
- для полей первичного ключа создается индекс, который может быть кластерным или некластерным;
- несмотря на то, что для полей первичного ключа не создается ограничение уникальности, оно подразумевается, и две записи не могут содержать одинаковых значений в первичном ключе.

Ограничение FOREIGN KEY (внешний ключ) гарантирует ссылочную целостность. Ограничение внешнего ключа определяет ссылку на колонку с первичным ключом или уникальную колонку в этой же или другой таблице. С помощью такого ключа обеспечивается целостность связей между таблицами.

Внешний ключ, как и первичный, может состоять из нескольких полей. При создании связующего ключа количество колонок внешнего ключа должно соответствовать количеству колонок первичного ключа, с которым происходит связь. Кстати, связываться можно не только с первичным ключом, но и с полем, содержащим ограничение уникальности.

Если в связующих таблицах достаточно много строк, то я рекомендую добавить к внешнему ключу еще и индекс. Дело в том, что для внешнего ключа индекс автоматически не создается. Благодаря индексу сервер сможет быстрее найти связанные строки в разных таблицах.

Ограничение внешнего ключа включает опцию CASCADE, которая позволяет любые изменения, сделанные в уникальной колонке или первичном ключе, автоматически переносить в значение внешнего ключа. Такое действие называется целостностью каскадных ссылок.

Опция REFERENCE операторов CREATE TABLE и ALTER TABLE поддерживает опции ON DELETE и ON UPDATE, где можно указать опции CASCADE и NO ACTION:

```
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]
```

NO ACTION указывает, что любые попытки удалить или обновить ключ, на который ссылается внешний ключ в другой таблице, заканчиваются ошибкой, и происходит откат изменений. Это значение по умолчанию, и я рекомендую остановиться на нем. Напоминаю, что без особой надобности не стоит включать каскадные действия.

И все же, бывают случаи, когда каскадные действия действительно упрощают программирование, но использовать их нужно очень аккуратно.

1.5.4. Уникальность

Ограничение UNIQUE (уникальность) указывает, что две строки в колонке не могут содержать одно и то же значение. Это ограничение обеспечивает целостность таблицы с уникальным индексом. Ограничение уникальности эффективно, когда вы уже имеете первичный ключ, но хотите гарантировать уникальность другого поля.

В отличие от первичного ключа у уникального поля может быть одна строка с нулевым значением. Это необходимо учитывать, и если вы хотите, чтобы поле не могло содержать нулевое значение, добавьте ограничение CHECK, например:

```
CREATE TABLE TestTable
(
 iID int DEFAULT 1,
 CONSTRAINT check_iID CHECK (iID is NOT NULL),
 CONSTRAINT u_iID
   UNIQUE NONCLUSTERED(iID)
)
```

В этом примере создается два ограничения на поле iID: одно на уникальность и одно на запрет значений NULL.

В таблице может быть несколько ограничений уникальности, и для каждого такого поля будет создаваться индекс.

1.5.5. Отключение ограничений

Для повышения производительности иногда разумно отключить ограничения. Для примера можно выполнить большую операцию обновления или вставки данных без ограничений.

Когда вы определяете ограничение на таблицу, которая уже содержит данные, MS SQL Server проверяет данные автоматически, гарантируя, что после создания ограничения существующие данные соответствуют требованиям.

Отключать можно только ограничения `CHECK` и `FOREIGN KEY`. Другие ограничения должны быть удалены и потом снова добавлены.

Для отключения проверки, когда вы добавляете ограничения `CHECK` и `FOREIGN KEY` на таблицу с существующими данными, включите опцию `WITH NOCHECK` в оператор `ALTER TABLE`.

В следующем примере мы добавляем ограничение `FOREING KEY`. Ограничение не проверяет существующие данные на момент добавления ограничения:

```
ALTER TABLE TestTable
WITH NOCHECK
 ADD CONSTRAINT FK_TestTable
 FOREIGN KEY (Field1)
 REFERENCES PrimaryTable(Field2)
```

Вы можете отключить проверку ограничений `CHECK` и `FOREIGN KEY` в следующих случаях:

- вы уже убедились, что данные соответствуют требованиям;
- вы хотите загрузить данные, которые не соответствуют ограничению. Позже вы можете выполнить запрос для изменения данных и включить ограничение.

Включение ограничения, отключенного ранее, требует выполнения другого оператора `ALTER TABLE`, которое содержит опцию `CHECK` или `CHECK ALL`.

```
ALTER TABLE имя таблицы
 {CHECK | NOCHECK} CONSTRAINT
 {ALL | ограничение [, …]}
```

1.5.6. Правила и объекты значений по умолчанию

Объект по умолчанию и правило — это объекты, которые могут ограничивать одну или несколько колонок или типы, определенные пользователем. т. е. можно создать их однажды, а затем использовать регулярно.

Объект по умолчанию работает так же, как и ограничение, но только этот объект принадлежит базе данных, а не таблице. Если значение не указано, когда вы вставляете данные, для него будет использовано значение по умолчанию объекта, связанного с полем.

Объект значения по умолчанию создается следующим образом:

```
CREATE DEFAULT имя
 AS выражение константы
```

После создания значения по умолчанию вы должны связать его с колонкой или типом данных, определенным пользователем, с помощью вызова системной процедуры `sp_bindefault`. Для отключения значения по умолчанию выполните системную процедуру `sp_unbindefault`.

Следующий пример задает шаблон номера телефона, если не указано реальное значение:

```
CREATE DEFAULT default_phone_no
 AS '(000)00-00-00'
```

Следующая команда связывает созданный объект с полем `Phone` таблицы `TestTable`:

```
EXEC sp_bindefault default_ phone_no, 'TestTable.Phone'
```

Объекты значений по умолчанию имеют свои ограничения:

- значения по умолчанию не применяются к уже существующим данным, а только к вновь введенным;
- можно использовать только простую логику с одной константой или функцией. Никакой иной логики здесь не может быть;
- тип данных значения должен соответствовать типу поля. Проверка этого соответствия произойдет только во время добавления новой строки, а не во время создания объекта, поэтому здесь необходимо быть очень внимательным;
- у каждого поля может быть только одно значение по умолчанию. Это логично, ведь поле не может принять два значения. Если попытаться задать второе значение, то оно заменит собой уже существующее;
- не может быть обращения к другим таблицам на выборку данных;
- значение по умолчанию добавляется до проверок, поэтому если значение нарушает одно из ограничений, добавление записей станет невозможным;
- для удаления значения по умолчанию его сначала нужно отсоединить от объектов базы данных, с которыми оно связано.

Теперь переходим к правилам. Правила указывают допустимые значения, которые вы можете поместить в колонку. Они гарантируют, что данные подпадают под определенный ряд значений, соответствуют определенному шаблону или списку.

Объявление правила может содержать любые выражения, действительные для оператора `WHERE`, который мы будем внимательно рассматривать в *главе 2*. В общем виде правило выглядит следующим образом:

```
CREATE RULE имя
 AS выражение условия
```

После создания правила вы должны связать его с колонкой или типом данных, определенным пользователем, с помощью вызова системной процедуры `sp_bindrule`. Для отключения правила выполните системную процедуру `sp_unbinrule`. В следующем примере правило гарантирует, что поле `Pol` может содержать только букву "М" или "Ж":

```
CREATE RULE rule_pol
 AS @Pol IN ('M', 'Ж')
```

Следующий пример связывает созданное правило с полем `Pol` таблицы `TestTable`:

```
EXEC sp_bindrule rule_pol, 'TestTable.Pol'
```

Для удаления значения по умолчанию из базы данных используйте оператор `DROP`:

```
DROP DEFAULT  имя [,…]
DROP RULE имя [,…]
```

Например, созданное ранее правило можно удалить командой:

```
DROP RULE rule_pol
```

Правила являются весьма мощным решением, но при этом на них накладывается достаточно большое количество запретов:

- новое правило не может повлиять на уже существующие значения, потому что оно срабатывает только при добавлении или изменении строк данных;
- в правилах можно использовать только несложные вычисления с константами и функциями MS SQL Server;
- в отличие от ограничений нельзя сравнивать значения полей;
- в правиле нельзя обращаться к таблицам для выборки данных. Чтобы была возможность для выборки, стоит использовать триггеры, которые мы будем рассматривать в *главе 3*;
- с одной колонкой можно связать только одно правило. Если бы можно было связывать два правила, то у нас появилась хоть какая-то возможность создавать более сложные запреты, но пока этого нет. Если попытаться связать правило с колонкой, у которой уже есть правило, то старое значение будет заменено новым;

- тип данных, используемый в правиле, должен совпадать с типом поля (что вполне логично);
- связанное правило нельзя удалить. Сначала необходимо удалить связь.

Из-за такого большого количества запретов для правил их не очень-то хочется использовать. Ограничения намного эффективнее и предоставляют нам больше возможностей. Кроме того, корпорация Microsoft сама не рекомендует использовать правила и оставила эту возможность в SQL Server только для совместимости с предыдущими версиями баз данных.

Мы рассмотрели правила потому, что они существуют, и вы должны о них знать, а может быть, вы найдете им применение, даже несмотря на то, что это не рекомендуется. Но прежде чем вы это сделаете, спешу вас предостеречь. То, что не рекомендуется к использованию производителем и оставлено только для совместимости, вполне вероятно, больше не появится в будущих версиях. Это значит, что в следующей версии MS SQL Server правила могут быть выведены из Transact-SQL, и ваша база данных перестанет работать или будет работать без учета ограничений, что потребует изменения ее структуры.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter1\integrity.sql на компакт-диске, прилагаемом к книге.

1.6. Именование

Отдельного разговора требует именование. Дело в том, что SQL Server достаточно лояльно относится к именам таблиц. Например, вы можете использовать в качестве имен русские слова и даже имена, состоящие из нескольких слов. Если имя состоит из нескольких слов, оно должно заключаться в квадратные скобки. Рассмотрим следующий пример:

```
CREATE TABLE Товары
(
  [Дата] datetime,
  [Название товара] varchar(50),
  [Цена] money,
  [Количество] numeric(10, 2)
)
```

В данном случае все поля названы по-русски. Я заключил эти имена в квадратные скобки, чтобы их явно было видно. В реальных условиях я рекомендую все же использовать латинские буквы.

С помощью квадратных скобок можно создавать поля с именами из зарезервированных слов. Например, если попытаться создать поле с именем `min`, то сервер выдаст сообщение об ошибке, потому что это слово зарезервировано (есть такой оператор). Но если такое имя заключить в квадратные скобки, то ошибки не будет.

Квадратными скобками нужно выделять и имена полей, состоящие из нескольких слов. Если в нашем примере убрать квадратные скобки с имени поля `Название товара`, то сценарий закончит выполнение ошибкой.

1.7. Резюме

Давайте соберем воедино всю рассмотренную в данной главе информацию и создадим базу данных из четырех таблиц:

- `tbPeoples` — информация о сотрудниках;
- `tbPositions` — справочник должностей. Несколько сотрудников могут иметь одинаковые должности. Например, даже небольшая организация может иметь в штате несколько водителей. Чтобы не писать в таблице у каждого сотрудника одну и ту же должность, достаточно только выбрать ее из справочника. К тому же, справочник будет иметь возможность отображения подчинения должностей, т. е. будет внешний ключ на саму таблицу должностей;
- `tbPhoneNumbers` — номера телефонов. Так как у одного сотрудника может быть несколько номеров (рабочий, сотовый, домашний), перенесем номера телефонов в отдельную таблицу;
- `tbPhoneType` — типы телефонов (стационарный рабочий, стационарный домашний, сотовый рабочий, сотовый собственный). Эта таблица также позволит не вводить для каждого телефона его тип буквами, достаточно будет только выбрать нужный вариант из справочника.

В нашей базе данных есть два полноценных справочника для хранения должностей и для хранения типов телефонов. Справочники позволяют:

- упростить работу операторам. Не надо каждый раз вводить названия, достаточно выбрать нужный номер из справочника;
- уменьшить вероятность ошибок при вводе символьной информации. Причина может скрываться в раскладках клавиатуры. Например, один пользователь введет слово "сотовый" и другой введет "сотовый", но SQL-сервер будет воспринимать их по-разному. Вы не видите ошибки? На взгляд ее заметить невозможно. Просто одно из слов может содержать первую букву в латинской раскладке. Дело в том, что русская "с" и латинская "c"

выглядят одинаково и находятся на одной клавише. Пользователи очень часто начинают вводить данные в латинском регистре, потом замечают ошибку, стирают лишнее, переключаются на русскую раскладку и вводят заново. При этом первая буква "с" остается в латинской раскладке;

- уменьшить размер базы данных. В базе данных название телефона присутствует только один раз и занимает, например, 30 символов. Во всех строках номеров телефонов стоит только ссылка, которая занимает 4 байта, что намного меньше. Если отказаться от справочника, то в каждой строке номера телефона уже будет занято не 4 символа, а 30, для хранения полного названия типа телефона. Конечно, можно использовать `varchar`, который хранит реальное значение и не резервирует полный доступный размер, но лишних затрат все равно будет довольно много.

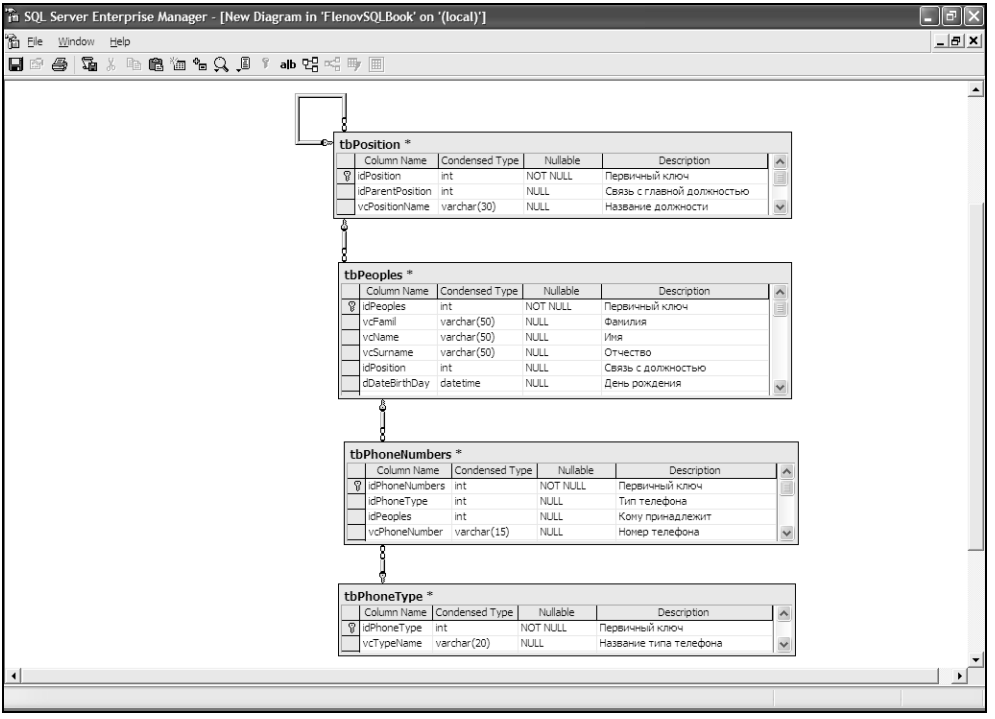


Рис. 1.11. Схема тестовой базы данных

На рис. 1.11 показано окно программы Enterprise Manager, которая входит в состав MS SQL Server и позволяет визуально работать с базой данных. Для наглядности я создал диаграмму и добавил в поля описания их предназначения.

Связь между таблицей персонала (`tbPeoples`) и таблицей должностей (`tbPosition`) происходит по схеме "один-ко-многим". Одной строке из таблицы должностей соответствует множество строк из `tbPeoples`. Связь между персоналом и телефонами (`tbPhoneNumbers`) происходит по схеме "один-ко-многим". Одной записи из таблицы персонала может соответствовать несколько номеров телефонов. Между номерами телефонов и их типами снова связь "один-ко-многим" — одному типу телефона соответствует много номеров.

В листинге 1.11 показан код создания этой базы данных. Внимательно изучите его, потому что он резюмирует все, что мы рассматривали в этой главе. Текст, который находится после символов -- (два тире), является комментарием и не выполняется программой, а используется только для того, чтобы вам проще было разобраться с выполняемыми действиями.

Листинг 1.11. Создание тестовой базы данных

```
-- Удаляем базу данных
DROP DATABASE FlenovSQLBook
GO

-- Создаем базу данных
CREATE DATABASE FlenovSQLBook
GO

-- Переходим в базу данных телефонов
USE FlenovSQLBook

-- Создаем таблицу должностей
CREATE TABLE tbPosition
(
 -- Описание полей
 idPosition int IDENTITY(1,1),
 idParentPosition int,
 vcPositionName varchar(30),

 -- Описание ограничений
 CONSTRAINT PK_idPosition PRIMARY KEY (idPosition),
 CONSTRAINT FK_idParentPosition FOREIGN KEY (idParentPosition)
   REFERENCES tbPosition (idPosition)
)
```

```
-- Создаем таблицу работников
CREATE TABLE tbPeoples
(
  -- Описание полей
  idPeoples int IDENTITY(1, 1),
  vcFamil varchar(50),
  vcName varchar(50),
  vcSurname varchar(50),
  idPosition int,
  dDateBirthDay datetime,

  -- Описание ключей
  CONSTRAINT PK_idPeoples PRIMARY KEY (idPeoples),
  CONSTRAINT FK_idPosition FOREIGN KEY (idPosition)
     REFERENCES tbPosition (idPosition),

  -- Описание ограничений
  CONSTRAINT check_dDateBirthDay CHECK (dDateBirthDay<getdate())
)

-- Создаем таблицу типов телефонов
CREATE TABLE tbPhoneType
(
 idPhoneType int IDENTITY(1, 1),
 vcTypeName varchar(20),

  -- Описание ограничений
 CONSTRAINT PK_idPhoneType PRIMARY KEY (idPhoneType),
)

-- Создаем таблицу телефонов
CREATE TABLE tbPhoneNumbers
(
  -- Описание полей
  idPhoneNumbers int IDENTITY(1, 1),
  idPhoneType int,
  idPeoples int,
  vcPhoneNumber varchar(15),

  -- Описание ключей
```

```
  CONSTRAINT PK_idPhoneNumbers PRIMARY KEY (idPhoneNumbers),
  CONSTRAINT FK_idPhoneType FOREIGN KEY (idPhoneType)
     REFERENCES tbPhoneType (idPhoneType),
  CONSTRAINT FK_idPeoples FOREIGN KEY (idPeoples)
     REFERENCES tbPeoples (idPeoples),

  -- Описание ограничений
  CONSTRAINT check_vcPhonenumber CHECK (vcPhonenumber LIKE
    '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]')
)
```

Помимо этого, создадим еще и таблицу с русскими именами полей, которая не связана с предыдущими таблицами:

```
CREATE TABLE Товары
(
  [Дата] datetime,
  [Название товара] varchar(50),
  [Цена] money,
  [Количество] numeric(10, 2)
)
```

Эта таблица проста, но пригодится в будущем при тестировании некоторых операторов, особенно во *главе 2*.

Обязательно создайте эту базу данных на своем SQL-сервере, потому что мы будем использовать эту структуру в последующих главах для тестирования примеров. Код сценария можно найти в файле \Chapter1\full_table.sql на компакт-диске, прилагаемом к книге.

В каталоге \Chapter1\Database вы можете найти два файла:

- FlenovSQLBook.bak — резервная копия тестовой базы данных для сервера MS SQL Server;
- FlenovSQLBook.mdb — файл базы данных Access, который содержит такую же структуру. Его можно использовать для тестирования простых SQL-запросов. Помните, что эта база данных не поддерживает Transact-SQL и ограничена в возможностях работы с SQL. И все же, она будет удобна, если на вашем компьютере не установлен MS SQL Server, или для того, чтобы убедиться, что запросы одинаково работают и в других базах данных.

Глава 2

Работа с данными

Мы уже умеем создавать хранилище для данных, а именно, базу данных и таблицы. Теперь нам предстоит увидеть, как можно получить данные из базы, записать новые данные в базу и изменять их.

Без данных база не имеет смысла, а данные существуют для того, чтобы их когда-либо просмотреть. Наша задача — научиться их эффективно читать. В этой главе я постараюсь показать всю мощь языка SQL. Большинство руководств, доступных в Интернете (в том числе и написанных когда-то мной и выложенных для бесплатного скачивания), описывают основы языка SQL, но их качество и полнота освещения темы далеки от идеала.

Здесь мы обсудим работу с данными достаточно подробно. А большое количество примеров, максимально приближенных к реальным задачам, помогут вам в будущем удачно находить собственные решения.

База уже создана, но для тестирования примеров необходимо наполнить ее данными. Добавление данных в таблицы мы будем рассматривать чуть позже, поэтому пока что я рекомендую выполнить запрос из файла \Chapter2\fill_data.sql с компакт-диска, прилагаемого к книге.

Но прежде чем мы начнем изучение операторов SQL, используемых для работы с данными, я хочу показать два оператора: `USE` и `GO`. Они поддерживаются не во всех программах и только с MS SQL Server (с Access работать не будут). Эти операторы точно работают в Query Analizer, который я рекомендую использовать для тестирования запросов к серверу MS SQL.

Оператор `USE` позволяет выбрать нужную базу данных. Например, вы хотите выполнить запрос, который будет работать с объектами базы данных master. Чтобы выбрать master в качестве текущей базы, напишем:

```
USE master
```

С помощью GO вы можете разбивать файл сценария на любое количество блоков. Например:

```
CREATE DATABASE dbNew
GO
CREATE DATABASE new
```

Если бы не было GO, то программа-клиент отправила бы оба оператора создания базы данных одним блоком. Но у нас есть GO, поэтому серверу будет отправлено два блока с запросом — текст до оператора и после.

2.1. Оператор *SELECT*

Оператор выборки является основой базы данных. Я специально сегодня сосчитал SQL-запросы в одном из своих проектов, и запросов на получение данных оказалось в 5 раз больше, чем всех остальных. И это при том, что я стараюсь писать достаточно универсальные программы, и запросы решают несколько задач одновременно.

Итак, для получения данных от сервера используется оператор SELECT. В самом общем и упрощенном виде он показан в листинге 2.1.

Листинг 2.1. Сокращенный формат оператора SELECT

```
SELECT select_list
[ INTO new_table ]
FROM table_source
[ WHERE search_condition ]
[ GROUP BY group_by_expression ]
[ HAVING search_condition ]
[ ORDER BY order_expression [ ASC | DESC ] ]
```

Эта упрощенная форма позволяет разделить выборку данных на части. Посмотрим, что дает такое разделение:

- SELECT — основной и обязательный оператор, затем перечисляются поля, которые необходимо получить от сервера;
- INTO — необязательная секция, описывающая таблицу, в которую будет записан результат выполнения. Чаще всего результат передается в памяти, но может быть сохранен в новой таблице;
- FROM — в этой секции идет описание таблиц, из которых необходимо прочитать данные;

- `WHERE` — в этой секции можно указать параметры поиска, ограничения выводимых данных;
- `GROUP BY` — в этой секции можно указать список полей, по которым необходимо сгруппировать данные;
- `HAVING` — в этой секции можно указать дополнительные ограничения поиска, чаще всего они необходимы для ограничения параметров группировки;
- `ORDER BY` — в этой секции можно указать параметры сортировки, а, точнее, поля, по которым будет отсортирован результат.

Более полный вариант оператора `SELECT` показан в листинге 2.2. Сейчас в нем не все понятно, но в будущем, когда вы научитесь пользоваться оператором `SELECT` и узнаете его поближе, этот листинг поможет вам вспомнить, где, что и как надо указывать, без обращения к главам, дающим соответствующее описание.

Листинг 2.2. Полный вариант оператора `SELECT`

```
SELECT statement ::=
    < query_expression >
    [ ORDER BY { order_by_expression | column_position [ ASC | DESC ] }
        [ ,...n ]    ]
    [ COMPUTE
        { { AVG | COUNT | MAX | MIN | SUM } ( expression ) } [ ,...n ]
        [ BY expression [ ,...n ] ]
    ]
    [ FOR { BROWSE | XML { RAW | AUTO | EXPLICIT }
            [ , XMLDATA ]
            [ , ELEMENTS ]
            [ , BINARY base64 ]
        }
]
    [ OPTION ( < query_hint > [ ,...n ]) ]

< query expression > ::=
    { < query specification > | ( < query expression > ) }
    [ UNION [ ALL ] < query specification | ( < query expression > ) [...n ] ]

< query specification > ::=
```

```
SELECT [ ALL | DISTINCT ]
    [ { TOP integer | TOP integer PERCENT } [ WITH TIES ] ]
    < select_list >
[ INTO new_table ]
[ FROM { < table_source > } [ ,...n ] ]
[ WHERE < search_condition > ]
[ GROUP BY [ ALL ] group_by_expression [ ,...n ]
    [ WITH { CUBE | ROLLUP } ]
]
[ HAVING < search_condition > ]
```

Как видите, у оператора `SELECT` достаточно много параметров, и для их изучения понадобится довольно много времени. Даже в этом листинге невозможно было указать все опции оператора, хотя данный общий вид является наиболее полным.

2.2. Выборка данных

Теперь мы переходим к изучению команд языка SQL. Для этого мы создадим простой запрос и рассмотрим его возможности:

```
SELECT vcFamil, vcName, vcSurName
FROM tbPeoples
```

Все запросы начинаются с оператора `SELECT`, что означает "выбрать". После команды перечисляются имена полей, которые необходимо выбрать из базы данных. Оператор `FROM` указывает таблицу (или несколько таблиц), из которой нужно получить указанные поля.

Теперь посмотрим, что и откуда выбирает вышеприведенный запрос. Выборка происходит из таблицы `tbPeoples`, выбираются три поля: `vcFamil`, `vcName`, `vcSurName`. Результат выполнения этой команды в Query Analyzer можно увидеть на рис. 2.1.

Ничего сложного тут нет. Такой маленький запрос возвращает нам поля с фамилией, именем и отчеством.

Операторы `SELECT` и `FROM` являются обязательными и присутствуют всегда, хотя есть исключение, когда `FROM` не будет обязательным, но, исходя из личной практики, могу сказать, что такие запросы используются очень редко. Напишем следующий запрос:

```
SELECT 1
```

В результате на экране будет показана таблица, состоящая из одной строки и одной колонки, где будет находиться только число 1.

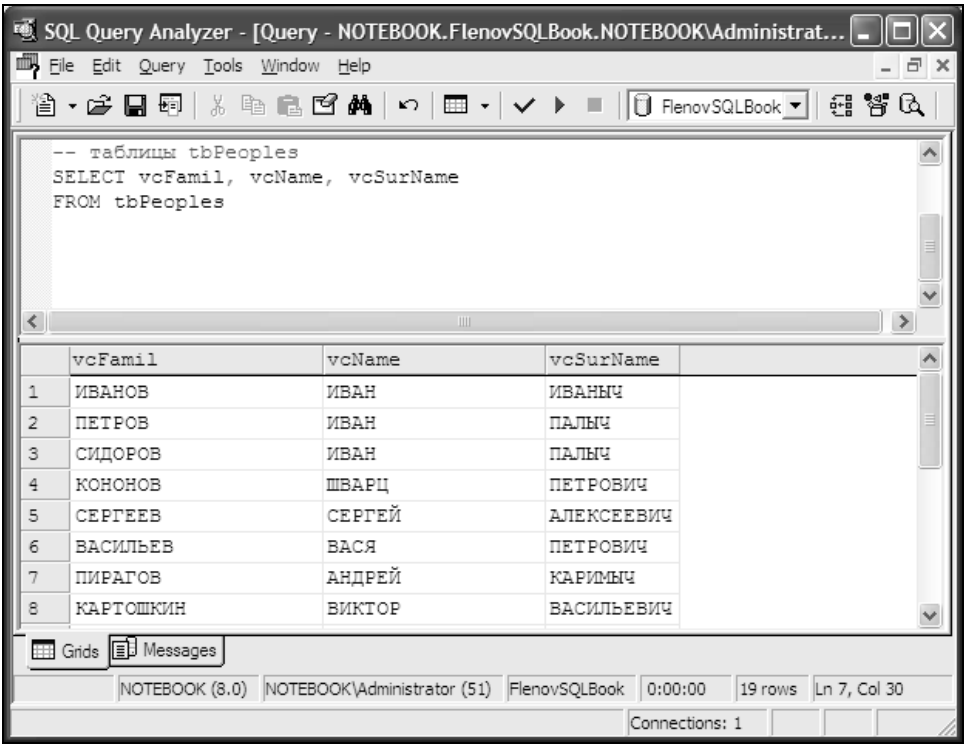


Рис. 2.1. Результат выборки полей фамилии, имени и отчества

Если нужно вывести все поля, то можно перечислить все их имена или воспользоваться значком "*", что соответствует всем полям таблиц, перечисленных в секции `FROM`. Следующий запрос выведет все поля из таблицы `tbPeoples`:

```
SELECT *
FROM tbPeoples
```

Если вы используете "*", то в результат будут включены все поля в том порядке, в каком они находятся в таблице. Если нужно упорядочить их как-то иначе, то перечислите все поля в требуемом порядке.

2.2.1. Полный путь

Полный формат обращения к таблице выглядит следующим образом:

```
База_Данных.Владелец.Таблица
```

Но если вы подключены к нужной базе данных, то можно сразу же указывать имя таблицы. Попробуйте подключиться к базе данных master и выполнить

этот же запрос. Результатом будет ошибка, потому что таблица `tbPeoples` в базе данных master будет не найдена. Чтобы, находясь в одной таблице, обратиться к другой, необходимо написать ее имя в полном формате:

```
SELECT *
FROM FlenovSQLBook.dbo.tbPeoples
```

Здесь:

- `FlenovSQLBook` — имя базы данных;
- `dbo` — владелец базы данных;
- `tbPeoples` — имя таблицы.

Если объект создан с правами `dbo`, то владельца указывать необязательно, поэтому я рекомендую создавать все объекты именно с этим владельцем. Итак, если вы находитесь в базе данных master и хотите обратиться к объекту из другой базы, и этот объект принадлежит `dbo`, то можно написать следующий запрос:

```
SELECT *
FROM FlenovSQLBook..tbPeoples
```

В данном примере имя владельца опускается, а сервер по умолчанию использует `dbo`, поэтому запрос будет выполнен корректно.

Чтобы точно быть уверенным, что объект создан от имени `dbo`, а не от имени конкретного пользователя базы данных, можно указывать полное имя и при создании объекта. Следующий пример создает таблицу `Peoples`, явно указывая владельца:

```
CREATE TABLE dbo.Peoples
(
 idPeoples int
)
```

Если вы находитесь в базе данных master, а таблицу нужно создать в базе данных TestDB, то можно указать полный путь, включая имя базы данных:

```
CREATE TABLE TestDB.dbo.Peoples
(
 idPeoples int
)
```

Если вы находитесь в той же базе данных, что и объект, но объект принадлежит не `dbo`, то базу данных указывать не надо, а вот владельца указать необходимо:

```
SELECT *
FROM Flenov.tbPeoples
```

В данном случае Flenov — это имя владельца.

Если вы находитесь в той же базе данных, что и объект, и при этом объект принадлежит dbo, то сразу указываем имя таблицы:

```
SELECT *
FROM tbPeoples
```

2.2.2. Ограничение вывода строк

В языке SQL есть два ключевых слова, которые характеризуют параметры вывода информации: ALL и DISTINCT. Параметр ALL означает, что выводить необходимо все строки, а DISTINCT означает, что не нужно выводить повторяющиеся строки. Следующий запрос выведет все возможные имена людей, содержащиеся в таблице:

```
SELECT vcName
FROM tbPeoples
```

Но у нас в таблице очень много записей, у которых в поле vcName находится одно и то же имя. Например, если вы для заполнения базы данными использовали запрос с компакт-диска, то можете увидеть, что имя Андрей повторяется пять раз. Зачем нам это в результате? Чтобы отбросить лишнее, указываем после оператора SELECT параметр DISTINCT:

```
SELECT DISTINCT vcName
FROM tbPeoples
```

Если в таблице окажутся две строки с одинаковым именем, то запрос выведет это имя только один раз. Если не задать параметр DISTINCT, то считается, что надо использовать параметр ALL.

Оператор DISTINCT действует только на полностью одинаковые строки. Например, предыдущий запрос возвратил 12 уникальных имен из таблицы. Но если попытаться запросить уникальное сочетание имени и фамилии, то результат будет другим:

```
SELECT DISTINCT vcFamil, vcName
FROM tbPeoples
```

Здесь мы просим вывести два поля vcFamil и vcName, при этом результат не должен содержать одинаковых строк. В моей таблице совпадений по двум полям нет ни в одной строке, поэтому в результат попали все 19 строк таблицы.

Мне не раз приходилось работать с таблицами, которые содержали до миллиона строк. Давайте немного посчитаем. Если одна строка будет занимать 100 байт, то общий размер таблицы будет равен размеру строки, умноженному на количество записей. В результате получится около 100 Мбайт. Если

для такой таблицы выполнить запрос `SELECT *`, то серверу придется по сети передать достаточно большой объем данных, а на компьютере-клиенте понадобится около 100 Мбайт памяти для хранения результата запроса. Это достаточно много, и не всякий клиентский компьютер согласится принять такое количество данных. Если на компьютере не хватит оперативной памяти, то он станет работать медленнее.

Но в языке Transact-SQL есть возможность ограничения вывода. Далеко не всегда пользователям нужно видеть все данные, поэтому вполне логично будет ограничиться только верхней частью результата, например, получать только первые сто строк. А если уж пользователь захотел явно увидеть все данные, то только тогда предоставлять ему такую возможность.

Ограничение количества выводимых данных происходит с помощью оператора `TOP`, который ставится сразу после `SELECT`, например, в следующем запросе мы запрашиваем только 10 первых строк результата:

```
SELECT TOP 10 *
FROM tbPeoples
```

После оператора `TOP` ставится число, указывающее количество необходимых записей. Если результат запроса состоит из большего количества строк, то лишнее просто отбрасывается. После этого числа можно указывать имена необходимых полей. В данном случае стоит знак "*", т. е. нам необходимы все поля.

А если нужно получить определенный процент от результата? Например, нужно увидеть 50% всех строк таблицы? В этом случае укажите число в процентах, а затем `PERCENT`:

```
SELECT TOP 50 PERCENT *
FROM tbPeoples
```

2.2.3. Псевдонимы полей

В нашей базе данных имена полей написаны латиницей. Дело в том, что далеко не все базы данных позволяют создавать поля, названные по-русски. Хотя Microsoft заботится о нас и позволяет создавать имена полей на родном языке, я привык давать имена, используя только латинские буквы, и вам советую то же самое. Почему? Проблемы кодировки решены еще не полностью, и вы можете столкнуться с ними. Особенно, если будете экспортировать данные в другие таблицы, такие как DBF.

А вот во время вывода данных с помощью запросов можно давать вполне понятные имена. После имени поля ставится ключевое слово `AS`, за ним идет

текст в одинарных кавычках, который как раз и будет выводиться в заголовке поля:

```
SELECT vcFamil AS 'Фамилия', vcName AS 'Имя',
   vcSurName AS 'Отчество', dDateBirthDay AS 'Дата рождения'
FROM tbPeoples
```

Псевдонимы не изменяют имя поля в базе данных и влияют только на результирующую таблицу. Но они бывают полезны не только для изменения заголовка имени поля в результирующей таблице, но и для его задания. Мы не раз будем встречаться со случаями, когда поле не имеет заголовка. Ярким примером является уже рассмотренный ранее запрос:

```
SELECT 1
```

В этом случае результатом будет просто число 1, которое не является полем и не может иметь заголовка. В некоторых программах в качестве заголовка для такого поля будет просто пустое поле, а в SQL Query Analyzer в заголовке будет "No column name" ("Колонка без имени"). Таким образом, с помощью псевдонима мы можем дать имя полю, у которого до этого вообще не было имени. Например:

```
SELECT 1 AS Num
```

Теперь в качестве результата будет поле с именем `Num`, а в единственной строке в этом поле можно будет увидеть число 1.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\select.sql на компакт-диске, прилагаемом к книге.

2.3. Ограничение выборки

Прежде чем мы двинемся дальше, должен предупредить, что во время отладки запросов вас могут подстерегать определенные опасности:

- некоторые базы данных чувствительны к регистру данных. Нет, не к операторам, а именно к данным в таблице, а значит, слова АНДРЕЙ и Андрей при сравнении будут восприниматься как разные. В MS SQL Server можно настроить, должны ли сравнительные операции для строк быть чувствительными к регистру букв;
- возможно, что ваша база данных способна воспринимать текст только в верхнем регистре;
- некоторые производители отклоняются от стандартов. Даже Microsoft в разных базах данных поддерживает стандарт SQL по-разному. Например,

для создания связи между таблицами в MS SQL Server есть возможность использовать ANSI SQL метод и оператор `JOIN`, а в MS Access работает только `JOIN`.

Теперь познакомимся с оператором `WHERE`. Он задает критерии поиска. Например, нам надо выбрать все записи из таблицы `tbPeoples`, где в поле `vcName` содержится имя `Андрей`. В этом случае необходимо написать следующий запрос:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
```

Результат выполнения команды можно увидеть на рис. 2.2.

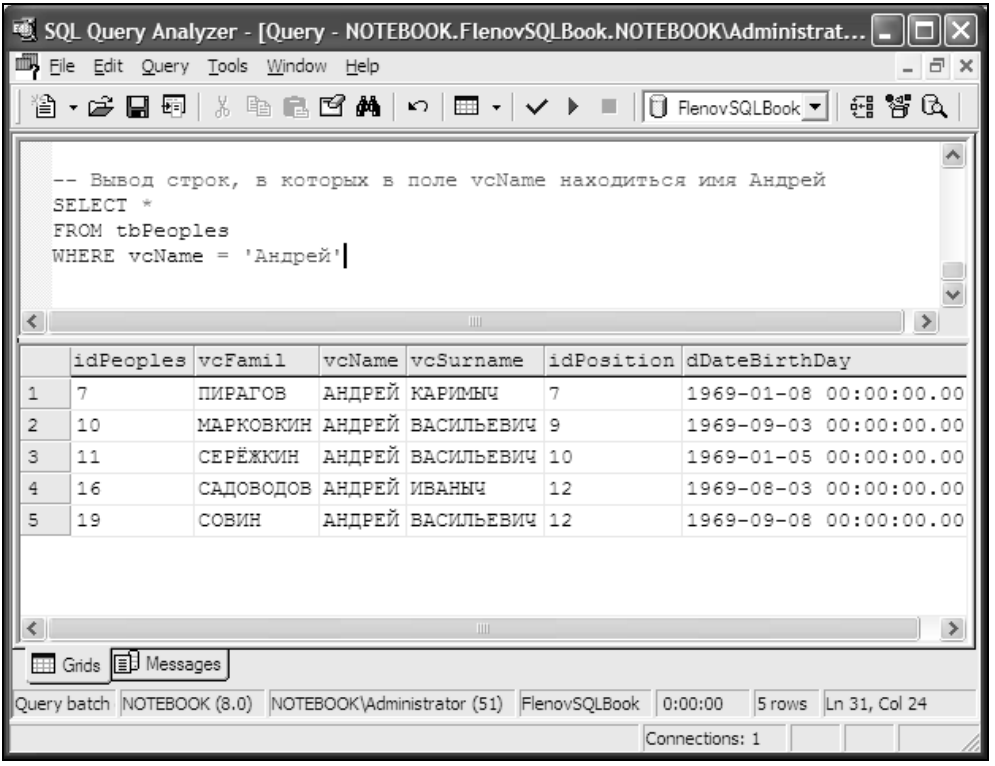


Рис. 2.2. Результат поиска всех Андреев

Если проговорить весь запрос словами, то он будет звучать так: "Выбрать все строки из таблицы `tbPeoples`, где поле `vcName` равно `Андрей`". Еще одно замечание: в запросах строки выделяются кавычками. В зависимости от базы

данных кавычки могут быть одинарными или двойными. Стандартом предусмотрены двойные, но в MS SQL Server используются одинарные кавычки. В стандарте для сравнения строк используется оператор `LIKE`, который ставится вместо знака равенства:

```
SELECT *
FROM tbPeoples
WHERE vcName LIKE 'Андрей'
```

В результате мы увидим одну или несколько строк, которые в поле `vcName` содержат имя `Андрей`.

Но MS SQL Server позволяет сравнивать строки с помощью знака равенства. К тому же, оператор `LIKE` работает медленнее, потому что позволяет использовать шаблоны, о которых мы еще поговорим.

Теперь рассмотрим, как этот запрос будет выглядеть при поиске по числам. Для этого найдем все строки из той же базы, где поле `idPosition` содержит значение 10.

```
SELECT *
FROM tbPeoples
WHERE idPosition = 10
```

В этом случае мы производим поиск по числовому полю, поэтому используем знак равенства, а само число не нужно выделять кавычками. Но даже если вы выделите число одинарными кавычками, ошибкой это считаться не будет, поэтому следующий запрос также является корректным:

```
SELECT *
FROM tbPeoples
WHERE idPosition = '10'
```

Язык SQL не жестко типизирован, и основные преобразования между типами данных производятся автоматически, без нашего вмешательства. В данном случае поле `idPosition` является числовым, а т. к. мы сравниваем его со строкой, то содержимое строки будет автоматически преобразовано в число. Другое дело, если вместо числа передать строку, которая будет содержать недопустимые символы, т. е. не цифры, например:

```
SELECT *
FROM tbPeoples
WHERE idPosition = '10f'
```

В этом случае SQL-сервер выдаст сообщение об ошибке:

Syntax error converting the varchar value '10f' to a column of data type int. (Синтаксическая ошибка преобразования типа varchar со значением '10f' в колонку с типом данных int).

Как видите, сервер пытался автоматически произвести преобразование и не смог, поэтому результатом явилась такая ошибка.

Пока что мы использовали только оператор равенства. Но это не значит, что больше ничего нет. Стандартом разрешены следующие операторы:

- = — равно
- > — больше
- < — меньше
- >= — больше или равно
- <= — меньше или равно
- <> — не равно

Операторы "больше", "меньше" и другие можно использовать не только с числами, но и со строками. В этом случае буква "А" будет меньше, чем "Р". При сравнении строк разного регистра меньшей оказывается строка, набранная в верхнем регистре, например, "А" будет меньше "а", и "Р" будет меньше "а". Но помните, что не всегда база данных чувствительна к регистру, это зависит от настроек.

Давайте выберем из базы данных все записи, в которых имена начинаются с буквы большей, чем буква "С":

```
SELECT *
FROM tbPeoples
WHERE vcName > 'C'
```

Из тестовой таблицы в результат попадут имена Сергей, Славик и Шварц. Почему Сергей и Славик попадают в результат, ведь первая буква равна "С", а мы запросили слова, где строки больше чем "С"? Дело в том, что происходит сравнение всего слова, а имена содержат более одной буквы. Если бы имя было сокращено до одной буквы "С", то такая строка не попала бы в результат, но слова с большим количеством букв попадают в результат. Это все равно как открыть словарь, начиная с буквы "С". Сама буква не включается в результат, но все слова, содержащие больше букв, будут показаны.

Иногда нужно вывести на экран результат запроса, но при этом он обязательно должен быть пустым. Для этого необходимо написать такое условие, которое однозначно не вернет ни одной строки. Можно выдумывать какие-то сложные сравнения в секции `WHERE`, но эффективнее всего написать так:

```
SELECT *
FROM tbPeoples
WHERE 1 = 0
```

Условие `1 = 0` заведомо невыполнимо, а значит, в результате никогда не будет ни одной строки.

В каком случае это может понадобиться? Допустим, вы не знаете, какие поля есть в таблице, и хотите их узнать. Для этого можно выполнить запрос:

```
SELECT *
FROM tbPeoples
```

Но если в таблице `tbPeoples` очень много записей, то выполнение будет долгим, потому что сервер должен вернуть все строки, которые нам не нужны. Намного проще поставить условие `1=0`, и результат появится на экране мгновенно.

Если создавать программу на языке высокого уровня, например, Delphi, то пока запрос не активен, на форме будет отображаться сетка в не очень удобном виде (без заголовков полей). Чтобы отобразить только заголовки, я всегда выполняю запрос с условием `1=0`. В этом случае у сервера и программы не будет больших накладных расходов, а вот для пользователя результат будет наглядным и красивым.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\where.sql на компакт-диске, прилагаемом к книге.

2.4. Булевы операторы

Запросы, в которых есть только одно условие, являются наиболее простыми. Чаще всего необходимо найти данные по нескольким условиям одновременно. Эта проблема решается с помощью булевых операторов. В стандарте предусмотрено три булевых оператора: `AND` (логическое И), `OR` (логическое ИЛИ), `NOT` (логическое НЕ). Например, если необходимо найти данные в таблице по двум условиям, и при этом оба условия должны выполняться одновременно, то необходимо объединить их с помощью оператора `AND`.

Сразу же рассмотрим пример, в котором найдем человека по имени Андрей и с идентификатором должности, равным 12:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 AND idPosition=12
```

Результат запроса — все строки, содержащие в поле vcName имя Андрей и в поле idPosition значение больше 12. Если какое-то из этих условий не выполнится, то строка не будет выбрана.

Теперь рассмотрим пример, в котором выбираются все люди с именем Андрей, и работник, у которого поле idPosition равно 1 (это должность генерального директора). Это значит, что объединить эти два условия нужно с помощью оператора OR:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 OR idPosition=1
```

Следующий пример отображает строку для Пирагова Андрея:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 AND vcFamil='Пирагов'
```

Для того чтобы строка была выбрана, должны быть верны оба условия (потому что мы используем оператор AND).

Теперь познакомимся с оператором NOT (НЕ). Посмотрите на следующий запрос:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 AND NOT idPosition=12
```

В этом запросе выбираются все Андреи, но при этом idPosition не должно быть равно 12. Обратите внимание, что во втором условии NOT idPosition = 12 оператор NOT стоит в начале. Вот именно там он и должен стоять, и даже не пробуйте его ставить в середину или в конец условия! В принципе, это условие идентично условию "не равно", т. е. <>:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 AND idPosition<>12
```

Но это не значит, что оператор NOT не нужен. Иногда он действительно очень удобен, в чем вы убедитесь при изучении операторов IN и BETWEEN или знакомстве с операцией сравнения с нулевым значением.

При работе с булевыми операторами вы можете использовать скобки для объединения условий. Допустим, что нужно вывести все строки с именем Андрей, и при этом `idPosition` должно быть равно 10 или 12. Напишем следующий запрос:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 AND idPosition=12 OR idPosition=10
```

Результат будет не совсем таким, как мы ожидали. Будут выбраны все Андреи с `idPosition = 12`, а также все записи с `idPosition` = 10, вне зависимости от имени:

```
11 СЕРЕЖКИН  АНДРЕЙ ВАСИЛЬЕВИЧ 10 1969-01-05
12 КОРОВА    ВЛАД   ИВАНОВИЧ   10 1969-08-02
16 САДОВОДОВ АНДРЕЙ ИВАНЫЧ     12 1969-08-03
19 СОВИН     АНДРЕЙ ВАСИЛЬЕВИЧ 12 1969-09-08
```

Обратите внимание, что во второй строке имя не Андрей, но она попала в результат. Правильным будет запрос:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 AND (idPosition=12 OR idPosition=10)
```

Результат запроса — все строки, содержащие в поле `vcName` значение Андрей и в поле `idPosition` значение 10 или 12.

Можно еще больше усложнить запрос и использовать вложенные скобки:

```
SELECT *
FROM tbPeoples
WHERE idPeoples>1 AND
   (idPosition>10 OR
     (vcName='ИВАН' AND vcSurname='ПАЛЫЧ'))
```

Получается, что у нас могут быть круглые скобки внутри круглых скобок, что позволяет создать логику выбора любой сложности.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\bool.sql на компакт-диске, прилагаемом к книге.

2.5. Улучшенный поиск

В этом разделе нам предстоит обсудить еще несколько операторов, с помощью которых можно улучшить поиск необходимой информации. Это операторы `IN`, `BETWEEN`, `IS NULL`. А также мы познакомимся с шаблонами.

Начнем с оператора `IN`. В принципе, можно создавать запросы и без него, но он упрощает SQL-код и делает его более наглядным. Сразу же рассмотрим пример:

```
SELECT *
FROM tbPeoples
WHERE vcName = 'Андрей'
 OR vcName = 'ИВАН'
 OR vcName = 'ВАСЯ'
 OR vcName = 'ШВАРЦ'
```

В этом запросе мы выбираем все записи, в которых имя равно Андрей, или Иван, или Вася, или Шварц. Этот запрос вполне работающий, но представьте, что так надо перечислить 20 или более имен. В этом случае запрос раздуется как воздушный шар.

Проблему решает оператор `IN`, который позволяет задать список значений, с которыми нужно производить сравнение. Список задается в скобках через запятую. Вот как этот же запрос будет выглядеть с оператором `IN`:

```
SELECT *
FROM tbPeoples
WHERE vcName in ('Андрей', 'ИВАН', 'ВАСЯ', 'ШВАРЦ')
```

Такой запрос писать намного проще, и он занимает меньше места. Если внимательно рассмотреть два предыдущих запроса, то можно и без моих объяснений понять, что делает оператор `IN`. В данном случае он выведет все записи, в которых поле имени содержит одно из значений, перечисленных через запятую в скобках. То есть мы задаем допустимые значения, с которыми необходимо производить сравнение.

Если используется числовое поле, то кавычки надо убрать. Например, следующий запрос выбирает все записи, в которых поле `idPosition` равно одному из значений 1, 2, 3, 4, 5:

```
SELECT *
FROM tbPeoples
WHERE idPosition in (1, 2, 3, 4, 5)
```

Вместе с оператором `IN` и со всеми последующими операторами, которые мы будем рассматривать в этой главе (например, `BETWEEN`), вы можете смело

использовать булевы операторы. Например, запросим все строки, кроме перечисленных в операторе `IN`:

```
SELECT *
FROM tbPeoples
WHERE idPosition NOT IN (1, 2, 3, 4, 5)
```

Этот запрос выбирает все записи, где поле `idPosition` не равно 1, 2, 3, 4, 5. Вот тут уже без булева оператора `NOT` не обойтись. С его помощью мы изменяем результат сравнения на противоположный.

Числа в скобках могут стоять в любом порядке, но я не зря выбрал числа, идущие последовательно. Просто хочется плавно перейти к оператору `BETWEEN`. Он позволяет выбрать все записи, в которых указанное поле содержит значения, попадающие в определенный диапазон. Например, вам нужно выбрать все строки, в которых `1 <= idPosition <= 5`. Это можно сделать следующим запросом, используя операторы сравнения:

```
SELECT *
FROM tbPeoples
WHERE idPosition >= 1 AND idPosition <= 5
```

Но оператор `BETWEEN` делает запрос намного красивее и элегантнее:

```
SELECT *
FROM tbPeoples
WHERE idPosition BETWEEN 1 AND 5
```

Как видите, оператор `BETWEEN` также нужен только для удобства, и вы можете спокойно обойтись без него. Этот оператор просто задает диапазон чисел. Вы должны помнить, что конечные точки включаются в результат запроса.

Теперь посмотрим, как выбрать данные со значениями вне указанного диапазона без применения оператора `BETWEEN`. Для этого можно использовать один из следующих запросов:

```
SELECT *
FROM tbPeoples
WHERE NOT (idPosition >= 1 AND idPosition <= 5)
```

Или

```
SELECT *
FROM tbPeoples
WHERE idPosition < 1 OR idPosition > 5
```

Какой из вариантов выбрать, зависит от личных предпочтений. Но та же задача с помощью BETWEEN решается более элегантно. Для этого необходимо написать NOT BETWEEN, как показано в следующем примере:

```
SELECT *
FROM tbPeoples
WHERE idPosition NOT BETWEEN 1 AND 5
```

Этот оператор можно использовать и со строковыми полями. Например, нужно выбрать все строки, в которых имя начинается с буквы "A" до "B":

```
SELECT *
FROM tbPeoples
WHERE vcName BETWEEN 'A' AND 'B'
```

В результат этого запроса войдут такие имена, как Андрей и Борис. Василий в результат не попадет, потому что это имя больше, чем просто буква "B".

Переходим к оператору IS, а точнее, IS NULL и IS NOT NULL, потому что IS отдельно не используется. С помощью этого оператора можно проверить, является ли значение нулевым. Вспомним, что нулевое значение не равно 0 или "" (пустой строке). Нулевое значение NULL — это незаполненное поле, где вообще нет никакого значения.

Этот оператор действительно может сыграть важную роль. Допустим, вам нужно выбрать все записи, в которых поле для хранения отчества (vcSurName) не заполнено:

```
SELECT *
FROM tbPeoples
WHERE vcSurName IS NULL
```

Когда мы заполняли базу данных значениями, то не рассчитывали проверять ее этим оператором, поэтому все поля заполнены, и в результат не попадет ни одной строки. С другой стороны, поскольку в результате мы не получили ни одной строки, значит, данные вполне корректны, и поле для отчества у всех записей заполнено.

Чтобы найти все ненулевые значения, используется оператор IS NOT NULL, Например, следующий запрос отображает все записи, в которых заполнено поле для хранения отчества vcSurName:

```
SELECT *
FROM tbPeoples
WHERE vcSurName IS NOT NULL
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\better.sql на компакт-диске, прилагаемом к книге.

2.6. Вставка в таблицу

Мы опустили на время секцию INTO, которая идет после SELECT и позволяет создавать новые таблицы. Это упущение было сделано намеренно, чтобы сначала научиться писать запросы на выборку и немного познакомиться с секцией WHERE. Теперь примеры с секцией INTO будут более интересными, и вы увидите ее мощь и удобство.

Допустим, что надо сохранить все строки с именем Андрей в отдельной таблице для дальнейшего использования. Выполним следующий запрос:

```
SELECT *
INTO tbAndrey
FROM tbPeoples
WHERE vcName='Андрей'
```

Здесь создается таблица tbAndrey. Такой таблицы не должно быть в вашей базе данных, иначе создание окажется невозможным. Если таблица tbAndrey уже существует, то запрос вернет ошибку, потому что SELECT..INTO не умеет перезаписывать таблицы. Вы должны сами удалять ненужное, гарантируя таким образом, что нужная таблица не будет уничтожена (перезаписана).

Преимущество запроса типа SELECT..INTO в том, что не надо создавать вручную таблицу с нужными полями, все делается автоматически.

Теперь мы можем без проблем работать с новой таблицей tbAndrey, потому что она ничем не отличается от всех тех, что мы создали с помощью оператора CREATE TABLE:

```
SELECT *
FROM tbAndrey
```

Оператор SELECT..INTO чаще всего используют для создания временных таблиц, а не постоянных, например, при создании отчетности. Напоминаю, что для создания временной таблицы необходимо указать символ # в начале имени:

```
SELECT *
INTO #tbAndrey
FROM tbPeoples
WHERE vcName='Андрей'
```

Для хранения постоянных данных лучше использовать оператор CREATE TABLE и настраивать поля и ограничения на таблицу должным образом. При использовании SELECT..INTO ограничения не переносятся, а копируются только данные и структура таблицы.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\into.sql на компакт-диске, прилагаемом к книге.

2.7. Шаблоны строк

Нередко пользователям нужно найти какие-то строки, в написании которых они не уверены. Например, пользователи могут ввести в базу данных отчество Иванович или, по ошибке, Иваныч. Чтобы отобразить и то, и другое (а, может быть, и третий вариант), нужно использовать шаблоны. В данном случае нужно найти все строки, в которых в поле отчества значения начинаются с "Иван". В SQL это вполне реально. Если вы не знаете, как должна закончиться строка, то можно поставить знак процента (%). Например:

```
SELECT *
FROM tbPeoples
WHERE vcSurName LIKE 'ИВАН%'
```

В результат попадут строки с отчеством Иваныч и Иванович.

Рассмотрим еще пример. Допустим, что нужно вывести все строки, в которых имя начинается с буквы "С". Это реализуется в следующем запросе:

```
SELECT *
FROM tbPeoples
WHERE vcName LIKE 'С%'
```

В результате будут строки, в которых имя работника равно Сергей и Славик. Если бы в базе были бы другие имена на букву "С", то эти имена также попали бы в результирующий набор.

Знак процента может быть в любом месте. Например, нужно вывести все фамилии, которые начинаются с буквы "С" и заканчиваются буквой "В":

```
SELECT *
FROM tbPeoples
WHERE vcFamil LIKE 'С%В'
```

Знак процента в данном случае заменяется любым количеством любых символов, но в начале и в конце обязательно должны стоять буквы "С" и "В". В результате мы увидим фамилии Сидоров, Сергеев, Смирнов, Садоводов.

Следующий пример разрешает наличие любых символов и перед буквой "С":

```
SELECT *
FROM tbPeoples
WHERE vcFamil LIKE '%С%В'
```

В результате будут фамилии: Сидоров, Сергеев, Васильев, Смирнов, Садоводов. Наиболее наглядной является фамилия Васильев. Она содержит букву "С" в середине и букву "В" в конце. Между этими буквами и до "С" может быть любое количество любых символов.

А если неизвестен только один символ? Например, в моей практике не раз встречались фамилии, начинающиеся с буквы "С". Эти буквы в русской и латинской раскладках находятся на одной клавише. Пользователи очень часто начинают ввод в латинской раскладке, а когда видят ошибку, стирают все, кроме злополучной буквы "С". Я этот пример привожу очень часто, потому что он очень популярен. Если клиентская программа не проверяет возможное наличие неправильных символов, то может возникнуть серьезная проблема. В полях, которые должны содержать только русские буквы, должна быть проверка на стороне клиента или на стороне сервера (с помощью ограничений) возможности использования только русских букв.

Итак, как найти фамилию Смирнов, если мы не знаем, в какой кодировке написана первая буква "С"? Нужно заменить ее на символ подчеркивания ("_"):

```
SELECT *
FROM tbPeoples
WHERE vcFamil LIKE '_МИРНОВ'
```

Символ подчеркивания означает один любой символ. В то время как "%" означал любое количество произвольных символов, "_" обозначает только один символ. Рассмотрим пример: "Т_К". В результат такого шаблона войдут слова: ТОК, ТУК, ТИК, но не войдет ТУПИК и другие слова, содержащие между буквами "Т" и "К" больше, чем одну букву. Этот символ очень удобен, особенно для тех, кто подзабыл русский язык.

Представьте, что надо найти слово "корова", а вы не знаете, как писать: "корова" или "карова". В этом случае просто пишите "к_рова", и вы не ошибетесь. Или вообще "к_р%". Красиво? А в принципе, это то же самое. Вот если бы в школе диктанты можно было бы так писать!

В одном шаблоне можно использовать и символ процента, и подчеркивание. Например, следующий пример показывает, как найти всех Смирновых вне зависимости от пола и первой буквы:

```
SELECT *
FROM tbPeoples
WHERE vcFamil LIKE '_МИРНОВ%'
```

В результат попадет как фамилия Смирнов, так и Смирнова.

Обратите внимание, что для работы с шаблоном обязательно нужно использовать вместо знака равенства оператор `LIKE`. Если поставить знак равенства, то шаблон действовать не будет.

Шаблоны действительно мощное средство поиска необходимых данных, а самое главное — очень простое.

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter2\template.sql на компакт-диске, прилагаемом к книге.

2.8. Работа с несколькими таблицами

На данный момент мы писали достаточно простые запросы, потому что использовали только одну таблицу. Но в *главе 1* мы уже создали четыре таблицы, и хотелось бы научиться связывать их в одно целое. Я бы не стал усложнять базу данных, создавая справочники, если бы связь не была возможной.

Рассмотрим пример. В таблице `Peoples` есть поле `idPosition`. В нем содержится идентификатор (первичный ключ) строки, с которой связана строка из таблицы `tbPosition`. Чтобы связать эти таблицы, напишем:

```
SELECT *
FROM tbPeoples, tbPosition
WHERE tbPeoples.idPosition=tbPosition.idPosition
```

На рис. 2.3 вы можете увидеть результат работы данного запроса.

Первая строчка, как всегда, говорит, что надо вывести все поля (`SELECT *`). Вторая строчка говорит, из каких таблиц надо получать данные (`FROM tbPeoples, tbPosition`). На этот раз здесь указано сразу две таблицы — с данными о работниках и должностях. Третья строчка показывает связь:

```
tbPeoples.idPosition=tbPosition.idPosition
```

Для того чтобы указать, к какой таблице относится поле `idPosition` (поля с таким именем есть в обеих таблицах, которые мы используем), мы записываем полное имя поля как `ИмяБазы.ИмяПоля`. Если имя поля уникально (как `vcFamil`, которое есть только в таблице `tbPeolpes`), то можно имя таблицы опускать. Как раз поэтому мы раньше опускали имя таблицы, когда использовали поля в секции `SELECT` и `WHERE`, ведь таблица была только одна, и никаких проблем возникнуть не могло. Если в секции `FROM` указать две таблицы, то можно столкнуться с конфликтами имен полей.

Итак, в секции `WHERE` мы указываем, что поле `idPosition` из таблицы `tbPeoples` равно полю `idPosition` из таблицы `tbPosition`.

SQL Query Analyzer - [Query - NOTEBOOK.FlenovSQLBook.NOTEBOOK\Administrator - E:\CyD\Book\SQL\Chapter2\link.sql]

File Edit Query Tools Window Help

FlenovSQLBook

```
-----------------------------------
-- Файл, содержащий примеры использования
-- связанных таблиц
-----------------------------------------

-- Связь с таблицей должностей
SELECT *
FROM tbPeoples, tbPosition
WHERE tbPeoples.idPosition=tbPosition.idPosition
```

	idPeoples	vcFamil	vcName	vcSurname	idPosition	dDateBirthDay	idPosition	idParentPosition	vcPositionName
1	1	ИВАНОВ	ИВАН	ИВАНЫЧ	1	1971-01-01 ...	1	NULL	Генеральный директор
2	2	ПЕТРОВ	ИВАН	ПАЛЫЧ	2	1971-04-01 ...	2	1	Коммерческий директор
3	3	СИДОРОВ	ИВАН	ПАЛЫЧ	3	1967-12-11 ...	3	1	Директор по общим вопросам
4	4	КОНОНОВ	ШВАРЦ	ПЕТРОВИЧ	4	1981-12-11 ...	4	2	Начальник отдела снабжения
5	5	СЕРГЕЕВ	СЕРГЕЙ	АЛЕКСЕЕВИЧ	5	1971-10-01 ...	5	2	Начальник отдела сбыта
6	6	ВАСИЛЬЕВ	ВАСЯ	ПЕТРОВИЧ	6	1981-09-01 ...	6	3	Начальник отдела кадров
7	7	ПИРАГОВ	АНДРЕЙ	КАРИМЫЧ	7	1969-01-08 ...	7	3	ОТиЗ
8	8	КАРТОШКИН	ВИКТОР	ВАСИЛЬЕВИЧ	8	1969-05-10 ...	8	2	Бухгалтерия
9	9	СМИРНОВ	ПЕТР	МИХАЛЫЧ	9	1969-06-01 ...	9	4	Менеджер по снабжению
10	10	МАРКОВКИН	АНДРЕЙ	ВАСИЛЬЕВИЧ	9	1969-09-03 ...	9	4	Менеджер по снабжению
11	11	СЕРЁЖКИН	АНДРЕЙ	ВАСИЛЬЕВИЧ	10	1969-01-05 ...	10	5	Менеджер по продажам
12	12	КОРОВА	ВЛАД	ИВАНОВИЧ	10	1969-08-02 ...	10	5	Менеджер по продажам
13	13	МИКОЛКИН	СЛАВИК	АЛЕКСЕЕВИЧ	11	1969-11-02 ...	11	5	Маркетолог
14	14	СУВОРКИН	ЛЕЛИК	ВИКТОРОВИЧ	11	1969-09-12 ...	11	5	Маркетолог
15	15	МОЛОТКОВ	БОЛИК	ПЕТРОВИЧ	12	1969-06-07 ...	12	5	Менеджер по рекламе
16	16	САДОВОДОВ	АНДРЕЙ	ИВАНЫЧ	12	1969-08-03 ...	12	5	Менеджер по рекламе
17	17	КУРОВ	ВИТЯ	НИКОЛАЕВИЧ	12	1969-04-03 ...	12	5	Менеджер по рекламе
18	18	МУРОВ	СЕРГЕЙ	СЕМЕНЫЧ	12	1969-02-02 ...	12	5	Менеджер по рекламе
19	19	СОВИН	АНДРЕЙ	ВАСИЛЬЕВИЧ	12	1969-09-08 ...	12	5	Менеджер по рекламе

Grids | Messages

NOTEBOOK (8.0) | NOTEBOOK\Administrator (51) | FlenovSQLBook | 0:00:01 | 19 rows | Ln 10, Col 1

Connections: 1 | Russian

Рис. 2.3. Результат запроса связанных таблиц

SQL Query Analyzer - [Query - NOTEBOOK.FlenovSQLBook.NOTEBOOK\Administrator - E:\CyD\Book\SQL\Chapter2\link.sql*]

File Edit Query Tools Window Help

FlenovSQLBook

```
-- Связь с таблицей должностей
SELECT *
FROM tbPeoples, tbPosition
WHERE tbPeoples.idPosition=tbPosition.idPosition
```

	idPeoples	vcFamil	vcName	vcSurname	idPosition	idPosition	idParentPosition	vcPositionName
1	1	ИВАНОВ	ИВАН	ИВАНЫЧ	1	1	NULL	Генеральный директор
2	1	ИВАНОВ	ИВАН	ИВАНЫЧ	1	2	1	Коммерческий директор
3	1	ИВАНОВ	ИВАН	ИВАНЫЧ	1	3	1	Директор по общим вопросам
4	2	ПЕТРОВ	ИВАН	ПАЛЫЧ	2	1	NULL	Генеральный директор
5	2	ПЕТРОВ	ИВАН	ПАЛЫЧ	2	2	1	Коммерческий директор
6	2	ПЕТРОВ	ИВАН	ПАЛЫЧ	2	3	1	Директор по общим вопросам
7	3	СИДОРОВ	ИВАН	ПАЛЫЧ	3	1	NULL	Генеральный директор
8	3	СИДОРОВ	ИВАН	ПАЛЫЧ	3	2	1	Коммерческий директор
9	3	СИДОРОВ	ИВАН	ПАЛЫЧ	3	3	1	Директор по общим вопросам

Grids | Messages

Query batch completed. | NOTEBOOK (8.0) | NOTEBOOK\Administrator (51) | FlenovSQLBook | 0:00:00 | Grid #1: 9 rows | Ln 1, Col 6

Connections: 1

Рис. 2.4. Результат запроса к двум таблицам без связи

Связь необходима. Если ее не указать, то результат будет совершенно другим. Посмотрим, что произойдет, если не указывать связь, а просто выбрать данные из двух таблиц:

```
SELECT *
FROM tbPeoples, tbPosition
```

Результат выполнения этого запроса показан на рис. 2.4. На рисунке я немного изменил результат, чтобы поле `idPosition` из таблицы `tbPeoples` находилось рядом с одноименным полем (с которым происходит связь) из таблицы `tbPosition`. Выделенный фрагмент содержит поля, которые принадлежат таблице должностей.

Теперь поищем строки с фамилией Иванов. Обратите внимание, что их много. Иванов оказался связан со всеми существующими должностями. Потом идут строки с фамилией Петров, который тоже связан со всеми возможными должностями. Таким образом, количество строк в результате равно количеству строк из первой таблицы, умноженному на количество строк из второй таблицы. Такое объединение называется ортогональным или декартовым.

Теперь попробуем понять, что означает связь:

```
tbPeoples.idPosition=tbPosition.idPosition
```

Будем смотреть на эту команду не как на связь, а как на простое ограничение `WHERE`, которое говорит, что в результате поля `idPosition` в обеих таблицах должны быть равны. Взгляните на результат работы запроса без связи. Где значения этих полей равны? Для Иванова это та строка, где обозначена связь с должностью генерального директора. Для Петрова это строка, содержащая связь с должностью коммерческого директора и т. д. Таким образом, все лишние записи отбрасываются, и мы получаем в результате только те строки, которые связаны по правильному ключу.

Почему база данных сама не указывает связь? Просто среди таблиц может быть несколько ключей, в том числе две таблицы могут связываться разными способами, и сервер просто не может знать, какая именно связь нам нужна в данный момент. Обычно фамилия, имя, отчество и дата рождения образуют уникальное сочетание. Вы можете попытаться связать две таблицы по этим данным, не обращая внимания на внешние ключи, чтобы определить для себя какие-то дополнительные сведения.

При написании связи нам пришлось указать полные имена таблиц. Чтобы обойтись без этого, можно в запросе создавать псевдонимы. Они создаются

в секции `FROM`, ставятся через пробел после имени таблицы и действуют только внутри этого запроса:

```
SELECT *
FROM tbPeoples pl, tbPosition ps
WHERE pl.idPosition=ps.idPosition
```

В данном запросе используются две таблицы, и для каждой указывается псевдоним. Для таблицы `tbPeoples` это псевдоним `pl`, а для таблицы `tbPosition` это `ps`. В качестве псевдонима может выступать любое имя, состоящее из любого количества букв. Я чаще всего использую первую букву, если она не будет конфликтовать с другими именами. В данном случае имена обеих таблиц начинаются с буквы `p`, вот и приходится использовать две буквы.

В секции `WHERE` теперь не надо писать полное имя таблицы. Достаточно только указывать псевдоним:

```
pl.idPosition=ps.idPosition
```

Удобство использования псевдонимов очень хорошо заметно при связывании нескольких таблиц. Давайте посмотрим, как связывать три таблицы:

```
SELECT *
FROM tbPeoples pl, tbPosition ps, tbPhoneNumbers pn
WHERE pl.idPosition=ps.idPosition
 AND pl.idPeoples=pn.idPeoples
```

В секции `FROM` перечислены уже три таблицы, а в секции `WHERE` установлены две связи. Таблица `tbPeoples` связана и с таблицей должностей, и с таблицей номеров телефонов.

В таблице работников 19 записей, а в таблице номеров телефонов 18 строк. Проанализируйте результат, и вы увидите, что некоторые работники имеют по несколько номеров телефонов. Например, строка с фамилией Иванов встречается 3 раза, но с разными номерами. Те работники, которые не имеют телефонов, в результат не попали. Почему? Просто нет связи, а значит, условие `pl.idPeoples=pn.idPeoples` не срабатывает.

Использование жесткого объединения с помощью знака равенства называется внутренним объединением. Чтобы увидеть записи, которые не связаны, нужно использовать внешнее объединение, которое бывает левым или правым.

Как же тогда увидеть всех работников и при этом установить связь? Для этого используются левые объединения:

```
SELECT *
FROM tbPeoples pl, tbPosition ps, tbPhoneNumbers pn
```

```
WHERE pl.idPosition=ps.idPosition
 AND pl.idPeoples*=pn.idPeoples
```

Самое интересное кроется как раз в последнем условии:

```
pl.idPeoples*=pn.idPeoples
```

Обратите внимание, что слева от знака равенства стоит знак "*". Это значит, что из таблицы работников (`tbPeoples`, которая находится со стороны звездочки) нужно взять все строки, а если есть связь с таблицей, указанной справа, то отобразить ее.

Выполните этот запрос, и вы увидите всех работников. У тех, кто не имеет телефона, поля из таблицы `tbPhoneNumbers` будут содержать нулевые значения `NULL`.

Использование знака "*" для нежесткого объединения описано в стандарте SQL, но я говорил, что не все базы данных поддерживают этот стандарт полностью. Например, MS Access позволяет создавать левые объединения, но здесь это делается совершенно по-другому. Мы рассмотрим этот метод в *разд. 2.9*.

Жесткое объединение таблиц с помощью знака равенства называют внутренним. Нежесткое объединение со звездочкой бывает правым или левым. Определить направление поможет звездочка. Мы ее поставили слева, значит, объединение было левым. Если бы звездочка была справа от знака равенства, то объединение стало бы правым.

Чтобы получить правое объединение, достаточно поменять поля местами:

```
SELECT *
FROM tbPeoples pl, tbPosition ps, tbPhoneNumbers pn
WHERE pl.idPosition=ps.idPosition
 AND pn.idPeoples=*pl.idPeoples
```

Вот теперь знак звездочки находится справа. На взгляд, разница в них небольшая, но она значительна при использовании объединения таблиц по методу MS.

Псевдонимы можно использовать в любой секции, даже в секции `SELECT`:

```
SELECT pl.vcFamil, pl.vcName, pl.vcSurname,
   ps.vcPositionName, pn.vcPhoneNumber
FROM tbPeoples pl, tbPosition ps, tbPhoneNumbers pn
WHERE pl.idPosition=ps.idPosition
 AND pl.idPeoples*=pn.idPeoples
```

Бывают случаи, когда использовать псевдонимы необходимо. Если есть имя поля, которое присутствует одновременно в обеих таблицах, то для его объ-

явления в секции `SELECT` необходимо явно указать таблицу. Например, попробуйте добавить в список `SELECT` поле `idPeoples` без указания имени таблицы или псевдонима. В ответ на этот запрос сервер выдаст ошибку: `Ambiguous column name 'idPeoples'` (двусмысленное имя колонки `idPeoples`). Сервер не знает, значение колонки `idPeoples` из какой таблицы нужно вернуть пользователю. Это вы знаете, что благодаря сравнению `pl.idPeoples*=pn.idPeoples` в результате все равно обе колонки будут содержать одно и то же значение, но сервер просто выдает сообщение о двусмысленности.

При использовании связанных таблиц очень часто возникает необходимость выбрать одну таблицу полностью, а другие только частично. Например, выберем из таблицы `tbPeoples` только фамилию, имя и отчество, а из таблиц должностей и телефонов все поля:

```
SELECT vcFamil, vcName, vcSurname, ps.*, pn.*
FROM tbPeoples pl, tbPosition ps, tbPhoneNumbers pn
WHERE pl.idPosition=ps.idPosition
 AND pl.idPeoples*=pn.idPeoples
```

Обратите внимание, что поля, которые нам нужны из таблицы `tbPeoples`, указаны явно, а чтобы не перечислять все поля остальных таблиц, мы просто пишем `ps.*` или `pn.*`. То есть знак звездочки, означающий вывод всех полей, относится не ко всем таблицам, а только к тем, которые он сопровождает.

В *разд. 1.2.6* мы рассматривали пример создания таблицы, в которой внешний ключ был связан с первичным ключом той же самой таблицы. В нашей тестовой базе данных такой таблицей является `tbPosition`, где хранятся должности работников. В этой таблице поле `idParentPosition` связано с первичным ключом `idPosition` этой же таблицы и предназначено для указания названия должности, которая является главной. Таким образом, можно построить дерево "главный—подчиненный".

Давайте попробуем вывести табличку из двух полей, где первое поле будет отображать название должности, а второе — главную должность. Вот как это будет выглядеть в виде SQL-запроса:

```
SELECT p1.vcPositionName AS 'Должность',
    p2.vcPositionName AS 'Главная должность'
FROM tbPosition p1, tbPosition p2
WHERE p1.idParentPosition*=p2.idPosition
```

Прежде чем мы разберем этот запрос, давайте посмотрим на результат его работы:

```
ДОЛЖНОСТЬ                    ГЛАВНАЯ ДОЛЖНОСТЬ

Генеральный директор         NULL

Коммерческий директор        Генеральный директор

Директор по общим вопросам   Генеральный директор

Начальник отдела снабжения   Коммерческий директор

Начальник отдела сбыта       Коммерческий директор

Начальник отдела кадров      Директор по общим вопросам

ОТиЗ                         Директор по общим вопросам

Бухгалтерия                  Коммерческий директор

Менеджер по снабжению        Начальник отдела снабжения

Менеджер по продажам         Начальник отдела сбыта
```

Первая строка соответствует должности генерального директора. Это самый главный человек в компании, поэтому здесь во второй колонке указан `NULL`, т. е. главной должности нет.

Следующая строка в первой колонке содержит должность коммерческого директора. Вполне логично, что он подчиняется генеральному директору, что и отображено во второй колонке результата.

Теперь посмотрим на секцию `FROM`, где дважды указана одна и та же таблица `tbPosition`, но с разными псевдонимами `p1` и `p2`. В секции `WHERE` мы устанавливаем связь между псевдонимами одной и той же таблицы:

```
p1.idParentPosition*=p2.idPosition
```

Таким образом, через псевдонимы мы указали связь внутри одной и той же таблицы. В секции `SELECT` мы отображаем название должности из первого псевдонима и название из второго.

Теперь посмотрите на следующий запрос:

```
SELECT p1.vcPositionName AS 'Должность',
 p2.vcPositionName AS 'Главная должность',
 p3.vcPositionName AS 'Главная для главной'
FROM tbPosition p1, tbPosition p2, tbPosition p3
WHERE p1.idParentPosition=p2.idPosition
 AND p2.idParentPosition*=p3.idPosition
```

Здесь мы дважды использовали связь таблицы саму на себя. Результат работы этого запроса:

Должность	Главная должность	Главная для главной
Коммерческий директор	Генеральный директор	NULL
Директор по общим вопросам	Генеральный директор	NULL
Начальник отдела снабжения	Коммерческий директор	Генеральный директор
Начальник отдела сбыта	Коммерческий директор	Генеральный директор

Давайте теперь напишем запрос, который отобразит все записи из всех связанных таблиц тестовой базы данных. Таблиц у нас всего четыре, но здесь в секции FROM будет пять таблиц, потому что дважды будет ссылка на таблицу должностей, чтобы отобразить должность текущего работника и должность его непосредственного начальника:

```
SELECT p1.vcFamil, p1.vcName, p1.vcSurname, dDateBirthDay,
 p1.vcPositionName AS 'Должность', p2.vcPositionName AS 'Начальник',
 pn.vcPhoneNumber, pt.vcTypeName1
FROM tbPeoples p1, tbPosition p1, tbPosition p2,
  tbPhoneNumbers pn, tbPhoneType pt
WHERE p1.idPosition=p1.idPosition
 AND p1.idParentPosition*=p2.idPosition
 AND pn.idPeoples=p1.idPeoples
 AND pt.idPhoneType=pn.idPhoneType
```

Попробуйте разобраться в этом запросе. Если вы поймете его, то можно считать, что тема связанных таблиц усвоена удачно.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\link.sql на компакт-диске, прилагаемом к книге.

2.9. Объединение в стиле Microsoft

Объединение по стандарту SQL, который мы рассматривали в *разд. 2.8*, описывает условие связи в секции WHERE. В MS связи перенесены в секцию FROM. На мой взгляд, это, как минимум, неудобно для создания и для чтения связей. И все же, метод MS мы рассмотрим, ведь только с его помощью в MS Access можно создать левое или правое объединение, и этот же метод поддерживается в MS SQL Server.

Ортогональное объединение по методу MS, т. е. без указания связи:

```
SELECT *
FROM tbPeoples CROSS JOIN tbPosition
```

Внутреннее объединение (эквивалентно знаку равенства) по методу MS описывается следующим образом:

```
SELECT *
FROM tbPeoples pl INNER JOIN tbPosition ps
     ON pl.idPosition=ps.idPosition
```

Как видите, для этого метода не нужна секция `WHERE`, но зато намного больше приходится писать. Вначале мы указываем, что нам нужно внутреннее объединение (`INNER JOIN`). Слева и справа от этого оператора задаем таблицы, которые нужно связать. После этого пишем ключевое слово `ON`, и только теперь устанавливаем связь между полями связанных таблиц. Таким образом, этот запрос эквивалентен следующему:

```
SELECT *
FROM tbPeoples pl, tbPosition ps
WHERE pl.idPosition=ps.idPosition
```

Самая большая путаница начинается, когда нужно объединить три таблицы в одно целое. Посмотрите на следующий запрос:

```
SELECT *
FROM tbPeoples pl LEFT OUTER JOIN tbPhoneNumbers pn
     ON pl.idPeoples=pn.idPeoples
     INNER JOIN tbPosition ps
     ON pl.idPosition=ps.idPosition
```

Сначала объединяются таблицы `tbPeoples` и `tbPhoneNumbers` через внешнее левое объединение (`LEFT OUTER JOIN`). Затем указывается связь между этими таблицами. А вот теперь результат объединения связываем внутренним объединением (`INNER JOIN`) с таблицей `tbPosition`. Внимательно осмотрите запрос, чтобы понять его формат, и что в нем происходит.

Чтобы получить правое объединение, необходимо просто поменять перечисление таблиц местами:

```
SELECT *
FROM tbPhoneNumbers pn RIGHT OUTER JOIN tbPeoples pl
     ON pl.idPeoples=pn.idPeoples
     INNER JOIN tbPosition ps
     ON pl.idPosition=ps.idPosition
```

Меняется местами перечисление таблиц, а вот порядок указания связанных полей не имеет особого значения, и здесь ничего переставлять не надо.

Вообще-то, мне не очень нравится объединение по методу Microsoft. В стандарте есть все то же самое, только намного проще и нагляднее.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\mslink.sql на компакт-диске, прилагаемом к книге.

2.10. Простейшие расчеты

В SQL включена возможность выполнения простейших математических операций над полями, и не только над числовыми, но и над строковыми. Например, иногда пользователю очень удобно вывести на экран фамилию, имя и отчество в одном поле, а ведь в нашей базе данных эти поля разделены.

```
SELECT vcFamil+' '+vcName+' '+vcSurname
FROM tbPeoples
```

Здесь мы объединяем текстовые поля таблицы в одно целое поле, вставляя между ними пробелы.

Если знак сложения стоит между числовыми полями, то происходит не объединение, а сложение числовых значений:

```
SELECT vcFamil+' '+vcName+' '+vcSurname, idPeoples+idPosition
FROM tbPeoples
```

В данном примере мы складываем значения полей `idPeoples` и `idPosition`. Конечно, именно в таком сложении смысла нет, потому что это идентификаторы. Но сама операция довольно полезна. Например, если у вас есть таблица, в которой указаны цены и количество товаров на складе, то можно перемножить цену и количество:

```
SELECT название, количество, цена, количество*цена AS сумма
FROM Таблица
```

Для удобства я использовал в качестве названий полей русские слова. В этом случае нет необходимости создавать вычисляемые поля, которые мы рассматривали в *разд. 1.2.8*, хотя, в зависимости от задачи, вычисляемое поле может сэкономить ресурсы. Ведь вычисление в нем происходит, только когда данные изменяются, а если расчеты производить в запросе, то они будут выполняться при каждом обращении к таблице с данными.

Если к таблице обращается 1000 пользователей в минуту, и при этом данные изменяются очень редко, то вычисляемое поле будет намного эффективнее. Но если к данным обращается 1 человек раз в час, а изменяют их 100 человек каждые 5 минут, то тут уже намного эффективнее будет выполнять вычисления в запросе. Как видите, выбираемый метод зависит от ситуации, поэтому трудно сказать, когда и что удобнее использовать.

Математические операции можно производить не только над полями. Например, давайте на время представим, что поле `idPosition` — это не ключ, а сумма заработной платы в месяц в тысячах долларов. Как узнать зарплату в рублях? Очень просто:

```
SELECT vcFamil+' '+vcName+' '+vcSurname, idPosition*1000*27
FROM tbPeoples
```

Здесь значение в поле `idPosition` сначала умножается на 1000 (мы же договорились, что зарплата в тысячах долларов), а потом на 27 (это примерный курс доллара на момент написания этой книги).

Вычисляемые поля не имеют имени, поэтому в результате в соответствующей колонке может быть написано что-то типа `(no column name)`. В некоторых программах вообще имя поля будет пустым. Чтобы задать имя, используйте ключевое слово `AS`.

А что получится, если прибавить к числовому полю текст? Если исходить из стандарта, то, по идее, все преобразования должны происходить автоматически. В SQL Server придется явно приводить тип:

```
SELECT vcFamil+' '+vcName+' '+vcSurname,
 cast((idPosition*1000*27) AS varchar(50))+' руб'
FROM tbPeoples
```

Результат работы запроса можно увидеть на рис. 2.5. В этом примере используется функция `cast`, которую мы еще не рассматривали, но обязательно рассмотрим в будущем. А пока вам достаточно видеть, что все возможно.

С помощью подобных операций можно улучшать вывод. Например, выведем в одной колонке фамилию, имя и отчество, а в другой — номер телефона. Но перед номером будем выводить слово `Телефон`. Эта задача решается следующим запросом:

```
SELECT vcFamil+' '+vcName+' '+vcSurname AS 'ФИО',
   'Телефон: '+vcPhoneNumber AS 'Номер телефона'
FROM tbPeoples pl, tbPhoneNumbers ps
WHERE pl.idPeoples=ps.idPeoples
```

В данном примере мы объединяем слово `'Телефон: '` и содержимое поля `vcPhoneNumber`.

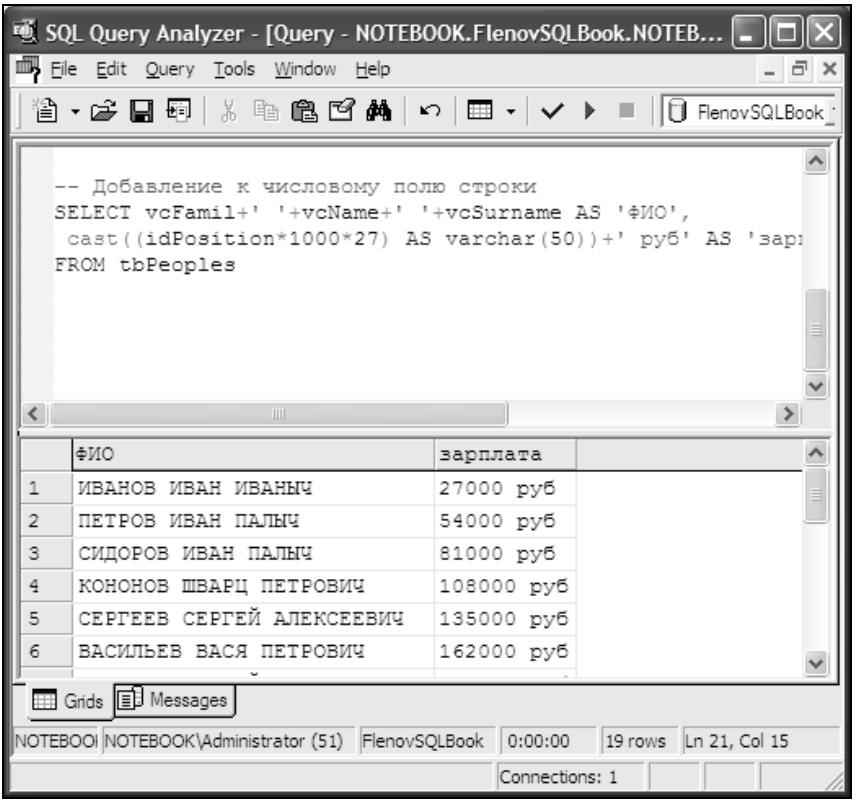


Рис. 2.5. Результат работы вычисляемых полей

Здесь есть маленький недостаток: приходится перечислять поля, которые нужно вывести, и "*" уже не работает. Так что, если понадобится вывести все поля, то придется их все писать после команды `SELECT`.

Помимо простых математических операций над отдельными строками, можно выполнять расчеты над всей таблицей. В стандарт заложены следующие команды:

- `COUNT` — подсчет количества строк;
- `SUM` — подсчет суммы;
- `AVG` — подсчет среднего значения;
- `MAX` — поиск максимального значения;
- `MIN` — поиск минимального значения.

Всем этим функциям нужно после имени в скобках указать имя поля, по которому нужно производить расчет. Единственная функция, которой все равно,

что указывать в скобках, — `COUNT()`. Неважно, в каком поле считать количество строк, оно всегда будет одним и тем же. Поэтому для этой функции можно указывать любое поле, а большинство программистов просто указывает знак звездочки (считать по всем полям).

Теперь рассмотрим примеры использования этих функций. Следующий запрос определяет количество записей в таблице `tbPeoples`:

```
SELECT COUNT(*)
FROM tbPeoples
```

В результате мы получим результат в виде таблицы, но всего из одной строки и одной колонки, которая будет содержать только одно число — количество записей. Для нашей таблицы `tbPeoples` это число 19.

А если нужно определить количество записей, в которых поле `vcName` содержит имя Андрей? Этот запрос не намного сложнее, нужно всего лишь в разделе `WHERE` указать нужное условие:

```
SELECT COUNT(*)
FROM tbPeoples
WHERE vcName='Андрей'
```

В нашей таблице целых 5 записей с таким именем.

Вторая часто используемая функция — `SUM()`, которая считает сумму значений во всех строках. Давайте сосчитаем сумму значений в колонке `idPosition`:

```
SELECT SUM(idPosition)
FROM tbPeoples
```

Допустим, что нам нужно определить самую позднюю дату рождения. Как это можно сделать? Конечно же, с помощью функции `MAX()`:

```
SELECT MAX(dDateBirthDay)
FROM tbPeoples
```

В скобках указывается поле, максимальное значение которого нужно найти.

Чтобы узнать самую раннюю дату рождения, нужно использовать функцию `MIN()`:

```
SELECT MIN(dDateBirthDay)
FROM tbPeoples
```

Для определения среднего значения используем функцию `AVG()`:

```
SELECT AVG(dDateBirthDay)
FROM tbPeoples
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\calculate.sql.

Зачем нужно вставлять в запросы математические вычисления, ведь это можно сделать программно? Вы можете получить данные из базы и потом выполнять вычисления, но по возможности лучше вставить вычисления в запрос. В этом случае вы освободите клиентскую машину от лишней загрузки, к тому же сервер будет выполнять эти расчеты намного быстрее. Когда вы делаете расчеты программно, то выполняются два просмотра всей базы. Во время первого выбираются данные, а во время второго идет расчет. Когда математические вычисления вставлены в запрос, то все действия выполняются за один проход: выбираются данные и одновременно производятся математические расчеты, и все это происходит на сервере, который, как показывает практика, сделает это намного быстрее, чем клиентский компьютер, ведь серверы обычно являются достаточно мощными.

2.11. Сортировка

Для сортировки используется оператор `ORDER BY`. Если посмотреть на общий вид команды `SELECT`, который мы обсуждали в *разд. 2.1*, то легко заметить, что этот оператор должен стоять последним. Мы не рассмотрели операторы `GROUP BY` и `HAVING`, которые описываются раньше сортировки, поскольку порядок указания операторов не всегда удобен с точки зрения последовательности их изучения. В данном случае `GROUP BY` и `HAVING` лучше рассмотреть в последнюю очередь.

После оператора `ORDER BY` через запятую указываются поля, по которым будет происходить сортировка. В самом конце нужно поставить `ASC` (в порядке возрастания) или `DESC` (в порядке убывания). Если ничего не указать, то подразумевается параметр `ASC`, и таблица сортируется по возрастанию. Например, выведем содержимое таблицы `tbPeoples`, отсортированное по фамилии:

```
SELECT *
FROM tbPeoples
ORDER BY vcFamil
```

Можно отсортировать и по нескольким полям. Для этого нужно перечислить их через запятую. Например, следующий запрос сортирует результат по фамилии, имени и отчеству:

```
SELECT *
FROM tbPeoples
ORDER BY vcFamil, vcName, vcSurname
```

Теперь посмотрим, как можно отсортировать все колонки результата по убыванию:

```
SELECT *
FROM tbPeoples
ORDER BY vcFamil DESC, vcName DESC, vcSurname DESC
```

Обратите внимание, что после каждого имени поля стоит ключевое слово `DESC`. Если хоть один раз не указать `DESC`, то будет использоваться значение по умолчанию — `ASC`, и по этому полю произойдет сортировка по возрастанию. Да, можно одно поле отсортировать в одну сторону, а другое поле в другую, например:

```
SELECT *
FROM tbPeoples
ORDER BY vcFamil ASC, vcName DESC, vcSurname DESC
```

Так как `ASC` является значением по умолчанию, то его можно опускать. Это значит, что такой запрос идентичен предыдущему:

```
SELECT *
FROM tbPeoples
ORDER BY vcFamil, vcName DESC, vcSurname DESC
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\order.sql на компакт-диске, прилагаемом к книге.

2.12. Группировка

Очень интересных эффектов можно добиться, если вместе с `GROUP BY` использовать математические вычисления. Как сосчитать количество различных имен в таблице `tbPeoples`?

Если использовать только ту информацию, которую мы уже знаем, то проблема решается достаточно сложно. Для начала мы должны определить уникальные имена, которые существуют в таблице:

```
SELECT DISTINCT vcName
FROM tbPeoples
```

После этого нужно определить количество появлений каждого имени в таблице. Например, количество Андреев можно узнать следующим образом:

```
SELECT count(*)
FROM tbPeoples
WHERE vcName='Андрей'
```

Но это сложно и требует ручного вмешательства. Конечно же, можно было бы использовать подзапросы, но их мы еще не рассматривали. Самое простое решение кроется как раз в операторе `GROUP BY`. Рассмотрим пример:

```
SELECT vcName, count(*)
FROM tbPeoples
GROUP BY vcName
```

Оператор `GROUP BY` группирует записи по указанным после оператора через запятую именам колонок. После оператора `SELECT` нужно перечислить те же имена колонок и математическую функцию, которую вы хотите использовать. В данном случае используется функция `COUNT()` для подсчета количества строк в группе. Итак, в нашей таблице несколько Андреев, в запросе они объединяются в группу, и в результате выводится на экран количество строк в этой группе.

Результат выполнения запроса:

```
АНДРЕЙ  5
БОЛИК   1
ВЛАД    1
ИВАН    3
ЛЕЛИК   1
СЕРГЕЙ..2
СЛАВИК. 1
...
```

В первой колонке показано имя работника, а во второй колонке количество записей с таким именем.

Рассмотрим еще пример. Сосчитаем, сколько раз встречаются в таблице одинаковые записи в полях фамилии и имени:

```
SELECT vcFamil, vcName, count(*)
FROM tbPeoples
GROUP BY vcFamil, vcName
```

Результат — количество повторений из сочетания полей фамилии и имени. В моей тестовой таблице содержимое этих двух полей образует уникальное значение, поэтому в колонке количества будет всегда единица.

Отсортируем строки по колонке количества записей:

```
SELECT vcName, count(*)
FROM tbPeoples
GROUP BY vcName
ORDER BY count(*) DESC
```

В операторе `ORDER BY` без проблем можно писать функции. Чтобы сценарий был более красивым, зададим псевдоним для поля количества записей:

```
SELECT vcName, count(*) AS ct
FROM tbPeoples
GROUP BY vcName
ORDER BY ct DESC
```

Чтобы лучше понять работу этого оператора, необходимо рассмотреть еще несколько примеров. Следующий запрос определяет количество повторений отчества:

```
SELECT vcSurName, count(*)
FROM tbPeoples
GROUP BY vcSurName
```

Обратите внимание, что поля, которые указываются в группировке, обязательно присутствуют в операторе `SELECT`. Другие поля там не могут присутствовать. Например, следующий запрос будет неверен:

```
SELECT vcFamil, vcSurName, count(*)
FROM tbPeoples
GROUP BY vcSurName
```

Чтобы лучше понять почему, давайте разберем его работу. Допустим, есть таблица из двух колонок — фамилии и отчества:

```
ИВАНОВ    ИВАНЫЧ
ПЕТРОВ    ПАЛЫЧ
СИДОРОВ   ПАЛЫЧ
```

Во время группировки по отчеству вторая и третья строки должны восприниматься как одно целое, но какую из двух фамилий вывести в результате: Петров или Сидоров? Вот по этой причине в разделе `SELECT` должны быть только те поля, по которым происходит группировка.

Давайте рассмотрим пример связанных таблиц. Допустим, нужно определить количество номеров телефонов для каждого пользователя. В этом случае должно быть подсчитано, сколько записей в таблице `tbPhoneNumbers` соответствует каждой записи в таблице `tbPeoples`. Лучше будет сгруппировать по первичному ключу таблицы `tbPeoples`, потому что он обеспечивает уникальность строк, для которых нужно определить количество записей в другой таблице. Так как у нас группировка происходит по одной таблице, а количество считается по другой таблице, в группировку можно добавлять любые поля помимо ключевого. Например:

```
SELECT pl.idPeoples, vcFamil, vcSurName, COUNT(vcPhoneNumber)
FROM tbPeoples pl, tbPhoneNumbers pn
```

```
WHERE pl.idPeoples *= pn.idPeoples
GROUP BY pl.idPeoples, vcFamil, vcSurName
ORDER BY COUNT(vcPhoneNumber) DESC
```

Рассмотрим этот запрос. Я решил вывести на экран помимо первичного ключа еще и фамилию и имя. Все эти поля перечислены в разделе `SELECT` и `GROUP BY`. Можно взять и другие поля из таблицы `tbPeoples`, но только из этой таблицы. В разделе `WHERE` устанавливается связь между таблицами, а в разделе `ORDER BY` сортируется количество найденных номеров телефонов.

С помощью `GROUP BY` можно не только определять количество записей с помощью оператора `COUNT`, но и суммы. Вспомним, что в нашей базе данных есть еще таблица товаров из следующих полей: `Дата покупки`, `Название товара`, `Цена`, `Количество`. Давайте сгруппируем таблицу по названию и определим количество каждого товара:

```
SELECT [Название товара], SUM(Количество)
FROM Товары
GROUP BY [Название товара]
```

В этом примере с помощью группировки мы определили сумму по колонке с помощью функции `SUM()`.

С помощью секции `HAVING` очень удобно ограничивать вывод. Например, вам нужно вывести сумму количества товаров, но при этом должны отражаться только те записи, в которых количество более 1. Просто `GROUP BY` тут уже не поможет. Надо добавить секцию `HAVING` с нужным условием:

```
SELECT [Название товара], SUM(Количество)
FROM Товары
GROUP BY [Название товара]
HAVING SUM(Количество)>1
```

В секции `HAVING` мы написали, что сумма количества товаров (`SUM(Количество)`) должна быть более 1.

Теперь посмотрим, как с помощью `HAVING` можно решить классическую задачу поиска двойных записей. Допустим, что нужно вывести на экран фамилии, которые повторяются в таблице более одного раза. Просто для подсчета фамилий достаточно использовать секцию `GROUP BY`, но если добавить еще и `HAVING`, то можно будет отобразить только двойные записи:

```
SELECT vcFamil, count(*)
FROM tbPeoples
GROUP BY vcFamil
HAVING count(*)>1
```

Где бы я ни работал, и как бы хорошо ни строилась база данных, мне приходилось регулярно выявлять двойные записи и избавляться от них, потому что они портят отчетность. В таких случаях группировка оказывается незаменимой. Да, можно вводить ограничения уникальности по тем полям, которые не должны дублироваться, но это не всегда эффективно.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\group_by.sql на компакт-диске, прилагаемом к книге.

2.13. Объединение запросов

Допустим, что есть какая-то задача, которую вы не можете решить в один запрос. Например, нужно выбрать из таблицы все записи, в которых в поле `vcName` указано имя `Андрей`, а в поле `vcSurName` указано отчество `Васильевич`. При этом, если запись удовлетворяет и тому, и другому условию, то она должна быть выведена в результате дважды. Как решить эту задачу? Если написать запрос:

```
SELECT vcFamil, vcName, vcSurName
FROM tbPeoples
WHERE vcName='Андрей'
  OR vcSurName='Васильевич'
```

то оба условия будут описаны верно, но в результате человек с именем и отчеством Андрей Васильевич попадет в результат только один раз, а нам надо два.

Проблема решается выполнением двух запросов. Сначала необходимо определить все записи с именем Андрей, а потом с отчеством Васильевич. Но это будет два набора результатов. Как их объединить в один? Для этого между запросами нужно написать один из следующих операторов:

- `UNION` — объединить результаты, и если есть строки, одинаковые для обоих результатов, то вывести только одну строку;
- `UNION ALL` — объединить результаты и включить туда все строки, даже повторяющиеся в обоих наборах.

Для решения нашей задачи нужно использовать `UNION ALL`, чтобы ни одна строка не исчезла, и если какая-то запись соответствует двум условиям, она должна попадать в результат дважды:

```
SELECT vcFamil, vcName, vcSurName
FROM tbPeoples
```

```
WHERE vcName='Андрей'

UNION ALL

SELECT vcFamil, vcName, vcSurName
FROM tbPeoples
WHERE vcSurName='Васильевич'
```

Как видите, это всего лишь два запроса, между которыми стоит ключевое слово `UNION ALL`. У объединения есть одно ограничение — сортировку надо указывать только во втором запросе. Если поставить оператор `ORDER BY` в первом запросе, то в результате вы получите ошибку. А если указать во втором, то сортироваться будет весь результат (обоих запросов) как одно целое.

Чтобы в результирующем наборе было видно, какая запись была получена с помощью первого запроса, а какая — второго, давайте добавим еще одну колонку к каждому из запросов, в которой будет отображаться текст с номером запроса:

```
SELECT vcFamil, vcName, vcSurName, 'Запрос 1'
FROM tbPeoples
WHERE vcName='Андрей'

UNION ALL

SELECT vcFamil, vcName, vcSurName, 'Запрос 2'
FROM tbPeoples
WHERE vcSurName='Васильевич'
```

У оператора `UNION` есть следующие ограничения в использовании:

- количество и типы полей должны быть одинаковыми;
- символьные поля должны иметь одинаковое количество символов.

Подобные запросы очень эффективны, когда нужно обращаться к разным базам данных. Например, один запрос выбирает данные из базы данных `Database1`, а другой из `Database2`.

Если одного поля в одном из запросов нет, то его можно заменить. Допустим, что во втором запросе происходит обращение к таблице, в которой нет поля фамилии. Как решить такую проблему? Вместо поля можно указать какой-либо текст, например:

```
SELECT vcFamil, vcName, vcSurName, 'Запрос 1'
FROM tbPeoples
```

```
WHERE vcName='Андрей'

UNION ALL

SELECT 'Не найдено', vcName, vcSurName, 'Запрос 2'
FROM tbPeoples
WHERE vcSurName='Васильевич'
ORDER BY vcFamil
```

В данном примере во втором запросе вместо поля фамилии указывается текст 'Не найдено'. Вместо несуществующего числового поля, конечно же, нужно указать число.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\union.sql на компакт-диске, прилагаемом к книге.

2.14. Подзапросы

Наиболее сложной, но в то же время наиболее интересной темой являются подзапросы. Это достаточно мощное средство получения необходимых данных, а с другой стороны, оно очень сильно снижает производительность обработки запроса сервером. Сначала мы научимся работать с подзапросами, потому что с их помощью можно быстро решить поставленную задачу, а потом будем учиться избавляться от подзапросов, что идентично оптимизации.

Рассмотрим пример, как можно определить номера телефонов, у которых установлен тип `'Сотовый рабочий'`. Для этого сначала необходимо узнать, какой первичный ключ у нужного типа телефона в таблице `tbPhoneType`:

```
SELECT idPhoneType
FROM tbPhoneType
WHERE vcTypeName = ('Сотовый рабочий')
```

После этого находим все записи в таблице `tbPhoneNumbers`, где поле `idPhoneType` содержит найденное значение типа телефона:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType = идентификатор
```

Эта задача достаточно просто решается с помощью подзапросов:

```
SELECT *
FROM tbPhoneNumbers
```

```
WHERE idPhoneType =
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName = ('Сотовый рабочий')
 )
```

В данном примере мы выбираем все записи из таблицы `tbPhoneNumbers`. При этом значение поля `idPhoneType` сравнивается с результатом подзапроса, который пишется в круглых скобках. Так как стоит знак равенства, то результат подзапроса должен состоять из одного поля и одной строки. Если в результате будет два поля или более одной строки, то сервер выдаст нам сообщение об ошибке.

Попробуем выполнить следующий запрос:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType =
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName in ('Сотовый рабочий', 'Сотовый домашний')
 )
```

Такой запрос вернет две строки с двумя значениями первичного ключа. В ответ на это сервер выдаст сообщение об ошибке:

Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

(Подзапрос возвращает более чем 1 значение. Это не позволено, когда подзапрос следует после знаков =, !=, <, <=, >, >= или когда подзапрос используется как выражение.)

А что же тогда можно использовать? Для подобного запроса знак равенства нужно заменить на оператор `IN`:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType IN
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName in ('Сотовый рабочий', 'Сотовый домашний')
 )
```

Сначала сервер SQL выполнит внутренний запрос, который расположен в скобках, и результат подставит во внешний запрос.

Я уже говорил, что результат подзапроса должен состоять только из одной колонки. Это значит, что нельзя написать во внутреннем запросе `SELECT *`, а можно только `SELECT ИмяОдногоПоля`. Помните, что имя должно быть только одно, и тип его должен совпадать с типом сравниваемого значения. Подзапросы нужно использовать очень аккуратно, потому что они могут привести к ошибке.

Очень важно, что подзапрос находится в скобках и справа от знака равенства. Стандарт не позволяет писать подзапросы слева. Это значит, что следующий запрос неверен:

```
SELECT *
FROM tbPhoneNumbers
WHERE
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName = ('Сотовый рабочий')
 ) = idPhoneType
```

В этом примере сначала идет подзапрос, потом знак равенства и только после этого указывается поле, с которым необходимо произвести сравнение.

Основной запрос (так же называемый внешним) может обращаться к подзапросу (внутренний запрос). Для этого таблицам необходимо указать псевдонимы. Посмотрим на следующий запрос:

```
SELECT *
FROM tbPhoneNumbers ot
WHERE idPhoneType IN
 (
  SELECT idPhoneType
  FROM tbPhoneType it
  WHERE vcTypeName in ('Сотовый рабочий', 'Сотовый домашний')
   AND ot.vcPhoneNumber LIKE '(923)%'
 )
```

Обратите внимание на предпоследнюю строчку:

```
AND ot.vcPhoneNumber LIKE '(923)%'
```

Здесь происходит сравнение поля `vcPhoneNumber` таблицы `ot` с шаблоном, где `ot` — псевдоним таблицы `tbPhoneNumbers`, которая описана в секции `FROM`

внешнего запроса. Но, несмотря на это, мы можем из подзапроса обращаться по псевдониму к столбцам внешних запросов. Таким образом можно устанавливать достаточно сложные связи между запросами.

Такой запрос будет выполняться по следующему алгоритму:

1. Выбрать строку из таблицы `tbPhoneNumbers` во внешнем запросе. Это будет текущая строка-кандидат.
2. Сохранить значения из этой строки-кандидата в псевдониме `ot`.
3. Выполнить подзапрос, при этом в поиске участвует и внешний запрос.
4. Оценить `idPhoneType` внешнего запроса на основе результатов подзапроса, выполняемого в предыдущем шаге. Он определяет, выбирается ли строка-кандидат для вывода на экран.

Подзапросы могут быть не только в секции `WHERE`, но и в секции `SELECT`, и в секции `FROM`. Давайте рассмотрим сначала подзапросы из `FROM`, для этого поставим достаточно сложную, но интересную задачу. Мы научились определять количество пользователей с именем Андрей с помощью группировки. А если создать таблицу, которая будет состоять из трех колонок:

- имя;
- количество пользователей с таким именем;
- количество номеров телефонов для каждого имени.

В виде двух запросов эта задача решается достаточно просто. Следующий запрос определяет количество строк для каждого имени в таблице:

```
SELECT p.vcName, COUNT (*) as PeopleNumber
FROM tbPeoples p
GROUP BY p.vcName
```

Такой запрос мы уже разбирали.

Теперь определим количество телефонов для каждого имени. Да, имени, а не работника. Нас интересует, сколько номеров телефонов у всех Андреев, Иванов и т. д. Эта задача решается с помощью связи из таблиц работников и телефонов, группировкой по имени и подсчетом количества телефонов:

```
SELECT pl.vcName, COUNT(vcPhoneNumber) AS PhoneNum
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pl.idPeoples *= pn.idPeoples
GROUP BY vcName
```

У нас получилось две разных таблицы. А как теперь их объединить в одно целое? Попробуйте самостоятельно решить эту задачу. Мое решение можно увидеть в листинге 2.3.

Листинг 2.3. Получение количества имен и количества телефонов у каждого имени

```
SELECT *
FROM
  (SELECT p.vcName, COUNT(*) as PeopleNumber
   FROM tbPeoples p
   GROUP BY p.vcName) p1,

  (SELECT pl.vcName, COUNT(vcPhoneNumber) AS PhoneNum
   FROM tbPeoples pl, tbPhoneNumbers pn
   WHERE pl.idPeoples *= pn.idPeoples
   GROUP BY vcName) p2
WHERE p1.vcName=p2.vcName
```

В секции `FROM` вместо указания таблиц стоят вышеописанные запросы, заключенные в круглые скобки. Для каждого запроса указывается псевдоним, иначе невозможно работать с полями запросов. Получается, что вместо того, чтобы получить данные непосредственно из таблицы, мы получаем их из запроса.

Для таких запросов есть только одно ограничение — у каждого поля в подзапросе секции `FROM` должно быть имя. У нас есть поля, подсчитывающие количество записей, и для таких полей имя не устанавливается, поэтому я установил псевдонимы. В первом подзапросе колонка с количеством записей названа `PeopleNumber`, а во втором подзапросе — `PhoneNum`.

Внешний объединяющий запрос связывает обе полученные таблицы через поле имени `vcName`, а результатом будет общая таблица (рис. 2.7), состоящая из четырех колонок — имя и количество из первого запроса, и имя и количество из второго запроса. Одну из колонок имен можно убрать, потому что они идентичны, но я решил оставить, чтобы вы могли увидеть связь.

Теперь посмотрим, как можно писать подзапросы в секции `SELECT`. Я решил просто вывести имена всех работников и номера их телефонов. Но при этом не устанавливать связь между таблицами работников и номерами телефонов в секции `FROM`, а искать номера с помощью подзапросов в секции `SELECT`:

```
SELECT pl.*,
 (
  SELECT vcPhoneNumber
  FROM tbPhoneNumbers pn
  WHERE pn.idPhoneNumbers=pl.idPeoples
 )
FROM tbPeoples pl
```

SQL Query Analyzer - [Query - NOTEBOOK.FlenovSQLBook.NOTEBOOK\Administrator - ...

File Edit Query Tools Window Help

FlenovSQLBook

```
-- Подзапросы в секции FROM
SELECT *
FROM
  (SELECT p.vcName, count(*) as PeopleNumber
   FROM tbPeoples p
   GROUP BY p.vcName)p1,

  (SELECT p1.vcName, COUNT(vcPhoneNumber) AS PhoneNum
   FROM tbPeoples p1, tbPhoneNumbers pn
   WHERE p1.idPeoples *= pn.idPeoples
   GROUP BY vcName) p2
WHERE p1.vcName=p2.vcName
```

	vcName	PeopleNumber	vcName	PhoneNum
1	АНДРЕЙ	5	АНДРЕЙ	2
2	БОЛИК	1	БОЛИК	0
3	ВАСЯ	1	ВАСЯ	2
4	ВИКТОР	1	ВИКТОР	1
5	ВИТЯ	1	ВИТЯ	0
6	ВЛАД	1	ВЛАД	0
7	ИВАН	3	ИВАН	8
8	ЛЕЛИК	1	ЛЕЛИК	1
9	ПЕТР	1	ПЕТР	1
10	СЕРГЕЙ	2	СЕРГЕЙ	1
11	СЛАВИК	1	СЛАВИК	0
12	ШВАРЦ	1	ШВАРЦ	2

Grids Messages

Query batch com NOTEBOOK (8.0) NOTEBOOK\Administrator (51) FlenovSQLBook 0:00:00 12 rows Ln 47, Col 23

Connections: 1

Рис. 2.7. Результат объединения двух таблиц

В секции `SELECT` сначала запрашиваем все колонки из таблицы `tbPeoples` (`p1.*`). После этого вместо очередного поля в скобках указывается подзапрос, который выбирает данные. При этом в подзапросе в секции `WHERE` устанавливается связь между таблицами. Получается, что и из этого подзапроса мы можем обращаться к полям внешнего запроса.

Единственное ограничение — подзапрос в секции `SELECT` должен возвращать только одну строку. Если результатом будет несколько строк, то запрос возвращает ошибку.

Все примеры, которые мы обсуждали выше, достаточно просто реализовать без использования подзапросов. Рассмотрим следующий запрос:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType =
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName = ('Сотовый рабочий')
 )
```

Эта же задача решается следующим образом:

```
SELECT pn.*
FROM tbPhoneNumbers pn, tbPhoneType pt
WHERE pn.idPhoneType = pt.idPhoneType
 AND vcTypeName = ('Сотовый рабочий')
```

Просто связываем обе таблицы и указываем то же самое условие. Таким образом мы избавились от подзапроса, и теперь сервер сможет выполнить задачу быстрее. Большинство задач можно решить без подзапросов, если правильно связать таблицы. Именно поэтому в данной книге я постараюсь как можно реже пользоваться подзапросами, и все решать одним оператором `SELECT`.

А вот в следующем запросе достаточно сложно обойтись без подзапросов. Допустим, что нужно определить данные последней добавленной в таблицу строки. Если в качестве первичного ключа используется автоматически увеличиваемое поле, то необходимо узнать наибольшее значение первичного ключа с помощью функции `MAX()`, а потом найти строку с этим ключом. Вот как определяется последняя строка в таблице `tbPeoples`:

```
SELECT *
FROM tbPeoples
WHERE idPeoples=
    (SELECT MAX(idPeoples) FROM tbPeoples)
```

Такой запрос позволит быстро увидеть результат работы, что окажется полезным, когда мы будем учиться добавлять записи в таблицу.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\subquery.sql на компакт-диске, прилагаемом к книге.

2.15. Операторы работы с подзапросами

Для работы с подзапросами есть несколько очень удобных операторов, которые мы еще не рассматривали. Первым оператором, с которым мы еще не работали, будет EXISTS. Этому оператору в качестве параметра передается запрос, и если запрос возвращает хотя бы одну строку, то оператор возвращает истинное значение, иначе ложь. В зависимости от этого внешний запрос выносит решение, включать текущую запись в результирующий набор или нет.

Операторы ANY, SOME и ALL также позволяют расширить возможности подзапросов и работы с ними. Операторы ANY и SOME чем-то похожи на IN, с которым мы работали. Но чтобы увидеть все преимущества, необходимо рассмотреть реальный запрос и разобрать его результат.

Теперь познакомимся поближе со всеми этими операторами и увидим их работу на практике.

2.15.1. Оператор *EXISTS*

По идее, оператор EXISTS нужно было рассмотреть намного раньше, потому что он достаточно прост. Но я решил повременить, потому что его удобно использовать совместно с подзапросами. Например, выведем на экран всех работников, у которых есть хоть какой-нибудь телефон. Как это сделать? Можно связать две таблицы. Тогда те записи, которые свяжутся, дадут необходимый результат. Это и будут работники, у которых есть номера телефонов.

Но не всегда так легко решить проблему. Иногда удобнее и проще воспользоваться оператором EXISTS, который возвращает истинное значение, если запрос, который указан в скобках после оператора, возвращает хотя бы одну строку. Например:

```
SELECT *
FROM tbPeoples pl
WHERE EXISTS (SELECT *
  FROM tbPhoneNumbers pn
  WHERE pl.idPeoples=pn.idPeoples)
```

В этом запросе мы запрашиваем все записи из таблицы tbPeoples, в качестве условия происходит проверка EXISTS. Для каждой записи работника выполняется запрос, который указан в скобках после EXISTS:

```
SELECT *
FROM tbPhoneNumbers pn
WHERE pl.idPeoples=pn.idPeoples
```

Здесь мы связываем с внешним запросом таблицу tbPhoneNumbers. Если этот запрос найдет хотя бы один телефон для текущего работника, то EXISTS возвращает истину, и строка попадает в результат.

Чтобы отобразить все записи работников, которые не имеют телефонов, достаточно перед EXISTS поставить оператор NOT:

```
SELECT *
FROM tbPeoples pl
WHERE NOT EXISTS (SELECT *
  FROM tbPhoneNumbers pn
  WHERE pl.idPeoples=pn.idPeoples)
```

Благодаря оператору NOT происходит обратная операция, т. е. в результат попадает та строка, для которой запрос после EXISTS не вернул ни одной строки.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\exists.sql на компакт-диске, прилагаемом к книге.

2.15.2. Операторы *ANY*, *SOME* и *ALL*

Операторы ANY и SOME абсолютно одинаковы. Оба они дают один и тот же результат, поэтому вы можете использовать тот, который больше нравится. Мне больше нравится ANY, потому что это слово короче аж на целую букву.

Я уже говорил, что ANY очень похож на IN. Рассмотрим следующий запрос с оператором IN:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType IN
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName in ('Сотовый рабочий', 'Сотовый домашний')
 )
```

В данном примере используется IN для сравнения поля idPhoneType с одним из значений, возвращаемым подзапросом. То же самое можно сделать с помощью оператора ANY, только используется он немного по-другому:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType > ANY
```

```
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName in ('Сотовый рабочий', 'Сотовый домашний')
 )
```

Так в чем же разница? А она кроется в том, как читается этот запрос. Попробуем проговорить его: "Выбрать все записи из таблицы `tbPhoneNumbers`, где поле `idPhoneType` равно любому из значений, полученных подзапросом". Разница в том, что `IN` работает так же, как и `= ANY`, но с оператором `ANY` можно использовать любые другие операторы сравнения (<>, !=, >, >=, !>, <, <=, !<). Например, следующий запрос выбирает все телефоны из таблицы `tbPhoneNumbers`, у которых `idPhoneType` больше любого из значений, полученных в подзапросе:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType > ANY
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName in ('Сотовый рабочий', 'Сотовый домашний')
 )
```

Если `ANY` сравнивает с любым из значений, и если `idPhoneType` будет больше хотя бы одного, то запись попадает в результат. Если `ANY` заменить на оператор `ALL`, то строка попадет в результирующий набор, только если она будет больше всех значений, которые возвращает подзапрос:

```
SELECT *
FROM tbPhoneNumbers
WHERE idPhoneType > ALL
 (
  SELECT idPhoneType
  FROM tbPhoneType
  WHERE vcTypeName in ('Сотовый рабочий', 'Сотовый домашний')
 )
```

Если `ANY` и `SOME` требуют, чтобы условие выполнилось хотя бы для одной строки результата подзапроса, то `ALL` будет требовать, чтобы условие выполнилось для всех строк.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\anyall.sql на компакт-диске, прилагаемом к книге.

2.16. Добавление записей

С выборкой данных мы разобрались и узнали все, что необходимо, об операторе SELECT. Я надеюсь, что мне удалось показать вам всю мощь выборки данных, и что теперь вы сможете самостоятельно писать запросы для решения своих задач.

Теперь давайте познакомимся с командами добавления данных в таблицу. Когда мы в начале главы заполняли таблицу данными, то вы должны были просто выполнить сценарий. Сейчас мы познакомимся с командой INSERT, которая добавляет в таблицу запись (листинг 2.4), и вы сможете понять, как работал сценарий заполнения таблицы из файла \Chapter2\fill_data.sql.

Листинг 2.4. Общий вид команды INSERT

```
INSERT [ INTO]
    { table_name WITH ( < table_hint_limited > [ ...n ] )
        | view_name
        | rowset_function_limited
    }

    {     [ ( column_list ) ]
        { VALUES
            ( { DEFAULT | NULL | expression } [ ,...n] )
            | derived_table
            | execute_statement
        }
    }
```

В принципе, ничего сложного в этой команде нет. Давайте рассмотрим различные варианты добавления данных в таблицу. Для начала добавим в таблицу строку, в которой все значения будут установлены сервером по умолчанию:

```
INSERT INTO tbPeoples
DEFAULT VALUES
```

Выясним, что добавил сервер в таблицу tbPeoples. Как? Нужно посмотреть последнюю строку, а этот запрос мы обсуждали в *разд. 2.14*. Напоминаю, как он выглядел:

```
SELECT *
FROM tbPeoples
```

```
WHERE idPeoples=
    (SELECT MAX(idPeoples) FROM tbPeoples)
```

В результате вы должны увидеть следующую таблицу:

```
idPeoples  vcFamil  vcName  vcSurname  idPosition  dDateBirthDay
---------------------------------------------------------------
20         NULL     NULL    NULL       NULL        NULL


(1 row(s) affected)
```

В таблице `tbPeoples` первое поле `idPeoples` является автоматически увеличиваемым, и в новой строке это поле стало равным 20. Остальные поля не имеют значений по умолчанию, и тут можно видеть только нулевые значения `NULL`.

Вернемся к запросу добавления записи и рассмотрим его подробнее. В данном запросе выполняется оператор `INSERT INTO` (вставить в), после которого указывается имя таблицы, в которую нужно вставить запись. Далее может идти описание имен полей, которые нужно заполнять, или список значений. В нашем случае нет ни того, ни другого, зато есть оператор `DEFAULT VALUES` (значения по умолчанию), который указывает на необходимость заполнить все поля значениями по умолчанию.

В таблице `tbPeoples` только у одного поля указывается значение по умолчанию — это автоматически увеличиваемое поле первичного ключа. Хотя мы не указывали значение `DEFAULT`, автоматически увеличиваемое поле обязательно получает значение, тем более что это первичный ключ, и такое поле обязательно должно быть заполнено. Все остальные поля будут нулевыми.

Следующий запрос пытается добавить запись с явно указанными данными:

```
INSERT INTO tbPeoples
VALUES(122, 'ИВАНОВ', 'ИВАН', 'СЕРГЕЕВИЧ', 1, '01.01.1971')
```

Первая строчка ничем не отличается от предыдущего запроса. Мы снова указываем, что строка добавляется в таблицу `tbPeoples`. Во второй строчке идет ключевое слово `VALUES`, после которого в скобках нужно перечислить значения для всех полей таблицы. Перечисление значений следует выполнить в том же порядке, что и список полей в таблице, и при этом указываемые значения должны иметь необходимый тип данных. Если хотя бы одно поле не указано вообще или указано неверно (например, вместо числа указана строка), то запрос не будет выполнен.

В данном запросе указаны все поля в соответствии со своими типами, но почему я сказал, что этот запрос "пытается добавить строку"? Догадались?

Первое поле имеет тип автоматически увеличиваемого числа. Это поле изменять нельзя, поэтому сервер вернет сообщение об ошибке:

An explicit value for the identity column in table 'tbPeoples' can only be specified when a column list is used and IDENTITY_INSERT is ON.

Явное значение для автоувеличиваемой колонки в таблице 'tbPeoples' может быть указано, только когда используется список колонок и свойство IDENTITY_INSERT установлено в ON.

Неужели нельзя добавить строку? Конечно же, есть способ решения этой проблемы, просто необходимо явным образом задать список колонок, которые должны изменяться, и установить для них нужные значения.

Список колонок указывается в скобках после имени таблицы:

```
INSERT INTO tbPeoples
 (idPeoples, vcFamil, vcName, vcSurname, idPosition, dDateBirthDay)
VALUES (122, 'ИВАНОВ', 'ИВАН', 'СЕРГЕЕВИЧ', 1, '01.01.1971')
```

В данном случае в скобках перечислены все поля в том порядке, в котором они описаны в таблице `tbPeoples`. Но это не обязательно. Поля могут быть указаны в любой последовательности, но значения в скобках после ключевого слова `VALUES` должны идти в том порядке, в котором вы их перечислили. Впрочем, об этом мы еще поговорим.

Далее необходимо установить нужное свойство. Мы пока никакие свойства не изменяли, а делается это с помощью оператора `SET`. Затем идет имя свойства (`IDENTITY_INSERT`), имя таблицы, свойство которой нужно изменить, и напоследок — значение (чаще всего `ON` или `OFF`):

```
SET IDENTITY_INSERT tbPeoples ON
```

Следует заметить, что изменение подобных свойств допустимо только в Transact-SQL и MS SQL Server.

Теперь посмотрим на полную SQL-команду добавления записи с явным указанием значения для автоматически увеличиваемого значения:

```
SET IDENTITY_INSERT tbPeoples ON
INSERT INTO tbPeoples
 (idPeoples, vcFamil, vcName, vcSurname, idPosition, dDateBirthDay)
VALUES(122, 'ИВАНОВ', 'ИВАН', 'СЕРГЕЕВИЧ', 1, '01.01.1971')
```

После выполнения команды желательно вернуть значение параметра `IDENTITY_INSERT` в `OFF`:

```
SET IDENTITY_INSERT tbPeoples OFF
```

Вот теперь команда будет выполнена корректно, и новая запись будет успешно добавлена в таблицу. Но я не рекомендую вам явно указывать значение

для автоматически увеличиваемого поля. Так вы можете нарушить целостность, поэтому доверьте выделение значения для поля `IDENTITY` серверу баз данных.

Как добавить строку и при этом не указывать значение ключевого поля, чтобы сервер смог его установить самостоятельно? Очень просто. Дело в том, что перечислять все поля необязательно. Можно указать только те имена полей, значения которых вы хотите явно изменить. Например, в следующем запросе мы не указываем значение для ключевого поля `idPeoples`:

```
INSERT INTO tbPeoples
 (vcFamil, vcName, vcSurname, idPosition, dDateBirthDay)
VALUES('ИВАНОВ', 'ИВАН', 'АЛЕКСЕЕВИЧ', 1, '01.01.1971')
```

В списке полей и в списке `VALUES` не указывается значение для поля `idPeoples`. Да и в списке значений после ключевого слова `VALUES` мы указываем значения, начиная с поля `vcFamil`. Вместо этого сервер самостоятельно добавит очередное значение точно так же, как когда мы добавляли строку из всех значений по умолчанию.

Давайте добавим строку, в которой укажем только фамилию работника. Для этого в списке полей нужно указать только имя поля `vcName`, а в списке значений должно быть только одно строковое значение, ведь фамилия имеет строковый тип:

```
INSERT INTO tbPeoples
 (vcFamil)
VALUES('ПЕТРОВ')
```

Если в вашей программе удобнее сделать перечисление всех полей, но при этом значениями заполняются далеко не все, вместо незаполняемых можно указать `NULL` или `DEFAULT`. В следующем примере вместо фамилии и даты рождения указано ключевое слово `DEFAULT`, и этим полям сервер установит значение по умолчанию. Для поля фамилии указывается значение `NULL`:

```
INSERT INTO tbPeoples
 (vcFamil, vcName, vcSurname, idPosition, dDateBirthDay)
VALUES('СЕРГЕЕВ', DEFAULT, NULL, 1, DEFAULT)
```

Ключевые слова `DEFAULT` и `NULL` можно указывать для любых типов полей, кроме автоматически увеличиваемого, потому что это поле не может содержать `NULL` и не имеет значения по умолчанию. Автоматическое увеличение к значению по умолчанию не относится.

В перечислении можно указывать поля в любом порядке, и только те, которые необходимы. При этом в списке `VALUES` значения должны идти в том же

порядке, в котором вы их перечисляли. Например, в следующем примере поля фамилии, имени и отчества заполняются в другом порядке. При этом между отчеством и именем идет поле `idPosition`:

```
INSERT INTO tbPeoples
 (vcSurname, idPosition, vcName, vcFamil)
VALUES('ПЕТРОВИЧ', 12, 'СЕРГЕЙ', 'СМИРНОВ')
```

В *разд. 2.6* мы рассматривали, как можно использовать `SELECT INTO` для создания новой таблицы. Теперь обсудим, как можно импортировать данные в уже существующую таблицу, ведь `SELECT INTO` не может выбирать данные в таблицу, если она уже существует.

Для этого создадим новую таблицу `tbPeoples2`, которая будет состоять из таких же полей, как и у `tbPeoples`. Единственное ее отличие заключается в том, что ключевое поле не будет автоматически увеличиваемым, чтобы избежать негативной реакции сервера на вставку значений в первичный ключ. В листинге 2.5 приведен SQL-код создания новой таблицы `tbPeoples2`.

Листинг 2.5. Создание новой таблицы — копии `tbPeoples`

```
CREATE TABLE tbPeoples2
(
  -- Описание полей
  idPeoples2 int,
  vcFamil varchar(50),
  vcName varchar(50),
  vcSurname varchar(50),
  idPosition int,
  dDateBirthDay datetime,

  -- Описание ключей
  CONSTRAINT PK_idPeoples2 PRIMARY KEY (idPeoples2),
  CONSTRAINT FK_idPosition2 FOREIGN KEY (idPosition)
     REFERENCES tbPosition (idPosition),

  -- Описание ограничений
  CONSTRAINT check_dDateBirthDay2 CHECK (dDateBirthDay<getdate())
)
```

Так как имена ограничений в таблице должны быть уникальными внутри базы данных, к каждому имени ограничения добавлена цифра 2. Имя таблицы

тоже содержит цифру два (`tbPeoples2`), т. е. все соответствует нашим договоренностям.

Теперь посмотрим, как заполнить новую таблицу записями о работниках из таблицы `tbPooples`, и при этом взять только те записи, у которых поле `idPosition` содержит значение большее или равное 10. Все достаточно просто:

```
INSERT INTO tbPeoples2
SELECT *
FROM tbPeoples
WHERE idPosition>=10
```

В первой строке пишем оператор `INSERT INTO` и имя таблицы `tbPeoples2`, в которую необходимо вставить данные. Затем идет просто запрос `SELECT`, в котором мы получаем нужные данные из таблицы `tbPeoples`. Все строки, полученные с помощью этого запроса, будут добавлены в таблицу `tbPeoples2`.

Все так просто только потому, что обе таблицы содержат одинаковое количество полей, и типы полей совпадают. А если поля будут разными? Давайте создадим третью таблицу `tbPeoples3`, в которой будут только поля для хранения фамилии, имени и даты рождения.

```
CREATE TABLE tbPeoples3
(
  -- Описание полей
  vcFamil varchar(50),
  vcName varchar(50),
  dDateBirthDay datetime,
)
```

Теперь посмотрим, как можно заполнить эту таблицу значениями:

```
INSERT INTO tbPeoples3
SELECT vcFamil, vcName, dDateBirthDay
FROM tbPeoples
WHERE idPosition>=10
```

Обратите внимание, что мы выбираем с помощью запроса `SELECT` только три поля и именно в таком порядке, в котором они описаны в таблице `tbPeoples3`. Названия полей в таблице и выбираемые поля могут не совпадать, но типы данных обязаны совпадать полностью, иначе выполнение запроса завершится ошибкой. Здесь сервер не может производить автоматическое преобразование, поэтому или типы должны совпадать, или вы должны их явно преобразовывать. О явном преобразовании мы поговорим в *разд. 2.21*.

С помощью оператора `INSERT INTO` вы можете копировать данные между таблицами или даже переносить их с сервера на сервер с помощью распределенных запросов, о которых мы поговорим в *разд. 5.1*.

Запрос `SELECT` в `INSERT INTO` может быть любой сложности, и тут ограничений на используемые операторы нет. Другое дело, что не всегда сортировка может иметь смысл. Например, если вставлять отсортированные данные в таблицу `tbPeoples`, то таким образом можно повлиять только на идентификатор добавляемых строк, потому что в таблице присутствует автоматически увеличиваемое поле, которое генерируется последовательно. Но если бы первичный ключ был GUID-полем, то сортировка потеряла бы смысл, потому что GUID-поле генерируется случайным образом.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\insert.sql.

2.17. Изменение данных

Очень редко встречаются таблицы и даже целые базы данных, в которых данные не меняются, а только добавляются. Поэтому теперь мы переходим к рассмотрению процесса изменения данных в таблице.

Для изменения данных в записях таблицы используется оператор `UPDATE`. Общий вид этого оператора приводится в листинге 2.6.

Листинг 2.6. Общий вид команды `UPDATE`

```
UPDATE
        {
         table_name WITH ( < table_hint_limited > [ ...n ] )
         | view_name
         | rowset_function_limited
        }
        SET
        { column_name = { expression | DEFAULT | NULL }
        | @variable = expression
        | @variable = column = expression } [ ,...n ]

    { { [ FROM { < table_source > } [ ,...n ] ]

        [ WHERE
```

```
        < search_condition > ] }
    |
    [ WHERE CURRENT OF
    { { [ GLOBAL ] cursor_name } | cursor_variable_name }
    ] }
    [ OPTION ( < query_hint > [ ,...n ] ) ]
```

Самый простейший вариант команды выглядит следующим образом:

```
UPDATE Таблица
SET Поле=Значение
```

В первой строчке после оператора `UPDATE` указывается имя таблицы, которую необходимо обновить. Далее идет оператор `SET`, а после него — имя поля и новое значение. При этом изменяются абсолютно все строки в таблице.

Давайте изменим дату рождения у всех записей в таблице `tbPeoples`:

```
UPDATE tbPeoples
SET dDateBirthDay='10.10.2004'
```

Можете запустить этот пример и обновить поле `dDateBirthDay` во всех строках, а можете поверить мне на слово, что пример работает, и не портить красоту данных. Просто сейчас незачем делать значения поля даты одинаковыми для всех работников.

Теперь рассмотрим другой пример изменения значения поля. Попробуем увеличить значение даты у всех строк на единицу. Я не зря выбрал дату. Если бы это было число, то оно увеличилось бы на 1, и это было бы вполне логично и понятно. А что произойдет с датой? Выполните запрос `SELECT` к таблице `tbPeoples` и запомните несколько значений. Теперь выполним следующий запрос на обновление:

```
UPDATE tbPeoples
SET dDateBirthDay = dDateBirthDay + 1
```

В данном примере мы устанавливаем полю `dDateBirthDay` значение `dDateBirthDay` плюс единица. Снова посмотрим содержимое таблицы `tbPeoples`. Обратите внимание, что все даты увеличились на один день.

Да, даты можно складывать с числами, потому что реально дата — это число. Более подробно о датах мы поговорим в *разд. 2.22*.

Из примера видно, что оператор `UPDATE` позволяет использовать математические вычисления, и это очень удобно.

Чтобы изменить только определенную строку или набор строк, нужно ограничить выборку с помощью секции `WHERE`, как мы это делали с оператором

SELECT. Например, следующий запрос изменяет дату рождения только для работника Иванова Ивана, для чего в секции WHERE заданы необходимые параметры поиска:

```
UPDATE tbPeoples
SET dDateBirthDay='1.1.2004'
WHERE vcFamil='ИВАНОВ' AND vcName='ИВАН'
```

Если вы выполните этот запрос в программе Query Analyzer, то увидите, что обновились две записи. Дело в том, что в нашей таблице два работника с таким именем. Вы должны учитывать эту особенность, когда используете оператор UPDATE, ведь он может обновить любое количество записей в таблице, если они удовлетворяют указанным условиям. Старайтесь четко указать, что и как вы хотите обновлять, поэтому при обновлении лучше использовать ключевое поле, которое может четко идентифицировать в таблице нужную строку.

Чтобы проще было писать запросы на обновление и избежать ошибок, я всегда начинаю с запроса SELECT. Например, для обновления даты рождения Иванова Ивана нужно сначала написать запрос на выборку:

```
SELECT *
FROM tbPeoples
WHERE vcFamil='ИВАНОВ' AND vcName='ИВАН'
```

Выполняем запрос и смотрим на результат. Если нас все устраивает, и мы увидели только те строки, которые должны измениться, то заменяем первые две строчки запроса SELECT на операторы UPDATE и SET.

Если вы заранее знаете, что нужно обновить только одну запись (например, для конкретного работника), то необходимо убедиться, что будет обновляться только одна строка. Как мы уже знаем, по фамилии искать бесполезно, бывают работники с одинаковыми фамилиями. Но у нас есть в поле первичный ключ. Именно он однозначно идентифицирует строку, и две записи таблицы не могут содержать в первичном ключе одно и то же значение:

```
UPDATE tbPeoples
SET dDateBirthDay='1.1.2004'
WHERE idPeoples=10
```

Тут очень важно создать механизм определения первичного ключа, но это уже ложится на программиста клиентских программ для конечного пользователя. Наша задача — SQL-запросы.

Обновление можно выполнять и на основе двух таблиц сразу. Например, допустим, что необходимо указать новый номер телефона для работника

по фамилии Кононов, и при этом тип телефона должен быть равен 1. Как написать такой запрос обновления? Чтобы вам проще было создавать такой запрос в будущем, давайте рассмотрим логику создания, которая опять же основывается на запросе выборки данных `SELECT`.

Для начала пишем запрос `SELECT` на выборку данных необходимого работника:

```
SELECT *
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pl.idPeoples=pn.idPeoples
 AND vcFamil='КОНОНОВ'
 AND idPhoneType=1
```

В этом запросе мы связываем таблицу работников с таблицей номеров телефонов и указываем необходимые условия выбора нужной строки по фамилии и типу телефона. Выполните этот запрос, и в результате у вас должна появиться только одна строка, которую как раз и надо обновить.

Теперь из запроса `SELECT` делаем `UPDATE`. Для этого первую строчку (оператор `SELECT`) заменяем на оператор `UPDATE` и указываем изменяемое поле:

```
UPDATE tbPhoneNumbers
SET vcPhoneNumber='(905) 100-10-10'
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pl.idPeoples=pn.idPeoples
 AND vcFamil='КОНОНОВ'
 AND idPhoneType=1
```

В первой строчке стоит оператор `UPDATE`, в котором указывается таблица, поле которой нужно обновить. Во второй строчке я указал оператор `SET`, и далее идет присвоение нужному полю нового значения. Остальные строки идентичны запросу `SELECT`. Точно также указывается секция `FROM`, устанавливается связь, и указаны дополнительные условия отбора нужной строки.

Если запрос `SELECT` написан верно, и в его результате есть изменяемое поле, то, изменив первую строку на `UPDATE` и оператор `SET`, вы верно напишете SQL-запрос на изменение нужной строки и нужного поля.

Несмотря на то, что выборка данных происходит из нескольких таблиц, изменяться может только одна — та, которая указана после ключевого слова `UPDATE`.

Данные для изменения можно брать из другой таблицы. Например, в следующем запросе изменяется поле `dDateBirthDay` у работника, который имеет идентификатор 1. При этом поле `dDateBirthDay` увеличивается на значение первичного ключа из таблицы номеров телефонов:

```
UPDATE tbPeoples
SET dDateBirthDay=dDateBirthDay+pn.idPhoneNumbers
```

```
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pl.idPeoples=pn.idPeoples
 AND pl.idPeoples=1
 AND idPhoneType=1
```

А как обновить сразу несколько полей? В этом случае в секции `SET` нужно перечислить операции изменения через запятую:

```
UPDATE tbPeoples
SET dDateBirthDay=dDateBirthDay+1,
   vcFamil='mr.'+vcFamil
```

Здесь после ключевого слова `SET` мы изменяем поле `dDateBithDay`, увеличивая его на 1, и поле `vcName`, добавляя в начало фамилии префикс `mr`.

Следующий пример изменяет содержимое двух полей `vcPhoneNumbers` и `vcName` в связанных таблицах:

```
UPDATE tbPhoneNumbers
SET vcPhoneNumber='(905) 100-10-10',
    vcName='Иван'
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pl.idPeoples=pn.idPeoples
 AND vcFamil='КОНОНОВ'
 AND idPhoneType=1
```

Количество изменяемых полей в одном запросе не ограничено, можно изменить хоть все поля. Но при этом вы должны учитывать, что нельзя изменять автоматически увеличиваемые поля. Мы не могли записать в такие поля значения при добавлении записей и не можем изменить их. Конечно же, мы можем насильно разрешить изменение поля, установив свойство `IDENTITY_INSERT` в `ON`:

```
SET IDENTITY_INSERT tbPeoples ON
```

Однако я не рекомендовал делать это при добавлении записей, а при редактировании не советую тем более. Нет, вы можете изменить поле, но тогда возрастет вероятность нарушения целостности. Если данный ключ используется для связи с записями другой таблицы, то связь просто нарушится.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\update.sql на компакт-диске, прилагаемом к книге.

2.18. Удаление данных

Теперь посмотрим, как можно удалять записи из таблицы. Для этого используется оператор DELETE. Общий вид оператора показан в листинге 2.7.

Листинг 2.7. Общий вид оператора удаления DELETE

```
DELETE
    [ FROM ]
        { table_name WITH ( < table_hint_limited > [ ...n ] )
         | view_name
         | rowset_function_limited
        }

        [ FROM { < table_source > } [ ,...n ] ]

    [ WHERE
        { < search_condition >
        | { [ CURRENT OF
                { { [ GLOBAL ] cursor_name }
                    | cursor_variable_name
                }
            ] }
        }
    ]
```

Минимальная команда удаления выглядит следующим образом:

```
DELETE tbPeoples
```

Эта команда удаляет все строки из таблицы tbPeoples. Те же самые действия можно выполнить с помощью вызова команды:

```
DELETE FROM tbPeoples
```

Отличие этой команды в том, что мы добавили ключевое слово FROM, которое в данной команде можно опускать.

Пока не выполняйте эти команды, потому что они очистят таблицу tbPeoples, а нам этого пока не нужно. Данные еще пригодятся для тестирования будущих запросов. Если хотите протестировать удаление, то создайте новую таблицу, заполните ее данными и очистите. Заодно вспомните и за-

крепите полученные ранее знания. Можно создать новую таблицу с помощью команды `SELECT INTO`, это будет проще и быстрее.

Я предпочитаю использовать вторую команду, хотя она выполняет те же самые действия, а содержит на одно слово больше. Просто второй вариант более понятен. В первом случае команда читается как "Удалить tbPeoples". Это не совсем корректно. Второй вариант команды звучит как "Удалить из tbPeoples". Это уже точнее отражает суть выполняемых действий.

Такая простота очистки всего содержимого таблицы, на мой взгляд, является недостатком. В моей практике уже не раз были случаи, когда удаление происходило случайно. Просто кто-то выделял в Query Analizyr не полную команду, и таблица очищалась сильнее, чем хотелось бы. Проблема кроется и в том, что в большинстве редакторов нет подтверждения при удалении. Я видел только одну программу, которая перед удалением отображала окно подтверждения и показывала, сколько строк будет удаляться, — phpMySQLAdmin. Это программа для управления через Web-интерфейс базой данных MySQL.

Чтобы ограничить количество удаляемых строк, используется секция `WHERE`. Она используется точно так же, как и в запросах `SELECT` или `UPDATE`. Когда мы тестировали оператор `INSERT`, то вставили в таблицу достаточно много записей. Давайте подчистим их и попрактикуемся в использовании оператора `DELETE`.

Для начала удалим все строки, в которых поле `vcFamil` содержит нулевое значение. Это делается с помощью следующего запроса:

```
DELETE FROM tbPeoples
WHERE vcFamil is NULL
```

С помощью секции `WHERE` мы требуем, чтобы сервер удалил только те записи, в которых поле `vcFamil` содержит нулевое значение.

Следующий запрос удаляет запись, в которой первичный ключ `idPeoples` содержит значение –22:

```
DELETE FROM tbPeoples
WHERE idPeoples = -22
```

Предыдущие два запроса можно было бы выполнить одной командой:

```
DELETE FROM tbPeoples
WHERE vcFamil is NULL
 OR idPeoples = -22
```

В данном случае удаляются записи, в которых или фамилия не заполнена, или ключевое поле равно –22.

Давайте теперь для закрепления материала создадим что-нибудь посложнее. Например, удалим все записи из таблицы `tbPeoples`, у которых нет записей в таблице телефонов `tbPhoneNumbers`. Для этого сначала пишем запрос `SELECT`, чтобы убедиться, что он выбирает необходимые данные:

```
SELECT *
FROM tbPeoples pl
WHERE NOT EXISTS
  (SELECT idPeoples
   FROM tbPhoneNumbers pn
   WHERE pl.idPeoples=pn.idPeoples)
```

Этот запрос выбирает все строки, для которых не существует (`NOT EXISTS`) записи в таблице `tbPhoneNumbers`. Отличный пример, ведь подобные запросы приходится писать достаточно часто.

Теперь модифицируем пример для того, чтобы он удалял выбранные данные. Для этого первую строку заменяем на `DELETE`:

```
DELETE
FROM tbPeoples
WHERE NOT EXISTS
  (SELECT idPeoples pl
   FROM tbPhoneNumbers pn
   WHERE pl.idPeoples=pn.idPeoples)
```

Но этого недостаточно, потому что при удалении в секции `FROM` не должно быть псевдонимов. Это значит, что от них нужно отказаться и в подзапросе использовать вместо псевдонима полное имя таблицы из внешнего запроса:

```
DELETE
FROM tbPeoples
WHERE NOT EXISTS
  (SELECT idPeoples
   FROM tbPhoneNumbers pn
   WHERE tbPeoples.idPeoples=pn.idPeoples)
```

Недавно меня один очень опытный программист спросил: "Можно ли использовать оператор `LIKE` при удалении?" А почему бы и нет? Можно использовать любые операторы, которые мы изучали для секции `WHERE`, и `LIKE` тут не исключение. Следующий запрос удалит все записи, в которых фамилия начинается с буквы "С":

```
DELETE
FROM tbPeoples
WHERE vcFamil LIKE 'C'
```

Теперь рассмотрим удаление из связанных таблиц. Допустим, что необходимо удалить номера телефонов определенного работника. Для этого выполняем следующий запрос:

```
DELETE pn
FROM tbPhoneNumbers pn, inserted i
WHERE pn.idPeoples=i.idPeoples
 AND vcFamil = Фамилия
```

Нельзя удалять данные сразу из двух таблиц. Так как в этом примере связываются две таблицы, то после оператора `DELETE` необходимо явно указать, из какой из двух происходит удаление.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\delete.sql на компакт-диске, прилагаемом к книге.

При удалении данных для каждой удаляемой строки в журнале транзакций создается отдельная запись. О транзакциях мы подробно поговорим в *разд. 2.19*, а сейчас нам необходимо отдавать себе отчет в том, что это достаточно трудоемкий процесс. Это можно понять, даже не вникая в суть самих транзакций. Журнал транзакций — это файл, и в него происходит запись для каждой удаляемой строки. Если в таблице 1 000 000 записей, то будет столько же обращений к жесткому диску, а если каждое обращение к диску отнимет хотя бы 10 байт данных, то мы потеряем на жестком диске 10 000 000 байт свободного пространства. Перспективы не очень радужные.

Но у MS SQL Server есть вариант решения этой проблемы — оператор `TRUNCATE TABLE`. Его можно использовать при удалении из таблицы всех записей, и при этом обращения к журналу будут сведены к минимуму. Для удаления всех записей из таблицы `Товары` нужно использовать следующий запрос:

```
TRUNCATE TABLE Имя_Таблицы
```

Теперь из таблицы будут удалены все записи, а в журнал попадет только информация о том, что все удалено.

С помощью оператора `TRUNCATE TABLE` можно удалять только все записи из таблицы, потому что нет секции `WHERE`. К тому же, нельзя удалять данные из таблицы, которая связана с другой внешним ключом.

Есть еще одно очень важное отличие между операторами `DELETE` и `TRUNCATE TABLE`. Если удалить все записи из таблицы с помощью `DELETE`, а затем добавить новую запись, то значение автоматически увеличиваемого поля будет увеличиваться дальше, т. е. счетчик не будет сброшен. А после удаления с помощью `TRUNCATE TABLE` значение счетчика сбрасывается, и если

после этого добавить строку, то значение автоматически увеличиваемого поля начнет свой отсчет с единицы. Учитывайте этот эффект при использовании определенного оператора удаления данных.

2.19. Транзакции

Язык запросов Transact-SQL получил свое название от слова "транзакция". Я думаю, что компания Microsoft не зря сконцентрировала на этом понятии особое внимание, ведь транзакции действительно являются очень мощным средством управления базой данных.

Прежде чем мы рассмотрим примеры, немного погрузимся в теорию, чтобы вы смогли лучше понять, что такое транзакции. Почему-то для многих этот вопрос является наиболее сложным, но все оказывается достаточно простым, если правильно подобрать пример и наглядно показать его работу.

Рассмотрим классическую задачу — банковскую операцию. Допустим, что у нас есть таблица из двух полей — номер счета в банке и сумма денег на этом счете. Нам необходимо перевести деньги с одного счета на другой. Для этого нужно выполнить запрос `UPDATE`, чтобы уменьшить первый счет на нужную сумму. После этого выполняем `UPDATE`, чтобы увеличить значение второго счета. Все вроде бы нормально. А если после уменьшения первого счета произойдет сбой электропитания, и сервер не успеет пополнить другой счет? Деньги уже сняты, но никуда не записаны, а значит, они пропали.

Если сначала пополнять счет, а потом снимать деньги, а снятие не успеет произойти, то банк может оказаться банкротом, ведь появится лишняя сумма, снятие не происходит.

Некоторые могут сказать, что сбои в подаче электроэнергии и работе компьютера происходят редко. Согласен, но вполне вероятна блокировка записей. Один пользователь может заблокировать запись для изменения, и это не позволит запросу изменить ее. В таком случае снятие денег или пополнение счета не произойдет.

Проблему решает транзакция. Перед выполнением операций обновления необходимо явно начать транзакцию. После этого выполняем две операции `UPDATE` и по их окончании завершаем транзакцию. Если в момент выполнения одного из запросов происходит сбой, то все изменения, происшедшие после начала транзакции, отменяются.

Существует две разновидности транзакций в SQL Server:

- скрытые транзакции, каждый оператор, такой как `INSERT`, `UPDATE` или `DELETE`, выполняется в транзакции. Неявными транзакциями можно управлять, и об этом мы поговорим в *разд. 4.1.2*;

- явные транзакции, объявленные пользователем, — операторы, сгруппированные в `BEGIN TRANSACTION` и `COMMIT TRANSACTION`.

Очень важно понимать, что транзакции необходимы только при модификации данных, т. е. при использовании операторов `INSERT`, `UPDATE` или `DELETE`. Простая выборка `SELECT` не изменяет данных. Конечно, выполнять операции выборки в транзакции можно, но если транзакция не изменяет данные, то незачем ее вообще начинать.

В транзакции можно включать далеко не все операторы. Внутри транзакции не может быть следующих операторов:

- `ALTER DATABASE`
- `BUCKUP LOG`
- `CREATE DATABASE`
- `DROP DATABASE`
- `RECONFIGURE`
- `RESTORE DATABASE`
- `RESTORE LOG`
- `UPDATE STATISTICS`

Работа транзакций обеспечивается с помощью журнала транзакций базы данных. Так как журнал записывает всю активность и происходящие изменения, SQL Server может восстанавливать данные автоматически в момент потери питания, ошибки системы, проблемы клиента или запроса отмены журнала.

Сервер SQL автоматически гарантирует, что все завершенные транзакции отражаются в базе данных в момент и после системной ошибки. Для этого используется журнал транзакций, с помощью которого завершаются удачно законченные транзакции и отменяются незавершенные.

С помощью скрытых транзакций сервер гарантирует, что если оператор добавления, изменения или удаления данных выполнен удачно, то данные будут сохранены в таблице. Если во время изменений произошла ошибка, то все изменения откатываются. Представим, что оператор `UPDATE` изменяет 1000 строк. Если на 500-й строке произошла ошибка, то сервер откатывает все уже сделанные изменения, как если бы они происходили в явной транзакции.

Пользователь может устанавливать точку сохранения или маркер внутри транзакции. Точка сохранения определяет расположение, до которого транзакция может быть возвращена. Транзакция должна продолжить выполнение или должна быть полностью откатана.

Начало транзакции в MS SQL Server имеет следующий синтаксис:

```
BEGIN TRAN[SACTION]
 [transaction name | @transaction name variable
 [WITH MARK['description]]]
```

Опция `Transaction name` указывает имя транзакции, определенное пользователем. Опция `WITH MARK` указывает, что транзакция маркирована в журнале транзакций.

По завершению транзакции изменения необходимо запомнить в базе данных. Для этого используется команда `COMMIT`:

```
COMMIT [ TRAN [ SACTION ]
[ transaction_name | @tran_name_variable ] ]
```

Отмена транзакции и всех изменений производится командой `ROLLBACK`, которая в общем виде выглядит следующим образом:

```
ROLLBACK [ TRAN [ SACTION ]
    [ transaction_name | @tran_name_variable
    | savepoint_name | @savepoint_variable ] ]
```

Очень важно понимать, если начата транзакция и изменены какие-то записи, то эти записи блокируются, пока транзакция не будет завершена. Давайте посмотрим это на примере, заодно познакомимся с самой командой. Выполните следующие команды в Query Analyzer:

```
-- Начинаем транзакцию
BEGIN TRANSACTION


-- Очищаем таблицу товаров
DELETE Товары
```

Теперь откройте еще одну копию программы или установите новое соединение, выбрав меню **File | Connect** (Файл | Соединиться). В новом окне напишем и выполним следующий запрос:

```
SELECT * FROM Товары
```

Таблица товаров довольно маленькая, но несмотря на это запрос будет выполняться долго. Точнее, он не выполнится вовсе, потому что во втором окне с другим сеансом выполняется транзакция удаления, и эта транзакция еще не завершилась.

Нам удалять данные не нужно, поэтому вернемся в первое окно, где мы создавали транзакции, и выполним откат:

```
ROLLBACK TRANSACTION
```

Вот теперь запрос завершит свое выполнение, и при этом все данные останутся на месте, несмотря на то, что мы выполняли команду удаления всех записей. Данные не удалились, потому что команда выполнялась внутри транзакции, которую мы откатили.

Теперь проведем еще один эксперимент. Снова начинаем транзакцию и удаляем все записи. Теперь попробуйте выполнить запрос на выборку данных из этого же окна. Запрос завершится моментально, и в результате будет пустая выборка (ни одной строки). Получается, что для нашего сеанса, который удалял строки, таблица является пустой.

Теперь откатываем транзакцию, выполняя команду `ROLLBACK TRANSACTION`. Снова выполняем запрос `SELECT` и видим, что данные вернулись на место. Транзакция удачно отклонена и физического удаления из базы данных не произошло. Почему мы в этот раз без проблем смогли просмотреть таблицу, а из другого сеанса просмотр изменяемой в транзакции таблицы не доступен? Блокировка происходит для всех сеансов кроме того, который выполняет транзакцию. В листинге 2.8 показан весь код эксперимента с подробными комментариями.

Листинг 2.8. Пример эксперимента с удалением данных в транзакции

```
-- Начинаем транзакцию
BEGIN TRANSACTION

-- Очищаем таблицу товаров
DELETE Товары

-- Проверяем и убеждаемся, что таблица пуста
SELECT * FROM Товары

-- Откатываем транзакцию
ROLLBACK TRANSACTION

-- Можете убедиться, что товары на месте
SELECT * FROM Товары
```

Все эти команды нужно выполнять в одном и том же окне. К тому же, если в одном окне (сеансе) вы начали транзакцию, то именно в этом окне вы должны ее завершить (`COMMIT`) или откатить (`ROLLBACK`).

Если в листинге 2.8 заменить вызов команды `ROLLBACK TRANSACTION` на `COMMIT TRANSACTION`, то произойдет физическое удаление всех записей из таблицы товаров. Теперь удаленные строки вернуть уже невозможно.

Рассмотрим еще один пример (листинг 2.9).

Листинг 2.9. Пример работы с транзакциями

```
-- Начинаем транзакцию
BEGIN TRANSACTION

-- Вставляем строку данных в
-- таблицу товаров
INSERT INTO Товары (Дата, [Название товара], Цена, Количество)
VALUES ('3.3.2005', 'КАРТОФЕЛЬ', 12.50, 10)

-- Обновить данные в последней строке
UPDATE Товары
SET Цена = 15
WHERE [Название товара] LIKE 'КАРТОФЕЛЬ'

COMMIT TRANSACTION

-- Обновить данные в последней строке
UPDATE Товары
SET Цена = 17
WHERE [Название товара] LIKE 'КАРТОФЕЛЬ'

-- Откатить транзакцию
ROLLBACK TRANSACTION

-- Выбрать все данные из таблицы
SELECT * FROM Товары
```

Тут достаточно много действий, поэтому давайте их рассмотрим поэтапно:

1. Начинаем транзакцию.
2. Добавляем запись о покупке товара с названием `Картофель`.
3. Обновляем цену картофеля, увеличив ее до 15 рублей.
4. Завершаем транзакцию, запоминая изменения.

5. Обновляем цену до 17 рублей.
6. Откатываем транзакцию.
7. Просматриваем содержимое таблицы.

Что произошло с содержимым таблицы? Запись о картофеле добавлена, а значит, все что было до запоминания изменений (шаг 4) выполнено удачно. А вот цена равна 17 рублям. Почему? Неужели на шаге 6 мы не откатили изменение цены? Да, отката не произошло, потому что новая транзакция не начиналась. На шаге 1 мы начали транзакцию, а на шаге 4 завершили. Новая транзакция не начиналась, а значит, откатывать нечего, и шаг 6 завершится ошибкой:

The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

Запрос ROLLBACK TRANSACTION не имеет соответствующего BEGIN TRANSACTION.

Мы говорили, что во время добавления, изменения и удаления записей автоматически и неявно начинается новая транзакция. Ее откатить мы не можем, и откатом неявных транзакций занимается только сам сервер.

Для каждого оператора `BEGIN TRANSACTION` должен быть только один оператор `COMMIT TRANSACTION` или `ROLLBACK TRANSACTION`.

А что если внутри транзакции начать новую транзакцию? Результат получится неожиданный. Рассмотрим пример (листинг 2.10).

Листинг 2.10. Вложенные транзакции

```
-- Начинаем транзакцию 1
BEGIN TRANSACTION

-- Вставляем строку данных в таблицу товаров
INSERT INTO Товары (Дата, [Название товара], Цена, Количество)
VALUES ('4.3.2005', 'МОРКОВЬ', 11.30, 1)

-- Начинаем транзакцию 2
BEGIN TRANSACTION

-- Обновить данные в последней строке
UPDATE Товары
SET Цена = 14
```

```
WHERE [Название товара] LIKE 'МОРКОВЬ'

ROLLBACK TRANSACTION

-- Запоминаем изменения
COMMIT TRANSACTION

-- Выбрать все данные из таблицы
SELECT * FROM Товары
```

Логика запроса следующая:

1. Начать транзакцию.
2. Вставить строку.
3. Начать транзакцию.
4. Обновить таблицу.
5. Откатить транзакцию.
6. Запомнить изменения.

По логике вещей, на шаге 5 мы должны были откатить вторую транзакцию (т. е. изменение таблицы), а на шаге 6 запоминаем транзакцию 1, в которой происходит добавление записи. Посмотрите содержимое таблицы. Ни добавления, ни тем более изменения. Почему? Если посмотреть сообщения, которые выдал сервер, то вы увидите, что на шаге 6 произошла ошибка, потому что нет соответствующего начала транзакции и серверу нечего начинать. Получается, что оператор `ROLLBACK TRANSACTION` откатывает все начатые транзакции.

Но это не значит, что невозможно использовать вложенные транзакции. Просто откатывать транзакции нельзя на один шаг назад. Если заменить оператор `ROLLBACK TRANSACTION` на `COMMIT`, то ошибки не будет.

Посмотрим на листинг 2.11. В нем показан такой же пример, но с именованными транзакциями и без отката.

Листинг 2.11. Использование вложенных транзакций

```
BEGIN TRANSACTION T1

-- Вставляем строку данных в таблицу товаров
INSERT INTO Товары (Дата, [Название товара], Цена, Количество)
```

```
VALUES ('4.3.2005', 'МОРКОВЬ', 11.30, 1)

-- Начинаем транзакцию 2
BEGIN TRANSACTION T2

-- Обновить данные в последней строке
UPDATE Товары
SET Цена = 14
WHERE [Название товара] LIKE 'МОРКОВЬ'

COMMIT TRANSACTION T2

-- Запоминаем изменения
COMMIT TRANSACTION T1

-- Выбрать все данные из таблицы
SELECT * FROM Товары
```

В данном примере после операторов `BEGIN TRANSACTION` и `COMMIT TRANSACTION` указывается имя `T1` или `T2`. Таким образом, мы идентифицируем транзакции и завершаем их в порядке, обратном объявлению.

Но как же тогда можно откатывать транзакции до определенной точки? Достаточно просто, и вложенные транзакции тут ни при чем. Для этого нужно просто поставить точку сохранения с помощью оператора `SAVE TRAN`, который имеет следующий вид:

```
SAVE TRAN [ SACTION ] { savepoint_name | @savepoint_variable }
```

Минимум, что необходимо указать, — это сам оператор и имя точки сохранения. Например, следующий оператор создает точку сохранения с именем `point1`:

```
SAVE TRAN point1
```

В листинге 2.12 показан пример, в котором наконец-то создаем строку, изменяем ее и откатываем только изменение, а не всю транзакцию, вместе с добавлением строки.

Листинг 2.12. Откат до определенной точки

```
-- Начинаем транзакцию 1
BEGIN TRANSACTION T1

-- Вставляем строку данных в
```

```
-- таблицу товаров
INSERT INTO Товары (Дата, [Название товара], Цена, Количество)
VALUES ('4.3.2005', 'МОРКОВЬ', 11.30, 1)

-- Сохраняем транзакцию
SAVE TRAN ins_complete

-- Обновить данные в последней строке
UPDATE Товары
SET Цена = 14
WHERE [Название товара] LIKE 'МОРКОВЬ'

-- Откатываем транзакцию
ROLLBACK TRANSACTION ins_complete

-- Запоминаем изменения
COMMIT TRANSACTION T1

-- Выбрать все данные из таблицы
SELECT * FROM Товары
```

Снова разобьем логику выполнения этого сценария на шаги, чтобы лучше понять происходящее:

1. Начинаем транзакцию.
2. Добавляем строку.
3. С помощью оператора `SAVE TRAN` сохраняем состояние таблицы. Точнее сказать, ставим точку в журнале, ведь пока все изменения происходят только в журнале транзакций.
4. Обновляем цену последней добавленной строки.
5. Восстанавливаем состояние таблицы на точку сохранения, установленную на шаге 3. В этот момент из журнала транзакций удаляется запись о необходимости обновить цену, а остается только запись о необходимости добавить строку.
6. Запоминаем изменения. В журнале транзакций находится только добавление строки (обновление цены мы откатили на шаге 5), поэтому только новая запись сохраняется в таблице товаров.

На последнем этапе выбираются данные из таблицы товаров. Убедитесь, что в добавленной строке присутствуют только те данные, которые были указаны при вставке таблицы.

Так как транзакции блокируют записи, я рекомендую делать их максимально быстрыми и не советую вам использовать вложенные транзакции. Долгие транзакции увеличивают вероятность того, что пользователи не получат доступ к заблокированным данным.

Следующие рекомендации позволят вам сделать транзакции как можно короче:

- без особой надобности не выполняйте запросы `SELECT`. Выборка данных должна происходить вне транзакции;
- будьте внимательны, когда используете долго выполняемые операторы Transact-SQL, такие как циклы и создание объектов базы данных;
- не требуйте от пользователя ввода данных во время выполнения транзакции. Делайте ввод данных до начала выполнения транзакции;
- операторы `INSERT`, `UPDATE` и `DELETE` должны быть главными в транзакции, и их следует писать так, чтобы получить минимальный набор строк и повысить скорость работы запроса;
- все проверки данных и подготовительные расчеты необходимо произвести до начала транзакции. После оператора `BEGIN TRANSACTION` должно выполняться только изменение данных.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\transaction.sql на компакт-диске, прилагаемом к книге.

2.20. Переменные

Язык запросов Transact-SQL отличается не только мощной поддержкой транзакций, но и поддержкой переменных, благодаря которым становится возможным создание более сложных запросов и даже целых программ. Переменные есть и в других вариантах языка SQL (в том числе и PL/SQL), так что они не являются особым достоинством MS SQL Server.

Переменные — элементы языка с присвоенными значениями. Если честно, то такое определение мне не очень нравится, потому что не отражает суть переменных и абсолютно непонятно, что это и для чего нужно. Чтобы понять, что такое переменная, нужно вспомнить, как данные хранятся в памяти.

Когда вы заносите в компьютер какое-то число или строку, то в оперативной памяти для хранения этих данных выделяется участок памяти. Этот участок имеет числовой адрес, а как известно, числа запоминать намного сложнее. Лично я могу помнить не более 10 номеров телефонов, а все остальное записываю в блокнот. Понятные слова запоминаются лучше и с ними проще

работать. Чтобы обратиться к данным в памяти, необходимо указать адрес, по которому находятся нужное нам число или строка.

Чтобы не мучаться с числами, придумали переменные, которые по своей сути являются именами участков в памяти, где находятся данные.

Вы объявляете локальные переменные в операторе `DECLARE` и после этого назначаете ему индивидуальное значение с помощью операторов `SET` или `SELECT`. Используйте оператор `SET`, когда желаемое значение известно, а оператор `SELECT`, когда вам нужно получить желаемое значение из таблицы. После того, как вы назначили переменной значение, вы можете использовать его в операторах, группах и процедурах, в которых они объявлены. Группы — это набор операторов Transact-SQL, которые отправляются вместе и исполняются как группа (несколько последовательно выполняемых команд). Мы не раз уже использовали группы, и вы их можете увидеть в листингах 2.9, 2.10 и т. д.

Имя локальной переменной начинается с одного символа `@` в начале имени. Общий вид объявления переменной:

```
DECLARE {@local_variable datatype} [,..n]
```

Пример установки значения переменной:

```
SET @имя = значение
```

После ключевого слова указывается локальная переменная (`@имя`). Далее идет знак равенства и значение, которое должно быть присвоено переменной.

Следующий пример объявляет две переменные и использует их (листинг 2.13).

Листинг 2.13. Пример использования переменных

```
-- Объявляем две переменные @vLastName и @vFirstName,
-- которые имеют тип varchar размером в 50 символов
DECLARE @vLastName   varchar(50),
        @vFirstName  varchar(50)

-- Устанавливаем переменную фамилии @vLastName
SET @vLastName='%СИДОРОВ'

-- Выбираем имя работника из таблицы и записываем его в
-- переменную @vFirstName
SELECT @vFirstName = vcName
```

```
FROM tbPeoples
WHERE vcFamil LIKE @vLastName

-- Печатаем результат
PRINT 'Имя Сидорова '+@vFirstName
```

В результате на экране мы должны увидеть сообщение:

```
Имя Сидорова ИВАН
```

Давайте посмотрим, что происходит в сценарии. Сначала мы объявляем две переменные. Это можно сделать двумя способами. Первый — после ключевого слова `DECLARE` перечислить через запятую имена переменных и их тип, как показано в листинге 2.13. Напоминаю, что имя переменной должно начинаться с символа `@`.

Второй способ — каждую переменную описать в отдельной секции `DECLARE`:

```
DECLARE @vLastName   varchar(50)
DECLARE @vFirstName  varchar(50)
```

Теперь переменной `@vLastName`, которая должна хранить фамилию работника, присваиваем значение `'%СИДОРОВ'`. В начале фамилии стоит знак %, потому что в конце *разд. 2.17* мы добавили в начало каждой фамилии буквы `mr`. Если вы выполняли этот запрос, то вам необходимо поставить знак процента или присвоить переменной значение `'mr.СИДОРОВ'`.

Затем выполняем запрос `SELECT`. Давайте рассмотрим его построчно.

После оператора `SELECT` указываем, что необходимо значение поля `vcName` присвоить переменной `@vFirstName`:

```
SELECT @vFirstName = vcName
```

Данные выбираются из таблицы `tbPeoples`:

```
FROM tbPeoples
```

В условии поиска мы указываем, что поле `vcFamil` должно соответствовать содержимому шаблона в переменной `@vLastName`:

```
WHERE vcFamil LIKE @vLastName
```

Новой для нас является и последняя строчка, в которой происходит вывод на экран сообщения с помощью оператора `PRINT`. После оператора указывается сообщение, которое необходимо вывести. Сообщение может быть указано через пробел или в скобках. Следующий пример показывает, как использовать сообщение в скобках:

```
print('Команда недопустима')
```

Содержимое переменной можно использовать и в качестве имени транзакции. В следующем примере мы объявляем переменную `@Transname`, присваиваем ей значение и используем в качестве имени транзакции:

```
DECLARE @Transname   varchar(50)

BEGIN TRANSACTION @Transname

INSERT INTO tbPeoples(vcFamil, vcName)
VALUES('ВАСИЛЬКОВ', 'ИВАН')

COMMIT TRANSACTION @Transname
```

Глобальные переменные начинаются с двух символов `@@`. Когда вы создаете собственные программы на языке Transact-SQL, то используйте только локальные переменные, чтобы случайно не спровоцировать конфликта имен или значений. Так как глобальные переменные доступны всем, то их может увидеть любой пользователь, подключенный в данный момент к серверу.

И все же, глобальные переменные нужны. Существует несколько глобальных переменных, с помощью которых можно получить полезную информацию о сервере или о времени выполнения. Давайте бегло рассмотрим некоторые из них:

- `@@SERVERNAME` — содержит имя экземпляра MS SQL Server;
- `@@VERSION` — информация о версии SQL Server;
- `@@ROWCOUNT` — содержит количество строк, обработанных последней выполненной командой;
- `@@SPID` — идентификатор текущего процесса;
- `@@TRANCOUNT` — уровень вложенности транзакций;
- `@@NESTLEVEL` — уровень вложенности при выполнении хранимых процедур и триггеров;
- `@@OPTIONS` — текущие настройки серверных параметров;
- `@@ERROR` — содержит код ошибки последней выполненной команды;
- `@@IDENTITY` — последнее значение счетчика, которое было использовано после добавления строки;
- `@@CURSOR_ROWS` — возвращает количество записей, полученных последней командой открытия курсора.

В следующих главах мы будем возвращаться к глобальным переменным и некоторые из них рассмотрим более подробно. Обсуждать все переменные

сейчас не имеет смысла, потому что у нас еще недостаточно информации о языке Transact-SQL и сервере баз данных MS SQL Server. Пока мы ограничимся только общими понятиями и простым примером.

Рассмотрим определение имени установленного экземпляра MS SQL Server. Для этого необходимо выполнить следующую команду:

```
SELECT @@SERVERNAME
```

На моем компьютере в качестве результата была выведена строка с единственным безымянным полем, в котором находился текст: `CYD\FLENOV`.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\variable.sql на компакт-диске, прилагаемом к книге.

2.21. Конвертирование типов

Язык SQL является интерпретируемым. В отличие от компилируемого языка, где программа с помощью компилятора переводится в машинный код, программа на интерпретируемом языке остается в виде своего исходного кода, который и выполняется сервером. Когда мы запрашиваем выполнение SQL-сценария, сервер интерпретирует SQL-команды и производит необходимые действия.

Большинство интерпретируемых языков не типизированы и автоматически конвертируют данные из одного формата в другой. Язык Transact-SQL привязан к типам, и вы должны явно преобразовывать форматы там, где это необходимо.

Для преобразования данных используются команды (функции) `CAST` и `CONVERT`. В общем виде `CAST` выглядит следующим образом:

```
CAST ( expression AS data_type )
```

Рассмотрим классический пример. Допустим, необходимо сложить два числовых поля. Для этого возьмем из таблицы `tbPeoples` поля `idPeoples` и `idPosition`. Попробуем выполнить следующий запрос:

```
SELECT idPeoples+idPosition
FROM tbPeoples
```

Мы получим таблицу с результатами арифметического сложения этих полей. А если нужно сложить эти колонки как строки? То есть '1'+'1' в результате должно дать '11', а не арифметическую сумму (число 2). Как решить эту проблему? Нужно просто привести значения этих колонок к строке:

```
SELECT cast(idPeoples as varchar(10)) +
   cast(idPosition as varchar(10))
FROM tbPeoples
```

Что здесь происходит? Каждое поле мы приводим к строковому типу `varchar` размером в 10 символов. Например, первое поле преобразуется к строке следующим образом:

```
cast(idPeoples as varchar(10))
```

После команды `cast` в круглых скобках сначала указываем имя поля, которое нужно привести, а после ключевого слова `as` необходимо указать новый тип для поля. Если это строка, то необходимо указать и ее размер.

Открою небольшой секрет. Автоматическое преобразование в SQL-сервере все же есть. Как вы думаете, что получится, если выполнить следующий запрос:

```
SELECT cast(idPeoples as varchar(10)) + idPosition
FROM tbPeoples
```

В этом примере первое поле приводится к строке, а второе остается числом. Что будет результатом сложения строки с числом? Если вы попробуете выполнить этот запрос, то увидите, что результатом будет арифметическое сложение двух чисел. Несмотря на то, что первую колонку мы привели к строке, во время выполнения запроса SQL-сервер автоматически вернул этому полю числовой тип и сложил колонки как числа. Таким образом, автоматическое преобразование есть, но автоматически можно привести далеко не все типы.

Я рекомендую вам не надеяться на автоматизм сервера и интерпретатора, а указывать необходимые типы явно. Я уже видел достаточно много запросов, в которых ошибка крылась именно в разном понимании преобразования. Программист считает, что сервер должен выбрать один тип, а сервер выбирает другой, тогда результат запроса получается не таким, как ожидает программист.

Во время преобразования нужно быть внимательным и при указании типа и размера. Например, следующий запрос приводит слово `'Привет'` к типу `char` размером в 10 символов. Для наглядности я прибавляю к результату содержимое поля `vcFamil`, где у нас хранится фамилия работника:

```
SELECT CAST('Привет' AS char(10)) + vcFamil
FROM tbPeoples
```

Результат может быть неожиданным. Например:

```
Привет    mr.ИВАНОВ
```

Обратите внимание, что между словом `Привет` и содержимым поля несколько пробелов, а ведь мы их не указывали. Это связано с правилами использования типа `char`. Дело в том, что переменные или поля этого типа полностью занимают отведенное пространство. Если переменная требует меньшее про-

странство (как слово Привет требует только 6 символов), то оставшееся пространство заполняется пробелами. Преобразуемое слово занимает 6 символов, а мы его приводим к типу char из 10 символов. Это значит, что после слова Привет в конец строки будет добавлено 4 пробела, чтобы строка занимала все отведенное пространство.

Теперь посмотрим на команду CONVERT, которая имеет чуть больше возможностей. Общий вид команды:

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

Более мощные возможности проявляются при конвертировании дат, но об этом мы поговорим в следующем разделе. При преобразовании чисел и строк принцип работы схожий. В следующем примере целочисленное поле приводится к дробному с пятью символами после запятой:

```
SELECT CONVERT(decimal(10,5), idPeoples)
FROM tbPeoples
```

С помощью команд CAST и CONVERT можно округлять числа. Например, в таблице товаров у нас есть поле для хранения цены, которое является дробным. Давайте округлим цену до рублей, отбросив все, что находится после запятой, т. е. копейки:

```
-- Округление с помощью CONVERT
SELECT CONVERT(decimal(10,0), Цена)
FROM Товары

-- Округление с помощью CAST
SELECT CAST(Цена AS decimal(10,0))
FROM Товары
```

Можно было бы привести и к целочисленному типу, но я решил использовать тип decimal, но явно указать, что после запятой должно быть 0 разрядов. В следующем примере после запятой остается только 1 разряд:

```
SELECT CONVERT(decimal(10,1), Цена)
FROM Товары
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\convert.sql на компакт-диске, прилагаемом к книге.

2.22. Работа с датами и временем

Если вы заглянете в *приложение 1*, то увидите, что тип `datetime` может принимать значения от 1 января 1753 года до 31 декабря 9999 года. Если посмотреть на дату 1 января 2005 года, окажется, что это число с плавающей точкой, обозначающее отрезок времени с точностью до 3.33 миллисекунд. Целая часть — дата, а дробная часть — время. В *разд. 2.17* мы уже использовали свойство даты быть числом, когда увеличивали ее на один день. В этом разделе мы поговорим о датах более подробно.

2.22.1. Преобразование дат

Для начала посмотрим, как можно преобразовывать дату, раз уж в предыдущем разделе мы затронули эту тему.

В *приложении 1* в описании типа `datetime` сказано, что SQL-сервер использует формат даты, при котором вначале идет месяц (мм/дд/гггг). Если все оставить так, как есть, то пользователям будет неудобно вводить данные в таблицы, ведь мы привыкли начинать ввод даты с числа, а потом уже указывать месяц.

Для преобразования дат лучше всего использовать функцию `CONVERT`. Самый простейший пример преобразования с помощью этой функции:

```
SELECT convert(datetime, '31.1.2005' ,103)
```

Как мы уже знаем, в скобках указываются новый тип и переменная. Но здесь три параметра. Третий параметр — это стиль, в котором представлена дата. Если посмотреть на дату `'31.1.2005'`, то будет понятно, что вначале идет число, ведь 31-го месяца не существует. Если бы команда выглядела так: `SELECT convert(datetime, '31.1.2005')`, то сервер выдал бы нам сообщение об ошибке, потому что не смог бы преобразовать дату.

Последний параметр как раз указывает номер стиля, в котором мы представили дату, и благодаря ему сервер правильно прочитает параметр `'31.1.2005'`. Наиболее часто используемые стили перечислены в табл. 2.1. Более полный вариант таблицы можно увидеть в файле помощи. В первой колонке содержится номер стиля для случая, когда год задан в сокращенном виде (без указания века, например, 05). Во второй колонке — номер стиля, если год указан полностью (например, 2005).

Таблица 2.1. *Стили преобразования типа данных даты*

Сокращенный год	Полный год	Формат даты
-	0 или 100	Месяц дд гггг чч:ммАМ (или РМ)
1	101	мм/дд/гггг
2	102	гг.мм.дд
3	103	дд/мм/гггг
4	104	дд.мм.гггг
5	105	дд-мм-гггг
6	106	дд месяц гггг
7	107	месяц дд гггг
8	108	чч:мм:сс
10	110	мм-дд-гггг
11	111	гг/мм/дд
12	112	Ггммдд

Стиль 103 соответствует принятому в России формату дд/мм/гггг, и именно его мы использовали.

В следующем запросе мы используем преобразование даты в операторе `INSERT` для вставки данных в таблицу:

```
INSERT INTO tbPeoples
 (vcFamil, vcName, vcSurname, idPosition, dDateBirthDay)
VALUES('СОДОРОЧКИН', 'ИВАН', 'СЕРГЕЕВИЧ', 11,
  convert(datetime, '31.1.2005' ,103))
```

Есть еще один способ указания формата даты — с помощью оператора `SET`. Этот оператор мы будем достаточно подробно обсуждать в *разд. 4.1*, но то, что касается преобразования даты, рассмотрим прямо сейчас. Взгляните на следующий пример:

```
SET DATEFORMAT dmy
GO
DECLARE @vdate datetime
SET @vdate = '29/11/00'
SELECT @vdate
```

В первой строчке после `SET DATEFORMAT` указывается формат даты в виде сочетания букв `d`, `m` и `y`, которые отражают необходимую нам последовательность. Здесь мы будем указывать дату так: число, месяц, год.

Теперь объявляется переменная `vdate`, которой присваивается дата в заданном формате, и эта дата выводится на экран.

Необходимо отметить, что после однократного выполнения команды `SET` установленное значение действует в течение всего сеанса. Но если отключиться и подключиться к серверу заново, сервер снова будет требовать ввода даты в формате по умолчанию.

2.22.2. Функции для работы с датами

Есть еще множество функций, встроенных в SQL-сервер, с помощью которых можно работать с типом даты. В табл. 2.2 вы можете увидеть функции, которые есть у MS SQL Server для работы с датами, и краткое описание. Далее мы рассмотрим примеры работы с этими функциями.

Таблица 2.2. *Функции MS SQL Server по работе с датами*

Название функции	Описание
`GETDATE`	Определение текущей даты
`DATEADD`	Удобная функция увеличения значения даты
`DATEDIFF`	Определение разницы между двумя датами
`DATENAME`	Отображение имени части даты
`DATEPART`	Определение определенной части даты
`DAY`	Возвращает число, отражающее дни в дате
`MONTH`	Возвращает число, отражающее месяц в дате
`YEAR`	Возвращает число, отражающее год в дате

Текущая дата

Наиболее часто используемая функция — `GETDATE()`, которая возвращает текущую дату. Если в таблице необходимо сохранять текущую дату, то лучше всего будет определять ее с помощью функции сервера, а не с помощью каких-либо сторонних средств (например, с помощью клиентской программы) на компьютере клиента. Почему? Во-первых, если в вашей сети компьютеры не настроены на синхронизацию времени, то часы на всех компьютерах могут идти по-разному. Во-вторых, за работу часов при выключенном питании в компьютере отвечает маленькая батарейка, которая имеет тенденцию разряжаться, и тогда часы начинают серьезно отставать.

Если время добавления записи получено от клиента, то нельзя гарантировать, что в таблице даты отражают реальную действительность, потому что появляется зависимость от правильности хода часов на всех компьютерах. Если дата устанавливается с помощью `GETDATE()`, то для всех клиентов при добавлении записи дата будет устанавливаться по часам сервера.

Я рекомендую для полей, которые должны отражать дату создания записи, еще при создании таблиц установить значение по умолчанию с помощью функции `GETDATE()`. Например:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1,
 dDate datetime DEFAULT (getdate()),
 vcName varchar(50) DEFAULT 'M'
)
```

Теперь при добавлении записей нет необходимости указывать дату, потому что текущее значение будет использоваться по умолчанию.

Если вы не указали значение по умолчанию для поля типа `datetime`, то используйте функцию `GETDATE()` в запросе:

```
INSERT INTO Товары
 (Дата, [Название товара], Цена, Количество)
VALUES(GETDATE(), 'Сок', 23, 1)
```

Для добавляемой строки будет установлена текущая дата по часам сервера, а не клиента, потому что именно сервер разбирает и выполняет запрос.

Увеличение даты

Для увеличения даты используется функция `DATEADD()`. В общем виде она выглядит следующим образом:

```
DATEADD (datepart, number, date)
```

В скобках указывается три параметра:

- `datepart` — какую часть даты необходимо увеличить. Возможные значения этого параметра показаны в табл. 2.3;
- `number` — число, на которое надо увеличить дату;
- `date` — дата, которую надо изменить.

Таблица 2.3. *Возможные значения параметра* `datepart`

Значение параметра	Описание
`Year`	Год
`Quarter`	Квартал
`Month`	Месяц
`Dayofyear`	День года
`Day`	День
`Week`	Неделя
`Hour`	Час
`Minute`	Минута
`Second`	Секунда
`Millisecond`	Миллисекунда

Допустим, что необходимо увеличить дату на 1 год. Для этого можно выполнить следующий запрос:

```
SELECT DATEADD(Year, 1, GETDATE())
```

Чтобы уменьшить значение даты, необходимо в качестве второго параметра указать отрицательное значение. Например, следующий пример уменьшает значение текущей даты на 1:

```
SELECT DATEADD(Year, -1, GETDATE())
```

Разница между двумя датами

Нередко необходимо узнать разницу между двумя датами. Например, чтобы пользователь не указал слишком большой диапазон, можно проверять, чтобы разница между двумя датами не превышала год, иначе запрос будет возвращать слишком много данных. Пусть пользователь конкретизирует свои потребности.

Разницу можно определить с помощью функции `DATEDIFF()`, которая имеет следующий вид:

```
DATEDIFF ( datepart , startdate , enddate )
```

Здесь также три параметра:

- `datepart` — часть даты, разницу в которой необходимо определить. Возможные значения показаны в табл. 2.3;
- `startdate` — начальная дата;
- `enddate` — конечная дата.

Следующая команда определяет количество дней между двумя датами:

```
SELECT DATEDIFF(Day, '03.04.2005', '07.08.2005')
```

В результате на экране мы должны увидеть число 126.

Имя даты

Некоторые пользователи любят смотреть на даты в виде символьных имен. Для этого используется функция `DATENAME`:

```
DATENAME ( datepart , date )
```

Посмотрим параметры этой функции:

- `datepart` — часть даты, символьное имя в которой необходимо определить. Возможные значения показаны в табл. 2.3;
- `date` — дата.

Несмотря на то, что в качестве первого параметра можно указывать любую часть даты, особого смысла в этом нет. Только при указании месяца будет возвращено символьное имя. Например, следующий запрос вернет нам в результате название 4-го месяца, т. е. April (Апрель):

```
SELECT DATENAME(Month, '04.03.2005')
```

Если попытаться узнать символьное имя числа, то функция вернет нам это число.

Разбиение даты на части

Для определения указанной части даты используются функции `DAY()`, `MONTH()` и `YEAR()` (рассмотрим ниже в этой главе), но они ограничены в возможностях. Более универсальной является функция `DATEPART()`, которая имеет следующий внешний вид:

```
DATEPART ( datepart , date )
```

Параметры функции:

- `datepart` — часть даты, значение которой необходимо определить. Возможные значения показаны в табл. 2.3;
- `date` — дата.

Следующий пример определяет значение месяца в дате:

```
SELECT DATEPART(Month, '04.03.2005')
```

В результате мы получим число 4.

Но если нужно определить дату, месяц или год, то проще будет воспользоваться одной из следующих функций:

- `DAY(дата)` — возвращает значение числа, указанное в дате;
- `MONTH(дата)` — возвращает значение месяца;
- `YEAR(дата)` — возвращает значение года.

Например, с помощью следующего запроса определяется год, который указан в дате '04.03.2005':

```
SELECT YEAR('04.03.2005')
```

2.22.3. Замечания по работе с датами

С типом `datetime` достаточно сложно работать. Рассмотрим следующие три запроса:

```
SELECT
FROM Товары
WHERE Дата<'5.11.2005'
```

В этом запросе мы выбираем товары, купленные до 11 числа пятого месяца. Пока ничего удивительного. Но посмотрите на следующий запрос:

```
SELECT
FROM Товары
WHERE Дата<='5.10.2005'
```

Здесь, по идее, запрашивается то же самое, ведь мы указываем товары, купленные до 10-го числа включительно. Знак "меньше" заменен на "меньше либо равно", а дата уменьшена на 1 день. Вы удивитесь, но результат может быть другим. Почему? Для этого нужно вспомнить, что представляет собой тип `datetime`. Если в поле для даты сохраняется не только дата, но и время, то числа будут дробными, а мы выполняем сравнение только с датой.

Возьмем число 10 мая 2005 года. Этой дате соответствует число 38481. Но если помимо даты в поле будет храниться и время, то число будет дробным, например, 38481,1943. Такое число удовлетворяет условию меньше `'5.11.2005'` (38481,1943 < 38482), но не удовлетворяет условию меньше либо равно `'5.10.2005'` (38481,1943 <= 38481 — это условие неверно).

Вы должны учитывать эту особенность при разработке собственных программ и четко понимать, какие данные хранятся в поле, и какие данные вы хотите получить из таблицы, иначе результат запроса может быть неточным.

Примечание

Примеры запросов из этом разделе можно найти в файле \Chapter2\dates.sql на компакт-диске, прилагаемом к книге.

2.23. Ход выполнения запроса

Все запросы, которые мы рассмотрели, были прямолинейными и просто последовательно выполняли определенные действия. Но логика не может быть всегда прямолинейной, она извилиста. Очень часто необходимо принимать решения на основе результатов, возвращаемых определенными запросами.

Для управления логикой в языке Transact-SQL есть следующие операторы:

- блок `BEGIN..END` — этот элемент заключает последовательность операторов Transact-SQL так, что SQL Server трактует их как одну команду. Чуть позже вы увидите, зачем это нужно;
- блок `IF Условие Выражение1 ELSE Выражение2` — эти элементы указывают, что SQL Server должен выполнить первое выражение, если определенное условие равно `true` (истина). Иначе он должен выполнить второе выражение;
- блок `CASE` — позволяет объединить несколько блоков сравнения в один оператор;
- конструкция `WHILE` — выполняет операторы циклически так долго, пока определенное выражение истинно (`true`). Операторы `BREAK` и `CONTINUE` контролируют выполнение операторов внутри цикла `WHILE`.

Эти операторы являются особенностью Transact-SQL и могут не восприниматься серверами баз данных, отличных от MS SQL Server. А теперь рассмотрим все эти возможности более подробно.

2.23.1. Условный оператор *IF*

В *главе 1* мы создавали и уничтожали объекты на свой страх и риск. Допустим, что вы хотите создать новую таблицу. Если просто выполнить оператор `CREATE TABLE`, когда объект с таким именем уже существует, то произойдет ошибка, и объект не будет создан. Чтобы объект создался, можно выполнить сначала удаление и сразу же создание. А если объект не существует? В этом случае при удалении произойдет ошибка, сценарий прекратит выполнение, и таблица не будет создана.

Чтобы решить эту проблему, можно запускать с клиентского компьютера два отдельных сценария. Первый будет удалять, а второй будет создавать базу. Но таким образом мы не сможем выполнить два оператора последовательно, и ошибка не будет отработана.

Намного элегантнее будет выглядеть код, который сначала проверит существование базы данных, и если она есть, то удалит ее и потом создаст. Как проверить

существование базы? В базе данных master есть таблица `sysdatabases`, в которой хранится описание всех созданных на сервере баз данных. В поле `name` этой таблицы находятся имена. Таким образом, следующий запрос отобразит информацию о базе данных с именем `NewDB`:

```
SELECT *
    FROM Master..sysdatabases
    WHERE Name='NewDB'
```

Но как теперь реализовать такую логику: "если запрос вернул строку, то удалить базу данных"? Для этого существует оператор `IF`, который имеет следующий вид:

```
IF Условие
    Команда 1
[ ELSE
    Команда 2 ]
```

Если условие возвращает истинное значение (`TRUE`), то выполняется первая команда. Можно добавить оператор `ELSE` и вторую команду, которая будет выполняться в случае ложного результата проверки условия, но это уже не является обязательным. Оператор `IF` самостоятелен и используется вне запроса `SELECT`.

Теперь наша проблема с созданием базы данных может быть решена следующим образом:

```
-- Проверка существования базы
IF EXISTS(SELECT *
    FROM Master..sysdatabases
    WHERE Name='NewDB'
   )
 DROP DATABASE NewDB

-- Создаем базу данных
CREATE DATABASE NewDB
```

Самое интересное находится после оператора `IF`. Здесь идет проверка `EXISTS`, которая возвращает истинное значение, если запрос в скобках после оператора возвращает хотя бы одну строку, т. е. база с именем `NewDB` существует. В этом случае база с таким именем будет удалена с помощью оператора `DROP DATABASE`. В случае ложного результата (нет записей в системной таблице) операция удаления будет пропущена, и сервер сразу же перейдет к выполнению следующей строки, где происходит создание.

Давайте улучшим вывод запроса и сделаем его более информативным:

```
-- Проверка существования базы
IF EXISTS(SELECT *
    FROM Master..sysdatabases
    WHERE Name='NewDB'
   )
 DROP DATABASE NewDB
ELSE
 print 'База данных NewDB не существует, создаем...'

-- Создаем базу данных
CREATE DATABASE NewDB
```

В этом примере, если запрос не вернул строк и условие оказалось ложным, выполнится строчка SQL-кода после оператора `ELSE`. Теперь рассмотрим следующий пример:

```
IF EXISTS(SELECT *
    FROM Master..sysdatabases
    WHERE Name='NewDB'
   )
 DROP DATABASE NewDB
ELSE
 print 'База данных NewDB не существует'
 print 'Создаем новую базу данных'
```

Как вы думаете, когда на экране появится надпись "Создаем новую базу данных"? Если условие ложное? Если внимательно читать определение оператора `IF`, то вы увидите, что ответ лежит на поверхности — в любом случае. Это сообщение выводится во втором операторе `print`, а после операторов `IF` и `ELSE` выполняется только один оператор. Это значит, что в случае ложного условия выполнится первый `print`, а второй уже не относится к условию и выполнится в любом случае, вне зависимости от условия.

Если необходимо, чтобы выполнилось два оператора, то вы должны объединить их с помощью `BEGIN` и `END`. Например:

```
IF Условие
 BEGIN
  -- Любое количество операторов
 END
ELSE
```

```
 BEGIN
  -- Любое количество операторов
 END
```

Между `BEGIN` и `END` можно указать любое количество операторов. Таким образом они объединяются в блок и воспринимаются как одна команда. Это значит, что для того, чтобы дважды выполнить функцию `print` при ложном результате условия, необходимо написать следующий код:

```
IF EXISTS(SELECT *
    FROM Master..sysdatabases
    WHERE Name='NewDB'
   )
 DROP DATABASE NewDB
ELSE
 BEGIN
  print 'База данных NewDB не существует'
  print 'Создаем новую базу данных'
 END
```

Рассмотрим еще один пример и заодно познакомимся с одной глобальной переменной MS SQL-сервера — `@@ROWCOUNT`. В ней хранится количество строк, которые были получены последним запросом. Давайте найдем количество работников с отчеством Палыч и выведем на экран соответствующее сообщение о количестве:

```
SELECT *
FROM tbPeoples
WHERE vcSurName LIKE '%ПАЛЫЧ'

DECLARE @rows_count int
SET @rows_count=@@ROWCOUNT

IF @rows_count=1
  print 'Один Палыч'

IF @rows_count=2
  print 'Два Палыча'
```

После выполнения запроса мы объявляем целочисленную переменную `@rows_count` и сохраняем в этой переменной значение `@@ROWCOUNT`. Зачем со-

хранять? После первого обращения эта глобальная переменная автоматически очищается. Рассмотрим следующий запрос:

```
SELECT *
FROM tbPeoples
WHERE vcSurName LIKE '%ПАЛЫЧ'
print @@ROWCOUNT
print @@ROWCOUNT
```

После выполнения `SELECT` дважды печатается значение глобальной переменной `@@ROWCOUNT`. В первый раз будет напечатано число 2, а после этого сразу 0, хотя других запросов не выполнялось.

Но вернемся к нашим Палычам. После сохранения количества записей в переменной `@rows_count` мы сравниваем эту переменную с 1, затем с цифрой 2 и выводим на экран соответствующее сообщение.

2.23.2. Условный оператор *CASE*

Если нужно сделать несколько сравнений с разными значениями, то лучше всего подойдет оператор `CASE`, который имеет следующий вид:

```
CASE input_expression
    WHEN when_expression THEN result_expression
        [ ...n ]
    [
        ELSE else_result_expression
    ]
END
```

Этот оператор должен использоваться внутри оператора `SELECT`. Это накладывает некоторые ограничения, но я бы даже сказал, что это делает `CASE` более мощным. Общий вид немного запутанный, поэтому рассмотрим оператор на условном примере:

```
CASE Переменная или поле
 WHEN Значение1 THEN Действие1
 ...
 ELSE ДействиеX
END
```

Итак, после `CASE` пишется имя поля или переменной. Затем идет оператор `WHEN`, который сравнивает указанное поле/переменную со значением. Если результат истина, то выполняется действие после `THEN`. Конструкции `WHEN .. THEN` может повторяться сколько угодно раз. Если ни одно из усло-

вий WHEN не выполнено и есть оператор ELSE, то будет выполнено его действие. Конец оператора CASE — ключевое слово END.

Также хочу отметить, что после THEN и ELSE выполняется только одно действие. Если нужно выполнить несколько действий, то их необходимо объединить с помощью BEGIN..END.

Давайте рассмотрим несколько примеров. Для начала переделаем пример с Палычем, используя CASE:

```
SELECT *
FROM tbPeoples
WHERE vcSurName LIKE '%ПАЛЫЧ'

SELECT
CASE @@ROWCOUNT
 WHEN 1 THEN
  'Один Палыч'
 WHEN 2 THEN
  'Два Палыча'
END
```

Сначала выполняется запрос на выборку всех работников с отчеством Палыч. Следующий запрос отображает результат проверки оператора CASE. В данном случае происходит проверка значения переменной @@ROWCOUNT с разными значениями. Если глобальная переменная равна 1, то в качестве результата оператор SELECT вернет текст "Один Палыч".

Давайте посмотрим, как можно использовать CASE с полями. Следующий запрос будет выводить на экран три колонки: фамилию, имя и приветствие. Третья колонка будет содержать текст приветствия, которое будет зависеть от должности работника (содержимого поля idPosition):

```
SELECT vcFamil, vcName,
 (CASE idPosition
  WHEN 1 THEN
   'Здравствуйте, Биг Босс'
  WHEN 2 THEN
   'Здравствуйте, Босс'
  ELSE
   'Здравствуйте'
 END) AS Приветствие
FROM tbPeoples
```

Если поле idPosition содержит число 1, то это генеральный директор, и для него будет приветствие "Здравствуйте, Биг Босс".

2.23.3. Оператор цикла *WHILE*

Допустим, что нужно выполнить определенную операцию несколько раз. Для этого можно несколько раз написать один и тот же запрос, но это будет довольно неудобно и некрасиво. А если количество повторений должно равняться 100? Тиражировать такой запрос достаточно сложно. Намного лучше написать что-то типа: "выполнять запрос 100 раз". И в Transact-SQL есть такая возможность — оператор `WHILE`, который в общем виде выглядит следующим образом:

```
WHILE Boolean_expression
    { sql_statement | statement_block }
    [ BREAK ]
    { sql_statement | statement_block }
    [ CONTINUE ]
```

Рассмотрим один шуточный пример. Представим, что мы сходили в универсам и занесли список покупок в таблицу `Товары`. Потом мы заметили, что если судить по базе данных, то общие затраты на покупки составили 320 рублей, а если подсчитать карманные деньги, получается, что должно быть 400. Где-то мы потеряли 80 рублей, и необходимо затраты подогнать под реальную сумму, чтобы избежать неприятного разговора с тем, кто отправил нас в магазин (например, со строгой женой). Для этого можно придумать одну покупку на 80 рублей, чтобы затраты в базе стали равны 400, а можно постепенно завышать цену товаров, пока общая стоимость покупок не достигнет 400. Конечно, это откровенный обман, и подтасовка данных наказуема, но чисто теоретически такая задача достаточно просто решается с помощью оператора `WHILE`:

```
WHILE (SELECT SUM(Цена*Количество)
       FROM Товары)<400
 UPDATE Товары SET Цена=Цена*1.2
```

После `WHILE` в скобках указан запрос на выборку суммы затрат на покупки. Результат сравнивается с цифрой 400, и если он меньше, то выполняется следующая команда (в нашем случае это `UPDATE`).

Увеличивать цены бесконечно невозможно, потому что иногда они превышают все разумные пределы и могут стать выше даже цен самого дорогого магазина Москвы. Поэтому вполне логично было бы сделать такую проверку: если цена моркови стала больше 25 рублей, то увеличивать уже не стоит, потому что цены станут нереальными. Это реализовано в следующем запросе:

```
WHILE (SELECT SUM(Цена*Количество)
       FROM Товары) < 1000
```

```
BEGIN
 UPDATE Товары SET Цена=Цена*1.2

 IF (SELECT Цена
     FROM Товары
     WHERE [Название товара]='Морковь') > 25
  BREAK
END
```

Выполните этот запрос в транзакции и откатите его, чтобы не изменять реальные данные таблицы, пусть данные останутся такими, как есть.

В этом примере в цикле нужно выполнить уже две операции — обновить цены в таблице и проверить цену моркови, поэтому обе операции объединены в `BEGIN..END`. После обновления проверяем цену моркови: если цена больше 25, то выполняется оператор `BREAK`, который прерывает выполнение цикла, даже если условие `WHILE` все еще не выполнено. Таким образом можно прерывать любой цикл.

Еще усложним задачу. Теперь будем одновременно увеличивать и количество, и цену. При этом если цена моркови превысила 25 рублей, то увеличивать надо только количество:

```
WHILE (SELECT SUM(Цена*Количество)
       FROM Товары) < 1000
BEGIN
 -- Обновляем количество
 UPDATE Товары SET Количество=Количество+1

 -- Проверяем цену
 IF (SELECT Цена
     FROM Товары
     WHERE [Название товара]='Морковь') > 25
  CONTINUE

 -- Обновляем цену
 UPDATE Товары SET Цена=Цена*1.2
END
```

Теперь в цикле выполняются три операции:

1. Первый `UPDATE` обновляет количество купленного товара, увеличивая его на 1.

2. Следующим шагом определяем цену моркови. Если она больше 25, то выполняется оператор `CONTINUE`, который заставляет повторить цикл сначала, без выполнения оставшихся операторов цикла.
3. Увеличить цену товара.

Таким образом, если цена моркови меньше 25, то произойдет обновление цены. Если больше 25, то цена обновляться не будет, а произойдет переход на начало цикла, т. е. произойдет проверка общих затрат на товары и, если необходимо, тело цикла будет выполняться сначала. Получается, что первое обновление будет происходить в любом случае, а второе обновление (шаг 3) будет выполняться, только если цена моркови меньше 25.

2.23.4. Прерывание работы сценария

Для того чтобы досрочно прервать работу сценария по определенному условию, нужно использовать оператор `RETURN`:

```
IF Условие
  RETURN
```

Рассмотрим пример. В *разд. 2.23.3* мы изменяли цену моркови, но что произойдет, если моркови вообще нет в таблице? В этом случае сценарий будет работать неверно, поэтому необходимо сначала сделать проверку существования соответствующей записи и только потом что-то обновлять. Таким образом, сценарий обновления может выглядеть следующим образом:

```
SELECT Цена
FROM Товары
WHERE [Название товара]='Морковь'

IF @rows_count=1
  RETURN

IF (SELECT Цена
     FROM Товары
     WHERE [Название товара]='Морковь') > 25
  RETURN

-- Обновляем цену
UPDATE Товары SET Цена=Цена*1.2
```

Обязательно проверяйте все параметры и результаты выполнения сценариев, это позволит сделать ваши сценарии более надежными и предсказуемыми.

Что значит предсказуемыми? В некоторых случаях сценарий будет выполняться не так, как мы ожидали, и все данные могут потеряться. Допустим, необходимо удалить все записи, у которых цена равна цене моркови:

```
DECLARE @cen int

SELECT @cen=Цена
FROM Товары
WHERE [Название товара]='Морковь'

DELETE
FROM Товары
WHERE [Цена]=@cen
```

В этом примере мы определяем цену моркови и сохраняем ее в переменной `@cen`. Затем выполняем запрос на удаление всех записей с такой ценой. А что произойдет, если моркови нет в таблице? Переменная `@cen` будет содержать нулевое значение, и в этом случае могут быть удалены записи с нулевой ценой, что нам абсолютно не нужно.

2.23.5. Подмена

Давайте вспомним запрос, в котором данные из таблицы должностей связываются сами с собой:

```
SELECT p1.vcPositionName AS 'Должность',
       p2.vcPositionName AS 'Главная должность'
FROM tbPosition p1, tbPosition p2
WHERE p1.idParentPosition*=p2.idPosition
```

В результате мы получаем примерно следующую таблицу:

```
Должность                      Главная должность

-----------------------------------------------------

Генеральный директор           NULL

Коммерческий директор          Генеральный директор

Директор по общим вопросам     Генеральный директор

Начальник отдела снабжения     Коммерческий директор

...
```

Понятно, что у генерального директора нет главной должности, ведь его должность является самой главной в организации. Но плохо то, что в резуль-

тате пользователь видит NULL. Нужно заменить NULL на более понятный текст, и это можно сделать с помощью функции ISNULL(). Эта функция имеет следующий вид:

```
ISNULL(Проверяемое значение, замена)
```

В скобках указываются два параметра:

- поле или выражение, которое необходимо сравнить с нулевым;
- если первый параметр равен NULL, то он будет заменен на значение, указанное во втором параметре.

Итак, подкорректируем наш запрос так:

```
SELECT p1.vcPositionName AS 'Должность',
  ISNULL(p2.vcPositionName, 'Это биг босс') AS 'Главная должность'
FROM tbPosition p1, tbPosition p2
WHERE p1.idParentPosition*=p2.idPosition
```

Теперь второе поле проверяется функцией ISNULL(), и если значение нулевое, то оно заменяется на текст из второго параметра. Таким образом, результат запроса будет следующим:

```
Должность                      Главная должность
-----------------------------------------------------
Генеральный директор           Это биг босс
Коммерческий директор          Генеральный директор
Директор по общим вопросам     Генеральный директор
Начальник отдела снабжения     Коммерческий директор
...
```

2.23.6. Ожидание

С помощью оператора WAITFOR можно перевести запрос в режим ожидания. Допустим, вы хотите выяснить, как изменились за один час данные в таблице. Для этого можно выбрать необходимые данные во временную таблицу, подождать час и сравнить содержимое таблицы с выбранными ранее данными. Если не использовать WAITFOR, то придется запускать два отдельных сценария с перерывом в один час. Но благодаря WAITFOR все можно сделать автоматически без лишних затрат.

Еще пример — допустим, что необходимо выполнить один раз некоторый сценарий в конце рабочего дня. Если бы такие действия требовались еже-

дневно, то можно было бы создать задание (о заданиях мы будем говорить в *гл. 3*), иначе это неэффективно. Усложним задачу — допустим, что вы не можете дожидаться конца рабочего дня, и вам нужно отлучиться. Чтобы все же выполнить команду в указанное время, можно запустить задержку, которая обеспечит, что по истечении указанного времени выполнится код, следующий после оператора `WAITFOR`.

В общем виде команда выглядит следующим образом:

```
WAITFOR { DELAY 'time' | TIME 'time' }
```

В качестве параметра передается или длительность задержки (максимум 24 часа), или время суток. Следующий пример показывает, как создать задержку в одну минуту:

```
WAITFOR DELAY '000:01:00'

SELECT *
FROM Таблица
```

В этом примере запускается задержка в одну минуту, после чего выполнится запрос `SELECT` на выборку данных.

Следующий пример создает задержку до 18:30 текущего дня:

```
WAITFOR TIME '18:30'

SELECT *
FROM Таблица
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\control.sql на компакт-диске, прилагаемом к книге.

2.24. Работа с GUID-полями

Мы уже не раз говорили о том, что для первичного ключа лучше всего использовать тип `uniqueidentifier`, который лучше с точки зрения обеспечения уникальности строк. Но, с другой стороны, этот тип хуже для индекса, потому что вновь добавляемые записи могут содержать GUID-идентификатор, меньший, чем уже существующие в таблице.

Работа с GUID-полями имеет достаточно преимуществ и недостатков. В этой главе мы рассмотрим и то, и другое, а также примеры обхода недостатков.

Давайте для иллюстрации примеров с GUID-полями создадим две связанные таблицы: типы телефонов и номера телефонов. Подобные таблицы у нас уже существуют в тестовой базе данных, поэтому я взял их копию, заменил тип ключевых полей на `uniqueidentifier` и добавил к именам таблиц и ограничений префикс `_G`. Все это в виде SQL-сценария можно увидеть в листинге 2.14.

Листинг 2.14. Создание таблиц, связанных через `uniqueidentifier`-поля

```
-- Создаем таблицу типов телефонов
CREATE TABLE tbPhoneType_G
(
 idPhoneType uniqueidentifier DEFAULT NEWID(),
 vcTypeName varchar(20),

 CONSTRAINT PK_idPhoneType_G PRIMARY KEY (idPhoneType),
)

-- Создаем таблицу телефонов
CREATE TABLE tbPhoneNumbers_G
(
  -- Описание полей
  idPhoneNumbers uniqueidentifier DEFAULT NEWID(),
  idPhoneType uniqueidentifier,
  vcPhoneNumber varchar(15),

  -- Описание ключей
  CONSTRAINT PK_idPhoneNumbers_G PRIMARY KEY (idPhoneNumbers),
  CONSTRAINT FK_idPhoneType_G FOREIGN KEY (idPhoneType)
     REFERENCES tbPhoneType_G (idPhoneType),

  -- Описание ограничений
  CONSTRAINT check_vcPhonenumber_G CHECK (vcPhonenumber LIKE
    '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]')
)
```

Обратите внимание, что для первичного ключа в качестве значения по умолчанию указана функция `NewID()`, которая генерирует уникальный идентификатор. Если пользователь будет добавлять запись и не укажет явное значение, то оно будет сгенерировано.

Следующие два примера добавляют в таблицу типов телефонов сотовый и домашний:

```
-- Добавление типа телефона - сотовый
INSERT INTO tbPhoneType_G
VALUES (NewID(), 'Сотовый')

-- Добавляем домашний тип телефона
INSERT INTO tbPhoneType_G (vcTypeName)
VALUES ('Домашний')
```

Теперь посмотрим, что у нас хранится в таблице:

```
SELECT *
FROM tbPhoneType_G
```

В моем случае результат получился следующим:

```
idPhoneType                          vcTypeName
-------------------------------------------------------
2A730DD9-3F89-48FD-A7C6-7FACDE496D2E   Домашний
BA80361E-1398-4C58-9DF9-EC25C18A2439   Сотовый
```

Мне повезло. Я наглядно смог увидеть, что домашний тип телефона добавлялся позже, а при отображении он стоит первым, потому что для него сгенерирован меньший идентификатор. Это и есть недостаток, который невозможно контролировать. Единственный удачный выход — добавить в таблицу типов еще одно поле, которое будет иметь автоматически увеличиваемое значение.

Давайте очистим таблицу (`DELETE FROM tbPhoneType_G`) и добавим новое поле:

```
ALTER TABLE tbPhoneType_G
ADD [id] int IDENTITY(1, 1)
```

Снова добавьте в таблицу записи типов телефонов. Чтобы просмотреть все строки в порядке добавления в таблицу, необходимо отсортировать вывод по полю `id`:

```
SELECT *
FROM tbPhoneType_G
ORDER BY [id]
```

От одного недостатка избавились. Давайте двигаться дальше. Теперь посмотрим, как можно добавлять записи в таблицу телефонов. В таблице телефонов нам нужно указать идентификатор типа и номер телефона. Проблема

возникает именно с первым значением, потому что вводить идентификаторы вручную проблематично. Эта проблема решается следующим образом:

```
-- Объявляем переменную
DECLARE @TypeID uniqueidentifier

-- Записываем в нее идентификатор домашнего телефона
SELECT @TypeID=idPhoneType
FROM tbPhoneType_G
WHERE vcTypeName='Домашний'

-- Добавляем запись с номером телефона
INSERT INTO tbPhoneNumbers_G (idPhoneType, vcPhoneNumber)
VALUES (@TypeID, '(000) 912-12-12')
```

В этом примере сначала объявляется переменная `@TypeID` типа `uniqueidentifier`. Затем в эту переменную с помощью запроса `SELECT` записывается идентификатор домашнего типа телефона. Теперь у нас есть все необходимое для добавления записи, что и происходит в последнем запросе.

Посмотрим, как можно увидеть записи в связанных таблицах. В данном случае безразлично, имеет ли ключевое поле тип GUID или является числом:

```
SELECT *
FROM tbPhoneType_G pt, tbPhoneNumbers_G pn
WHERE pt.idPhoneType=pn.idPhoneType
```

Допустим, что вы четко знаете, что GUID-идентификатор типа телефона равен 09F972F4-CFF9-4E96-9B7C-14AA04B835F3. Как его можно использовать напрямую? Чтобы SQL-запрос воспринял это число как GUID, его необходимо заключить в фигурные скобки и передать в виде строки.

Следующий пример объявляет переменную `@TypeID` типа `uniqueidentifier` и с помощью `SET` помещает в эту переменную явное значение, которое потом будет использоваться при добавлении записи с номером телефона:

```
-- Объявляем переменную
DECLARE @TypeID uniqueidentifier

-- Записываем в переменную значение
SET @TypeID='{09F972F4-CFF9-4E96-9B7C-14AA04B835F3}'

-- Вставляем запись
INSERT INTO tbPhoneNumbers_G (idPhoneType, vcPhoneNumber)
VALUES (@TypeID, '(000) 112-10-10')
```

Обратите внимание, что значение GUID сначала заключается в фигурные скобки и после этого еще и в одинарные кавычки. Если что-то из этого не указать, сервер вернет ошибку.

Зная идентификатор, можно добавить строку одной командой `INSERT INTO`. Например:

```
INSERT INTO tbPhoneNumbers_G (idPhoneType, vcPhoneNumber)
VALUES ('{09F972F4-CFF9-4E96-9B7C-14AA04B835F3}', '(000) 222-10-10')
```

Благодаря использованию GUID-идентификаторов упрощается добавление записей сразу в две таблицы. Например, в следующем запросе в таблицу типов добавляется новая строка "`Сотовый личный`", и тут же мы добавляем номер телефона с соответствующим типом:

```
-- Объявляем переменную
DECLARE @TypeID uniqueidentifier

-- Генерируем GUID значение
SET @TypeID=NewID()

-- Вставляем запись в таблицу типов
INSERT INTO tbPhoneType_G
VALUES (@TypeID, 'Сотовый личный')

-- Вставляем запись в таблицу телефонов
INSERT INTO tbPhoneNumbers_G (idPhoneType, vcPhoneNumber)
VALUES (@TypeID, '(901) 111-11-11')
```

Здесь в переменную `@TypeID` заносится результат выполнения функции `NewID()`, т. е. сгенерированное значение. Затем строки с таким идентификатором добавляются в таблицу типов телефонов и в таблицу номеров телефонов.

У нас достаточно знаний, чтобы усложнить пример и сделать проверку, существует ли в таблице нужный телефон, и добавлять запись только в том случае, если не существует. Если нужный тип уже есть в таблице, то нужно использовать его GUID. Все это реализовано в листинге 2.15.

Листинг 2.15. Добавление типа только в случае необходимости

```
Объявляем переменную
DECLARE @TypeID uniqueidentifier

-- Проверяем, есть ли нужная запись в таблице
```

```
IF NOT EXISTS (SELECT idPhoneType
               FROM tbPhoneType_G
               WHERE vcTypeName='Сотовый личный')
 BEGIN
  -- Если не существует, то генерируем GUID
  -- и добавляем строку, используя сгенерированное
  -- значение в качестве ключа
  SET @TypeID=NewID()
  INSERT INTO tbPhoneType_G
  VALUES (@TypeID, 'Сотовый личный')
  print 'Создаем тип "Сотовый личный"';
 END
ELSE
 BEGIN
  -- Если существует, то ищем GUID в таблице
  print 'Такой тип уже существует, используем его';

  SELECT @TypeID=idPhoneType
  FROM tbPhoneType_G
  WHERE  vcTypeName='Сотовый личный'
 END

-- Вставляем строку
INSERT INTO tbPhoneNumbers_G (idPhoneType, vcPhoneNumber)
VALUES (@TypeID, '(901) 111-11-11')
```

Как видите, проблемы при использовании GUID есть, но они решаемы. А ведь преимущества от использования GUID затмевают все недостатки. Главное преимущество — возможность самостоятельного задания первичного ключа без риска нарушения целостности. Это преимущество особенно проявляется при разработке клиентского приложения на языках высокого уровня, например, Delphi или C++. На этих языках пользователь может вводить данные на стороне клиента и сохранять их в памяти. Когда данные накопятся, их можно одним блоком сохранить на сервере.

Пока данные находятся в памяти клиентской машины, первичный ключ с автоматически увеличиваемым полем не может содержать значения, а значит, нельзя создавать связи. Если первичный ключ имеет тип GUID, то программа может сама создать значение для ключа и использовать его для создания связей в памяти клиентской машины с данными, которые еще даже не сущест-

вуют на сервере. Сейчас уже почти во всех языках есть собственные возможности генерирования GUID-значений, результат которых абсолютно одинаков.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\guid.sql на компакт-диске, прилагаемом к книге.

2.25. Функции работы со строками

У SQL-сервера достаточно много мощных функций для работы со строками, и в этом разделе мы рассмотрим наиболее интересные и часто используемые из них. Из моего личного опыта (ваши задачи могут дать другой результат), наиболее часто используемой является функция `SUBSTRING`. Именно с нее мы и начнем.

2.25.1. Функция *SUBSTRING*

Помнится, что мы добавили к значениям в колонке имен работников префикс `'mr.'` *(см. разд. 2.17)*. А как теперь от него избавиться во время обращения к таблице? Достаточно просто, если воспользоваться функцией `SUBSTRING()`, которая возвращает указанную часть строки. Этой функции необходимо передать три параметра:

- поле, часть строки которого нужно получить;
- первый символ;
- количество интересующих нас символов.

Посмотрим, как вышесказанное можно реализовать в виде запроса:

```
SELECT idPeoples,
      CASE SUBSTRING(vcFamil, 1, 3)
        WHEN 'mr.' THEN SUBSTRING(vcFamil, 4, 255)
        ELSE vcFamil
       END
FROM tbPeoples
```

В этом примере мы выбираем только два поля: `idPeoples` и поле, результат которого зависит от проверки `CASE`. В данном случае `CASE` проверяет результат работы функции `SUBSTRING()`, которая выбирает символы из поля `vcFamil`, начиная с первого по третий. Если результат равен `'mr.'`, то необходимо удалить этот префикс.

Для того чтобы отбросить ненужные символы, мы снова пользуемся функцией `SUBSTRING()`, но теперь выбираем символы, начиная с четвертого (т. е. с первого после `'mr.'`). В качестве параметра количества символов я указал число 255, что больше максимального значения поля, а значит, строка будет выбрана до конца, начиная с четвертого.

Теперь попробуем обновить данные в таблице, чтобы в поле `vcFamil` не было лишних символов `'mr.'`. Для этого выполним следующий запрос:

```
UPDATE tbPeoples
SET vcFamil=(case SUBSTRING(vcFamil, 1, 3)
      WHEN 'mr.' THEN SUBSTRING(vcFamil, 4, 255)
      ELSE vcFamil
     END)
```

В этом примере полю `vcFamil` присваивается результат сравнения `CASE`, который мы уже рассмотрели выше. Таким образом мы избавились от лишних букв в фамилиях.

2.25.2. Функция *LEFT*

Задачу удаления лишних символов из начала строки можно было бы решить и с помощью функции `LEFT()`, которая возвращает указанное количество символов, начиная с первого. Функции нужно передать следующие два параметра:

- поле, подстроку которого нужно получить;
- количество символов.

Следующий пример формирует ФИО, в котором имя и отчество сокращены:

```
SELECT vcFamil+' '+left(vcName, 1)+'. '+left(vcSurName, 1)+'.'
FROM tbPeoples
```

Поле `vcFamil` выводится полностью, а вот от имени и отчества выводится только один левый (первый) символ.

Теперь посмотрим, как можно было использовать `LEFT()` для удаления префикса `'mr.'`:

```
UPDATE tbPeoples
SET vcFamil=(case LEFT(vcFamil, 3)
      WHEN 'mr.' THEN SUBSTRING(vcFamil, 4, 255)
      ELSE vcFamil
     END)
```

2.25.3. Функция *LEN*

Функция `LEN()` позволяет определить длину строки или значения поля. Функции достаточно передать строку или имя поля, длина значения которого нас интересует. Например, следующий запрос отобразит длину всех значений в поле `vcFamil`:

```
SELECT vcFamil, len(vcFamil)
FROM tbPeoples
```

В следующем примере мы ищем записи, в которых фамилия состоит из семи символов:

```
SELECT vcFamil
FROM tbPeoples
WHERE len(vcFamil)=7
```

2.25.4. Функция *LOWER*

Если ваш сервер настроен так, что строки чувствительны к регистру букв, то при поиске по строковым полям могут возникнуть серьезные проблемы. Если вы указали фамилию как Иванов, то это значение не будет равно ИВАНОВ, а значит, мы не увидим необходимую запись. Проблему решает функция `LOWER()`, которая приводит указанную строку к нижнему регистру.

Рассмотрим пример. В следующем запросе мы выбираем все фамилии, при этом они отображаются в нижнем регистре (маленькими буквами):

```
SELECT LOWER(vcFamil)
FROM tbPeoples
```

Теперь рассмотрим следующий пример:

```
SELECT *
FROM tbPeoples
WHERE LOWER(vcFamil)=LOWER('Сидоров')
```

В секции `WHERE`, где мы сравниваем значение поля с введенной пользователем фамилией, и то, и другое приводится к нижнему регистру. Таким образом, в каком бы виде ни хранилась фамилия в базе, все Ивановы будут найдены.

2.25.5. Функция *UPPER*

Функция `UPPER()` также изменяет регистр букв, только делает их все большими. Это значит, что функцию можно тоже использовать для сравнения двух строк разного регистра, если все буквы привести к большим:

```
SELECT *
FROM tbPeoples
WHERE UPPER(vcFamil)=UPPER('Сидоров')
```

Если вам нужно сравнить две строки без учета используемого регистра букв внутри строк, можно использовать как `UPPER()`, так и `LOWER()`. Разницы никакой нет, поэтому выбирайте то, что больше нравится.

2.25.6. Функции *LTRIM* и *RTRIM*

Функция `LTRIM()` убирает все символы пробела в начале строки, а `RTRIM()` убирает пробелы в конце строки. Допустим, что пользователь при вводе фамилии в самом начале случайно задел клавишу пробела. Получилось, что в базе хранятся две фамилии:

```
Иванов
 Иванов
```

Когда смотришь на эти фамилии, то видно, что вторая строка сдвинута вправо за счет пробела вначале. Это значит, что база данных будет воспринимать эти значения по-разному. Чтобы избавиться от лишних пробелов, как раз используют функции `LTRIM()` и `RTRIM()`. Например:

```
SELECT *
FROM tbPeoples
WHERE LTRIM(vcFamil)=LTRIM(' Сидоров')
```

В этом примере поле `vcFamil` сравнивается с фамилией Сидоров, с пробелом в начале. Чтобы убрать пробел, используется функция `LTRIM()`. В следующем примере мы убираем и левые, и правые пробелы:

```
-- Убрать лишние пробелы
SELECT *
FROM tbPeoples
WHERE vcFamil=LTRIM(RTRIM(' Сидоров '))
```

Вообще-то, пробелы справа убираются сервером автоматически. Выполните следующий запрос и убедитесь сами:

```
SELECT *
FROM tbPeoples
WHERE vcFamil='Сидоров '
```

Если работник с фамилией Сидоров (без пробелов в конце) существует в таблице, и запрос отобразил его, то сервер автоматически убрал пробел.

2.25.7. Функция *PATINDEX*

С помощью функции PATINDEX() можно искать часть строки по определенному шаблону. Допустим, что надо найти все фамилии, в которых есть две буквы "о", между которыми может находиться любой символ. Эту задачу можно решить с помощью следующего запроса:

```
SELECT vcFamil, PATINDEX('%O_O%', vcFamil)
FROM tbPeoples
```

Если посмотреть на функцию PATINDEX(), то пока непонятно, чем она отличается от LIKE с шаблоном? Все очень просто — LIKE используется для создания ограничений в секции WHERE, а PATINDEX() возвращает индекс символа, начиная с которого идет подстрока, соответствующая указанному шаблону. Если бы мы использовали LIKE, то сервер вернул бы нам только те строки, где найдена подстрока, соответствующая шаблону:

```
SELECT vcFamil
FROM tbPeoples
WHERE vcFamil LIKE '%O_O%'
```

Если использовать функцию PATINDEX(), то в результат попадут все строки (мы не ограничиваем вывод в секции WHERE), но в тех строках, где в фамилии нет подстроки, соответствующей шаблону, будет стоять 0, а там, где есть, будет стоять 1. Посмотрим на результат выполнения запроса с использованием функции PATINDEX():

```
vcFamil                                   Ind
---------------------------------------------
ПОЧЕЧКИН                                  0
ПЕТРОВ                                    0
СИДОРОВ                                   4
КОНОНОВ                                   2
СЕРГЕЕВ                                   0
```

В данном примере подстрока, соответствующая шаблону '%O_O%', присутствует в фамилии Сидоров. Начиная с четвертого символа, идут буквы "оро".

2.25.8. Функция *REPLACE*

Функция REPLACE() позволяет найти в значении поля подстроку и заменить ее на новое значение. У этой функции три параметра:

- строка, в которой нужно искать подстроку;
- искомая подстрока;
- значение, которое нужно подставить.

Посмотрим пример использования этой функции:

```
SELECT vcFamil, REPLACE(vcFamil, 'opo', 'apo') AS Ind
FROM tbPeoples
WHERE PATINDEX('%O_O%', vcFamil)>0
```

Мы выбираем из таблицы два поля: фамилию и результат функции `REPLACE()`. Функция ищет в поле `vcFamil` строку "opo" и заменяет ее на строку "apo". Чтобы лучше было понятно, посмотрим на результат работы функции:

```
vcFamil                        Ind
---------------------------------------------
СИДОРОВ                        СИДароВ
КОНОНОВ                        КОНОНОВ
КОРОВА                         КароВА
МОЛОТКОВ                       МОЛОТКОВ
САДОВОДОВ                      САДОВОДОВ
СОДОРОЧКИН                     СОДароЧКИН

(6 row(s) affected)
```

В первой колонке показана фамилия из таблицы, а во второй колонке можно увидеть модифицированный с помощью функции `REPLACE()` вариант. Я думаю, что все понятно и без дополнительных комментариев.

2.25.9. Функция *REPLICATE*

С помощью функции `REPLICATE()` можно размножать строку. У функции два параметра:

- строка или имя поля, которое нужно вывести несколько раз;
- количество необходимых повторений.

Переходим к примеру. Следующий запрос выводит в результирующий набор дважды значение поля фамилии:

```
SELECT REPLICATE(vcFamil, 2)
FROM tbPeoples
```

В результате мы увидим нечто подобное:

```
ПОЧЕЧКИНПОЧЕЧКИН
ПЕТРОВПЕТРОВ
СИДОРОВСИДОРОВ
КОНОНОВКОНОНОВ
```

```
СЕРГЕЕВСЕРГЕЕВ
ВАСИЛЬЕВВАСИЛЬЕВ
...
```

В данном примере мало практического смысла, но функция не совсем бесполезна. Например, вы хотите нарисовать длинную двойную полоску. Можно нажать клавишу равенства и ждать, когда появится на экране полоска нужной длины, а можно просто клонировать знак равенства нужное количество раз. Следующий пример клонирует знак равенства 50 раз:

```
SELECT REPLICATE('=', 50)
```

Результат:

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

Красиво? А главное, удобно в управлении.

2.25.10. Функция *REVERSE*

Пару раз я встречался с необходимостью перевернуть строку "задом наперед", и в этом мне помогала функция `REVERSE()`. Ей нужно передать строку, и результатом будет та же строка, только буквы будут идти в обратном порядке. Например, следующий запрос переворачивает все фамилии:

```
SELECT REVERSE(vcFamil)
FROM tbPeoples
```

В реальных приложениях полностью строку вы будете менять достаточно редко, а вот часть строки может меняться. Например, в следующем запросе в фамилии меняются местами первые два символа:

```
SELECT REPLACE(vcFamil,
               LEFT(vcFamil, 2),
               REVERSE(LEFT(vcFamil, 2))
               )
FROM tbPeoples
```

Пример достаточно интересен тем, что лишний раз показывает, как использовать уже известные нам функции работы со строками. В результирующем наборе отображается результат работы функции `REPLACE()`. Функции нужно передать такие параметры:

- название поля, где хранится фамилия;
- первые два символа. Для получения первых двух символов используем уже знакомую нам функцию `LEFT()`;
- в качестве строки, которая будет поставлена вместо первых двух символов фамилии, выступают те же два символа, только перевернутые.

2.25.11. Функция *SPACE*

С помощью функции `SPACE()` можно создавать пробелы. В качестве единственного параметра нужно указать число, которое определяет количество возвращаемых пробелов. Работа функции идентична `REPLICATE()`, если в качестве клонируемого символа указать пробел.

Допустим, что нужно вывести на экран поля фамилию и имя, разделенные пятью пробелами. Можно сделать так:

```
SELECT vcFamil+'     '+vcName
FROM tbPeoples
```

А можно воспользоваться функцией `SPACE()`:

```
SELECT vcFamil+SPACE(5)+vcName
FROM tbPeoples
```

Зачем нужна функция, когда можно обойтись без нее? Допустим, что нужно использовать 5 пробелов в нескольких местах большого сценария. Все легко решается без функций, но впоследствии оказалось, что количество пробелов должно быть не 5, а 10. Придется пересматривать весь сценарий и корректировать пробелы. А если бы мы использовали `SPACE()` в сочетании с переменными, то проблема решилась бы намного проще.

Рассмотрим пример, в котором множественные пробелы используются дважды и для задания количества используется переменная:

```
DECLARE @sp int
SET @sp=10

SELECT vcFamil+SPACE(@sp)+vcName+SPACE(@sp)+vcSurName
FROM tbPeoples
```

Теперь достаточно только изменить значение переменной, и количество пробелов изменится во всем сценарии. А главное — количество пробелов может быть определено динамически, на основе запросов к таблице.

2.25.12. Функция *STR*

С помощью функции `STR()` можно форматировать дробные числа в строку. Чем это отличается от преобразования типов? Тип остается тем же, а на экран мы выводим строку в нужном виде. Функции нужно передать три параметра:

- дробное число, которое нужно форматировать;
- общее количество символов, включая числа до и после запятой, пробелы и знак;
- количество знаков после запятой.

Допустим, что нам нужно вывести название и цену товара. Но цена имеет тип money, который содержит слишком большое количество нулей. Чтобы избавиться от лишних цифр после запятой и получить строку, можно сначала привести тип money к типу number(10, 2), а потом результат привести к строке. Но можно решить все одной командой форматирования STR():

```
SELECT [Название товара], STR(Цена, 10, 2)
FROM Товары
```

Выполните этот запрос и обратите внимание, что второе поле (отформатированная цена) выровнена вправо:

```
Название товара
------------------------------------------------ ----------
КАРТОФЕЛЬ                                             13.60
Сок                                                   23.00
Шоколад                                               25.00
Хлеб                                                   6.00
Сок                                                   18.40
...
```

Выравнивание происходит из-за второго параметра — числа 10. Мы задали общее число символов, и выравнивание будет происходить по правой позиции указанного значения. Если второй параметр равен 10, а число состоит из четырех символов, то в начало результирующей строки будет добавлено 6 пробелов. Учитывайте это при использовании функции STR().

2.25.13. Функция *STUFF*

Функция STUFF() позволяет вставить строку в определенную позицию другой строки. У этой функции четыре параметра:

- строка, которую нужно изменить;
- позиция, в которую должна произойти вставка;
- количество удаляемых символов;
- вставляемая строка.

Длина вставляемой строки необязательно должна быть равна значению из третьего параметра. Во время выполнения функция сначала удаляет определенное количество символов, начиная с позиции из второго параметра, а затем вставляет новую строку.

Рассмотрим пример, в котором цена вставляется в поле названия товара, начиная с первой позиции, не удаляя ни одного из символов:

```
SELECT STUFF([Название товара], 1, 0, STR(Цена, 10, 2)+' ')
FROM Товары
```

Результат работы функции будет следующим:

```
-----------------------------------------
     13.60 КАРТОФЕЛЬ
     23.00 Сок
     25.00 Шоколад
      6.00 Хлеб
     18.40 Сок
     12.00 Молоко
      6.00 Хлеб
...
```

На этом примере лучше видно, что вставляемая цена выравнивается вправо. Так как мы указали в функции `STR()` количество символов, равное 10, то вставляется не реальный размер поля цены, а именно 10 символов.

Попробуйте увеличить третий параметр до 1. В этом случае первый символ в названии товара будет удален, а вместо него будет вставлена цена.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\string.sql на компакт-диске, прилагаемом к книге.

2.26. Математические функции

У MS SQL Server есть несколько математических функций, которые позволят нам перенести на сервер максимальный объем вычислений и освободить клиентские компьютеры от лишних расчетов, а значит, и нагрузки.

2.26.1. Знаки

Иногда бывает необходимо определить значение числового поля, не обращая внимания на знак (абсолютное значение). Для этого используется функция `ABS()`, которой в качестве параметра передается значение поля или переменной, а на выходе будет то же число, но без знака.

В следующем примере объявляется целочисленная переменная, которой присваивается число –10. После выполнения `ABS()` на экране мы увидим число 10, а знак будет отброшен:

```
DECLARE @num int
SET @num=-10
SELECT ABS(@num)
```

Следующий пример отображает название товара и абсолютное значение количества, ведь количество товара не может быть отрицательным (хотя в некоторых задачах встречаются исключения):

```
SELECT [Название товара], ABS(Количество)
FROM Товары
```

Для определения знака можно использовать функцию `SIGN()`. В качестве параметра этой функции нужно передать число, а в качестве результата мы получаем:

- –1, если число отрицательное;
- 0, если число равно нулю;
- 1, если положительное.

Следующий запрос вернет нам значение –1:

```
SELECT SIGN(-12)
```

2.26.2. Округление

Функция `CEILING()` возвращает ближайшее целое число, большее указанного в качестве параметра. Другими словами, происходит округление в большую сторону. Например, если в качестве параметра передать число 1.3, то ближайшее целое число, которое будет больше указанного, это 2. Следующий пример демонстрирует это в виде SQL-запроса:

```
DECLARE @num float
SET @num = 1.3
SELECT CEILING(@num)
```

Если в качестве параметра передать отрицательное число, например, –12.5, то будет возвращено число –12, которое больше и при этом является ближайшим.

Давайте посмотрим, как можно округлить цены в таблице товаров до ближайшего целого:

```
SELECT [Название товара], CEILING(Цена), ABS(Количество)
FROM Товары
```

Если в качестве параметра передать целое число, то функция вернет его, потому что оно уже является целым.

Обратную задачу выполняет функция `FLOOR()`, которая округляет дробное число до ближайшего меньшего числа. Например:

```
DECLARE @num float
SET @num = 1.3
SELECT FLOOR(@num)
```

В этом примере мы округляем число 1.3 до ближайшего меньшего, поэтому результатом будет 1.

Для отрицательного числа –4.6 ближайшим будет –5.

Для округления дробного числа до ближайшего целого (не имеет значения, в какую сторону) используется функция `ROUND()`. У нее два параметра: дробное число, которое нужно округлить, и целое число, определяющее, до какого разряда необходимо округлить.

Давайте рассмотрим несколько примеров округления на примере числа 262.1537. Для начала округлим до целого числа, для этого второй параметр должен быть равен нулю:

```
SELECT ROUND(262.1537, 0)
```

В результате мы увидим число 262.0000.

Если необходимо оставить после запятой один символ, то в качестве второго параметра нужно указать число 1. "Запятая" — это по российским правилам, а SQL-сервер работает по американским стандартам, и тут дробную часть отделяет точка. Итак, следующий пример оставляет один разряд после запятой:

```
SELECT ROUND(262.1537, 1)
```

В результате мы получаем число 262.2000.

Можно округлять и до любого разряда до запятой, для этого можно во втором параметре указать отрицательное значение. Если необходимо округлить до одного разряда до запятой, то в качестве второго параметра указываем число –1:

```
SELECT ROUND(262.1537, -1)
```

Результатом будет 260.0000.

2.26.3. Сложная математика

В этот раздел я включил логарифмы и экспоненту, т. к. для их вычислений нужно немалое процессорное время, поэтому лучше перенести эти расчеты на сервер.

Функция `EXP()` возвращает экспоненту указанного в качестве параметра числа:

```
DECLARE @angle float
SET @angle = 90
SELECT EXP(@angle)
```

Для определения логарифма указанного числа используется функция `LOG()`:

```
DECLARE @angle float
SET @angle = 5.3
SELECT LOG(@angle)
```

2.26.4. Случайное значение

Для генерации случайного значения используется функция `RAND()`. В качестве параметра передается переменная, которая определяет начальное значение или источник. Например:

```
DECLARE @r int
SET @r=20000000
SELECT RAND(@r)
```

Попробуйте выполнить этот запрос несколько раз, и вы увидите в результате одно и то же значение. Это серьезный недостаток. Для одного и того же значения переменной, указанного в качестве параметра, будет одно и то же случайное число. Получается, что результат `RAND()` нельзя назвать случайным.

2.26.5. Тригонометрические функции

Тригонометрические функции слишком схожи, поэтому будем рассматривать их все вместе.

Для определения арккосинуса используется функция `ACOS()`, которой необходимо передать дробное число от –1 до 1, которое определяет необходимый угол:

```
DECLARE @angle float
SET @angle = -1
SELECT ACOS(@angle)
```

В результате мы увидим арккосинус угла –1, который равен 3.1415926535897931. Ничего вам это не напоминает? Это число Пи.

Для получения числа Пи можно воспользоваться и более простым методом — вызвать функцию `PI()`:

```
SELECT PI()
```

Функция `ASIN()` возвращает арксинус указанного угла. Угол также задается в диапазоне от –1 до 1.

```
DECLARE @angle float
SET @angle = -1
SELECT ASIN(@angle)
```

Функция `ATAN()` возвращает арктангенс:

```
DECLARE @angle float
SET @angle = -1
SELECT ATAN(@angle)
```

Функция `COS()` возвращает косинус:

```
DECLARE @angle float
SET @angle = 90
SELECT COS(@angle)
```

Для определения синуса используем функцию `SIN()`:

```
DECLARE @angle float
SET @angle = 90
SELECT SIN(@angle)
```

Для определения тангенса используется функция `TAN()`:

```
DECLARE @angle float
SET @angle = 90
SELECT TAN(@angle)
```

2.26.6. Степень

Для работы со степенями есть только две функции: `SQRT()` — квадратный корень и `SQUARE()` — возведение числа в квадрат. Следующий пример показывает, как извлечь квадратный корень из числа 4:

```
SELECT SQRT(4)
```

В результате мы увидим число 2.

Следующий пример показывает, как возвести в квадрат число 6:

```
SELECT SQUARE(6)
```

В результате мы увидим число 36.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\math.sql на компакт-диске, прилагаемом к книге.

2.27. Связь "многие-ко-многим"

В *разд. 1.2.6* мы увидели, как создавать таблицы с различными типами связей. В этой главе мы научились работать с такими таблицами, но только поверхностно затронули тему "многие-ко-многим". Давайте восполним этот пробел. Для тестирования создадим три таблицы, которые мы рассматривали в *разд. 1.2.6*. Код создания таблиц для тестирования примеров из этой главы можно увидеть в листинге 2.16.

Листинг 2.16. Код создания трех таблиц со связью "многие-ко-многим"

```
-- Создаем таблицу фамилий
CREATE TABLE Names
(
 idName uniqueidentifier DEFAULT NEWID(),
 vcName varchar(50),
 vcLastName varchar(50),
 vcSurName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY (idName)
)

-- Создаем таблицу телефонов
CREATE TABLE Phones
(
 idPhone uniqueidentifier DEFAULT NEWID(),
 vcPhone varchar(10),
 CONSTRAINT PK_idPhone PRIMARY KEY (idPhone)
)

-- Создаем связующую таблицу
CREATE TABLE LinkTable
(
 idLinkTable uniqueidentifier DEFAULT NEWID(),
 idName uniqueidentifier,
 idPhone uniqueidentifier,

 CONSTRAINT PK_idLinkTable PRIMARY KEY (idLinkTable),

 CONSTRAINT FK_idPhone FOREIGN KEY (idPhone)
```

```
   REFERENCES Phones (idPhone),
 CONSTRAINT FK_idName FOREIGN KEY (idName)
   REFERENCES Names (idName)
)
```

Это немного модифицированный пример того, что мы рассматривали в *разд. 1.2.6*. В качестве ключевого поля используется уникальный идентификатор GUID.

Теперь попробуем разом добавить новую запись для человека, номер телефона и запись в связующей таблице. Весь этот код можно увидеть в листинге 2.17.

Листинг 2.17. Добавление записи в три связанных таблицы

```
-- Вставляем записи в связанные таблицы
DECLARE @ID1 uniqueidentifier
DECLARE @ID2 uniqueidentifier

-- Генерируем идентификаторы
SET @ID1=NEWID()
SET @ID2=NEWID()

-- Вставить жителя
INSERT INTO Names
VALUES(@ID1, 'ИВАНОВ', 'ИВАН', 'ИВАНОВИЧ')

-- Вставить в таблицу телефонов
INSERT INTO Phones
VALUES(@ID2, '39927321')

-- Вставить строку в связующую таблицу
INSERT INTO LinkTable (idName, idPhone)
VALUES(@ID1, @ID2)
```

Сначала мы объявляем две переменные `@ID1` и `@ID2` типа `uniqueidentifier`, которые будут хранить ключи для добавляемых записей. Затем генерируем эти ключи. Теперь можно добавлять записи во все три таблицы.

Теперь давайте посмотрим, как работает эта связь, и отобразим фамилию и соответствующий номер телефона:

```
SELECT vcName, vcPhone
```

```
FROM Names n, Phones p, LinkTable l
WHERE n.idName=l.idName AND p.idPhone=l.idPhone
```

В результате вы должны увидеть таблицу из двух полей:

```
vcName                   vcPhone
------------------------ ----------
ИВАНОВ                   39927321


(1 row(s) affected)
```

Нашу жизнь упрощает тот факт, что в качестве первичного ключа используется уникальный идентификатор, который достаточно сгенерировать на стороне клиента или до выполнения вставки данных и использовать в операторе `INSERT`. Если в качестве первичного ключа используется число с автоматическим увеличением, то, вставив одну строку с номером телефона, мы не знаем, какой ключ был ей назначен, и не можем установить связь. Необходимо сначала определить этот ключ запросом `SELECT`. Но решение этой проблемы существует, хотя и не очень элегантное.

Рассмотрим код из листинга 2.18. Здесь мы наполняем таблицы и устанавливаем связь без предварительной генерации уникального идентификатора для первичного ключа.

Листинг 2.18. Наполнение таблицы значениями

```
-- Заполняем фамилиями
INSERT INTO Names (vcName) VALUES('ИВАНОВ')
INSERT INTO Names (vcName) VALUES('ПЕТРОВ')
INSERT INTO Names (vcName) VALUES('СИДОРОВ')

-- Заполняем номерами телефонов
INSERT INTO Phones (vcPhone) VALUES('12387264')
INSERT INTO Phones (vcPhone) VALUES('45435345')
INSERT INTO Phones (vcPhone) VALUES('89349234')

-- Наводим связь 1
DECLARE @ID1 uniqueidentifier
DECLARE @ID2 uniqueidentifier

SELECT @ID1=idName FROM Names WHERE vcName='ИВАНОВ'
```

```
SELECT @ID2=idPhone FROM Phones WHERE vcPhone='12387264'

INSERT INTO LinkTable (idName, idPhone)
VALUES(@ID1, @ID2)
...
...
```

Как видите, связь "многие-ко-многим" ненамного сложнее, чем связь между двумя таблицами. Если четко представлять себе структуру данных и как происходит связь, то запросы написать несложно.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter2\linked.sql на компакт-диске, прилагаемом к книге.

Глава 3

Программирование на сервере

Мы уже научились писать довольно сложные запросы, и, казалось бы, что еще нужно? Полученных знаний достаточно, чтобы написать даже очень сложную программу, но вся ее логика будет располагаться на стороне клиента. Программа-клиент должна будет выполнять все необходимые действия по обработке данных. На первый взгляд, в этом ничего плохого нет, но если посмотреть на проблему ближе, то в таком решении можно увидеть много недостатков.

Выполнение сложных задач на сервере имеет следующие преимущества:

- улучшенная безопасность и более простые средства для управления правами доступа;
- нет необходимости передавать данные от сервера к клиенту по сети для выполнения каждого расчета. Достаточно направить серверу команду, и сервер выполнит все действия сам, без вмешательства клиента. Это позволяет освободить сетевые ресурсы от излишней нагрузки;
- нет необходимости в регулярном обновлении парка клиентских машин. Пользователи могут работать и на более старых машинах, потому что все сложные расчеты берет на себя сервер, который выполнит задачу быстрее;
- упрощается разработка клиентских приложений.

Это далеко не все преимущества. Читая эту главу, вы постепенно увидите, что использование программ на языке SQL на сервере действительно необходимо и удобно.

Почему глава называется "Программирование на сервере"? Дело в том, что в ней будут рассматриваться такие возможности базы данных, как представления, процедуры и функции, которые хранятся и выполняются на серверной стороне. Пользовательская программа только говорит, что нужно выполнить, а сервер выполняет нужный запрос или программу.

3.1. Представления

Первое, с чем мы начнем знакомиться, — представления (view), я их больше люблю называть просмотрщиками или объектами просмотра.

Представления позволяют хранить предопределенные запросы как объекты в базе данных для дальнейшего использования. Таблицы, запрашиваемые в представлении, называются базовыми таблицами. С некоторыми ограничениями вы можете именовать и хранить любой запрос `SELECT` как представление.

Для чего создаются представления? Можно выделить следующее:

- когда необходимо дать пользователю возможность просматривать не всю таблицу, а только ее часть. В этом случае пользователь обращается к представлению как к таблице и видит только те поля, которые ему нужны;
- для объединения двух или более таблиц. Таким образом можно упростить работу со связанными таблицами;
- вывод статистических данных. Например, чтобы не писать каждый раз запрос на вывод статистики, можно создать представление и использовать его по мере надобности.

3.1.1. Создание представления

Для создания представления используется оператор `CREATE VIEW`, который в общем виде выглядит следующим образом:

```
CREATE VIEW [ < database_name > . ] [ < owner > . ] view_name [ ( column
[ ,...n ] ) ]
[ WITH < view_attribute > [ ,...n ] ]
AS
select_statement
[ WITH CHECK OPTION ]

< view_attribute > ::=
    { ENCRYPTION | SCHEMABINDING | VIEW_METADATA }
```

После оператора `CREATE VIEW` идет имя представления, далее ключевое слово `AS` и запрос на выборку данных, который и будет отражать содержимое представления.

Посмотрим на этот оператор в действии. Следующий пример создает представление `PhoneView` для просмотра имен работников и номеров их телефонов:

```
CREATE VIEW PhoneView
AS
```

```
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

Как теперь можно использовать это представление? Точно так же, как и таблицу, т. е. выполнять запрос `SELECT`:

```
SELECT *
FROM PhoneView
```

Результат выполнения запроса:

```
vcFamil      vcName      vcSurName   dDateBirthDay   vcPhoneNumber
-----------------------------------------------------------------
mr.ИВАНОВ    ИВАН        ИВАНЫЧ      2004-01-31      (925) 102-51-01
mr.ИВАНОВ    ИВАН        ИВАНЫЧ      2004-01-31      (925) 163-31-52
mr.ПЕТРОВ    ИВАН        ПАЛЫЧ       1971-04-03      (923) 112-02-46
mr.СИДОРОВ   ИВАН        ПАЛЫЧ       1967-12-13      (923) 152-52-04
mr.СИДОРОВ   ИВАН        ПАЛЫЧ       1967-12-13      (095) 125-16-63
mr.КОНОНОВ   ШВАРЦ       ПЕТРОВИЧ    1981-12-13      (905) 100-10-10
...
(33 row(s) affected)
```

Представление дает несколько преимуществ:

- фокусирует данные для пользователей. Все пользователи будут обращаться к одним и тем же таблицам, но благодаря представлению каждый может видеть только то, что ему нужно, и так, как нужно;
- делает данные комплексными;
- упрощает управление разрешениями;
- организует данные для экспорта в другие приложения.

Представление создает контролируемое окружение, позволяющее сделать доступными только нужные данные, а ненужное отключить. Данные, которые по каким-либо причинам не должны отображаться, могут быть спрятаны с помощью представления.

Представления позволяют вам хранить результат комплексного запроса. Другие запросы могут использовать этот суммирующий результат.

Итак, представление — это запрос на языке SQL, который выбирает данные, а в базе данных оно выглядит как таблица, и работа с ним происходит так же. Из представления можно выбирать данные SQL-запросами и так же назначать права доступа. Получается, что будет выполняться запрос к запросу.

Давайте рассмотрим еще один псевдопример. Допустим, что есть таблица с доходами работников, и мы хотим спрятать от налоговой инспекции некоторые поля. Чтобы решить эту проблему, можно создать представление, которое будет выбирать только те поля, которые можно показывать налоговым инспекторам:

```
CREATE VIEW Зарплата AS
SELECT разрешенные для налоговой поля
FROM Работники, Доходы
WHERE навести связи
```

Теперь в нашей базе данных появился новый объект — `Зарплата`. Так как обращение к представлению происходит как к простой таблице, то налоговый инспектор тоже будет думать, что он видит реальные данные, хотя результат такого запроса будет содержать только то, что нам нужно.

В реальной жизни налоговую полицию так не обманешь (это псевдопример). Но на этом примере видно, что представление может оказаться отличным методом обеспечения безопасности. Мы можем показывать пользователям только те данные, которые им нужны, и ничего лишнего. При этом в наших руках остаются все инструменты по управлению правами доступа на представление, а права доступа на сами таблицы мы не затрагиваем.

Ограничение доступа к определенным полям называется вертикальной защитой. Представления позволяют создавать и горизонтальную защиту. Например, в таблице есть некоторые записи, которые должны быть видны только привилегированным пользователям. У таких пользователей в поле `Категория` стоит значение 1. Если запретить прямой доступ к таблице, а для всех пользователей создать представление, то можно скрыть записи. Представление может выглядеть следующим образом:

```
CREATE VIEW Зарплата AS
SELECT Список полей
FROM Работники, Доходы
WHERE навести связи
  AND Категория=1
```

В этом примере мы ограничиваем доступ к определенным записям, а не полям. Такая защита называется горизонтальной.

Конечно же, вы можете создать и горизонтальную защиту, и вертикальную в одном представлении. Этого нам никто запретить не может.

С помощью разных представлений одних и тех же таблиц экономисты могут видеть одни данные, бухгалтерия другие, а отдел кадров третьи. Если нужно

показать какую-то дополнительную колонку, то просто добавляем в запрос представления. Никаких прав изменять уже не понадобится.

В каждой базе данных могут быть системные представления, которые создаются сервером автоматически. Не советую разрешать к ним доступ, потому что они могут показать что-нибудь лишнее, что поможет хакеру повысить свои права или просто испортить данные. Системные представления начинаются с префикса `sys`, и в колонке `Type` списка светится надпись `System` (если просматривать хранимые на сервере представления с помощью программы Enterprise Manager).

Когда вы создаете представление, SQL Server проверяет существование объектов, на которые есть ссылки в объявлении представления. Имя представления должно удовлетворять правилам именования объектов базы данных. Вы должны именовать представления так, чтобы их имена отличались от имен таблиц, и их можно было выделить среди других объектов. Это значит, что имя представления не должно конфликтовать не только с именами уже существующих представлений, но и с именами таблиц базы данных. Если бы можно было создать представление с именем `tbPeoples`, то следующий запрос не смог бы определить, откуда выбирать данные — из представления или из таблицы с таким именем:

```
SELECT FROM tbPeoples
```

Для выполнения оператора `CREATE VIEW` вы должны иметь соответствующие права, например, быть владельцем базы данных. Вы также должны иметь права на выполнение оператора `SELECT` всех таблиц, используемых в представлении. Чтобы избежать ситуации, когда владельцем представления является один человек, а владельцем таблиц другой, всеми объектами должен владеть `dbo`. Всегда указывайте имя `dbo` при создании объектов. В следующем примере показано, как создать представление `PhoneView` на запрос, который мы рассматривали ранее в этой главе, с явным указанием владельца (`dbo`):

```
CREATE VIEW dbo.PhoneView
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

У всех полей представления должны быть уникальные имена. Поэтому необходимо во время выполнения оператора `CREATE VIEW` учитывать следующие особенности:

- если какая-нибудь из колонок представления получается с помощью арифметических расчетов или встроенных функций, то имя для такого поля будет отсутствовать, и его необходимо явно задать;

- если у вас есть связанная таблица, и вы выбираете поля из двух таблиц с одинаковым названием, то одному из полей нужно дать уникальный псевдоним.

Например, создадим представление, которое будет считать количество различных имен работников в таблице. Для этого запрос будет использовать группировку и функцию `count()`:

```
CREATE VIEW NamesView
AS
SELECT pl.vcName, count(*)
FROM tbPeoples pl
GROUP BY vcName
```

Если попытаться создать такое представление, то сервер вернет сообщение об ошибке, что для второй колонки не указано имя. Поэтому нужно явно указать это имя:

```
CREATE VIEW NamesView
AS
SELECT pl.vcName, count(*) AS Количество
FROM tbPeoples pl
GROUP BY vcName
```

Есть еще один способ задания полей для представления — в скобках после его имени:

```
CREATE VIEW NamesView1(Имя, Количество)
AS
SELECT pl.vcName, count(*)
FROM tbPeoples pl
GROUP BY vcName
```

Обратите внимание, что в секции `SELECT` имена не задаются, но они есть в скобках, после имени создаваемого представления. Посмотрите содержимое представления, и вы увидите следующий результат:

```
Имя                       Количество
------------------------------------
АНДРЕЙ                    5
БОЛИК                     1
ВАСЯ                      1
ВИКТОР                    1
ВИТЯ                      1
...
```

Теперь рассмотрим пример конфликта колонок. Допустим, что мы написали следующий запрос и решили его превратить в представление:

```
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber, pl.idPeoples,
  pn.idPeoples
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

В секции `SELECT` поле `idPeoples` выбирается из таблицы работников и из таблицы номеров телефонов. Если смотреть на этот код как на запрос, то ошибок в нем нет, и он должен выполниться. Теперь попытаемся создать представление:

```
CREATE VIEW ConflictView
AS SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber, pl.idPeoples,
  pn.idPeoples
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

В ответ на это мы получим сообщение об ошибке. Сервер предупредит о том, что поля представления должны быть уникальными. Проблема опять же решается с помощью задания псевдонима для одного из конфликтующих полей. Например, в следующем запросе полю `idPeoples` из таблицы `tbPhoneNumbers` задается псевдоним `PhoneNumbersID`:

```
CREATE VIEW ConflictView
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber, pl.idPeoples,
  pn.idPeoples AS PhoneNumbersID
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

Прежде чем создавать представление, вы должны протестировать запрос `SELECT`, чтобы убедиться, что он возвращает правильный набор. После этого проверьте результат, созданный представлением. Как мы видели, не каждый запрос `SELECT` может стать представлением, потому что есть некоторые ограничения на запрос `SELECT` и результат его работы. В примере выше оказалось, что камнем преткновения во время создания представления может стать банальный конфликт имен полей.

Существуют и другие ограничения на запрос SELECT, которые необходимо четко соблюдать:

- нельзя использовать ключевое слово INTO;
- нельзя использовать опции COMPUTE или COMPUTE BY;
- можно использовать оператор ORDER BY, только если используется ключевое слово TOP;
- представление не может ссылаться на временные таблицы;
- представление, как и таблица, не может содержать более 1024 колонок.

3.1.2. Редактирование представления

Очень редко бывает так, что какой-то объект в таблице остается без изменений на протяжении всего цикла жизни объекта. Все в жизни развивается и изменяется, и мы должны иметь возможность изменения данных без удаления объекта. Удаление ни одного из объектов не приносит пользы, потому что, как минимум, теряются права доступа. Ведь права назначаются не имени объекта, а идентификатору объекта, а он генерируется при создании объекта.

Оператор ALTER VIEW изменяет объявление представления. Это позволяет вам сохранить разрешения. В общем виде команда выглядит следующим образом:

```
ALTER VIEW [ < database_name > . ] [ < owner > . ]
         view_name [ ( column [ ,...n ] ) ]
[ WITH < view_attribute > [ ,...n ] ]
AS
    select_statement
[ WITH CHECK OPTION ]
```

Следующий пример изменяет PhoneView так, чтобы помимо номера телефона отражалось и название должности работника:

```
ALTER VIEW PhoneView
AS
SELECT vcFamil, vcName, vcSurname,
  vcPositionName, vcPhoneNumber, pn.idPeoples
FROM tbPeoples pl, tbPosition ps, tbPhoneNumbers pn
WHERE pl.idPosition=ps.idPosition
 AND pl.idPeoples=pn.idPeoples
```

Как видите, изменение происходит так же, как и создание представления.

3.1.3. Удаление представления

Если представление вам больше не нужно, то его следует удалить. Ничего лишнего в базе данных не должно быть, ведь неиспользуемые объекты отрицательно влияют на безопасность. Для удаления представления используется оператор `DROP VIEW`.

```
DROP VIEW { view } [ ,...n ]
```

Можно удалять сразу несколько представлений:

```
DROP VIEW NamesView, NamesView1, ConflictView
```

Я удалил все, что было создано в этой главе, кроме представления `PhoneView`. Возможно, что оно нам еще пригодится в будущем для тестирования каких-либо запросов.

3.1.4. Изменение содержимого представления

Я уже говорил о том, что с представлением можно работать так же, как и с простой таблицей. Это значит, что возвращаемый результат можно воспринимать как таблицу и редактировать. Давайте попробуем изменить в представлении `PhoneView` фамилию работника с идентификатором 1 на "Почечкин":

```
UPDATE PhoneView
SET vcFamil='ПОЧЕЧКИН'
WHERE idPeoples=1
```

Как видите, запрос идентичен изменению таблицы, и используется уже знакомый нам оператор `UPDATE`. Давайте еще изменим название должности генерального директора:

```
UPDATE PhoneView
SET vcPositionName='Самый генеральный директор'
WHERE vcPositionName='Генеральный директор'
```

Просмотрите содержимое таблицы `tbPosition` и убедитесь, что название должности генерального директора изменилось, хотя мы изменяли представление.

Представление `PhoneView` выводит результат из трех таблиц сразу. А если мы хотим изменить поля из нескольких таблиц одновременно? Давайте выполним следующий запрос:

```
UPDATE PhoneView
SET vcFamil='ПОЧЕЧКИН',
```

```
    vcPositionName='Самый генеральный директор'
WHERE idPeoples=1
```

В этом примере изменяются поле фамилии из таблицы `tbPeoples` и поле названия должности из таблицы `tbPosition`. Если попытаться выполнить этот запрос, то сервер вернет сообщение об ошибке, что нельзя изменять одновременно несколько полей.

3.1.5. Удаление строк из представления

Попробуем удалить запись из представления `PhoneView`:

```
DELETE
FROM PhoneView
WHERE idPeoples=1
```

Результатом снова будет ошибка, потому что во время удаления будет производиться попытка удалить полученные записи из связанных таблиц. Получается, что нельзя удалять строки из представлений, если выбираются строки из нескольких таблиц. А если выбрать из нескольких?

Давайте создадим представление, которое будет выбирать записи только из одной таблицы:

```
CREATE VIEW PhoneView1
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName
FROM tbPeoples pl
```

Теперь попробуем удалить запись:

```
DELETE
FROM PhoneView1
WHERE vcFamil='КОТИКОВ'
```

На этот раз все пройдет успешно, и запись будет удалена, потому что представление использует поля только одной таблицы.

3.1.6. Опции представления

Очень интересной опцией является `ENCRYPTION`, которая заставляет шифровать текст представления в системных таблицах SQL-сервера. Таким образом, даже если злоумышленник получил доступ к системе, то просмотр текста представления останется затруднительным. Это действительно очень полезное свойство. Защита лишней не бывает.

Давайте создадим представление, данные которого (исходный код) будут зашифрованы в системной таблице:

```
CREATE VIEW PhoneView1
WITH ENCRYPTION
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName
FROM tbPeoples pl
```

Перед ключевым словом `AS` мы поставили опцию `WITH ENCRYPTION`. Теперь текст запроса просмотреть нельзя, даже если вы получите полный доступ к базе и будете смотреть системные таблицы напрямую.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter3\view.sql на компакт-диске, прилагаемом к книге.

3.2. Хранимые процедуры

На мой взгляд, возможности представлений слишком малы и далеки от идеала. Главная проблема — статичность. Чтобы получить новый результат (добавить или изменить критерий поиска), приходится изменять саму хранимую процедуру, что довольно неудобно, и большая часть преимуществ представлений просто теряется.

Хранимые процедуры — это именованный набор операторов Transact-SQL, хранящийся на сервере. Они используются для выполнения повторяющихся задач и при этом обладают бóльшими возможностями, чем представления.

Сервер SQL поддерживает 5 типов встроенных процедур:

- системные хранимые процедуры — хранятся в базе данных master. Система хранит процедуры (определяющиеся по префиксу `sp_`), предоставляющие эффективные методы получения информации из системных таблиц. Они позволяют системному администратору выполнять администраторские задачи над базой данных, которые обновляют необходимые таблицы напрямую. Системные встроенные процедуры могут быть выполнены из любой базы данных;
- локальные хранимые процедуры — создаются в определенных пользовательских таблицах;
- временные хранимые процедуры — могут быть локальными с именами, начинающимися с единичного знака #, или глобальными, начинающимися

со знака ## (как и локальные/глобальные временные таблицы). Локальные временные процедуры доступны только в единственной пользовательской сессии. Глобальные доступны всем пользователям. Как для таблиц, так и для процедур я не рекомендую использовать временные процедуры. Я еще не встречался с такой задачей, которую нельзя было решить без временных процедур;

- удаленные хранимые процедуры — устаревшая технология MS SQL Server. На данный момент эту задачу решают распределенные запросы;
- расширенные встроенные процедуры (содержат в имени префикс `xp_`) — разрабатываются в виде DLL (Dynamic Link Library, динамически подключаемая библиотека) и выполняются вне окружения SQL Server. Обычно такие процедуры идентифицируются по префиксу `xp_`.

Хранимые процедуры в MS SQL Server похожи на процедуры в других языках программирования. Если вы имеете опыт программирования на каком-либо языке и не понаслышке знаете о таком понятии, как процедуры, то материал этой главы покажется вам слишком легким. Но уровень подготовки читателей может быть разным, поэтому я постараюсь описать все максимально простым и доступным языком.

Итак, процедура — это блок из одной или более команд. Это может быть не просто один запрос, а целая программа с собственной логикой (операторы `IF`), циклами. Процедура может принимать заранее определенные переменные и использовать их в своих расчетах, благодаря чему результат работы процедуры может быть динамическим и будет зависеть от определенных условий и/или состояния получаемых переменных.

В процедуре вы можете:

- использовать операторы, которые выполняют любые операции в базе данных (выборка, вставка, изменение или удаление данных), включая возможность вызова других встроенных процедур;
- использовать входные параметры;
- возвращать статус выполнения в вызывающую процедуру или модуль для отображения удачного или ошибочного выполнения;
- возвращать несколько значений в вызывающую процедуру или модуль в форме выходных параметров.

Выполнение хранимой процедуры включает в себя создание плана выполнения, который при первом выполнении помещается в кэш. Кэш процедур — это пространство памяти, содержащее планы выполнения всех выполняемых сейчас операторов Transact-SQL. Размер кэша изменяется, динамически соответствуя необходимому уровню. Если в кэше есть план выполнения, то про-

цедура выполняется быстрее за счет того, что серверу не нужно разбирать запрос и вырабатывать необходимые действия для решения поставленной задачи.

Когда хранимая процедура создается, операторы проверяются на синтаксическую корректность, чтобы в коде не было явных ошибок написания и использования операторов. На этом этапе логика работы еще не может быть проверена. При наличии синтаксической ошибки сервер выдаст сообщение об ошибке, и встроенная процедура не сохранится. Если ошибок нет, то сервер SQL сохранит имя процедуры в системной таблице `sysobjects`, а текст в системной таблице `syscomments` текущей базы данных.

Вызов имен с задержкой позволяет хранимым процедурам ссылаться на объекты, которые не существуют в момент создания процедуры. Этот процесс добавляет гибкости в работу процедуры, потому что те процедуры и объекты, на которые она ссылается, могут создаваться в любом порядке. Но при этом все необходимые объекты должны существовать в момент выполнения процедуры. Имена проверяются как раз во время выполнения процедуры.

Некоторые изменения в базе данных могут повлиять на план, делая его неэффективным или невыполнимым. Сервер определяет эти изменения и автоматически перекомпилирует план, когда возникает одна из следующих ситуаций:

- произошли структурные изменения в таблице или в представлении (операторы `ALTER TABLE` или `ALTER VIEW`);
- сгенерирована новая статистика с помощью оператора `UPDATE STATISTIC`;
- индекс, который использовался планом выполнения, удален;
- сделаны значительные изменения в ключах (операторы `INSERT`, `DELETE`).

Для построения оптимального плана выполнения сервер определяет количество данных в используемых таблицах, наличие и тип индексов таблицы, распределение данных в индексных колонках, наличие объединений.

3.2.1. Создание хранимых процедур

Теперь обсудим создание хранимых процедур. Для этого используется оператор `CREATE PROCEDURE`, который выглядит следующим образом:

```
CREATE PROC [ EDURE ] procedure_name [ ; number ]
    [ { @parameter data_type }
        [ VARYING ] [ = default ] [ OUTPUT ]
```

```
    ] [ ,...n ]

[ WITH
    { RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION } ]

[ FOR REPLICATION ]

AS sql_statement [ ...n ]
```

Для создания процедуры вы должны иметь соответствующие права, например, быть владельцем базы данных или администратором сервера базы данных.

Вы можете создать процедуры только в текущей базе данных, исключая временные процедуры, которые всегда создаются в базе данных tempdb. Создание процедуры похоже на создание представления. Первым делом напишите и протестируйте операторы Transact-SQL. После этого, если вы получили результат, который ожидали, создавайте процедуру.

Для имен процедур лучше всего выбрать префикс, который будет указывать, что эта процедура создана именно вами. Только не используйте для этого префикс `sp_`, чтобы не возникли конфликты с именами системных процедур.

Давайте попробуем создать процедуру, которая будет получать из таблицы данные о работниках и их телефонах, т. е. следующий запрос:

```
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

Это простейшая процедура, которая не будет использовать переменных, поэтому для ее создания необходимо написать:

```
CREATE PROCEDURE GetPhones
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

Как видите, в начало запроса всего лишь добавляется две строчки (хотя, можно написать и в одну). В первой мы пишем оператор `CREATE PROCEDURE` и имя процедуры, а во второй строчке ключевое слово `AS`. После этого идет простой запрос `SELECT`, который выбирает данные.

3.2.2. Выполнение процедур

Здесь мы узнаем, как можно выполнять процедуры. Для этого используется оператор EXECUTE, который выглядит следующим образом:

```
[ [ EXEC [ UTE ] ]
    {
        [ @return_status = ]
            { procedure_name [ ;number ] | @procedure_name_var
    }
    [ [ @parameter = ] { value | @variable [ OUTPUT ] | [ DEFAULT ] ]
        [ ,...n ]
[ WITH RECOMPILE ]
```

В общем виде команда выглядит пугающе, но к концу главы мы рассмотрим достаточно примеров, и вы увидите, что ничего страшного тут нет. Для процедуры GetPhones, которую мы создали ранее, необходимо указать только:

```
EXEC GetPhones
```

или

```
EXECUTE GetPhones
```

Результат выполнения команды:

```
vcFamil       vcName  vcSurName  dDateBirthDay   vcPhoneNumber
-------------------------------------------------------------------
ПОЧЕЧКИН      ИВАН    ИВАНЫЧ     2004-01-31      (925) 102-51-01
ПОЧЕЧКИН      ИВАН    ИВАНЫЧ     2004-01-31      (925) 163-31-52
ПОЧЕЧКИН      ИВАН    ИВАНЫЧ     2004-01-31      (095) 162-02-46
mr.ПЕТРОВ     ИВАН    ПАЛЫЧ      1971-04-03      (923) 112-02-46
mr.ПЕТРОВ     ИВАН    ПАЛЫЧ      1971-04-03      (095) 632-06-58
mr.СИДОРОВ    ИВАН    ПАЛЫЧ      1967-12-13      (923) 152-52-04
mr.СИДОРОВ    ИВАН    ПАЛЫЧ      1967-12-13      (095) 622-31-42
mr.СИДОРОВ    ИВАН    ПАЛЫЧ      1967-12-13      (095) 125-16-63
mr.КОНОНОВ    ШВАРЦ   ПЕТРОВИЧ   1981-12-13      (905) 100-10-10
...
```

Наша процедура просто выбирает данные, и именно их мы видим в результате.

3.2.3. Удаление процедур

Теперь посмотрим, как можно удалять процедуры. Для этого используется оператор DROP PROCEDURE, который позволяет удалять несколько процедур сразу. В общем виде этот оператор выглядит так:

```
DROP PROCEDURE { procedure } [ ,...n ]
```

3.2.4. Использование параметров

Давайте создадим процедуру с параметрами. Допустим, что мы хотим создать процедуру, которая будет искать номер телефона нужного работника по фамилии. Для этого необходимо в качестве параметра получать искомую фамилию и использовать ее во время поиска:

```
CREATE PROCEDURE GetPhones
   @Famil varchar(50)
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
  AND vcFamil=@Famil
```

Параметры перечисляются через запятую после имени процедуры в виде `имя тип`. Я выделил параметры отдельной строчкой (вторая), которая идет после имени процедуры, но до ключевого слова `AS`. В данном примере только один параметр с именем `@Famil` и типом `varchar` длиной в 50 символов.

Имена параметров подчиняются тем же правилам именования, что и переменные, и используются так же. В данном примере в секции `WHERE` происходит сравнение значения поля `vcName` со значением параметра `@Famil`.

Чтобы выполнить процедуру с параметром, нужно написать следующий запрос:

```
EXECUTE GetPhones 'ВАСИЛЬКОВ'
```

Значения, которые будут присвоены параметрам во время выполнения процедуры, перечисляются через запятую после имени процедуры.

Необходимо заметить, что некоторые программы, например, Query Analyzer, не требуют писать оператор `EXEC`. Достаточно написать имя процедуры и перечислить параметры:

```
GetPhones 'ВАСИЛЬКОВ'
```

Но такой пример может сработать далеко не всегда, вернее, не во всех программах. Я рекомендую всегда писать вначале оператор `EXECUTE` или, сокращенно, `EXEC`.

Параметры нужно передавать в том же порядке, как они были указаны в объявлении, но можно сделать и отступление, если указывать их в виде `имя=значение`. Например, в процедуре `GetPhones` параметр называется `@Famil`. Это значит, что мы можем вызвать процедуру следующим образом:

```
EXECUTE GetPhones @Famil='ВАСИЛЬКОВ'
```

В этом примере после имени процедуры мы пишем имя параметра и присваиваем ему значение знаком равенства.

3.2.5. Преимущества хранимых процедур

В коде процедуры вы можете использовать практически любые объекты базы данных MS SQL Server, а именно: представления, таблицы, функции, определенные пользователем, другие процедуры, а также временные таблицы. Если процедура создает временную локальную таблицу, то она существует только во время выполнения и невидима после завершения выполнения.

Хранимые процедуры предоставляют множество преимуществ, среди которых можно выделить следующие:

- разделение кода — во время решения различных задач очень часто приходится выполнять одни и те же действия. Чтобы не писать один и тот же код в разных задачах, его можно вынести в отдельную процедуру;
- разделение логики приложений с другими, таким образом гарантируется совместимый доступ и модификация данных;
- если процедуры поддерживают все бизнес-функции, которые необходимы пользователю, он не нуждается в прямом доступе к таблицам. Все можно делать через процедуры, которые могут выступать дополнительным гарантом целостности данных и безопасности базы данных;
- предоставление механизма защиты. Пользователи могут получать право на выполнение процедуры, даже если у них нет права на использование представления или таблицы, на которую ссылается процедура;
- повышение производительности за счет выполнения нескольких задач в виде набора операторов Transact-SQL и хранения плана выполнения в кэше.

Помимо этого, процедура обладает всеми преимуществами, которыми обладают представления, например, уменьшает сетевой трафик. Пользователи могут выполнять комплекс операций, посылая только один запрос, что уменьшает количество запросов между клиентом и сервером.

3.2.6. Практика создания и использования процедур

Давайте создадим процедуру, которая будет принимать несколько параметров и при этом использовать логику. В нашей таблице товаров нет ограничения уникальности на сочетание полей `название товара` и `дата`, а ведь, по идее, оно должно быть. Если в таблице есть две записи для товара, купленного

в один день, то результат получается некрасивым. Такие покупки должны объединяться, ведь у нас есть поле для хранения количества.

Проблему можно решить и без добавления индекса уникальности, хотя в реальном приложении я бы создал. Будем считать, что индекс не создается только для того, чтобы администратор мог добавлять дубликаты покупок. Пользователям можно разрешить добавление записей только через процедуру.

Итак, создадим такую процедуру (листинг 3.1).

Листинг 3.1. Процедура добавления товаров

```
CREATE PROCEDURE AddGoods
 @Date datetime,
 @Name varchar(50),
 @Cost money,
 @Number integer
AS
 IF EXISTS (SELECT *
   FROM Товары
   WHERE Дата=@Date
    AND [Название товара]=@Name)
  PRINT 'Товар уже существует'
 ELSE
  BEGIN
   INSERT INTO Товары
   VALUES (@Date, @Name, @Cost, @Number)
   PRINT 'Товар добавлен'
  END
```

В качестве параметров передаются значения всех полей. Внутри процедуры делаем запрос на выборку строки с переданной в качестве параметра датой и названием товара. Если запрос вернет хоть одну строку (а больше и не должен, потому что это уже дубликат), то выводим на экран сообщение. Иначе добавляем в таблицу запись с указанными параметрами.

Для выполнения процедуры выполним следующий запрос:

```
EXEC AddGoods '01.01.2003', 'Сок', 23, 1
```

Значения всех полей перечислены в том же порядке, в котором они перечислены при создании процедуры. Попробуйте выполнить добавление одного и того же товара дважды. В ответ на это вы должны увидеть сообщение о том, что товар уже существует.

Если вы хотите передать параметры в другом порядке, то необходимо указывать их имена. В следующем примере мы передаем сначала значение количества товаров:

```
EXEC AddGoods @Number=1, @Date='01.01.2003', @Name='Сок', @Cost=23
```

Благодаря явному указанию имен порядок может быть любым.

Процедуры могут быть вложенными (одна процедура может вызывать другую). Вложенные процедуры должны удовлетворять следующим условиям:

- процедуры могут быть вложены до 32 уровней. Если более 32 уровней, то происходит ошибка;
- текущей уровень вложенности хранится в системной переменной `@@nestlevel`;
- если первая процедура вызывает вторую, то вторая может получить доступ ко всем объектам первой, включая временные таблицы, потому что они в этот момент существуют;
- встроенные процедуры могут быть рекурсивными. Например, если процедура 1 вызвала процедуру 2, то процедура 2 может вызвать первую.

Давайте рассмотрим вызов пользовательской процедуры из другой. Для этого создадим следующий вариант процедуры добавления товара:

```
CREATE PROCEDURE AddGoods1
 @Date datetime,
 @Name varchar(50),
 @Cost money,
 @Number integer
AS
 IF @Date>GETDATE()
  PRINT 'Дата больше текущей'
 ELSE
  EXEC AddGoods @Date, @Name, @Cost, @Number
```

В этом примере создается процедура `AddGoods1`, в которой есть проверка даты. Если дата приобретения товара больше текущей, то выводится соответствующее сообщение. Если дата корректна, то будет вызвана уже существующая процедура `AddGoods`.

Желательно, чтобы права на объекты, которые используются в процедуре, и сама процедура принадлежали одному и тому же пользователю. Для исключения ситуации, когда владелец процедуры и владелец таблицы, на которую ссылается процедура, различны, все объекты должны принадлежать `dbo`.

Я уже не раз говорил об этом и напоминаю еще раз — без особой надобности не указывайте владельцев. Лучше всего будет, если объекты будут принадлежать пользователю dbo.

Старайтесь создавать процедуры так, чтобы они выполняли по одной задаче. Дело в том, что какая-нибудь из задач может понадобиться в сочетании с другими. Например, если бы мы добавили в процедуру AddGoods проверку даты, которую сделали в процедуре AddGoods1, то нельзя было бы добавить товар с датой более текущей. А т. к. у нас каждая процедура выполняет свою задачу, можно позволить пользователям выполнять более защищенную процедуру AddGoods1, а администраторы будут иметь возможность работы с процедурой AddGoods и при особой надобности добавлять товары с любой датой.

Вы можете создавать процедуры, которые будут работать как системные. Для этого вы должны подключиться к серверу под учетной записью администратора и использовать базу данных master, чтобы процедуры сохранялись в этой базе. В этом случае такие процедуры будут доступны из любой базы данных.

Процедуры могут и не выполнять каких-либо запросов из базы данных. Например, следующая пользовательская процедура просто возвращает текстовую строку:

```
CREATE PROCEDURE TestData
AS
 SELECT 'Тестовая строка'
```

После выполнения команды EXEC TestData на экране появится таблица с одной строкой. Но процедура все же выполняет запрос SELECT, а ведь можно обойтись и без него, если достаточно только вывести на экран строку. Для этого можно воспользоваться оператором PRINT:

```
CREATE PROCEDURE PrintData
AS
 PRINT 'Тестовая строка'
```

3.2.7. Изменение процедур

Для изменения процедуры используйте оператор ALTER PROCEDURE. Сервер заменяет существующее описание процедуры тем, что указано в ALTER PROCEDURE. Настоятельно не рекомендую изменять системные процедуры напрямую. Вместо этого создавайте свой собственный вариант, копируйте в нее операторы из существующей процедуры и после этого делайте необходимые изменения.

Если вы хотите изменить процедуру, которая была создана с какими-нибудь опциями, например, `WITH ENCRYPTION`, вы должны включить их в число опций команды `ALTER PROCEDURE` для сохранения функциональности, которую предоставляет опция.

Во время выполнения оператора `ALTER PROCEDURE` изменяется только одна процедура. Если она ссылается на другие, то они не изменяются.

Оператор `ALTER PROCEDURE` в общем виде выглядит следующим образом:

```
ALTER PROC [ EDURE ] procedure_name [ ; number ]
    [ { @parameter data_type }
        [ VARYING ] [ = default ] [ OUTPUT ]
    ] [ ,...n ]
[ WITH
    { RECOMPILE | ENCRYPTION
        | RECOMPILE , ENCRYPTION
    }
]
[ FOR REPLICATION ]
AS
    sql_statement [ ...n ]
```

Следующий пример изменяет процедуру `AddGoods1`:

```
ALTER PROCEDURE AddGoods1
 @Date datetime,
 @Name varchar(50),
 @Cost money,
 @Number integer = 1
AS
 IF @Date>GETDATE()
  PRINT 'Дата больше текущей'
 ELSE
  EXEC AddGoods @Date, @Name, @Cost, @Number
```

Изменения произошли в последнем параметре — `@Number`. Я установил для него значение по умолчанию, равное 1. Теперь при вызове можно указывать только три значения. Если количество не указано, то будет использоваться значение 1.

```
EXEC AddGoods1 '01.01.2005', 'Шоколад', 25
```

3.2.8. Использование процедур при вставке данных

Оператор INSERT может заполнять локальную таблицу результирующим набором, который возвращается из локальной или удаленной процедуры. Сервер SQL заполняет таблицу данными, которые возвращаются оператором SELECT в процедуре. Таблица должна существовать, и типы данных должны совпадать.

Рассмотрим эту возможность. Для начала создадим процедуру, которая будет выбирать записи с указанной датой:

```
CREATE PROCEDURE GetGoods
 @Date datetime
AS
 SELECT *
 FROM Товары
 WHERE Дата=@Date
```

Следующий пример использует процедуру для вставки данных в таблицу товаров:

```
INSERT INTO Товары
EXEC GetGoods '01.01.2005'
```

Данный запрос вставляет в таблицу товаров результат выполнения процедуры GetGoods. Количество и типы полей должны совпадать, а у нас они будут совпадать, потому что процедура выбирает данные из той же таблицы товаров за указанную дату.

Теперь, если вы просмотрите таблицу товаров, то увидите, что данные за 1 января 2005 года в таблице содержатся дважды. Именно на эту дату процедура выбирала данные, и они были вставлены в таблицу товаров. Чтобы лучше было видно двойные записи, отсортируйте их по дате и названию:

```
SELECT *
FROM Товары
ORDER BY Дата, [Название товара]
```

3.2.9. Опции

Теперь посмотрим, какие дополнительные параметры можно использовать во время создания процедуры. Таких параметров два:

- RECOMPILE — указывает на то, что MS SQL Server не должен сохранять план выполнения, компиляция будет происходить при каждом выполнении;

- ENCRYPTION — запись в таблице syscomments с текстом процедуры должна шифроваться.

Посмотрим, как можно использовать шифрование:

```
CREATE PROCEDURE GetPhones
WITH ENCRYPTION
AS
SELECT pl.vcFamil, pl.vcName, pl.vcSurName,
  dDateBirthDay, vcPhoneNumber
FROM tbPeoples pl, tbPhoneNumbers pn
WHERE pn.idPeoples=*pl.idPeoples
```

Опция WITH ENCRYPTION указывается после всех параметров процедуры, но до ключевого слова AS.

Теперь посмотрим, что сервер сохранил в своих системных таблицах. Для этого выполните следующий запрос:

```
SELECT *
FROM sysobjects so, syscomments sc
WHERE name='GetPhones'
 AND so.id=sc.id
```

В колонке text вы увидите только бессмысленные символы. Для любой другой процедуры здесь будет ее текст, а для зашифрованной будут данные, которые не несут полезной информации.

Давайте посмотрим на запрос, который мы использовали для получения информации о процедуре. Здесь у нас выбираются данные из двух таблиц sysobjects и syscomments. В первой таблице находятся имена всех объектов базы данных, а в таблице syscomments находятся параметры объекта. Для хранимой процедуры здесь можно увидеть текст самой процедуры в поле text, если он не зашифрован.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter3\procs.sql на компакт-диске, прилагаемом к книге.

3.3. Хранимые функции

С SQL Server вы можете создавать ваши собственные функции, добавляющие и расширяющие функции, предоставляемые системой. Функции могут получать некоторое количество параметров (возможно, ни одного) и возвращать

скалярное значение или таблицу. Входные параметры могут быть любого типа, исключая timestamp, cursor, table.

Сервер SQL поддерживает три типа функций, определенных пользователем:

- скалярная функция — похожа на встроенную функцию;
- функция, возвращающая таблицу, — возвращает результат единичного оператора SELECT. Она похожа на представление, но имеет большую гибкость благодаря использованию параметров и расширяет возможности индексированного представления;
- многооператорная функция — возвращает таблицу, созданную одним или несколькими операторами Transact-SQL, чем напоминает хранимые процедуры. В отличие от процедур, на такие функции можно ссылаться в WHERE как на представления.

3.3.1. Создание функции

Создание функций очень похоже на создание процедур и представлений. Недаром мы рассматриваем все эти темы в одной главе. Для создания функции используется оператор CREATE FUNCTION. Рассмотрим все три типа объявления.

- Скалярная функция:

```
CREATE  FUNCTION [ owner_name. ] function_name
 ( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] }
   [ ,...n ] ] )

RETURNS scalar_return_data_type

[ WITH < function_option> [ [,] ...n] ]

[ AS ]

BEGIN
    function_body
    RETURN scalar_expression
END
```

- Функция, возвращающая таблицу:

```
CREATE FUNCTION [ owner_name. ] function_name
 ( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] }
```

```
    [ ,...n ] ] )

RETURNS TABLE

[ WITH < function_option > [ [,] ...n ] ]

[ AS ]

RETURN [ ( ] select-stmt [ ) ]
```

- Многооператорные функции:

```
CREATE FUNCTION [ owner_name. ] function_name
 ( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] }
   [ ,...n ] ] )

RETURNS @return_variable TABLE < table_type_definition >

[ WITH < function_option > [ [,] ...n ] ]

[ AS ]

BEGIN
    function_body
    RETURN
END

< function_option > ::=
    { ENCRYPTION | SCHEMABINDING }

< table_type_definition > :: =
    ( { column_definition | table_constraint } [ ,...n ] )
```

3.3.2. Скалярные функции

Давайте для примера создадим функцию, которая будет возвращать скалярное значение. Например, результат умножения цены на количество указанного товара. Товар будет идентифицироваться по названию и дате, ведь мы договорились, что сочетание этих полей дает уникальность. Но будьте осторожны при тестировании запроса, если в *разд. 3.2.8* вы выполнили

запрос на изменение данных и создали дубликаты покупок за 1 января 2005 года.

Создадим скалярную функцию:

```
CREATE FUNCTION GetSumm
 (@name varchar(50), @date datetime)
RETURNS numeric(10,2)
 BEGIN
  DECLARE @Summ numeric(10,2)
  SELECT @Summ = Цена*Количество
  FROM Товары
  WHERE [Название товара]=@name
    AND Дата=@date;
  RETURN @Summ
 END
```

После оператора `CREATE FUNCTION` мы указываем имя функции. Далее в круглых скобках через запятую перечисляются необходимые параметры. В этом объявление функции отличается от объявления процедуры.

Далее указывается ключевое слово `RETURNS`, за которым идет описание типа возвращаемого значения. Для скалярной функции это могут быть любые типы (строки, числа, даты и т. д.).

Код, который должна выполнять функция, пишется между ключевыми словами `BEGIN` (начало) и `END` (конец). В коде можно использовать любые операторы Transact-SQL, которые мы изучали ранее. Итак, объявление нашей функции в упрощенном виде можно описать следующим образом:

```
CREATE FUNCTION GetSumm
 (@name varchar(50), @date datetime)
RETURNS numeric(10,2)
 BEGIN
  -- Код функции
 END
```

Между ключевыми словами `BEGIN` и `END` выполняется следующий код:

```
-- Объявление переменной
DECLARE @Summ numeric(10,2)

-- Выполнение запроса на выборку суммы
SELECT @Summ = Цена*Количество
FROM Товары
```

```
WHERE [Название товара]=@name
  AND Дата=@date;

-- Возврат результата
RETURN @Summ
```

В первой строчке объявляется переменная `@Summ`. Она нужна для хранения промежуточного результата расчетов. Далее выполняется запрос `SELECT`, в котором происходит поиск строки по дате и названию товара в таблице товаров. В найденной строке перемножаются значения полей цены и количества, и результат записывается в переменную `@Summ`.

Обратите внимание, что в конце запроса стоит точка с запятой (;). Каждый запрос должен заканчиваться этим символом, но в большинстве примеров мы этим пренебрегали, но в функции отсутствие символа ";" может привести к ошибке.

В последней строчке возвращаем результат. Для этого нужно написать ключевое слово `RETURN`, после которого пишется возвращаемое значение или переменная. В данном случае возвращаться будет содержимое переменной `@Summ`.

Так как функция скалярная, то и возвращаемое значение должно быть скалярным и при этом соответствовать типу, описанному после ключевого слова `RETURNS`.

3.3.3. Использование функций

Как выполнить такую функцию? Да так же, как и многие другие системные функции (например, `GETDATE()`). Следующий пример использует функцию в операторе `SELECT`:

```
SELECT dbo.GetSumm('Картофель', '03.03.2005')
```

Здесь оператор `SELECT` возвращает результат выполнения функции `GetSumm()`. Функция принадлежит пользователю `dbo`, поэтому перед именем я указал владельца. После имени в скобках должны быть перечислены параметры в том же порядке, что и при объявлении функции. В данном примере я запрашиваю затраты на картофель, купленный 3.3.2005.

Выполните следующий запрос и убедитесь, что он вернул тот же результат, что и созданная нами функция:

```
SELECT Цена*Количество
FROM Товары
WHERE [Название товара]='Картофель'
  AND Дата='03.03.2005'
```

Функции можно использовать не только в операторе SELECT, но и напрямую, присваивая значение переменной. Например:

```
DECLARE @Summ numeric(10,2)
SET @Summ=dbo.GetSumm('Картофель', '03.03.2005')
PRINT @Summ
```

Здесь мы объявили переменную @Summ типа numeric(10,2). Значение именно такого типа возвращает функция. В следующей строчке переменной присваивается результат выполнения функции GetSumm() с помощью SET.

Давайте посмотрим, что произойдет, если передать функции такие параметры, при которых запрос функции вернет более одной строки. В нашей таблице товаров сочетание даты и названия не дает уникальности, потому что мы ее нарушили. Первичного ключа в таблице также нет, поэтому в результате четыре строки имеют свои точные копии. Это нарушает правило уникальности строк в реляционных базах, но очень наглядно показывает, что в реальной жизни нарушать его нельзя.

Итак, в таблице есть две записи о покупке хлеба за 1.1.2005. Попробуем запросить у функции сумму:

```
SELECT dbo.GetSumm('Хлеб', '01.01.2005')
```

Результатом будет только одно число, хотя строки две. А какую строку из двух вернул сервер? Никто точно сказать не может, потому что они совершенно одинаковы. Скорее всего, сервер вернул первую из строк.

3.3.4. Функция, возвращающая таблицу

В качестве примера создадим функцию, которая будет возвращать таблицу товаров, и для каждой строки рассчитаем произведение колонок количества и цены:

```
CREATE FUNCTION GetPrice()
RETURNS TABLE
AS
 RETURN
 (
  SELECT Дата, [Название товара], Цена,
    Количество, Цена*Количество AS Сумма
  FROM Товары
 )
```

Начало функции такое же, как у скалярной, — указываем оператор CREATE FUNCTION и имя функции. Я специально создал эту функцию без параметров, чтобы вы увидели, как это делается. Несмотря на то, что параметров нет,

после имени должны идти пустые круглые скобки. Если не указать скобок, то сервер выдаст сообщение об ошибке, и функция не будет создана.

Разница есть и в секции `RETURNS`, после которой указывается тип `TABLE`, что говорит о необходимости вернуть таблицу. Далее идут ключевые слова `AS` и `RETURN`, а затем возвращаемое значение. Для функции данного типа в секции `RETURN` нужно в скобках указать запрос, результат которого и будет возвращаться функцией.

У всех полей запроса должны быть имена. Если у одного из полей имени нет, то результатом выполнения оператора `CREATE FUNCTION` будет ошибка. В нашем примере последнее поле является результатом перемножения значений полей `Цена` и `Количество`, а у такого поля нет имени, поэтому мы его задаем с помощью ключевого слова `AS`.

Посмотрим, как использовать такую функцию с помощью оператора `SELECT`:

```
SELECT *
FROM GetPrice()
```

Так как мы используем простой оператор `SELECT`, то мы можем и ограничивать вывод определенными строками с помощью ограничений в секции `WHERE`. В следующем примере выбираем из результата функции только те строки, в которых поле `Количество` содержит значение 1:

```
SELECT * FROM GetPrice()
WHERE Количество=1
```

Функция возвращает в качестве результата таблицу, которую вы можете использовать как любую другую таблицу базы данных. Рассмотрим пример. Для начала создадим функцию, которая будет возвращать идентификаторы работников из таблицы `tbPeoples`, объединенные в одно поле:

```
CREATE FUNCTION GetPeoples()
RETURNS TABLE
AS
 RETURN
  (
   SELECT idPeoples, vcFamil+' '+vcName+' '+vcSurName AS FIO
   FROM tbPeoples
  )
```

Функция возвращает нам идентификатор строки, с помощью которого мы легко можем связать результат с таблицей номеров телефонов. Попробуем сделать это с помощью простого SQL-запроса:

```
SELECT *
FROM GetPeoples() p, tbPhoneNumbers pn
WHERE p.idPeoples=pn.idPeoples
```

Как видите, функции, возвращающие таблицы, очень удобны. Они в большей степени, чем процедуры, похожи на представления, но при этом позволяют принимать параметры. Следовательно, можно сделать так, чтобы сама функция возвращала нам только нужные записи. Для представления приходится выполнять два запроса SELECT, что для большой таблицы довольно накладно. Функция же может сразу вернуть только то, что нужно.

Рассмотрим пример. Функция GetPeoples возвращает все строки таблицы. Чтобы получить только нужную фамилию, надо написать такой запрос:

```
SELECT *
FROM GetPeoples()
WHERE FIO LIKE 'ПОЧЕЧКИН%'
```

В этом случае будут выполняться два запроса: этот и еще один внутри функции. Но если передавать фамилию в качестве параметра в функцию и там сделать секцию WHERE, то можно обойтись и одним запросом SELECT:

```
CREATE FUNCTION GetPeoples1(@Famil varchar(50))
RETURNS TABLE
AS
 RETURN
 (
  SELECT idPeoples, vcFamil+' '+vcName+' '+vcSurName AS FIO
  FROM tbPeoples
  WHERE vcFamil=@Famil
 )
```

3.3.5. Многооператорная функция, возвращающая таблицу

Все функции, созданные в *разд. 3.3.4*, могут возвращать таблицу, сгенерированную только одним оператором SQL. А как же тогда обеспечить возможность выполнения нескольких операций? Например, неплохо было бы выполнять дополнительные проверки входных параметров для обеспечения безопасности. Проверки лишними не бывают, особенно проверки входных данных, тем более, когда эти данные указываются пользователем.

Следующий пример показывает, как создать функцию, которая вернет в качестве результата таблицу, и при этом в теле функции могут выполняться несколько операторов:

```
CREATE FUNCTION имя (параметры)
RETURNS  имя_переменной TABLE
```

```
  (описание вида таблицы,
   в которой будет представлен результат)
AS
 BEGIN
  Выполнение любого количества операций
  RETURN
 END
```

Это упрощенный вид создания процедуры. Более полный вид мы рассматривали в начале главы, а сейчас я упростил объявление, чтобы проще было его разбирать.

Объявление больше похоже на создание скалярных функций. Первая строчка без изменений. В секции `RETURNS` объявляется переменная, которая имеет тип `TABLE`. После этого в скобках нужно описать поля результирующей таблицы. После ключевого слова `AS` идет пара операторов `BEGIN` и `END`, между которыми может выполняться какое угодно количество операций. Выполнение операций заканчивается ключевым словом `RETURN`.

Вот тут есть одно отличие от скалярных функций — после `RETURN` мы указывали имя переменной, значение которой должно стать результатом. В данном случае ничего указывать не надо. Мы уже объявили переменную в секции `RETURNS` и описали формат этой переменной. В теле функции мы можем и должны наполнить эту переменную значениями и именно это попадет в результат.

Теперь рассмотрим пример создания функции:

```
CREATE FUNCTION getFIO ()
RETURNS  @ret TABLE
  (idPeoples int primary key,
   vcFIO varchar(100))
AS
 BEGIN
  INSERT @ret
  SELECT idPeoples, vcFamil+' '+vcName+' '+vcSurName
  FROM tbPeoples;

  RETURN
 END
```

В данном примере в качестве результата объявлена переменная `@ret`, которая является таблицей из двух полей: `idPeoples` типа `int` и `vcFIO` типа `varchar` длиной в 50 символов. В теле функции в эту таблицу записываются значения

из таблицы tbPeoples, и выполняется оператор RETURN, завершающий выполнение функции.

В использовании такая функция ничем не отличается от рассмотренных ранее. Например, следующий запрос выбирает все данные, которые возвращает функция:

```
SELECT *
FROM GetFIO()
```

3.3.6. Опции функций

При создании функций могут использоваться следующие опции SCHEMABINDING (привязать к схеме) и/или ENCRYPTION (шифровать текст функции). Если вторая опция нам уже известна по представлениям и процедурам (позволяет шифровать исходный код функции в системных таблицах), то вторая встречается впервые, но при этом предоставляет удобное средство защиты данных.

Если функция создана с опцией SCHEMABINDING, то объекты базы данных, на которые ссылается функция, не могут быть изменены (с помощью оператора ALTER) или удалены (с помощью оператора DROP). Например, следующая функция использует таблицу tbPeoples, и при этом задействована опция SCHEMABINDING:

```
CREATE FUNCTION GetPeoples2(@Famil varchar(50))
RETURNS TABLE
WITH SCHEMABINDING
AS
 RETURN
 (
  SELECT idPeoples, vcFamil+' '+vcName+' '+vcSurName AS FIO
  FROM dbo.tbPeoples
  WHERE vcFamil=@Famil
 )
```

Функция может быть связанной со схемой, только если следующие ограничения истинны:

- все функции, объявленные пользователем, и представления, на которые ссылается функция, также связаны со схемой с помощью опции SCHEMABINDING;
- объекты, на которые ссылается функция, нужно именовать так: owner.objectname. При создании функции GetPeoples2 ссылка на таблицу указана именно в таком формате — dbo.tbPeoples;

- функция и объекты должны быть расположены в одной базе данных;
- пользователь, который создает функцию, имеет право доступа ко всем объектам, на которые ссылается функция.

Создайте функцию и попробуйте после этого удалить таблицу `tbPeoples`.

```
DROP TABLE tbPeoples
```

В ответ на это сервер выдаст сообщение об ошибке, что объект не может быть удален из-за присутствия внешнего ключа. Даже если избавиться от ключа, удаление будет невозможно, потому что на таблицу ссылается функция, привязанная к схеме.

Чтобы увидеть сообщение без удаления ключа, добавим к таблице колонку, а потом попробуем ее удалить:

```
-- Добавим колонку
ALTER TABLE dbo.tbPeoples
ADD
 vcTemp VARCHAR(30) NOT NULL default ''

-- Попробуем ее удалить
ALTER TABLE dbo.tbPeoples
DROP COLUMN vcTemp
```

Создание пройдет успешно, а вот во время удаления произойдет ошибка с сообщением о том, что существует ограничение, которое зависит от колонки. Мы же не создавали никаких ограничений, а просто добавили колонку и попытались ее удалить. Ограничение уже давно существует, но не на отдельную колонку, а на все колонки таблицы, и это ограничение создано функцией `GetPeoples2`, которая связана со схемой.

3.3.7. Изменение функций

Вы можете изменять функцию с помощью оператора `ALTER FUNCTION`. Общий вид оператора для каждого варианта функции свой. Давайте рассмотрим каждый из них.

- Общий вид команды изменения скалярной функции:

```
ALTER FUNCTION [ owner_name. ] function_name
    ( [ { @parameter_name scalar_parameter_data_type [ = default ] }
      [ ,...n ] ] )

RETURNS scalar_return_data_type
```

```
[ WITH < function_option> [,...n] ]

[ AS ]

BEGIN
    function_body
    RETURN scalar_expression
END
```

- Общий вид изменения функции, возвращающей таблицу:

```
ALTER FUNCTION [ owner_name. ] function_name
    ( [ { @parameter_name scalar_parameter_data_type [ = default ] }
      [ ,...n ] ] )

RETURNS TABLE

[ WITH < function_option > [ ,...n ] ]

[ AS ]

RETURN [ ( ] select-stmt [ ) ]
```

- Общий вид команды изменения функции с множеством операторов, возвращающей таблицу:

```
ALTER FUNCTION [ owner_name. ] function_name
    ( [ { @parameter_name scalar_parameter_data_type [ = default ] }
      [ ,...n ] ] )

RETURNS @return_variable TABLE < table_type_definition >

[ WITH < function_option > [ ,...n ] ]

[ AS ]

BEGIN
    function_body
    RETURN
END

< function_option > ::=
```

```
        { ENCRYPTION | SCHEMABINDING }

    < table_type_definition > :: =
        ( { column_definition | table_constraint } [ ,...n ] )
```

Следующий пример показывает упрощенный вариант команды, изменяющей функцию:

```
ALTER FUNCTION dbo.tbPeoples
AS
 -- Новое тело функции
```

3.3.8. Удаление функций

Для удаления функции используется оператор `DROP FUNCTION`:

```
DROP FUNCTION dbo.GetPeoples2
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter3\func.sql на компакт-диске, прилагаемом к книге.

3.4. Триггеры

Триггер — это специальный вид хранимых процедур, которые выполняются на определенные события в таблице. Триггер связывается с определенной таблицей и чаще всего выполняет защитную роль для данных. В *разд. 1.5* мы говорили о целостности данных и упомянули, что триггер является наиболее мощным средством защиты.

Существуют три события, на которые могут реагировать триггеры, — добавление, изменение и вставка данных, т. е. любые попытки повлиять на данные. Когда происходит попытка вставки, обновления или удаления данных в таблице, и для такого действия над этой таблицей объявлен триггер, он вызывается автоматически. Его нельзя обойти. В отличие от встроенных процедур, триггеры не могут вызываться напрямую и не принимают параметры.

Триггер — лучшее средство для обеспечения низкоуровневой целостности данных с единственным недостатком: он работает медленнее ограничений. Основное преимущество триггеров заключается в том, что они могут содержать комплексно выполняемую логику. Они могут:

- делать каскадные изменения зависимых таблиц в базе данных, обеспечивая более комплексную целостность данных, чем ограничение `CHECK`;

- объявлять индивидуальные сообщения об ошибках;
- содержать ненормализованные данные;
- сравнивать состояние данных до и после изменения.

Это лишь основные их достоинства, а к концу раздела вы убедитесь, что их намного больше.

Вы можете использовать триггеры для каскадного изменения или удаления в зависимых таблицах базы данных. Например, триггер на удаление данных из таблицы `tpPhoneType` может удалять соответствующие строки в других таблицах, которые имеют строки, связанные с удаляемым идентификатором типа телефона. Если этого не сделать, то связь нарушается и целостность данных считается разрушенной.

В отличие от ограничения `CHECK`, триггеры могут ссылаться на поля в другой таблице. Например, вы можете поместить триггер на добавление данных для таблицы `tbPosition`, который будет искать главную должность для добавляемой и проверять наличие работника с соответствующей должностью.

3.4.1. Создание триггера

Для создания триггеров используйте оператор `CREATE TRIGGER`. В операторе указывается таблица, для которой объявляется триггер, событие, для которого триггер выполняется, и индивидуальные инструкции для триггера. В общем команда показана в листинге 3.2.

Листинг 3.2. Общий вид команды `CREATE TRIGGER`

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{
    { { FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE ] }
        [ WITH APPEND ]
        [ NOT FOR REPLICATION ]
        AS
        [ { IF UPDATE ( column )
            [ { AND | OR } UPDATE ( column ) ]
                [ ...n ]
        | IF (COLUMNS_UPDATED() {bitwise_operator} updated_bitmask)
                { comparison_operator } column_bitmask [ ...n ]
```

```
        } ]
        sql_statement [ ...n ]
    }
}
```

Прежде чем перейти к реальному примеру, сделаю два замечания. Когда вы создаете триггер, информация о триггере вставляется в системные таблицы `sysobjects` и `syscomments`. Если триггер создается с таким же именем, как и существующий, новый триггер перезаписывает существующий. Сервер SQL не поддерживает добавления триггеров, объявленных пользователем на системные таблицы, поэтому вы не можете создавать их для системных таблиц.

Сервер SQL не позволяет использовать следующие операторы в теле триггера:

- `ALTER DATABASE`
- `CREATE DATABASE`
- `DISK INIT`
- `DISK RESIZE`
- `DROP DATABASE`
- `LOAD DATABASE`
- `LOAD LOG`
- `RECONFIGURE`
- `RESTORE DATABASE`
- `RESTORE LOG`

Проще говоря, нельзя изменять структуру базы данных.

3.4.2. Откат изменений в триггере

Объявление триггера может содержать оператор `ROLLBACK TRANSACTION`, даже если не существует соответствующего `BEGIN TRANSACTION`. Как мы уже говорили, для любого изменения SQL-сервер требует транзакции. Если она не указана явно, то создается неявная транзакция. Если выполняется оператор `ROLLBACK TRANSACTION`, то все изменения в триггере и изменения, которые стали причиной срабатывания триггера, откатываются.

При использовании отката изменений вы должны учитывать следующее:

- если срабатывает оператор `ROLLBACK TRANSACTION`, содержимое транзакции откатывается. Если есть операторы, следующие за `ROLLBACK TRANSACTION`, операторы выполняются. Это может быть необязательным при использовании команды `RETURN`;

- если триггер откатывает транзакцию, определенную пользователем, то она откатывается полностью. Если триггер сработал на выполнение модуля, для модуля команды также отменяются. Последующие операторы модуля не выполняются;
- вы должны минимизировать использование `ROLLBACK TRANSACTION` в коде триггера. Откат транзакции создает дополнительное задание, потому что все задания, которые не были закончены на данный момент в транзакции, окажутся незавершенными. Это будет негативно сказываться на производительности. Запускайте транзакцию после того, как все проверено, чтобы не пришлось ничего откатывать в триггере.

Давайте продолжим изучение триггеров на практике. Для примера создадим триггер, который будет выполнять только откат транзакции:

```
CREATE TRIGGER u_tbPeoples ON dbo.tbPeoples
FOR UPDATE
AS
 ROLLBACK TRANSACTION
```

Обсудим этот код подробно. В первой строчке после оператора `CREATE TRIGGER` стоит название. При именовании триггеров я придерживаюсь следующего правила:

- имя начинается с определенного префикса: `u` (update — обновление), `i` (insert — вставка) или `d` (delete — удаление). По этим буквам вы легко можете определить, на какие действия срабатывает триггер;
- после подчеркивания идет имя таблицы, для которой создается триггер.

Далее идет ключевое слово `ON` и имя таблицы, для которой создается триггер.

Во второй строчке идет ключевое слово `FOR` и событие, на которое срабатывает триггер. В данном примере указано действие `UPDATE`, т. е. обновление. И, наконец, после ключевого слова `AS` идет тело триггера, т. е. команды, которые должны выполняться. В данном примере выполняется только одна команда — `ROLLBACK TRANSACTION`, т. е. откат.

Теперь попробуем изменить данные в таблице `tbPeoples`, чтобы сработал триггер:

```
UPDATE tbPeoples
SET vcFamil='dsfg'
```

В данном примере мы пытаемся изменить содержимое поля `vcFamil` для всех записей таблицы `tbPeoples`. Почему пытаемся? Да потому что при изменении срабатывает триггер с откатом транзакции. Выполните выборку данных, чтобы убедиться, что все данные на месте и не изменились:

```
SELECT *
FROM tbPeoples
```

Несмотря на то, что при обновлении данных мы не запускали транзакцию, оператор `ROLLBACK TRANSACTION` был выполнен без ошибок, и изменения отменились.

3.4.3. Изменение триггера

Если вы хотите изменить объявление существующего триггера, вы можете изменить его без удаления и воссоздания. Вы можете ссылаться в объявлении триггера на объекты, которые не существуют. Если во время создания объявления какой-то объект не существует, то вы увидите только предупреждение.

Для обновления триггера используется оператор `ALTER TRIGGER`. Общий вид оператора можно увидеть в листинге 3.3.

Листинг 3.3. Оператор обновления триггера

```
ALTER TRIGGER trigger_name
ON ( table | view )
[ WITH ENCRYPTION ]
{
    { ( FOR | AFTER | INSTEAD OF ) { [ DELETE ] [ , ]
       [ INSERT ] [ , ] [ UPDATE ] }
        [ NOT FOR REPLICATION ]
        AS
        sql_statement [ ...n ]
    }
    |
    { ( FOR | AFTER | INSTEAD OF ) { [ INSERT ] [ , ] [ UPDATE ] }
        [ NOT FOR REPLICATION ]
        AS
        { IF UPDATE ( column )
        [ { AND | OR } UPDATE ( column ) ]
        [ ...n ]
        |IF(COLUMNS_UPDATED() { bitwise_operator } updated_bitmask)
        { comparison_operator } column_bitmask [ ...n ]
        }
        sql_statement [ ...n ]
    }
}
```

Давайте изменим наш триггер `u_tbPeoples` так, чтобы он реагировал и на добавление записей. Для этого выполним следующий запрос:

```
ALTER TRIGGER u_tbPeoples ON dbo.tbPeoples
FOR UPDATE, INSERT
AS
 ROLLBACK TRANSACTION
```

Как видите, оператор обновления похож на создание триггера. Разница в том, что в первой строчке стоит оператор `ALTER TRIGGER`. Во второй строчке произошло изменение, и теперь триггер будет срабатывать не только на обновление (`UPDATE`), но и на добавление (`INSERT`).

Попробуйте добавить запись, и убедитесь после этого, что она не добавлена, ведь теперь триггер отката изменений срабатывает и на добавление записей. Пример добавления записи:

```
INSERT INTO tbPeoples(vcFamil)
VALUES('ПЕТЕЧКИН')
```

Вы можете включать и выключать определенный триггер или все триггеры на таблицу. Когда триггер отключен, он все еще существует в таблице, однако не выполняется на указанные события. Вы можете отключить триггер с помощью команды `ALTER TABLE`. В общем виде оператор выглядит следующим образом:

```
ALTER TABLE table
 {ENABLE | DISABLE} TRIGGER
 {ALL | trigger_name [,..n]}
```

Как видите, изменение касается непосредственно таблицы, а не триггера. Попробуем отключить ранее созданный триггер:

```
ALTER TABLE tbPeoples
DISABLE TRIGGER u_tbPeoples
```

В первой строчке мы пишем оператор `ALTER TABLE` и имя изменяемой таблицы. Во второй строчке нужно указать ключевое слово `DISABLE` (отключить) или `ENABLE` (включить) и ключевое слово `TRIGGER`. И, наконец, имя триггера.

Попробуйте теперь добавить запись в таблицу `tbPeoples`. На этот раз все пройдет успешно.

Вместо имени триггера можно указать ключевое слово `ALL`, которое требует воздействия на все триггеры указанной таблицы. В следующем примере мы включаем все триггеры:

```
ALTER TABLE tbPeoples
ENABLE TRIGGER ALL
```

3.4.4. Удаление триггера

Для удаления триггера вы можете воспользоваться оператором `DROP TRIGGER`. Он удаляется автоматически, когда связанная с ним таблица удаляется.

Пример удаления триггера:

```
DROP TRIGGER u_tbPeoples
```

Для выполнения этого действия вы должны обладать соответствующими правами. Как и в случае с процедурами, функциями и представлениями, вы можете удалять сразу несколько триггеров, указав их имена через запятую.

3.4.5. Как работают триггеры?

Здесь мы подробно рассмотрим, как работают различные типы триггеров. Для этого напишем множество примеров, максимально приближенных к реальным.

Оператор *INSERT*

Что происходит, когда срабатывает триггер добавления записей? Давайте рассмотрим выполняемые сервером шаги:

1. Пользователем выполняется оператор `INSERT` для добавления записей.
2. Сервер сохраняет информацию о запросе в журнале транзакций.
3. Вызывается триггер.
4. Происходит подтверждение изменений и физическое изменение данных.

Во время вызова триггера физического изменения в базе еще не произошло. В теле триггера вы можете увидеть добавляемые записи в виде таблицы `inserted`. Нет, такой таблицы в базе данных не существует, `inserted` — это логическая таблица, содержащая копии строк, которые должны быть вставлены в таблицу. Если говорить точнее, она содержит журнал активности оператора `INSERT`. Вы можете использовать данные из этой таблицы для определения вставляемых данных. Строки из таблицы `inserted` всегда дублируют одну или несколько строк таблицы триггера.

Вся активность по изменению данных записывается в журнал, но информация в журнале транзакций нечитаема. Однако таблица `inserted` позволяет вам определить изменения.

Таблица `inserted` всегда имеет такую же структуру, что и таблица, на которую установлен триггер.

Давайте запретим с помощью триггера добавление записей, в которых имя работника равно `Вася`. Пример такого триггера можно увидеть в листинге 3.4.

Листинг 3.4. Использование таблицы `inserted`

```
CREATE TRIGGER i_tbPeoples ON dbo.tbPeoples
FOR INSERT
AS
 DECLARE @Name varchar(50)
 SELECT @Name=vcName
 FROM inserted
 IF @Name='ВАСЯ'
  BEGIN
   PRINT 'ОШИБКА'
   ROLLBACK TRANSACTION
  END
```

В данном примере мы создаем триггер на добавление записей. Внутри триггера мы объявляем переменную `@Name` типа `varchar` длиной в 50 символов. В эту переменную мы сохраняем содержимое поля `vcName` таблицы `inserted`. Далее проверяем, если имя равно `Вася`, то сообщаем об ошибке и откатываем транзакцию. Иначе строка будет удачно добавлена.

Давайте для закрепления материала напишем триггер, который запретит нулевые значения для поля `vcName`. Код такого триггера можно увидеть в листинге 3.5.

Листинг 3.5. Запрет нулевых значений в поле с помощью триггера

```
CREATE TRIGGER i_tbPeoples ON dbo.tbPeoples
FOR INSERT
AS
 IF EXISTS (SELECT *
            FROM inserted
            WHERE vcName is NULL)
  BEGIN
   PRINT 'ОШИБКА, вы должны заполнить поле vcName'
   ROLLBACK TRANSACTION
  END
```

В этом примере мы проверяем, если в таблице `inserted` есть записи с нулевым значением поля `vcName`, то откатываем попытку добавления.

Оператор *DELETE*

Когда срабатывает триггер удаления, срабатывает примерно та же логика, что и при добавлении записей:

1. Пользователем выполняется оператор `DELETE` для добавления записей.
2. Сервер сохраняет информацию о запросе в журнале транзакций.
3. Вызывается триггер.
4. Происходит подтверждение изменений и физическое изменение данных.

Удаляемые строки помещаются в таблицу `deleted`, с помощью которой вы можете увидеть удаляемые строки. Это логическая таблица, которая ссылается на данные журнала оператора `DELETE`.

Вы должны учитывать:

- когда строки добавляются в таблицу `deleted`, они еще существуют в таблице базы данных;
- для таблицы `deleted` выделяется память, поэтому она всегда в кэше;
- триггер удаления не выполняется на операцию `TRUNCATE TABLE` (очистка таблицы), потому что эта операция не заносится в журнал и не удаляет строк.

Давайте попробуем создать триггер, который запретит удаление пользователя с определенным именем. Пример такого триггера можно увидеть в листинге 3.6.

Листинг 3.6. Пример запрета удаления с помощью триггера

```
CREATE TRIGGER d_tbPeoples ON dbo.tbPeoples
FOR DELETE
AS
 IF EXISTS (SELECT *
            FROM deleted
            WHERE vcName='рлр')
  BEGIN
   PRINT 'ОШИБКА, нельзя удалить этого пользователя'
   ROLLBACK TRANSACTION
  END
```

В этом примере мы проверяем, если в таблице `deleted` существует запись с именем `рлр`, то откатываем удаление. Добавьте в таблицу запись с именем `рлр` и попытайтесь ее удалить. В ответ вы должны увидеть сообщение об ошибке.

А что произойдет, если попытаться удалить несколько записей? В следующем примере удаляются две записи:

```
DELETE
FROM tbPeoples
WHERE vcName='рлр' or
     vcName='ВАСИЛИЙ'
```

Ни одна из них не будет удалена, даже несмотря на то, что запрет установлен только на имя `рлр`, а `Василий` не вызывает конфликтов в триггере. Отменяется вся транзакция.

Рассмотрим еще один пример, в котором запрещается удаление записи о генеральном директоре. Без триггера такое сделать невозможно:

```
CREATE TRIGGER d_tbPeoples ON dbo.tbPeoples
FOR DELETE
AS
 IF EXISTS (SELECT *
            FROM deleted
            WHERE idPosition=1)
  BEGIN
   PRINT 'ОШИБКА, нельзя удалить этого пользователя'
   ROLLBACK TRANSACTION
  END
```

В этом примере запрещается удаление записи, если поле `idPosition` равно 1. Попробуйте удалить такую запись:

```
DELETE
FROM tbPeoples
WHERE idPosition=1
```

Самое интересное, что вы увидите не сообщение триггера об ошибке, а ограничение внешнего ключа. У генерального директора есть номера телефонов, а запись нельзя удалять, если существует внешняя связь, иначе нарушится целостность. Значит, триггеры срабатывают после проверки всех ограничений `CHECK` и внешних ключей. Вполне логично, ведь ограничения работают быстрее, и желательно проверить сначала их. Если быстрая проверка даст отрицательный результат, зачем выполнять более сложные проверки в триггере?

Это относится не только к триггерам, срабатывающим на удаление, но и на изменение и вставку. Пример мы рассмотрели только сейчас, потому что в нем имеется интересный запрос, на котором удобно продемонстрировать порядок выполнения проверок.

Оператор *UPDATE*

Обновление происходит в два этапа — удаление и вставка. Нет, физически в базе данных происходит изменение, это триггер видит два этапа. Поэтому существующие строки помещаются в таблицу `deleted` (т. е. то, что было), а новые данные помещаются в таблицу `inserted`. Триггер может проверять эти таблицы для определения, какие строки и как могут измениться.

Вы можете объявить триггер для мониторинга обновления определенного поля с помощью указания опции `IF UPDATE`. Это позволяет триггеру изолировать активность определенной колонки. Когда обнаруживается обновление определенной колонки, триггер может выполнить определенные действия, такие как вывод сообщения о невозможности обновления колонки.

Давайте создадим триггер на таблицу `tbPeoples`, который будет выводить на экран сообщение, если изменяется поле `vcName`:

```
CREATE TRIGGER u_tbPeoples ON dbo.tbPeoples
FOR UPDATE
AS
 IF UPDATE (vcName)
   PRINT 'Я надеюсь, что вы правильно указали имя'
```

После оператора `IF UPDATE` в скобках указано поле, которое необходимо проверить. Если оно было изменено, то выполнится следующий за проверкой оператор. В данном случае это вывод на экран сообщения с помощью `PRINT`. Когда указанное поле не изменилось, то оператор, конечно же, не выполнится. Если нужно выполнить несколько операторов, то объедините их с помощью `BEGIN` и `END`.

Следующий запрос тестирует триггер:

```
UPDATE tbPeoples
SET vcName='ИВАНУШКА'
WHERE vcFamil='ПОЧЕЧКИН'
```

Убедитесь, что сообщение из триггера выводится на экран.

Теперь с помощью триггера попробуем запретить изменение полей, составляющих фамилию, имя и отчество (`vcFamil`, `vcName` и `vcSurName`). Если кто-то попытается изменить одно из этих полей, то выводим на экран сообщение о запрете и откатываем транзакцию:

```
CREATE TRIGGER u_tbPeoples ON dbo.tbPeoples
FOR UPDATE
AS
 IF UPDATE (vcName)
```

```
    OR UPDATE (vcFamil)
    OR UPDATE (vcSurname)
  BEGIN
   PRINT 'Нельзя изменять фамилию, имя и отчество'
   ROLLBACK TRANSACTION
  END
```

С помощью такого запроса легко увидеть, как проверять обновление сразу нескольких полей и выводить несколько операторов. Обратите внимание, что проверку делает именно оператор `UPDATE`, а не `IF UPDATE`. Я даже не знаю, почему разработчики SQL Server объединяют эти два оператора. Первый является логическим оператором, а второй — проверкой, было ли обновлено поле.

3.4.6. Триггер *INSTEAD OF*

Вы можете указать триггер `INSTEAD OF` для таблиц и представлений. Действия такого триггера выполняются вместо операторов, сгенерировавших триггер. Непонятно? Рассмотрим пример. Допустим, что у вас есть триггер `INSTEAD OF` на событие обновления таблицы. Если пользователь выполняет обновление, то выполняется триггер, но при этом оператор, запущенный пользователем, только генерирует событие. Реальное обновление данных должно происходить с помощью операторов триггера.

Каждая таблица или представление ограничены одним триггером `INSTEAD OF` на каждое событие. Вы не можете создавать триггеры `INSTEAD OF` на представление, у которого включена опция `CHECK OPTIONS`.

Как можно использовать `INSTEAD OF`? Допустим, что у нас есть представление, которое выбирает данные из двух таблиц. Как мы уже знаем, данные представления можно изменять, только если все они принадлежат одной таблице. Но с помощью триггера можно сделать обновление любого количества таблиц.

Давайте создадим представление, которое будет выбирать фамилию работника и название должности. Назовем это представление `Peoples`:

```
CREATE VIEW People
AS
 SELECT vcFamil, vcPositionName
 FROM tbPosition ps, tbPeoples pl
 WHERE ps.idPosition=pl.idPosition
```

Теперь создадим триггер `INSTEAD OF` на это представление, с помощью которого можно будет добавлять записи, и при этом они корректно будут прописываться, каждая в свою таблицу (листинг 3.7).

Листинг 3.7. Триггер INSTEAD OF для вставки данных

```
CREATE TRIGGER i_People ON dbo.People
INSTEAD OF INSERT
AS
 BEGIN
  -- Добавление должности
  INSERT INTO tbPosition (vcPositionName)
  SELECT vcPositionName
  FROM inserted i

  -- Добавление работника
  INSERT INTO tbPeoples (vcFamil, idPosition)
  SELECT vcFamil, idPosition
  FROM inserted i,tbPosition pn
  WHERE i.vcPositionName=pn.vcPositionName
 END
```

В этом примере все интересное начинается со второй строчки. Здесь указываются оператор INSTEAD OF и событие, на которое нужно реагировать. В данном случае в качестве события выступает вставка (INSERT).

В триггере мы выполняем два SQL-запроса: добавление должности работника и самого работника. Первый запрос весьма прост, потому что достаточно лишь выбрать все имена должностей из таблицы inserted и вставить их в таблицу tbPosition. А вот во втором запросе, помимо вставки фамилии работника, нужно найти должность и установить связь, иначе нет смысла затевать такие сложные манипуляции. Вот как я решаю эту проблему:

```
INSERT INTO tbPeoples (vcFamil, idPosition)
 SELECT vcFamil, idPosition
 FROM inserted i,tbPosition pn
 WHERE i.vcPositionName=pn.vcPositionName
```

Попробуйте выполнить следующий запрос на добавление записей в представление:

```
INSERT INTO People
VALUES('ИВАНУШКИН', 'Клерк')
```

Выполните следующий запрос и убедитесь, что новая запись добавлена:

```
SELECT * FROM People
```

При обновлении таблицы есть одна проблема — нужно связать обновляемые данные с существующими. Самым очевидным решением является запрос типа:

```
UPDATE tbPosition
 SET vcPositionName=i.vcPositionName
 FROM tbPosition pn, inserted i
 WHERE i.vcPositionName = pn.vcPositionName
```

Здесь мы связываем таблицу должностей с таблицей `inserted`. Но такой запрос никогда не будет выполнен. Почему? В таблице `inserted` находятся новые значения, а в таблице `tbPosition` еще старые, и названия должностей никогда не свяжутся. Если связать таблицу должностей с таблицей `deleted`, то записи свяжутся, но мы не будем знать новых значений, которые нужно занести в таблицу. Лучшим решением проблемы будет добавление ключевых полей в представление:

```
ALTER VIEW People
AS
 SELECT idPeoples, pl.idPosition, vcFamil, vcPositionName
 FROM tbPosition ps, tbPeoples pl
 WHERE ps.idPosition=pl.idPosition
```

Теперь триггер `INSTEAD OF` для обновления данных будет выглядеть, как показано в листинге 3.8.

Листинг 3.8. Обновление связанного представления с помощью триггера

```
CREATE TRIGGER u_People ON dbo.People
INSTEAD OF UPDATE
AS
BEGIN
 UPDATE tbPosition
 SET vcPositionName=i.vcPositionName
 FROM tbPosition pn, inserted i
 WHERE i.idPosition=pn.idPosition

 UPDATE tbPeoples
 SET vcFamil=i.vcFamil
 FROM tbPeoples pl, inserted i
 WHERE i.idPeoples=pl.idPeoples
END
```

Прежде чем обновлять данные, я рекомендую удалить все триггеры, которые мы создавали в этом разделе, иначе могут возникнуть проблемы, ведь некоторые из этих триггеров запрещают обновление. Итак, попробуйте сделать клерка генеральным директором:

```
UPDATE People
SET vcFamil='ИВАНУШКИН',
   vcPositionName='Генеральный директор'
WHERE idPeoples=40
  AND idPosition=13
```

Такое обновление не является идеальным, ведь обновляя название должности одного работника, мы изменяем название этой должности для всех работников. Справочники нужно редактировать очень аккуратно.

Пользователь, который запрашивает триггер, должен также иметь права на выполнение всех операторов в таблице. Следовательно, ему необходимо иметь права на обновление таблицы работников и таблицы должностей.

3.4.7. Дополнительные сведения о триггерах

Вы можете использовать триггеры для обеспечения комплексной целостности ссылок с помощью:

- выполнения действий или каскадного обновления или удаления. Целостность ссылок может отличаться при использовании ограничений `FOREIGN KEY` и `REFERENCE` в операторе `CREATE TABLE`. Но триггер выгоден для гарантирования необходимых действий, когда должны быть произведены каскадные удаления или обновления, потому что триггеры мощнее. Если ограничение существует для таблицы с триггером, оно проверяется до выполнения триггера. Если ограничение нарушено, то триггер не работает. Если ограничение не сработает, то с помощью триггера можно реализовать более сложные проверки, которые уж точно будут гарантировать, что не нарушится целостность, и пользователь внесет только те данные, которые разрешены;
- вы должны учитывать, что в таблицу может вставляться сразу несколько строк;
- ограничения, правила и значения по умолчанию могут генерировать только стандартные системные ошибки. Если вам нужны собственные сообщения, вы должны использовать триггеры.

При разработке триггеров вы должны учитывать, что таблицы могут иметь несколько триггеров для любого действия. Каждый триггер может быть

объявлен для нескольких или одного действия. В следующем примере обрабатывается два события INSERT и UPDATE:

```
CREATE TRIGGER iu_tbPeoples ON dbo.tbPeoples
FOR INSERT, UPDATE
AS
 Действие
```

Если на одно действие назначено несколько триггеров, во избежание конфликта имен можно к имени добавить слово, которое будет описывать выполняемые действия или назначение.

Владелец таблицы может указывать первый и последний триггеры. Когда несколько триггеров помещены на таблицу, владелец может использовать процедуру sp_settriggerorder (о хранимых системных таблицах мы будем говорить в следующей главе) для указания первого выполняемого триггера и последнего. На порядок остальных триггеров повлиять невозможно.

Владельцы таблицы не могут создавать триггеры на представления и временные таблицы. Однако триггеры могут ссылаться на представления и временные таблицы.

Триггеры не должны возвращать результирующих наборов, хотя не запрещается что-то выводить на печать с помощью оператора PRINT, но вы должны отдавать себе отчет, что пользователь увидит это только при откате транзакции. Таким образом, можно сообщить только об ошибке, но не об удачном выполнении, хотя, в большинстве случаев, этого достаточно.

Теперь поговорим о производительности триггеров. Они выполняются довольно быстро, потому что:

- расположены на сервере и не требуют для своего выполнения сетевых обращений, если только в самом коде триггера нет обращений по сети;
- таблицы inserted и deleted расположены в кэше, поэтому обращение к ним происходит достаточно быстро, если только они не содержат слишком большого количества строк, а само обращение не содержит сложных связей с другими таблицами.

Используйте триггеры только там, где это необходимо. Старайтесь возложить основные операции по обеспечению целостности на ограничения. Если нельзя найти другого выхода, то для повышения производительности сервера делайте объявления операторов триггеров простыми, насколько это возможно. Так как триггер является частью транзакции, блокировки сохраняются, пока транзакция не завершится, поэтому здесь скорость обработки очень важна.

3.4.8. Практика использования триггеров

Рассмотрим несколько полезных примеров.

Очень часто в базах данных необходимо сохранять историю. Для хранения изменений многие выбирают отдельную таблицу. Зачем? Основная таблица будет содержать только последние данные, иметь минимальный размер, и запросы к ней, за счет этого, будут выполняться максимально быстро. История будет находиться в отдельной таблице, и ее можно даже хранить в отдельной файловой группе, что предоставит нам достаточно мощные возможности при резервном копировании данных.

Итак, давайте создадим триггер, который при изменении или удалении строк в таблице `tbPeoples` будет копировать их в таблицу истории `tbpeoplesHistory`. Если бы первичный ключ был в виде уникального идентификатора, то задача решалась бы следующим образом:

```
CREATE TRIGGER ud_tbPeoples ON dbo. tbPeoples
FOR UPDATE, DELETE
AS
 INSERT INTO tbPeoplesHistory
  SELECT newid(), del.*
  FROM Deleted del
```

Таблица `tbPeoplesHistory` в точности повторяет таблицу `tbPeoples`, но у нее свой первичный ключ, т. е. к структуре `tbPeoples` мы добавили в начало одно поле. Зачем, когда можно было бы использовать поле первичного ключа основной таблицы? Дело в том, что этот ключ лучше всего сохранить нетронутым, чтобы в любой момент можно было восстановить связь записи из таблицы истории с другими таблицами базы данных.

В данном примере содержимое таблицы `deleted` копируется в таблице `tbPeoplesHistory`. Запрос упрощается тем, что первичный ключ можно сгенерировать с помощью функции `newid()`.

Но в нашей задаче первичный ключ является автоматически увеличиваемым, и его нельзя генерировать. Придется перечислять все поля:

```
CREATE TRIGGER ud_tbPeoplesHistory ON dbo.tbPeoples
FOR UPDATE, DELETE
AS
 INSERT INTO tbPeoplesHistory
   (idPeoples, vcFamil, vcName, vcSurname,
        idPosition, dDateBirthDay)
 SELECT del.*
 FROM Deleted del
```

Теперь посмотрим, как запретить удаление более чем одной строки:

```
CREATE TRIGGER d_tbPeoples ON dbo.tbPeoples
FOR DELETE
AS
 IF (SELECT count(*)
   FROM deleted)>1
 BEGIN
  PRINT 'Нельзя удалять более одной строки'
  ROLLBACK TRANSACTION
 END
```

Любой триггер может содержать операторы `UPDATE`, `INSERT` или `DELETE`, которые воздействуют на другие таблицы, как это происходило в примере создания истории изменений. При включенном вложении триггер, который изменяет таблицу, может активировать (за счет выполнения операции изменения другой таблицы, на которую есть свой триггер) другой триггер, который, в свою очередь, может активировать третий и т. д.

Вложение триггеров при инсталляции включено, но вы можете отключить их и снова включить с помощью системной процедуры `sp_configure`. Следующий пример отключает вложенные триггеры:

```
sp_configure 'nested triggers', 0
```

Триггеры могут иметь до 32 уровней вложенности. Если выполнение какого-нибудь триггера зациклится, то будет превышен этот предел. В таком случае триггер прерывается, и транзакция откатывается.

Вложенный триггер не будет вызываться дважды в одной транзакции, триггер не вызывает сам себя в ответ на второе обновление к одной и той же таблице. Например, если триггер изменяет таблицу, которая по цепочке изменяет оригинальную таблицу, триггер не вызывается снова.

Если на одном из уровней возникнет ошибка, то все изменения данных откатятся. Все вложенные триггеры воспринимаются как одна транзакция, а значит, никакие изменения во время выполнения `ROLLBACK TRANSACTION` не сохранятся.

Вложенные триггеры сложны в разработке, потому что к ним требуется комплексный и хорошо спланированный подход. Каскадное изменение может затронуть данные, на которые вы не хотели воздействовать. Именно поэтому иногда вложенные триггеры проще отключить.

Любой триггер может воздействовать на другие таблицы или ту же самую. Если включена опция рекурсивного вызова, то триггер, который изменяет

данные в таблице, может активировать себя снова. По умолчанию эта опция отключена, когда база данных создается. Вы можете включить эту опцию с помощью оператора `ALTER DATABASE`. Пример включения рекурсивных триггеров:

```
ALTER DATABASE FlenovSQLBook SET RECURSIVE_TRIGGERS ON
```

Если опция вложения отключена, то и рекурсия тоже отключена, и это необходимо всегда помнить.

С помощью триггера можно создать и рекурсивное удаление. В следующем примере мы создаем триггер, в котором при удалении записи о работнике удаляются и его номера телефонов, потому что номера без работника в таблице просто не нужны:

```
CREATE TRIGGER d_tbPeoples ON dbo.tbPeoples
FOR DELETE
AS
DELETE pn
FROM tbPhoneNumbers pn, inserted i
WHERE pn.idPeoples=i.idPeoples
```

С помощью триггеров можно выполнять определенные вычисления. Допустим, что в таблице `tbPeoples` должно быть поле, в котором сохраняется количество номеров телефонов. Конечно же, это денормализация данных, ведь количество можно всегда подсчитать, но не забывайте, что это пример. Для поддержки поля можно создать следующие триггеры:

- при добавлении записи в таблицу телефонов увеличиваем значение поля в таблице работников;
- при удалении номера телефона уменьшаем значение поля.

Попробуйте реализовать это самостоятельно, чтобы закрепить знания и потренироваться в работе с SQL-запросами.

Для определения таблиц с триггером выполните процедуру `sp_depends`. Например, выполните следующую команду, чтобы увидеть все зависимости для таблицы `tbPeoples`:

```
EXEC sp_depends 'tbPeoples'
```

Для определения, какие триггеры существуют на определенную таблицу и на какие действия, выполните процедуру `sp_helptrigger`. Следующий пример отображает все триггеры, которые принадлежат представлению `People` (если нужно просмотреть триггеры таблицы, то укажите ее имя):

```
EXEC sp_helptrigger People
```

Для просмотра кода существующего триггера используйте процедуру `sp_helptext`. Например, следующая команда позволяет увидеть текст триггера `u_People`, который мы создавали для объекта просмотра:

```
EXEC sp_helptext u_People
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter3\trigger.sql на компакт-диске, прилагаемом к книге.

3.5. SQL Server Agent

В состав сервера MS SQL Server входит служба SQL Server Agent, которая состоит из сообщений, операторов и заданий. Наибольший интерес программистов и администраторов вызывают задания, и этой теме мы уделим достаточно много внимания.

Работа администратора очень часто связана с выполнением однообразных задач, что превращает рабочий день в серые будни. Для меня это самое сложное, поэтому многократно выполняемые задачи я стремлюсь автоматизировать. У MS SQL Server есть очень мощное средство автоматизации — задания (job). Задания — это набор определенных действий (например, SQL-запросов), которые могут выполняться сервером автоматически в определенное время с помощью планировщика (Schedule) или запускаться администратором вручную.

Ярким примером задач администратора, которые могут вызывать скуку, является обслуживание баз данных, о чем мы будем довольно много говорить в *гл. 4*. Например, можно запрограммировать сервер так, чтобы он каждый день в конце рабочего дня создавал резервную копию базы данных.

Задания состоят из шагов, которые последовательно выполняются сервером MS SQL. Выполнение каждого последующего шага может зависеть от результата предыдущего. Таким образом, можно строить определенную логику задач.

Вы должны учитывать, что задания выполняются службой SQL Server Agent, которая входит в поставку MS SQL Server. Поэтому убедитесь, что эта служба работает, иначе задания не смогут выполняться по расписанию.

Помимо этого, если служба SQL Server Agent обращается к удаленным серверам по сети, то она должна работать под реальной учетной записью, а не под системной. Чтобы выбрать имя пользователя, обладающего правами, с которыми работает служба, запустите оснастку **Services** (Службы), выбрав

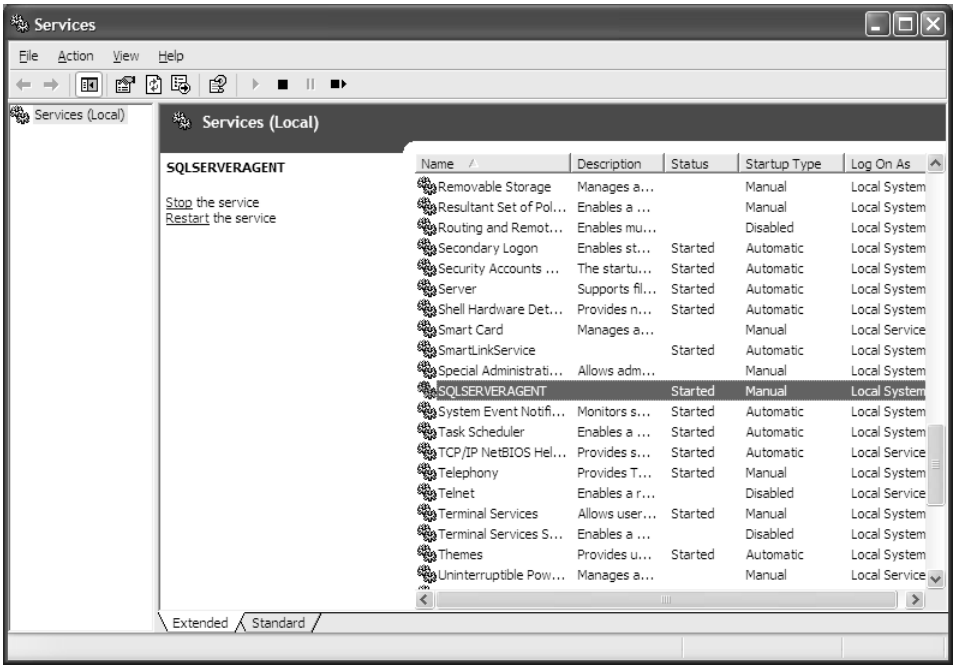


Рис. 3.1. Оснастка **Services**

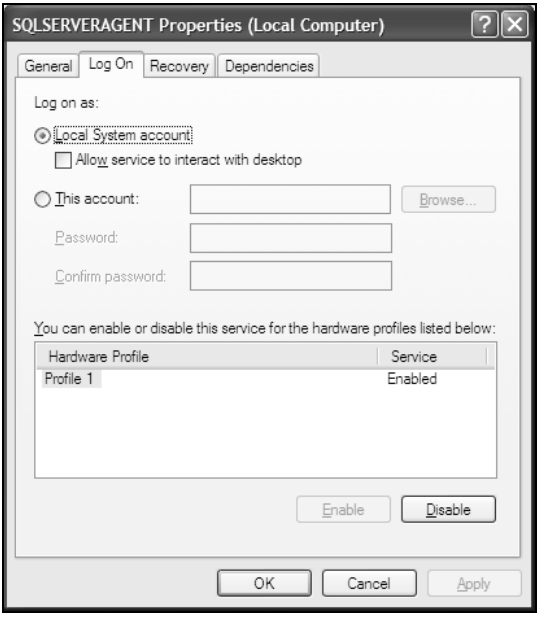


Рис. 3.2. Настройка службы SQL Service Agent

меню **Start | Control Panel | Administrative Tools | Services** (Пуск | Панель управления | Администрирование | Службы). Перед вами откроется окно, показанное на рис. 3.1. Найдите строку с именем службы SQLSERVERAGENT и дважды щелкните по ней. Перейдите на вкладку **Log On** (Вход в систему) и укажите реальную учетную запись пользователя, который существует в системе и обладает правами на необходимые ресурсы вашего компьютера и удаленного сервера, к которому будет происходить подключение по сети (рис. 3.2). Если служба SQL Server Agent будет работать с правами системной учетной записи, то у нее не хватит прав на подключение к удаленной системе, потому что системная учетная запись не содержит имени пользователя и пароля, необходимых для аутентификации.

3.5.1. Добавление задания

Начнем с добавления записей. Для этого используется хранимая процедура `sp_add_job`, которая выглядит следующим образом:

```
sp_add_job [ @job_name = ] 'job_name'
    [ , [ @enabled = ] enabled ]
    [ , [ @description = ] 'description' ]
    [ , [ @start_step_id = ] step_id ]
    [ , [ @category_name = ] 'category' ]
    [ , [ @category_id = ] category_id ]
    [ , [ @owner_login_name = ] 'login' ]
    [ , [ @notify_level_eventlog = ] eventlog_level ]
    [ , [ @notify_level_email = ] email_level ]
    [ , [ @notify_level_netsend = ] netsend_level ]
    [ , [ @notify_level_page = ] page_level ]
    [ , [ @notify_email_operator_name = ] 'email_name' ]
    [ , [ @notify_netsend_operator_name = ] 'netsend_name' ]
    [ , [ @notify_page_operator_name = ] 'page_name' ]
    [ , [ @delete_level = ] delete_level ]
    [ , [ @job_id = ] job_id OUTPUT ]
```

Рассмотрим параметры, которые передаются данной процедуре:

- `@job_name` — имя задания, которое должно быть уникальным и не может содержать символ процента;
- `@enabled` — индикатор активности задания. Если параметр равен 1, то задание активно и может выполняться из планировщика заданий. Если 0, то запустить задание можно только вручную;
- `@description` — текстовое описание задания;

- `@start_step_id` — идентификатор задания, которое должно выполняться первым, по умолчанию используется число 1;
- `@category_name` — имя категории;
- `@category_id` — идентификатор категории. Этот параметр может использоваться в определенных языках программирования;
- `@owner_login_name` — имя пользователя, который будет являться владельцем задания. Если ничего не указано, то владельцем будет текущий пользователь;
- `@notify_level_eventlog` — какие сообщения должны сохраняться в журнале событий Windows. Здесь можно указать одно из следующих значений:
  - 0 — никакие сообщения;
  - 1 — сообщения об удачном завершении;
  - 2 — сообщения об ошибках (значение по умолчанию);
  - 3 — все сообщения;
- `@notify_level_email` — какие сообщения должны направляться на электронный адрес оператора. Возможные значения этого параметра идентичны значениям параметра `@notify_level_eventlog`;
- `@notify_level_netsend` — какие сообщения должны направляться оператору по сети с помощью команды `NET SEND`, которая используется в ОС семейства NT (Windows NT/2000/XP/2003). Возможные значения этого параметра идентичны значениям параметра `@notify_level_eventlog`;
- `@notify_level_page` — какие сообщения должны направляться оператору на пейджер. Возможные значения этого параметра идентичны значениям параметра `@notify_level_eventlog`;
- `@notify_email_operator_name` — имя оператора, которому будут направляться сообщения электронной почты;
- `@notify_netsend_operator_name` — имя оператора, которому будут направляться сетевые сообщения;
- `@notify_page_operator_name` — имя оператора, которому будут направляться сообщения на пейджер;
- `@delete_level` — параметр определяет, когда нужно удалять задание. Возможные значения этого параметра идентичны значениям параметра `@notify_level_eventlog`. Если вы хотите, чтобы задание не удалялось, укажите 0. Если задание нужно удалить после удачного завершения, укажите 1. Если указать число 3, то задание будет выполняться только один раз, вне зависимости от результата;

- `@job_id` — этот параметр должен содержать переменную, в которую будет записан идентификатор, назначенный заданию.

Прежде чем использовать процедуру, необходимо отметить, что она принадлежит базе данных msdb, поэтому необходимо подключиться именно к этой базе.

Есть еще одно ограничение — вы должны указывать имена только реально существующих в базе данных операторов. Рассмотрим пример без указания операторов:

```
USE msdb
EXEC sp_add_job @job_name = 'Тестовое задание',
   @enabled = 1,
   @description = 'Это тестовое задание',
   @notify_level_eventlog = 3,
   @notify_level_email = 2,
   @notify_level_netsend = 1
```

3.5.2. Управление операторами

Оператор — это человек, который должен получать сообщения о ходе выполнения задания. Для создания оператора используется процедура `sp_add_operator`, которая выглядит следующим образом:

```
sp_add_operator [ @name = ] 'name'
    [ , [ @enabled = ] enabled ]
    [ , [ @email_address = ] 'email_address' ]
    [ , [ @pager_address = ] 'pager_address' ]
    [ , [ @weekday_pager_start_time = ] weekday_pager_start_time ]
    [ , [ @weekday_pager_end_time = ] weekday_pager_end_time ]
    [ , [ @saturday_pager_start_time = ] saturday_pager_start_time ]
    [ , [ @saturday_pager_end_time = ] saturday_pager_end_time ]
    [ , [ @sunday_pager_start_time = ] sunday_pager_start_time ]
    [ , [ @sunday_pager_end_time = ] sunday_pager_end_time ]
    [ , [ @pager_days = ] pager_days ]
    [ , [ @netsend_address = ] 'netsend_address' ]
    [ , [ @category_name = ] 'category' ]
```

Рассмотрим параметры этой процедуры:

- `@name` — имя оператора;
- `@enabled` — если параметр равен 1, то оператор активен, а если указать 0, то оператор недоступен и не должен получать сообщения;

- `@email_address` — электронный адрес (e-mail) оператора. Если значение этого параметра является физическим адресом SMTP:flenov@mail.ru, а не псевдонимом flenov, то он должен быть заключен в квадратные скобки [SMTP:flenov@mail.ru]. Дело в том, что по умолчанию MS SQL Server использует почтовую службу Exchange Service, у которого своя система именования ящиков;
- `@pager_address` — адрес интернет-пейджера;
- `@weekday_pager_start_time` — время, после которого можно отправлять сообщения на пейджер в рабочий день. Время указывается в формате ЧЧММСС, например, число 100100 соответствует 10 часам, 01 минуте и 00 секундам;
- `@weekday_pager_end_time` — время, до которого можно отправлять сообщения на пейджер в рабочий день. Время указывается в формате ЧЧММСС, например, число 100100 соответствует 10 часам, 01 минуте и 00 секундам;
- `@saturday_pager_start_time` и `@sunday_pager_start_time` — время, после которого можно отправлять сообщения на пейджер в субботу и в воскресенье соответственно. Время указывается в формате ЧЧММСС, например, число 100100 соответствует 10 часам, 01 минуте и 00 секундам;
- `@saturday_pager_end_time` и `@sunday_pager_end_time` — время, до которого можно отправлять сообщения на пейджер в субботу и в воскресенье соответственно. Время указывается в формате ЧЧММСС, например, число 100100 соответствует 10 часам, 01 минуте и 00 секундам;
- `@pager_days` — этот параметр определяет дни, в которые оператор доступен для получения сообщений на пейджер. Здесь можно указать одно из следующих значений или сумму из нескольких чисел:
  - 0 — никогда (это значение по умолчанию);
  - 1 — по воскресеньям;
  - 2 — по понедельникам;
  - 4 — по вторникам;
  - 8 — по средам;
  - 16 — по четвергам;
  - 32 — по пятницам;
  - 64 — по субботам.

  Как указать, что пользователь доступен с понедельника по пятницу? Для этого складываем соответствующие числа 2+4+8+16+32. В результате мы

получим 62, и именно это значение необходимо указать в параметре `@pager_days`;

- `@netsend_address` — сетевой адрес оператора (имя компьютера), на который нужно отправлять сообщения;
- `@category_name` — имя категории сообщений.

Посмотрим, как можно создать оператора, который будет получать сообщения на электронный адрес:

```
use msdb
exec sp_add_operator @name = 'Андрей',
   @enabled = 1,
   @email_address ='[SMTP:flenov@mail.ru]'
```

Следующий пример создает оператора, который может получать сообщения по электронной почте и по сети:

```
use msdb
exec sp_add_operator @name = 'Михаил',
   @enabled = 1,
   @email_address ='[SMTP:admin@mail.ru]',
   @netsend_address ='admincomp'
```

Теперь посмотрим, как можно создать задание, в котором сообщения о статусе выполнения задания передаются операторам:

```
USE msdb
EXEC sp_add_job @job_name = 'Тестовое задание 2',
   @enabled = 1,
   @description = 'Это тестовое задание с указанием оператора',
   @notify_level_eventlog = 3,
   @notify_level_email = 2,
   @notify_level_netsend = 1,
   @notify_email_operator_name = 'Андрей',
   @notify_netsend_operator_name = 'Михаил'
```

Если хотя бы один из операторов, указанных в примере, не будет существовать в базе данных, выполнение процедуры завершится неудачей.

Изменение оператора

Для изменения параметров оператора используется процедура `sp_update_operator`, которая выглядит так:

```
sp_update_operator [@name =] 'name'
    [, [@new_name =] 'new_name']
```

```
    [, [@enabled =] enabled]
    [, [@email_address =] 'email_address']
    [, [@pager_address =] 'pager_number']
    [, [@weekday_pager_start_time =] weekday_pager_start_time]
    [, [@weekday_pager_end_time =] weekday_pager_end_time]
    [, [@saturday_pager_start_time =] saturday_pager_start_time]
    [, [@saturday_pager_end_time =] saturday_pager_end_time]
    [, [@sunday_pager_start_time =] sunday_pager_start_time]
    [, [@sunday_pager_end_time =] sunday_pager_end_time]
    [, [@pager_days =] pager_days]
    [, [@netsend_address =] 'netsend_address']
    [, [@category_name =] 'category']
```

Параметры процедуры изменения оператора такие же, как и при создании, поэтому не будем тратить время на обсуждение оператора, а лучше рассмотрим его работу на практике. Следующий пример изменяет электронный и сетевой адреса:

```
exec sp_update_operator @name = 'Михаил',
   @email_address ='[SMTP:mikhail@mail.ru]',
   @netsend_address ='notebook'
```

Указание имени оператора является обязательным, потому что процедура должна знать, какого именно оператора нужно обновлять.

Следующий пример делает оператора неактивным, после чего он не будет получать информационные сообщения:

```
exec sp_update_operator @name = 'Михаил',
   @enabled=0
```

Изменяется только параметр `@enabled`, а все остальные не изменяются и сохраняют свои значения.

Информация об операторе

Чтобы убедиться в том, что изменения прошли успешно, можно воспользоваться процедурой `sp_help_operator`, которая выводит информацию об операторе. В качестве параметра `@operator_name` нужно передать имя интересующего вас оператора, например, так:

```
exec sp_help_operator
  @operator_name = 'Михаил'
```

Давайте снова сделаем Михаила активным, чтобы он мог получать информационные сообщения:

```
exec sp_update_operator
   @name = 'Михаил',
   @enabled=1
```

Удаление оператора

Для удаления оператора используется процедура `sp_delete_operator`, которая выглядит следующим образом:

```
sp_delete_operator [ @name = ] 'name'
    [ , [ @reassign_to_operator = ] 'reassign_operator' ]
```

Здесь всего два параметра:

- `@name` — имя удаляемого оператора;
- `@reassign_to_operator` — необязательный параметр, где можно указать оператора, которому должны быть назначены события, отслеживаемые удаляемым оператором.

Следующий пример удаляет оператора Михаил, а все события, которые он отслеживал, будет теперь отслеживать Андрей:

```
EXEC sp_delete_operator
    @name = 'Михаил',
    @reassign_to_operator = 'Андрей'
```

3.5.3. Добавление шага

Мы научились создавать и удалять задания, а также добавлять операторов, но все это пока лишено смысла, ведь создаваемое задание еще ничего не умеет делать. Теперь необходимо научиться создавать шаги задания. Для этого используется процедура `sp_add_jobstep`. В общем виде процедура выглядит следующим образом:

```
sp_add_jobstep [ @job_id = ] job_id | [ @job_name = ] 'job_name'
    [ , [ @step_id = ] step_id ]
    { , [ @step_name = ] 'step_name' }
    [ , [ @subsystem = ] 'subsystem' ]
    [ , [ @command = ] 'command' ]
    [ , [ @additional_parameters = ] 'parameters' ]
    [ , [ @cmdexec_success_code = ] code ]
    [ , [ @on_success_action = ] success_action ]
```

```
[ , [ @on_success_step_id = ] success_step_id ]
[ , [ @on_fail_action = ] fail_action ]
[ , [ @on_fail_step_id = ] fail_step_id ]
[ , [ @server = ] 'server' ]
[ , [ @database_name = ] 'database' ]
[ , [ @database_user_name = ] 'user' ]
[ , [ @retry_attempts = ] retry_attempts ]
[ , [ @retry_interval = ] retry_interval ]
[ , [ @os_run_priority = ] run_priority ]
[ , [ @output_file_name = ] 'file_name' ]
[ , [ @flags = ] flags ]
```

В первой строчке указано, что процедуре необходимо передать идентификатор или имя задания, куда нужно добавить новый шаг. Рассмотрим параметры более подробно:

- `@job_id` — идентификатор задания, куда нужно добавить шаг;
- `@job_name` — имя задания, куда нужно добавить шаг;
- `@step_id` — автоматически увеличиваемое число, начиная с единицы. Если параметр не указан, то значение будет установлено автоматически (следующее за максимально существующим номером шага), т. е. шаг будет добавлен в конец цепочки шагов. Если у вас уже есть шаги с номерами 1, 2 и 3, и вы указали в параметре `@step_id` число 3, то новое значение будет вставлено на третью позицию, а шаг, который был на этом месте ранее, сместится на четвертую. При этом идентификаторы шагов будут перенумерованы автоматически;
- `@step_name` — имя шага;
- `@subsystem` — система команд, которые будут выполняться сервером на данном шаге. В этом параметре можно указать одно из следующих значений:
  - `ACTIVESCRIPTING` — выполняться должен активный сценарий;
  - `CMDEXEC` — на данном шаге будет выполняться команда ОС или внешняя программа;
  - `DISTRIBUTION` — выполнение дистрибутора репликации;
  - `SNAPSHOT` — выполнение агента репликации снятия снимка;
  - `LOGREADER` — выполнение агента чтения журнала;
  - `MERGE` — выполнение агента репликации слиянием;
  - `TSQL` — шаг будет выполнять команду Transact-SQL (это значение по умолчанию);

- `@command` — непосредственно команда, которая должна выполняться. Параметр `@command` должен содержать корректные команды, в соответствии с указанным типом в параметре `@subsystem`;
- `@additional_parameters` — дополнительные параметры команды. На данный момент этот параметр зарезервирован;
- `@cmdexec_success_code` — если выполняется команда ОС, то через этот параметр можно получить результат выполнения;
- `@on_success_action` — этот параметр определяет, что делать при удачном завершении выполнения шага. Здесь можно указать одно из следующих значений:
  - 1 — выход с сообщением об удачном завершении. Если заданию установлен оператор, отслеживающий удачное завершение задания, то он получит соответствующее уведомление;
  - 2 — выход с ошибкой. Иногда удачное выполнение команды является отрицательным результатом. Например, запрос может проверять наличие несвязанных строк. Если строки найдены, то, по идее, шаг должен завершиться успешно, но несвязанные строки — это нарушение целостности. Поэтому можно указать, что при удачном выполнении сценария задание должно завершиться с ошибкой и проинформировать оператора, если он установлен;
  - 3 — перейти к выполнению следующего оператора;
  - 4 — перейти к выполнению шага, указанного в параметре `@on_success_step_id`;
- `@on_success_step_id` — шаг, на который нужно перейти, если в параметре `@on_success_action` указано число 4;
- `@on_fail_action` — действие, которое необходимо выполнить при неудачном завершении шага. В этом параметре можно указывать те же значения, что и у параметра `@on_success_action`;
- `@on_fail_step_id` — шаг, на который нужно перейти, если в параметре `@on_fail_action` указано число 4;
- `@server` — зарезервировано;
- `@database_name` — если на данном шаге выполняются операторы Transact-SQL, то необходимо в данном параметре указать, какую базу данных нужно использовать. Если ничего не указано, то будет использоваться база данных master;
- `@database_user_name` — имя пользователя, с правами которого будет выполняться оператор Transact-SQL в базе данных. Необходимо убедиться,

что прав указанного пользователя достаточно для выполнения оператора. Если параметр не задан, то операторы Transact-SQL будут выполняться с правами владельца задания;

- `@retry_attempts` — количество попыток выполнить данный шаг в случае ошибки. По умолчанию сервер не будет пытаться повторять выполнение указанного шага;
- `@retry_interval` — промежуток времени в минутах между попытками повторить выполнение шага в случае ошибки;
- `@os_run_priority` — зарезервировано;
- `@output_file_name` — имя файла, в который будет сохранен результат выполнения команды. Файл может использоваться при выполнении Transact-SQL или системных команд `CMDEXEC`;
- `@flags` — дополнительные флаги. В этом параметре может быть одно из следующих значений:
  - 0 — нет никаких дополнительных опций (это значение по умолчанию);
  - 2 — результат выполнения должен добавляться в выходной файл, указанный в параметре `@output_file_name`;
  - 4 — перезаписать выходной файл. Существующее содержимое будет уничтожено.

Давайте добавим в задание с именем `Тестовое задание 2` два шага. На первом шаге будет удаляться таблица `tbAndrey`, а на втором эта же таблица будет создаваться с помощью оператора `SELECT INTO`. Создание первого шага может выглядеть примерно так:

```
EXEC sp_add_jobstep
   @job_name = 'Тестовое задание 2',
   @step_name = 'Удаляем таблицу',
   @subsystem = 'TSQL',
   @command = 'DROP TABLE tbAndrey',
   @database_name = 'FlenovSQLBook',
   @on_success_action = 3,
   @on_fail_action = 3
```

Данный шаг будет выполнять команду Transact-SQL, а значит, в параметре `@subsystem` указываем значение `TSQL`. В параметре `@command` указываем непосредственно SQL-команду. Так как по умолчанию запрос будет выполняться в базе данных master, то в параметре `@database_name` явно указываем свою базу.

Основная цель задания — создать таблицу `tbAndrey` и заполнить значениями, но для этого сначала старую таблицу нужно удалить. А если старой таблицы

нет (ее кто-то удалил или вообще ее не было)? В этом случае все равно задание должно продолжать выполнение, поэтому в параметрах `@on_success_action` и `@on_fail_action` указываем значение 3, т. е. переход на следующий шаг.

Теперь напишем второй шаг, на котором будет производиться создание таблицы:

```
EXEC sp_add_jobstep
   @job_name = 'Тестовое задание 2',
   @step_name = 'Выбираем Андреев',
   @subsystem = 'TSQL',
   @command = 'SELECT * INTO tbAndrey FROM tbPeoples
       WHERE vcName='Андрей',
   @database_name = 'FlenovSQLBook',
   @on_success_action = 1,
   @on_fail_action = 2
```

В данном случае переход на следующий шаг не ожидается, поэтому завершаем задание с соответствующим кодом возврата.

Когда вы пишете сценарий для задания, вы можете использовать некоторые вспомогательные конструкции, которые во время выполнения будут заменяться на определенные параметры. Рассмотрим возможные конструкции, и на что они заменяются:

- `[A-DBN]` — заменяется именем базы данных;
- `[A-SVR]` — заменяется именем сервера;
- `[A-ERR]` — номер ошибки;
- `[A-SEV]` — критичность ошибки;
- `[A-MSG]` — заменяется текстовым сообщением;
- `[DATE]` — текущая дата;
- `[JOBID]` — идентификатор задания;
- `[MACH]` — имя компьютера;
- `[MSSA]` — имя службы SQLServerAgent;
- `[SQLDIR]` — каталог, в котором установлен SQL-сервер;
- `[STEPCT]` — сколько раз было запущено задание (количество попыток);
- `[STEPID]` — идентификатор шага;
- `[TIME]` — текущее время в формате HHMMSS;
- `[STRTTM]` — время в формате HHMMSS, когда задание было запущено;
- `[STRTDT]` — дата в формате YYYYMMDD, когда задание было запущено.

Будьте внимательны, все конструкции должны заключаться в квадратные скобки, и все они чувствительны к регистру.

Давайте посмотрим, как использовать эти конструкции, а заодно увидим, как можно вставлять новые шаги. Следующий пример не добавляет новый шаг, а вставляет его на первую позицию (параметр `@step_id` равен 1):

```
EXEC sp_add_jobstep
   @job_name = 'Тестовое задание 2',
   @step_id=1,
   @step_name = 'Вставляем строку',
   @subsystem = 'TSQL',
   @command = 'INSERT INTO tbPeoples (vcName, vcSurname)
      VALUES(''[MACH]'', ''[STEPID]'')',
   @database_name = 'FlenovSQLBook',
   @on_success_action = 3,
   @on_fail_action = 3
```

В данном примере в качестве команды в таблицу `tbPeoples` вставляется строка, в которой имени назначается имя компьютера (конструкция `[MACH]`), а в качестве фамилии указывается текущий номер шага (конструкция `[STEPID]`).

Обратите внимание, что в параметре `@command` в конструкции `VALUES` значения, вставляемые в таблицу, должны заключаться в одинарные кавычки, а мы указали по две одинарных с каждой стороны. Почему? Дело в том, что вся команда `INSERT` должна быть в одинарных кавычках:

```
@command='КОМАНДА'
```

Если внутри команды используется одинарная кавычка, то сервер преждевременно воспримет ее как конец команды. Например:

```
@command = 'INSERT INTO tbPeoples (vcName, vcSurname)
   VALUES('[MACH]', '[STEPID]')'
```

В данном случае сервер поместит в параметр `@command` только строку:

```
@command = 'INSERT INTO tbPeoples (vcName, vcSurname)
   VALUES('
```

Именно этот текст находится между первыми двумя кавычками. Чтобы этого не произошло, внутри команды нужно продублировать все одинарные кавычки.

Автоматически заменяемые во время выполнения конструкции действительно могут упростить разработку заданий и иногда оказываются незаменимыми, особенно даты и время выполнения задания.

Давайте создадим еще один шаг, на котором будет выполняться системная команда, и вставим его под номером 2:

```
EXEC sp_add_jobstep
   @job_name = 'Тестовое задание 2',
   @step_id=2,
   @step_name = 'Системная команда',
   @subsystem = 'CMDEXEC',
   @command = 'del c:\text.txt',
   @on_success_action = 3,
   @on_fail_action = 3
```

Так как будет выполняться системная команда, параметр `@subsystem` устанавливаем в `CMDEXEC`. В параметре `@command` для примера я указал команду удаления файла text.txt из корня диска C:. Когда вы будете тестировать пример (о том, как это сделать, мы узнаем в *разд. 3.5.4*), не забудьте создать этот файл, чтобы убедиться в том, что файл после выполнения задания исчезает.

При выполнении системных команд вы должны учитывать следующее:

- команды выполняются в контексте и с правами MS SQL Server. Это значит, что на экране не будет отображаться результат, даже если вы запускаете программу, которая должна что-то отображать на экране или должна иметь визуальный интерфейс. Чтобы убедиться в этом, можно в качестве системной команды указать calc (калькулятор). После запуска задания вы ничего не увидите на экране;
- не запускайте в качестве команды программы с визуальным интерфейсом. Такие программы не видны на экране, поэтому их невозможно будет закрыть, а значит, задание просто зависнет и не сможет завершить работу;
- будьте аккуратны при задании команд, чтобы их выполнение было конечным, и не произошло зависание всего задания. Например, если задать команду `ping -t localhost`, то задание будет бесконечно "пинговать" локальный компьютер. Это приведет к зацикливанию шага, и дальнейшее выполнение задания станет невозможным.

3.5.4. Запуск задания

У нас уже есть задание и шаги, которые оно выполняет, но запускать задание еще рано. Необходимо сначала привязать его к серверу, иначе выполнение завершится ошибкой. Для привязки задания к серверу необходимо выполнить

процедуру sp_apply_job_to_targets, которая в общем виде выглядит следующим образом:

```
sp_apply_job_to_targets [ @job_id = ] job_id |
    [ @job_name = ] 'job_name'
    [ , [ @target_server_groups = ] 'target_server_groups' ]
    [ , [ @target_servers = ] 'target_servers' ]
    [ , [ @operation = ] 'operation' ]
```

- @job_id и @job_name — позволяют с помощью идентификатора или имени задания соответственно указать задание, которое мы хотим привязать к серверу;
- @target_server_groups — строка из имен групп серверов, разделенных запятыми, к которым нужно произвести привязку;
- @target_servers — строка из имен серверов, разделенных запятыми, к которым нужно произвести привязку;
- @operation — операция, которую необходимо выполнить. В данном параметре можно указать одно из следующих значений:
  - APPLY — необходимо привязать задание к указанным серверам;
  - REMOVE — необходимо удалить привязку задания от указанных серверов.

Рассмотрим пример связывания задания с сервером:

```
EXEC sp_apply_job_to_targets
  @job_name = 'Тестовое задание 2',
  @target_servers='NOTEBOOK',
  @operation=APPLY
```

В параметре @target_servers указано имя NOTEBOOK. Это имя моего компьютера и соответственно имя установленного экземпляра SQL Server по умолчанию, на котором я и пишу примеры для данной книги. Ваш экземпляр сервера, скорее всего, будет называться по-другому, поэтому укажите его имя.

Если вы используете именованный экземпляр MS SQL Server, то необходимо указать имя полностью. Например, ваш экземпляр может называться Econom и находиться на сервере BigServer. В этом случае в параметре @target_servers необходимо указать строку 'BigServer\Econom'.

В параметре @operation указано значение APPLY, т. е. данный сервер должен быть назначен заданию.

Вот теперь мы готовы запускать задание на выполнение. Существует две возможности сделать это: автоматически с помощью планировщика и вручную.

Для ручного запуска задания на выполнение используется хранимая процедура `sp_start_job`, которая в общем виде выглядит следующим образом:

```
sp_start_job [@job_name =] 'job_name' | [@job_id =] job_id
    [,[@error_flag =] error_flag]
    [,[@server_name =] 'server_name']
    [,[@step_name =] 'step_name']
    [,[@output_flag =] output_flag]
```

Давайте рассмотрим параметры этой команды:

- `@job_id` и `@job_name` — позволяют с помощью идентификатора или имени задания соответственно указать задание, которое мы хотим выполнить;
- `@error_flag` — зарезервировано;
- `@server_name` — имя сервера, на котором нужно запустить задание. Данное имя должно быть связано с заданием с помощью системной процедуры `sp_apply_job_to_targets`;
- `@step_name` — имя шага, с которого нужно начать выполнение. Если здесь ничего не указано, то задание будет запущено с самого первого шага. Иногда бывает необходимо запустить задание не с первого шага. Например, в *разд. 3.5.3* мы создали два шага удаления и создания таблицы. Если вы уверены, что создаваемой таблицы нет, то первый шаг можно пропустить;
- `@output_flag` — зарезервировано.

Минимальная команда запуска задания на выполнение будет выглядеть следующим образом:

```
EXEC sp_start_job @job_name = 'Тестовое задание 2'
```

Если в *разд. 3.5.3* вы создавали все шаги, которые мы рассматривали, то можете просмотреть таблицу `tbPeoples` и убедиться, что в нее вставлена новая строка, где поле имени равно имени компьютера, а фамилия равна номеру шага.

Если задание связано с несколькими серверами, то для запуска на конкретном сервере используем параметр `@server_name`. Следующий пример запускает задание на сервере `NOTEBOOK`, начиная со второго шага, имя которого `Удаляем таблицу`:

```
EXEC sp_start_job @job_name = 'Тестовое задание 2',
    @server_name = 'notebook',
    @step_name = 'Удаляем таблицу'
```

Теперь создадим файл text.txt на диске C: и выполним команду, начиная со второго шага, где запускается системная команда:

```
EXEC sp_start_job @job_name = 'Тестовое задание 2',
    @step_name = 'Системная команда'
```

Убедитесь, что созданный файл удален.

3.5.5. Информация о задании

Мы научились создавать задание, его шаги и запускать задание на выполнение. По логике вещей мы должны сейчас узнать, как все это редактировать и удалять, но эту тему мы ненадолго отложим, а пока обсудим, как можно получить информацию о задании. Это поможет нам впоследствии увидеть результат редактирования задания и его шагов.

Для просмотра информации о задании используется хранимая процедура `sp_help_job`, которая выглядит следующим образом:

```
sp_help_job [ [ @job_id = ] job_id ]
    [ , [ @job_name = ] 'job_name' ]
    [ , [ @job_aspect = ] 'job_aspect' ]
    [ , [ @job_type = ] 'job_type' ]
    [ , [ @owner_login_name = ] 'login_name' ]
    [ , [ @subsystem = ] 'subsystem' ]
    [ , [ @category_name = ] 'category' ]
    [ , [ @enabled = ] enabled ]
    [ , [ @execution_status = ] status ]
    [ , [ @date_comparator = ] 'date_comparison' ]
    [ , [ @date_created = ] date_created ]
    [ , [ @date_last_modified = ] date_modified ]
    [ , [ @description = ] 'description_pattern' ]
```

Рассмотрим параметры этой команды:

- `@job_id` и `@job_name` — позволяют с помощью идентификатора или имени задания соответственно указать задание, информация о котором нас интересует;
- `@job_aspect` — атрибуты задания, которые необходимо отобразить. В этом параметре можно указывать одно из следующих значений:
  - `ALL` — полную информацию (значение по умолчанию);
  - `JOB` — информацию о задании;
  - `STEPS` — информацию о шагах;

- `SCHEDULES` — информацию о плане выполнения;
- `TARGET` — информацию о серверах, к которым привязано задание;

❒ `@job_type` — тип задания, включаемый в отчет. В этом параметре можно указать одно из следующих значений:

- `LOCAL` — локальный, т. е. привязанный к текущему серверу;
- `MULTISERVER` — привязанный к нескольким серверам;

❒ `@owner_login_name` — имя пользователя — владельца задания;

❒ `@subsystem` — отобразить задания, которые выполняют указанные в этом параметре системы команд;

- `ACTIVESCRIPTING` — активный сценарий;
- `CMDEXEC` — команда ОС;
- `DISTRIBUTION` — дистрибутор репликации;
- `SNAPSHOT` — агент репликации снятия снимка;
- `LOGREADER` — агент чтения журнала;
- `MERGE` — агент репликации слиянием;
- `TSQL` — Transact-SQL;

❒ `@category_name` — имя категории;

❒ `@enabled` — если параметр равен 1, то информация должна отображаться для активных заданий, иначе для неактивных;

❒ `@execution_status` — отобразить задания с указанным статусом. В этом параметре можно указать одно из следующих значений:

- 0 — отобразить задания, которые находятся в состоянии ожидания или приостановленные;
- 1 — выполняющиеся задания;
- 2 — задания, ожидающие потока;
- 3 — попытка повторить шаг;
- 4 — задания, находящиеся в состоянии ожидания;
- 5 — приостановленные задания;
- 7 — задания, выполняющие завершающее действие;

❒ `@date_comparator` — оператор сравнения, который будет использоваться для сравнения дат из параметров `@date_created` и `@date_last_modified`. Здесь можно указывать знаки равенства, больше или меньше;

- `@date_created` — дата создания;
- `@date_last_modified` — дата последнего изменения;
- `@description` — описание.

Если выполнить процедуру без параметров, то она отобразит на экране все доступные на сервере задания:

```
EXEC sp_help_job
```

Следующая команда позволяет увидеть задания, которые выполняют запросы Transact-SQL:

```
EXEC sp_help_job @subsystem='TSQL'
```

Следующий пример демонстрирует, как можно просмотреть задания, созданные после 1 января 2005 года:

```
EXEC sp_help_job
    @date_comparator = '>',
    @date_created='01.01.2005'
```

В параметре `@date_created` указываем необходимую дату, а в параметре `@date_comparator` символ сравнения. Если изменить знак сравнения на знак равенства, то мы увидим задания, созданные именно 1 января 2005 года:

```
EXEC sp_help_job @date_comparator = '=',
    @date_created='01.01.2005'
```

Минимальная команда отображения информации о задании будет выглядеть следующим образом:

```
EXEC sp_help_job @job_name='Тестовое задание 2'
```

По умолчанию процедура возвращает подробную информацию о задании (рис. 3.3), которая состоит из четырех таблиц, содержащих следующую информацию: о задании, о шагах, о плане выполнения, о связанных с заданием серверах. Чтобы отобразить только одну из таблиц, используйте параметр `@job_aspect`. Например, следующая команда отображает только информацию о шагах:

```
EXEC sp_help_job @job_name='Тестовое задание 2',
    @job_aspect='STEPS'
```

В результате мы получим таблицу шагов, в которой столбцы идентичны именам параметров процедуры создания шагов `sp_add_jobstep`. Строки отсортированы по полю `step_id`, т. е. отображаются в том же порядке, в котором они должны выполняться сервером.

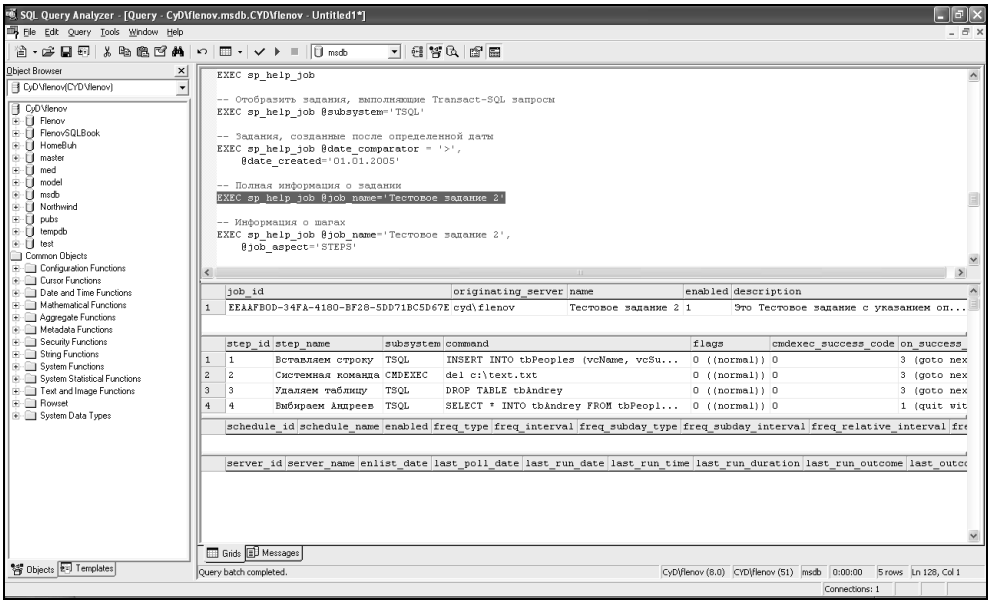


Рис. 3.3. Пример отображения подробной информации о задании

Для получения информации о шаге задания есть процедура `sp_help_jobstep`, которая выглядит следующим образом:

```
sp_help_jobstep [ @job_id = ] 'job_id' |
    [ @job_name = ] 'job_name'
    [ , [ @step_id = ] step_id ]
    [ , [ @step_name = ] 'step_name' ]
    [ , [ @suffix = ] suffix ]
```

Рассмотрим параметры этой процедуры:

- `@job_id` и `@job_name` — позволяют с помощью идентификатора или имени задания соответственно указать задание, информация о шагах которого нас интересует;
- `@step_id` — идентификатор шага, информация о котором нас интересует;
- `@step_name` — имя интересующего вас шага;
- `@suffix` — если параметр равен 1, то в поле `flag` будет добавлено описание.

Простейший способ узнать информацию о шагах задания — выполнить процедуру `sp_help_jobstep`, передав ей только имя интересующего вас задания:

```
USE msdb
EXEC sp_help_jobstep @job_name='Тестовое задание 2'
```

Результат эквивалентен выполнению процедуры `sp_help_job` с указанием этого же имени задания и параметра `@job_aspect` со значением `STEPS`:

```
EXEC sp_help_job
    @job_name='Тестовое задание 2',
    @job_aspect='STEPS'
```

Следующий пример отображает информацию о первом шаге задания:

```
EXEC sp_help_jobstep @job_name='Тестовое задание 2',
  @step_id='1'
```

3.5.6. Управление заданиями

Все в нашем мире изменяется, и только исторические ценности должны оставаться в своем первоначальном виде. Код SQL-запросов к историческим ценностям отнести трудно, поэтому нередко приходится вносить в него изменения в соответствии с изменяющимися условиями задачи, а как показывает практика, даже простая задача может измениться.

Для изменения задания используется процедура `sp_update_job`, которая выглядит следующим образом:

```
sp_update_job [@job_id =] job_id | [@job_name =] 'job_name'
    [, [@new_name =] 'new_name']
    [, [@enabled =] enabled]
    [, [@description =] 'description']
    [, [@start_step_id =] step_id]
    [, [@category_name =] 'category']
    [, [@owner_login_name =] 'login']
    [, [@notify_level_eventlog =] eventlog_level]
    [, [@notify_level_email =] email_level]
    [, [@notify_level_netsend =] netsend_level]
    [, [@notify_level_page =] page_level]
    [, [@notify_email_operator_name =] 'email_name']
    [, [@notify_netsend_operator_name =] 'netsend_operator']
    [, [@notify_page_operator_name =] 'page_operator']
    [, [@delete_level =] delete_level]
    [, [@automatic_post =] automatic_post]
```

Даже первый взгляд показывает, что процедура очень похожа на процедуру добавления задания `sp_add_job`. Параметры идентичны, поэтому не будем тратить время на их обсуждение, а лучше рассмотрим, как можно использовать процедуру `sp_update_job` на практике.

Давайте изменим у задания `Тестовое задание 2` описание и оператора, который будет получать сообщения по электронной почте или по сети:

```
USE msdb

EXEC sp_update_job
   @job_name = 'Тестовое задание 2',
   @description = 'Тестовое задание',
   @notify_email_operator_name = 'Михаил',
   @notify_netsend_operator_name = 'Михаил'
```

Для удаления задания используется процедура `sp_delete_job`, которой в качестве параметра нужно указать идентификатор или имя удаляемого задания. Например, следующая команда удалит созданное в *разд. 3.5.1* задание с именем `Тестовое задание`:

```
EXEC sp_delete_job @job_name = 'Тестовое задание'
```

3.5.7. Управление шагами

Для обновления шагов задания используется хранимая процедура `sp_update_jobstep`. Она выглядит следующим образом:

```
sp_update_jobstep [@job_id =] job_id, | [@job_name =] 'job_name',
    [@step_id =] step_id
    [, [@step_name =] 'step_name']
    [, [@subsystem =] 'subsystem']
    [, [@command =] 'command']
    [, [@additional_parameters =] 'parameters']
    [, [@cmdexec_success_code =] success_code]
    [, [@on_success_action =] success_action]
    [, [@on_success_step_id =] success_step_id]
    [, [@on_fail_action =] fail_action]
    [, [@on_fail_step_id =] fail_step_id]
    [, [@server =] 'server']
    [, [@database_name =] 'database']
    [, [@database_user_name =] 'user']
    [, [@retry_attempts =] retry_attempts]
    [, [@retry_interval =] retry_interval]
    [, [@os_run_priority =] run_priority]
    [, [@output_file_name =] 'file_name']
    [, [@flags =] flags]
```

Параметры у процедуры обновления такие же, как и при добавлении. С помощью параметров `@job_id` или `@job_name` процедура определяет задание, которое нужно изменить, а с помощью параметра `@step_id` задается номер изменяемого шага.

Давайте изменим второй шаг нашего задания, где мы выполняем системную команду удаления файла на копирование файла в каталог c:\backup. При этом имя файла, в который будет происходить копирование, должно состоять из текущей даты и расширения txt:

```
EXEC sp_update_jobstep
   @job_name = 'Тестовое задание 2',
   @step_id=2,
   @step_name = 'Команда копирования',
   @subsystem = 'CMDEXEC',
   @command = 'copy c:\text.txt c:\backup\[DATE].txt',
```

Изменяются только указанные в процедуре `sp_update_jobstep` параметры. Все остальные остаются без изменений, а не обнуляются.

Для удаления шага задания используется процедура `sp_delete_jobstep`, которая выглядит следующим образом:

```
sp_delete_jobstep [ @job_id = ] job_id , |
    [ @job_name = ] 'job_name' ,
    [ @step_id = ] step_id
```

С помощью параметров `@job_id` или `@job_name` процедура определяет задание, в котором нужно удалить шаг, а с помощью параметра `@step_id` задается номер удаляемого шага.

Следующий пример удаляет шаг 2, который выполняет системную команду:

```
EXEC sp_delete_jobstep
   @job_name = 'Тестовое задание 2',
   @step_id=2
```

Выполните следующую команду, чтобы убедиться, что шаг удален:

```
EXEC sp_help_job
    @job_name='Тестовое задание 2',
    @job_aspect='STEPS'
```

Обратите внимание, что идентификаторы автоматически перестроены. Теперь под номером 2 находится шаг, который был до этого под номером 3.

3.5.8. Эффективное использование заданий

Задания — достаточно эффективное средство для решения большинства рутинных задач администратора, и при этом их мощь не имеет границ. Но далеко не все администраторы используют задания хотя бы на 50% их возможностей. Большинство ограничивается резервным копированием базы данных с помощью планировщика системы, в крайнем случае реализует еще пару простых задач. При этом используется простейшая логика, а ведь существующие средства позволяют реализовать довольно сложную логику.

Давайте вспомним параметры, которые есть у шагов задания. Наиболее интересными являются: `@on_success_action`, `@on_success_step_id`, `@on_fail_action` и `@on_fail_step_id`. С их помощью можно построить логику выполнения довольно сложного задания. Например, у вас есть задание, выполняющее резервное копирование. В конце рабочего дня создается полная резервная копия базы, а в течение дня выполняется резервное копирование изменений (дифференцированное копирование или резервное копирование журнала). Более подробно о резервном копировании мы поговорим в *разд. 4.10*, а пока ограничимся общими понятиями.

Большинство напишет задание для создания полной резервной копии, состоящее из одного шага, ведь для решения задачи необходима только одна команда. А что произойдет, если данные разрушены, и сделать полную резервную копию нельзя? Задание завершится ошибкой, и придется вызывать администратора для решения проблемы. Но можно поступить более эффективно и выполнить следующее:

1. Первый шаг задания пытается создать полную резервную копию со следующими параметрами:
   - в параметре `@on_success_action` указываем значение 1, т. е. удачное завершение задания;
   - в параметре `@on_fail_action` устанавливаем значение 3, т. е. в случае ошибки переходим на следующий шаг.
2. Второй шаг будет выполняться, если на первом произошла ошибка. Раз данные нарушены, то пока администратор разбирается с проблемой, можно автоматически запустить резервное копирование изменений.

Администратор, узнав об ошибке, начнет восстанавливать данные, и для этого у него уже будет все готово, а именно: полная резервная копия за вчерашний день и все копии изменений за день, когда прошла ошибка. Администратору достаточно восстановить все резервные копии, и база данных будет готова к работе в кратчайший срок.

Данный пример выбран не случайно, потому что он интересует любого специалиста, который встречался с разрушением данных. Начальство всегда требует от нас, чтобы база данных была доступна постоянно, и не принимает никаких объяснений.

Если есть возможность автоматизировать какую-либо задачу, то необходимо сделать это. Но действовать нужно аккуратно. В примере с созданием резервной копии мы могли бы добавить в задание еще один шаг — при удачном выполнении перейти на третий шаг, где будет автоматически происходить восстановление данных, но эту операцию лучше не делать без вмешательства человека. Сначала сервер должен осмотреть специалист и убедиться, что данные действительно разрушены и требуется восстановление. В идеальном случае администратор должен еще и определить источник ошибки.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter3\job.sql на компакт-диске, прилагаемом к книге.

3.6. Планировщик заданий

Задания сами по себе слишком мало значат. Они представляют собой всего лишь наборы команд с возможностью построения логики, а мы пока умеем только выполнять задания вручную, а это идентично простому сценария на языке Transact-SQL и выполнению его в программе Query Analyzer. Если же мы научимся доверять наши задания планировщику, то это будет нечто действительно полезное, что невозможно сделать с помощью обычного файла сценария.

Создание задания — довольно трудоемкий процесс, но если задание будет выполняться по расписанию, то все затраты на создание самого задания и его шагов окупятся уменьшением затрат на сопровождение базы данных. Вы сможете уделять свое рабочее время другим задачам или заняться изучением чего-то нового. Хороший администратор — это тот, который 99% рабочего времени спит, а остальное время пьет кофе. Это является признаком того, что все работает, а администратор должен только наблюдать и вмешиваться в работу только в экстренных ситуациях. Штатные проблемы должны решаться автоматически, и это можно сделать именно с помощью заданий.

Планировщики, как и задания, выполняются службой SQL Service Agent, поэтому она должна быть запущена. Желательно, чтобы она запускалась автоматически после загрузки ОС.

3.6.1. Добавление плана выполнения

Для добавления задания в планировщик используется процедура `sp_add_jobschedule`, которая выглядит следующим образом:

```
sp_add_jobschedule [ @job_id = ] job_id, | [ @job_name = ] 'job_name',
    [ @name = ] 'name'
    [ , [ @enabled = ] enabled ]
    [ , [ @freq_type = ] freq_type ]
    [ , [ @freq_interval = ] freq_interval ]
    [ , [ @freq_subday_type = ] freq_subday_type ]
    [ , [ @freq_subday_interval = ] freq_subday_interval ]
    [ , [ @freq_relative_interval = ] freq_relative_interval ]
    [ , [ @freq_recurrence_factor = ] freq_recurrence_factor ]
    [ , [ @active_start_date = ] active_start_date ]
    [ , [ @active_end_date = ] active_end_date ]
    [ , [ @active_start_time = ] active_start_time ]
    [ , [ @active_end_time = ] active_end_time ]
```

Рассмотрим параметры процедуры:

- `@job_id` и `@job_name` — позволяют с помощью идентификатора или имени задания соответственно указать задание, которое необходимо выполнять с помощью планировщика по определенному расписанию;
- `@name` — имя нового планировщика;
- `@enabled` — если в этом параметре указать 1, то задание активно и будет выполняться по расписанию. Если 0, то планировщик не активен, и запуск задания будет возможен только вручную;
- `@freq_type` — определяет, как часто должно выполняться задание. Здесь можно указывать следующие значения:
  - 1 — однажды;
  - 4 — ежедневно;
  - 8 — еженедельно;
  - 16 — ежемесячно;
  - 32 — ежемесячно, относительно параметра `@freq interval`;
  - 64 — выполнять после запуска службы SQL Service Agent;
  - 128 — выполнять, когда компьютер нагружен минимально;
- `@freq_interval` — дни, в которые должно выполняться задание;

- `@freq_subday_type` — определяет, что указано в параметре `@freq_subday_interval`. Здесь может быть одно из следующих значений:
  - 1 — выполнять в указанное время;
  - 4 — в параметре `@freq_subday_interval` интервал в минутах между запусками задания;
  - 8 — в параметре `@freq_subday_interval` интервал в часах между запусками;
- `@freq_subday_interval` — интервал между запусками задания;
- `@freq_recurrence_factor` — количество недель или месяцев между запланированными выполнениями задания. Этот параметр действует, если `@freq_type` равен 8, 16 или 32;
- `@active_start_date` — дата начала выполнения задания. Дата указывается в виде числа в формате ГГГГММДД;
- `@active_end_date` — дата окончания выполнения задания. Дата указывается в виде числа в формате ГГГГММДД;
- `@active_start_time` — время начала выполнения задания в формате ЧЧММСС;
- `@active_end_time` — время окончания выполнения задания в формате ЧЧММСС.

На первый взгляд, все очень сильно усложнено, и разработчики немного перемудрили. Но все встанет на свои места после того, как мы рассмотрим несколько примеров создания планировщика. Но сначала необходимо дать некоторые пояснения параметра `@freq_interval`.

Смысл значения параметра `@freq_interval` зависит от `@freq_type`, поэтому тут требуется дополнительное пояснение:

- `@freq_type` равен 4 (ежедневное выполнение). В этом случае значение параметра `@freq_interval` содержит интервал в днях, через которые нужно выполнять задание;
- `@freq_type` равен 8 (еженедельное выполнение). В параметре `@freq_interval` содержится одно из следующих значений или их комбинация:
  - 1 — по воскресеньям;
  - 2 — по понедельникам;
  - 4 — по вторникам;
  - 8 — по средам;

- 16 — по четвергам;
- 32 — по пятницам;
- 64 — по субботам;

Если нужно, чтобы задание выполнялось по средам и четвергам, то необходимо указать число 24 (8 + 16);

- `@freq_type` равен 16. В параметре `@freq_interval` содержится день месяца, когда нужно выполнять задание;
- `@freq_type` равен 32. В параметре `@freq_interval` содержится:
  - 1 — в воскресенье;
  - 2 — в понедельник;
  - 3 — во вторник;
  - 4 — в среду;
  - 5 — в четверг;
  - 6 — в пятницу;
  - 7 — в субботу;
  - 8 — каждый день;
  - 9 — в рабочие дни;
  - 10 — в выходные дни.

В остальных случаях (при `@freq_type` равном 1, 64 или 128) значение в параметре `@freq_interval` не используется.

Если в параметре `@freq_type` указано значение 32, то в параметре `@freq_relative_interval` указывается, в какую неделю нужно выполнять задание. Здесь могут быть следующие значения:

- 1 — в первую неделю;
- 2 — во вторую неделю;
- 4 — в третью неделю;
- 8 — в четвертую неделю;
- 16 — в последнюю неделю.

Теперь рассмотрим простой пример создания планировщика для нашего тестового задания:

```
EXEC sp_add_jobschedule
   @job_name = 'Тестовое задание 2',
   @name = 'План выполнения тестового задания',
```

```
   @freq_type = 4,
   @freq_interval = 2,
   @active_start_date = 20050102,
   @active_start_time = 020300,
   @active_end_time = 30300
```

Параметр `@freq_type` равен 4, значит, выполнение задания будет происходить с интервалом в несколько дней. Параметр `@freq_interval` равен 2, а значит, выполнение будет происходить каждые два дня. Параметр `@active_start_date` задает начальную дату 2 января 2005 года, после которой задание станет активным и сможет выполняться.

Время выполнения с 2:03 (параметр `@active_start_time`). Именно в это время служба SQL Server Agent запустит задание, и оно должно закончить выполнение до 3:03 (параметр `@active_end_time`).

```
EXEC sp_add_jobschedule
   @job_name = 'Тестовое задание 2',
   @name = 'План выполнения тестового задания',
   @freq_type = 8,
   @freq_interval = 32,
   @freq_recurrence_factor = 2,
   @active_start_time = 020300
```

Интервал выполнения задан в неделях (параметр `@freq_type` равен 8), и задание будет выполняться по пятницам (параметр `@freq_interval` равен 32). Так как параметр `@freq_recurrence_factor` равен 2, выполнение будет происходить каждые две недели в 2 часа 3 минуты.

Рассмотрим еще один пример:

```
EXEC sp_add_jobschedule
   @job_name = 'Тестовое задание 2',
   @name = 'План выполнения тестового задания',
   @freq_type = 32,
   @freq_interval = 3,
   @freq_relative_interval = 2,
   @freq_recurrence_factor = 2,
   @active_start_time = 020300
```

Так как параметр `@freq_type` равен 32, то значение 3 в параметре `@freq_interval` указывает на то, что задание будет выполняться по вторникам. Параметр `@freq_relative_interval` со значением 2 означает, что выполнение будет во вторую неделю. Параметр `@freq_recurrence_factor` означает выполнение через каждые два месяца.

У одного задания может быть несколько планировщиков, например, один планировщик выполняет задание каждый понедельник, а другой — в первый день каждого месяца. Одним планировщиком такое реализовать невозможно, а вот с помощью двух достаточно просто.

3.6.2. Обновление планировщика

Для обновления планировщика используется процедура `sp_update_jobschedule`, которая выглядит следующим образом:

```
sp_update_jobschedule [@job_id =] job_id, |
    [@job_name =] 'job_name',
    [@name =] 'name'
    [, [@new_name =] 'new_name']
    [, [@enabled =] enabled]
    [, [@freq_type =] freq_type]
    [, [@freq_interval =] freq_interval]
    [, [@freq_subday_type =] freq_subday_type]
    [, [@freq_subday_interval =] freq_subday_interval]
    [, [@freq_relative_interval =] freq_relative_interval]
    [, [@freq_recurrence_factor =] freq_recurrence_factor]
    [, [@active_start_date =] active_start_date]
    [, [@active_end_date =] active_end_date]
    [, [@active_start_time =] active_start_time]
    [, [@active_end_time =] active_end_time]
```

С помощью параметров `@job_id` или `@job_name` процедура определяет задание, планировщик которого нужно изменить, а с помощью параметра `@name` задается обновляемый планировщик. Остальные параметры такие же, как и у процедуры создания планировщика. Пример использования:

```
EXEC sp_update_jobschedule
   @job_name = 'Тестовое задание 2',
   @name = 'План выполнения тестового задания',
   @freq_type = 32,
   @freq_interval = 4,
   @freq_relative_interval = 1,
   @freq_recurrence_factor = 3,
   @active_start_time = 043000
```

3.6.3. Удаление планировщика

Для удаления планировщика используется процедура `sp_delete_jobschedule`, которой нужно передать в параметре `@job_id` или `@job_name` задание, планировщик которого нужно удалить, а с помощью параметра `@name` задается удаляемый планировщик.

Пример использования:

```
EXEC sp_delete_jobschedule
   @job_name = 'Тестовое задание 2',
   @name = 'План выполнения тестового задания',
```

3.6.4. Информация о планировщике

Для получения информации о планировщиках используется процедура `sp_help_jobschedule`, которая выглядит следующим образом:

```
sp_help_jobschedule
    [ @job_id =] job_id |
    [ @job_name = ] 'job_name'
    [ , [ @schedule_name = ] 'schedule_name' ] |
    [ , [ @schedule_id = ] schedule_id ]
```

С помощью параметров `@job_id` или `@job_name` мы указываем задание, информацию о планировщиках которого мы хотим получить. С помощью параметров `@schedule_name` или `@schedule_id` можно указать конкретный планировщик. Если планировщик не указан, а только задание, то будут показаны все планировщики для данного задания.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter3\schedule.sql на компакт-диске, прилагаемом к книге.

3.7. Оповещения

Оповещения (alert) позволяют контролировать работу вашего сервера. Вы можете создавать оповещения на определенные события сервера (ошибки данных, ошибки запросов, ошибки или удачные выполнения заданий и т. д.) и направлять сообщения определенным операторам. Когда мы создаем задания, для них автоматически создаются оповещения, если указан оператор, который должен получать информацию об удачном или неудачном выполнении задания.

Благодаря оповещениям администратор может находиться в нескольких тысячах километров от SQL-сервера. Если произойдет нештатная ситуация, то администратор получит сообщение по электронной почте и сможет подключиться к серверу и исправить ошибку. Без оповещения администратору пришлось бы постоянно контролировать работу сервера и наблюдать за большим количеством параметров.

Планировщики, как и задания, выполняются службой SQL Service Agent, поэтому она должна быть запущена. Желательно, чтобы она запускалась автоматически после загрузки ОС.

3.7.1. Создание сообщения

Для создания собственных сообщений (message) используется процедура SQL-сервера `sp_addmessage`. В общем виде эта процедура выглядит следующим образом:

```
sp_addmessage [ @msgnum = ] msg_id ,
    [ @severity = ] severity ,
    [ @msgtext = ] 'msg'
    [ , [ @lang = ] 'language' ]
    [ , [ @with_log = ] 'with_log' ]
    [ , [ @replace = ] 'replace' ]
```

Рассмотрим параметры процедуры:

- `@msgnum` — номер (идентификатор) сообщения, который должен начинаться с 500001;
- `@severity` — уровень критичности. Он определяется по тем же правилам, как и у функции `RAISERROR()`;
- `@msgtext` — текст сообщения, максимальный размер которого 255 символов;
- `@lang` — язык сообщения. По умолчанию используется нулевое значение и язык, установленный в системе;
- `@with_log` — нужно ли писать о событии в журнал сообщений Windows. Если в этом параметре указано `true`, то в журнал будет записано сообщение об ошибке. Если `false`, сообщение не обязательно будет записано в журнал, тут уже все зависит от того, как оно было сгенерировано;
- `@replace` — если сообщение с указанным номером существует, то в этом параметре можно указать команду `REPLACE`. Это означает, что существующую ошибку с указанным номером надо заменить.

Создадим свое сообщение:

```
EXEC sp_addmessage 60001, 16,
   'Ошибка добавления записи'
```

Для удаления сообщения используется процедура `sp_dropmessage`:

```
sp_dropmessage [ @msgnum = ] message_number
    [ , [ @lang = ] 'language' ]
```

Процедуре передается два параметра: номер удаляемого сообщения и язык. Если язык не указан, то будут удалены сообщения для всех национальных языков, существующих в сообщении.

3.7.2. Создание оповещения

Для создания оповещения используется процедура `sp_add_alert`, которая выглядит следующим образом:

```
sp_add_alert [ @name = ] 'name'
 [,[@message_id = ] message_id ]
 [,[@severity = ] severity ]
 [,[@enabled = ] enabled ]
 [,[@delay_between_responses = ] delay_between_responses ]
 [,[@notification_message = ] 'notification_message' ]
 [,[@include_event_description_in = ] include_event_description_in ]
 [,[@database_name = ] 'database' ]
 [,[@event_description_keyword=]'event_description_keyword_pattern']
 [,{[@job_id = ] job_id | [ @job_name = ] 'job_name' } ]
 [,[@raise_snmp_trap = ] raise_snmp_trap ]
 [,[@performance_condition = ] 'performance_condition' ]
 [,[@category_name = ] 'category' ]
```

Рассмотрим параметры этой процедуры:

- `@name` — имя оповещения, которое должно быть уникальным, и по нему система будет в дальнейшем идентифицировать оповещение;
- `@message_id` — номер ошибки, на которую будет создано оповещение. В MS SQL Server предопределено достаточно много ошибок, на которые вы можете создать оповещения. Чтобы увидеть их, просмотрите таблицу `sysmessages` в базе данных master:

  ```
  SELECT *
  FROM master..sysmessages
  ```

  В *разд. 3.7.1* мы увидели, как создавать собственные сообщения об ошибках;

- `@severity` — число от 1 до 25, определяющее уровень критичности оповещения. Если вы указали параметр `@message_id`, то параметр `@severity` должен быть равен нулю;
- `@enabled` — если параметр равен 1, то оповещение является активным, иначе (если равно нулю) оно не будет генерироваться, и операторы не получат уведомление;
- `@delay_between_responses` — задержка в секундах между событием и действием на это событие. В качестве действия может быть одно или более уведомлений на электронный адрес или пейджер оператора или выполнение определенного задания. По умолчанию задержки нет (значение 0), и действие произойдет сразу после генерации оповещения;
- `@notification_message` — в этом параметре вы можете задать дополнительный текст, который будет добавлен к сообщению, отправляемому оператору;
- `@include_event_description_in` — параметр определяет, куда необходимо добавлять сообщение. В этом параметре можно указывать одно из следующих значений (или сумму):
  - 0 — никуда;
  - 1 — к сообщению, отправляемому по электронной почте;
  - 2 — к сообщению на пейджер;
  - 4 — к сетевому сообщению.

  Например, если вы хотите, чтобы текст добавлялся и к сообщению электронной почты, и к сетевому сообщению, то необходимо указать число 5 (сумма чисел 1 и 4);
- `@database_name` — в этом параметре можно задать базу данных. Если этот параметр не задан, то сообщение будет генерироваться для всех баз данных;
- `@job_id` — позволяет указать идентификатор задания, которое должно выполняться в ответ на оповещение. Если этот параметр указан, то нельзя указывать `@job_name`;
- `@job_name` — позволяет указать имя задания, которое должно выполняться в ответ на оповещение. Если этот параметр указан, то нельзя указывать `@job_id`;
- `@raise_snmp_trap` — не используется;
- `@performance_condition` — оповещения могут создаваться для параметров производительности, например, генерация сообщения в случае превышения сервером загрузки в 90%. Если вы создаете такое сообщение,

то в этом параметре вы можете указать параметр, условие и значение. В качестве параметра может использоваться объект производительности, счетчик производительности или имя экземпляра счетчика. В качестве условия могут быть знаки больше, меньше или равно. Значение — это числовое значение счетчика;

- `@category_name` — имя категории оповещения.

Процедура должна выполняться в базе данных msdb, потому что процедура принадлежит этой базе данных.

Примеры использования оповещений

Добавим собственное оповещение:

```
EXEC sp_add_alert
   @name = 'Тестовое оповещение',
   @message_id = 60001,
   @severity = 0,
   @include_event_description_in = 7,
   @notification_message = 'Было сгенерировано сообщение 60001'
```

В данном примере мы создаем оповещение с названием `Тестовое оповещение`, которое будет реагировать на сообщение с номером 60001. Сообщение с таким номером было создано нами в *разд. 3.7.1*.

Теперь посмотрим, как создать оповещение для уже существующих сообщений. В таблице `sysmessages` базы данных master я нашел сообщение с номером 9002, которое генерируется в случае, если файл журнала полный. Давайте рассмотрим, как с помощью оповещения и задания решить классическую задачу увеличения файла базы данных и очистки журнала.

Как мы уже знаем, файлы базы данных и файлы журналов могут увеличиваться бесконечно, но это далеко не всегда удобно. Многие администраторы хотят контролировать выделяемое сервером дисковое пространство, поэтому отказываются от автоматического увеличения. Для журнала транзакций можно не увеличивать размер файла, а уменьшать его с помощью очистки. Получается, что наша задача — создать оповещение, которое будет реагировать на переполнение журнала и запускать задание очистки журнала. Это можно сделать следующим образом:

```
EXEC sp_add_alert
  @name = 'Файл журнала полный',
  @message_id = 9002,
  @severity = 0,
  @include_event_description_in = 4,
```

```
  @database_name = 'FlenovSQLBook',
  @job_name = 'Задание очистки журнала',
```

Теперь создадим задание, которое будет очищать журнал. Да, мы еще не рассматривали резервное копирование, которое необходимо для решения данной задачи (эта тема будет обсуждаться в *гл. 4*). Если вы не знакомы с этими командами, то не обращайте внимания на операторы Transact-SQL. Вернитесь к ним после прочтения *гл. 4*. Сейчас наша задача — создать задание, а заодно лишний раз потренироваться и закрепить пройденный материал на практике. Итак, задание и два шага, выполняющих резервное копирование и усечение журнала, вы можете увидеть в листинге 3.9.

Листинг 3.9. Создание задания резервного копирования журнала из двух шагов

```
-- Создание задания резервного копирования и очистки журнала
EXECUTE sp_add_job
  @job_name = 'Задание очистки журнала',
  @enabled = 1,
  @owner_login_name = 'sa',
  @description = 'Задание, которое резервирует и очищает журнал',
  @notify_level_eventlog = 2,
  @notify_level_email = 3,
  @notify_level_netsend = 3,
  @delete_level= 0

-- Добавить шаги
EXECUTE sp_add_jobstep
  @job_name = 'Задание очистки журнала',
  @step_id = 1,
  @step_name = 'Резервное копирование журнала',
  @subsystem = 'TSQL',
  @command = 'BACKUP LOG FlenovSQLBook to disk =
      ''C:\Backup\FlenovSQLBook_log.bak''',
  @server = '',
  @database_name = 'FlenovSQLBook'

EXECUTE sp_add_jobstep
   @job_name = 'Задание очистки журнала',
   @step_id = 2,
   @step_name = 'Усечение журнала',
```

```
    @command = 'Усечение журнала',
    @database_name = 'master',
    @subsystem = 'TSQL',
    @command = 'BACKUP LOG FlenovSQLBook WITH TRUNCATE_ONLY'
```

Сообщение 9002 является критичным и генерируется, когда журнал уже полный, и сервер не смог сохранить в нем информацию о транзакции. Если нет автоматического увеличения, то серверу негде будет хранить информацию о транзакции, и такая транзакция откатывается, а значит, данные не будут сохранены. Так будет со всеми последующими транзакциями, пока в журнале не появится свободное пространство, следовательно, данный метод увеличения журнала нельзя назвать эффективным, поскольку с момента переполнения журнала до момента освобождения пространства в журнале сервер будет доступен только для чтения. Именно поэтому освобождением должно заниматься задание, которое будет выполняться мгновенно, без задержек, а не ждать, пока в работу сервера вмешается администратор.

Файл данных просто так обрезать не получится. Вы же не можете просто так удалить все данные из таблиц и начать их заполнять с нуля. В этом случае увеличение должно происходить, но все же может существовать задание, которое будет выполнять увеличение с помощью запроса Transact-SQL и выделять пространства там, где предусмотрено в сценарии администратором.

Обновление оповещения

Для обновления оповещения используется процедура `sp_update_alert`. У этой процедуры такие же параметры, как и у процедуры создания оповещения `sp_add_alert`. Общий вид процедуры довольно громоздкий, содержит много параметров, и мы не будем его рассматривать ради экономии места.

Для удаления оповещения используется процедура `sp_delete_alert`, которой нужно передать в качестве параметра только имя удаляемого оповещения, например, так:

```
EXEC sp_delete_alert 'Файл журнала полный'
```

Получение информации об оповещении

Для получения информации об оповещениях используется процедура `sp_help_alert`, которая выглядит следующим образом:

```
sp_help_alert [ [ @alert_name = ] 'alert_name' ]
    [ , [ @order_by = ] 'order_by' ]
    [ , [ @alert_id = ] alert_id ]
    [ , [ @category_name = ] 'category' ]
```

У этой процедуры четыре параметра, и все они являются необязательными. Рассмотрим каждый параметр:

- `@alert_name` — имя сообщения, информацию о котором необходимо получить;
- `@order_by` — отсортировать список по определенному параметру;
- `@alert_id` — идентификатор сообщения, информацию о котором необходимо получить;
- `@category_name` — имя категории.

Если выполнить процедуру `sp_help_alert` без параметров, то результатом будут все оповещения SQL-сервера:

```
EXEC sp_help_alert
```

Результатом будет таблица, состоящая из полей, которые идентичны по имени и назначению параметрам, которые мы рассматривали у процедуры добавления оповещения `sp_add_alert`.

Отобразим тот же список, но отсортируем результирующий список по параметру `message_id`:

```
EXEC sp_help_alert @order_by='message_id'
```

3.7.3. Создание уведомления

У нас есть оповещение и есть сообщение. Нам необходимо связать эти два объекта, чтобы в ответ на оповещение генерировалось сообщение и направлялось оператору. Такую связь можно назвать уведомлением, и для его создания используется процедура `sp_add_notification`, которая выглядит следующим образом:

```
sp_add_notification [ @alert_name = ] 'alert' ,
    [ @operator_name = ] 'operator' ,
    [ @notification_method = ] notification_method
```

Здесь три параметра:

- `@alert_name` — имя оповещения;
- `@operator_name` — имя оператора, который должен получать уведомление;
- `@notification_message` — метод, которым оператор будет получать уведомления об ошибке:
  - 1 — на электронный адрес;
  - 2 — на пейджер;
  - 4 — командой `NET SEND`.

Может быть несколько методов получения уведомления. Для этого в параметре `@notification_message` нужно указать сумму значений методов. Например, если нужно отправить оператору сообщение по электронной почте и на пейджер, то в параметре `@notification_message` указываем значение 3 (1+2).

Примеры использования уведомлений

Прежде чем создавать уведомление, добавим оператора:

```
exec sp_add_operator
   @name = 'Администратор',
   @netsend_address ='192.168.77.11'
```

Чтобы увидеть результат, я задал IP-адрес своего компьютера. Это позволит мне получать сообщение по сети.

Следующий пример показывает, как создать уведомление, которое будет получать администратор для оповещения, созданного в *разд. 3.7.2*:

```
EXEC sp_add_notification
  @alert_name='Тестовое оповещение',
  @operator_name='Администратор',
  @notification_method=4
```

Одно оповещение может направлять сообщения нескольким операторам. Следующий пример добавляет уведомление еще одного оператора для оповещения с именем `Тестовое оповещение`:

```
EXEC sp_add_notification
  @alert_name='Тестовое оповещение',
  @operator_name='Михаил',
  @notification_method=4
```

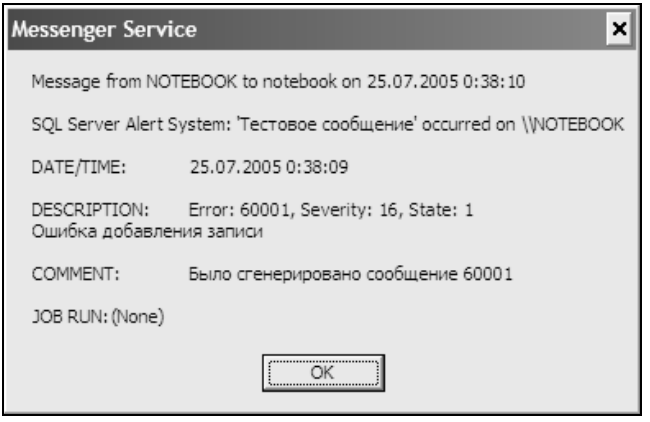


Рис. 3.4. Пример уведомления

Вот теперь вы можете увидеть результат работы на примере. Для этого необходимо сгенерировать сообщение с помощью функции `RAISERROR()` (более подробно о ней мы поговорим в *разд. 4.3.2*):

```
RAISERROR (60001, 16, 1)
```

В ответ на это я получил сетевое сообщение, пример которого можно увидеть на рис. 3.4.

Если вы используете сетевые сообщения, то убедитесь, что в вашей ОС запущена служба Messenger, иначе отправка сообщения будет невозможной.

Обновление уведомления

Для обновления уведомления используется процедура `sp_update_notification`, которая выглядит следующим образом:

```
sp_update_notification [@alert_name =] 'alert',
    [@operator_name =] 'operator',
    [@notification_method =] notification
```

Параметры такие же, как и при создании уведомления `sp_add_notification`, только параметр `@alert_name` определяет оповещение, которое надо обновить, а параметр `@operator_name` определяет оператора. С помощью параметра и `@notification_method` можно задать новый метод уведомления.

Удаление уведомления

Для удаления уведомления используется процедура `sp_delete_notification`, которая в общем виде выглядит следующим образом:

```
sp_delete_notification [ @alert_name = ] 'alert' ,
    [ @operator_name = ] 'operator'
```

Параметр `@alert_name` определяет оповещение, которое надо удалить, а параметр `@operator_name` определяет удаляемого оператора.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter3\alert.sql на компакт-диске, прилагаемом к книге.

Глава 4

Дополнительные возможности Transact-SQL

Возможности языка Transact-SQL настолько широки, что трудно описать их все даже коротко в одной книге. Одних только свойств у объектов и таблиц столько, что необходимо писать отдельный справочник. В данной главе мы будем рассматривать более сложные запросы и возможности, которые предоставляет MS SQL Server.

Начнем мы с рассмотрения свойств, с помощью которых можно влиять на обработку сервером запросов и добиться более эффективного решения задачи. В этой главе нас ждет множество интересных SQL-запросов, которые помогут вам в решении типовых задач.

4.1. Свойства сервера

У сервера MS SQL достаточно много параметров, изменение которых влияет на обработку запроса сервером. Но есть одно ограничение — эти параметры действуют только в течение одного сеанса. Если после изменения параметра отключиться от сервера и подключиться заново, то параметр снова устанавливается в значение по умолчанию.

Параметры устанавливаются с помощью оператора `SET`. В общем виде он имеет следующий формат:

```
SET ИМЯ ЗНАЧЕНИЕ
```

где:

- `ИМЯ` — имя параметра;
- `ЗНАЧЕНИЕ` — значение, которое необходимо установить.

4.1.1. Ограничение выводимых строк

В *разд. 2.2.2* мы научились ограничивать количество возвращаемых сервером значений с помощью оператора `TOP`. Есть еще один способ ограничить вывод — использование параметра `ROWCOUNT`:

```
SET ROWCOUNT 10
```

В данном запросе мы устанавливаем максимальное значение выдаваемых сервером строк, делая его равным 10. Теперь, из какой бы таблицы мы ни выбирали данные, мы увидим не более 10 строк.

Помимо глобального действия параметра `ROWCOUNT` на все запросы (вне зависимости от таблицы) есть еще одно отличие от оператора `TOP`. Рассмотрим следующий запрос:

```
SELECT TOP 10 *
FROM tbPeoples
ORDER BY vcFamil
```

Здесь мы выбираем 10 первых строк из таблицы `tbPeoples`, отсортированных по полю фамилии. Сервер выберет все данные из таблицы, отсортирует их и вернет нам первые 10 строк.

При выполнении следующего запроса без указания оператора `TOP` будут возвращены все записи.

```
SELECT *
FROM tbPeoples
ORDER BY vcFamil
```

А что получится, если выполнить такой запрос:

```
SET ROWCOUNT 10
SELECT *
FROM tbPeoples
ORDER BY idPeoples
```

Результат снова будет равен 10 строкам, но теперь любые запросы будут возвращать 10 строк, пока мы не подключимся к серверу заново или не изменим свойство `ROWCOUNT`. Чтобы отключить ограничение вывода, можно изменить свойство `ROWCOUNT`, установив его в 0. Например:

```
SET ROWCOUNT 0
```

Теперь, если запросить все данные с сервера, будут возвращены все строки.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\set.sql на компакт-диске, прилагаемом к книге.

4.1.2. Управление неявными транзакциями

Иногда бывает необходимо поместить соединение в неявную транзакцию. Это делается с помощью оператора `IMPLICIT_TRANSACTIONS`:

```
SET IMPLICIT_TRANSACTIONS { ON | OFF }
```

Чтобы поместить соединение в неявную транзакцию, выполните следующую команду:

```
SET IMPLICIT_TRANSACTIONS ON
```

Когда режим неявной транзакции включен, выполнение любых из следующих операторов запускает транзакцию: `ALTER TABLE`, `INSERT`, `CREATE`, `OPEN`, `DELETE`, `REVOKE`, `DROP`, `SELECT`, `FETCH`, `TRUNCATE TABLE`, `GRANT`, `UPDATE`.

При этом не поддерживаются вложенные транзакции. Если для соединения уже открыта транзакция, оператор не может начать новую транзакцию.

Что значит "неявная транзакция"? Когда вы выполняете один из операторов, перечисленных выше, запускается транзакция, которую вы должны явно завершить с помощью вызова оператора `COMMIT TRANSACTION`. После завершения транзакции, когда вы снова вызываете один из операторов обновления, запускается еще одна — новая транзакция.

Когда вы подключаетесь к серверу с помощью драйверов Microsoft OLE DB Provider for SQL Server или SQL Server ODBC, то параметр `IMPLICIT_TRANSACTIONS` автоматически устанавливается в `OFF`. В этом случае вы сами должны начинать транзакции.

Следующий параметр `REMOTE_PROC_TRANSACTIONS` позволяет настроить поведение сервера при вызове удаленных процедур. Если этот параметр установлен, то при каждом вызове удаленной процедуры сервер RPC SQL Server (RPC — remote procedure call, удаленный вызов процедур) начинает новую распределенную транзакцию.

Распределенные запросы мы будем рассматривать в *разд. 5.1*, но сейчас достаточно понимать, что сервер SQL позволяет вызывать код с других SQL-серверов, что иногда может оказаться очень удобным решением. За выполнение таких запросов отвечает MS Distributed Transaction Server (сервер распределенных транзакций), который управляет удаленным вызовом.

Если параметр `REMOTE_PROC_TRANSACTIONS` отключен, то автоматического создания транзакции не произойдет. Для установки параметра выполните следующий код:

```
SET REMOTE_PROC_TRANSACTIONS ON
```

А для снятия параметра необходимо выполнить:

```
SET REMOTE_PROC_TRANSACTIONS OFF
```

4.1.3. Управление блокировками

При рассмотрении транзакций мы уже затронули тему блокировок. Пока выполняется транзакция, данные в таблицах не изменяются, а только блокируются. Вместо физического изменения происходит запись в журнал транзакций о необходимости произвести эти изменения. Когда мы подтверждаем транзакцию, сервер может физически изменить данные в таблице.

MS SQL Server использует динамическую архитектуру блокировок для определения наиболее эффективной блокировки. Он автоматически определяет, какая блокировка наиболее подходит во время выполнения запроса, основываясь на характеристиках схемы и запроса.

Сервер SQL автоматически увеличивает и уменьшает детализацию и типы блокировок. Оптимизатор запросов обычно выбирает корректную блокировку, детализированную на момент компилирования плана выполнения, что минимизирует необходимость изолирования блокировок.

Для примера, если требуется обновление большого количества блокировок уровня строк, и заблокирован значительный процент таблицы, блокировка уровня строк повышается до блокировки уровня таблицы.

Сервер SQL позволяет контролировать опции блокировки на уровне сеанса с помощью установки уровня изолированности транзакций. Для управления блокировками используется оператор `SET TRANSACTION ISOLATION LEVEL`. Общий вид оператора:

```
SET TRANSACTION ISOLATION LEVEL
 {
    READ COMMITTED
  | READ UNCOMMITTED
  | REPEATABLE READ
  | SERIALIZABLE
 }
```

Параметр `ISOLATION LEVEL` можно отнести как к управлению транзакциями, так и к управлению блокировками, потому что воздействие параметра происходит и на то, и на другое.

Давайте рассмотрим опции, которые можно устанавливать:

- `READ COMMITED` — указывает на использование разделяемых блокировок для чтения. На этом уровне вы не можете использовать "грязное" чтение;
- `READ UNCOMMITED` — на этом уровне можно читать еще не подтвержденные изменения;

- `REPEATABLE READ` — указывает, что "грязное" чтение и неповторяющееся чтение не может использоваться. Чтение блокируется и удерживается, пока транзакция не завершится;
- `SERIALIZABLE` — не допускает других пользователей к обновлению и вставке новых строк.

Следующий пример устанавливает уровень изолированности для текущего сеанса в `READ UNCOMMITTED`:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

Давайте посмотрим, как можно использовать эту опцию. Вспомним пример из *разд. 2.19*, когда мы в одном соединении начинали транзакцию и удаляли данные. Без завершения транзакции в другом окне (другой сеанс) мы пытались прочитать данные, и запрос зависал, пока транзакция не завершится. Давайте повторим этот эксперимент. Создадим новую транзакцию и удалим все записи из таблицы товаров:

```
BEGIN TRANSACTION
DELETE Товары
```

Теперь создадим новое соединение и в этом окне попытаемся прочитать данные:

```
SELECT * FROM Товары
```

Конечно же, запрос зависнет. Прервите его выполнение. Не прерывая транзакции и при наличии блокировки в первом окне, во втором перейдем в режим "грязного" чтения:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

Теперь нам станут доступны данные незавершенных транзакций. Повторите выборку, и она закончится удачно. Вы увидите, что таблица пуста, хотя транзакция еще не завершена, а значит, это не может быть точным результатом.

Откатите транзакцию в первом окне, чтобы не удалять данные. Возможно, что они нам еще пригодятся.

Я не рекомендую вам использовать "грязное" чтение (незавершенных транзакций), потому что нельзя гарантировать, что полученные данные являются абсолютно точными. "Грязное" чтение необходимо только во время отладки, чтобы со стороны можно было видеть работу сценария.

С помощью оператора `SET LOCK_TIMEOUT` можно установить максимальное количество времени, которое SQL Server позволяет транзакции ожидать освобождения заблокированных ресурсов.

```
SET LOCK_TIMEOUT период
```

Период указывает количество миллисекунд. Если указать –1 (значение по умолчанию), то ожидание будет бесконечным. После изменения этого значения его эффект сохраняется до конца сеанса. При повторном подключении к серверу снова устанавливается значение по умолчанию.

Следующий пример устанавливает максимальное время ожидания в три минуты:

```
SET LOCK_TIMEOUT 180000
```

Для определения значения текущего сеанса запросите глобальную переменную сервера `@@LOCK_TIMEOUT`:

```
SELECT @@LOCK_TIMEOUT
```

Может возникнуть *мертвая блокировка*: две транзакции заблокировали отдельные объекты и ждут, когда каждая из них освободит другой объект. Это может произойти, когда различные продолжительные транзакции выполняются параллельно на одной и той же базе данных.

Сервер SQL завершает мертвые блокировки с помощью автоматического прерывания одной из транзакций. Процесс SQL-сервера выполняет следующие действия.

- Откатывает транзакцию, являющуюся жертвой мертвой блокировки.
- Отправляет приложению-жертве мертвой блокировки сообщение 1205.
- Отменяет текущий запрос жертвы мертвой блокировки.
- Позволяет другим транзакциям продолжать работу.

В многопользовательской системе каждый клиент должен регулярно проверять сообщение 1205, которое указывает на откат транзакции. Если найдено это сообщение, приложение должно повторить попытку снова.

Хотя это не всегда возможно, вы должны минимизировать риск возникновения мертвых блокировок. Для этого, конечно же, нужно добиваться минимизации времени выполнения транзакции. Чем быстрее она работает, тем меньше вероятность, что два пользователя одновременно заблокируют один и тот же ресурс. Старайтесь внутри транзакции использовать ресурсы в одной и той же последовательности.

Если транзакция создала мертвую блокировку в ожидании ресурса, сервер прерывает одну из транзакций без времени ожидания.

Оператор `SET LOCK_TIMEOUT` позволяет приложениям установить максимальное количество времени, которое оператор ждет заблокированный ресурс, прежде чем заблокированный оператор автоматически отменится. Такая отмена не откатывает транзакцию и не отменяет ее. Приложение должно

отслеживать сообщение и выполнять восстановительные действия, такие как повторная отправка транзакции или отмена.

Вы можете использовать программу Enterprise Manager или процедуру `sp_lock` для отображения отчета об активных блокировках. Просто выполните команду `sp_lock`. Результат показан на рис. 4.1.

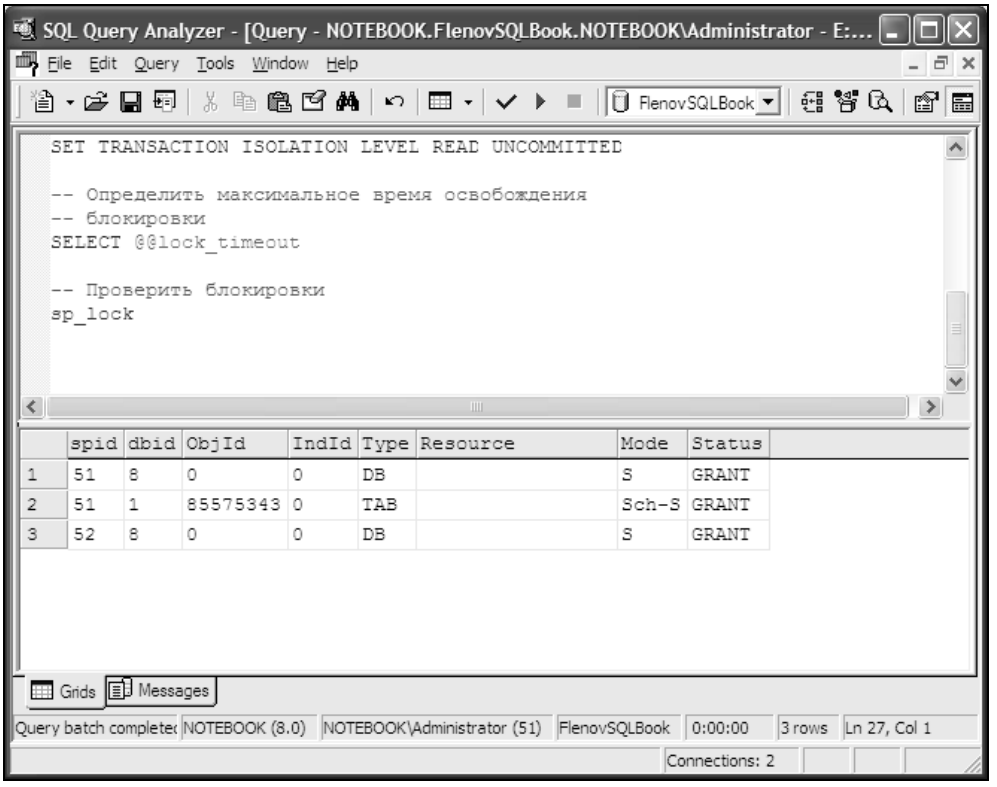


Рис. 4.1. Результат выполнения команды `sp_lock`

В результате в первой колонке находится идентификатор процесса. Чтобы уничтожить зависший процесс, нужно выполнить команду `kill` и указать номер процесса. Например, следующая команда закрывает процесс с номером 52:

```
kill 52
```

Можно использовать SQL Profiler для получения информации на указанный набор транзакций. Вы также можете использовать системный монитор для просмотра истории блокировок.

Следующий параметр MS SQL Server, который поможет вам в управлении блокировками, — DEADLOCK_PRIORIRY. Допустим, что один пользователь заблокировал страницу данных *X*, а другой пользователь заблокировал страницу *Y*. При этом первому пользователю для продолжения работы нужна страница *Y*, а второму нужна страница *X*. Получается эффект взаимной блокировки, который будет снят только по прошествии максимального времени блокировки. Но какую из блокировок отбросит сервер, чтобы другой пользователь смог завершить свою работу? Это уже решает сам сервер баз данных, но мы можем немного повлиять на процесс выбора через параметр DEADLOCK_PRIORIRY.

Серверный параметр DEADLOCK_PRIORIRY задает приоритет вашему соединению при выборе блокировки, которую необходимо отклонить. Можно установить одно из следующих значений:

- LOW — при этом значении повышается вероятность, что сервер отклонит именно ваше соединение;
- NORMAL — нормальный режим, при котором права на дальнейшее выполнение у всех практически равны.

Если вы выполняете запрос невысокой важности, но который может вызвать блокировку, то рекомендуется установить приоритет равным LOW:

```
SET DEADLOCK_PRIORIRY LOW
```

В этом случае с большей вероятностью будет прерван именно ваш запрос, но при этом более важная программа (например, формирования отчетности) сможет корректно завершить свою работу.

4.1.4. Управление датой

У MS MQL Server есть два параметра, которые вы можете использовать для управления датой: DATEFIRST и DATEFORMAT. Конечно же, оба параметра устанавливаются с помощью оператора SET и могут пригодиться вам при написании приложений.

Параметр DATEFIRST задает день недели, который будет первым. Дело в том, что в разных национальных стандартах первым днем недели может быть не понедельник, как в России, а воскресенье. Команда изменения первого дня недели выглядит следующим образом:

```
SET DATEFIRST число
```

Здесь число изменяется от 1 до 7. Если указать 1, то первым днем будет считаться понедельник, 2 — вторник и т. д. По умолчанию сервер устанавливает

первый день недели — воскресенье, т. е. параметр равен 7. Чтобы узнать текущее значение, воспользуйтесь функцией `@@datefirst`:

```
SELECT @@datefirst
```

Следующий параметр SQL-сервера, который нам предстоит рассмотреть, — `DATEFORMAT`. Он изменяет формат даты. Команда изменения формата выглядит следующим образом:

```
SET DATEFORMAT формат
```

Формат задается тремя буквами:

- `m` — месяц;
- `d` — день;
- `y` — год.

По умолчанию сервер использует формат `mdy`, т. е. первым идет месяц, потом день и, наконец, год. Если вам неудобно работать в таком формате, то можно изменить его на более привычный `dmy`:

```
SET DATEFORMAT dmy
```

4.1.5. Объединение с *NULL*

Тот факт, что в SQL значение `NULL` интерпретируется как пустое значение и оно не равно пустой строке или числу 0, иногда бывает действительно удобным решением, а иногда создает проблемы. Дело в том, что если сложить строку с `NULL`, то в результате будет `NULL`.

Допустим, что у нас есть таблица сотрудников фирмы с тремя полями: `Фамилия`, `Имя` и `Отчество`. Выполним запрос, который объединяет все три поля:

```
SELECT Фамилия+' '+Имя+' '+Отчество
FROM Таблица
WHERE ...
```

Если хотя бы одно из полей в какой-либо строке не будет заполнено, то результатом сложения будет `NULL`, что явно некорректно.

Поведение сервера при объединении строк можно контролировать через серверный параметр `CONCAT_NULL_YIELDS_NULL`. Вы должны учитывать, что этот флаг есть как у подключения, так и у базы данных. Если этот параметр подключения установить в значение `ON`, то при сравнении строки с `NULL` результатом будет `NULL`. Если установить параметр в значение `OFF`, то результат определяется настройкой `CONCAT_NULL_YIELDS_NULL` базы данных.

Пример установки параметра:

```
SET CONCAT_NULL_YIELDS_NULL OFF
```

4.1.6. Запрет на подсчет строк

Сервер SQL может выводить информацию о том, сколько записей участвовало в выполнении команды. Иногда это полезно, но иногда только мешает. Чтобы отключить подсчет строк, выполните команду:

```
SET NOCOUNT OFF
```

Чтобы восстановить возможность подсчета, выполните команду:

```
SET NOCOUNT ON
```

Необходимо учитывать, что команда не влияет на работу функции `@@ROWCOUNT`, которая возвращает количество записей, обработанных последним запросом.

4.1.7. Закрытие курсора

Параметр `CURSOR_CLOSE_ON_COMMIT` присутствует среди параметров как подключения к базе данных, так и самой базы. Он отвечает за поведение курсора при завершении транзакции, а именно, нужно ли закрывать курсор по завершении транзакции. Если параметр соединения установлен в `ON`, то при завершении транзакции с помощью `COMMIT TRANSACTION` или `ROLLBACK TRANSACTION` курсоры закрываются. Иными словами, поведение курсора определяется соответствующей настройкой базы данных, к которой вы подключены.

Следующий пример показывает, как можно включить закрытие курсора:

```
SET CURSOR_CLOSE_ON_COMMIT ON
```

Чтобы отключить параметр, необходимо заменить `ON` на `OFF`.

4.1.8. План выполнения

С помощью параметра `FORCEPLAN` вы можете повлиять на порядок объединения таблиц. Если флаг включен, то сервер объединяет таблицы в порядке их перечисления, а если выключен, то MS SQL Server имеет право выбрать оптимальный вариант объединения.

Иногда сервер ошибается, и вы на опыте убеждаетесь, что при объединении в другом порядке запрос выполняется быстрее. В таком случае параметр можно включить. Но все же ошибки происходят редко, поэтому лучше довериться серверу.

Следующий пример показывает, как можно выключить параметр:

```
SET FORCEPLAN OFF
```

Есть еще серверные параметры SHOWPLAN_ALL, SHOWPLAN_TEXT и STATISTIC IO, которые позволяют непосредственно увидеть план выполнения и логику, по которой сервер выбрал именно указанный план, но с ними мы подробно познакомимся в *разд. 5.2.1*, когда будем говорить об оптимизации.

4.1.9. Соответствие ANSI

Большинство баз данных не во всем следуют рекомендациям стандарта ANSI SQL, и в этом мы уже не раз убеждались. MS SQL Server также идет на нарушения стандарта, дабы сделать работу с базами более удобной и более мощной. Но все же, у нас есть возможность с помощью оператора SET настроить сервер так, чтобы он работал максимально близко к стандарту.

Параметр *ANSI_DEFAULTS*

Серверный параметр ANSI_DEFAULTS включает (ON) и выключает (OFF) целую группу параметров. Чтобы включить параметр, выполните команду:

```
SET ANSI_DEFAULTS ON
```

При этом сервер включит следующие параметры:

- ANSI_NULL_DFLT_ON;
- ANSI_NULLS;
- ANSI_PADDING;
- ANSI_WARNINGS;
- CURSOR_CLOSE_ON_COMMIT;
- IMPLICIT TRANSACTION;
- QUOTED_IDENTIFIER.

Некоторые из перечисленных параметров нам уже знакомы, а некоторые нам предстоит рассмотреть далее в этой главе.

Если установить параметр в значение OFF, то эти же параметры будут отключены.

```
SET ANSI_DEFAULTS OFF
```

Параметр *ANSI_NULL_DFLT*

Серверные параметры ANSI_NULL_DFLT_ON и ANSI_NULL_DFLT_OFF определяют, можно ли устанавливать NULL по умолчанию для полей. Если установить значение ANSI_NULL_DFLT_OFF, то нулевые значения не допускаются, иначе NULL разрешено.

Следующий пример показывает, как включить параметр ANSI_NULL_DFLT_ON:

```
SET ANSI_NULL_DFLT_ON ON
```

При установке одного из этих флагов другой автоматически сбрасывается.

Параметр *ANSI_NULLS*

Если следовать стандарту ANSI SQL-92, то при сравнении поля с нулевым значением всегда будет результат false, потому что необходимо использовать конструкцию IS NULL. Чтобы разрешить сравнение с NULL, можно установить серверный параметр ANSI_NULLS в значение OFF:

```
SET ANSI_NULLS OFF
```

Теперь в условии WHERE вы можете вместо ПОЛЕ IS NULL писать сравнение ПОЛЕ = NULL. Да, использование знака равенства более наглядно и привычно, но программисту лучше следовать стандарту. В данном случае нарушение требований ANSI SQL является неоправданным, потому что вызовет проблемы при переходе на другие базы данных и может даже привести к ошибкам при написании сценариев.

Я рекомендую следовать стандарту и включить параметр ANSI_NULLS, чтобы не было соблазна написать знак равенства.

Параметр *ANSI_PADDING*

Серверный параметр ANSI_PADDING позволяет определить, как сервер будет хранить строковые типы. Дело в том, что если строка меньше, чем размер поля, то по стандарту она должна дополняться пробелами до размера поля. Это не всегда удобно и может вызвать проблемы, поэтому желательно настроить параметр ANSI_PADDING по своему вкусу.

Если серверный параметр ANSI_PADDING включен, то данные строковых типов ведут себя следующим образом:

- varchar — завершающие пробелы удаляются, а если строка состоит только из пробелов, то останется один пробел. (Обратите внимание, что в результате не получится пустая строка.);
- varbinary — удаляются завершающие нули.

Удаление завершающих нулей в данных типов varchar и varbinary не всегда бывает необходимо. Параметр ANSI_PADDING влияет только на описанные типы, поэтому, чтобы решить проблему и сохранять данные так, как их вводит пользователь, можно выбрать другой строковый тип. Благо их у MS SQL Server достаточно.

Следующий пример включает серверный параметр ANSI_PADDING:

```
SET ANSI_PADDING ON
```

Если параметр выключен, то данные этих же типов ведут себя следующим образом:

- `varchar` — данные не обрезаются и не дополняются;
- `varbinary` — данные не обрезаются и не дополняются.

Данные строкового типа `char` в любом случае дополняются пробелами до ширины столбца, и это также надо учитывать при разработке структуры базы данных. Я стараюсь использовать тип `char` только там, где данные всегда должны иметь строго определенный размер, например, для хранения пола достаточно одного символа типа `char`.

Параметр *ANSI_WARNINGS*

Серверный параметр `ANSI_WARNINGS` определяет действия при возникновении ошибок. Если параметр установлен, то ошибки обрабатываются по стандарту ANSI, а именно:

- при использовании функции агрегирования (`SUM`, `MAX`, `MIN`...) при наличии в какой-либо записи значения `NULL` выдается предупреждение;
- деление на ноль или переполнение данных приводит к отмене выполнения команды с выдачей сообщения об ошибке.

По умолчанию этот параметр отключен, но если подключиться к серверу через драйвер OLE DB или ODBC, то переменная для подключения автоматически получит значение `ON`. А если учесть, что к серверу MS SQL подключаются только через эти драйверы, то можно считать, что серверный параметр `ANSI_WARNINGS` по умолчанию включен.

Следующий код отключает параметр:

```
SET ANSI_WARNINGS OFF
```

4.2. Информация о системе

В этой главе мы рассмотрим примеры запросов, с помощью которых можно получить информацию об используемой базе данных, параметрах сервера и т. д. Большинство из параметров информативны, но могут использоваться и при создании программ на языке SQL.

4.2.1. Информация о базе данных

Первым примером будет определение текущей базы данных, к которой подключен пользователь:

```
SELECT DB_NAME() AS 'Текущая база'
```

Функция DB_NAME() возвращает имя базы данных, идентификатор которой передан в качестве параметра. Если в скобках ничего не указано, то возвращается имя текущей базы.

Как узнать идентификаторы и имена баз? В таблице sysdatabases базы данных master находится информация о всех базах данных сервера. Первая колонка — название, а вторая — идентификатор:

```
SELECT *
FROM Master..sysdatabases
```

Давайте посмотрим, как можно использовать функцию DB_NAME() с этой таблицей:

```
SELECT dbid AS 'Идентификатор', DB_NAME(dbid) AS 'Имя'
FROM Master..sysdatabases
```

Особого смысла в этом запросе нет, потому что имена баз и так находятся в таблице, и использование функции DB_NAME() является даже лишним.

А где же тогда может понадобиться функция определения имени текущей базы данных? Допустим, что ваш сценарий использует несколько таблиц одновременно, а сложная логика не позволяет отследить, куда мы сейчас переключились. В определенном месте сценария нам может понадобиться использовать объект базы данных master. Можно написать безусловное переключение на нужную базу, но более элегантным будет такой вариант:

```
IF DB_NAME()<>'master'
 USE master
```

В этом случае переключение произойдет, только если сейчас активной базой является не master. Не забывайте, что оператор USE работает не во всех программах.

В зависимости от текущей базы данных можно выполнять определенные действия.

С помощью функции DB_ID() можно, наоборот, по имени базы данных определить ее идентификатор. Когда это может пригодиться? В *разд. 2.23.1* мы рассматривали пример, в котором проверяли, если база данных существует, то удалить ее перед созданием. Для этого приходилось обращаться к базе данных master и системным таблицам. Но это не всегда возможно.

Более эффективный метод — определить идентификатор базы с помощью функции DB_ID(). Если результат окажется ненулевым, значит, база существует. Следующий пример решает эту задачу с помощью функции DB_ID():

```
-- Определить идентификатор базы данных
-- Если он ненулевой, то база существует
```

```
IF DB_ID('NewDB') IS NOT NULL
 DROP DATABASE NewDB

-- Создаем базу данных
CREATE DATABASE NewDB
```

С помощью SQL можно узнать основные свойства базы данных. Для этого используется функция `DATABASEPROPERTYEX()`. Этой функции нужно передать два параметра: имя базы и имя свойства, которое вы хотите определить. Имена свойств можно увидеть в табл. 4.1.

Таблица 4.1. *Параметры базы данных*

Параметр	Описание	Возвращаемое значение
`Collation`	Сопоставление по умолчанию для базы данных	Имя сопоставления
`IsAnsiNullDefault`	База данных использует стандарт SQL-92 для `NULL`-значений	1 — Да 0 — Нет `NULL` — неправильный ввод
`IsAutoClose`	База данных закрывается и очищает ресурсы, когда отключается последний пользователь	1 — Да 0 — Нет `NULL` — неправильный ввод
`IsInStandBy`	База данных находится в режиме "только для чтения" с возможностью восстановления дополнительных журналов транзакций	1 — Да 0 — Нет `NULL` — неправильный ввод

`Recovery`	Возвращает модель восстановления базы данных	`FULL`
		`BULK_LOGGED`
		`SIMPLE`

`Status`	Состояние базы данных	`ONLINE` — доступна `OFFLINE` — отключена `RESTORING` — восстанавливается `RECOVERING` — идет восстановление, выполнять запросы еще нельзя `SUSPECT` — база данных не может быть восстановлена

Таблица 4.1 *(окончание)*

Параметр	Описание	Возвращаемое значение
Updateability	Определяет, как данные могут обновляться	READ_ONLY — данные могут читаться, но не изменяться READ_WRITE — данные могут читаться и изменяться
UserAccess	Определяет, как пользователи могут получать доступ к базе	SINGLE_USER — только один пользователь db_owner, dbcreator или sysadmin в один момент времени MULTI_USER — все пользователи могут работать с базой
Version	Внутренний номер SQL Server, создаваемый для каждой базы данных	Номер версии, если база открыта, или NULL, если закрыта

Если запрос не выполняется, то с помощью следующей команды можно проверить состояние базы данных:

```
SELECT DATABASEPROPERTYEX('FlenovSQLBook', 'Status')
```

Можно также проверить параметры Updateability и UserAccess.

4.2.2. Имя пользователя

С помощью функции user_name() можно определить имя пользователя, под которым мы работаем в системе. Например, если вы подключились к серверу под именем пользователя с администраторскими правами, то следующий запрос вернет имя dbo:

```
SELECT user_name()
```

Это можно использовать для определения прав, с которыми подключился пользователь. Если это не dbo, то запретить создание объектов. Это очень удобно, но иногда лучше использовать возможности по распределению прав самого MS SQL Server. Это надежнее и элегантнее.

4.2.3. Имя приложения

Для определения имени приложения, с которого пользователь подключился к серверу, используется функция app_name(). Я подключаюсь к MS SQL Server

с помощью Query Analyzer, и следующий запрос возвращает SQL Query Analyzer:

```
SELECT app_name()
```

Однажды мне пришлось делать сложную защиту доступа, и в одном из случаев очень удобно было установить защиту по приложению:

```
if app_name()<>'SQL Query Analyzer'
 print('Выполнение команды запрещено');
```

На всех компьютерах клиентов не было установленного SQL Query Analyzer, а значит, они не могли получить доступ к запрещенной мною функции. Только администраторы могли выполнить сценарий.

4.2.4. Информация об объекте

Все объекты в базе данных имеют свой идентификатор, который назначается сервером. Чтобы определить идентификатор, используется функция `OBJECT_ID()`, которой передается имя интересующего вас объекта. Например, следующая функция вернет идентификатор таблицы `tbPeoples`:

```
SELECT OBJECT_ID('tbPeoples')
```

Сам по себе идентификатор не несет смысловой нагрузки, но он может использоваться в других функциях сервера MS SQL Server. Например, с помощью функции `COLUMNPROPERTY()` можно узнать, поддерживает ли поле таблицы определенное свойство. У этой функции три параметра:

- идентификатор таблицы, объект которого нас интересует;
- имя поля;
- имя свойства.

Допустим, что мы хотим определить, является ли поле `idPeoples` в таблице `tbPeoples` автоматически увеличиваемым. Для этого необходимо выполнить следующий запрос:

```
SELECT COLUMNPROPERTY
 (
   OBJECT_ID('tbPeoples'),
   'idPeoples',
   'IsIdentity'
 )
```

В табл. 4.2 вы можете увидеть основные свойства, которые можно запросить у сервера, и результат, который может вернуть функция `COLUMNPROPERTY()`. Более полный список представлен в файле помощи по Transact-SQL.

Таблица 4.2. *Параметры объектов*

Имя свойства	Описание	Возвращаемое значение
AllowNull	Позволяет ли поле нулевые значения	1 — Да 0 — Нет NULL — неправильный ввод
isComputed	Является ли поле вычисляемым	1 — Да 0 — Нет NULL — неправильный ввод
IsCursorType	Относится ли к типу Cursor	1 — Да 0 — Нет NULL — неправильный ввод
IsDeterministic	Это свойство бывает у вычисляемых колонок и колонок просмотра	1 — Да 0 — Нет NULL — неправильный ввод
IsIdentity	Автоматически увеличиваемое поле	1 — Да 0 — Нет NULL — неправильный ввод
IsIndexable	Является ли поле проиндексированным	1 — Да 0 — Нет NULL — неправильный ввод
IsRowGuidCol	Определить, имеет ли колонка тип uniqueidentifier с установленным свойством ROWGUIDCOL	1 — Да 0 — Нет NULL — неправильный ввод
Precision	Точность типа данных колонки. Например, для колонки Цена таблицы товаров будет возвращено число 19	Значение точности NULL — неправильный ввод

Функцию `OBJECT_ID()` очень часто используют для проверки наличия объекта. Допустим, что ваш сценарий должен создавать таблицу. Если таблица уже существует в базе данных, то сценарий завершит выполнение ошибкой. Поэтому, прежде чем выполнять оператор `CREATE TABLE`, необходимо проверить, существует ли таблица. Такую проверку можно сделать следующим образом:

```
IF OBJECT_ID('tbPeoples') IS NULL
 CREATE TABLE Имя
  (
   Описание столбцов
  )
```

В этом примере оператор `CREATE TABLE` выполняется только в том случае, если таблица не существует. Для определения существования мы вызываем функцию `OBJECT_ID()`, указывая в качестве параметра имя необходимой таблицы. Если такой таблицы не существует, то функция `OBJECT_ID()` ничего не возвращает, т. е. будет нулевое значение `NULL`.

Если объект должен быть создан в любом случае, то можно выполнить сначала удаление таблицы (`DROP TABLE`), а потом создание (`CREATE TABLE`):

```
DROP TABLE Имя
GO

CREATE TABLE Имя
  (
   Описание столбцов
  )
```

Если указанной таблицы не существует во время удаления, то сценарий сгенерирует ошибку, но продолжит работу, и новая таблица будет создана.

Более эффективным решением будет сначала проверить наличие объекта и выполнять удаление только в том случае, если таблица существует.

```
IF OBJECT_ID('tbPeoples') IS NOT NULL
  DROP TABLE tbPeoples
GO

CREATE TABLE tbPeoples
  (
   Описание столбцов
  )
```

Если объект существует, то функция `OBJECT_ID()` вернет его идентификатор, поэтому мы проверяем, если результат выполнения функции не равен нулю, то необходимо удалить таблицу, чтобы создать новую.

Такой код очень часто приходится писать для поддержки постоянных таблиц, которые должны хранить временные данные. Дело в том, что иногда проще воссоздать таблицу, чем удалять данные из большой таблицы. Что является более правильным решением, сказать трудно, но, на мой взгляд, лучше удалить записи, чем воссоздавать таблицу, ведь каждое выполнение оператора `CREATE TABLE` производит обращение к системным таблицам, что, вообще говоря, плохо.

4.2.5. Информация о журнале транзакций

При мониторинге активности базы данных администратор всегда должен знать состояние файлов журналов транзакций. Если один из файлов заполнился на 100%, и нет автоматического увеличения, то данные в базе больше нельзя будет изменять.

Для определения размера журналов и процента заполнения используется команда `DBCC SQLPERF` с параметром `LOGSPACE`:

```
DBCC SQLPERF (LOGSPACE)
```

В ответ на это сервер вернет таблицу примерно следующего вида:

```
Database Name Log Size (MB) Log Space Used (%) Status
------------- ------------- ------------------ -----------
master        1.2421875     38.128929          0
tempdb        0.4921875     50.423176          0
model         0.4921875     75.595238          0
msdb          2.2421875     35.670731          0
pubs          0.7421875     41.184212          0
FlenovSQLBook 0.9921875     72.687004          0
```

Таблица состоит из четырех колонок:

- имя базы данных;
- общий размер журнала;
- процент использованного пространства;
- состояние (всегда содержит значение 0).

4.2.6. Свойство *IDENTITY*

Иногда бывает необходимо получить информацию об автоматически увеличиваемом поле со свойством `IDENTITY`. Для это в MS SQL Server есть несколько полезных функций. В этом разделе мы подробно рассмотрим их.

Функции `IDENT_INCR()` передается имя таблицы, а в результате мы получаем значение приращения (число, на которое автоматически увеличивается значение поля). Следующий пример определяет приращение таблицы `tbPeoples`:

```
SELECT IDENT_INCR ('tbPeoples')
```

Начальное значение `Identity`-поля можно узнать с помощью функции `IDENT_SEED()`. Например:

```
SELECT IDENT_INCR('tbPeoples'),
   IDENT_SEED ('tbPeoples')
```

Чтобы определить строку, которая была добавлена в таблицу последней, после добавления мы запрашивали запись, у которой значение первичного ключа максимально (первичный ключ в таблицах тестовой базы автоматически увеличиваемый). Не очень хороший вариант, тем более, что у MS SQL Server есть способ лучше. Значение последнего сгенерированного `Identity`-поля можно узнать через глобальную переменную `@@identity`. Для определения значения этой переменной нужно выполнить:

```
SELECT @@identity
```

В ответ на это вы, скорее всего, увидите результат `NULL`, потому что мы давно не добавляли в таблицу никаких строк.

Следующий пример добавляет в таблицу `tbPeoples` новую строку и тут же запрашивает ее с помощью оператора `SELECT`. При этом поиск строки производится с помощью глобальной переменной `@@identity`:

```
INSERT INTO tbPeoples
 (vcFamil, vcName, vcSurname, idPosition, dDateBirthDay)
VALUES('КОТИКОВ', 'ИВАН', 'СЕРГЕЕВИЧ', 1, '01.01.1971')

SELECT *
FROM tbPeoples
WHERE idPeoples=@@identity
```

Функция `SCOPE_IDENTITY()` возвращает последнее значение, вставленное в автоматически увеличиваемое поле в некоторых рамках. Рамки определяются процедурой, триггером, функцией или сеансом. Например, выполните сейчас команду:

```
SELECT SCOPE_IDENTITY()
```

В моем примере я получил значение 27. Это последнее значение для Identity-поля, которое я добавил. Теперь создайте новое соединение с сервером и добавьте там строку. Если снова посмотреть содержимое SCOPE_IDENTITY, то мы увидим число 28. Возвращаемся в первое соединение, снова смотрим SCOPE_IDENTITY и видим число 27. Почему в двух разных сеансах мы видим разные числа? Переменная SCOPE_IDENTITY определяет последний идентификатор для текущего сеанса. Иногда это намного удобнее, чем глобальная переменная. Например, если с сервером работает сразу несколько пользователей, то может возникнуть следующая конфликтная ситуация:

1. Мы добавляем строку в таблицу.
2. Сразу после нас другой пользователь добавляет строку данных.
3. При выборке последней строки с идентификатором @@identity мы увидим не свою строку, а ту, которую добавил другой пользователь.

Чтобы избежать этого, лучше использовать функцию SCOPE_IDENTITY():

```
INSERT INTO tbPeoples
 (vcFamil, vcName, vcSurname, idPosition, dDateBirthDay)
VALUES('ВЕРТИКОВ', 'ИВАН', 'СЕРГЕЕВИЧ', 1, '01.01.1971')
```

4.2.7. Информационные процедуры

Самым мощным способом получения информации от сервера является процедура sp_help. Ей необходимо передать в качестве параметра имя интересующего нас объекта, и в результате мы получим несколько таблиц исчерпывающей информации. Например, следующая команда отображает сведения о таблице tbPeoples:

```
EXEC sp_help tbPeoples
```

На рис. 4.2 вы можете увидеть результат работы команды. Как видите, он разбит на несколько таблиц. Вторая таблица отображает подробную информацию о полях таблицы. У меня на экран не поместились все таблицы, но самая нижняя из видимых содержит список ограничений, которые установлены на таблицу, в том числе и индексы.

Если вызвать эту функцию без параметров, то сервер отобразит все таблицы текущей базы данных. Это довольно опасно с точки зрения обеспечения защиты сервера, потому что если хакер узнает имена таблиц, то это может развязать ему руки. Никогда и ни при каких условиях непривилегированный пользователь (гость, особенно из Всемирной сети) не должен иметь возможности выполнить эту команду на сервере.

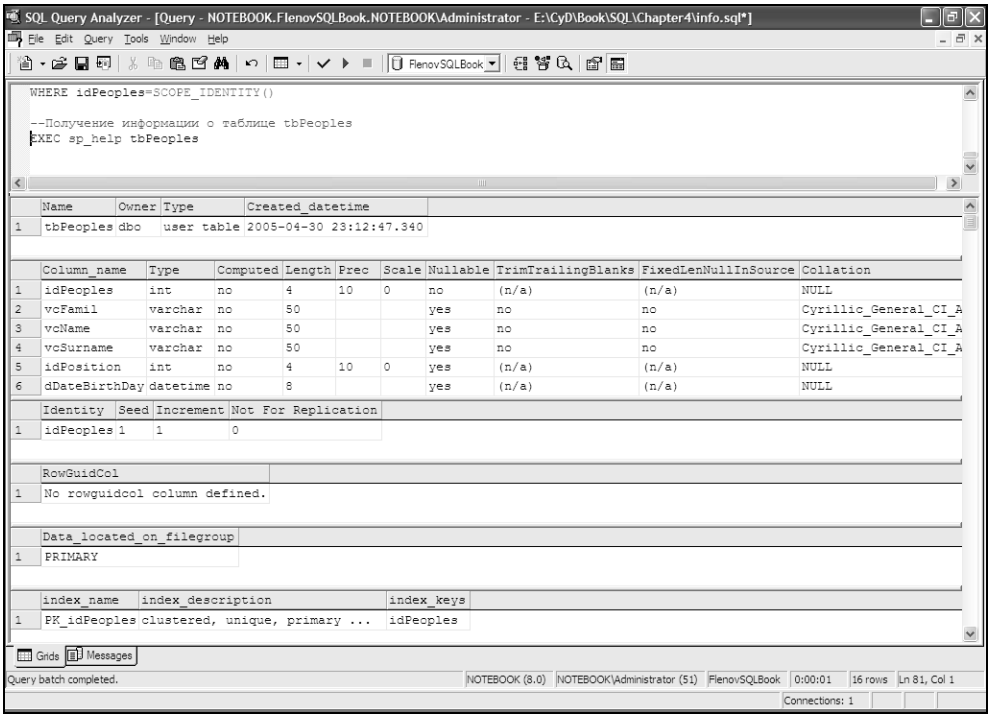


Рис. 4.2. Результат выполнения процедуры `sp_help`

Для получения информации о базе данных используется процедура `sp_helpdb`. Ей необходимо передать в качестве параметра имя интересующей нас базы данных. Например:

```
EXEC sp_helpdb FlenovSQLBook
```

Информация об опциях базы может быть определена с помощью процедуры `sp_dboption`:

```
EXEC sp_dboption FlenovSQLBook
```

С помощью этой же процедуры можно установить параметры базы. Для этого нужно передать три параметра:

- имя базы данных;
- имя параметра;
- значение.

Следующий пример делает базу данных доступной только для чтения:

```
EXEC sp_dboption 'FlenovSQLBook', 'read only', 'TRUE'
```

В табл. 4.3 перечислены параметры, которые вы можете изменять с помощью системной процедуры `sp_dboption`.

Таблица 4.3. *Параметры, изменяемые с помощью* `sp_dboption`

Опция	Описание
`auto create statistics`	Когда этот параметр равен `true`, любая отсутствующая, но необходимая для выполнения запроса статистика будет создаваться автоматически
`auto update statistics`	Устаревшая статистика будет обновляться автоматически
`Autoclose`	Когда этот параметр равен `true`, база данных останавливается, и все ресурсы очищаются, когда последний пользователь отключается от сервера
`Autoshrink`	Если этот параметр равен `true`, то файлы базы данных периодически автоматически сжимаются
`ANSI null default`	Если этот параметр равен `true`, оператор `CREATE TABLE` следует правилам стандарта SQL-92 для определения колонок, позволяющих нулевые значения
`dbo use only`	Если этот параметр равен `true`, то только владельцы базы данных могут использовать ее
`offline`	Когда этот параметр равен `true`, то база данных становится недоступной
`read only`	Значение `true` переводит базу данных в режим "только для чтения"
`single user`	Значение `true` переводит базу данных в однопользовательский режим

Информация о самом сервере определяется с помощью процедуры `sp_helpserver`. Эта процедура вернет нам имя сервера и основные параметры:

```
EXEC sp_helpserver
```

В результате мы получим две таблицы, в которых содержится информация о самой базе данных и используемых ею файлах (тип файла, расположение, размер и т. д.).

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\info.sql на компакт-диске, прилагаемом к книге.

4.2.8. Пользовательские параметры конфигурации

В *разд. 4.1* мы рассмотрели достаточно много конфигурационных параметров, которые можно установить с помощью оператора `SET`, но нам же необходимо знать, как определить текущие настройки, чтобы зря не влиять на них.

Информацию обо всех параметрах сервера и подключения можно узнать с помощью вызова команды `DBCC USEROPTIONS`. Вызовите эту команду, и перед вами появится таблица из установленных параметров. Если какой-либо параметр оператора `SET` не установлен, то его значения в результирующей таблице не будет.

Пример результата выполнения команды `DBCC USEROPTIONS`:

```
Set Option                    Value
--------------------------------------------------
textsize                      64512
language                      us_english
dateformat                    mdy
datefirst                     7
quoted_identifier             SET
arithabort                    SET
ansi_null_dflt_on             SET
ansi_defaults                 SET
ansi_warnings                 SET
ansi_padding                  SET
ansi_nulls                    SET
concat_null_yields_null       SET


(12 row(s) affected)


DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
```

При работе с командой `DBCC USEROPTIONS` вы должны быть предельно внимательными, потому что имена параметров в результирующей таблице и имена параметров, которые мы устанавливаем с помощью оператора `SET`, могут не совпадать.

Так как имена не совпадают, иногда бывает удобнее использовать переменную `@@OPTIONS`, которая возвращает битовую маску из установленных параметров. Чтобы увидеть эту маску, можно выполнить следующий запрос `SELECT`:

```
SELECT @@OPTIONS
```

Результирующее число — это маска (сумма констант параметров), с помощью которой достаточно просто определить любой параметр сервера. В табл. 4.4 перечислены константы, связанные с определенными параметрами сервера или подключения.

Таблица 4.4. *Константы серверных параметров*

Значение	Параметр
1	DISABLE DEFAULT CONSTRAIN CHECK
2	IMPLICIT TRANSACTION
4	CURCOR CLOSE ON COMMIT
8	ANSI WARNINGS
16	ANSI PADDING
32	ANSI NULLS
64	ARITHABORT
128	ARITHIGNORE
256	QUOTED IDENTIFIER
512	NOCOUNT
1024	ANSI NULL DEFAULT ON
2048	ANSI NULL DEFAULT OFF

Если вы посмотрите на параметры из табл. 4.4, то заметите, что некоторые из них мы не обсуждали. Я опустил те параметры, которые не стоит изменять, или устаревшие параметры сервера. С другой стороны, в этой таблице перечислены не все возможные параметры.

Следующий пример показывает, как можно проверить наличие установленного параметра ANSI_NULLS, и, если этот параметр не установлен, — установить его:

```
IF @@OPTIONS & 32 =0
 BEGIN
  SET ANSI_NULLS OFF
  PRINT('Параметр установлен')
 END
ELSE
 BEGIN
  PRINT('Параметр уже был установлен')
 END
```

Серверный параметр `ANSI_NULLS` имеет константу 32, и в первой строке мы побитно сравниваем переменную `@@OPTIONS` с этим значением. Если результат равен нулю, то параметр не установлен. Если результат больше, то `ANSI_NULLS` равен `ON`, и повторная установка не требуется.

Некоторые параметры относятся только к подключению и сбрасываются при отключении и повторном подключении к серверу. Чтобы не восстанавливать параметры по одиночке, вы можете воспользоваться процедурой `sp_configure` и установить все значения одной маской. Таким образом, достаточно один раз настроить сервер, определить маску и потом устанавливать ее с помощью `sp_configure`.

Следующий пример показывает, как установить одной командой маску 5496:

```
sp_configure 'user options', 5496
```

После выполнения этой команды, чтобы настройки вступили в действие, необходимо вызвать команду `RECONFIGURE`. Если выполнить команду с параметром `WITH OVERRIDE`, то во время принятия изменений не будет лишних проверок на допустимость параметров или на возможные рекомендации (рекомендуется или нет устанавливать параметр).

Следующий пример показывает, как изменить маску и принять изменения:

```
sp_configure 'user options', 5496
RECONFIGURE WITH OVERRIDE
```

С помощью команды `sp_configure` легко изменить маску значений, но ведь в маске могут быть константы только из табл. 4.4. Мы уже отмечали недостаток того, что не все параметры есть в этой таблице, но хотя бы что-то — тоже преимущество при конфигурировании подключения.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\setinfo.sql на компакт-диске, прилагаемом к книге.

4.3. Обработка ошибок

При создании программ на языке SQL очень важно правильно обрабатывать ошибки, чтобы пользователь видел более осмысленные сообщения. Сервер SQL не всегда может дать понятное описание проблемы, которая возникла во время выполнения сценария и чаще всего выдает непонятные конечному пользователю сообщения. Только программист, создававший программу, может сделать сообщения более осмысленными.

Зачем нужно писать свои сообщения об ошибках? Допустим, что в одном сценарии вы выполняете 100 обновлений таблицы. Если при одном из обновлений данные оказались заблокированными, то сервер не сможет изменить нужные записи таблицы и сообщит о блокировке. Пользователь, увидев сообщение об ошибке, позвонит программисту или администратору, т. е. вам, и зачитает текст сообщения, но вам этот текст ничего не скажет, потому что блокировка могла возникнуть в любом из 100 запросов.

Если вы потратите немного времени и реализуете обработку ошибок после каждого обновления, так что в случае возникновения проблемы будет выведено более понятное сообщение о запросе, который не смог выполниться, то вы сбережете свои нервы во время сопровождения программы. В случае ошибки пользователь скажет вам по телефону то, что написали в сценарии вы, по этому сообщению вы быстро найдете проблемный запрос и с помощью средств SQL-сервера сможете определить, заблокирована ли обновляемая строка, или это иная проблема.

4.3.1. Глобальная переменная @@*ERROR*

В обработке ошибок нам помогут глобальная переменная `@@ERROR` и функция `RAISERROR()`. Переменная содержит номер ошибки, которая возникла во время выполнения последнего SQL-оператора. Если эта переменная равна нулю, значит, ошибок не было, и последний оператор выполнился удачно.

Рассмотрим пример:

```
BEGIN TRANSACTION
UPDATE tbPeoples
SET dDateBirthDay='31.31.2004'
WHERE vcFamil='mr.ИВАНОВ'
  AND vcName='ИВАН'

IF @@ERROR<>0
 BEGIN
  Print 'Ошибка обновления даты рождения Иванова'
  ROLLBACK TRANSACTION
 END
ELSE
 COMMIT TRANSACTION
```

В этом примере мы начинаем транзакцию, в которой обновляется дата рождения работника Иванова. После этого проверяем переменную `@@ERROR`. Если она не равна нулю, значит, во время обновления произошла ошибка. В этом

случае на экран выводится сообщение об ошибке с помощью функции `print()` и транзакция откатывается.

Server: Msg 242, Level 16, State 3, Line 1

The conversion of a char data type to a datetime data type resulted in an out-of-range datetime value.

The statement has been terminated.

Ошибка обновления даты рождения Иванова

То есть:

Сервер: Сообщение 242, Уровень 16, Состояние 3, Строка 1

Преобразование символьного типа данных в тип данных datetime завершилось ошибкой — значение datetime выходит за пределы

Оператор был прерван

Ошибка обновления даты рождения Иванова

Даже если бы у нас было несколько запросов на обновление данных, то мы легко бы определили, что ошибка возникла именно при обновлении даты рождения Иванова, а не какой-то иной информации.

Если ошибок не было, то транзакция подтверждается оператором `COMMIT TRANSACTION`. Сервер вернет сообщение об ошибке.

В глобальной переменной `@@ERROR` хранится идентификатор ошибки. Все идентификаторы и описания хранятся в таблице `sysmessages` базы данных master. С помощью следующего запроса можно получить описание именно нашей ошибки:

```
SELECT *
FROM master..sysmessages
WHERE Error=242
```

Посмотрите на результирующую строку. Это то, что сервер нам отобразил после выполнения запроса. Таким образом, во время выполнения сценария мы можем узнать с помощью таблицы `sysmessages` описание ошибки и вывести его в своем окне или распечатать.

4.3.2. Генерирование сообщений

Как мы уже заметили, в обработке ошибок нам помогает функция `RAISERROR()`, которая генерирует ошибку. В общем виде эта функция выглядит следующим образом:

```
RAISERROR ( { msg_id | msg_str } { , severity , state }
    [ , argument [ ,...n ] ] )
    [ WITH option [ ,...n ] ]
```

Минимум, что необходимо указать в скобках, — это три параметра:

- первый параметр — это идентификатор пользовательского сообщения или текстовое сообщение;
- уровень критичности. Вы можете указывать здесь любое число в следующих диапазонах:
  - 0—18 — может использоваться любыми пользователями и указывает самый низший уровень;
  - 19—25 — такие сообщения могут создаваться только пользователями, которые обладают правами `sysadmin` в сервере MS SQL Server;
- состояние — число от 1 до 127, если указать отрицательное значение, то по умолчанию будет использоваться 1.

Давайте попробуем создать собственное сообщение об ошибке. Следующий пример отображает сообщение о том, что нельзя добавлять стоку:

```
RAISERROR('Нельзя добавлять такую строку', 16, 1)
```

Уровень критичности я установил равным 16, а состояние — 1. Вообще-то, эти числа вы выбираете на свой вкус, главное, чтобы они попали в нужные диапазоны. Большинство программистов не утруждают себя установкой разных значений критичности и состояний и всегда используют одни и те же значения.

Выполните запрос `RAISERROR`, генерирующий ошибку, и сервер вернет следующие две строки:

Server: Msg 50000, Level 16, State 1, Line 1

Нельзя добавлять такую строку

В первой строке отображается номер сообщения (Msg), уровень (Level), состояние (State) и строка, в которой найдена ошибка. Уровень и состояние соответствуют тем значениям, которые мы указали в функции `RAISERROR()`. То, что строка с ошибкой равна 1, тоже понятно, ведь наш запрос состоит всего из одной строки. А вот номер сообщения вызывает вопрос — почему именно 50000? Мы же ничего не указывали.

Все сообщения, создаваемые пользователем, должны иметь номера, бóльше или равные 50000. Если вы указали в качестве первого параметра функции `RAISERROR()` текст ошибки, а не номер, то сервер автоматически будет использовать 50000.

Давайте теперь посмотрим, как можно использовать номер сообщения в первом параметре. Но мы не можем генерировать системные сообщения. Например, мы уже видели ошибку с номером 242, и для ее генерации нужно было бы выполнить код:

```
RAISERROR(242, 16, 1)
```

Но в ответ на это сервер выдаст сообщение, что первый параметр должен быть более 13000. До этого значения находятся системные сообщения, которые генерируются только самим сервером. Начиная с 13000, идут сообщения, которые можно использовать, но те сообщения об ошибках, которые уже существуют на сервере (в таблице `sysmessages`), не несут в себе полезной информации, поэтому лучше создать собственное сообщение об ошибке.

4.3.3. Создание собственных сообщений

В *разд. 3.7.1* мы уже рассматривали теорию создания собственных сообщений с помощью процедуры `sp_addmessage`. В данном разделе мы обсудим, как можно использовать эту процедуру совместно с функцией `RAISERROR()`.

Давайте посмотрим, как можно добавить сообщение:

```
USE master
EXEC sp_addmessage 50001, 16,
   'Ошибка добавления записи'
```

В качестве идентификатора я выбрал число 50001, уровень равен 16, а текст сообщения вы видите. Давайте теперь попробуем сгенерировать это сообщение:

```
RAISERROR(50001, 16, 1)
```

В результате сервер вернет следующий текст:

Server: Msg 50001, Level 16, State 1, Line 1

Ошибка добавления записи

Теперь попробуем изменить сообщение и явно указать, что оно на русском языке. Для этого нужно будет добавить параметр `@lang` и указать последний параметр `REPLACE`, потому что мы заменяем уже существующее сообщение:

```
USE master
EXEC sp_addmessage 50001, 16,
   'Ошибка добавления записи',
   @lang = 'Russian', @replace=REPLACE
```

В данном примере мы явно указали русский язык для сообщения. Можно добавить еще одну запись с таким же идентификатором, но на другом языке. Например, так будет добавлено сообщение на английском:

```
EXEC sp_addmessage 50001, 16,
  'Add data error',
  @lang = 'us_english',
  @replace=REPLACE
```

4.3.4. Резюме

Обработку ошибок и понятные пользовательские сообщения лучше всего (и нужно!) использовать в хранимых на сервере процедурах, функциях и триггерах. Чем более понятными будут сообщения сервера, тем легче будет пользователю сообщить программисту и администратору о возможных проблемах. По личному опыту могу сказать, что чем лучше пользователь сможет понять проблему, тем быстрее он объяснит ее мне, и тем быстрее я смогу ее устранить. А проблемы возникают регулярно, от них никуда не деться. Уж такова специфика нашей работы.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\errors.sql на компакт-диске, прилагаемом к книге.

4.4. Поддержка XML

XML — это язык, который могут использовать Web-разработчики для представления информации из базы данных MS SQL Server в виде Web-страницы. Чтобы результат выполнения запроса имел формат XML, необходимо в конце оператора `SELECT` указать оператор `FOR XML`. В результате сервер возвращает атрибуты данных, таких как имена полей таблиц, в виде тегов. Клиентский браузер может потом использовать эти теги для форматирования возвращаемой информации.

Самый простой способ — указать опцию `FOR XML AUTO` для возврата результата запроса в стандартном формате. Следующий пример выбирает три колонки из таблицы `tbPeoples`. Обратите внимание, что результат комбинирует все колонки в одну текстовую строку:

```
SELECT TOP 5 vcFamil, vcName
FROM tbPeoples
FOR XML AUTO
```

Результат:

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
------------------------------------------------------------<tbPeoples
vcFamil="mr.ИВАНОВ" vcName="ИВАН"/>
<tbPeoples vcFamil="mr.ПЕТРОВ" vcName="ИВАН"/>
```

```
<tbPeoples vcFamil="mr.СИДОРОВ" vcName="ИВАН"/>
<tbPeoples vcFamil="mr.КОНОНОВ" vcName="ШВАРЦ"/>
<tbPeoples vcFamil="mr.СЕРГЕЕВ" vcName="СЕРГЕЙ"/>
(5 row(s) affected)
```

В некоторых случаях Web-разработчики хотят избежать автоматического форматирования. Вы можете указать опцию `RAW` для превращения каждой строки в результирующем наборе в XML-элемент с общим идентификатором строки как тег элемента.

Сравните результат следующего запроса с предыдущим. Этот пример возвращает такие же данные, но форматирование более общее. Обратите внимание, что таблицы не именованы и колонки не сгруппированы по имени таблицы:

```
SELECT TOP 5 vcFamil, vcName
FROM tbPeoples
FOR XML RAW
```

Результат:

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
----------------------------------------------------------------
<row vcFamil="mr.ИВАНОВ" vcName="ИВАН"/>
<row vcFamil="mr.ПЕТРОВ" vcName="ИВАН"/>
<row vcFamil="mr.СИДОРОВ" vcName="ИВАН"/>
<row vcFamil="mr.КОНОНОВ" vcName="ШВАРЦ"/>
<row vcFamil="mr.СЕРГЕЕВ" vcName="СЕРГЕЙ"/>
(5 row(s) affected)
```

Оператор `SELECT`, который содержит выражение `FOR XML`, переформатирует выходные данные. Вы не можете использовать данные в формате XML в следующих конструкциях:

- `SELECT INTO`;
- `COMPUTE BY`;
- встроенных процедурах, вызванных из оператора `INSERT`;
- представлениях (view) или функциях, определенных пользователем, которые возвращают наборы строк.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\xml.sql на компакт-диске, прилагаемом к книге.

4.5. Типы данных, определенные пользователем

Типы данных, определенные пользователем, основываются на типах данных, предоставляемых системой. Они позволяют вам добиться совместимости, когда вы работаете с часто используемыми элементами данных в разных таблицах или базах данных.

Наиболее часто приходится создавать типы именно ради совместимости. Например, в сервере SQL тип данных для целых чисел может называться `int`, а в других базах это может быть `integer`. Вы можете в MS SQL Server создать тип `integer`, который будет как бы псевдонимом для целочисленного типа.

Тип данных, определенный пользователем, создается для конкретной базы данных и принадлежит ей. Тип данных из базы данных NewDB не может быть использован в других базах данных. Типы данных, созданные для базы данных model, автоматически помещаются во все новые базы данных. Все типы данных помещаются в качестве строк в системную таблицу `systypes`.

Для создания типа данных, определенного пользователем, служит хранимая процедура `sp_addtype`, которая в общем виде выглядит следующим образом:

```
sp_addtype [ @typename = ] type,
    [ @phystype = ] system_data_type
    [ , [ @nulltype = ] 'null_type' ]
    [ , [ @owner = ] 'owner_name' ]
```

Рассмотрим параметры этой процедуры:

- `@typename` — имя типа, создаваемого пользователем;
- `@phystype` — имя системного типа;
- `@nulltype` — способ, которым новый тип данных будет хранить нулевое значение. Здесь можно указать одно из следующих значений: `NULL`, `NOT NULL` или `NONULL`;
- `@owner` — владелец.

Следующий пример создает три новых типа:

```
EXEC sp_addtype mtCity, 'nvarchar(15)', NULL
EXEC sp_addtype mtFamil, 'nvarchar(25)', NULL
EXEC sp_addtype mtAge, 'int', NULL
```

Первый тип имеет тип `mtCity` и соответствует системному типу `nvarchar` длиной 15 символов. Теперь, если в какой-либо из таблиц нужно будет создать поле для хранения названий городов, можно указывать в качестве типа вместо `nvarchar` тип данных `mtCity`.

Для имен типов данных, объявленных пользователем, я настоятельно рекомендую вам придумать какой-нибудь префикс, чтобы они явно выделялись. Например, я использую префикс `mt` (сокращение от слов my type, мой тип). Он действительно очень полезен. Допустим, что вы объявили тип данных `City` и хотите создать поле для хранения названий городов. Вы уже не можете просто назвать это поле как `City`, придется имя поля заключать в квадратные скобки, что не всегда удобно.

Посмотрим, как можно создать таблицу, в которой поля будут использовать объявленные ранее типы данных:

```
CREATE TABLE TestTable
(
 id int DEFAULT 1,
 vcFamil mtFamil,
 iAge mtAge,
 vcCity mtCity
)
```

Для удаления пользовательской переменной используется процедура `sp_droptype`:

```
sp_droptype [ @typename = ] 'type'
```

Следующий пример удаляет пользовательский тип:

```
EXEC sp_droptype mtCity
```

Удалять можно только неиспользуемые типы данных. Если удаляемый тип данных используется хотя бы в таблице, то его удалить нельзя.

Выполните процедуру `sp_help` для получения списка всех текущих объявленных типов данных.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\types.sql на компакт-диске, прилагаемом к книге.

4.6. Поддержка индексов

В *разд. 1.2.7* мы достаточно много внимания уделили индексам. Мы подробно разобрали, что это такое и как нужно создавать различные варианты индексов. Но этого недостаточно. Индексы нужно контролировать и поддерживать в оптимальном состоянии. MS SQL Server предоставляет все необходимые средства для удобного анализа состояния индексов.

У MS SQL Server есть в утилите Enterprise Manager мастер Index Tuning Wizard, который автоматически отслеживает использование индекса.

Даже если вы опытный программист, в построении индексов вам поможет мастер Index Tuning Wizard, который просматривает запрос, чтобы определить, какие индексы полезны, принимая во внимание план выполнения, и отображает используемые индексы. Мастер Index Tuning Wizard может:

- рекомендовать или проверить оптимальную конфигурацию индекса для базы данных, используя указанный файл или анализируя файл с помощью оптимизатора запросов;
- указать критерии и принять к сведению расчеты Index Tuning Wizard, такие как максимальное количество запросов для точной настройки, максимальное пространство для рекомендуемых индексов и максимальное количество колонок;
- рекомендовать пути для настройки базы данных, чтобы уменьшить вероятность возникновения проблем.

Перед выполнением мастера рекомендуется удалить все комментарии, потому что SQL Server может неправильно их обработать, и мастер выдаст неоптимальные рекомендации.

Не рекомендуется использовать мастер на системных таблицах, ограничении `PRIMARY KEY` или уникальном индексе. Не рекомендуется удалять любые индексы при включенной опции `Keep existing indexes`. Мастер предлагает только новые индексы, если это необходимо.

Чтобы получить информацию об индексах, можно использовать системную процедуру `sp_helpindex`, указав имя нужной таблицы. Например:

```
EXEC sp_helpindex tbPeoples
```

Через определенное время данные становятся фрагментированными за счет разбиения страниц, когда они переполняются. Фрагментация замедляет работу индекса, и может случиться даже так, что выгода от использования индексов исчезнет.

Фрагментация возникает при частых изменениях данных. Например, когда строка данных добавлена или удалена из таблицы, или когда значение в индексной колонке изменилось, сервер приводит в порядок индексные страницы для согласования изменений и хранилища индексных данных. Приведение в порядок индексных страниц называется разделением страниц. Разделение увеличивает размер таблицы и время, необходимое для обработки запроса.

Существуют два метода управления фрагментацией в SQL Server. Первый метод — удаление и воссоздание кластерного индекса и указания опции

FILLFACTOR. Второй метод — перестроение индексов и указания опции FILLFACTOR.

Для начала посмотрим, как можно узнать состояние индекса. Для этого используется оператор DBCC SHOWCONTIG, который отображает информацию о фрагментации данных и индексов указанной таблицы. В общем виде команда выглядит следующим образом:

```
DBCC SHOWCONTIG
    [    ( { table_name | table_id | view_name | view_id }
            [ , index_name | index_id ]
        )
    ]
    [ WITH { ALL_INDEXES
                | FAST [ , ALL_INDEXES ]
                | TABLERESULTS [ , { ALL_INDEXES } ]
                [ , { FAST | ALL_LEVELS } ]
            }
    ]
```

Когда вы выполняете оператор DBCC SHOWCONTIG, MS SQL Server проходит через индексные страницы и листовой уровень для определения, сильно ли фрагментирована таблица или индекс. Он также определяет, насколько заполнены страницы данных и индексов.

Просмотрите состояние индексов с помощью DBCC SHOWCONTIG в следующих случаях:

- таблицы сильно изменились;
- произошло массовое изменение данных, которые входят в индекс;
- таблицы содержат импортированные данные;
- таблица начала медленно обрабатывать запросы.

Таблица 4.5 описывает статистику, которую возвращает оператор DBCC SHOWCONTIG.

Таблица 4.5. *Статистика, отображаемая командой DBCC SHOWCONTIG*

Параметр	Описание
Pages scanned	Количество страниц в таблице или индексе
Extents scanned	Количество пространств в таблице или индексе
Average pages per extend	Среднее количество страниц в пространстве

Таблица 4.5 *(окончание)*

Параметр	Описание
`Extend switches`	Количество раз, которые оператор покидал пространство во время пересечения страниц пространства
`Scan density [Best count: Actual Count]`	Плотность сканирования 100 (процентов). Если меньше 100, значит, существует некоторая фрагментация
`Best Count`	Идеальное число изменений пространств
`Actual Count`	Текущее значение изменений пространств
`Logical scan fragmentation`	Процент страниц вне порядка возвращенных сканированием листьев страниц и индексов
`Extent scan fragmentation`	Процент пространств вне порядка возвращенных сканированием листьев страниц и индексов
`Average bytes free per page`	Среднее количество свободных байт на сканированных страницах. Большое значение указывает на невысокую плотность заполнения. Малое значение лучше. Длинные строки могут привести к увеличению этого параметра
`Average page density (full)`	Значение, указывающее на плотность страниц. Принимается во внимание размер строки, так что это значение более актуально при оценке плотности страниц. Большие значения предпочтительнее

В следующем примере выполняется оператор доступа к таблице `tbPeoples`:

```
DBCC SHOWCONTIG ('tbPeoples', PK_idPeoples)
```

В скобках нужно указать в качестве первого параметра имя таблицы в одинарных кавычках, а в качестве второго — имя или идентификатор индекса. Имена индексов можно узнать, выполнив команду `sp_helpindex`.

Вместо имени индекса можно указать его идентификатор. Например, в таблице `tbPeoples` только один индекс, и для его просмотра можно выполнить команду:

```
DBCC SHOWCONTIG ('tbPeoples', 1)
```

Результат выполнения команды:

```
DBCC SHOWCONTIG scanning 'tbPeoples' table...
Table: 'tbPeoples' (2025058250); index ID: 1, database ID: 8
TABLE level scan performed.
- Pages Scanned.............................: 1
- Extents Scanned...........................: 1
- Extent Switches...........................: 0
```

```
- Avg. Pages per Extent.......................: 1.0
- Scan Density [Best Count:Actual Count].......: 100.00% [1:1]
- Logical Scan Fragmentation ..................: 0.00%
- Extent Scan Fragmentation ...................: 0.00%
- Avg. Bytes Free per Page.....................: 6668.0
- Avg. Page Density (full).....................: 17.62%

DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
```

Если указать `WITH ALL_INDEXES`, то будет собрана информация о всех индексах таблицы:

```
DBCC SHOWCONTIG ('tbPeoples') WITH ALL_INDEXES
```

Для дефрагментации фрагментированных данных листового уровня индекса используется оператор `DBCC INDEXDEFRAG`. Дефрагментация упорядочивает страницы так, чтобы их физическое расположение максимально совпадало с логическим. Это может значительно повысить производительность. В общем виде команда выглядит следующим образом.

```
DBCC INDEXDEFRAG
    ( { database_name | database_id | 0 }
        , { table_name | table_id | 'view_name' | view_id }
        , { index_name | index_id }
    )    [ WITH NO_INFOMSGS ]
```

Общий вид немного непонятен, но в упрощенном виде все выглядит следующим образом:

```
DBCC INDEXDEFRAG(Имя базы, Имя таблицы, Имя индекса)
```

Когда вы используете `DBCC INDEXDEFRAG`, он упаковывает страницы индекса, делая его соответствующим параметру `FILLFACTOR`. Любые пустые страницы удаляются.

Я рекомендую запускать дефрагментацию одного файла в определенный момент времени, когда индекс разделяет несколько файлов. Страницы не разделяются между файлами.

Дефрагментация — это операция в режиме "онлайн". Она не делает блокировок на долгое время и не блокирует запущенные запросы и обновления. Информация об этом процессе всегда полностью записывается в журнал, независимо от модели восстановления.

Время, требуемое для дефрагментации, зависит от степени фрагментации. Сильно фрагментированные данные могут потребовать больше времени, чем перестроение индекса. Мало фрагментированные данные требуют меньше времени на дефрагментацию, чем перестроение индекса.

Оператор DBCC INDEXDEFRAG не повышает производительность, если индексы фрагментированы на диске физически. Для физической дефрагментации таблицы нужно перестроить индексы.

Следующий пример запускает оператор DBCC INDEXDEFRAG на индекс PK_idPeoples таблицы tbPeoples базы данных FlenovSQLBook:

```
DBCC INDEXDEFRAG(FlenovSQLBook, tbPeoples, PK_idPeoples)
```

Оператор DROP_EXISTING совместно с CREATE INDEX используется для изменения характеристики индекса или перестроения индекса без его удаления и воссоздания. Выгода от использования DROP_EXISTING заключается в том, что вы можете изменять индексы, созданные с ограничением PRIMARY KEY или UNIQUE.

Выполняйте оператор CREATE INDEX с опцией DROP_EXISTING для перестроения именованного кластерного или некластерного индекса. В этот момент происходит:

- реорганизация листового уровня страниц с помощью сжатия или расширения строк;
- удаление фрагментации;
- перерасчет статистики индекса.

Когда вы используете опцию DROP_EXISTING, вы можете изменить следующие характеристики индекса:

- тип:
  - вы можете изменить некластерный индекс на кластерный;
  - вы не можете изменить кластерный индекс на некластерный;
- колонки индекса:
  - вы можете изменить объявление индекса с помощью указания других колонок;
  - вы можете указать дополнительные колонки или удалить определенные колонки из составного индекса;
  - вы можете изменить индексную колонку, чтобы она была уникальной или нет;
- настройки:
  - вы можете изменить процентные значения FILLFACTOR или PAD INDEX.

При использовании DROP_EXISTING для кластерного индекса сервер требует пространство для физического перестроения индекса.

В системной таблице нельзя перестраивать индексы. Да и не стоит их вообще трогать. В системных таблицах не так много данных, и они изменяются не так часто, чтобы возникала сильная фрагментация. Изменения происходят, в основном, на этапе проектирования базы данных.

Опция `DROP_EXISTING` на кластерном индексе помогает вам избежать ненужной работы по удалению и воссозданию некластерного индекса, если кластерный индекс перестраивается на той же колонке.

Следующий пример перестраивает существующие индексы. Индекс перестраивается как кластерный со значением заполнения страниц 65%. Этот оператор выдаст сообщение об ошибке, если кластерный индекс для данной таблицы уже существует:

```
CREATE UNIQUE NONCLUSTERED INDEX PK_idPeoples
ON [tbPeoples] (idPeoples)
WITH DROP_EXISTING, FILLFACTOR=65
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\index.sql на компакт-диске, прилагаемом к книге.

Может показаться, что наилучшим вариантом для индекса будет полное удаление ключевой колонки и воссоздание ее заново. Такое мнение возникает, когда программисты используют поля `IDENTITY`. Например, вы заполнили таблицу строками, очистили все записи и начали наполнять таблицу новыми данными. У новых строк в ключевом поле счетчик будет продолжать увеличивать значение с последнего добавленного номера, а не с единицы. Чтобы нумерация пошла снова с 1, поле пересоздают.

Это хороший выход, но только в том случае, если ваша таблица действительно пуста, и при этом нет связанных таблиц, содержащих поврежденные ссылки, иначе может возникнуть коллизия. Рассмотрим пример, использующий связанные таблицы номеров телефонов и их типов. Допустим, что вы очистили таблицу типов, но еще остались номера телефонов, которые ссылаются на несуществующий тип. Это возможно, если вы не создавали внешнего ключа, и нарушилась ссылочная целостность. При заполнении таблицы типов новыми значениями с идентификаторами, начинающимися с 1, может обнаружиться неверная ссылка. Например, до удаления таблица типов выглядела следующим образом:

```
ID      Тип
-------------------
1       Сотовый
2       Домашний
```

В таблице номеров телефонов осталась запись со ссылкой на тип телефона с `ID`, равным 2. После воссоздания таблицы типов вы добавили запись в другом порядке

```
ID      Тип
--------------------
1       Домашний
2       Сотовый
```

Теперь номер телефона, который был домашним, стал сотовым. Это серьезная ошибка, которую невозможно будет потом выявить. Пусть номера идентификаторов всегда увеличиваются, и таблица после очистки и наполнения ее новыми значениями выглядит так:

```
ID      Тип
--------------------
3       Домашний
4       Сотовый
```

Вот теперь записи с `ID`, равным 2, не существует. Такой номер телефона легко найти в базе данных и исправить.

4.7. Работа со статистикой

Мы уже говорили о том, что статистика позволяет серверу сделать наиболее правильный выбор, когда обсуждали индексы *(см. разд. 1.2.7)*. Давайте рассмотрим пример, который наглядно продемонстрирует достоинства статистики. Допустим, что у нас есть список работников литейного цеха. В такой таблице, наверно, 90% (если не более) будут мужчины, ведь работа в литейном производстве достаточно тяжела для женщин.

Теперь допустим, что нам нужно найти всех женщин. Так как их мало, наиболее эффективным вариантом будет использование индекса. Но если нужно найти мужчин, то эффективность индекса падает. Количество выбираемых записей слишком велико, и для каждой из них обходить дерево индекса довольно накладно. Намного проще просканировать всю таблицу, что выполнится намного быстрее, потому что серверу достаточно по одному разу прочитать все листья нижнего уровня индекса без необходимости многоразового чтения всех уровней.

Сервер SQL собирает статистику с помощью чтения всех значений полей или шаблона для создания равномерного распределенного и отсортированного списка значений. Сервер SQL динамически определяет процент строк, которые должны быть опробованы, основываясь на количестве строк в таблице.

Оптимизатор запросов выполняет одно из двух — полное сканирование или шаблоны строк при сборе статистики.

Для создания статистики для поля или набора полей используется оператор `CREATE STATISTICS`, который в общем виде выглядит следующим образом:

```
CREATE STATISTICS statistics_name
ON { table | view } ( column [ ,...n ] )
    [ WITH
        [ [ FULLSCAN
            | SAMPLE number { PERCENT | ROWS } ] [ , ] ]
        [ NORECOMPUTE ]
    ]
```

После оператора `CREATE STATISTICS` мы должны указать имя, которое задается пользователем. Желательно, чтобы в имени отражался смысл содержимого колонок, статистика которых обрабатывается.

После этого идет ключевое слово `ON` и имя таблицы или представления. Далее в скобках перечисляются поля, которые сервер должен просмотреть для сбора статистики.

Если нужны дополнительные опции, то они пишутся после `WITH`. Давайте посмотрим, что здесь можно указывать:

- `SAMPLE` — простой вариант сбора статистики. После этого ключевого слова задается число, определяющее, сколько строк нужно просканировать. Оно может быть выражено в процентах или в фактическом количестве строк. В первом случае после числа нужно написать `PERCENT`, во втором — `ROWS`;
- `FULLSCAN` — полное сканирование, которое желательно использовать, когда создается индекс или обновляется статистика. Таблица размером менее 8 Мбайт для сбора статистики всегда сканируется полностью.

Давайте создадим статистику для поля имен (`vcName`) в таблице `tbPeoples`:

```
CREATE STATISTICS stat_Name
   ON tbPeoples (vcName)
   WITH FULLSCAN
```

В нашей тестовой таблице не так уж много строк, поэтому я выбрал полное сканирование. В качестве имени статистики указано `stat_Name`.

Каждое значение, хранящееся в поле, называется распределенным шагом. Распределенный шаг показывает, как много строк существует между началом и концом шага. Первое и последнее значения всегда включаются в статистику. Здесь может быть до 300 значений.

Вместе со статистикой сохраняется:

- количество строк в таблице;
- дата и время, когда статистика последний раз обновлялась;
- количество строк шаблона для создания гистограммы и определения плотности;
- количество распределенных шагов;
- среднее значение поля;
- плотность индивидуального поля и всех комбинированных полей;
- количество строк, попавших в шаг гистограммы;
- количество строк, которые равны значению верхней границы шага;
- количество определенных значений в шаге гистограммы.

Вы можете создавать статистику автоматически или вручную. Желательно позволять серверу SQL создавать статистику автоматически. Когда опция базы данных `auto create statistics` включена, сервер автоматически предоставляет статистику для индексных колонок, которые содержат данные, или для неиндексированных колонок, которые используются для объединения.

Оптимизатор запросов активирует создание статистики, когда оптимизирует запросы. Это может оказаться неудобным, если оптимизатор запросов определил, что статистика отсутствует. План выполнения потребует действия по созданию статистики, на что понадобится дополнительное время при выполнении запроса. Таким образом, при отсутствии статистики выполнение запроса может оказаться слишком долгим, и вы не сможете реально оценить производительность запроса.

Когда вы выполняете запросы и просматриваете план, оптимизатор запросов может потребовать выполнения некоторых корректирующих действий, таких как создание и обновление статистики или создание индекса. В этот момент вы можете напрямую создать или обновить статистику индексов. С помощью оператора `CREATE STATISTICS` можно создавать статистику для:

- неиндексированных колонок;
- всех колонок, отличных от первой колонки составного индекса;
- вычисляемых колонок, только если на такую колонку можно создать индекс;
- колонок, которые объявлены как данные типов `image`, `text` или `ntext`.

Создание статистики вручную может принести пользу, когда вы имеете колонку, которая не нуждается в индексе, но статистика на эту колонку может

оказаться полезной для создания более эффективного плана. На индексированные колонки тоже нужна статистика, но здесь сервер может создавать статистику автоматически, и это предпочтительнее.

В определенный момент статистика может оказаться устаревшей, что может повлиять на производительность и работу оптимизатора запросов. Это особенно заметно после массового обновления данных, ведь распределение данных меняется. При этом может измениться процент появления определенных значений.

Сервер SQL обновляет статистическую информацию, когда она устаревает. Количество, а точнее, процент изменений в данных определяет частоту обновления статистики. Например, для таблицы, содержащей 10 000 строк, обновление может потребоваться, когда 1000 индексных значений изменятся (1%). Но если таблица содержит 100 000 записей, то 1000 — это всего лишь 0,1, и обновлять что-то еще рано.

Сервер SQL выдает предупреждение, когда статистика устарела или недоступна. Это предупреждение появляется, когда просматривается план выполнения, иначе пользователь его не увидит. Вы можете использовать утилиту SQL Profiler для наблюдения за событием Missing Column Statistic (отсутствует статистика поля). Этот класс сообщений определяет, когда статистика отсутствует.

Вы можете выполнять оператор `UPDATE STATISTICS` для обновления статистики вручную. Общий вид команды:

```
UPDATE STATISTICS table | view
    [
        index
        | ( statistics_name [ ,...n ] )
    ]
    [    WITH
        [
            [ FULLSCAN ]
            | SAMPLE number { PERCENT | ROWS } ]
            | RESAMPLE
        ]
        [ [ , ] [ ALL | COLUMNS | INDEX ]
        [ [ , ] NORECOMPUTE ]
    ]
```

Обновляйте статистику каждый раз, когда данные в таблице значительно изменены или пополнены.

Вы можете просматривать статистику, выполняя оператор `DBCC SHOW_STATISTICS`. В общем виде команда выглядит следующим образом:

```
DBCC SHOW_STATISTICS (Таблица, Статистика)
```

Например, следующая команда отобразит статистику, собранную для созданной ранее статистики с именем `stat_Name`:

```
DBCC SHOW_STATISTICS (tbPeoples, stat_Name)
```

Листинг 4.1 содержит основные данные, которые были показаны мне после выполнения этой команды. В табл. 4.6 описаны параметры, которые вы можете увидеть в статистике.

Листинг 4.1. Результат просмотра статистики

```
Statistics for collection 'stat_Famil'.

-----------------------------------------------------------------
Updated               Rows   Rows Sampled  Steps  Density Average key
-----------------------------------------------------------------
May 16 2005 10:32PM  26     26            12     0.0     8.7307692

------------------------ ------------------------ -----------------
All density              Average Length           Columns
------------------------ ------------------------ -----------------
0.07692308               4.7307692                vcName
0.03846154               8.7307692                vcName, idPeoples

-------------------------------------------------------------
RANGE_HI_KEY           RANGE_ROWS  EQ_ROWS
-------------------------------------------------------------
АНДРЕЙ                   0.0        5.0
БОЛИК                    0.0        1.0
ВАСЯ                     0.0        1.0
ВИКТОР                   0.0        1.0
ВИТЯ                     0.0        1.0
ВЛАД                     0.0        1.0
ИВАН                     0.0        8.0
...
...
(12 row(s) affected)
```

Таблица 4.6. *Параметры статистики*

Имя параметра	Описание
`Updated`	Дата и время последнего обновления статистики
`Rows`	Количество строк в таблице
`Rows sampled`	Количество строк шаблонов для статистической информации
`Steps`	Количество распределенных шагов
`Density`	Плотность первой индексной колонки
`Average key length`	Средняя длина первой индексной колонки
`All density`	Набор префиксов индексной колонки
`Average length`	Средняя длина индексной колонки
`Columns`	Имена индексных колонок
`RANGE_HI_KEY`	Верхняя граница значений шага гистограммы
`RANGE_ROWS`	Количество строк из шаблона, которые попали в шаг гистограммы, исключая верхнюю границу
`EQ_ROWS`	Количество строк из шаблона, которые равны значению верхней границы
`DISTINCT_RANGE_ROWS`	Количество отдельных значений в шаге гистограммы, исключая верхний предел
`AVG_RANGE_ROWS`	Среднее значение повторяющихся значений шага гистограммы, исключая верхнюю границу

Просмотр статистики обычно помогает повысить производительность определенного запроса или выяснить возможную причину того, что сервер начал обрабатывать ваш запрос слишком медленно. Еще одно преимущество использования статистики заключается в том, что собранные ею данные могут оказаться полезными для анализа таблицы. Например, как мы видели в листинге 4.1, результат состоит из трех таблиц. Последняя таблица содержит список имен (именно по этому полю мы создавали статистику), а в третьей колонке показано количество появлений этого поля в таблице. Это то же самое, что выполнить оператор группировки по имени и сосчитать количество имен:

```
SELECT vcName, count(*)
FROM tbPeoples
GROUP BY vcName
```

Вы должны учитывать, что результат `GROUP BY` может отличаться от статистики, потому что SQL-запрос отображает текущее состояние таблицы, а ста-

тистика может содержать устаревшие данные, если после последнего обновления статистики были изменены данные в таблице.

Раньше команда `UPDATE STATISTICS` выполнялась для одной таблицы. Если в базе много таблиц, то чтобы обновить их все, приходилось серьезно потрудиться. Чтобы упростить жизнь программистам, была введена процедура `sp_updatestats`. В общем виде она выглядит следующим образом:

```
sp_updatestats [[@resample =] 'resample']
```

После вызова этой процедуры будет обновлена статистика всех таблиц.

В качестве параметра процедура может получать только одно значение: `resample`. Если оно указано, то при обновлении статистики (`UPDATE STATISTICS`) для каждой таблицы будет использоваться опция `RESAMPLE`.

Чтобы не заботиться об обновлении статистики, можно сделать его автоматическим. Для этого используется процедура `sp_autostats`, которая в общем виде выглядит следующим образом:

```
sp_autostats [ @tblname = ] 'table_name'
    [ , [ @flagc = ] 'stats_flag' ]
    [ , [ @indname = ] 'index_name' ]
```

Здесь три параметра:

- `@tblname` — имя таблицы или представления, для которой должно быть включено автоматическое обновление;
- `@flagc` — в этом параметре указывается действие, которое нужно выполнить: включить (`ON`) или выключить (`OFF`) автоматическое обновление статистики;
- `@indname` — имя индекса, для которого должно быть включено автоматическое обновление.

Обязательным параметром является только имя таблицы. Если выполнить команду, указав только таблицу, то в результате мы получим информацию о существующей статистике и индексах:

```
EXEC sp_autostats tbPeoples
```

Результат состоит из трех полей:

- имя индекса;
- состояние автоматического сбора `ON` (включено) или `OFF` (выключено);
- дата последнего обновления статистики.

Если указать таблицу и действие, которое нужно выполнить, то действие будет применено ко всем существующим в таблице индексам:

```
EXEC sp_autostats tbPeoples, 'ON'
```

В результате вы увидите сообщение о количестве индексов, автоматический сбор статистики на которые был включен:

Automatic statistics maintenance turned ON for 5 indices. (Поддержка автоматической статистики включена для пяти индексов).

Следующий пример показывает, как отключить статистику для первичного ключа таблицы `tbPeoples`:

```
EXEC sp_autostats tbPeoples, 'OFF', PK_idPeoples
```

Если первичный ключ является автоматически увеличиваемым числом, то для него сбор статистики в 90% случаев не нужен, т. к. это не принесет никакой пользы.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\stat.sql на компакт-диске, прилагаемом к книге.

4.8. Управление пользователями

В MS SQL Server есть два способа авторизации пользователей:

- Windows — авторизация на основе системы безопасности, встроенной в ОС Windows. На данный момент в ОС есть достаточно мощные средства проверки авторизации и проверки подлинности, и почему бы их не использовать?
- смешанный режим — при этом пользователь не обязательно должен быть авторизован в самой ОС, могут использоваться учетные записи, созданные в сервере MS SQL Server.

Безопасность авторизации MS SQL Server оставляет желать лучшего, да и в большинстве случаев просто неудобна. Если пользователь должен иметь доступ к ОС, а для него уже есть учетная запись, то создавать вторую запись в сервере баз данных неудобно и неэффективно. В этом случае придется поддерживать слишком много учетных записей.

4.8.1. Управление пользователями сервера

Как мы уже сказали, при использовании смешанного режима к базе данных могут подключаться как пользователи SQL Server, так и пользователи Windows. Но для этого пользователи должны быть зарегистрированы в базе данных. Как регистрировать, мы увидим чуть ниже, а сейчас посмотрим, как добавлять пользователя к SQL Server.

Добавление пользователя SQL-сервера

Итак, если пользователь уже существует в Windows, то его не надо добавлять в SQL Server, достаточно зарегистрировать в базе данных. Но если вы не хотите "прописывать" пользователя в Windows, а необходимо разрешить только доступ к SQL Server, то сначала нужно добавить новую учетную запись пользователя в SQL Server.

Процедура добавления учетной записи в общем виде выглядит следующим образом:

```
sp_addlogin [ @loginame = ] 'login'
    [ , [ @passwd = ] 'password' ]
    [ , [ @defdb = ] 'database' ]
    [ , [ @deflanguage = ] 'language' ]
    [ , [ @sid = ] sid ]
    [ , [ @encryptopt = ] 'encryption_option' ]
```

Рассмотрим параметры, которые нужно передать этой процедуре:

- имя пользователя (login);
- пароль, который должен использоваться при авторизации пользователя. Если пароль не указан, то будет использоваться пустой пароль. Если пароль указан, то он будет зашифрован и сохранен в системной таблице;
- база данных, которая будет использоваться по умолчанию при подключении пользователя. Если этот параметр не указан, то будет использоваться база данных master;
- язык по умолчанию. Если параметр не указан, то будет использоваться значение по умолчанию для сервера;
- идентификатор. Каждый пользователь имеет свой идентификатор. Если не указать явно этот параметр, то он будет сгенерирован сервером автоматически;
- метод шифрования пароля. Если этот параметр не указан, то для шифрования будет использоваться метод по умолчанию. Но можно указать одно из следующих значений:
  - `skip_encryption` — пароль уже зашифрован, и сервер не должен повторять шифрование, а сохранить результат как есть;
  - `skip_encryption_old` — пароль уже зашифрован старой версией SQL Server, и повторно шифровать не надо. Этот параметр нужно использовать только при переходе со старой версии SQL Server на новую.

Как видите, обязательным является только указание имени пользователя. Все остальные параметры можно опускать, и в этом случае сервер будет использовать нулевые значения или значения по умолчанию.

Давайте попробуем создать нового пользователя MS SQL Server с именем `Hacker`, указав ему пароль и базу данных по умолчанию:

```
EXEC sp_addlogin 'Hacker', 'qwerty', 'FlenovSQLBook'
```

Для удаления пользователя используется процедура `sp_droplogin`, которой нужно передать имя удаляемого пользователя сервера. Например:

```
EXEC sp_droplogin 'Hacker'
```

Управление пользователями SQL-сервера

Редко встретишь систему, в которой учетные записи не изменяются. Все в жизни подвержено изменению, особенно пароли пользователей, которые должны регулярно меняться для повышения безопасности. Для смены паролей используется хранимая процедура `sp_password`, которой нужно передать три параметра:

- старый пароль;
- новый пароль;
- имя пользователя, пароль которого нужно сменить.

Например, следующая команда меняет пароль для пользователя `Hacker`, которого мы создали ранее:

```
EXEC sp_password 'qwerty', 'ytrewq', 'Hacker'
```

С помощью системной процедуры `sp_defaultdb` можно поменять базу данных по умолчанию. В качестве параметров нужно передать имя пользователя и имя новой базы данных по умолчанию. Например, следующий запрос устанавливает для пользователя `Hacker` по умолчанию базу данных master:

```
EXEC sp_defaultdb 'Hacker', 'master'
```

Регистрация на сервере пользователя Windows NT

Несмотря на то, что учетные записи Windows уже готовы к использованию в SQL Server, их можно явно зарегистрировать на сервере баз данных. Для этого используется хранимая процедура `sp_grantlogin`. Этой процедуре нужно передать в качестве параметра имя пользователя или группы из Windows NT. Следующий пример регистрирует гостевую учетную запись:

```
EXEC sp_grantlogin 'notebook\Гость'
```

В данном случае добавляется учетная запись пользователя Windows. Такие имена пишутся в формате `домен\имя`, где `домен` — это имя домена или компьютера, а `имя` — имя пользователя в домене или в указанном компьютере. Мой компьютер называется `notebook`, поэтому и такое имя гостевой учетной записи.

Можно регистрировать целые группы пользователей Windows NT, указав вместо имени пользователя имя группы.

Чтобы отменить регистрацию, используется процедура `sp_revokelogin`, которой нужно передать имя удаляемой учетной записи:

```
EXEC sp_revokelogin 'notebook\Гость'
```

Чтобы явно запретить подключение конкретного пользователя Windows NT к MS SQL Server, можно вызвать процедуру `sp_denylogin`. Ей следует передать действительное имя пользователя Windows NT или группы, которой нужно запретить доступ.

```
EXEC sp_denylogin 'notebook\Администратор'
```

Процедуры `sp_grantlogin`, `sp_revokelogin` и `sp_denylogin` нельзя вызывать из транзакций, определенных пользователем. Выполняйте эти процедуры вне транзакций. Для выполнения процедур вам необходимы права роли `securityadmin` или `sysadmin`.

4.8.2. Управление пользователями базы данных

Для добавления учетных записей к базе данных используются системные процедуры `sp_adduser` и `sp_grantdbaccess`. Для начала рассмотрим первую из них. Процедуре `sp_adduser` нужно передать три параметра (только первый параметр является обязательным):

- имя пользователя (login);
- имя пользователя в базе данных. Если этот параметр не указан, то будет использовано имя из первого параметра;
- имя группы или роли, в которую автоматически попадает пользователь.

При добавлении пользователя указанное имя уже должно существовать на сервере MS SQL Server или в ОС Windows.

Рассмотрим пример. В ОС Windows уже существует гостевая учетная запись. Давайте выдадим ей права на доступ к текущей базе данных:

```
EXEC sp_adduser 'notebook\Гость'
```

Чтобы добавить пользователя `Hacker`, которого мы создали ранее, в текущую базу данных, выполним следующую команду:

```
EXEC sp_adduser 'Hacker'
```

Процедура `sp_adduser` считается устаревшей и оставлена только для совместимости со старыми приложениями. На данный момент рекомендуется использовать процедуру `sp_grantdbaccess`, которой нужно передать следующие параметры:

- имя пользователя (login), которое зарегистрировано в ОС Windows NT или создано на сервере SQL Server;
- имя пользователя в базе данных. Если этот параметр не указан, то будет использовано имя из первого параметра.

Таким образом, добавление гостевой учетной записи с помощью процедуры `sp_grantdbaccess` будет выглядеть следующим образом:

```
EXEC sp_grantdbaccess 'notebook\Гость'
```

Помните, что пользователь добавляется только в текущую базу. Если пользователь должен иметь право доступа к объектам разных баз данных, то необходимо добавить его в каждую.

При создании реальных приложений никогда не разрешайте доступ непривилегированным учетным записям. Гостевой пароль используется в Windows для предоставления доступа всем пользователям сети и меньше всего защищается администраторами, потому что обладает минимальными правами.

Если вы разрабатываете службу для Web, то для каждого пользователя Интернета невозможно завести на сервере учетную запись. Вот в таких случаях используется гостевой доступ, но при этом посетитель должен обладать минимальными правами и только на просмотр тех данных, которые необходимы. О правах доступа мы еще поговорим в *разд. 4.9*.

При разработке корпоративных программ лучше всего сделать так, чтобы каждый пользователь работал под своей учетной записью и с собственными правами. Если нужно объединить права, то можно использовать группы или роли.

Для удаления используется процедура `sp_dropuser`, которой нужно передать имя пользователя базы данных (мы его указывали во втором параметре процедуры `sp_adduser` или `sp_grantdbaccess`). В следующем примере мы удаляем гостевую учетную запись из текущей базы:

```
EXEC sp_dropuser 'notebook\guest'
```

Процедура `sp_dropuser` является устаревшей, и вместо нее рекомендуется использовать `sp_revokedbaccess`, которой также нужно передать имя удаляемой учетной записи:

```
EXEC sp_revokedbaccess 'notebook\guest'
```

Управление пользователями — дело хорошее, но вы должны уметь определять, какие пользователи существуют в системе. Для этого используется хранимая процедура `sp_helpuser`. Выполните эту процедуру, и перед вами появится таблица с информацией о пользователях текущей базы данных. Результирующая таблица состоит из следующих полей:

- `UserName` — имя пользователя;
- `GroupName` — название роли, в которую входит пользователь;
- `LoginName` — имя, используемое для входа на сервер;
- `DefDBName` — база данных по умолчанию;
- `UserID` — идентификатор пользователя в базе данных;
- `SID` — пользовательский идентификатор безопасности.

4.8.3. Роли

В Windows и других ОС для управления правами существуют группы и пользователи. С помощью групп мы можем объединять пользователей и назначать права всем сразу. Это проще, чем назначать права каждому в отдельности. В базах данных для этих целей существует понятие *роли*.

Допустим, что 100 пользователей должны иметь право читать данные из определенной таблицы. Давать каждому из них это право утомительно. Намного проще создать роль, которой разрешено читать, а потом всех нужных пользователей включить в нее. Результат подобен группировке.

В MS SQL Server существует два типа ролей — серверные и баз данных. Серверные роли заранее определены, и их изменять нельзя. Откройте в Enterprise Manager ветку **Security | Server roles** (Безопасность | Серверные роли), и в правой части окна можно будет увидеть список встроенных ролей (рис. 4.3). Что разрешено делать пользователю соответствующей роли, можно определить по описанию.

У каждой базы данных могут быть свои роли, которые определяют права доступа на объекты этой базы данных. Многие администраторы не любят возиться с этими правами, а всем устанавливают встроенную по умолчанию роль `public`, которая позволяет делать практически все. Если прав роли `public` не хватает, тогда просто включают пользователя в серверную роль

`System Administrators`. И вот тут база данных становится уязвимой. Если хакер получит доступ к базе данных с такими правами, то он становится богом не только базы, но и всего сервера, если сервер работает под системной учетной записью.

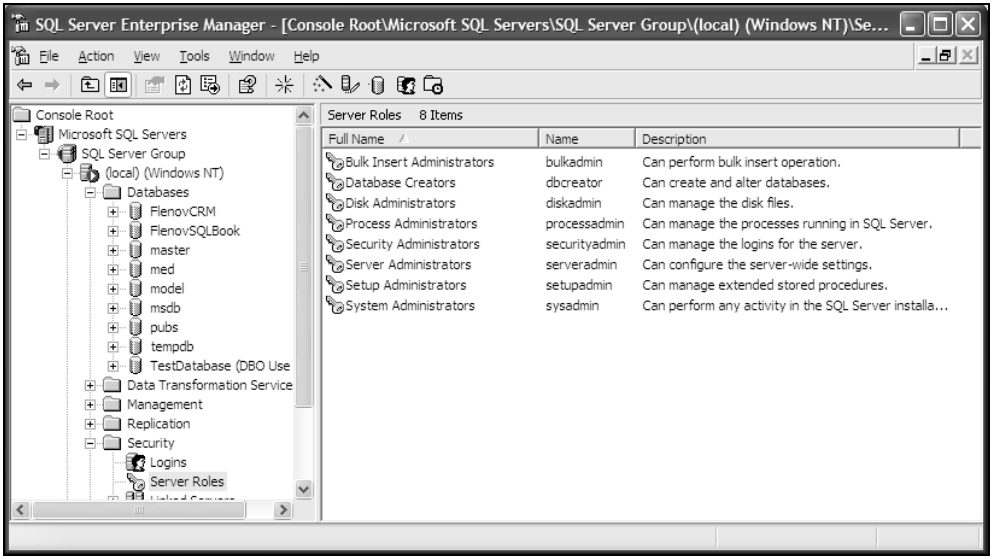


Рис. 4.3. Просмотр серверных ролей

Каждому пользователю нужно давать свои права, в которых разрешены только необходимые действия, а то, что не разрешено, обязательно должно быть запрещено. Роли, которые уже существуют в сервере, использовать не рекомендуется, потому что их права слишком демократичны. Чтобы они вас не смущали, лучше удалить их все, особенно любимую ленивыми администраторами роль `public`.

Мы уже достаточно сказали о роли `public`, пора узнать, что это такое и откуда она берется. Это специальная серверная роль, которая существует в каждой базе данных и обладает следующими свойствами:

- содержит права по умолчанию для пользователей базы данных;
- не может содержать пользователей или ролей;
- она содержится в каждой базе данных, включая системные;
- эту роль нельзя удалять.

4.8.4. Создание и удаление ролей

Для создания новой роли используется хранимая процедура `sp_addrole`. Ей нужно передать два параметра:

- имя новой роли;
- имя владельца. Этот параметр необязательный, и если его не указать, то будет использоваться значение по умолчанию — `dbo`.

Посмотрим пример использования процедуры:

```
EXEC sp_addrole 'Buh'
```

В данном запросе `Buh` — сокращение от слова "бухгалтерия".

Для удаления используется хранимая процедура `sp_droprole`, которой нужно передать имя удаляемой роли. Например, следующая команда удалит ранее созданную роль:

```
EXEC sp_droprole 'Buh'
```

Можете выполнить команду, чтобы убедиться в ее работоспособности, но после этого восстановите бухгалтеров, чтобы мы с ними еще поиграли.

4.8.5. Управление ролями

Раз есть роль, которая содержит пользователей, то должна быть и процедура, которая будет добавлять этих пользователей. Это действие выполняет процедура `sp_addrolemember`. Процедуре нужно передать два параметра:

- название роли, в которую нужно добавить существующего в этой же базе данных пользователя;
- имя пользователя.

Если вы не удалили пользователя и роль, которых мы создали ранее в этой главе, то давайте добавим гостевую учетную запись в роль `Buh`:

```
EXEC sp_addrolemember 'Buh', 'notebook\guest'
```

Удалить пользователя из роли можно с помощью процедуры `sp_droprolemember`, которой нужно передать имя роли и имя удаляемой из этой роли учетной записи.

Серверные роли создавать нельзя, но можно использовать. С помощью системной процедуры `sp_addsrvrolemember` можно добавить пользователя в серверную роль. Процедуре нужно передать два параметра:

- имя пользователя, добавляемое в роль;
- имя роли, в которую нужно добавить.

У сервера есть следующие роли, которые вы можете указывать в качестве второго параметра:

- `sysadmin`;
- `securityadmin`;
- `serveradmin`;
- `setupadmin`;
- `processadmin`;
- `diskadmin`;
- `dbcreator`;
- `bulkadmin`.

Следующий пример показывает, как можно добавить пользователя `Hacker` в серверную роль `sysadmin`:

```
EXEC sp_addsrvrolemember 'hacker', 'sysadmin'
```

Для удаления пользователя из серверной роли используется процедура `sp_dropsrvrolemember`, которой передается два параметра:

- имя пользователя, которого надо удалить из роли;
- имя роли, из которой происходит удаление.

Следующий пример удаляет пользователя `Hacker` из серверной роли `sysadmin`:

```
EXEC sp_dropsrvrolemember 'hacker', 'sysadmin'
```

Если вы хотите узнать, какие пользователи уже существуют в базе данных, используйте системную процедуру `sp_helprole`. Она возвращает таблицу из названий ролей и их идентификаторы.

Для того чтобы сменить пользователю роль, можно использовать процедуру `sp_changegroup`, которой нужно передать два параметра: роль, в которую нужно добавить пользователя, и имя пользователя. Оба параметра являются обязательными.

Следующий пример позволяет поместить пользователя `Hacker` в роль `Buh`:

```
EXEC sp_changegroup 'Buh', 'Hacker'
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\user.sql на компакт-диске, прилагаемом к книге.

4.9. Права доступа

Права доступа — достаточно сложная, но очень важная тема. В чем состоит сложность? Как раз в управлении правами с помощью операторов Transact-SQL. Что бы ни говорили, но визуальное представление намного упрощает восприятие, особенно если видно сразу всю карту прав доступа.

Для предоставления прав доступа используется три оператора:

- `GRANT` — разрешить доступ;
- `DENY` — запретить доступ;
- `REVOKE` — отменить выданные ранее разрешения или запреты.

Чем можно управлять? Ну конечно же, это объекты базы данных — таблицы, процедуры, функции, представления. Помимо этого, вы можете разрешать или запрещать выполнение определенных операторов. Например, если запретить выполнение оператора `CREATE DATABASE`, то пользователь не сможет создавать собственные базы данных.

Но прежде чем мы начнем рассматривать операторы, нужно поговорить о стратегии безопасного распределения прав. Если вы хотите, чтобы ваш сервер был максимально безопасен, то необходимо следовать правилу — запрещено все, что явно не разрешено. Каждый пользователь должен иметь только те права, которые ему действительно необходимы, и ничего лишнего.

Права для каждого пользователя делегируются не только правами, которые вы предоставили конкретному пользователю, но и правами роли, в которую включен пользователь, а ролей может быть несколько. Одна роль может предоставлять доступ, другая запрещать. Что тогда получится? Какие будут права, и сможет ли пользователь получить доступ к объекту? Все это и многое другое вы узнаете первыми.

Теперь давайте рассмотрим каждый из этих операторов. Я всегда стараюсь в своих работах уделять большое внимание безопасности, а для базы данных основой безопасности как раз является распределение прав.

4.9.1. Разрешение доступа

Для назначения разрешающих прав доступа используется оператор `GRANT`. Внешний вид зависит от того, на что выделяются права. Если это операторы, то `GRANT` выглядит следующим образом:

```
GRANT { ALL | statement [ ,...n ] }
TO security_account [ ,...n ]
```

Операторы SQL, на которые вы можете назначать права доступа для пользователя:

- `CREATE DATABASE`;
- `CREATE DEFAULT`;
- `CREATE FUNCTION`;
- `CREATE PROCEDURE`;
- `CREATE RULE`;
- `CREATE TABLE`;
- `CREATE VIEW`;
- `BACKUP DATABASE`;
- `BACKUP LOG`.

Следующие операторы не требуют распределения прав для того, чтобы быть выполненными пользователями, потому что они автоматически назначаются всем пользователям:

- `BEGIN TRANSACTION`;
- `COMMIT TRANSACTION`;
- `ROLLBACK TRANSACTION`;
- `SAVE TRANSACTION`;
- `PRINT`;
- `RAISERROR`;
- `SET`.

Рассмотрим пример, в котором пользователю с именем `Mikhail` выделяются права на создание таблиц и представлений:

```
GRANT CREATE TABLE, CREATE VIEW
TO Mikhail
```

Как мы уже говорили, для упрощения работы с правами доступа можно использовать роли. Допустим, что у нас есть 10 учетных записей для работников бухгалтерии, и все они объединены в одну роль `Buh`. Если все работники роли нуждаются в возможности создания таблиц, то можно назначить это разрешение всей роли:

```
GRANT CREATE TABLE, CREATE VIEW
TO Buh
```

Если нужно разрешить выполнение всех перечисленных выше операторов, то, чтобы не писать их все, можно воспользоваться ключевым словом `ALL`. Следующий пример предоставляет полный доступ роли `Buh`:

```
GRANT ALL
TO Buh
```

Если права назначаются на объект базы данных, то оператор выглядит следующим образом:

```
GRANT
    { ALL [ PRIVILEGES ] | permission [ ,...n ] }
    {
        [ ( column [ ,...n ] ) ] ON { table | view }
        | ON { table | view } [ ( column [ ,...n ] ) ]
        | ON { stored_procedure | extended_procedure }
        | ON { user_defined_function }
    }
TO security_account [ ,...n ]
[ WITH GRANT OPTION ]
[ AS { group | role } ]
```

При добавлении прав доступа на объекты необходимо указать оператор `GRANT`, за которым идет перечисление разрешений на объект. После ключевого слова `ON` пишем имя объекта, а после `TO` — имя пользователя или роли. В упрощенном варианте распределение прав выглядит следующим образом:

```
GRANT разрешения
ON объект
TO пользователь
```

Например, следующей командой мы разрешаем пользователю `Hacker` выполнять оператор `SELECT` в таблице `tbPeoples`:

```
GRANT SELECT
ON tbPeoples
TO hacker
```

Еще пример — бухгалтерия должна иметь право добавлять, изменять и удалять записи из таблицы работников. Для разрешения этих действий выполняем следующий запрос:

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON tbPeoples
TO Buh
```

Если пользователю нужно предоставить все права на объект, то, чтобы не перечислять их, можно написать ключевое слово `ALL`:

```
GRANT ALL
ON tbPeoples
TO Buh
```

Необходимо отметить, что по стандарту надо писать `ALL PRIVILEGES`, это Microsoft разрешила ленивым программистам не писать длинное слово `PRIVILEGES`. Я, например, всегда забываю, как оно пишется, поэтому благодарен корпорации Microsoft. Итак, если следовать стандарту, то мы должны были бы написать запрос на изменение привилегий следующим образом:

```
GRANT ALL PRIVILEGES
ON tbPeoples
TO Buh
```

У оператора `GRANT` есть одна очень интересная опция — `WITH GRANT OPTION`. Если ее указать, то пользователь сможет делегировать свои права другим пользователям:

```
GRANT SELECT, INSERT, DELETE
ON tbPeoples
TO Buh
WITH GRANT OPTION
```

После выполнения этого оператора пользователи `Buh` могут не только сами производить выборку, обновление и удаление, но и выдавать эти же права другим пользователям. Обновление бухгалтерам запрещено, а значит, это право нельзя будет делегировать.

Выделять права можно и на отдельные колонки таблицы или представления. Например, следующая команда разрешает выборку данных только из полей `idPeoples`, `vcFamil` и `vcName` таблицы `tbPeoples`:

```
GRANT SELECT
ON tbPeoples (idPeoples, vcFamil, vcName)
TO Buh
```

4.9.2. Запрещение доступа

Зачем нужен запрет, ведь если пользователю не предоставлено право, то он его уже не имеет? Я тоже думал, что не нужно, пока не встретился с настройкой сложных прав из нескольких ролей и множеством пользователей. Когда с сервером работают несколько пользователей с одинаковыми правами, то просто назначаем им одно и то же и только надеемся, что пользователи не натворят бед.

Если в системе сотня пользователей, то надеяться на их знания и умения абсолютно бесполезно.

Когда пользователей много, приходится строить сложную структуру ролей и прав пользователей. И вот теперь мы подходим к проблеме необходимости

запрета. Допустим, что у вас есть группа бухгалтеров. На работу нанимают нового бухгалтера, и вы, конечно же, включаете его в соответствующую роль. Но эта роль обладает слишком высокими правами, и вы хотите защититься от нелепых и неумелых действий нового работника. Для этого нужно запретить новому работнику изменять особо важные таблицы и данные. Как это сделать? Можно не вводить пользователя в опасную роль, а выдать собственные права, но это может быть накладно. Если запретов немного, то можно оставить пользователя в роли, но явно сформулировать запрет на опасные объекты.

Если пользователю делегируются разрешающие права от роли, то они не действуют, если непосредственно самому пользователю добавлен запрет. Таким образом, нового бухгалтера можно добавить в роль `Buh`, и ему будут делегированы права этой роли. Но чтобы новичок не натворил бед, ему можно явно запретить изменение особо важных данных. Для этого запрещаем изменение непосредственно пользователю.

Для задания запретов используется оператор `DENY`, который так же имеет два варианта — на операторы и на представления. Рассмотрим каждый из них.

Общий вид команды `DENY`:

```
DENY { ALL | statement [ ,...n ] }
TO security_account [ ,...n ]
```

Операторы, которые могут использоваться, такие же, как и у `GRANT`. Например, следующий запрос явно запрещает пользователю `Hacker` создавать таблицы и представления:

```
DENY CREATE TABLE, CREATE VIEW
TO Hacker
```

Если нужно отменить все права на операторы, то можно указать ключевое слово `ALL`. В следующем примере отменяются права для бухгалтеров:

```
REVOKE All
FROM Buh
```

Запрещение для объектов выглядит так:

```
DENY
    { ALL [ PRIVILEGES ] | permission [ ,...n ] }
    {
        [ ( column [ ,...n ] ) ] ON { table | view }
        | ON { table | view } [ ( column [ ,...n ] ) ]
        | ON { stored_procedure | extended_procedure }
        | ON { user_defined_function }
```

```
    }
TO security_account [ ,...n ]
[ CASCADE ]
```

Давайте попробуем создать разрешение на чтение данных из таблицы `tbPeoples`:

```
GRANT SELECT
ON tbPeoples
TO public
```

Несмотря на то, что по умолчанию для пользователя доступ на чтение будет разрешен, мы запретим все для пользователей `Hacker` и `Mikhail`:

```
DENY SELECT, INSERT, UPDATE, DELETE
ON tbPeoples
TO Hacker, Mikhail
```

При запрете, так же как и при разрешении, можно указывать колонки, на которые мы хотим воздействовать. Следующий пример запрещает работать с ключевой колонкой и фамилией:

```
DENY SELECT, INSERT, UPDATE, DELETE
ON tbPeoples (idPeoples, vcFamil)
TO Hacker
```

В операторе `GRANT` была возможность разрешать пользователю делегировать свои права. Как отследить, кому разрешалось, а кому нет? А определять и не надо. Просто можно при запрещении указать опцию `CASCADE`, в этом случае `DENY` повлияет не только на указанного пользователя, но и на тех, кому были делегированы права. Следующий пример запрещает работу с таблицей `tbPeoples` пользователю `Hacker` и тем пользователям, кому `Hacker` делегировал свои права:

```
DENY SELECT, INSERT, UPDATE, DELETE
ON tbPeoples
TO Hacker
CASCADE
```

4.9.3. Отмена прав доступа

Иногда бывает необходимо отменить назначенные права, чтобы действовали значения по умолчанию или делегированные права доступа. Вспоминаем пример с начинающим бухгалтером, которому мы хотели явно запретить выполнение опасных действий. Когда бухгалтер набрался опыта, то, чтобы он получил все права роли `Buh`, нужно просто удалить все, что мы явно запретили

для его учетной записи. Не стоит менять запрещение на разрешение, иначе вы запутаетесь в правах, лучше использовать отмену.

Для отмены прав доступа, которые были делегированы ранее, служит команда REVOKE, которая может использоваться для операторов или объектов. Рассмотрим оба варианта. Для начала рассмотрим отмену разрешений для операторов:

```
REVOKE { ALL | statement [ ,...n ] }
FROM security_account [ ,...n ]
```

Операторы, которые вы можете указывать в этой команде, такие же, как у операторов GRANT и DENY. В работе всех этих операторов также есть большое сходство. Следующий пример показывает, как можно отменить выделенные ранее права на создание таблицы и представления пользователю Hacker:

```
REVOKE CREATE TABLE, CREATE VIEW
FROM Hacker
```

Как видите, разница только в имени оператора — REVOKE, и во второй строчке вместо TO мы написали FROM, хотя можно и TO, результат работы не изменится. Я больше люблю FROM (от), потому что мы отбираем права от пользователя или роли, и такое написание более соответствует смыслу выполняемой команды.

После выполнения этой команды права на создание таблицы и представления отменяются (будь то разрешение или запрет), и будет использоваться значение по умолчанию.

Для отмены прав на объекты служит оператор REVOKE следующего вида:

```
REVOKE [ GRANT OPTION FOR ]
    { ALL [ PRIVILEGES ] | permission [ ,...n ] }
    {
        [ ( column [ ,...n ] ) ] ON { table | view }
        | ON { table | view } [ ( column [ ,...n ] ) ]
        | ON { stored_procedure | extended_procedure }
        | ON { user_defined_function }
    }
{ TO | FROM }
    security_account [ ,...n ]
[ CASCADE ]
[ AS { group | role } ]
```

Следующий пример отменяет созданный в предыдущем разделе запрет для пользователя Hacker на работу с таблицей tbPeoples:

```
REVOKE SELECT, INSERT, UPDATE, DELETE
ON tbPeoples
```

```
FROM Hacker
CASCADE
```

При отмене прав вы также можете использовать опцию `CASCADE`, при этом будут отменены все права, делегированные указанным пользователем.

4.9.4. Информация о правах доступа

Для распределения прав используется всего три команды, но даже при этом очень сложно уследить за тем, что мы разрешили, а что запретили. Именно поэтому в начале главы я говорил, что хорошо бы иметь удобную графическую утилиту для управления правами, и у MS SQL Server есть такая утилита — Enterprise Manager.

Но использование графической утилиты не всегда удобно, поэтому можно обойтись и командами Transact-SQL, а точнее, системной процедурой `sp_helprotect`. Этой процедуре следует передать четыре параметра, и мы рассмотрим их по очереди.

В общем виде команда выглядит следующим образом:

```
sp_helprotect [ [ @name = ] 'object_statement' ]
    [ , [ @username = ] 'security_account' ]
    [ , [ @grantorname = ] 'grantor' ]
    [ , [ @permissionarea = ] 'type' ]
```

Если параметров нет, то будут показаны все разрешения текущей базы данных. В следующем примере вы видите права доступа моей тестовой базы:

```
Owner Object     Grantee   Grantor ProtectType Action     Column
-------------------------------------------------------------
dbo   syscolumns  public    dbo     Grant       Select     (All+New)
dbo   syscomments public    dbo     Grant       Select     (All+New)
...
dbo   tbPeoples   Buh       dbo     Grant       Delete      .
dbo   tbPeoples   Buh       dbo     Grant       Insert      .
dbo   tbPeoples   Buh       dbo     Grant       Select     (All+New)
dbo   tbPeoples   Buh       dbo     Grant       Update     (All+New)
.     .           Buh       dbo     Grant       Backup Database  .
.     .           Buh       dbo     Grant       Create Function  .
.     .           Buh       dbo     Grant       Create Rule      .
.     .           Buh       dbo     Grant       Create Table     .
.     .           Buh       dbo     Grant       Create View      .
.     .           Hacker    dbo     Deny        Create Table     .
```

```
.     .           Hacker    dbo     Deny        Create View      .
...

(33 row(s) affected)
```

Результат состоит из 7 колонок:

- `Owner` — владелец объекта. Владелец может быть только у объектов, но не у операторов;
- `Object` — объект, права которого описаны в строке;
- `Grantee` — кому назначены права;
- `Grantor` — кто назначил права;
- `Protect Type` — тип прав доступа. Здесь может быть одно из двух значений `Grant` (разрешение на объект или оператор) или `Deny` (запрет);
- `Action` — SQL-оператор, на который установлено разрешение;
- `Column` — колонки. Если при создании прав доступа колонка не указывалась, то разрешение идет на все колонки и в соответствующей строке будет выведено (`All+New`), т. е. все колонки, существующие в объекте, и все, которые будут созданы.

Теперь посмотрим, какие параметры мы можем использовать с системной процедурой. Первый параметр — это объект, права которого мы хотим увидеть (`@name`). Следующий пример отображает права на таблицу `tbPeoples`:

```
EXEC sp_helprotect 'tbPeoples'
```

Чтобы посмотреть права на оператор, нужно указать этот оператор. Например:

```
EXEC sp_helprotect 'CREATE TABLE'
```

Следующий параметр, который мы можем указывать, — имя пользователя (`@username`). Вот как получить все записи о безопасности таблицы `tbPeoples` и пользователя `Hacker`:

```
EXEC sp_helprotect 'tbPeoples', 'Hacker'
```

Имя объекта является необязательным параметром, поэтому можно увидеть все записи пользователя `Hacker`, но для этого нужно указать имя параметра `@username`:

```
EXEC sp_helprotect @username='Hacker'
```

Параметр `@grantorname` позволяет ограничить вывод только записями, в которых право выдано определенным пользователям. Например, вы хотите узнать, кому делегировал права `Hacker`. Для этого выполняем запрос:

```
EXEC sp_helprotect @grantorname='Hacker'
```

Последний параметр `@permissionarea` позволяет искать записи, относящиеся к определенному типу, — права доступа к объекту или к оператору. Чтобы получить все записи, касающиеся ограничений, относящихся к объектам, укажите в качестве значения букву `o` (object permission, разрешения объектов), для получения ограничений операторов нужно указать `s` (statement permission, разрешения операторов). Например:

```
EXEC sp_helprotect @permissionarea='s'
```

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\grant.sql на компакт-диске, прилагаемом к книге.

4.10. Резервное копирование и восстановление

Многие считают, что техника сейчас надежна, и из-за своей лени никогда не делают резервных копий. Техника хороша, но на моих глазах "вылетело" уже несколько винчестеров, украдено из офисов 5 компьютеров, и даже полностью сгорел вместе с кабинетом один сервер. А ураганы и наводнения? А действия террористов? Закрывать на это глаза нельзя. Необходимо делать все, чтобы данные были сохранены в любой ситуации.

Резервное копирование/восстановление данных — хороший способ перенести данные с одного сервера на другой. Например, вам нужно скопировать текущую базу данных на свой домашний компьютер, чтобы можно было в спокойной обстановке проанализировать данные в выходные дни. Для этого создается резервная копия на рабочем сервере и восстанавливается на вашем домашнем компьютере.

От крушения данных из-за проблем с жестким диском спасает использование технологии RAID с зеркальным отображением. Но от других проблем, например, стихийных бедствий, это не спасет.

В MS SQL Server есть три метода создания резервных копий:

- Simple — это самая простая модель. При ее использовании после каждого создания резервной копии высвобождается свободное место на диске, занимаемое журналом транзакций. Это значит, что файл журнала не будет расти беспредельно. При использовании этой модели достигается самая высокая производительность при массовой загрузке данных в базу. С другой стороны, нет никакой возможности восстановить изменения, которые были сделаны с момента создания последней резервной копии. В случае

аварии просто восстанавливаем последнюю копию, а изменения, сделанные после этого, бесследно теряются.

- Full — полное резервное копирование. Это самая мощная модель, при которой можно создавать промежуточные копии, например, резервировать журнал транзакций. Мощь модели заключается и в том, что базу можно восстановить по состоянию на любой момент времени. Например, если данные были разрушены в какой-то момент времени, то мы всего лишь восстанавливаем их по состоянию на пять минут раньше этого неприятного события.

  Журнал при таком подходе не очищается и растет, пока явно не будет зарезервирован, поэтому будьте готовы к хранению на жестком диске ненужных данных. У меня есть базы, в которых журнал за месяц вырастает так, что становится больше, чем файл данных. Помимо этого, база становится чувствительной к целостности журнала, и если целостность нарушена, то восстановление последних изменений становится проблематичным.

  Самый главный недостаток — каждая операция подробно резервируется. Это значит, что при массовой загрузке каждая операция записи журналируется, и скорость работы сервера оставляет желать лучшего.

- Bulk-Logged — это упрощенный вариант создания полной резервной копии. В этой модели при выполнении массовых операций (загрузка большого числа данных или создание индекса) в журнале сохраняется минимум необходимой информации, а именно, только факт выполнения этой операции. Именно поэтому в модели Bulk-Logged массовые изменения выполняются быстрее. При этом если выполнена подобная операция, то восстановить данные по состоянию на определенную точку времени станет уже невозможно.

4.10.1. Стратегия резервного копирования

Теперь поговорим о том, когда и что надо резервировать. При использовании простой модели вы можете выбрать создание полной резервной копии (Database complete) или дифференцированной (Database differential).

Если выбрать Database complete, то создается полная копия всех страниц базы данных. Если база занимает пару сотен мегабайт, то на простейшем сервере эта операция не отнимет много времени. А если размер достигает нескольких гигабайт, то это уже тяжело для простого сервера с малым количеством памяти, ведь резервное копирование может происходить одновременно с работой пользователей. В этом случае производительность резко упадет. Поэтому полные резервные копии нужно создавать только в нерабочее время. Я рекомендую это делать после окончания рабочего дня, а лучше ночью.

В случае использования Database differential в резервной копии сохранятся только те страницы данных, в которых произошли изменения с момента создания последней полной резервной копии. Это значит, что полная копия у вас уже должна быть. Обратите внимание, что сохраняются страницы с измененными данными, а не журнал изменений. Это значит, что если в этой странице было 10 изменений, то в резервную копию попадет только последнее состояние, а все промежуточные хранятся в журнале транзакций.

Если после создания полной резервной копии было произведено немного изменений, то намного выгоднее использовать дифференцированное копирование. Результат — маленький файл резервной копии и достаточно быстрое выполнение. Если произошло массовое изменение данных, то затраты на создание дифференцированной копии будут слишком большими, а итоговые затраты превысят экономию. В случае восстановления придется сначала восстановить полную копию и потом большую дифференцированную копию.

Итак, для маленьких баз данных можно делать полную резервную копию хоть каждый час. Для этого сценарий резервного копирования можно поместить в планировщик заданий, и, пока сервер работает, заниматься другими делами. Если база данных большая, то ее резервное копирование можно проводить ежедневно по ночам, и если есть возможность, то днем в обеденный перерыв. А в рабочее время иногда делать дифференцированное копирование.

Если вы ограничиваетесь созданием только полных копий, то я рекомендую завести для их хранения десяток носителей, например, DVD-RW-дисков. Семи носителей хватит для хранения резервных копий каждого дня недели. Помимо этого, я рекомендую сохранять на отдельных носителях резервные копии каждого последнего рабочего дня недели.

Модель Simple не позволяет восстанавливать данные по состоянию на определенный период, но благодаря описанным выше рекомендациям вы сможете вернуться в прошлое.

В моделях Full и Bulk-Logged сервер позволяет делать еще резервное копирование журнала транзакций или отдельных файлов (файловых групп) базы данных. Это мощная возможность, но нужно быть готовым к тому, что резервные копии будут занимать достаточно много места. Эти модели рекомендуются для крупных баз с важными данными.

Если база очень большая, то ее резервное копирование будет происходить долго, даже при использовании дифференцированного режима. Намного быстрее создать копию только журнала изменений, и по ним сервер сможет восстановить сами данные.

Преимущество модели Full (Bulk-Logged с ограничениями) в том, что данные могут быть восстановлены по состоянию на любой момент времени. Если

кто-то натворил бед, удалил данные или просто сделал что-то, разрушающее целостность данных, то можно восстановить базу на тестовом сервере по состоянию на пять минут раньше разрушения и перенести потерянные данные в рабочий сервер. У меня уже были случаи, когда операторы базы нарушали целостность (например, случайно удаляли данные), а потом пытались списать это дело на меня. Возможность восстановления данных по состоянию на определенный период спасала меня от жестокой расправы.

Возможность резервного копирования отдельных файлов и файловых групп тоже является достаточно мощной. В одной базе данных может быть множество таблиц, и, как показывает практика, наиболее часто изменяется не более 5 из них. Все остальные служат справочниками и редактируются раз в месяц, а то и раз в год. Вы можете поместить часто изменяемые таблицы в одну файловую группу, а неизменяемые таблицы в другую. Теперь резервные копии файловой группы с изменяемыми таблицами можно создавать сколь угодно часто и не тратить время и не нагружать процессор справочными таблицами.

Вечная проблема — как часто нужно создавать резервную копию? Все зависит от количества данных, частоты их обновления и важности. Рассмотрим основные варианты сочетания количества, частоты обновления и важности.

Если данные являются справочными, изменяются редко и в случае потери не составляет труда произвести ручное восстановление, то можно делать полную резервную копию раз в день. Если данные изменяются часто, но их немного, то возможно полное копирование каждый час, потому что малое количество данных копируется быстро.

Когда база данных достаточно большая, но изменения проходят нечасто, то мы уже говорили о том, что полная копия должна создаваться в нерабочее время. В рабочие периоды, например, ежечасно, можно выполнять дифференцированное резервное копирование или копирование журнала транзакций.

Если данных много и изменения происходят постоянно, то тут уже нужны модели Full или Bulk-Logged. Полную резервную копию делаем в нерабочее время, а в рабочее время (пару раз в день или даже каждые два часа) можно создавать резервную копию журнала.

Когда в базе данных очень часто происходят массовые загрузки или изменения, а данных мало, то используйте модель Simple. Если данных много, то лучше использовать Bulk-Logged, а не Full. Это позволит увеличить производительность и уменьшить журнал транзакций.

Вне зависимости от выбранной частоты резервного копирования я всегда делаю внеплановые копии в следующих случаях:

- по возможности полная резервная копия до и после массового обновления данных или загрузки;

- по возможности полная резервная копия после изменения структуры данных или добавления/удаления индексов;
- обязательно полная резервная копия перед праздниками;
- перед и после осуществления репликации данных.

Напоследок хочу заметить, что нельзя хранить резервные копии на том же сервере, где работает реальная база данных. Это бесполезно и не поможет в случае разрушения диска. Такая копия спасает только от нарушения данных пользователем.

Некоторые ленивые администраторы не любят заниматься этим, надеясь на бога и оборудование. Но, как говорится, "на бога надейся, а сам не плошай", поэтому я рекомендую вам один раз потратить время на разработку плана резервного копирования, создать задания, автоматизирующие резервное копирование, и поместить их в планировщик MS SQL Server. Только не забудьте после этого протестировать резервные копии на предмет корректного восстановления. Если вы неправильно создадите полную копию, то могут возникнуть проблемы при восстановлении журналов.

4.10.2. Стратегия восстановления

Восстановление данных зависит от модели резервного копирования. Если делалась полная копия, то достаточно просто ее восстановить. Если производилось дифференцированное копирование, то сначала нужно восстановить последнюю полную копию, а потом надо восстановить дифференцированную.

Для полной модели с использованием резервных копий журналов тоже сначала нужно восстановить последнюю полную копию. После этого необходимо последовательно восстановить резервные копии журнала транзакций. Нужно четко соблюдать последовательность, иначе процесс восстановления закончится ошибкой.

При восстановлении журнала нужно оставлять возможность восстановить другие журналы, если они есть. В этом случае база данных будет находиться в режиме "только для чтения".

Все вышеописанные рекомендации относятся только к восстановлению данных при нормальной работе (или при сбое) и использовании модели Simple. Такое восстановление позволяет вернуть базу данных по состоянию на момент последнего резервного копирования. Все изменения, сделанные после этого, теряются, и их вернуть не удастся.

В случае непредвиденной ситуации, и если используется модель Full или Bulk-Logged, можно попытаться восстановить все данные и вернуться к состоя-

нию непосредственно перед моментом сбоя. Если файлы данных и журнала находятся на разных дисках, то вероятность того, что диски одновременно выйдут из строя, стремится к нулю. Если пришел в негодность диск с журналом, то делайте полную копию данных и восстанавливайте их на жестком диске. А если вышел из строя диск с данными, то прежде чем начать процесс восстановления данных, необходимо попытаться сделать резервную копию журнала, в котором хранится информация о последних изменениях данных. Только после этого можно приступать к восстановлению работы сервера на новом жестком диске, и данные будут восстановлены полностью.

Недавно я увидел отличное решение с точки зрения резервного копирования. Хотя это не совсем резервное копирование, но решает именно эту задачу. Итак, в современных базах данных есть возможность репликации данных, принцип работы которой мы рассмотрим на примере.

Допустим, что у вас есть два удаленных офиса, в каждом из которых происходит массовая нагрузка на сервер данных. Можно расположить один сервер в главном офисе, а второй будет обращаться к базе через сеть. Но это достаточно большая нагрузка на сеть с точки зрения трафика и скорость доступа невысока. Намного эффективнее расположить два сервера в каждом из офисов и синхронизировать их данные. Процесс синхронизации как раз и называется репликацией.

А что произойдет, если в эту систему поставить еще один сервер, который тоже будет участвовать в репликации, но не будет использоваться в работе? Получится реальный сервер, который будет физической резервной копией. В случае возникновения проблем можно в считанные секунды превратить резервный сервер в рабочий, и он будет содержать реально работающие данные. Правда, свежесть этих данных зависит от настроек репликации (как восстановление зависит от настроек резервного копирования). Зато у вас всегда под рукой будет сервер, пусть даже намного слабее основного.

4.10.3. Резервное копирование

Для резервного копирования данных используется оператор `BACKUP DATABASE`, у которого четыре разных варианта: резервное копирование содержимого базы данных, резервное копирование файлов и файловых групп, резервное копирование журнала, усечение журнала.

Резервное копирование содержимого базы данных

В общем виде команда выглядит, как показано в листинге 4.2. На первый взгляд все очень сложно, но в действительности этот оператор не сложнее тех, что мы уже рассмотрели, просто здесь очень много опций и настроек.

Листинг 4.2. Оператор `BACKUP DATABASE` для резервного копирования базы данных

```
BACKUP DATABASE { database_name | @database_name_var }
TO < backup_device > [ ,...n ]
[ WITH
    [ BLOCKSIZE = { blocksize | @blocksize_variable } ]
    [ [ , ] DESCRIPTION = { 'text' | @text_variable } ]
    [ [ , ] DIFFERENTIAL ]
    [ [ , ] EXPIREDATE = { date | @date_var }
        | RETAINDAYS = { days | @days_var } ]
    [ [ , ] PASSWORD = { password | @password_variable } ]
    [ [ , ] FORMAT | NOFORMAT ]
    [ [ , ] { INIT | NOINIT } ]
    [ [ , ] MEDIADESCRIPTION = { 'text' | @text_variable } ]
    [ [ , ] MEDIANAME = { media_name | @media_name_variable } ]
    [ [ , ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
    [ [ , ] NAME = { backup_set_name | @backup_set_name_var } ]
    [ [ , ] { NOSKIP | SKIP } ]
    [ [ , ] { NOREWIND | REWIND } ]
    [ [ , ] { NOUNLOAD | UNLOAD } ]
    [ [ , ] RESTART ]
    [ [ , ] STATS [ = percentage ] ]
]
```

Как всегда, начнем двигаться от простого к сложному, изучая параметры оператора `BACKUP DATABASE`. Минимальная команда выглядит следующим образом:

```
BACKUP DATABASE Имя Базы данных
TO Устройство резервного копирования
```

В первой строчке мы пишем `BACKUP DATABASE` и имя резервируемой базы данных, во второй строчке — ключевое слово `TO`, после которого идет устройство резервного копирования. Как создавать устройства, мы рассмотрим чуть позже, а на начальном этапе будем указывать реальный путь в файловой системе. Для этого после `TO` необходимо указать `DISK` и путь к файлу, где будет сохранена резервная копия.

Следующий пример создает резервную копию базы данных FlenovSQLBook в файл c:\Temp\test1.bak:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
```

Явно указывать путь довольно неудобно, особенно при регулярном использовании, поэтому у MS SQL Server есть возможность создавать устройства, в которые будет происходить резервное копирование. Для этого используется хранимая процедура `sp_addumpdevice`, которая в общем виде выглядит следующим образом:

```
sp_addumpdevice [ @devtype = ] 'device_type' ,
    [ @logicalname = ] 'logical_name' ,
    [ @physicalname = ] 'physical_name'
    [ , { [ @cntrltype = ] controller_type
            | [ @devstatus = ] 'device_status'
        }
    ]
```

У процедуры пять параметров. Первые три из них являются обязательными:

- `@devtype` — это тип устройства резервного копирования. Здесь можно указывать одно из трех значений:
  - `disk` — в качестве устройства будет использоваться файл на жестком диске. При этом может использоваться только локальный диск, но не сетевой;
  - `pipe` — именованный канал;
  - `tape` — любое ленточное устройство, поддерживаемое Windows NT;
- `@logicalname` — логическое имя устройства, которое может использоваться при резервном копировании и восстановлении. Этот параметр не может быть нулевым;
- `@physicalname` — физическое имя, которое должно соответствовать всем правилам именования файлов в ОС или универсального указания сетевых имен. Максимальная длина параметра 260 символов. Если вы используете ленточное устройство, то оно должно быть поименовано как сетевое устройство. Следующий пример соответствует первому ленточному устройству — `\\.\TYPE0`;
- `@cntrltype` — в этом параметре указывается числовое значение типа устройства. Можно указывать одно из следующих значений:
  - 2 — если устройство имеет тип `disk`;
  - 5 — если устройство имеет тип `tape`;
  - 6 — если устройство имеет тип `pipe`;
- `@devstatus` — нужно ли читать метку устройства `noskip` (читать) и `skip` (пропускать).

Рассмотрим пример создания дискового устройства с именем Book_Back, которое будет связано с файлом c:\temp\SQLBook.bak:

```
EXEC sp_addumpdevice 'disk',
   'Book_Back',
   'c:\temp\SQLBook.bak'
```

Теперь, чтобы создать резервную копию в этот файл, нужно выполнить команду:

```
BACKUP DATABASE FlenovSQLBook
TO Book_Back
```

Это немного проще и удобнее. Не нужно теперь каждый раз писать путь к файлу, а достаточно только использовать имя созданного ранее устройства.

Вернемся к нашему оператору создания резервной копии и его параметрам. Дополнительные параметры указываются после ключевого слова `WITH`. Рассмотрим их:

- `BLOCKSIZE` — размер блока. Этот параметр не является обязательным, и ОС выберет размер блока по умолчанию для используемого устройства. В системах Windows 2000 по умолчанию будет использоваться число 65 536 (64 Кбайт), это максимум, который поддерживает SQL Server;
- `DESCRIPTION` — произвольный текст, содержащий описание резервной копии. Я рекомендую указывать в этом параметре особые метки, т. е. когда и зачем была сделана копия;
- `DIFFERENTIAL` — при использовании этого параметра будет происходить дифференцированное копирование, т. е. измененная часть базы. Напоминаю, что при этом у вас уже должна быть полная резервная копия;
- `EXPIREDATE` — дата, когда резервная копия считается устаревшей и может быть перезаписана;
- `RETAINDAYS` — количество дней, в течение которых нельзя перезаписать файл;
- `PASSWORD` — пароль для резервной копии. Этот же пароль необходимо будет указать во время восстановления;
- `FORMAT` или `NOFORMAT` — если указан параметр `FORMAT`, то заголовок должен быть записан на все носители резервной копии. Любые существующие заголовки перезаписываются и становятся недействительными;
- `INIT` или `NOINIT` — если указана опция `INIT`, то все существующие в файле резервные копии должны быть уничтожены. Текущая копия перезаписывает данные. При указании опции `NOINIT` существующая резервная копия остается в устройстве действительной, а новая просто добавляется.

Получается, что устройство (например, файл) может содержать несколько резервных копий. Данные не могут быть перезаписаны в следующих случаях:

- резервные копии не устарели, т. е. было указано время действия копии с помощью параметров EXPIREDATE или RETAINDAYS;
- имя создаваемой резервной копии не соответствует перезаписываемой;

- MEDIADESCRIPTION — свободный текст длиной до 255 символов, который может содержать описание устройства;
- MEDIANAME — имя устройства, с помощью которого обеспечивается дополнительная защита. При рассмотрении оператора INIT мы уже сказали, что резервная копия не может быть перезаписана, если имя создаваемой копии не соответствует имени существующей копии;
- MEDIAPASSWORD — пароль устройства. Если этот параметр задан, то при создании резервных копий на устройстве нужно будет указывать пароль. Если вы установили пароль, то такой файл можно будет перезаписать только с опцией FORMAT;
- NOSKIP или SKIP — если указан параметр NOSKIP, то, прежде всего, будут проверены даты действия, и перезапись произойдет только в том случае, если резервные копии устарели. Если указать SKIP, то проверок не будет;
- NOREWIND или REWIND — если указана опция REWIND, то ленточный накопитель после окончания резервного копирования будет перемотан и извлечен. Если указано NOREWIND, то ленточный накопитель должен остаться открытым после завершения резервного копирования. В этом случае простейший способ закрыть устройство — выполнить команду:

```
RESTORE LABELONLY FROM TAPE = <name> WITH REWIND
```

- RESTART — эта опция позволяет перезапустить прерванную операцию резервного копирования;
- NOUNLOAD — определяет, что ленточный носитель не должен быть автоматически извлечен из устройства чтения после завершения резервного копирования;
- STATS — в этом параметре можно указать некоторое значение в процентах. Каждый раз, когда будет скопирована очередная порция, составляющая указанное количество процентов от базы данных, на экран будет выводиться сообщение. Таким образом можно контролировать процесс.

Среди параметров мы встретили несколько имен, которые начинаются с MEDIA, и говорили о том, что эти параметры влияют на устройство резервного копирования. Чтобы лучше понять эти параметры, нужно четко представлять себе, из чего состоит файл резервной копии.

В начале файла идет заголовок, который называется MEDIA. Это заголовок для всего файла. Параметры MEDIADESCRIPTION, MEDIANAME, MEDIAPASSWORD влияют именно на этот заголовок. После этого друг за другом идут резервные копии. Их может быть несколько.

Вот теперь, наверное, понятно, чем отличаются параметры DESCRIPTION от MEDIADESCRIPTION. Первый влияет на конкретную резервную копию, а второй влияет на файл.

Практика резервного копирования баз данных

Теперь необходимо на практике закрепить все вышесказанное, или хотя бы основную часть. Попробуем создать несколько резервных копий в одном файле:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
```

После каждого резервного копирования смотрите на размер файла. Обратите внимание, что он постоянно растет. Это значит, что предыдущие копии не уничтожаются, и по умолчанию действует параметр NOINIT. Чтобы уничтожить старые копии, выполним операцию резервного копирования с опцией INIT:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH INIT
```

Теперь размер файла резко уменьшится, потому что в нем останется только одна копия, которую мы сейчас создали. Все предыдущие будут уничтожены.

Попробуем создать резервную копию, которая будет защищена с помощью имени устройства (параметр MEDIANAME):

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test2.bak'
WITH INIT, MEDIANAME='Test'
```

Обратите внимание, что я изменил имя файла на test2.bak. Дело в том, что в файле test1.bak уже существует резервная копия, и у нее нет имени. Если попытаться создать резервную копию в этот файл, то произойдет ошибка. Сервер нам сообщит о конфликте имен.

Теперь попробуем создать резервную копию с другим именем в файл test2.bak:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH MEDIANAME='Test2'
```

Несмотря на то, что мы не указали INIT, и новая копия должна быть добавлена, произойдет ошибка — разные имена. Даже если указать INIT и попытаться перезаписать существующую копию, все равно будет выдано сообщение об ошибке.

Единственный способ перезаписать существующую копию, защищенную именем, — указать опцию FORMAT, благодаря которой будет перезаписан заголовок устройства, а старое имя будет уничтожено:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH MEDIANAME='Test2', FORMAT
```

Теперь рассмотрим еще один способ защиты резервной копии от перезаписи — указать дату действия копии с помощью оператора EXPIREDATE или количество дней с помощью RETAINDAYS. Следующий пример говорит, что резервная копия действительна до 10 октября 2006 года:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH EXPIREDATE='10.10.2006'
```

До указанной даты перезаписать копию без указания опции FORMAT нельзя. Это значит, что следующий запрос завершит выполнение ошибкой:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH INIT
```

Напоследок создадим резервную копию, которая будет хранить описание и отображать ход выполнения (с помощью оператора STATS) после копирования каждых десяти процентов базы данных:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH INIT, FORMAT, STATS=10,
  DESCRIPTION='Это тестовая резервная копия'
```

Для создания дифференцированной копии необходимо указать параметр DIFFERENTIAL. Например:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH DIFFERENTIAL
```

Резервное копирование файлов и файловых групп

Переходим к рассмотрению следующего способа резервного копирования — файлов и файловых групп. В *главе 1* мы не раз говорили, что файловые группы

позволяют лучше управлять данными и помогут нам при резервном копировании. В один файл или группу можно поместить часто изменяемые таблицы, а в другой файл справочники, которые изменяются очень редко. Таким образом, каждый файл можно резервировать с разной частотой, а это позволит сократить время резервного копирования.

В общем виде оператор `BACKUP DATABASE` для файлов и файловых групп показан в листинге 4.3.

Листинг 4.3. Оператор `BACKUP DATABASE` для файлов и файловых групп

```
BACKUP DATABASE { database_name | @database_name_var }
    < file_or_filegroup > [ ,...n ]
TO < backup_device > [ ,...n ]
[ WITH
    [ BLOCKSIZE = { blocksize | @blocksize_variable } ]
    [ [ , ] DESCRIPTION = { 'text' | @text_variable } ]
    [ [ , ] DIFFERENTIAL ]
    [ [ , ] EXPIREDATE = { date | @date_var }
        | RETAINDAYS = { days | @days_var } ]
    [ [ , ] PASSWORD = { password | @password_variable } ]
    [ [ , ] FORMAT | NOFORMAT ]
    [ [ , ] { INIT | NOINIT } ]
    [ [ , ] MEDIADESCRIPTION = { 'text' | @text_variable } ]
    [ [ , ] MEDIANAME = { media_name | @media_name_variable } ]
    [ [ , ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
    [ [ , ] NAME = { backup_set_name | @backup_set_name_var } ]
    [ [ , ] { NOSKIP | SKIP } ]
    [ [ , ] { NOREWIND | REWIND } ]
    [ [ , ] { NOUNLOAD | UNLOAD } ]
    [ [ , ] RESTART ]
    [ [ , ] STATS [ = percentage ] ]
]
```

Если посмотреть на общий вид команды, то окажется, что она очень похожа на команду резервного копирования базы данных. Удивительно, почему Microsoft выделяет этот вариант как отдельный, я бы объединил с предыдущим.

Все параметры резервного копирования идентичны параметрам предыдущей команды, разница только в первых строчках, до ключевого слова `WITH`.

Рассмотрим пример создания резервной копии файла (файловой группы). Для этого я создал базу из нескольких файлов и двух файловых групп (PRIMARY и пользовательская группа). Сценарий создания базы данных можно увидеть в листинге 4.4. Нечто подобное мы уже создавали в *главе 1*.

Листинг 4.4. Создание базы данных из нескольких файлов и двух групп

```
CREATE DATABASE TestBatabase
ON PRIMARY
(
  NAME = tdb1_dat,
  FILENAME = 'c:\data\tdb1_data.mdf',
  SIZE = 10MB,
  MAXSIZE = 100MB,
  FILEGROWTH = 10MB
),
FILEGROUP DBGroup1
(
  NAME = tdb_group1_dat,
  FILENAME = 'c:\data\tdb_gr1_data.ndf',
  SIZE = 10MB,
  FILEGROWTH = 5%
),
(
  NAME = tdb_group2_dat,
  FILENAME = 'c:\data\tdb_gr2_data.ndf',
  SIZE = 10MB,
  FILEGROWTH = 5%
)
```

Теперь создадим резервную копию файла с именем tdb1_dat:

```
BACKUP DATABASE TestBatabase
 FILE = 'tdb1_dat'
TO DISK='c:\Temp\BackFile.bak'
WITH INIT
```

После имени резервируемой базы данных указывается параметр FILE, которому нужно присвоить имя необходимого файла.

Резервное копирование файловой группы происходит аналогичным образом, только вместо параметра `FILE` нужно указать `FILEGROUP` и присвоить этому параметру имя файловой группы:

```
BACKUP DATABASE TestBatabase
 FILEGROUP = 'DBGroup1'
TO DISK='c:\Temp\BackGroup.bak'
WITH INIT
```

При создании резервной копии файла или файловой группы необходимо правильно указывать имя. Имя файла, которое вы должны передать во время резервного копирования, — это не физический путь к файлу, а логическое имя, которое мы указывали файлу в параметре `NAME` при создании базы данных. Имя файловой группы — это имя, которое указывалось во время создания базы после ключевого слова `FILEGROUP`.

Резервное копирование журнала транзакций

Следующий вариант оператора `BACKUP` позволяет создать резервную копию журнала транзакций. В общем виде он выглядит, как показано в листинге 4.5.

Листинг 4.5. Общий вид команды резервного копирования журнала транзакций

```
BACKUP LOG { database_name | @database_name_var }
{
    TO < backup_device > [ ,...n ]
    [ WITH
        [ BLOCKSIZE = { blocksize | @blocksize_variable } ]
        [ [ , ] DESCRIPTION = { 'text' | @text_variable } ]
        [ [ ,] EXPIREDATE = { date | @date_var }
            | RETAINDAYS = { days | @days_var } ]
        [ [ , ] PASSWORD = { password | @password_variable } ]
        [ [ , ] FORMAT | NOFORMAT ]
        [ [ , ] { INIT | NOINIT } ]
        [ [ , ] MEDIADESCRIPTION = { 'text' | @text_variable } ]
        [ [ , ] MEDIANAME = { media_name | @media_name_variable } ]
        [ [ , ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable
} ]
        [ [ , ] NAME = { backup_set_name | @backup_set_name_var } ]
        [ [ , ] NO_TRUNCATE ]
        [ [ , ] { NORECOVERY | STANDBY = undo_file_name } ]
        [ [ , ] { NOREWIND | REWIND } ]
```

```
        [ [ , ] { NOSKIP | SKIP } ]
        [ [ , ] { NOUNLOAD | UNLOAD } ]
        [ [ , ] RESTART ]
        [ [ , ] STATS [ = percentage ] ]
    ]
}
```

Тут уже больше отличий от первой команды `BACKUP DATABASE`. Хотя большинство параметров повторяется, есть и новые параметры, которые мы обсудим чуть позже. Сейчас рассмотрим простейший пример использования команды:

```
BACKUP LOG TestBatabase
TO DISK='c:\Temp\BackGroup_Log.bak'
```

Теперь рассмотрим новые параметры, которые мы еще не обсуждали:

- `NO_TRUNCATE` — если база данных разрушена, то резервное копирование, скорее всего, невозможно. С помощью этого параметра можно сохранить журнал в любом состоянии базы данных;
- `NORECOVERY` или `STANDBY` — если используется параметр `STANDBY`, то создается резервная копия журнала, а база данных остается в режиме "только для чтения". В параметре можно указать файл, в котором будет сохранена информация о восстановлении, и с помощью этого файла можно будет произвести откат изменений. Если указан параметр `NORECOVERY`, то будет создана копия журнала, и база останется в состоянии восстановления. Этот параметр чаще всего используется, когда нужно зарезервировать последние данные журнала в случае сбоя базы данных. По этим данным журнала вы сможете впоследствии восстановить базу данных.

Остальные параметры нам уже знакомы, и их можно использовать так же, как и с остальными командами резервного копирования.

Усечение журнала транзакций

Журналы транзакций могут очень быстро расти, особенно, если происходит массовая загрузка данных. Однажды я заполнял базу данными объемом в 300 Мбайт. С первого раза загрузить их не удалось, поэтому сначала вставлялись записи с основными данными, а потом происходило несколько запросов на обновление, которые устанавливали связи. После окончания работы оказалось, что размер журнала превысил 1 Гбайт. Оставлять журнал в таком состоянии просто не имеет смысла, ведь его даже резервировать дольше, чем сами данные. Вполне логично было бы выполнить усечение журнала после таких изменений.

Для усечения журнала используется оператор BACKUP следующего вида:

```
BACKUP LOG { database_name | @database_name_var }
{
    [ WITH
        { NO_LOG | TRUNCATE_ONLY } ]
}
```

В этом варианте команды можно указать только один из двух параметров: NO_LOG или TRUNCATE_ONLY. Какой из них выбрать? Безразлично — это синонимы. Я больше привык писать TRUNCATE_ONLY. Таким образом, усечение журнала базы данных TestDatabase будет выглядеть следующим образом:

```
BACKUP LOG TestBatabase
WITH TRUNCATE_ONLY
```

Пример резервного копирования базы данных с журналами

Рассмотрим пример, в котором стратегия резервного копирования состоит в резервировании базы данных и журнала транзакций. Допустим, что вы хотите каждую ночь выполнять резервное копирование базы данных. Это можно делать следующим сценарием:

```
BACKUP DATABASE ИмяБазы
TO  DISK = 'Путь к файлу'
WITH  INIT ,  NOUNLOAD ,  NOSKIP,  STATS = 10,  NOFORMAT
```

Параметры NOFORMAT и NOSKIP необходимы, если вы хотите иметь возможность восстановить базу данных, начиная с последней имеющейся полной резервной копии. Например, у вас есть предпоследняя полная резервная копия и все последующие журналы транзакций. Восстановите полную копию, и после этого восстанавливайте все созданные с этого момента журналы.

Для создания резервной копии журнала используйте команду:

```
BACKUP LOG ИмяБазы
TO  DISK = 'Путь к файлу'
WITH  INIT, NOUNLOAD,
  NOSKIP, STATS = 10,  NOFORMAT
```

Такую схему я применял при работе с удаленными офисами. Раз в месяц мне привозили полную копию, а в конце каждого дня я получал через Интернет журнал транзакций, который содержал изменения за день. Помимо этого, в удаленном офисе ежедневно в конце рабочего дня создавалась полная резервная копия. Эти копии я не имел, но благодаря сочетанию параметров NOFORMAT и NOSKIP я мог восстанавливать журнал и имел последнюю копию

данных офиса, просто мое восстановление шло, начиная с последней имеющейся полной.

Если пропустить восстановление хотя бы одного журнала, то последующее восстановление станет невозможным. Поэтому, чтобы одна копия журнала не перезаписала другую (которую я не восстановил у себя), в качестве имени файла резервной копии устанавливалась дата и время (часы, минуты и секунды) создания. За одну секунду нельзя создать две копии (пользователь просто не успеет выбрать дважды одну и ту же команду из меню, а сервер не успеет отработать все действия), поэтому каждая копия будет сохраняться в своем файле. При наличии всех файлов журнала не составляет труда их восстановить.

Примечание

Примеры запросов из этой главы можно найти в файле \Chapter4\backup.sql на компакт-диске, прилагаемом к книге.

4.10.4. Восстановление данных

Теперь подробнее поговорим о восстановлении данных. К восстановлению прибегают, в основном, в двух случаях: при необходимости перенести базу данных на другой сервер и при восстановлении утерянных или разрушенных данных. Если вы столкнулись со вторым случаем, т. е. данные разрушены, то я рекомендую прежде всего отключить все задания, которые выполняются по расписанию и при этом работают с разрушенной базой данных. Задание может содержать команды, которые вызовут лишние проблемы при восстановлении.

После отключения заданий проанализируйте оставшиеся данные. Возможно, есть еще какая-нибудь информация, которую можно сохранить и впоследствии восстановить. Желательно также узнать причину разрушения базы. Чем лучше вы разберетесь в происшедшем, тем более качественно сможете восстановить данные и избежать подобных ситуаций в будущем.

Для восстановления данных используется оператор `RESTORE DATABASE`, который также имеет несколько вариантов:

- восстановление целой базы данных;
- восстановление части базы данных;
- восстановление определенных файлов и файловых групп;
- восстановление журнала транзакций.

Рассмотрим каждый из этих вариантов.

Восстановление целой базы данных

Общий вид команды восстановления целой базы данных показан в листинге 4.6.

Листинг 4.6. Команда восстановления базы данных

```
RESTORE DATABASE { database_name | @database_name_var }
[ FROM < backup_device > [ ,...n ] ]
[ WITH
    [ RESTRICTED_USER ]
    [ [ , ] FILE = { file_number | @file_number } ]
    [ [ , ] PASSWORD = { password | @password_variable } ]
    [ [ , ] MEDIANAME = { media_name | @media_name_variable } ]
    [ [ , ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
    [ [ , ] MOVE 'logical_file_name' TO 'operating_system_file_name' ]
            [ ,...n ]
    [ [ , ] KEEP_REPLICATION ]
    [ [ , ] { NORECOVERY | RECOVERY | STANDBY = undo_file_name } ]
    [ [ , ] { NOREWIND | REWIND } ]
    [ [ , ] { NOUNLOAD | UNLOAD } ]
    [ [ , ] REPLACE ]
    [ [ , ] RESTART ]
    [ [ , ] STATS [ = percentage ] ]
]
```

Простейшая команда восстановления полной копии базы данных выглядит следующим образом:

```
RESTORE DATABASE FlenovSQLBook
FROM DISK='c:\Temp\test1.bak'
```

В первой строчке после `RESTORE DATABASE` мы указываем имя восстанавливаемой базы. Во второй строчке пишем `FROM` и устройство. В данном случае в качестве устройства выступает файл, поэтому указываем параметр `DISK` и полный путь к файлу. Таким образом, можно восстановить только полную резервную копию, например, созданную следующим запросом:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
```

Во время восстановления желательно быть подключенным к базе данных master, и при этом восстанавливаемая база должна быть свободна, т. е. к ней

никто не должен быть подключен. Иначе запрос завершит выполнение ошибкой.

Большинство параметров команды `RESTORE DATABASE` нам уже знакомы по оператору `BACKUP DATABASE` и имеют тот же смысл, потому что эти команды направлены на выполнение одной задачи. Допустим, что во время создания резервной копии вы указали пароль, значит, во время восстановления должен быть указан именно этот пароль. Например, следующая резервная копия создается защищенной паролем:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\test1.bak'
WITH PASSWORD='qwerty', INIT
```

Если при восстановлении не указать пароль, то произойдет ошибка. Пароль можно указать в качестве параметра следующим образом:

```
RESTORE DATABASE FlenovSQLBook
FROM DISK='c:\Temp\test1.bak'
WITH PASSWORD='qwerty'
```

Рассмотрим новые параметры, которые мы не видели при рассмотрении оператора `BACKUP DATABASE`:

- `MOVE` — позволяет изменить путь к восстанавливаемому файлу. Во время создания резервной копии в файл попадает и полный путь. Допустим, что на вашем сервере база данных физически находится в файле e:\data\database.mdf. Вы переносите базу на новый сервер, где диска `e`: нет. В этом случае при восстановлении произойдет ошибка, потому что по умолчанию сервер будет пытаться восстановить базу в файле с тем же расположением, т. е. e:\data\database.mdf. Для изменения пути как раз используется оператор `MOVE`;
- `KEEP_REPLICATION` — позволяет восстановить информацию о репликации. Если у вас в репликации используются удаленные серверы, которые не связаны по сети, и база данных переносится через резервное копирование/восстановление, то необходимо включать эту опцию. Если не указать ее, то после восстановления информация о настройках репликации исчезнет. Есть еще один способ перенести базу с одного сервера на другой — отключить (`Detach`), перенести файлы на другой сервер и подключить (`Attach`). Подключение мы рассматривали в *разд. 1.1.2*;
- `NORECOVERY` — восстановить базу данных, при этом база останется недоступной, но можно будет восстанавливать дополнительные журналы транзакций. Используйте эту опцию, если вы восстанавливаете полную копию, и после этого требуется еще восстановить дополнительные журналы. По умолчанию используется параметр `RECOVERY`;

- `RECOVERY` — восстановить базу данных и сделать ее доступной для пользователей. Дополнительные журналы транзакций, которые были созданы после этой копии, восстановить не удастся, и эти изменения будут утеряны;
- `STANDBY` — восстановить базу данных, но оставить ее в режиме "только для чтения" и разрешить восстанавливать дополнительные журналы. В отличие от `NORECOVERY`, пользователи смогут подключаться к базе данных, но смогут только читать. Изменение будет недоступно. Основываясь на личном опыте, могу сказать, что опция удобна в следующих случаях:
  - если необходимо максимально быстро восстановить базу, пусть даже она будет недоступна для изменений;
  - есть несколько удаленных офисов со своими базами данных. Чтобы в главном офисе были доступны базы данных других офисов, можно каждый день собирать журналы из удаленных офисов и восстанавливать их в главном, оставляя в режиме "только для чтения", что позволит восстановить журнал на следующий день. Пользователи главного офиса смогут вести мониторинг происходящих событий. Если нужно вносить изменения в данные на любом из серверов, то тут уже журналы транзакций не помогут. Обратите свое внимание на репликацию;
- `REPLACE` — требует, чтобы база данных была создана заново, даже если с таким именем уже существует другая база;
- `FILE` — с помощью этого параметра можно указать, какую именно копию нужно восстановить из файла. Как мы уже говорили, в один и тот же файл может быть записано несколько копий, и каждая из них будет иметь свой номер, начиная с 1.

Теперь рассмотрим несколько примеров, чтобы увидеть различные параметры в работе. Давайте попробуем удалить базу и восстановить ее:

```
-- Удаление базы
DROP DATABASE FlenovSQLBook

-- Восстановление базы
RESTORE DATABASE FlenovSQLBook
FROM DISK='c:\Temp\test1.bak'
```

Обратите внимание, что файлы базы данных восстанавливаются в том же месте. Чтобы изменить это, используем оператор `MOVE`:

```
RESTORE DATABASE FlenovSQLBook
FROM DISK='c:\Temp\test1.bak'
WITH MOVE 'FlenovSQLBook' TO 'E:\Test.mdf',
     MOVE 'FlenovSQLBook_Log' TO 'E:\Test.ldf'
```

После MOVE мы указываем логическое имя файла, а после ключевого слова TO идет новый путь к файлу.

Когда еще нужен оператор MOVE? Допустим, что у вас есть рабочая база данных на одном сервере и тестовая на другом, и вы хотите копию рабочей восстановить на тестовой базе. Если пути файлов не совпадают, то при восстановлении вы должны с помощью MOVE указать такой путь, как у уже существующей базы. Если путь из резервной копии не будет совпадать с путем базы данных, то произойдет ошибка, исправить которую можно только с помощью указания правильного пути в MOVE.

Я не думаю, что вы помните все логические имена своих файлов, которые создавались для баз данных. Тем более, вы можете не знать эти имена для тех резервных копий, которые созданы не вами. Чтобы узнать параметры файла резервной копии, воспользуйтесь следующей командой:

```
RESTORE FILELISTONLY
FROM DISK='c:\Temp\test1.bak'
```

В результате будет показана таблица из следующих полей:

- LogicalName — логическое имя файла;
- PhysicalName — физическое расположение файла, куда будет происходить восстановление, если не указать другого пути с помощью оператора MOVE;
- Type — тип файла. Если здесь будет буква D (Database, база данных), то это файл данных; если буква L (Log, журнал), то это файл журнала транзакций;
- FileGroupName — имя файловой группы;
- Size — размер файла;
- MaxSize — максимальный размер.

Теперь посмотрим, как можно восстановить сначала полную резервную копию, а затем дифференцированную:

```
-- Восстановление полной копии
RESTORE DATABASE MyNwind
   FROM DISK='c:\Temp\test1.bak'
   WITH NORECOVERY

-- Восстановление дифференцированной копии
RESTORE DATABASE MyNwind
   FROM DISK='c:\Temp\test1.bak'
   WITH FILE = 2
```

Первый запрос восстанавливает полную копию, и при этом база данных остается недоступной (указан оператор `NORECOVERY`). В файле находятся две резервные копии: первая полная, а вторая дифференцированная. В первом запросе мы не указали, какая из них нас интересует, поэтому по умолчанию будет использоваться вторая.

При выполнении второго запроса не надо указывать, что это дифференцированная копия. Но нам нужна вторая копия из файла, поэтому явно указываем это через параметр `FILE`.

Теперь посмотрим, как можно перенести базу данных с одного сервера на другой с помощью операций резервного копирования и восстановления. Сначала необходимо создать полную резервную копию на сервере, с которого мы хотим скопировать базу:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Temp\move.bak'
```

Теперь эту резервную копию переносим на нужный компьютер и восстанавливаем базу данных:

```
RESTORE DATABASE NewDatabase
FROM DISK='c:\Temp\move.bak'
WITH
 MOVE 'FlenovSQLBook' TO 'c:\Temp\NewDatabase.mdf',
 MOVE 'FlenovSQLBook_Log' TO 'c:\Temp\NewDatabase.ldf'
```

Обратите внимание, что после оператора `RESTORE DATABASE` указано другое имя базы данных. Дело в том, что необязательно указывать такое же имя, как и у базы данных, которую сохраняли в файле.

Я производил резервное копирование и восстановление на одном и том же сервере, поэтому мне пришлось изменять имя базы, потому что FlenovSQLBook у меня уже существует. Чтобы не было конфликтов файлов с существующей базой, мне пришлось использовать оператор `MOVE`, чтобы указать новое расположение файлов на диске.

Восстановление части базы данных

Синтаксис команды восстановления показан в листинге 4.7. Попробуйте найте 10 отличий от команды полного восстановления? Я думаю, что вы выдохнетесь где-то на пяти, хотя сам я и не пытался.

Листинг 4.7. Восстановление части базы данных

```
RESTORE DATABASE { database_name | @database_name_var }
    < file_or_filegroup > [ ,...n ]
[ FROM < backup_device > [ ,...n ] ]
[ WITH
    { PARTIAL }
    [ [ , ] FILE = { file_number | @file_number } ]
    [ [ , ] PASSWORD = { password | @password_variable } ]
    [ [ , ] MEDIANAME = { media_name | @media_name_variable } ]
    [ [ , ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
    [ [ , ] MOVE 'logical_file_name' TO 'operating_system_file_name' ]
            [ ,...n ]
    [ [ , ] NORECOVERY ]
    [ [ , ] { NOREWIND | REWIND } ]
    [ [ , ] { NOUNLOAD | UNLOAD } ]
    [ [ , ] REPLACE ]
    [ [ , ] RESTRICTED_USER ]
    [ [ , ] RESTART ]
    [ [ , ] STATS [= percentage ] ]
]
```

Иногда случаются ошибки пользователей, или встречаются приложения, нарушающие определенные таблицы. Для поддержки восстановления данных после таких ошибок MS SQL Server позволяет восстановить часть базы данных в новое место, после чего разрушенные или удаленные данные могут быть скопированы в рабочую базу.

Допустим, что пользователь, администратор или программист случайно удалил одну таблицу во время тестирования новой возможности программы. Восстанавливать всю базу данных не имеет смысла, потому что будут перезаписаны все объекты, которые могут иметь более новую версию. Намного эффективнее восстановить только часть базы, которая содержит эту таблицу.

Для частичного восстановления используются файловые группы. Первичную (`PRIMARY`) группу восстанавливать обязательно, а вот пользовательские — не обязательно. Так как первичная восстанавливается в любом случае, и в ней хранится информация обо всех объектах, то и в базе будут созданы все объекты. Другое дело, не все объекты могут быть заполнены. Мы можем восстановить только те файловые группы, которые содержат необходимые данные (разрушенные или уничтоженные).

Допустим, что у нас есть база данных с тремя файловыми группами `group1`, `group2` и `group3`, и нам необходимо восстановить таблицу из первой группы. Пример создания такой таблицы показан в листинге 4.8.

Листинг 4.8. Создание базы данных с тремя группами

```
-- Создаем базу данных
CREATE DATABASE TestDatabase
GO

-- Добавляем две файловые группы
ALTER DATABASE TestDatabase
  ADD FILEGROUP group1

ALTER DATABASE TestDatabase
  ADD FILEGROUP group2

ALTER DATABASE TestDatabase
  ADD FILEGROUP group3
GO

-- Добавляем в каждую группу по файлу
ALTER DATABASE TestDatabase
 ADD FILE
 (
  NAME='data1',
  FILENAME='c:\Temp\data1.dat'
 )
 TO FILEGROUP group1

ALTER DATABASE TestDatabase
 ADD FILE
 (
  NAME='data2',
  FILENAME='c:\Temp\data2.dat'
 )
 TO FILEGROUP group2
GO

-- Создаем файл для третьей группы
```

```
...
...

-- Переходим в таблице TestDatabase
USE TestDatabase

-- Создаем тестовую таблицу
CREATE TABLE TestTable
(
   Famil varchar(10),
   Name varchar(10) collate latin1_general_cs_as
) ON group1

-- Создание других таблиц
...
...
```

Теперь необходимо создать полную резервную копию:

```
BACKUP DATABASE TestDatabase
TO DISK = 'c:\Temp\part.bak'
```

Выполните команду `RESTORE FILELISTONLY`, чтобы определить, какие имена файлов созданы для основной файловой группы и для пользовательских:

```
RESTORE FILELISTONLY
FROM DISK='c:\Temp\part.bak'
```

Это необходимо, потому что мы будем восстанавливать базу данных на том же сервере, а чтобы не было конфликтов имен, переместим файлы в новое место с помощью опции `MOVE`.

Восстанавливать базу данных нужно также под новым именем или на другом сервере, чтобы не испортить существующую базу. Восстановив, можно будет нужную таблицу скопировать в рабочую базу данных. Итак, восстановим файловую группу 1 в новой базе данных с именем PartData:

```
RESTORE DATABASE PartData
FILEGROUP = 'group1'
FROM DISK= 'c:\Temp\part.bak'
WITH PARTIAL,
  MOVE 'TestDatabase' TO 'c:\Temp\part.mdf',
  MOVE 'TestDatabase_log' TO 'c:\Temp\part.log',
  MOVE 'data1' TO 'c:\Temp\part_data1.dat'
```

Во второй строчке указан параметр `FILEGROUP`, которому присваивается имя восстанавливаемой группы. После `WITH` обязательно нужно указать параметр `PARTIAL`. С помощью `MOVE` мы указываем новое расположение файлов. Обратите внимание, что перемещаем мы только основной файл, журнал и файл группы `group1`. Остальные файлы групп не нужно указывать, потому что они не восстанавливаются.

Теперь данные из таблицы `TestTable` базы данных PartData можно перенести в рабочую базу с помощью следующего запроса:

```
INSERT INTO TestDatabase..TestTable
SELECT * FROM PartData..TestTable
```

Восстановление файлов и файловых групп

Для востановления файлов и файловых групп используется команда `RESTORE`, общий вид которой показан в листинге 4.9.

Листинг 4.9. Команда восстановления файлов и файловых групп

```
RESTORE DATABASE { database_name | @database_name_var }
    < file_or_filegroup > [ ,...n ]
[ FROM < backup_device > [ ,...n ] ]
[ WITH
    [ RESTRICTED_USER ]
    [ [ , ] FILE = { file_number | @file_number } ]
    [ [ , ] PASSWORD = { password | @password_variable } ]
    [ [ , ] MEDIANAME = { media_name | @media_name_variable } ]
    [ [ , ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
    [ [ , ] MOVE 'logical_file_name' TO 'operating_system_file_name' ]
            [ ,...n ]
    [ [ , ] NORECOVERY ]
    [ [ , ] { NOREWIND | REWIND } ]
    [ [ , ] { NOUNLOAD | UNLOAD } ]
    [ [ , ] REPLACE ]
    [ [ , ] RESTART ]
    [ [ , ] STATS [ = percentage ] ]
]
```

Обратите внимание, что можно указывать только `NORECOVERY`. Опции `RECOVERY` и `STANDBY` при этом отсутствуют. В остальном оператор похож на

уже знакомые. Следующий пример показывает, как можно восстановить файловые группы:

```
RESTORE DATABASE PartData
FILEGROUP = 'group1'
FILEGROUP = 'group2'
FILE = 'file1'
FROM DISK= 'c:\Temp\part.bak'
WITH NORECOVERY
```

Восстановление журналов транзакций

С восстановлением журналов дела обстоят гораздо интереснее. Для восстановления журналов транзакций используется команда `RESTORE LOG`, общий вид которой показан в листинге 4.10.

Листинг 4.10. Восстановление журнала транзакций

```
RESTORE LOG { database_name | @database_name_var }
[ FROM < backup_device > [ ,...n ] ]
[ WITH
    [ RESTRICTED_USER ]
    [ [ , ] FILE = { file_number | @file_number } ]
    [ [ , ] PASSWORD = { password | @password_variable } ]
    [ [ , ] MOVE 'logical_file_name' TO 'operating_system_file_name' ]
            [ ,...n ]
    [ [ , ] MEDIANAME = { media_name | @media_name_variable } ]
    [ [ , ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
    [ [ , ] KEEP_REPLICATION ]
    [ [ , ] { NORECOVERY | RECOVERY | STANDBY = undo_file_name } ]
    [ [ , ] { NOREWIND | REWIND } ]
    [ [ , ] { NOUNLOAD | UNLOAD } ]
    [ [ , ] RESTART ]
    [ [ , ] STATS [= percentage ] ]
    [ [ , ] STOPAT = { date_time | @date_time_var }
        | [ , ] STOPATMARK = 'mark_name' [ AFTER datetime ]
        | [ , ] STOPBEFOREMARK = 'mark_name' [ AFTER datetime ]
    ]
]
```

Здесь снова доступны параметры `NORECOVERY`, `RECOVERY` и `STANDBY`. Восстанавливая журналы транзакций, вы должны оставлять их открытыми, и только при восстановлении последнего журнала нужно указать параметр `RECOVERY`, чтобы база данных стала доступной для пользователей. Если вы указали эту опцию раньше времени, то единственный способ добавить невосстановленные журналы — начать процесс восстановления журналов.

Во время восстановления базы данных вы должны учитывать, что все журналы должны восстанавливаться в том же порядке, в котором и резервировались. Определить порядок помогают даты создания файлов. Но если файлы передаются по сети, то даты нарушаются, и в этом случае будет лучше, если имя файла будет отображать дату его создания. Эта рекомендация основана на личном опыте.

Самая простейшая команда восстановления журнала выглядит следующим образом:

```
RESTORE LOG TestDatabase
FROM DISK = 'c:\temp\log_data.bak'
```

Очень интересной является возможность восстановления базы по состоянию на указанную дату, что достигается с помощью журнала транзакций. Для этого используется один из следующих параметров:

- `STOPAT дата_время` — в качестве параметра передаются дата и время, по состоянию на которые нужно восстановить базу данных. Для удобства в качестве параметра можно передавать переменную. В результате восстановлению подлежат только записи журнала транзакций, сделанные до указанного момента времени. Все изменения, которые были зафиксированы позже, восстанавливаться не будут;
- `STOPATMARK имя_метки` — восстановить журнал до указанной метки. Будут восстановлены все записи, которые были записаны до указанной метки, плюс сама запись с меткой;
- `STOPBEFOREMARK имя_метки` — восстановить все записи до указанной метки. Сама транзакция с меткой восстанавливаться не будет.

Следующий пример показывает, как восстановить базу, в которой будет отражено состояние данных до 1 января 2005 года:

```
RESTORE LOG FlenovSQLBook
FROM DISK='c:\Temp\test_log.bak'
WITH STOPAT = '01.01.2005 00:00AM'
```

Рассмотрим пример восстановления базы в соответствии с определенной меткой в транзакции. Для этого я написал несколько команд, которые можно

увидеть в листинге 4.11. Если вы хотите протестировать этот пример, то у вас уже должна существовать полная резервная копия базы данных.

Листинг 4.11. Пример восстановления журнала по состоянию на определенный момент

```
-- Начало транзакции
BEGIN TRANSACTION IvanUpdate
  WITH MARK 'Большое обновление данных'
UPDATE tbPeoples
   SET vcName = 'Андрей'
   WHERE vcName = 'Иван'
COMMIT TRANSACTION IvanUpdate

-- Здесь могут быть другие изменения

USE Master
GO

-- Резервное копирование
BACKUP LOG FlenovSQLBook
TO DISK='c:\Temp\test_db.bak'

-- Восстанавливаем базу
RESTORE DATABASE FlenovSQLBook
FROM DISK='c:\Temp\test_log.bak'
WITH NORECOVERY

-- Восстанавливаем журнал по определенной
-- метке
RESTORE LOG FlenovSQLBook
FROM DISK='c:\Temp\test_log.bak'
WITH STOPATMARK = 'IvanUpdate'
```

Первый сценарий запускает именованную транзакцию с именем `IvanUpdate`, в которой происходит обновление таблицы `tbPeoples`. После этого могут идти другие команды обновления данных.

Теперь переходим в базу данных master, потому что нельзя восстанавливать базу, если к ней есть подключения, в том числе и наше. Конечно же, до восстановления необходимо создать резервную копию журнала.

Для восстановления сначала необходимо выполнить восстановление полной базы, а потом уже журнала. При выполнении оператора `RESTORE DATABASE` для базы мы должны указать параметр `NORECOVERY`, чтобы можно было восстанавливать дополнительные журналы. При восстановлении журнала нужно использовать параметр `STOPATMARK` и присвоить ему имя транзакции, на которой нужно остановиться. Изменения после этой транзакции не будут восстановлены.

Возможность восстановления на определенный момент времени действительно мощная, потому что она позволяет откатиться до того момента, когда данные были утеряны. Допустим, что пользователь в определенный момент случайно удалил какие-то данные (а может, не случайно, и не пользователь, а хакер). Вы просто выполняете откат до момента `X` и восстанавливаете корректность данных.

Самое главное — вовремя заметить потерю данных и успеть произвести откат. После момента `X` могут произойти важные изменения, потеря которых может оказаться катастрофической, и вам придется задуматься — потерять эти изменения, но восстановить данные или сохранить важные изменения, но потерять данные. Если вы пропустили момент, то можно восстановить данные на определенный момент в отдельной базе данных (временной) и из нее уже перенести потерянные данные в рабочую базу.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\restore.sql на компакт-диске, прилагаемом к книге.

4.10.5. Замечания по резервному копированию

От утраты данных никто не застрахован, а они регулярно теряются, и причин этого много: выход оборудования из строя, воровство техники, стихийные бедствия, пожары, терроризм. Конечно же, вероятность возникновения некоторых из них слишком мала, но она есть. А если сложить все вместе, то общая вероятность потери данных становится достаточно высокой, чтобы задуматься об этом. Особенно, если добавить ко всем описанным бедам еще и хакеров, которые также могут уничтожить данные, если проникнут на сервер.

Так почему же большинство из нас не задумывается о возможных проблемах, пока не встретится с бедой? Виновником как всегда является простая человеческая лень, с которой надо бороться.

Необходимо делать все, чтобы данные ни в коем случае не были утеряны бесследно и затраты на восстановление были минимальны.

Планируя резервное копирование, необходимо четко представлять себе размер резервной копии. Если для хранения копий базы данных вы выбрали сменный носитель, то убедитесь, что его емкость достаточна для хранения необходимых данных. Если резервная копия не поместится на носитель, то операция завершится ошибкой, и вы потеряете возможность восстановления этой копии.

Если для резервного копирования вы используете задание, выполняемое в определенное время, то это не значит, что процесс не нужно контролировать. Существует множество возможных ошибок, которые могут нарушить процесс резервного копирования. Да, задание позволяет избавиться от лишних проблем по ручному выполнению команд, но контролировать систему все же нужно.

Если ваша база данных слишком велика, и резервная копия не помещается на доступные вам носители информации, не стоит создавать копию на том же компьютере (и тем более на том же диске), что и работающая база. Как показывает практика, причиной потери данных чаще всего становятся именно жесткие диски, а если диск выйдет из строя, то вы потеряете не только рабочую базу, но и резервную копию, и польза от такой стратегии равна нулю. Поэтому обязательно копируйте резервную копию на съемные носители.

У MS SQL Server есть один недостаток — во время создания резервной копии не происходит никакого сжатия. Видимо, это сделано для повышения производительности и максимального сокращения времени на копирование и восстановление. Даже самый быстрый алгоритм сжатия отнимет лишние миллисекунды у сервера, а это может оказаться неприемлемым. С другой стороны, не помешал бы дополнительный ключ, включающий сжатие, потому что базы данных сжимаются очень хорошо, особенно, если вы слишком расточительно выделяли место для хранения значений полей.

Если вы выделили для хранения строки поле размером в 250 символов типа `char`, а реально храните там не более 10 символов, то для каждой строки в файле данных будет по 240 символов пробела, а это и простейший архиватор может сжать в несколько раз. Но даже если выделять пространство по минимуму, четко спланировать размер данных не удастся, да и некоторые поля могут оставаться пустыми, а в файле данных для них будет выделяться пространство, которое очень хорошо сожмется архиватором.

Для примера я взял одну из своих рабочих баз данных размером в 360 Мбайт и упаковал ее простым архиватором ZIP (встроенным в программу Windows Commander) с настройками по умолчанию. В результате получился архив размером в 180 Мбайт. Я достиг сжатия примерно в два раза, притом, что в базе почти все поля являются обязательными для заполнения, а размер полей

выбирался минимальным. А если перед резервным копированием сократить размер базы данных *(см. разд. 4.11)*, то размер резервной копии может быть еще меньше. Уменьшение может оказаться полезным, если в вашей базе данных очень часто происходит вставка и удаление данных.

Архивация — не такой уж и сложный процесс, и его также можно возложить на задание, которое выполняет у вас резервное копирование. Таким образом, задание резервного копирования может состоять из следующих шагов:

1. Уменьшение размера базы.
2. Создание резервной копии базы данных.
3. Архивирование данных на локальный диск.
4. Копирование на съемный носитель.

Обратите внимание, что архивирование данных происходит на локальный диск. Дело в том, что большинство съемных носителей работает намного медленнее жесткого диска и создает излишнюю нагрузку на центральный процессор. Лучше будет заархивировать данные на локальном диске и потом скопировать небольшой объем информации на съемный носитель. В большинстве случаев это позволит сэкономить время резервного копирования.

Никогда не архивируйте сразу на съемный носитель архиватором ZIP. Большинство его реализаций создают временный файл на носителе — получателе данных и используют его для хранения временных данных. Во-первых, это займет дополнительное место на носителе, тогда его может не хватить для выполнения архивирования. Во-вторых, как мы уже знаем, носитель может быть намного медленнее жесткого диска и обращение к временному файлу на таком носителе создаст излишнюю нагрузку на процессор и увеличится время резервного копирования.

4.11. Уменьшение базы данных

На этапе проектирования очень сложно рассчитать необходимое базе данных пространство, и бывают случаи, когда выделяется слишком много места под одну базу и слишком мало под другую. Если еще остается достаточно свободного места, то это не проблема. А если оно заканчивается? Слишком большую базу данных необходимо уменьшить, но с помощью `ALTER TABLE` можно только увеличивать, уменьшение запрещено!

В *разд. 4.10* мы говорили о том, что для уменьшения размера журнала необходимо выполнить команду `RESTORE LOG` с оператором `TRUNCATE ONLY`. Размер базы данных при резервном копировании не уменьшается, тут нужна другая команда.

Вы можете сократить размер базы данных с помощью оператора SHRINKDATABASE. Если у базы несколько файлов, то он уменьшит все файлы (файлы журнала уменьшаются другой командой).

Общий вид команды SHRINKDATABASE:

```
DBCC SHRINKDATABASE
    ( database_name [ , target_percent ]
        [ , { NOTRUNCATE | TRUNCATEONLY } ]
    )
```

Рассмотрим параметры более подробно:

- target_percent — указывает требуемый процент свободного пространства, оставшегося в базе данных после операции усечения;
- NOTRUNCATE — указывает на необходимость освобождения свободного пространства так, чтобы оно осталось доступным базе данных. В этот момент очищаются помеченные на удаление данные, но физически файл базы не уменьшается;
- TRANCATEONLY — освобождение свободного пространства и оптимизация размера файла без перемещения каких-либо данных внутри файла. В этом случае SQL Server игнорирует параметр target_percent.

Например:

```
DBCC SHRINKDATABASE (FlenovSQLBook , 25)
```

Этот пример уменьшает размер базы данных FlenovSQLBook так, что файл будет иметь 25% свободного пространства. В этом примере, если база данных содержит 6 Мбайт данных, а размер файла 8 Мбайт, то после усечения размер будет уменьшен до 6 Мбайт.

Сервер SQL не может сделать размер файла меньше необходимого для хранения существующих данных, потому что в момент очистки пространства данные не удаляются. Помимо этого, размер базы данных не может быть сделан меньшим, чем тот, что был указан при ее создании. Иногда даже лучше во время создания базы указать минимальный размер и сразу после этого увеличить его до необходимого во время работы.

При уменьшении базы, состоящей из нескольких файлов, сервер сокращает их размеры по возможности пропорционально. Но можно уменьшить и конкретные файлы базы данных. Для этого используется оператор SHRINKFILE. Общий вид команды:

```
DBCC SHRINKFILE
    ( { file_name | file_id }
        { [ , target_size ]
```

```
            | [ , { EMPTYFILE | NOTRUNCATE | TRUNCATEONLY } ]
        }
    )
```

Рассмотрим параметры этой команды:

- `target_size` — указывает требуемый размер файла данных в мегабайтах. Если не указано, то размер файла будет уменьшен насколько это возможно;
- `EMPTYFILE` — перемещает все данные из указанного файла в другие файлы той же файловой группы. Используйте эту опцию перед удалением файла, чтобы сначала очистить его переносом данных в другие файлы, а потом удалить.

Следующий пример уменьшает размер файла данных FlenovSQLBook до 10 Мбайт:

```
DBCC SHRINKFILE (FlenovSQLBook, 10)
```

Вы можете уменьшать файл автоматически. По умолчанию эта опция отключена. С использованием опции базы данных `autoshrink` вы можете установить опцию восстановления пространства автоматически.

При использовании уменьшения размера вы должны учитывать, что:

- результирующая база данных должна быть больше базы данных model или существующих данных в базе данных или файле;
- `SHRINKFILE` позволит уменьшить базу данных до размера менее указанного при ее создании, но не менее необходимого для хранения содержащихся данных;
- `SHRINKDATABASE` и `SHRINKFILE` выполняют некоторые операции по-разному, так что вы можете не заметить уменьшения базы данных или файла;
- операция уменьшения базы данных достаточно трудоемка для сервера, поэтому старайтесь правильно планировать размер файлов для данных, чтобы не пришлось их уменьшать.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\shrink.sql на компакт-диске, прилагаемом к книге.

4.12. Отключение базы данных

В *разд. 1.1.2* мы уже познакомились с подключением/отключением базы данных. Для отключения используется процедура `sp_detach_db`, но мы ее рассмотрели бегло, потому что нам не хватало знаний о процедурах. Сейчас мы восполним этот пробел.

Процедура sp_detach_db в общем виде выглядит следующим образом:

```
sp_detach_db [ @dbname = ] 'dbname'
    [ , [ @skipchecks = ] 'skipchecks' ]
```

Здесь два параметра, первый из которых является обязательным:

- @dbname — имя базы данных, которую необходимо отключить;
- @skipchecks — если этот параметр равен false, то перед отключением будет выполнен оператор обновления статистики UPDATE STATISTICS, иначе эта операция будет пропущена.

Давайте создадим тестовую базу, на примере которой проверим отключение и подключение:

```
CREATE DATABASE TestDatabase
ON
 (
   NAME = TestDatabase_data,
   FILENAME = 'c:\data\test.mdf',
   SIZE = 10MB,
   MAXSIZE = 100GB,
   FILEGROWTH = 10%
 )
```

Теперь выполним отключение следующей командой:

```
EXEC sp_detach_db 'TestDatabase', 'true'
```

В данном случае второй параметр равен true, а значит, при отключении статистика обновляться не будет.

Чтобы снова подключить эту базу, можно воспользоваться одним из следующих способов:

- выполнить оператор создания базы данных CREATE DATABASE с параметром FOR ATTACH;
- воспользоваться процедурой sp_attach_db.

Рассмотрим сначала метод с использованием CREATE DATABASE, ведь эта команда нам уже знакома:

```
CREATE DATABASE Archive
ON PRIMARY
 (
   FILENAME = 'c:\data\test.mdf'
 )
FOR ATTACH
```

Напоминаю, что имя подключаемой базы может отличаться от имени базы, которую мы отключали. Раньше имя было TestDatabase, а после подключения оно превратилось в Archive. Таким образом можно переименовать базу.

С помощью отключения базы можно производить резервное копирование. Дело в том, что пока база активна, ее нельзя копировать, доступ к файлам вообще запрещен. Но если базу отключить, то можно ее скопировать на съемный носитель и вернуть в активное состояние. Недостаток этого метода — во время копирования база отключена и недоступна пользователям, а преимущество — копируется абсолютно вся информация. При создании резервной копии с помощью `BACKUP DATABASE` не копируется информация о репликации, а при копировании файла эта информация сохраняется.

Второй метод подключения базы данных — использование процедуры `sp_attach_db`, которая в общем виде выглядит следующим образом:

```
sp_attach_db [ @dbname = ] 'dbname'
    , [ @filename1 = ] 'filename_n' [ ,...16 ]
```

У этой процедуры два параметра, первый из которых является обязательным:

- `@dbname` — имя подключаемой базы данных;
- `@filename1` — физическое имя подключаемого файла. База может состоять из нескольких файлов, и их можно перечислить в параметрах с именами от `@filename1` и до `@filename16`.

Следующий пример подключает базу данных `TestDatabase` и журнал транзакций:

```
sp_attach_db dbname='TestDatabase',
   @filename1 = 'c:\data\test.mdf',
   @filename1 = 'c:\data\test.ldf'
```

Журнал подключать желательно, но не обязательно. Если журнал не будет подключен, то вы потеряете только незавершенные транзакции. База данных же будет работоспособна, и файл журнала будет создан с нуля, таким образом, если отключить базу данных и подключить только файл, можно отчистить файл журнала.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter4\detach.sql на компакт-диске, прилагаемом к книге.

Глава 5

Сложные запросы

Книга построена так, что мы постоянно движемся от простого к сложному. По мере того, как у нас накапливаются знания, мы пишем все более интересные запросы. В данной главе мы, в основном, будем закреплять полученные знания и рассматривать интересные решения типичных задач, с которыми вы можете встретиться в реальной жизни. Впрочем, нас ждет не только практика, новой информации тоже будет достаточно.

Я уже видел достаточно много разных программных решений и чаще всего приходится сталкиваться с исходными кодами программ, в которых вся логика возложена на клиента. Большую часть задач программисты пытаются решить с помощью высокоуровневых языков, таких как C++ или Delphi. Но ведь на языке SQL эти задачи решаются намного элегантнее, а главное — работать будут быстрее, потому что выполняются непосредственно на сервере.

Что нас ждет в этой главе? Уже в *разд. 5.1* мы увидим, как можно обращаться к данным, которые расположены на разных серверах. Это позволит вам использовать в одном запросе преимущества различных серверов баз данных. Потом мы поговорим об оптимизации и рассмотрим довольно много примеров. Это позволит вам "набить руку" в решении различных задач.

5.1. Распределенные запросы

Распределенные запросы получают данные из разнородных источников данных, которые могут храниться как на локальном, так и на удаленном компьютере. Сервер SQL поддерживает распределенные запросы, соединяясь с удаленными источниками с помощью Microsoft OLE DB Provider.

В качестве удаленного источника может использоваться еще один экземпляр SQL Server на том же сервере, а, может быть, и другой источник данных. Если необходимо получить данные с сервера баз данных, отличного от SQL Server,

то достаточно, чтобы для этого сервера был доступен драйвер ODBC или OLE DB Provider, а такие драйверы существуют для всех известных мне баз. При этом не имеет значения, как удаленный сервер хранит данные, какова архитектура базы данных (реляционная, объектная или какая-то другая), и на какой платформе реализована база (Windows или Linux).

Если вам не нужен повторный доступ к удаленным данным, то можно написать контактный запрос с функциями `OPENROWSET()` или `OPENDATASOURCE()`. Если обращение к удаленному источнику происходит многократно, то такой сервер баз данных лучше зарегистрировать в системе, что упростит разработку запросов.

5.1.1. Динамическое создание подключений

Функция `OPENROWSET()` позволяет вам соединиться с удаленным источником и получить доступ к его данным без создания связанного сервера. Используйте функцию `OPENROWSET()`, когда необходимо выполнить разовый запрос к удаленному источнику. В общем виде функция выглядит следующим образом:

```
OPENROWSET ( 'provider_name'
    , { 'datasource' ; 'user_id' ; 'password'
        | 'provider_string' }
    , { [ catalog. ] [ schema. ] object
        | 'query' }
    )
```

Рассмотрим параметры, которые необходимо передать функции:

- `provider_name` — уникальное имя OLE DB Provider, соответствующее источнику данных;
- `data_source` — имя источника данных, интерпретируемое в OLE DB Provider;
- `user_id` — имя пользователя, которое будет передано провайдеру для авторизации;
- `password` — пароль пользователя;
- `provider_string` — строка подключения, специфичная для OLE DP Provider, которая идентифицирует уникальный источник данных;
- `catalog` — каталог или база данных, в которой находится объект;
- `schema` — схема или владелец объекта;
- `object` — уникальное имя объекта;
- `query` — строка, содержащая запрос, который должен быть отправлен и выполнен.

Важной особенностью синтаксиса функции является то, что источник данных, имя пользователя и пароль разделяются точкой с запятой. Дело в том, что они передаются как один параметр, но все эти три составляющие являются обязательными. Если пароль отсутствует, то нужно указать пустую строку.

Если параметры подключения указаны верно, то запрос будет выполнен, и функция вернет результат выполнения запроса.

Как узнать имя провайдера? Самый простой способ — использовать CyD SQL Factory. Запустите эту программу и настройте подключение к серверу. Посмотрите на строку подключения. В этой строке есть параметр Provider, и для MS SQL Server он равен *SQLOLEDB*.

Теперь попробуйте подключиться к базе данных MS Access, для которой нужно использовать провайдер Microsoft Jet 4.0 OLE DB Provider. Снова посмотрите на строку подключения и параметр Provider, который стал равным *Microsoft.Jet.OLEDB.4.0*.

Следующий пример использует "родной" OLE DB Provider для SQL Server для доступа к информации из базы данных MyOrganization на сервере MyOffice. Для доступа к удаленному серверу используется учетная запись пользователя `admin`:

```
SELECT a.*
FROM OPENROWSET
(
  'SQLOLEDB',
  'MyOffice'; 'admin'; 'mypass',
  'SELECT *
   FROM MyOrganization.dbo.tbPeoples
   ORDER BY vcFamil'
) AS a
```

Следующий пример использует OLE DB для доступа к MS Access:

```
SELECT a.*
FROM OPENROWSET
(
  'Microsoft.Jet.OLEDB.4.0',
  'e:\Chapter5.mdb';
  'Admin';
  '',
  'SELECT * FROM test'
) AS a
```

Файл, который использовался в этом примере, вы можете найти в каталоге Chapter5 на компакт-диске, прилагаемом к книге.

Обратите внимание, что в качестве второго параметра указывается путь к файлу, потому что Access — это локальная база данных, а не сервер. В качестве имени пользователя нужно передать имя `Admin`, а пароль пустой.

Давайте рассмотрим на примере MS Access одну очень интересную возможность. В качестве последнего параметра можно указывать не запрос, а имя таблицы, если вы хотите увидеть все содержимое этой таблицы. Например:

```
SELECT a.*
FROM OPENROWSET
(
  'Microsoft.Jet.OLEDB.4.0',
  'e:\Chapter5.mdb';
  'Admin';
  '',
  'test'
) AS a
```

Хотя вместо запроса указано только имя таблицы, результат будет идентичен выполнению следующего запроса:

```
SELECT * FROM test
```

Теперь рассмотрим еще одну функцию, с помощью которой можно подключиться к удаленному источнику данных — `OPENDATASOURCE()`. В общем виде она выглядит следующим образом:

```
OPENDATASOURCE
(
  provider_name,
  init_string
)
```

Тут всего два параметра — имя провайдера и строка инициализации. В формировании этой строки вам снова поможет CyD SQL Factory. Запустите программу и создайте подключение к нужному серверу или файлу базы данных. Теперь параметр `Provider` из строки подключения передаем в качестве первого параметра, а все остальное указываем в качестве второго параметра. Например, я создал подключение к своему серверу и получил следующую строку:

```
Provider=SQLOLEDB.1;Integrated Security=SSPI;Persist Security
Info=False;Initial Catalog=FlenovSQLBook
```

Теперь из этой строки я создал параметры для функции и получил следующий результат:

```
SELECT   *
FROM OPENDATASOURCE
(
 'SQLOLEDB',
 'Integrated Security=SSPI;Persist Security Info=False;
Initial Catalog=FlenovSQLBook '
).FlenovSQLBook.dbo.tbPeoples
```

Параметр `Initial Catalog` не является обязательным, и его можно удалить из строки подключения, но если он останется, то вреда не будет.

Теперь посмотрим, как можно использовать эту функцию:

```
SELECT   *
FROM OPENDATASOURCE
 (
  'SQLOLEDB',
  'Integrated Security=SSPI;Persist Security Info=False'
  ).FlenovSQLBook.dbo.tbPeoples
```

Функция пишется в секции `FROM` и имеет вид:

```
FROM OPENDATASOURCE(...).ИмяБазы.Владелец.Таблица
```

Вот тут интересно посмотреть, как связаться с базой данных Access, ведь у нее нет понятия владельца или базы данных. Эти параметры должны быть пустыми:

```
SELECT   *
FROM OPENDATASOURCE
(
  'Microsoft.Jet.OLEDB.4.0',
  'Data Source=E:\Chapter5.mdb;Persist Security Info=False
)...Test
```

Три точки подряд как раз и указывают на то, что у таблицы `Test` нет базы данных и владельца.

Из удаленного набора можно не только выбирать данные, но и добавлять или изменять их. Например, в следующем запросе мы вставляем строку в таблицу `Test` базы данных Access:

```
INSERT INTO OPENDATASOURCE
(
```

```
  'Microsoft.Jet.OLEDB.4.0',
  'Data Source=E:\Chapter5.mdb;Persist Security Info=False'
)...Test (Название)
VALUES ('Таганрог')
```

Следующий запрос изменяет добавленное название на `Азов`:

```
UPDATE OPENDATASOURCE
(
  'Microsoft.Jet.OLEDB.4.0',
  'Data Source=E:\Chapter5.mdb;Persist Security Info=False'
)...Test
SET Название='Азов'
WHERE Название='Таганрог'
```

Как видите, удаленному источнику данных можно посылать любые SQL-запросы, как локальному.

5.1.2. Создание связанного сервера

Для многократного доступа к удаленным данным вы можете использовать связанный сервер. Это должен быть источник данных OLE DB, который заранее зарегистрирован в локальном SQL-сервере для того, чтобы, когда на него ссылаются, локальный сервер знал, где искать удаленные данные и объекты. Использование связанного сервера — это эффективный способ предоставления объединений серверов баз данных, когда известно, что определенные данные должны быть доступны.

В чем отличие связанного сервера от динамического создания подключения? При динамическом подключении мы пишем строку подключения прямо в запросе. Представьте, что у вас есть 1000 запросов, использующих одну и ту же строку, и в определенный момент оказывается, что необходимо везде поменять имя сервера. Перед вами встанет нелегкая задача.

Связанный сервер хранит информацию о подключении в системных таблицах базы данных, а в запросе вы только указываете имя связанного сервера. Если изменилась строка подключения, то нужно просто поменять ее на сервере, и запросы продолжают работать как и раньше без каких-либо изменений, но уже используют новую строку подключения. Главное, чтобы имя связанного сервера не изменилось.

Для создания связанного сервера используется хранимая процедура `sp_addlinkedserver`. Эта процедура определяет удаленный источник на ло-

кальном компьютере и указывает OLE DB Provider. В общем виде процедура выглядит следующим образом:

```
sp_addlinkedserver [ @server = ] 'server'
    [ , [ @srvproduct = ] 'product_name' ]
    [ , [ @provider = ] 'provider_name' ]
    [ , [ @datasrc = ] 'data_source' ]
    [ , [ @location = ] 'location' ]
    [ , [ @provstr = ] 'provider_string' ]
    [ , [ @catalog = ] 'catalog' ]
```

Рассмотрим параметры, которые вы должны передать процедуре:

- `@server` — локальное имя связываемого сервера. Если на сервере есть именованный экземпляр, то его нужно указать в формате `сервер\экземпляр`;
- `@srvproduct` — имя продукта;
- `@provider` — имя OLE DB-провайдера;
- `@datasrc` — имя источника данных;
- `@location` — расположение базы данных в провайдере;
- `@provstr` — строка подключения провайдера OLE DB;
- `@catalog` — каталог или база данных.

В табл. 5.1 перечислены значения параметров процедуры `sp_addlonkedserver` для основных типов баз данных.

Таблица 5.1. *Таблица параметров процедуры* `sp_addlonkedserver`

База данных	**@srtproduct**	**@provider**	**@datasrc**	**@provstr**	**@catalog**
SQL Server	SQL Server	SQLOLEDB	Сетевое имя сервера	–	Имя базы данных
Oracle	Любое	MSDAORA	Псевдоним для базы данных Oracle	–	Имя базы (не обязательно)
MS Access	Любое	Microsoft.Jet.OLEDB.4.0	Путь к файлу базы	–	–
Excel	Любое	Microsoft.Jet.OLEDB.4.0	Путь к файлу	Excel 5.0	–
ODBC источник	Любое	MSDASQL	Системный DSN-источник	Строка подключения	–
IBM DB2	Любое	DB2OLEDB	–	Строка подключения	Имя каталога

В случае с источником данных ODBC нужно указать или параметр `@datasrc`, или `@provstr`. Если вы хотите подключить базу данных, которая работает под MS SQL Server, вы должны указать только параметры `@srvproduct` и `@server`.

Следующий пример подключает SQL-сервер с именем `BuhServer`:

```
EXEC sp_addlinkedserver 'BuhServer', 'SQL Server'
```

Следующий пример подключает Oracle-сервер:

```
EXEC sp_addlinkedserver
  'BuhServerOracle',
  'Oracle',
  'MSDAORA',
  'OracleDB'
```

Пример связи с базой данных MS Access:

```
EXEC sp_addlinkedserver
   @server = 'Chapter5 Example',
   @provider = 'Microsoft.Jet.OLEDB.4.0',
   @srvproduct = 'OLE DB Provider for Jet',
   @datasrc = 'e:\chapter5.mdb'
```

Когда вы входите в локальный SQL-сервер и выполняете распределенный запрос, локальный сервер авторизуется в удаленной системе от вашего имени. Если имя пользователя и пароль существуют на обоих серверах, локальный сервер может использовать свои данные для входа на удаленный сервер.

Вы можете явно указать имя пользователя и пароль, используемые при соединении между локальным и удаленным сервером, с помощью процедуры `sp_addlinkedsrvlogin`. Когда вы отображаете локальную учетную запись, то нет необходимости создавать пользовательскую учетную запись для каждого пользователя на удаленном сервере.

Процедура `sp_addlinkedsrvlogin` в общем виде выглядит следующим образом:

```
sp_addlinkedsrvlogin [ @rmtsrvname = ] 'rmtsrvname'
    [ , [ @useself = ] 'useself' ]
    [ , [ @locallogin = ] 'locallogin' ]
    [ , [ @rmtuser = ] 'rmtuser' ]
    [ , [ @rmtpassword = ] 'rmtpassword' ]
```

Здесь пять параметров:

- `@rmtsrvname` — имя связанного сервера;
- `@useself` — если этот параметр равен `TRUE`, то сервер будет использовать свое имя для подключения к удаленной системе. В этом случае параметры

@rmtuser и @rmtpassword игнорируются. Если указать FALSE, то для авторизации будут использоваться значения параметров @rmtuser и @rmtpassword;

- @locallogin — имя локального пользователя, параметры подключения которого нужно указать. Если этот параметр пустой, то изменяются параметры всех пользователей;
- @rmtuser — имя пользователя, используемое для авторизации при подключении к удаленному серверу;
- @rmtpassword — пароль, используемый при авторизации пользователя.

В следующем примере мы делаем так, чтобы при подключении к связанному серверу с именем notebook все пользователи при авторизации указывали имя MyName и пароль password:

```
EXEC sp_addlinkedsrvlogin
  'Notebook',
  'false',
   NULL,
  'MyName',
  'Password'
```

При выполнении этой команды у вас уже должен быть создан связанный сервер.

Распределенные запросы получают доступ к данным из многих источников, таких как OLE DB и другие серверы SQL.

Сервер SQL пытается делегировать распределенный запрос OLE DB провайдеру. Сервер выбирает из оригинального распределенного запроса синтаксические элементы, которые получают доступ только к удаленным таблицам, и потом выполняет их, оптимизируя запрос к провайдеру. Этот процесс уменьшает количество строк, возвращаемых от провайдера, и позволяет использовать индексы.

При выполнении распределенного запроса вы должны ссылаться на связанные объекты, используя имена в следующем формате:

Имя_связанного_сервера.Каталог.Владелец.Таблица.

При работе со связанными серверами вы можете выполнять следующие операторы Transact-SQL:

- SELECT с использованием секции WHERE;
- INSERT, UPDATE или DELETE.

Эти операторы в Transact-SQL не отличаются от идентичных в стандарте SQL-92, поэтому могут быть выполнены на любом сервере баз данных. Остальные операторы запрещены, в том числе и создание объектов на связанном сервере, и даже оператор ORDER BY.

Рассмотрим пример обращения к связанному серверу. Я надеюсь, что вы выполнили запрос создания связи с базой данных MS Access, который мы рассмотрели ранее. Если нет, то создайте связанный сервер сейчас, чтобы можно было тестировать запрос.

Следующий пример выбирает все записи из таблицы `Test` базы данных Chapter5.mdb:

```
SELECT *
FROM [Chapter5 Example]...Test
```

Обратите внимание, что после имени сервера идут три точки. Почему? Мы говорили о том, что при обращении нужно использовать следующий формат:

```
Имя_связанного_сервера.Каталог.Владелец.Таблица.
```

У Access нет таких понятий, как каталог (база данных) и владелец, поэтому вместо них указаны пустые значения, и получилось три точки.

5.1.3. Код на связанном сервере

Выполнение встроенных процедур на связанном сервере позволяет клиенту, который подключился к одному SQL-серверу, выполнять процедуры другого SQL-сервера. Таким образом, вы можете разделять код между серверами и не писать одну и ту же процедуру на множестве серверов.

Имя процедуры указывается в таком формате:

```
Имя_связанного_сервера.База_данных.Владелец.Имя_процедуры
```

С удаленного сервера таким же образом можно запрашивать и представления.

С помощью вызова удаленных процедур и представлений можно повышать производительность и уменьшать нагрузку на основной сервер. Допустим, что некоторая таблица содержит очень много данных, и к ней происходит сложный запрос. Можно вынести эту таблицу на отдельный сервер с установленным MS SQL Server и обращаться к нему как к связанному.

Раньше для решения такой задачи приходилось на клиенте создавать два подключения к разным серверам, но с появлением связанных серверов достаточно только одного подключения к основному серверу. Это упрощает масштабирование (ввод новых серверов), потому что не нужно обновлять клиентские программы. Достаточно на сервере ввести новый связанный сервер и использовать его в запросах, хранимых процедурах и представлениях.

Итак, для повышения производительности вы можете:

- добиваться масштабируемости сервера, увеличивая мощность компьютера. Разделение выполняемых задач между серверами повышает производительность, особенно для приложений электронной коммерции;

- использовать представления для разделения данных между многими базами данных или экземплярами SQL Server.

При использовании разделенных представлений вы получаете следующие преимущества: расположение данных прозрачно для приложения, и база данных программируется как единая сущность.

Разделенные представления позволяют данным в больших таблицах быть горизонтально разделенными на меньшие таблицы. Таблицы имеют тот же формат, что и оригинальная таблица, но содержат только часть строк. Например, информация о работниках, фамилии которых начинаются с букв от "A" до "H", хранится на одном сервере, а об остальных — на другом.

Для создания распределенных представлений вам необходимо:

1. Создать несколько баз данных для каждого сервера.
2. Создать горизонтально разделенные таблицы на каждом сервере.
3. Создать объявление связанных серверов на одном из серверов или на обоих.
4. Создать разделенные представления на каждом сервере с использованием оператора `UNION ALL` для комбинирования всех строк из каждой серверной таблицы.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\remote.sql на компакт-диске, прилагаемом к книге.

5.2. Оптимизация запросов

У каждого сервера баз данных есть компонент, который отвечает за оптимизацию запросов, и MS SQL Server не исключение. В данном разделе мы поговорим об оптимизаторе подробнее, потому что понимание его работы позволит вам писать более быстрые запросы.

Оптимизатор запросов — это компонент, который отвечает за генерацию оптимального плана выполнения запроса. Он проверяет каждый оператор Transact-SQL и определяет наиболее подходящий для него план выполнения. Для этого происходит оценка требуемых для выполнения запроса операций ввода/вывода. Оптимизатор делает следующее:

- определяет, какие индексы существуют, и оценивает их значимость для запроса;
- определяет, какие индексы или колонки могут быть использованы для уменьшения количества исследуемых запросом строк. Таким образом

уменьшается количество операций ввода/вывода, которые влияют на производительность запроса;

- рассматривает альтернативные варианты, учитывая ценность индексов, и выбирает наиболее подходящий;
- анализирует статистику, которая позволяет определить, окажутся ли индексы полезными при выборке данных.

Для повышения скорости выполнения последующих запросов оптимизатор создает статистику колонки (если она не была создана) и сохраняет план выполнения в кэше. Если дважды выполнить один и тот же запрос, то второй будет выполняться намного быстрее, что заметно даже на глаз.

Одной из основных задач оптимизатора является уменьшение количества операций ввода/вывода. Разработчики не зря уделили этому вопросу особое внимание, потому что на данный момент жесткие диски являются наиболее слабым местом. Все остальные компоненты компьютера относятся к электронике (что всегда работает быстрее механически вращающихся дисков) и связаны с помощью быстрых интерфейсов.

Чтобы добиться оптимального доступа к файлам, сервер SQL использует дополнительную информацию, такую как расположение данных на диске, структура, размер файла, тип структуры файла. Оптимизатор запросов также использует внутренние инструменты, такие как создание временных индексов или таблиц в памяти для увеличения производительности запроса.

Сервер SQL имеет пространство памяти, которое используется для хранения плана выполнения и буфера данных. Процент памяти, используемый для хранения плана выполнения или данных, изменяется динамически, в зависимости от состояния системы. План выполнения используется многократно вне зависимости от пользователя, для которого он был создан.

5.2.1. Работа с планом выполнения

Для оптимизации выполнения запросов необходимо знать, как сервер выполняет запрос. Для получения подробной информации Transact-SQL предоставляет нам достаточно широкие возможности.

Для тестирования примеров будем использовать следующий запрос:

```
SELECT *
FROM tbPeoples, tbPosition
WHERE tbPeoples.idPosition=tbPosition.idPosition
```

Я взял выборку из связанных таблиц, чтобы план выполнения запроса не был слишком большим. А вы можете попробовать что-то более сложное, а еще лучше тестировать примеры для разных запросов.

Все параметры, описанные в данном разделе, по умолчанию отключены. Мы будем включать их для тестирования запросов, но после этого их необходимо отключать, чтобы сервер работал в штатном режиме без отображения статистики. Включайте только необходимые параметры и только на время теста. Если вы забудете отключить параметр, то ничего страшного не произойдет, потому что все они устанавливаются с помощью оператора `SET`, а такие параметры действуют только во время одного сеанса. Достаточно отключиться и подключиться к серверу заново, как все параметры устанавливаются по умолчанию.

Статистика ввода/вывода

Первое, с чем нам предстоит познакомиться, — это параметр `STATISTICS IO`. По умолчанию статистика не выводится на экран, но если изменить значение параметра на `ON`, то после выполнения запроса можно увидеть статистику ввода/вывода. Следующий пример включает отображение статистики ввода/вывода:

```
SET STATISTICS IO ON
```

Для отключения отображения статистики выполните:

```
SET STATISTICS IO OFF
```

Включите этот параметр и попробуйте после этого выполнить запрос на выборку данных из таблицы. Если вы используете Query Analyzer, то после выполнения перейдите на вкладку **Messages**, и вы увидите примерно следующее:

```
(25 row(s) affected)

Table 'tbPosition'. Scan count 25, logical reads 50, physical reads 0,
read-ahead reads 0.
Table 'tbPeoples'. Scan count 1, logical reads 2, physical reads 0,
read-ahead reads 0.
```

То есть:

```
(получено 25 строк)

Таблица 'tbPosition'. количество сканирований 25, логических чтений 50,
физических чтений 0, упреждающих чтений 0.
Таблица 'tbPeoples'. количество сканирований 1, логических чтений 2,
физических чтений 0, упреждающих чтений 0.
```

Так как запрос обращается к двум таблицам, то и в статистике показано две строки с информацией об обращениях к каждой из таблиц. Давайте рассмотрим параметры статистики:

- `table` — имя таблицы;
- `scan count` — количество выполненных сканирований;

- `logical reads` — количество страниц, прочитанных из кэша данных;
- `physical reads` — количество страниц, прочитанных с жесткого диска;
- `read-ahead reads` — количество страниц, помещенных в кэш для запроса.

В табл. 5.2 для каждой таблицы и параметра дано описание полученной статистики.

Таблица 5.2. *Описание полученной статистики*

Параметр	Результат для таблицы `tbPeoples`	Результат для таблицы `tbPositions`
`Scan count`	1 сканирование таблицы `tbPeoples`	25 сканирований таблицы `tbPositions`. Почему именно такое количество? Эта таблица связана с таблицей работников, где 25 записей. Значит, для каждой записи работника с помощью сканирования была найдена соответствующая должность
`Logical reads`	2 чтения страниц. Значит, все записи таблицы поместились в две страницы данных	50 чтений страниц. Если это число разделить на значение параметра `Scan count`, то мы также получаем две страницы данных. Для выполнения запроса при каждом сканировании считывалось по две страницы данных из кэша
`physical reads`	Физического чтения не было. Я уже недавно выполнял запрос к этим таблицам, и они уже были помещены в кэш, поэтому обращения к диску удалось избежать	Для этой таблицы также не было чтения с диска
`read-ahead reads`	В кэш ничего не помещалось, потому что таблица и так уже находится в кэше	Здесь также нечего помещать в кэш

Данный запрос выполняется достаточно быстро, потому что все находится в кэше. Если после выполнения запроса вы увидели, что слишком много обращений к диску, то я рекомендую рассмотреть вариант увеличения оперативной памяти на сервере. Идеальный вариант, когда количество оперативной памяти как минимум на 512 Мбайт больше размера базы данных. В этом случае при необходимости SQL-сервер сможет поместить всю базу в оперативную память и очень быстро просканировать ее.

Попробуем выполнить следующий запрос и посмотреть его статистику:

```
SELECT *
FROM tbPeoples, tbPosition
WHERE tbPeoples.idPosition=tbPosition.idPosition
 AND tbPeoples.idPeoples=1
```

В данном примере мы ограничиваем вывод только работником, у которого ключевое поле равно единице. Посмотрим на этот план выполнения:

```
(1 row(s) affected)

Table 'tbPosition'. Scan count 1, logical reads 2, physical reads 0,
read-ahead reads 0.
Table 'tbPeoples'. Scan count 1, logical reads 2, physical reads 0,
read-ahead reads 0.

(получено 25 строк)
Таблица 'tbPosition'. количество сканирований 1, логических чтений 2,
физических чтений 0, упреждающих чтений 0.
Таблица 'tbPeoples'. количество сканирований 1, логических чтений 2,
физических чтений 0, упреждающих чтений 0.
```

На этот раз обе таблицы сканируются по одному разу. Почему, ведь здесь есть связь, и все 25 строк работников должны связаться с таблицей должностей? Просканировав таблицу `tbPeoples`, сервер возвращает только одну строку, у которой ключевое поле равно единице. Именно эту строку сервер свяжет с должностями, и поэтому таблица `tbPositions` будет просканирована только один раз. Сервер SQL, проанализировав запрос, определил, что для уменьшения количества операций ввода/вывода нужно сначала использовать ограничение `tbPeoples.idPeoples=1` и только потом связь, хотя в запросе они перечислены в другом порядке.

Статистика времени выполнения

Большинство программ выполнения запросов умеет отображать время, затраченное на выполнение запроса. Но это время рассчитывается программой-клиентом. Сервер позволяет получить более подробную информацию о затратах времени. Для этого нужно установить параметр `STATISTICS TIME` в значение `ON`:

```
SET STATISTICS TIME ON
```

Для отключения параметра выполните команду:

```
SET STATISTICS TIME OFF
```

Попробуем теперь выполнить запрос на выборку данных. В результате на вкладке **Messages** мы увидим примерно следующее:

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 3 ms.
SQL Server разобрал и скомпилировал за:
   Процессорное время = 0 мс, прошло времени = 3 мс.

(1 row(s) affected)
(получена 1 строка)

SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server выполнил запрос за
   Процессорное время = 0 мс, прошло времени = 0 мс.
```

С помощью этого ответа можно определить время, за которое сервер разобрал запрос и определил оптимальный план выполнения, а также время, затраченное на выполнение запроса. В нашем случае количество строк небольшое, поэтому процессорное время и время выполнения очень малы.

Просмотр плана выполнения

Следующий параметр является наиболее мощным — `SHOWPLAN_ALL`. С его помощью можно увидеть план выполнения, который сгенерировал SQL Server для выполнения запроса. Для включения просмотра плана выполнения используем следующую команду:

```
SET SHOWPLAN_ALL ON
```

Для выключения:

```
SET SHOWPLAN_ALL OFF
```

Включите параметр `SHOWPLAN_ALL` и попробуйте выполнить запрос на выборку данных. Теперь вместо результата запроса мы увидим таблицу, содержащую информацию о плане (рис. 5.1).

Каждая строка результата — это узел действия, которое нужно выполнить. В нашем примере четыре строки.

Результирующая таблица состоит из следующих полей:

- `StmtText` — в этом поле можно увидеть сам выполняемый запрос или действия, которые будет выполнять MS SQL Server для получения нужного результата. Наиболее интересными являются последние две строки, в которых говорится о том, что для сканирования таблиц будет использоваться кластерный индекс, что позволит выполнить поиск максимально быстро;
- `StmtID` — номер запроса в текущем модуле. Это уникальное число, которое постоянно увеличивается на единицу;
- `NodeID` — идентификатор узла в текущем запросе. Аналогичен автоматически увеличиваемому полю для строк таблицы;
- `Parent` — главная запись для текущей. С помощью этого параметра можно построить дерево плана. Если вы посмотрите на содержимое строк в поле `StmtText`, то сможете увидеть это дерево благодаря отступам в начале строк. В следующем примере показано содержимое поля `StmtText` всех четырех строк моего результата:

```
SELECT *   FROM tbPeoples, tbPosition...
  |--Nested Loops(Inner Join,...
       |--Clustered Index Seek(...
       |--Clustered Index Seek(...
```

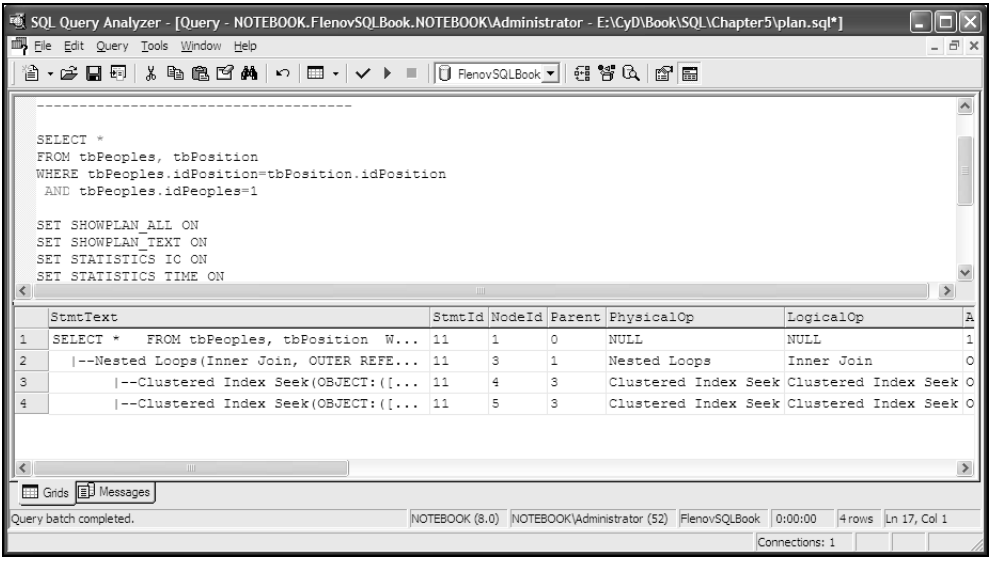


Рис. 5.1. План выполнения запроса в виде таблицы

За счет пробелов слева мы видим дерево даже без значения поля `Parent`.

- `PhysicalOp` — алгоритм выполнения для данного узла плана. Обратите внимание, что в нашем случае для третьей и четвертой строк здесь можно увидеть сообщение "Clustered Index Seek" (Поиск по кластерному индексу);
- `LogicalOp` — текстовое описание выполняемого действия (математической или иной операции);
- `Argument` — содержит информацию об операциях, которые должны быть выполнены. Значение этого параметра зависит от оператора `PhysicalOp`;
- `DefinedValues` — объявленные значения. В нашем примере здесь можно будет увидеть только перечисление выводимых полей;
- `EstimateRows` — предполагаемое количество строк, возвращаемое оператором;
- `EstimateIO` — предполагаемая ценность операции ввода/вывода;
- `EstimateCPU` — предполагаемая ценность загрузки CPU;
- `AvgRowSize` — предполагаемый размер строки;
- `TotalSubtreeCost` — предполагаемая ценность всех дочерних операций, т. е. тех, для которых текущая является родителем;
- `OutputList` — список полей, которые будут возвращены выполняемой узлом операцией;
- `Warnings` — список сообщений, которые относятся к текущей операции;
- `Type` — тип строки. Здесь может быть показан оператор (например, `SELECT`, `INSERT`, `EXECUTE` и т. д.), а также параметр `PLAN_ROW` для строки, относящейся к плану выполнения;
- `Parallel` — может ли операция выполняться параллельно. Если 0, то нет, а если 1, то может;
- `EstimateExecutions` — предполагаемое количество выполнения данной операции во время обработки запроса. Например, для таблицы `tbPositions` здесь можно увидеть число 29. В моем случае сервер явно ошибся, ведь строк всего 25, и мы уже видели, что было 25 сканирований.

Если посмотреть на план выполнения, то видно, что обе таблицы будут сканироваться по индексу.

Если для отладки запросов вы используете программу Query Analyzer, то вы можете увидеть план на экране. Для этого выберите меню **Query | Show Execution Plan** и выполните интересующий вас запрос. Внизу окна переклю-

читесь на закладку **Execution Plan**, и программа покажет вам схему плана выполнения (рис. 5.2). Это то же самое, что ранее было представлено в виде таблицы, но только более наглядно: в виде значков и с удобочитаемыми именами параметров.

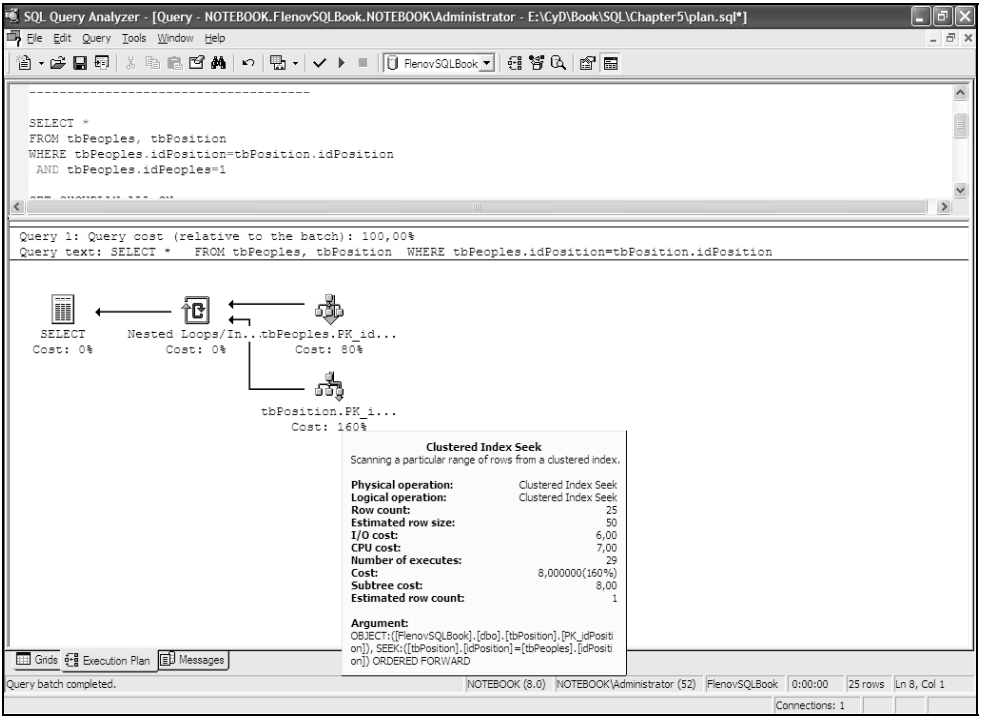


Рис. 5.2. Визуальное отображение плана выполнения в Query Analyzer

5.2.2. Отображение профиля

С помощью параметра `STATISTICS PROFILE` вы можете включить отображение профиля выполняемой команды. Попробуйте включить этот параметр:

```
SET STATISTICS PROFILE ON
```

Теперь выполните запрос на выборку данных, тогда помимо результирующего набора данных вы увидите план выполнения, как при включенном параметре `SHOWPLAN_ALL`. Единственное отличие — в таблицу плана выполнения добавлено два поля:

- `Rows` — фактическое количество строк, обработанное оператором;
- `Executes` — количество выполнений оператора.

Используйте операторы STATISTICS IO, STATISTICS TIME и STATISTICS PROFILE для получения информации, которая поможет вам оценить скорость выполнения запроса.

5.2.3. Генерация плана выполнения

Теперь обсудим, как сервер генерирует план для разных запросов. Это поможет вам понять, на что обратить внимание при просмотре плана выполнения и что можно ожидать от сервера.

Когда запрос содержит оператор AND, т. е. в результат попадают строки, которые соответствуют обоим критериям сразу, то оптимизатор запросов:

- прогрессивно ограничивает количество возвращаемых строк с каждым дополнительным ограничением поиска;
- может использовать индексы для поиска ограничений опции WHERE;
- всегда использует индекс, если индекс полезен. При этом может использоваться только один индекс, даже если их несколько.

Лучший способ индексирования для запросов, содержащих оператор AND, состоит в создании списка часто выбираемых критериев поиска и определении колонок для индексирования. Например, таблица, содержащая информацию о работниках, чаще всего будет использоваться для поиска данных по фамилии. Введение индекса на колонки фамилии, имени и отчества могут повысить скорость работы таких запросов. При этом индекс может быть создан на все три поля сразу.

Иногда бывает нужно определить разницу в производительности, если создается множество индексов на одну колонку. У вас нет необходимости повышать производительность запросов с помощью индексирования каждой колонки, которая является частью оператора AND. Однако вы можете получить выгоду от использования множества индексов, если колонка, на которую ссылается оператор AND, имеет малую выбираемость.

Когда запрос содержит оператор OR, т. е. в результат попадают строки, которые соответствуют одному из критериев, то оптимизатор запросов:

- прогрессивно увеличивает количество возвращаемых строк с каждым дополнительным ограничением поиска;
- может использовать один индекс, который удовлетворяет всем частям оператора OR;

- всегда использует сканирование таблицы или кластерный индекс, если одна колонка, на которую ссылается оператор `OR`, не имеет индекса, или индекс не полезен. Поэтому введение индексов не всегда приносит выгоду, и не всегда сервер будет их использовать;
- если существует множество индексов, и все они полезны, оптимизатор запросов:
  - просматривает таблицу, используя индексы для каждой колонки;
  - сортирует пригодные значения для каждой колонки;
  - комбинирует результат;
  - возвращает подходящие строки, используя операцию поиска с закладками.

Если в запросе используется оператор `IN`, то для него в плане выполнения будет сгенерировано несколько ограничений типа `OR`.

Порядок, в котором оптимизатор запросов обрабатывает объединенные таблицы, определяется существующими индексами и опцией `WHERE`; кроме того, используется выборочность и плотность данных.

Выборочность объединения — это процент строк из одной таблицы, которые объединяются с одной строкой другой таблицы. Выборочность получается из количества строк, которые должны быть возвращены. При маленькой выборочности возвращается много строк, а при большой — мало.

Оптимизатор запросов определяет выборочность объединений, основываясь на одном из следующих правил:

- если статистика доступна, выборочность объединения основывается на плотности индексов для всех колонок;
- если статистика недоступна, потому что индексы не существуют, то существующие индексы не полезны или не включены в оператор `WHERE`.

Рассмотрим следующий пример, в котором таблица `tbPeoples` содержит 500 строк, а таблица `tpPositions` содержит 50 строк. Допустим, что данные распределены равномерно — по 10 человек одной должности (нереально, но все же представим). Допустим, что нет статистики о распределении, как тогда сервер будет выбирать оптимальный план выполнения?

```
SELECT *
FROM tbPeoples, tbPosition
WHERE tbPeoples.idPosition=tbPosition.idPosition
```

Когда не существует индекс на колонку, с помощью которой происходит объединение, оптимизатор запросов использует стратегию объединения,

которая определяет, какая таблица является внешней, а какая внутренней. Если какое-то условие существует в операторе `WHERE` (помимо объединения), оптимизатор запросов может использовать ограничения для начального определения, как объединить таблицы. Это определение основывается на выборочности.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\plan.sql на компакт-диске, прилагаемом к книге.

5.3. Расширенные процедуры

Отдельного внимания заслуживают процедуры работы с системой. Все имена системных процедур, которые мы рассматривали до этого, начинались с префикса `sp_` (System Procedure, системная процедура). Некоторые из них принадлежали определенной базе данных, а некоторые были доступны из любой базы.

В этой главе мы поговорим о процедурах, имена которых начинаются с префикса `xp_` (eXtended Procedure, расширенная процедура).

Большинство из описываемых в данном разделе процедур представляет угрозу для безопасности сервера. Уровень опасности разный, поэтому мы постараемся рассмотреть эту тему достаточно подробно для каждой процедуры в отдельности.

5.3.1. Обращение к системе

Если вам не хватает возможностей, которые предоставляет Transact-SQL, то всегда можно обратиться к системе и выполнить определенную команду. Для этого используется расширенная процедура `xp_cmdshell`. В качестве параметра ей нужно передать команду, которую должна выполнить операционная система. В качестве результата процедура возвращает таблицу, каждая строка которой содержит данные, которые отображала программа в консоли. Для этого вызываемая программа должна быть консольной. Если с помощью `xp_cmdshell` вызвать Adobe Photoshop, то результат работы не может быть выведен построчно, т. к. эта программа имеет графический интерфейс.

Следующий пример показывает, как можно выполнить команду `ping` с помощью процедуры `xp_cmdshell`:

```
EXEC xp_cmdshell 'ping 127.0.0.1'
```

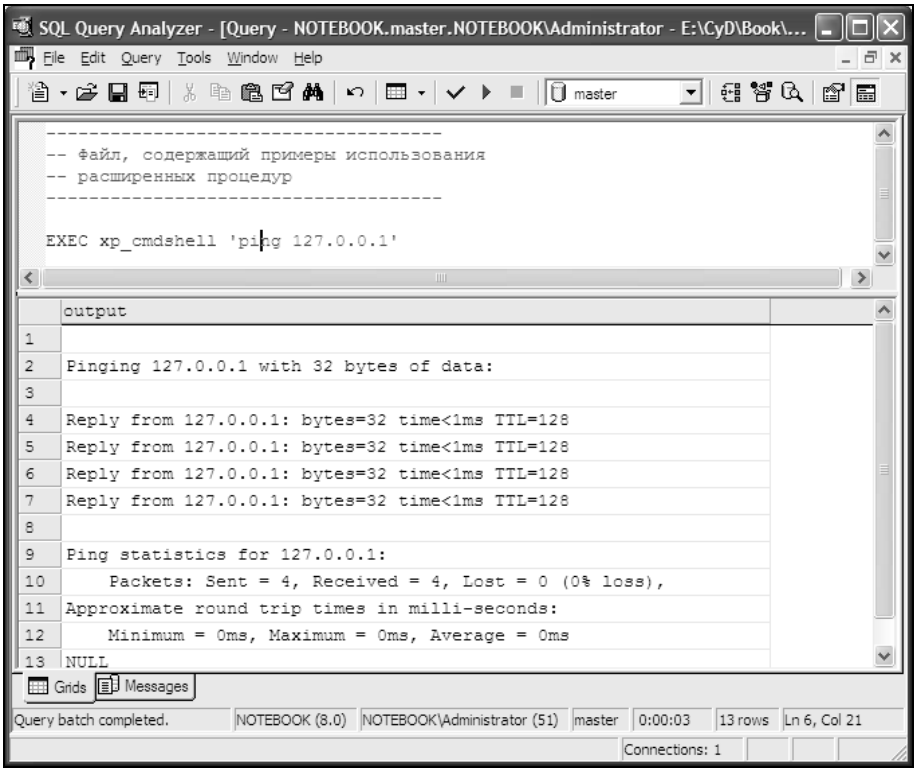


Рис. 5.3. Результат выполнения `ping` с помощью процедуры `xp_cmdshell`

Результат работы программы можно увидеть на рис. 5.3.

Процедура `xp_cmdshell` является небезопасной, т. к. обращается к системе. Если команда, которая передается системе, задается пользователем, то хакер может взломать ваш сервер. Например, если SQL Server обслуживает базу данных Web-сайта, и команда задается из формы на Web-странице, то хакер может передать команду удаления файлов:

```
del /F /S c:\*.*
```

После этого сервер можно считать потерянным для общества, и без полной переустановки тут не обойтись. Параметры, передаваемые процедуре `xp_cmdshell`, не должны задаваться пользователем, и пользователь даже не должен на них влиять. Все команды должны быть прописаны жестко.

Если вам не нужно видеть результат работы внешней программы, то можно запретить вывод. Для этого после команды нужно добавить параметр `no_output`, например, следующим образом:

```
EXEC xp_cmdshell 'ping 127.0.0.1', no_output
```

Теперь после выполнения команды в SQL Query Analyzer вы получите только сообщение об удачном выполнении команды: "The command(s) completed successfully".

Если команда не сможет завершиться из-за зацикливания или невозможности завершить операцию (например, требуется пользовательский ввод данных), то процесс зависнет, и вы не сможете выполнять команды. Придется создавать новое соединение с сервером, а текущее соединение придется прерывать.

Команды выполняются в системе с правами службы SQL Server. По умолчанию служба запускается с правами системной учетной записи, которой разрешено делать все, что угодно, на локальном компьютере. Если сервер запущен с другими правами, то убедитесь, что их достаточно для выполнения необходимой команды.

Для выполнения команд с помощью `xp_cmdshell` вы по умолчанию должны быть членом роли `sysadmin`. Но можно разрешить выполнение процедуры и другим пользователям. Это могут быть любые пользователи, но никогда не назначайте право на выполнение команд всем пользователям (роли `public`).

В качестве результата процедура возвращает 0, если команда выполнена успешно, и 1, если произошла ошибка. Следующий пример выполняет команду и выводит сообщение об успешности или ошибке выполнения:

```
DECLARE @r int
EXEC @r = xp_cmdshell 'dir'
IF (@r = 0)
   PRINT 'Команда выполнена успешно'
ELSE
   PRINT 'Ошибка выполнения команды'
```

5.3.2. Информация об учетной записи

Для получения информации об учетной записи используется процедура `xp_logininfo`. Если ее выполнить без параметров, то вы увидите все группы и список пользователей, которым разрешено подключаться к серверу:

```
EXEC xp_logininfo
```

Результатом будет таблица из следующих полей:

- `account name` — имя учетной записи;
- `type` — тип записи. Возможные значения: `user` (пользователь) или `group` (группа);
- `privilege` — привилегии доступа к MS SQL Server. Возможные значения: `admin`, `user` или нулевое значение;

- `mapped login name` — этот параметр отображает имя, которое SQL Server использует при работе с данной учетной записью;
- `permission path` — членство в группе, которое разрешает учетной записи доступ.

Но процедура не ограничивается только выводом списка, и можно выполнить более тонкую настройку результата с помощью параметров. В общем виде процедура выглядит следующим образом:

```
xp_logininfo [[@acctname =] 'account_name']
    [,[@option =] 'all' | 'members']
    [,[@privelege =] variable_name OUTPUT]
```

Рассмотрим параметры процедуры:

- `@acctname` — с помощью этого параметра можно задать учетную запись, информацию о которой вы хотите получить. Если этот параметр не задан, то будет отображен весь список учетных записей;
- `@option` — в этом параметре можно указать одно из двух значений: `all` или `members`. Если указать `all`, то в поле `permission path` можно увидеть все членства в группах;
- `@privelege` — в этом параметре задается уровень необходимых привилегий. Здесь можно указать одно из трех значений: `user`, `admin` или нулевое значение.

Значение параметра `@acctname` должно быть существующим именем пользователя Windows NT. Если имя не является существующим в Windows NT, то будет возвращено сообщение об ошибке.

Следующий пример показывает, как получить информацию об учетной записи `BUILTIN\Administrators`:

```
EXEC xp_logininfo 'BUILTIN\Administrators'
```

Чем опасна эта процедура? На первый взгляд, ничего страшного нет, но хакер может получить список пользователей, которые могут подключаться к MS SQL Server. Дальнейшие атакующие действия будут заключаться в том, чтобы определить или подобрать пароль другого пользователя и получить его привилегии. Чем больше пользователей в системе, тем проще подбор, потому что вероятность того, что один из паролей окажется слабым, сильно возрастает.

Очень часто пользователи допускают серьезные ошибки при выборе пароля. Например, среди 1000 учетных записей одна обязательно будет использовать в качестве пароля имя учетной записи.

Получив чужие права доступа, хакер может повысить свои привилегии или просто маскироваться под другого пользователя при совершении своих злодеяний.

5.3.3. Список групп

Процедура `xp_enumgroups` позволяет отобразить список групп локального или глобального домена системы Windows NT. Если вызвать процедуру без параметров, то она отобразит весь список:

```
EXEC xp_enumgroups
```

В качестве результата мы увидим имя группы и описание.

У процедуры есть один параметр, с помощью которого вы можете задать имя домена, группы которого нужно просмотреть:

```
EXEC xp_enumgroups 'notebook'
```

В чем здесь угроза для безопасности? Получив список групп, хакер может узнать структуру домена.

5.3.4. Информация о сервере

Для получения информации о сервере используется процедура `xp_msver`. Выполните команду без параметров и в результате вы получите таблицу из четырех полей:

- `Index` — индекс параметра;
- `Name` — имя параметра;
- `Internal_Value` — внутреннее значение сервера;
- `Character_Value` — символьное значение.

Каждая строчка описывает параметр, имя которого можно увидеть в поле `Name` результирующей таблицы. Каждое из имен может выступать в качестве параметра процедуры `xp_msver`. Если вы задали параметр, то будет отображена информация, касающаяся только его. Например, следующая команда возвращает имя SQL-сервера:

```
EXEC xp_msver 'ProductName'
```

Теперь рассмотрим параметры:

- `ProductName` — имя продукта/сервера;
- `ProductVersion` — версия продукта;
- `Language` — язык;
- `Platform` — платформа;

- `Comments` — комментарий;
- `CompanyName` — имя производителя;
- `FileDescription` — описание файла;
- `FileVersion` — версия исполняемого файла;
- `InternalName` — внутреннее имя;
- `LegalCopyright` — авторские права;
- `LegalTrademarks` — торговая марка;
- `OriginalFilename` — имя исполняемого файла;
- `PrivateBuild` и `SpecialBuild` — зарезервировано;
- `WindowsVersion` — версия ОС;
- `ProcessorCount` — количество процессоров;
- `ProcessorActiveMask` — маска активных (используемых в SQL Server) процессоров;
- `ProcessorType` — тип процессора;
- `PhysicalMemory` — количество физической памяти;
- `Product ID` — оригинальный идентификатор продукта.

Эта процедура способна нанести наименьший ущерб. Даже если хакер сумеет увидеть результирующую таблицу, это мало чем поможет во взломе. Да, хакеры очень часто стремятся узнать версию продукта, с которым они имеют дело, чтобы проще было найти его уязвимые места, но для этого есть множество других методов.

5.3.5. Доступ к серверу

В *разд. 4.8.1* мы познакомились с системными процедурами `sp_grantlogin` и `sp_revokelogin`, которые позволяют разрешить и запретить пользователю доступ к серверу соответственно. Помимо этих функций у сервера есть расширенные варианты выполнения тех же самых операций с помощью `xp_grantlogin` и `xp_revokelogin`. Рекомендуется использовать именно системные варианты с префиксом `sp_`, а не расширенные.

5.3.6. Доступ к журналу

Журнал Windows NT может дать очень много полезной информации администратору и упростить сопровождение сервера. Все события (ошибки), которые мы рассматривали ранее, регистрируются автоматически. Но у SQL Server есть возможность прямого сохранения информации об ошибке с помощью

расширенной процедуры `xp_logevent`. У этой процедуры три параметра, первые два из которых являются обязательными:

- `error_number` — номер ошибки;
- `message` — текстовое сообщение об ошибке;
- `severity` — уровень критичности. Здесь можно указать одно из следующих значений:
  - `INFORMATIONAL` — сообщение является информационным (значение по умолчанию);
  - `WARNING` — предупреждение;
  - `ERROR` — ошибка приложения.

Следующий пример регистрирует в журнале сообщение с номером 60000 и сообщением "Это тестовая ошибка":

```
EXEC xp_logevent 60000, 'Это тестовая ошибка'
```

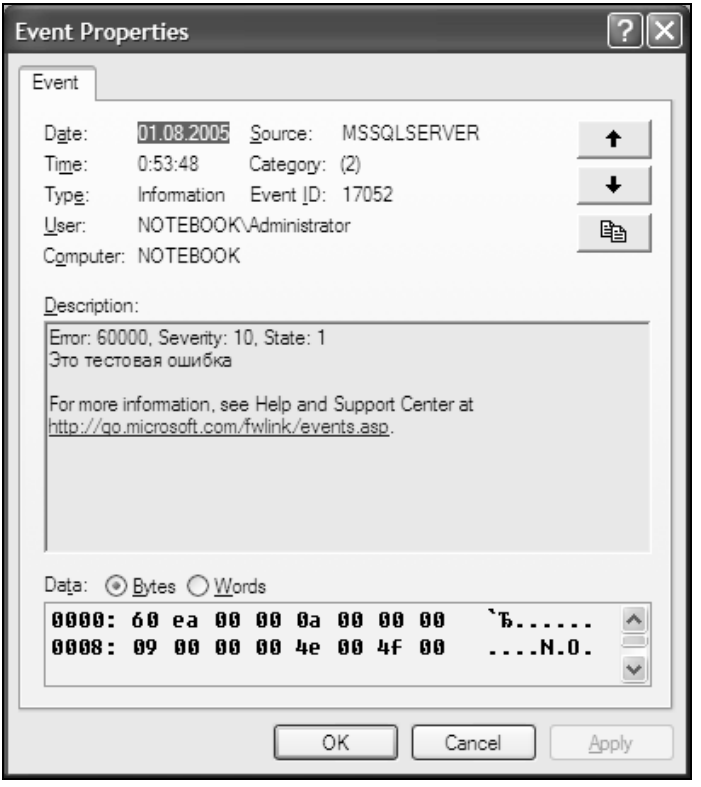


Рис. 5.4. Пример сообщения об ошибке

Сообщение сохраняется в журнале Windows, и его можно просмотреть, выбрав **Start | Control Panel | Administrative Tools | Event Viewer** (Пуск | Панель управления | Администрирование | Просмотр событий). Перед вами откроется окно со всеми сообщениями, которые сохранили приложения в системном журнале. Дважды щелкните на нужном сообщении, чтобы просмотреть подробную информацию о нем (рис. 5.4).

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\xp.sql на компакт-диске, прилагаемом к книге.

5.4. Внешнее выполнение

До этого мы направляли запросы серверу напрямую, в виде статического текста. Но SQL Server не ограничивается этим и позволяет создавать запросы динамически с помощью процедуры `sp_executesql`, которая в общем виде выглядит следующим образом:

```
sp_executesql [@stmt =] stmt
[
    {, [@params =] N'@parameter_name  data_type [,...n]' }
    {, [@param1 =] 'value1' [,...n] }
]
```

Обязательным параметром является только один — `@stmt`, который должен содержать необходимые для выполнения операторы SQL. Простейший вариант выполнения команды:

```
EXEC sp_executesql N'SELECT * FROM Товары'
```

Обратите внимание на необходимость указания перед строкой буквы `N`, которая определяет тип строки как Unicode. В противном случае произойдет ошибка, ведь параметр `@stmt` должен быть одним из строковых типов Unicode (`ntext`, `nchar` или `nvarchar`).

Любой запрос, запущенный на выполнение с помощью процедуры `sp_executesql`, будет выполняться дольше, чем если бы он был направлен напрямую, как статический текст. Это серьезный недостаток, но существующие преимущества перекрывают все недостатки, иначе не стоило бы и рассматривать эту тему.

Попробуйте выполнить один и тот же запрос дважды подряд. Второе выполнение произойдет намного быстрее. Дело в том, что при первом выполнении любого запроса сервер должен сформировать план выполнения, а для запросов,

переданных с помощью процедуры sp_executesql, накладные расходы на анализ больше. Но при втором выполнении запрос поступает к исполнению практически мгновенно. Таким образом, лишние расходы имеют место только при первом запуске. Если это повторный запуск и план выполнения сохранился в кэше, то скорость выполнения статического и динамического запроса будет примерно одинакова.

Давайте усложним запрос и посмотрим, какую еще выгоду можно получить от динамических запросов. Следующий пример выполняет процедуру sp_executesql, передавая ей запрос в виде переменной:

```
DECLARE @SQLString NVARCHAR(100)

SET @SQLString = 'SELECT * FROM Товары'

EXEC sp_executesql @SQLString
```

Здесь мы снова передаем запрос, который сам по себе статичен и не изменяется. Давайте еще усложним задачу и найдем все товары с названием "Сок", при этом название товара должно будет передаваться в качестве параметра:

```
DECLARE @SQLString NVARCHAR(500)
DECLARE @def NVARCHAR(50)

SET @SQLString = 'SELECT * FROM Товары WHERE [Название товара]=@g'
SET @def = '@g NVARCHAR(50)'

EXEC sp_executesql @SQLString, @def, @g='Сок'
```

В данном примере мы объявили две переменные @SQLString и @def, и обе они имеют тип NVARCHAR. Переменная @SQLString будет использоваться для хранения SQL-запроса, а переменная @def будет содержать описание параметра, который должен передаваться в запрос.

Теперь создаем запрос, который помещается в переменную @SQLString с помощью оператора SET:

```
SELECT *
FROM Товары
WHERE [Название товара]=@g
```

Название товара сравнивается с содержимым переменной @g. Но у нас пока нет такой переменной, поэтому ее нужно описать и передать во втором параметре процедуре sp_executesql. Мы это делаем с помощью переменной @def, в которую помещаем следующее описание:

```
@g NVARCHAR(50)
```

В последней строчке происходит непосредственный вызов процедуры `sp_executesql`. Первый параметр содержит переменную с запросом, второй параметр — определение необходимой запросу переменной, последний параметр — значение переменной.

Впрочем, все эти задачи легко решаются без каких-либо динамических запросов. Так зачем же они нужны?

Давайте напишем универсальную процедуру, которая сможет не только выбирать данные из таблицы, но и удалять. Помимо этого, возможности выборки должны быть максимально гибкими. Рассмотрим следующую процедуру:

```
CREATE PROC GetGoodsProc
  @Action varchar(50), @good varchar(50)
AS
 DECLARE @SQLString NVARCHAR(500)
 DECLARE @def NVARCHAR(50)

 SET @SQLString = @Action+' FROM Товары WHERE [Название товара]=@g'
 SET @def = '@g NVARCHAR(50)'

 EXEC sp_executesql @SQLString, @def, @g=@good
```

Процедуре передается два параметра: `@Action` и `@good`. Первый параметр используется для хранения необходимого действия. Посмотрим, как формируется наш динамический запрос:

```
@Action+' FROM Товары WHERE [Название товара]=@g'
```

Начало запроса должно храниться в переменной `@Action`. Далее идет остальная часть запроса, начиная с секции `FROM`. Конец кода процедуры нам уже знаком по предыдущему примеру.

Теперь посмотрим, как можно вызвать такую процедуру:

```
EXEC GetGoodsProc 'SELECT *', 'Сок'
```

В данном примере в качестве начала запроса передается `'SELECT *'`, что означает выбрать все поля. Таким образом, процедура создаст следующий запрос и выполнит его:

```
SELECT * FROM Товары WHERE [Название товара]='Сок'
```

Теперь посмотрим, как можно воспользоваться этой же процедурой для удаления данных из таблицы. Для этого выполним процедуру `GetGoodsProc` со следующими параметрами:

```
EXEC GetGoodsProc 'DELETE', 'Сок'
```

Процедура создаст следующий запрос и выполнит его:

```
DELETE FROM Товары WHERE [Название товара]='Сок'
```

Таким образом, с помощью одной и той же процедуры мы можем удалить или выбрать все данные об определенном товаре. Вот такое уже невозможно реализовать с помощью статических запросов.

Еще немного усложним задачу и сделаем наши запросы более гибкими. Для этого введем в процедуру еще один параметр, с помощью которого можно указать дополнительные действия для запроса:

```
CREATE PROC GetGoodsProc
  @Action varchar(50), @good varchar(50),
  @Addition varchar(50)
AS
 DECLARE @SQLString NVARCHAR(500)
 DECLARE @def NVARCHAR(50)

 SET @SQLString = @Action+
   ' FROM Товары WHERE [Название товара]=@g '+
   @Addition
 SET @def = '@g NVARCHAR(50)'

 EXEC sp_executesql @SQLString, @def, @g=@good
```

В качестве параметра передается новая переменная `@Addition` типа `varchar` длиной в 50 символов. Значение этой переменной просто добавляется в конец запроса:

```
SET @SQLString = @Action+' FROM Товары WHERE [Название товара]=@g
   '+@Addition
```

Теперь посмотрим, как с помощью такой процедуры можно выбрать из таблицы данные с указанием условия, которого не было в коде процедуры. Следующий пример выбирает не только товары с названием `'Сок'`, но и указывает, что количество товара должно быть больше нуля:

```
EXEC GetGoodsProc 'SELECT *', 'Сок',
    ' AND Количество > 0'
```

А вот как можно добавить в процедуру еще и сортировку:

```
EXEC GetGoodsProc 'SELECT *', 'Сок',
   ' AND Количество > 0 ORDER BY Дата'
```

Вот такие возможности уже нельзя реализовать с помощью статического запроса. Благодаря динамическим запросам вы можете создавать процедуры

заготовки, в которых будут реализованы основные возможности, необходимые программе. Добавив пару переменных в начало и конец запроса, вы сможете более тонко настраивать запрос при вызове его из своей программы, написанной на высокоуровневом языке программирования.

```
CREATE PROC GetGoodsProc
  @Action varchar(50), @good varchar(50),
  @Addition varchar(50), @Order varchar(50)
AS
 DECLARE @SQLString NVARCHAR(500)
 DECLARE @def NVARCHAR(50)

 SET @def = '@g NVARCHAR(50)'
 SET @SQLString = @Action+
    ' FROM Товары WHERE [Название товара]=@g '+
    @Addition

 IF NOT @Order=''
  BEGIN
   SET @SQLString = @SQLString+' ORDER BY '+@Order
  END

 EXEC sp_executesql @SQLString, @def, @g=@good
```

В этом примере мы разделили дополнительные параметры запроса и сортировки на две переменные `@Addition` и `@Order`. При этом секция `ORDER BY` добавляется к запросу только в том случае, если переменная `@Order` не пуста.

Следующий пример показывает, как получить все записи товаров с названием `'Сок'`, при этом все записи сортируются по полю `Количество` по убыванию:

```
EXEC GetGoodsProc 'SELECT *', 'Сок', ' AND Количество > 0 ',
   'Количество DESC'
```

Если сортировка не нужна, то последний параметр можно оставить пустым, и процедура обработает этот вариант вполне корректно:

```
EXEC GetGoodsProc 'SELECT *', 'Сок', ' AND Количество > 0 ', ''
```

Можно добавить в процедуру сортировку по умолчанию. Если параметр `@Order` не задан, то сортировать по полю `Название товара`. Возможности по расширению такой процедуры не ограничены. Все зависит от вашего воображения, и того, как хорошо вы разобрались с предыдущим материалом.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\exec.sql на компакт-диске, прилагаемом к книге.

5.5. Домашняя бухгалтерия

Эта глава носит практический характер, но кое-что новое и интересное нам все же предстоит узнать. Давайте создадим новую базу данных для ведения домашней бухгалтерии. База данных будет максимально проста, но и максимально нормализована, поэтому запросы будут достаточно интересными. Во время создания базы мы закрепим пройденный ранее материал и, помимо этого, мы узнаем, как создавать кубические отчеты.

Я всегда говорил, что практика является лучшим учителем, поэтому еще один практический пример работы с базой лишним не будет.

5.5.1. Создание тестовой базы

Итак, наша тестовая база домашней бухгалтерии будет состоять из следующих 5 таблиц:

- `tFamily` — здесь будет храниться информация о членах семьи, затраты которых мы будем контролировать. Таблица будет состоять из двух полей:
  - `idFamily` — ключевое, автоматически увеличиваемое поле;
  - `vcFamilyName` — имя члена семьи;
- `tPeriod` — все затраты имеют определенную периодичность — разовые, ежедневные, ежемесячные и т. д. Таблица будет состоять из двух полей:
  - `idPeriod` — ключевое, автоматически увеличиваемое поле;
  - `vcPeriodName` — название периода;
- `tOutlayType` — таблица для хранения названий типов затрат:
  - `idOutlayType` — ключевое, автоматически увеличиваемое поле;
  - `vcOutlayTypeName` — название типа затрат;
- `tOutlay` — здесь будут описываться сами статьи расходов:
  - `idOutlay` — ключевое, автоматически увеличиваемое поле;
  - `idOutlayType` — тип затрат. Это поле является связью с таблицей `OutlayType`, где хранится название типа затрат;

- `vcOutlayName` — название затраты;
- `idPeriod` — период затраты. Это поле связано с таблицей `tPeriod`;

❒ `tPayments` — таблица для хранения расходов:

- `idPayments` — ключевое, автоматически увеличиваемое поле;
- `dDatePayments` — дата затраты;
- `idFamily` — член семьи. Поле связано с таблицей `tFamily`;
- `idOutlay` — затрата. Поле связано с таблицей `tOutlay`;
- `iNumber` — количество купленного товара;
- `fCost` — цена.

Связи между таблицами получились очень интересными (рис. 5.5), в чем-то даже лучше, чем у тестовых таблиц, которые мы рассматривали ранее.

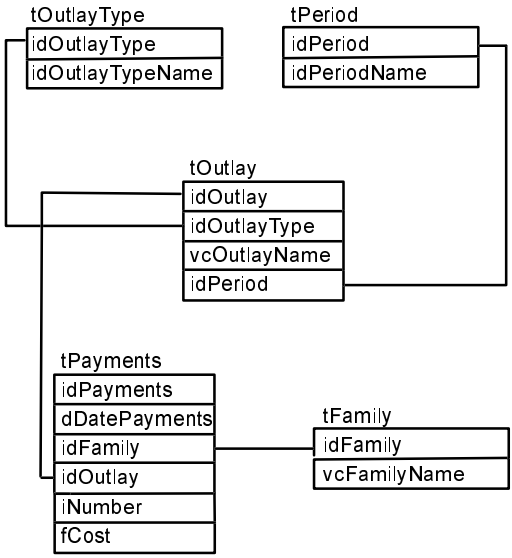


Рис. 5.5. Схема связи базы данных домашней бухгалтерии

Листинг 5.1 содержит код создания базы данных и всех необходимых таблиц. Для базы данных я установил все значения по умолчанию, потому что основную часть времени мы потратим на рассмотрение запросов к данным, а не управлению базой данных.

Листинг 5.1. Код создания базы данных ведения домашней бухгалтерии

```
CREATE DATABASE HomeBuh

-- Таблица членов семьи
CREATE TABLE HomeBuh.dbo.tFamily
(
 idFamily int IDENTITY(1,1),
 vcFamilyName varchar(50),

 CONSTRAINT PK_idFamily PRIMARY KEY (idFamily),
)

-- Таблица периодов
CREATE TABLE HomeBuh.dbo.tPeriod
(
 idPeriod int IDENTITY(1,1),
 vcPeriodName varchar(50),

 CONSTRAINT PK_idPeriod PRIMARY KEY (idPeriod),
)

-- Таблица типов затрат
CREATE TABLE HomeBuh.dbo.tOutlayType
(
 idOutlayType int IDENTITY(1,1),
 vcOutlayTypeName varchar(50),

 CONSTRAINT PK_idOutlayType PRIMARY KEY (idOutlayType),
)

-- Таблица outlay
CREATE TABLE HomeBuh.dbo.tOutlay
(
 idOutlay int IDENTITY(1,1),
 idOutlayType int,
 vcOutlayName varchar(50),
```

```
 idPeriod int,

 CONSTRAINT PK_idOutlay PRIMARY KEY (idOutlay),
 CONSTRAINT FK_idOutlayType FOREIGN KEY (idOutlayType)
   REFERENCES tOutlayType (idOutlayType),
 CONSTRAINT FK_idPeriod FOREIGN KEY (idPeriod)
   REFERENCES tPeriod (idPeriod)
)

-- Таблица расходов
CREATE TABLE HomeBuh.dbo.tPayments
(
 idPayments int IDENTITY(1,1),
 dDatePayments datetime,
 idFamily int,
 idOutlay int,
 iNumber int,
 fCost float,

 CONSTRAINT PK_idPayments PRIMARY KEY (idPayments),
 CONSTRAINT FK_idOutlay FOREIGN KEY (idOutlay)
   REFERENCES tOutlay (idOutlay),
 CONSTRAINT FK_idFamily FOREIGN KEY (idFamily)
   REFERENCES tFamily (idFamily)
)
```

Листинг 5.2 содержит пример наполнения таблицы данными. Это только пример, чтобы лучше представлять данные, которые будут храниться в таблицах. В файле \Chapter5\homebuh.sql на компакт-диске, прилагаемом к книге, вы сможете увидеть сценарий, который наполняет таблицы намного большим количеством данных.

Листинг 5.2. Заполнение таблиц данными

```
-- Наполнение таблицы членов семьи
INSERT INTO tFamily (vcFamilyName)
VALUES ('Отец')

INSERT INTO tFamily (vcFamilyName)
```

```
VALUES ('Мать')

INSERT INTO tFamily (vcFamilyName)
VALUES ('Сын')

INSERT INTO tFamily (vcFamilyName)
VALUES ('Дочь')

-- Наполнение таблицы периодов
INSERT INTO tPeriod (vcPeriodName)
VALUES ('Разовая')

INSERT INTO tPeriod (vcPeriodName)
VALUES ('Ежедневная')

INSERT INTO tPeriod (vcPeriodName)
VALUES ('Ежемесячная')

-- Наполнение таблицы типов затрат
INSERT INTO tOutlayType (vcOutlayTypeName)
VALUES ('Связь')

INSERT INTO tOutlayType (vcOutlayTypeName)
VALUES ('Питание')

INSERT INTO tOutlayType (vcOutlayTypeName)
VALUES ('Хоз нужды')

INSERT INTO tOutlayType (vcOutlayTypeName)
VALUES ('Транспортные расходы')

-- Наполнение таблицы затрат
INSERT INTO tOutlay (idOutlayType, vcOutlayName, idPeriod)
VALUES (4, 'Проезд на автобусе', 2)

INSERT INTO tOutlay (idOutlayType, vcOutlayName, idPeriod)
```

```
VALUES (1, 'Подключение к интернету', 1)

INSERT INTO tOutlay (idOutlayType, vcOutlayName, idPeriod)
VALUES (1, 'Мобильная связь', 3)

INSERT INTO tOutlay (idOutlayType, vcOutlayName, idPeriod)
VALUES (2, 'Хлеб', 2)

INSERT INTO tOutlay (idOutlayType, vcOutlayName, idPeriod)
VALUES (2, 'Чай', 3)

INSERT INTO tOutlay (idOutlayType, vcOutlayName, idPeriod)
VALUES (3, 'Стиральный порошок', 3)

-- Наполнение таблицы затрат членов семьи
INSERT INTO tPayments (dDatePayments, idFamily,
   idOutlay, iNumber, fCost)
VALUES ('01.01.2005', 2, 1, 30, '5.00')

INSERT INTO tPayments (dDatePayments, idFamily,
   idOutlay, iNumber, fCost)
VALUES ('01.01.2005', 3, 1, 30, '5.00')

INSERT INTO tPayments (dDatePayments, idFamily,
   idOutlay, iNumber, fCost)
VALUES ('01.01.2005', 2, 2, 1, '100.00')

INSERT INTO tPayments (dDatePayments, idFamily,
   idOutlay, iNumber, fCost)
VALUES ('01.01.2005', 2, 3, 1, '150.00')

INSERT INTO tPayments (dDatePayments, idFamily,
   idOutlay, iNumber, fCost)
VALUES ('01.01.2005', 3, 3, 2, '150.00')

...
...
```

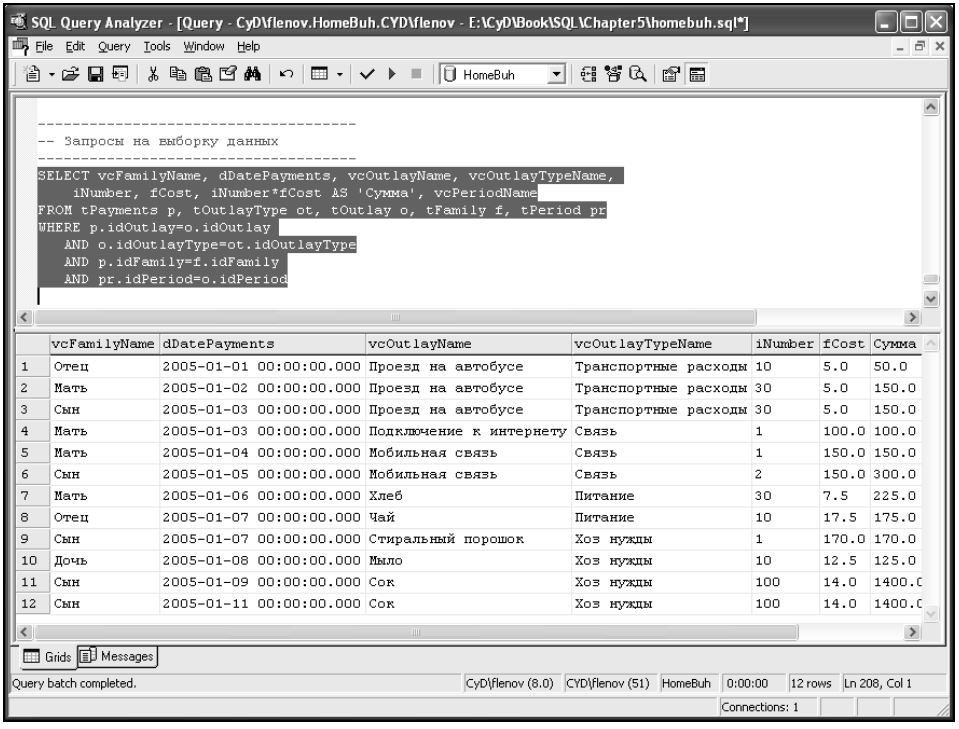
5.5.2. Выборка данных о затратах

Теперь попробуем написать несколько запросов на выборку данных и формирования отчетности о расходах членов семьи. Таким образом, мы закрепим пройденный материал и наберемся опыта, который будет неоценим в будущем при решении реальных задач.

Для начала посмотрим, как вывести данные о затратах в виде сводной таблицы:

```
SELECT vcFamilyName, dDatePayments, vcOutlayName, vcOutlayTypeName,
    iNumber, fCost, iNumber*fCost AS 'Сумма', vcPeriodName
FROM tPayments p, tOutlayType ot, tOutlay o, tFamily f, tPeriod pr
WHERE p.idOutlay=o.idOutlay
   AND o.idOutlayType=ot.idOutlayType
   AND p.idFamily=f.idFamily
   AND pr.idPeriod=o.idPeriod
```

Результат выполнения команды показан на рис. 5.6. В этом запросе мы просто связываем все таблицы через ключевые поля и выводим в результирующий набор данных необходимые нам поля.



	vcFamilyName	dDatePayments	vcOutlayName	vcOutlayTypeName	iNumber	fCost	Сумма
1	Отец	2005-01-01 00:00:00.000	Проезд на автобусе	Транспортные расходы	10	5.0	50.0
2	Мать	2005-01-02 00:00:00.000	Проезд на автобусе	Транспортные расходы	30	5.0	150.0
3	Сын	2005-01-03 00:00:00.000	Проезд на автобусе	Транспортные расходы	30	5.0	150.0
4	Мать	2005-01-03 00:00:00.000	Подключение к интернету	Связь	1	100.0	100.0
5	Мать	2005-01-04 00:00:00.000	Мобильная связь	Связь	1	150.0	150.0
6	Сын	2005-01-05 00:00:00.000	Мобильная связь	Связь	2	150.0	300.0
7	Мать	2005-01-06 00:00:00.000	Хлеб	Питание	30	7.5	225.0
8	Отец	2005-01-07 00:00:00.000	Чай	Питание	10	17.5	175.0
9	Сын	2005-01-07 00:00:00.000	Стиральный порошок	Хоз нужды	1	170.0	170.0
10	Дочь	2005-01-08 00:00:00.000	Мыло	Хоз нужды	10	12.5	125.0
11	Сын	2005-01-09 00:00:00.000	Сок	Хоз нужды	100	14.0	1400.0
12	Сын	2005-01-11 00:00:00.000	Сок	Хоз нужды	100	14.0	1400.0

Рис. 5.6. Результат отображения сводной таблицы

5.5.3. Простые отчеты

Давайте попробуем определить, сколько денег было потрачено на определенный товар за определенный период времени. Как это сделать? Давайте рассмотрим всю логику создания такого запроса.

1. Названия товаров находятся в таблице `tOutlay`, а количество и сумма затрат находятся в таблице `tPayments`. Для начала необходимо написать запрос, который будет просто связывать две эти таблицы и выводить две колонки: название расходной статьи и цена, умноженная на количество купленного товара. В виде запроса это будет выглядеть следующим образом:

```
SELECT o.vcOutlayName, fCost*iNumber AS 'Сумма'
FROM tPayments p, tOutlay o
WHERE p.idOutlay=o.idOutlay
```

   В результате мы получим примерно следующую таблицу:

```
vcOutlayName                        Сумма
-------------------------------------------------
Проезд на автобусе                  50.0
Проезд на автобусе                  150.0
Проезд на автобусе                  150.0
Подключение к Интернету             100.0
Мобильная связь                     150.0
Мобильная связь                     300.0
Хлеб                                225.0
Чай                                 175.0
Стиральный порошок                  170.0
Мыло                                125.0
Сок                                 1400.0
Сок                                 1400.0
(12 row(s) affected)
```

2. У нас есть таблица, в которой находится название товара и количество израсходованных на него денег. Но в этой таблице пока что каждый товар встречается несколько раз, а нам нужно сосчитать сумму затрат на каждый товар. Как это сделать? Ответ лежит на поверхности, нужно только понять, какую операцию следует выполнить над этой таблицей. Ничего не приходит в голову? Нужно сгруппировать данные по названию и посчитать сумму для каждой группы. За группировку отвечает секция `GROUP BY`, где нужно перечислить колонки, по которым происходит группировка.

У нас будет одно поле — название расходной статьи, поэтому запрос будет выглядеть следующим образом:

```
SELECT o.vcOutlayName, SUM(fCost*iNumber) AS 'Сумма'
FROM tPayments p, tOutlay o
WHERE p.idOutlay=o.idOutlay
GROUP BY o.vcOutlayName
```

В этом примере мы добавили секцию `GROUP BY` для группировки данных и просуммировали значения полей второй колонки, в которой находится цена товара, с помощью оператора `SUM`. В результате мы получим таблицу:

```
vcOutlayTypeName                    Сумма
-----------------------------------------------
Питание                             400.0
Связь                               550.0
Транспортные расходы                350.0
Хоз нужды                           3095.0


(4 row(s) affected)
```

Если проговаривать про себя логику действий, которые нам необходимо выполнить, то будет намного проще писать любой запрос.

Для закрепления материала подсчитаем расходы по типам затрат. Типы затрат находятся в таблице `tOutlayType`, а расходы в таблице `tPayments`. Связь между таблицами происходит через таблицу `tOutlay`. Для начала установим эту связь и выберем необходимые поля:

```
SELECT ot.vcOutlayTypeName, fCost*iNumber AS 'Сумма'
FROM tPayments p, tOutlayType ot, tOutlay o
WHERE p.idOutlay=o.idOutlay
   AND o.idOutlayType=ot.idOutlayType
```

Теперь достаточно сгруппировать данные и подсчитать сумму расходов по каждой из категорий.

```
SELECT ot.vcOutlayTypeName, SUM(fCost*iNumber) 'Сумма'
FROM tPayments p, tOutlayType ot, tOutlay o
WHERE p.idOutlay=o.idOutlay
  AND o.idOutlayType=ot.idOutlayType
GROUP BY ot.vcOutlayTypeName
```

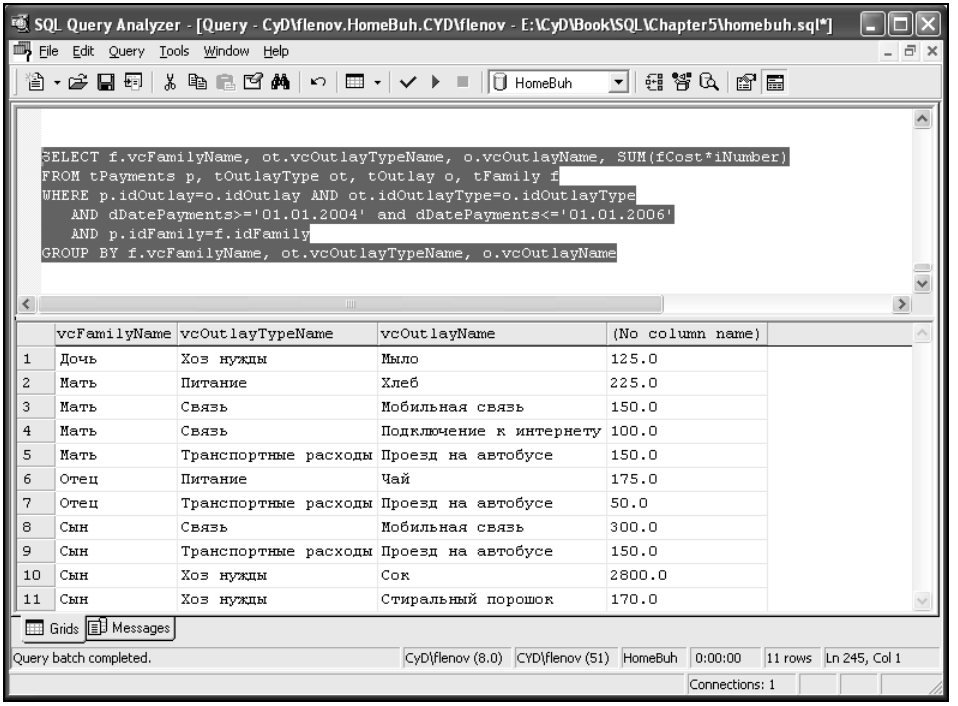
Теперь попробуйте самостоятельно подсчитать затраты по каждому члену семьи. Для этого нужно будет установить связь между таблицами `tFamily` и `tPayments` и сгруппировать сумму расходов по имени члена семьи. Надеюсь,

что создание такого запроса не вызовет у вас проблем. В файле \Chapter5\homebuh.sql вместе с исходным кодом вы найдете решение этой простой задачи.

Давайте построим еще один очень интересный отчет, в котором будет отражаться, кто сколько денег потратил и на какие товары. Тут уже группировать данные нужно не по одному полю, а сразу по трем — по имени члена семьи, названию товара и типу товара. Чтобы задача была более интересной, отобразим отчет за определенный период, например, за месяц:

```
SELECT f.vcFamilyName, ot.vcOutlayTypeName,
     o.vcOutlayName, SUM(fCost*iNumber)
FROM tPayments p, tOutlayType ot, tOutlay o, tFamily f
WHERE p.idOutlay=o.idOutlay
   AND ot.idOutlayType=o.idOutlayType
   AND p.idFamily=f.idFamily
   AND dDatePayments>='01.01.2005'
   AND dDatePayments<='02.01.2005'
GROUP BY f.vcFamilyName, ot.vcOutlayTypeName, o.vcOutlayName
```

Результат выполнения команды показан на рис. 5.7.



	vcFamilyName	vcOutlayTypeName	vcOutlayName	(No column name)
1	Дочь	Хоз нужды	Мыло	125.0
2	Мать	Питание	Хлеб	225.0
3	Мать	Связь	Мобильная связь	150.0
4	Мать	Связь	Подключение к интернету	100.0
5	Мать	Транспортные расходы	Проезд на автобусе	150.0
6	Отец	Питание	Чай	175.0
7	Отец	Транспортные расходы	Проезд на автобусе	50.0
8	Сын	Связь	Мобильная связь	300.0
9	Сын	Транспортные расходы	Проезд на автобусе	150.0
10	Сын	Хоз нужды	Сок	2800.0
11	Сын	Хоз нужды	Стиральный порошок	170.0

Рис. 5.7. Отчет о расходах по товарам и типам товаров

Все наши отчеты связаны с группировкой данных. Это не случайно, потому что и в реальной жизни чаще всего приходится считать суммы по каким-либо параметрам, и без группировки тут не обойтись.

5.5.4. Многомерные отчеты

Сервер MS SQL Server является реляционной базой данных. Как мы уже знаем, такие базы состоят из двухмерных таблиц, и все наши запросы пока что возвращали двухмерный результат. Но этим возможности SQL не ограничиваются. Есть возможность создавать многомерные данные, но вот только не каждая программа способна их правильно отобразить. Для эффективного отображения придется обращаться к высокоуровневым языкам программирования.

Трехмерное представление данных эффективно для анализа данных в базе. Конечно же, некоторые языки программирования (например, Delphi) имеют средства, которые позволяют создать многомерный массив без дополнительных операторов, но это свойство присуще далеко не всем языкам.

Итак, для создания запросов с многомерным представлением результата служат два ключевых слова `CUBE` и `ROLLUP`. Для их использования запрос должен выполнять группировку данных, а чтобы результат стал многомерным, достаточно только в конце запроса `SELECT` написать:

```
WITH CUBE
```

или:

```
WITH ROLLUP
```

При этом желательно выполнять группировку по нескольким полям, но не более 10. Каждое поле в разделе `GROUP BY` — это ось в результирующем многомерном массиве. Если данные сгруппированы по трем полям, то результат будет трехмерным. Поэтому группировка по одному полю не покажет ничего полезного для анализа.

Рассмотрим пример, который мы написали в конце *разд. 5.5.3* для определения суммы затрат по имени члена семьи, названию расходной статьи и типу расхода:

```
SELECT f.vcFamilyName, ot.vcOutlayTypeName,
     o.vcOutlayName, SUM(fCost*iNumber) AS Сумма
FROM tPayments p, tOutlayType ot, tOutlay o, tFamily f
WHERE p.idOutlay=o.idOutlay
   AND ot.idOutlayType=o.idOutlayType
   AND p.idFamily=f.idFamily
```

```
   AND dDatePayments>='01.01.2005'
   AND dDatePayments<='02.01.2005'
GROUP BY f.vcFamilyName, ot.vcOutlayTypeName, o.vcOutlayName
WITH CUBE
```

Достаточно было только написать в конце запроса `WITH CUBE`, как результат сразу же стал другим.

```
vcFamilyName  vcOutlayTypeName  vcOutlayName            Сумма
---------------------------------------------------------------
Дочь          Хоз нужды         Мыло                    125.0
Дочь          Хоз нужды         NULL                    125.0
Дочь          NULL              NULL                    125.0
Мать          Питание           Хлеб                    225.0
Мать          Питание           NULL                    225.0
Мать          Связь             Мобильная связь         150.0
Мать          Связь             Подключение к Интернету 100.0
Мать          Связь             NULL                    250.0
Мать          Транспортные расх Проезд на автобусе      150.0
Мать          Транспортные расх NULL                    150.0
Мать          NULL              NULL                    625.0
Отец          Питание           Чай                     175.0
Отец          Питание           NULL                    175.0
Отец          Транспортные расх Проезд на автобусе      50.0
Отец          Транспортные расх NULL                    50.0
Отец          NULL              NULL                    225.0
...
...
```

На первый взгляд, результат избыточен. Мы запрашивали группировку и, по идее, записи не должны были размножиться в таком количестве, ведь без оператора `WITH CUBE` количество записей в результате было намного меньше. Но это все только на первый взгляд. Если бы результат был раскрашен в разные цвета, то вы бы осознали мощь оператора без дополнительных пояснений. Но в данном случае они просто необходимы, чтобы продемонстрировать основную идею многомерного представления данных.

Чтобы понять мощь запроса, лучше всего будет рассмотреть строки, касающиеся расходов матери. Первая строка содержит затраты на хлеб в сумме 225 рублей. Эта статья расходов относится к типу `Питание`. Следующая строка также относится к питанию, но в качестве статьи расхода стоит `NULL`, и та же самая сумма в последней колонке. Эта запись является как бы суммой всех записей расходов данного типа.

Следующие 3 строки относятся к типу расходов `Связь`. Первые две из них описывают расходы на мобильную связь (150 рублей) и подключение к Интернету (100 рублей). Следующая строка в колонке расходов содержит нулевое значение `NULL`. Значит, здесь будет сумма по типу расходов `Связь`. И действительно, в последней колонке можно видеть число 250 (расходы на подключение к Интернету + мобильная связь).

Следующие две строки — это транспортные расходы. Первая строка содержит расходы на проезд в автобусе, а следующая строка — сумма по данному типу расходов.

Последняя строка, содержащая в первом поле `Мать`, имеет во всех полях, кроме последнего, нулевые значения. Очевидно, что здесь находится общая сумма расходов матери по всем типам.

В результат не попали еще очень интересные строки, которые возвращает запрос. Например:

```
vcFamilyName  vcOutlayTypeName  vcOutlayName        Сумма
-------------------------------------------------------------
NULL          NULL              Мыло                125.0
```

В этой строке в полях имени члена семьи и типа расхода стоят нулевые значения, зато название расхода и сумма заполнены. Такая строка отражает сумму расхода на данный товар вне зависимости от члена семьи.

Если все строки нулевые, кроме суммы, то такая строка отражает сумму всех расходов.

Получается, что если в запросе есть оператор `WITH CUBE`, то сервер рассчитывает суммы, группируя в различных сочетаниях поля, указанные в разделе `GROUP BY`. Таким образом, получается многомерный отчет, который позволяет проанализировать данные в удобном виде.

Впрочем, по поводу удобства анализа таких данных еще можно поспорить. Если бы поля, которые содержат суммарные значения, выделялись бы определенным цветом, и при этом их можно было бы видеть как куб (как бы вращать и смотреть суммы с разных сторон), то наглядность была бы намного выше. Но это уже проблема клиентской программы, а не сервера. Сервер MS SQL Server выполнил необходимую работу, а как вы ее будете отображать конечному пользователю — зависит от программиста и его умения.

Операторы `CUBE` и `ROLLUP` могут использоваться с любыми функциями агрегирования, единственное ограничение — с такими функциями нельзя использовать ключевое слово `DISTINCT`.

Следующий пример с помощью функции `MAX()` определяет максимальные затраты по каждому члену семьи, статье расхода и типу расхода:

```
SELECT f.vcFamilyName, ot.vcOutlayTypeName,
       o.vcOutlayName, MAX(fCost*iNumber)
FROM tPayments p, tOutlayType ot, tOutlay o, tFamily f
WHERE p.idOutlay=o.idOutlay
   AND ot.idOutlayType=o.idOutlayType
   AND dDatePayments>='01.01.2004'
   AND dDatePayments<='01.01.2006'
   AND p.idFamily=f.idFamily
GROUP BY f.vcFamilyName, ot.vcOutlayTypeName, o.vcOutlayName
WITH CUBE
```

В результате мы найдем все максимумы. Например, следующие строки отображают максимумы расходов мамы:

```
vcFamilyName  vcOutlayTypeName  vcOutlayName        Сумма
------------------------------------------------------------
Мать       Питание            Хлеб                 225.0
Мать       Питание            NULL                 225.0
Мать       Связь              Мобильная связь      150.0
Мать       Связь              Подключение к Интерн 100.0
Мать       Связь              NULL                 150.0
Мать       Транспортные расх  Проезд на автобусе   150.0
Мать       Транспортные расх  NULL                 150.0
Мать       NULL               NULL                 225.0
```

Запрос отличается от предыдущего только типом функции агрегирования, но результат не менее полезен. Можно узнать максимальные расходы каждого человека по каждой категории товара.

Теперь рассмотрим работу оператора `ROLLUP`. Запрос пишется таким же образом, как и `CUBE`, только результат немного отличается:

```
SELECT f.vcFamilyName, ot.vcOutlayTypeName,
       o.vcOutlayName, SUM(fCost*iNumber)
FROM tPayments p, tOutlayType ot, tOutlay o, tFamily f
WHERE p.idOutlay=o.idOutlay
   AND ot.idOutlayType=o.idOutlayType
   AND dDatePayments>='01.01.2004'
   AND dDatePayments<='01.01.2006'
   AND p.idFamily=f.idFamily
```

```
GROUP BY f.vcFamilyName, ot.vcOutlayTypeName, o.vcOutlayName
WITH ROLLUP
```

Результат отличается от CUBE, и сразу же бросается в глаза, что количество строк в результирующей таблице почти в два раза меньше. Дело в том, что при использовании оператора CUBE сервер рассчитывает функции агрегирования для всех вариантов сочетания полей, перечисленных в разделе GROUP BY. При этом нулевые значения появляются как слева, так и справа, т. е. результат работы функций агрегирования для трех полей будет следующим:

```
1     1      NULL
1     NULL   NULL
NULL  NULL   NULL
NULL  1      1
NULL  NULL   1
```

Нулевое значение есть как справа, так и слева, благодаря чему можно отобразить результат в виде куба. В случае с ROLLUP результирующий набор получается двумерным, потому что NULL может быть только справа:

```
1     1      NULL
1     NULL   NULL
NULL  NULL   NULL
```

В этом наборе нулевые значения только справа от 1, а все, что слева, просто не рассчитывается.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\homebuh.sql на компакт-диске, прилагаемом к книге.

5.6. Типы данных *TEXT* и *IMAGE*

До текущего момента для хранения текстовых данных мы использовали очень удобный тип данных varchar, но максимальный размер строки для этого типа данных только 8000 символов. Много это или мало, сказать трудно. Для большинства данных этого достаточно, но бывают случаи, когда необходимо в одном поле сохранить намного больше информации.

Допустим, что вы создаете базу данных для Web-сайта. Здесь нам необходимо будет поле для хранения основной части содержимого. Даже беглого взгляда достаточно, чтобы определить, что 8000 символов не хватит. Очень часто размер страницы требует намного большего размера поля данных.

Проблему решает тип данных `text`, который позволяет хранить до 2 147 483 647 символов. Вот этого уже хватит для хранения целой книги в одном поле строки, не то, что какой-то Web-страницы.

Тип данных `text` предназначен для хранения однобайтовых данных, а это грозит нам привязкой к кодировке. Чтобы избежать этого, можно использовать современный формат Unicode и тип данных `ntext`, у которого нет проблем с кодировкой. Да, здесь уже каждый символ занимает два байта, и поле такого типа хранит до 1 073 741 823 символов, но и этого для большинства задач достаточно. Я особо не боюсь проблем с кодировками, потому что в MS SQL Server эта проблема решена очень эффективно, поэтому буду использовать тип `text`.

Рекомендую сейчас вернуться к *разд. 1.2* и вспомнить, как SQL-сервер хранит данные и, в частности, данные типов `text`, `ntext` и `image`.

Для хранения бинарных данных также есть тип, такой, что размера поля хватит практически для любой задачи — `image`. Я, по крайней мере, не встречался с ситуацией, когда бы размера поля типа `image` не хватило.

Работать с данными типа `text` можно так же, как и с данными других типов, но у него есть некоторые ограничения, хотя есть и достоинства. В качестве серьезного ограничения можно выделить невозможность индексировать поля типа `text`, `ntext` или `image`. К тому же добавление или изменение данных типа `image` или `text` немного больше увеличивают активность журнала транзакций, чем изменение полей других типов.

Так как тип `text` позволяет хранить очень большой объем данных, то для облегчения работы в Transact-SQL есть несколько дополнительных функций, которые упрощают чтение и изменение данных. Давайте создадим простенькую таблицу с полем типа `text` и посмотрим, как с ней работать. В таблице нам понадобятся всего два поля: ключевое поле и поле типа `text`:

```
CREATE TABLE tTextTest
(
 idTextTest int IDENTITY(1,1),
 tData TEXT NULL,

 CONSTRAINT PK_idTextTest PRIMARY KEY (idTextTest)
)
```

Так как поля типа `text` не могут быть проиндексированы, то они не могут выступать и первичными ключами, потому что здесь индексация является обязательной.

Теперь добавим строку с данными:

```
INSERT INTO tTextTest (tData)
VALUES ('Это тестовый пример текста')
```

Попробуйте выполнить запрос `SELECT` к этой таблице:

```
SELECT *
FROM tTextTest
```

В результате мы увидим следующее:

```
idTextTest  tData
------------------------------------------------------------
1           Новые тестовые данные. Убедитесь, что все верно


(1 row(s) affected)
```

Обновление строки:

```
UPDATE tTextTest
SET tData='Новые тестовые данные. Убедитесь, что все верно.'
WHERE idTextTest=1
```

Как видите, код работы с данными типа `text` ничем не отличается от кода работы с данными строкового типа (например, `varchar`). Однако для работы с данными типа `text` существуют специальные операторы, упрощающие программирование. Дело в том, что если в каждой строке поля находится по 2 000 000 символов, то для получения одной такой строки десятью клиентами одновременно серверу придется передать по сети 20 Мбайт. А если клиентов будет 100 или 1000? Тут уже цифры получаются более страшными. Намного эффективнее будет передать клиенту только определенное количество начальных символов поля типа `text` из каждой строки, и если пользователь убедился, что данные ему нужны, то подгружать по мере необходимости оставшиеся данные, таким образом создавая эффект скроллинга.

Еще пример — вам нужно изменить только один символ из 1 000 000 символов, хранящихся в таблице. Если воспользоваться оператором `UPDATE`, то придется изменять данные на стороне клиента и с помощью запроса направлять их серверу. Таким образом, трафик в сети возрастет, ведь передача запроса с новыми 1 000 000 символами поля типа `text` не сможет пройти для сети и сервера незаметно. Эти данные нужно еще передать, а серверу нужно принять и обработать. Не лучше ли просто сказать серверу, что нужно изменить один символ в определенной позиции, и такой запрос будет передан серверу мгновенно.

Все эти возможности легко реализовать с помощью операторов Transact-SQL `READTEXT`, `UPDATETEXT` и `WRITETEXT`.

5.6.1. Чтение больших объемов данных

Для чтения части данных из поля используется оператор READTEXT, который в общем виде выглядит следующим образом:

```
READTEXT { table.column text_ptr offset size } [ HOLDLOCK ]
```

Общий вид немного странноватый, поэтому без хороших примеров не обойтись. Рассмотрим оператор, который позволяет прочитать определенное количество символов:

```
READTEXT Таблица.Поле Указатель Смещение Размер
```

После оператора READTEXT пишем через точку таблицу и поле, данные из которого нужно прочитать. Далее нужно поместить указатель на данные, смещение (символ, начиная с которого нужно читать) и количество читаемых символов. Следующий пример демонстрирует, как прочитать первые 22 символа из поля типа text:

```
DECLARE @ptrval varbinary(20)

SELECT @ptrval = TEXTPTR(tData)
FROM tTextTest
WHERE idTextTest=1

READTEXT tTextTest.tData @ptrval 0 22
```

В первой строчке мы объявляем переменную с именем @ptrval для хранения 20 символов типа varbinary. Следующий запрос SELECT сохраняет указатель на данные типа text в объявленной переменной. Для получения указателя используется функция TEXTPTR(), которой нужно передать имя интересующего нас поля.

Функция TEXTPTR() возвращает указатель на данные типа text, ntext и image. Если запрос возвращает несколько строк, то TEXTPTR() сохранит в переменной @ptrval указатель на данные запрошенного поля из последней строки. Указатель имеет размер 16 байт и представляет собой строку, поэтому длина переменной должна быть достаточной для хранения необходимых данных. Идеальным вариантом будет использование типа varbinary размером в 16 байт, но я в этом случае выделил 20 байт.

Последняя строчка вызывает оператор READTEXT для чтения первых 22 символов, которому мы передаем следующие параметры:

- tTextTest.tData — здесь указаны через точку имя таблицы tTextTest и имя поля tData типа text;

- `@ptrval` — в этой переменной мы сохранили указатель на прочитанные данные;
- 0 — читаются символы, начиная с нулевого;
- 22 — количество читаемых символов.

Обратите внимание, что в качестве начального символа указано число 0. Да, символы в строке нумеруются именно с нуля, и это необходимо учитывать.

Теперь попробуем прочитать оставшиеся символы. Если вы имеете опыт программирования на высокоуровневых языках, то, скорее всего, подумали, что нужно прочитать большое количество символов, начиная с двадцать второго:

```
READTEXT tTextTest.tData @ptrval 22 100
```

В этом примере читается 100 символов, начиная с двадцать второго. Так как у нас в поле `tData` сейчас хранится строка намного меньшего размера, то функция должна была бы вернуть все оставшееся. Но это заблуждение, потому что в Transact-SQL нельзя указывать количество символов больше, чем есть в таблице, поэтому мы увидим следующее сообщение об ошибке:

The offset and length specified in the READTEXT statement is greater than the actual data length of 47.

Смещение и длина, указанные в операторе READTEXT, больше чем 47 — размер текущих данных.

Таким образом, чтобы прочитать остаток данных, необходимо указывать точное количество символов, которое не превысит текущего размера данных:

```
DECLARE @ptrval varbinary(100)

SELECT @ptrval = TEXTPTR(tData)
FROM tTextTest
WHERE idTextTest=1

READTEXT tTextTest.tData @ptrval 22 25
```

Теперь будут прочитаны 25 символов, начиная с двадцать второго. Если сложить выбранное нами смещение 22 с количеством символов 25, то получается число 47, которое равно текущему размеру данных.

5.6.2. Обновление данных

Для обновления данных можно воспользоваться уже рассмотренным ранее оператором `UPDATE`, но у MS SQL Server есть более мощные операторы `UPDATETEXT` и `WRITETEXT`, которые работают быстрее.

Оператор WRITETEXT в общем виде выглядит следующим образом:

```
WRITETEXT { table.column text_ptr }
    [ WITH LOG ] { data }
```

Этот оператор похож на оператор чтения данных. Здесь нужно задать таблицу и поле, которые нужно изменить, указатель на данные и сами данные. Но перед данными есть новая опция — WITH LOG. По идее, без этой опции информация о команде не должна сохраняться в журнале транзакций, и только при наличии WITH LOG журнал узнает об обновлении. Это корректно, но только для MS SQL Server версии 7. Начиная с MS SQL Server 2000, эта опция игнорируется, и необходимость записи в журнал транзакций определяется по используемой модели восстановления.

Следующий пример показывает, как можно изменить данные в первой строке нашей тестовой таблицы:

```
DECLARE @ptrval varbinary(100)

SELECT @ptrval = TEXTPTR(tData)
FROM tTextTest
WHERE idTextTest=1

WRITETEXT tTextTest.tData @ptrval
   'В лесу родилась елочка, в лесу она росла'
```

Оператор WRITETEXT полностью заменяет текущие данные в указанном поле.

Есть еще один оператор, с помощью которого вы можете обновлять данные в поле типа text, ntext или image, — UPDATETEXT. В общем виде он выглядит следующим образом:

```
UPDATETEXT { table_name.dest_column_name dest_text_ptr }
    { NULL | insert_offset }
    { NULL | delete_length }
    [ WITH LOG ]
    [ inserted_data
        | { table_name.src_column_name src_text_ptr } ]
```

Здесь есть возможность задавать позицию, в которую должны вставляться новые данные. Старые данные в указанном месте удаляются. Начало удаляемых данных определяется значением параметра insert_offset. Конец данных определяется параметром delete_length. Если значение этого параметра равно NULL, то будут удалены все данные до конца.

Размер удаляемых данных не обязательно должен быть равен размеру вставляемых. В следующем примере мы заменяем первый символ в поле B на два символа Из:

```
DECLARE @ptrval varbinary(100)

SELECT @ptrval = TEXTPTR(tData)
FROM tTextTest
WHERE idTextTest=1

UPDATETEXT tTextTest.tData @ptrval 0 1   'Из'
```

Таким образом, мы можем простыми запросами UPDATETEXT редактировать большие объемы данных в полях типа text, ntext или image прямо на сервере и не пересылать весь объем информации между клиентом и сервером.

Так как оператор сначала удаляет текст в указанной позиции, то мы можем просто удалить первые два символа следующей командой:

```
UPDATETEXT tTextTest.tData @ptrval 0 2   ''
```

В этом примере мы заменяем первые два символа пустой строкой, что равносильно удалению.

Чтобы вставить строку без удаления, необходимо указать в качестве количества заменяемых символов значение 0. В этом случае новый текст будет просто вставлен в указанную позицию. Следующий пример вставляет букву "B" в начало строки:

```
UPDATETEXT tTextTest.tData @ptrval 0 0   'B'
```

Как мы уже говорили, если в качестве количества символов указать NULL, то будут удалены все символы, начиная со смещения и до конца строки. Например, следующий запрос удаляет все, начиная с двадцать третьего символа:

```
UPDATETEXT tTextTest.tData @ptrval 23 NULL  ''
```

Теперь, когда мы знаем, как использовать курсор и обновлять данные, попробуйте решить задачу, которую мы поставили в самом начале главы — увеличения и уменьшения цены. Я только подскажу логику решения задачи, а реализацию вы должны написать сами:

- объявите курсор, который будет выбирать все цены из таблицы товаров;
- откройте курсор;
- выберите первую запись из курсора;
- запустите цикл перебора всех записей, в котором будут выполняться следующие действия:
  - если цена в текущей строке больше нужного значения, то уменьшаем цену;

- иначе цена меньше нужного значения, то увеличиваем цену;
- выбрать следующую запись.

❒ освободите ресурсы курсора.

Таким образом, задача решится за один просмотр всей таблицы, а значит, двойного изменения цены не будет. Обратите внимание, что проверка цены в цикле должна иметь логику `IF..ELSE`. Если вы напишете просто две проверки цены подряд, то двойное изменение все равно будет возможным:

❒ если цена в текущей строке больше нужного значения, то уменьшаем цену;

❒ если цена меньше нужного значения, то увеличиваем цену.

Теперь посмотрим, что произойдет, если мы сравним цену 10 рублей 50 копеек с числом 10. Первая проверка покажет, что цена больше 10, и ее надо уменьшить. В результате получится 9 рублей 50 копеек. При выполнении второй проверки результат снова будет положительным, и цена увеличится. Проблема в том, что здесь две проверки, и возможно выполнение обоих действий.

При использовании `IF..ELSE` происходит только одна проверка, и может быть выполнено только одно действие, а оба одновременно не будут выполнены никогда.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\textimage.sql на компакт-диске, прилагаемом к книге.

5.7. Курсоры

На мой взгляд, курсоры — достаточно сложная тема, потому что они требуют немного другого взгляда на вещи, особенно необходимо уметь правильно определить ситуацию, когда нужны курсоры. Лично я в первое время не мог увидеть, когда могут пригодиться курсоры, и не мог понять, как их использовать. Чтобы у вас не было таких проблем, приведу классический пример с изменением значения поля в две стороны.

У нас в тестовой базе еще должна была остаться таблица `Товары`, где есть названия товаров и цены. Допустим, что нам нужно уменьшить цену всех товаров на 1 рубль, если цена больше либо равна 15, и увеличить цену, если она меньше 15. Вроде бы простая задача, и мы могли бы написать следующий запрос:

```
BEGIN TRANSACTION

-- Уменьшение цены
```

```
UPDATE Товары
SET Цена=Цена-1
WHERE Цена>=15

-- Увеличение цены
UPDATE Товары
SET Цена=Цена+1
WHERE Цена<15

-- Проверка результата
SELECT *
FROM Товары

ROLLBACK TRANSACTION
```

В этом примере использовались транзакции, дабы изменения не сохранились в базе. Теперь допустим, что цена какого-либо товара равна 15 рублей 50 копеек. Первый запрос `UPDATE` уменьшит цену на 1 рубль, потому что она соответствует условию `Цена>=15`, и она превратится в 14,50. Теперь выполнится второй запрос `UPDATE`, и здесь снова будет выбрана эта же строка, потому что цена товара теперь меньше 15 рублей, и цена будет увеличена на 1 рубль. Получается, что цена товаров от 15 до 16 рублей при выполнении данного запроса не будет изменена.

Курсоры относятся к дополнительным возможностям SQL. Курсоры замедляют выполнение запроса. А так как любое обновление должно выполняться как можно быстрее (чтобы избежать возникновения блокировок), курсоры нужно использовать очень аккуратно.

Логика работы курсоров проста, но количество действий может испугать. Сначала необходимо получить набор данных и связать с ним курсор. Во время создания курсора можно задать его характеристики. Далее идет непосредственно работа и обработка данных. После завершения всех работ необходимо закрыть курсор.

Существуют три типа курсоров:

- динамические курсоры — при перемещении курсора измененные данные сразу же отображаются. Этот тип в наименьшей степени использует ресурсы в клиентской программе и минимально использует временные таблицы на сервере;
- статические курсоры — изменения не отображаются. Весь итоговый набор, связанный с курсором, сохраняется во временной базе данных сервера, и обработка идет непосредственно с ним;

- ключевые курсоры — изменения отображаются, а новые записи — нет. В этом типе курсора во временной базе данных сохраняются только ключи (отсюда и такое название). Новые записи в курсор не включаются, а при обращении к удаленной записи происходит ошибка.

В зависимости от типа курсора работа сервера с курсором имеет свои особенности, но в большинстве случаев это работа с временной таблицей в базе данных tempdb. В этой таблице может быть полный снимок данных, а могут быть только ключевые поля выбранных курсором данных.

5.7.1. Объявление курсора

Для объявления курсора используется оператор `DECLARE CURSOR`. В стандарте SQL-92 этот оператор выглядит следующим образом:

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
```

Но Transact-SQL предоставляет больше возможностей, поэтому тут оператор был расширен за счет добавления новых свойств курсора, которые могут повысить эффективность их использования. В общем виде оператор `DECLARE CURSOR` в Transact-SQL выглядит следующим образом:

```
DECLARE cursor_name CURSOR
[ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
```

Давайте, как всегда, сначала рассмотрим параметры:

- `LOCAL` — указывает на то, что курсор должен быть локальным по отношению к процедуре, функции, триггеру или пакету, в котором он выполняется, по завершении выполнения которого курсор автоматически будет уничтожен. Если курсор указан как выходной параметр из процедуры, где он был объявлен, то он будет уничтожен, когда уничтожится последняя переменная, ссылающаяся на курсор. Например, вы вызываете процедуру, которая создает локальный курсор и возвращает его вам. Несмотря на то, что он был создан в процедуре и должен был бы уничтожиться, курсор остается доступным, пока на него ссылаются переменные;

- GLOBAL — глобальный курсор доступен в течение всего подключения к MS SQL Server, или пока его явно не освободят;
- FORWARD_ONLY — записи можно просматривать только в прямом направлении — от начала до конца. По умолчанию такой курсор является динамическим, если не указано другое;
- STATIC — статический курсор, который доступен только для чтения;
- DYNAMIC — динамический курсор;
- READ_ONLY — только для чтения;
- KEYSET — ключевой курсор;
- INSENSITIVE — создать копию выбираемых данных во временной таблице базы данных tempdb;
- SCROLL_LOCKS — операции обновления и удаления с использованием WHERE CURRENT гарантированно будут успешными, а блокировки будут доступны до выбора следующей записи;
- OPTIMISTIC — успешность обновления с помощью WHERE CURRENT не гарантируется, и блокировки не сохраняются;
- FOR READ ONLY — обновление командой WHERE CURRENT недоступно;
- FOR UPDATE — для обновления. Этот параметр используется по умолчанию;
- COLUMN_LIST — список обновляемых полей.

Следующий код содержит простейший пример объявления динамического курсора:

```
DECLARE курсор_для_товаров CURSOR
  DYNAMIC
FOR
 SELECT *
 FROM Товары
FOR UPDATE
```

Между ключевыми словами DECLARE и CURSOR находится имя курсора. Оно должно быть уникальным, и желательно, чтобы оно отражало смысл выполняемых действий или выборки данных, которая будет связана с курсором. Вы можете создать с одинаковым именем не более двух курсоров, при этом один из них должен быть глобальным, а другой локальным. Одно и то же имя для курсоров одинакового типа указывать нельзя.

5.7.2. Открытие курсора

После объявления курсора его необходимо открыть. Для этого используется команда OPEN, которая в общем виде выглядит следующим образом:

```
OPEN { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

У команды только одна необязательная опция — GLOBAL, которая означает, что открываться должен глобальный курсор. Эта опция обязательно должна указываться, если существуют глобальный и локальный курсоры с одинаковыми именами.

Следующий пример создает курсор с именем Person_Curs и открывает его:

```
DECLARE Person_Curs CURSOR
FOR
 SELECT *
 FROM tbPeoples
 WHERE vcName LIKE 'A%'
FOR UPDATE
OPEN Person_Curs
```

Если курсор создан с параметром INSENSITIVE или STATIC, то количество записей в итоговом наборе курсора должно быть меньше количества записей в таблице.

После открытия курсора можно воспользоваться переменной @@CURSOR_ROWS, чтобы определить количество записей, попавших в результирующий набор.

5.7.3. Выборка записей из курсора

Когда мы открыли курсор, сервер выбрал данные из базы данных, и мы можем читать их. Для этого используется команда FETCH. В общем виде команда выглядит следующим образом:

```
FETCH
        [ [ NEXT | PRIOR | FIRST | LAST
                | ABSOLUTE { n | @nvar }
                | RELATIVE { n | @nvar }
            ]
            FROM
        ]
{ { [ GLOBAL ] cursor_name } | @cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]
```

Параметров немного, но рассмотреть их надо. Если при объявлении курсора вы указали свойство SCROLL, то можно указывать следующие параметры:

- NEXT — если это первое обращение к команде, то она вернет первую строку, иначе происходит перемещение на следующую строку, которая и будет возвращена;
- PRIOR — если это первое обращение к команде, то ничего не возвращается, а курсор будет позиционирован перед первой строкой, иначе будет возвращена текущая строка, а курсор переместится на предыдущую;
- FIRST — возвратить первую строку в курсоре и сделать ее текущей;
- LAST — возвратить последнюю строку в курсоре и сделать ее текущей;
- ABSOLUTE n/-n — если n является положительным числом, то курсор перемещается к n-ой записи от начала, иначе к n-ой записи от конца набора данных курсора;
- RELATIVE n/-n — если n является положительным числом, то курсор перемещается на n-ой записи от текущей к концу набора, иначе на n-ой записи от текущей позиции к началу набора данных курсора.

Если курсор создавался без параметра SCROLL, то в качестве параметров оператора FETCH можно указывать только NEXT, потому что в этом случае разрешено путешествие по курсору только от начала в конец.

Простейший вариант вызова команды FETCH для чтения следующей строки выглядит так:

```
FETCH NEXT
FROM ИмяКурсора
```

При этом данные строки возвращаются в виде таблицы значений из одной (текущей) строки. Вы можете выбрать данные не в виде таблицы, а в виде переменных. Для этого необходимо объявить переменные и в операторе FETCH указать опцию INTO и перечислить переменные в таком порядке, в каком находятся соответствующие поля в результирующем наборе курсора. При этом типы переменных и соответствующих полей должны совпадать. Следующий пример показывает, как выбрать данные из курсора в переменные:

```
FETCH NEXT
FROM ИмяКурсора
INTO @Переменная1, @Переменная2, @Переменная3...
```

В листинге 5.3 вы можете увидеть пример объявления, открытия курсора и чтения данных из набора данных курсора.

Листинг 5.3. Пример чтения из курсора

```
DECLARE курсор_для_товаров CURSOR
  SCROLL
FOR
 SELECT *
 FROM Товары

OPEN курсор_для_товаров

--  Так как это первое обращение к FETCH,
-- то будет возвращена первая строка
FETCH NEXT FROM курсор_для_товаров

-- Читаем следующую строку
FETCH NEXT FROM курсор_для_товаров

DECLARE @date datetime
DECLARE @name varchar(50)
DECLARE @cost float
DECLARE @number int

-- Читаем данные строки переменные
FETCH NEXT FROM курсор_для_товаров
INTO @date, @name, @cost, @number

PRINT @name
```

Если уж мы выбираем данные из набора, в котором несколько строк, то хорошо бы было иметь возможность перебора всех записей. Можно реализовать цикл, который будет перебирать все записи, пока не выберем все строки, количество которых мы можем определить через глобальную переменную `@@CURSOR_ROWS`. Но эта переменная возвращает количество строк последнего открытого курсора, а если у вас их будет несколько?

На мой взгляд, намного удобнее использовать цикл `WHILE`, который будет выполняться, пока последняя команда обращения к курсору выполнена

успешно. Для определения статуса выполнения команды FETCH есть глобальная переменная @@FETCH_STATUS. Она может содержать следующие значения:

- 0 — выборка прошла успешно;
- –1 — выборка неудачна из-за выхода за границы набора;
- –2 — выборка неудачна из-за отсутствия записи в результирующем наборе.

В листинге 5.4 показан пример использования цикла WHILE для перебора всех записей результирующего набора курсора.

Листинг 5.4. Перебор всех записей курсора

```
DECLARE курсор_для_товаров1 CURSOR
  SCROLL
FOR
 SELECT *
 FROM Товары

OPEN курсор_для_товаров1

--  Так как это первое обращение к FETCH,
-- то будет возвращена первая строка
DECLARE @date datetime
DECLARE @name varchar(50)
DECLARE @cost float
DECLARE @number int

FETCH NEXT
FROM курсор_для_товаров1
INTO @date, @name, @cost, @number

-- Запускаем цикл
WHILE @@FETCH_STATUS = 0
 BEGIN
  PRINT @name

  FETCH NEXT
  FROM курсор_для_товаров1
  INTO @date, @name, @cost, @number
END
```

Обратите внимание, что до начала цикла мы считываем одну строку. Дело в том, что в цикле происходит проверка переменной `@@FETCH_STATUS`. Если предварительного чтения не будет, то переменная может содержать неверное значение. А вот вывод результата на экран происходит только после проверки значения переменной `@@FETCH_STATUS`.

Переменную `@@FETCH_STATUS` желательно проверять всегда. В листинге 5.3 мы читали данные и выводили их на экран без проверки. Это некорректно, ведь может возникнуть ситуация, когда выборка данных не возвратит строк, и вызов оператора `PRINT` будет бессмысленным. После любого вызова оператора `FETCH` необходимо производить проверку корректности получения строки:

```
FETCH NEXT
FROM курсор_для_товаров
INTO @date, @name, @cost, @number

IF @@FETCH_STATUS = 0
 BEGIN
  PRINT @name
  -- Обработка данных
 END
```

Вот теперь работа с курсором более корректна.

5.7.4. Закрытие курсора

Теперь нам нужно научиться освобождать ресурсы. Если этого не сделать, то повторный вызов сценариев без завершения соединения может вызвать проблемы, если SQL Server не освободит ресурсы автоматически. Именно из-за этого в листингах 5.3 и 5.4 я дал курсорам разные имена, чтобы не возникло проблем во время их тестирования.

Для закрытия курсора используется оператор `CLOSE`. После этого курсор переходит в неактивное состояние. В общем виде команда выглядит следующим образом:

```
CLOSE { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Здесь все нам уже знакомо. Параметр `GLOBAL` указывает на глобальность курсора, а после этого идет имя курсора.

Следующий пример закрывает курсор `курсор_для_товаров1`:

```
CLOSE курсор_для_товаров1
```

После деактивации курсор недоступен, но ресурсы еще не освобождены. Для освобождения ресурсов используется оператор DEALLOCATE:

```
DEALLOCATE { { [ GLOBAL ] cursor_name } | @cursor_variable_name }
```

Этот оператор похож на закрытие с помощью оператора CLOSE. Следующий пример освобождает ресурсы курсора курсор_для_товаров1:

```
DEALLOCATE курсор_для_товаров1
```

После выполнения оператора DEALLOCATE удаляются временные таблицы.

Написать операторы закрытия и освобождения ресурсов курсора не сложно, поэтому пишите их явно и не дожидайтесь автоматического удаления. Если ресурсы не нужны, то зачем заставлять SQL Server хранить их? Пусть сервер удалит ненужные временные таблицы, освободит память и продолжает в полную силу обрабатывать ваши запросы.

5.7.5. Изменение данных в курсоре

Теперь познакомимся с редактированием. Для этого используется уже знакомый нам оператор UPDATE, только при этом необходимо указывать опцию CURRENT OF. Минимальная команда обновления будет выглядеть следующим образом:

```
UPDATE имя таблицы
SET поле=значение
WHERE CURRENT OF курсор
```

Вы можете и удалять данные из курсоров. Для этого также нужно указать ключевое слово CURRENT OF. Следующий пример показывает, как могут быть удалены записи:

```
DELETE имя таблицы
WHERE CURRENT OF курсор
```

Обновление происходит достаточно интересно, поэтому давайте рассмотрим его на примере. В листинге 5.5 приводится код изменения цены хлеба.

Листинг 5.5. Пример обновления данных курсора

```
-- Объявление курсора
DECLARE курсор_для_товаров1 CURSOR
FOR
 SELECT *
 FROM Товары
 WHERE [Название товара]='Хлеб'
```

```
FOR UPDATE

-- Открытие курсора
OPEN курсор_для_товаров1

-- Выборка записи из курсора
FETCH NEXT
FROM курсор_для_товаров1

-- Обновление
IF @@FETCH_STATUS = 0
 BEGIN
  UPDATE Товары
  SET Цена=10
  WHERE CURRENT OF курсор_для_товаров1
 END

-- Закрыть курсор
CLOSE курсор_для_товаров1
DEALLOCATE курсор_для_товаров1
```

Запрос курсора выбирает все записи, в которых товар имеет название `Хлеб`. Именно эти записи мы будем изменять. В нашей тестовой таблице должно быть три записи о покупке хлеба, и у всех должна быть цена 7,5. Попробуйте выполнить обновление через курсор из листинга 5.5 и посмотрите, что произошло в таблице:

```
SELECT *
FROM Товары
WHERE [Название товара]='Хлеб'
```

Самое интересное, что изменилась только одна запись — самая первая. А ведь запрос на обновление `UPDATE` не делал никаких условий, кроме `CURRENT OF`, и по идее, должны были обновиться все записи, но этого не произошло.

Если обновление происходит в курсоре, то обновляется та запись, которая выбрана с помощью оператора `FETCH`. Все остальные в обновлении не участвуют. При этом, если с помощью `FETCH` не выбрана ни одна из записей, то обновление станет невозможным, а вы получите сообщение об ошибке, что необходимо выбрать запись из курсора:

There are no rows in the current fetch buffer. The statement has been terminated.

(Нет записей в текущем fetch буфере. Оператор был прерван)

При обновлении вы должны учитывать, что обновляться может только одна таблица. Даже если курсор выбирает данные из нескольких связанных таблиц, изменения могут происходить только в одной из них.

В листинге 5.6 показан пример удаления данных из курсора. Если вы сравните этот листинг с примером редактирования из листинга 5.5, то вы увидите, что они очень сильно похожи. Разница только в том, что мы вызываем оператор `DELETE`. При удалении вы также должны выполнять оператор с опцией `CURRENT OF`, при этом удаляется только запись, которая находится в буфере. Это значит, что сначала необходимо выбрать запись из курсора, которую надо удалить.

Листинг 5.6. Удаление записей из курсора

```
-- Объявление курсора
DECLARE курсор_для_товаров1 CURSOR
FOR
 SELECT *
 FROM Товары
 WHERE [Название товара]='Хлеб'
FOR UPDATE

-- Открытие курсора
OPEN курсор_для_товаров1

-- Выборка записи
FETCH NEXT
FROM курсор_для_товаров1

-- Удаление
IF @@FETCH_STATUS = 0
 BEGIN
  DELETE Товары
  WHERE CURRENT OF курсор_для_товаров1
 END

-- Закрыть курсор
CLOSE курсор_для_товаров1
DEALLOCATE курсор_для_товаров1
```

Удалять можно только из одной таблицы. Если курсор выбирает данные из нескольких связанных таблиц, то удаление становится недоступным. Сервер просто не сможет определить, из какой таблицы необходимо удалить запись.

Обратите внимание, что перед обновлением или удалением мы проверяем состояние переменной `@@FETCH_STATUS`. Если этого не делать и в результирующем наборе курсора не окажется записей, то мы столкнемся с нежелательной ошибкой. Чтобы ошибок не было, лучше лишний раз проверить, а в случае необходимости можно сгенерировать ошибку самостоятельно, мы уже умеем это делать.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\cursors.sql на компакт-диске, прилагаемом к книге.

5.8. Полнотекстовый поиск

Название "полнотекстовый поиск" не очень удачно, потому что по нему очень трудно определить, для чего нужна эта технология. Но так как никто не смог придумать более удачного термина, придется и нам пользоваться им. Полнотекстовый поиск чем-то похож на поисковую машину в Интернете. Вы задаете слова, которые нужно найти, а сервер ищет их по всей таблице внутри указанных полей. Имея в руках такой мощный инструмент, достаточно просто написать простую поисковую машину. Правда поисковые системы типа **yandex.ru**, **google.com** и др. не ограничиваются только поиском слов, поэтому, если быть объективным, то одного полнотекстового поиска недостаточно для создания системы поиска в Интернете.

Полнотекстовый поиск призван оптимизировать поиск по текстовым полям типа `char`, `nchar`, `varchar` или `nvarchar`. Итак, допустим, что вы хотите найти слово "Сок" в таблице товаров. Но соки бывают разные, и это может быть "Сок Апельсиновый" или просто "Сок". Чтобы решить эту проблему, можно написать запрос следующим образом:

```
SELECT *
FROM Товары
WHERE [Название товара] LIKE 'Сок%'
```

Если нужно найти товары, названия которых идут после слова "Сок", то запрос может выглядеть следующим образом:

```
SELECT *
FROM Товары
WHERE [Название товара] >= 'Сок'
```

В результат этого запроса попадут все соки, а также товары, названия которых начинаются с букв от "Т" до "Я".

Оба запроса будут выполнены достаточно быстро, если поле `Название товара` проиндексировано. Так как в искомом шаблоне начало задано жестко, и только окончание может изменяться, оптимизатор MS SQL Server может принять решение использовать существующий индекс, что значительно повысит производительность.

А что, если в таблице есть соки, названия которых записаны так: "Ананасовый Сок". Здесь уже слово "Сок" стоит не в начале, и для поиска таких строк нужно использовать запрос следующего вида:

```
SELECT *
FROM Товары
WHERE [Название товара] LIKE 'Сок%'
```

Когда начало шаблона является неопределенным (стоят символы подчеркивания или процента), то сервер никогда не будет использовать индекс, потому что он не сможет помочь при поиске данных. Индекс имеет древовидную структуру и для путешествия по нему необходимо знать первые буквы искомого слова.

Без использования индекса серверу приходится сканировать всю таблицу. Это достаточно накладно. Если запрос должен вернуть 10 записей из миллиона, то с помощью индекса эти записи найти можно как минимум в 100 раз быстрее, чем с помощью сканирования.

Если простой индекс сортирует данные полей, как единое целое, то полнотекстовый поиск индексирует слова и фразы внутри поля. Текстовые поля могут иметь очень большой размер, и из-за этого на них нельзя создавать классический индекс, зато полнотекстовый — без проблем.

Как и при использовании простого индекса, для полнотекстового поиска также нужно создавать индекс, потому что автоматически его создавать никто не будет. Этот процесс состоит из следующих этапов:

1. Включение полнотекстового поиска.
2. Построение каталога для индекса.
3. Регистрация таблиц.
4. Регистрация столбцов.
5. Активизация таблиц.
6. Запуск полнотекстового поиска.

5.8.1. Включение поиска

Необходимо отметить, что полнотекстовый поиск реализован в качестве отдельного компонента, который должен быть установлен во время инсталляции ОС. Если вы этого не сделали, то выполнение последующих команд может привести к ошибке, а значит, вы не сможете протестировать примеры, и придется верить мне на слово.

Прежде чем использовать полнотекстовый поиск, его нужно сначала разрешить. Для этого используется хранимая процедура `sp_fulltext_database`. В качестве параметра ей нужно передать `enable` (разрешить) или `disable` (запретить). Следующий пример разрешает полнотекстовый поиск:

```
EXEC sp_fulltext_database 'enable'
```

При включении полнотекстового режима создается каталог индексов. Будьте аккуратны при выполнении команды, потому что, если каталог уже существует, то он будет уничтожен и создан новый.

Чтобы проверить, включен ли сейчас полнотекстовый поиск, необходимо выполнить следующую команду:

```
SELECT DATABASEPROPERTY ('Имя базы', 'IsFullTextEnabled')
```

Если не включить полнотекстовый поиск, то все остальные команды этой главы работать не будут.

5.8.2. Создание каталога

Для создания каталога для полнотекстового поиска используется процедура `sp_fulltext_catalog`. Этот каталог является ничем иным, как файлом, в котором будет находиться индекс. В общем виде эта процедура выглядит следующим образом:

```
sp_fulltext_catalog [ @ftcat = ] 'fulltext_catalog_name' ,
    [ @action = ] 'action'
    [ , [ @path = ] 'root_directory' ]
```

Здесь три параметра. Давайте их рассмотрим:

- `fcat` — имя каталога, которое должно быть уникальным для каждой базы данных;
- `action` — действие, которое необходимо выполнить. Здесь можно указывать одно из следующих значений:
  - `create` — создать новый пустой каталог для полнотекстового поиска в файловой системе;

- drop — удалить каталог из системы. При этом команда завершится ошибкой, если в каталоге есть индекс хотя бы на одну таблицу;
- start_incremental — начать заполнение полнотекстового индекса для изменившихся записей. В индексируемой таблице должно быть поле типа TIMESTAMP;
- start_full — начать заполнение каталога для всех записей таблицы;
- stop — остановить заполнение индекса;
- rebuild — перестроить полнотекстовый индекс;

❒ path — необязательный параметр, в котором можно указать путь к нужному каталогу, так что имени файла здесь быть не должно. Если параметр не указан, то будет использоваться значение по умолчанию.

Следующий пример создает каталог с именем Каталог_Товаров:

```
EXEC sp_fulltext_catalog 'Каталог_Товаров', 'create'
```

5.8.3. Регистрация таблиц

Для регистрации таблиц используется хранимая процедура sp_fulltext_table. В общем виде она выглядит следующим образом:

```
sp_fulltext_table [ @tabname = ] 'qualified_table_name'
    , [ @action = ] 'action'
    [ , [ @ftcat = ] 'fulltext_catalog_name'
    , [ @keyname = ] 'unique_index_name' ]
```

Здесь четыре параметра:

❒ @tabname — имя таблицы, которая должна уже существовать в базе данных;

❒ @action — параметр определяет выполняемое действие, где можно указывать одно из следующих значений:

- create — создать метаданные для полнотекстового индекса указанной таблицы. Чтобы команда завершилась успешно, необходимо, чтобы в таблице был хотя бы один уникальный индекс, но это не проблема, если у вас есть ключевое поле;
- drop — удалить метаданные полнотекстового индекса указанной таблицы. Перед удалением необходимо деактивировать таблицы;
- activate — активировать для указанной таблицы сбор данных для полнотекстового поиска. При этом у таблицы должно быть хотя бы одно поле, участвующее в индексе;

- `deactivate` — запретить использование существующего полнотекстового индекса. Никакие данные при этом не уничтожаются, и индекс продолжает существовать, но он не активен и не используется;
- `start_change_tracking` — начать сбор информации об измененных данных. Если в таблице нет поля типа `TIMESTAMP`, то необходимо выполнить полный сбор;
- `stop_change_tracking` — остановить сбор информации об изменениях;
- `update_index` — обновить индекс;
- `start_background_updateindex` — запустить сбор данных в фоновом режиме и обновление индекса в момент изменений;
- `stop_background_updateindex` — остановить сбор данных в фоновом режиме;
- `start_full` — запустить полное обновление индекса;
- `stop` — остановить создание индекса;

❒ `@ftcat` — существующий каталог для полнотекстового поиска. Параметр необходим при регистрации таблицы (когда в параметре `@action` указано `create`). Для остальных действий этот параметр должен быть нулевым;

❒ `@keyname` — имя уникального индекса, который должен состоять из одного поля. Параметр необходим при регистрации таблицы (когда в параметре `@action` указано `create`). Для остальных действий этот параметр должен быть нулевым.

Следующий пример показывает, как зарегистрировать таблицу в каталоге полнотекстового поиска:

```
EXEC sp_fulltext_table 'Товары', 'create',
    'Каталог_товаров', 'PK_Key'
```

Следующий пример показывает, как активировать полнотекстовый поиск:

```
EXEC sp_fulltext_table 'Товары', 'activate'
```

Но не стоит сейчас пытаться что-то активировать, потому что для этого нужно сначала добавить в поиск хотя бы одно поле.

5.8.4. Регистрация полей

Как мы уже выяснили в *разд. 5.8.3*, для того, чтобы активировать полнотекстовый поиск в таблице, нужно добавить хотя бы одну колонку. Это вполне логично, ведь должен же происходить поиск по какой-либо из колонок.

Для добавления колонки используется хранимая процедура `sp_fulltext_column`, которая в общем виде выглядит так:

```
sp_fulltext_column [ @tabname = ] 'qualified_table_name' ,
    [ @colname = ] 'column_name' ,
    [ @action = ] 'action'
    [ , [ @language = ] 'language' ]
    [ , [ @type_colname = ] 'type_column_name' ]
```

Тут все достаточно просто по сравнению с тем, через что нам уже пришлось пройти:

- `@tabname` — имя таблицы;
- `@colname` — имя поля;
- `@action` — действие. Здесь может быть одно из двух значений:
  - `add` — добавить поле к полнотекстовому поиску;
  - `drop` — удалить указанное поле из поиска;
- `@language` — язык (возможные значения этого параметра показаны в табл. 5.3). Параметр необязательный, а по умолчанию будет использоваться нейтральный язык Neutral (с идентификатором 0);

Таблица 5.3. *Языки и соответствующие константы для текстовых полей*

Локализация	Идентификатор
Neutral	0
Chinese_Simplified	0x0804
Chinese_Traditional	0x0404
Dutch	0x0413
English_UK	0x0809
English_US	0x0409
French	0x040c
German	0x0407
Italian	0x0410
Japanese	0x0411
Korean	0x0412
Spanish_Modern	0x0c0a
Swedish_Default	0x041d

- `@type_colname` — название типа поля. Здесь можно указать одно из следующих значений: `char`, `nchar`, `varchar` или `nvarchar`.

Следующий пример показывает, как добавить поле `Название товара` к полнотекстовому поиску:

```
EXEC sp_fulltext_column 'Товары',
    'Название товара' , 'add'
```

5.8.5. Информация о каталоге

Мы уже рассмотрели необходимые нам хранимые процедуры, а теперь пришло время увидеть все на практике. В листинге 5.7 содержит код, необходимый для создания, описания и активации полнотекстового поиска по полю `Название товара` таблицы `Товары`.

Листинг 5.7. Пример создания полнотекстового поиска

```
-- Разрешить полнотекстовый поиск
EXEC sp_fulltext_database 'enable'

-- Проверить, включен ли полнотекстовый поиск
SELECT DATABASEPROPERTY ('FlenovSQLBook', 'IsFullTextEnabled')

-- Создание каталога
EXEC sp_fulltext_catalog 'Каталог_Товаров', 'create'

--
EXEC sp_fulltext_table 'Товары', 'create',
    'Каталог_товаров', 'PK_Key'

-- Добавление поля к полнотекстовому поиску
EXEC sp_fulltext_column 'Товары', 'Название товара' , 'add'

-- Активирование
EXEC sp_fulltext_table 'Товары','activate'

-- Запуск сбора индекса
EXEC sp_fulltext_catalog 'Товары','start_full'
```

После всех подготовительных действий мы запустили формирование индекса с помощью процедуры `sp_fulltext_catalog` с параметром `start_full`.

Несмотря на то, что запрос завершится удачно, сервер продолжит работать и собирать метаданные об индексе. Пока сервер формирует индекс, пользоваться им нежелательно.

Чтобы узнать, работает ли сервер сейчас, выполните команду:

```
SELECT fultextcatalogproperty('имя_каталога',
    'PopulateStatus')
```

Для созданного ранее каталога с именем `Каталог_Товаров` эта команда будет выглядеть так:

```
SELECT fultextcatalogproperty('Каталог_Товаров',
    'PopulateStatus')
```

Но создать индекс мало. Он не будет использоваться, если вы просто напишете простой запрос `SELECT`, потому что серверу нужно сообщить о необходимости воспользоваться такой удобной и мощной возможностью.

Если необходимо узнать, входит ли запрос в полнотекстовый поиск, можно выполнить следующий запрос `SELECT`:

```
SELECT ColumnProperty (ObjectID('имя_таблицы'),
   'Имя поля', IsFullTextIndexed)
```

Если результат равен 1, то поле входит в полнотекстовый поиск, иначе (если результат равен 0) — нет.

С помощью хранимой процедуры `sp_help_fulltext_catalogs` можно получить информацию об указанном в качестве единственного параметра каталоге. В общем виде процедура выглядит следующим образом:

```
sp_help_fulltext_catalogs [ @fulltext_catalog_name = ]
    'fulltext_catalog_name'
```

Процедура `sp_help_fulltext_tables` позволяет получить информацию о зарегистрированных в полнотекстовом поиске таблицах:

```
sp_help_fulltext_tables
    [ [ @fulltext_catalog_name = ] 'fulltext_catalog_name' ]
    [ , [ @table_name = ] 'table_name' ]
```

В параметре `@fulltext_catalog_name` указывается имя интересующего вас каталога, а в параметре `@table_name` — имя таблицы. Если таблица зарегистрирована, то в результирующей таблице вы увидите информацию о таблице. Если параметр `@table_name` не указан, то в результирующей таблице будет информация обо всех таблицах указанного каталога.

Для получения информации о том, какие поля задействованы в полнотекстовом поиске, можно воспользоваться хранимой процедурой sp_help_fulltext_columns:

```
sp_help_fulltext_columns [ [ @table_name = ] 'table_name' ] ]
    [ , [ @column_name = ] 'column_name' ]
```

Здесь у нас два параметра — имя таблицы и имя поля. Если имя поля не указано, то будут отображены все поля таблицы, участвующие в полнотекстовом поиске.

5.8.6. Использование поиска

Мы уже научились конфигурировать полнотекстовый поиск, и теперь пора бы уже научиться им пользоваться. Несмотря на то, что поиск выполняется с помощью уже знакомого нам запроса SELECT, тут есть кое-какие нововведения, а точнее, два ключевых слова: CONTAINS и FREETEXT. Рассмотрим каждое из них.

С помощью CONTAINS мы можем задать поиск слова или фразы в зарегистрированном для полнотекстового поиска поле. Если мы ищем одно слово, то оно может заключаться в кавычки или нет, но для целой фразы кавычки являются обязательными.

Общий вид команды CONTAINS показан в листинге 5.8.

Листинг 5.8. Общий вид команды CONTAINS

```
CONTAINS
    ( { column | * } , '< contains_search_condition >'
    )

< contains_search_condition > ::=
        { < simple_term >
        | < prefix_term >
        | < generation_term >
        | < proximity_term >
        | < weighted_term >
        }
        | { ( < contains_search_condition > )
        { AND | AND NOT | OR } < contains_search_condition > [ ...n ]
        }

< simple_term > ::=
    word | " phrase "

< prefix term > ::=
```

```
    { "word * " | "phrase * " }

< generation_term > ::=
    FORMSOF ( INFLECTIONAL , < simple_term > [ ,...n ] )

< proximity_term > ::=
    { < simple_term > | < prefix_term > }
    { { NEAR | ~ } { < simple_term > | < prefix_term > } } [ ...n ]

< weighted_term > ::=
    ISABOUT
        ( { {
                < simple_term >
                | < prefix_term >
                | < generation_term >
                | < proximity_term >
                }
            [ WEIGHT ( weight_value ) ]
            } [ ,...n ]
        )
```

Общий вид команды CONTAINS достаточно объемистый, но если ограничиться самыми важными элементами ее синтаксиса, то все описывается в первых трех строчках:

```
CONTAINS
 (
   { column | * } , '< contains_search_condition >'
 )
```

Все остальное — это дополнительные возможности (FORMSOF, ISABOUT и т. д.), используемые совместно с CONTAINS. Они позволяют более тонко настроить поиск. Их нам предстоит изучить в этом разделе, так что скоро все встанет на свои места.

После ключевого слова CONTAINS в скобках указываются два параметра: имя поля, в котором будет происходить поиск, и искомая фраза или слово. Следующий пример показывает, как можно найти все соки в таблице товаров:

```
SELECT *
FROM [Товары]
WHERE CONTAINS([Название товара], 'Сок')
```

Так как мы ищем только одно слово, то его не обязательно было заключать в кавычки, но я всегда ставлю их и вам рекомендую.

Следующий пример ищет в полнотекстовом индексе фразу:

```
SELECT *
FROM [Товары]
WHERE CONTAINS([Название товара], 'Сок Апельсиновый')
```

В полнотекстовом индексе можно использовать и шаблоны слов, правда, они сильно ограничены по сравнению с шаблонами, которые мы использовали с оператором `LIKE`. Шаблоны полнотекстового поиска должны начинаться с букв, и только после этого можно указать символ звездочки (*), который означает любые символы. Звездочку нельзя указывать в начале шаблона, потому что в этом случае индекс не может использоваться.

Следующий пример показывает, как найти все слова на букву "С":

```
SELECT *
FROM [Товары]
WHERE CONTAINS([Название товара], 'С*')
```

Помните, что при полнотекстовом поиске вместо символа процента (%) для указания шаблона необходимо использовать звездочку. Такая разница была сделана, видимо, из-за того, что смысл у символов немного разный. Процент может ставиться в любом месте строки, а звездочка — только в конце шаблона и означает, что возможно любое окончание указанной строки.

Но на этом возможности полнотекстового поиска не заканчиваются. Вы можете искать не только фразы из нескольких слов, идущих подряд, но и близлежащие слова. Следующий пример ищет любые строки, в которых слова "Сок" и "Апельсиновый" встречаются рядом:

```
SELECT *
FROM [Товары]
WHERE CONTAINS
(
  [Название товара],
  'Сок near() Апельсиновый'
)
```

В данном примере в искомой фразе между словами "Сок" и "Апельсиновый" стоит вызов процедуры `near()`. Это значит, что между этими словами может быть некоторое количество слов и в результат попадут строки со следующими названиями товаров: "Сок 0.2 литра (Апельсиновый)", "Сок элитный Апельсиновый" и т. д.

В свое время поисковая система **yandex.ru** завоевала большую популярность за счет того, что позволяла искать слова с учетом грамматических форм. Например, если запросить поиск слова "Сок", поисковая система найдет все страницы в Интернете, где есть это слово и его различные формы (множественное число, падежи и т. д.), а значит, в результат попадут "Сок", "Соки", "Соку" и т. д. Благодаря таким гибким возможностям качество поиска действительно улучшилось.

В MS SQL Server есть подобная возможность поиска слов с учетом различных грамматических форм. Функция `FORMSOF()` ищет единственное или множественное число указанного существительного или глагола. В следующем примере мы ищем различные формы слова "сок":

```
SELECT *
FROM [Товары]
WHERE CONTAINS
(
  [Название товара],
  FORMSOF(INFLECTIONSL, 'Сок')
)
```

Когда мы ищем какой-либо текст в поисковой системе, то чтобы сократить количество результатов, стараемся задать несколько слов, но при этом наибольшей значимостью обладают только несколько из них (чаще всего не более двух). Функция `ISABOUT()` позволяет задать весовой коэффициент каждого используемого вами слова. Эта процедура в общем виде выглядит следующим образом:

```
ISABOUT('фраза', WEIGHT(вес))
```

Вес фразы может задаваться в качестве дробного числа от 0 до 1.

Например, следующий запрос ищет записи со словами "Сок" и "Апельсиновый", но при этом слову "Сок" дается больший коэффициент:

```
SELECT *
FROM [Товары]
WHERE CONTAINS
(
  [Название товара],
  'ISABOUT(Сок, weight(1), Апельсиновый, weight(0.5)')
)
```

В данном примере слово "Сок" имеет коэффициент 1, а слово "Апельсиновый" — в два раза меньшее значение. Нам нужны соки, а слово "Апельсиновый" — это название, которое не отображает суть искомого. Если ему дать

больший коэффициент, то в результате мы получим документы, в которых говорится об апельсине, а сок может упоминаться вскользь.

Допустим, что нужно найти все строки, где в названии товара упоминается сок, но при этом нет слова "Апельсиновый". Тогда можно воспользоваться одним из логических операторов (AND, OR и AND NOT). В данном случае нам понадобится оператор AND NOT. Посмотрим на следующий запрос:

```
SELECT *
FROM [Товары]
WHERE CONTAINS([Название товара], 'Сок AND NOT Апельсиновый')
```

Эта логика на русском языке звучит так: "найти все из таблицы товаров, где в поле [название товара] содержится Сок и не Апельсиновый".

Усложним задачу — найдем все соки, кроме Апельсинового и Вишневого:

```
SELECT *
FROM [Товары]
WHERE CONTAINS
(
  [Название товара],
  'Сок AND NOT (Апельсиновый OR Вишневый)'
)
```

Таким образом, можно строить достаточно сложную логику поиска необходимых строк. Однажды мне пришлось разрабатывать простую систему документооборота, и функция поиска оказалась незаменимой, т. к. позволяла искать документы по содержащимся в них словам.

Но мы совсем забыли про оператор FREETEXT, который также используется при полнотекстовом поиске. Этот оператор выглядит следующим образом:

```
FREETEXT ( { column | * } , 'freetext_string' )
```

Первый параметр — это поле, в котором должен производиться поиск, а второй параметр — искомый текст. Например:

```
SELECT *
FROM [Товары]
WHERE FREETEXT ([Название товара], 'Сок Апельсиновый')
```

Как видите, использование оператора схоже с CONTAINS, но результат отличается. Оператор FREETEXT возвращает все строки, в которых были найдены слова, похожие на перечисленные во втором параметре.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter5\fulltext.sql на компакт-диске, прилагаемом к книге.

Глава 6

Transact-SQL и 1С

Очень популярные в нашей стране программы 1С позволяют решать большой круг бизнес-задач. Они используются в любых организациях, начиная с частных предпринимателей и заканчивая большими корпорациями. В данной главе, говоря "1С", мы будем подразумевать 1С:Предприятие, но некоторые вопросы из рассматриваемого материала можно в равной степени отнести и к любому другому программному продукту, который работает совместно с сервером баз данных MS SQL Server.

Есть мнение, что продукты 1С предназначены только для решения небольших задач, но существует немало реальных примеров, когда с помощью этой системы решались достаточно сложные производственные задачи. Я видел, как на некоторых предприятиях, выбравших в качестве системы автоматизации другую платформу, для решения некоторых задач все равно использовали 1С, как более простую. Нет, я не берусь утверждать, что 1С проще, но бытует именно такое мнение, потому что достаточно легко найти специалиста (благодаря распространенности), который сможет запрограммировать с помощью 1С решение необходимой задачи. А пользователей 1С найти еще легче. Я думаю, что 90% бухгалтеров знает программу 1С:Бухгалтерия.

Лично мое мнение: 1С — очень неудачная разработка. Ее проблема в том, что система разрабатывалась слишком давно, и основная часть кода "жива" до сих пор. В дальнейшем вы увидите всю сложность этой системы, когда мы начнем знакомиться с ее базой данных.

Если система автоматизации 1С установлена для обслуживания небольших предприятий, то в качестве базы данных могут использоваться проверенные временем таблицы DBF. Но если с программой должны работать несколько пользователей, то желательно перейти на архитектуру "клиент-сервер". Да, это довольно дорого, и можно использовать сетевую модель все с теми же DBF-файлами, но надежность и производительность в этом случае неоправданно снижаются.

Сервер SQL намного эффективнее решает задачи, и если бы его цена не была такой высокой, то я рекомендовал бы работать с ним даже в однопользовательских системах. А теперь пора перейти к обсуждению вопроса совместного использования 1С и MS SQL Server.

Если вы не собираетесь работать с программой 1С:Предприятие, я все же рекомендую прочитать эту главу, потому что здесь мы будем учиться планировать размер данных, политику резервного копирования и лишний раз потренируемся в создании запросов на языке SQL. Можно считать, что эта глава является практическим закреплением изученного материала.

6.1. Конфигурирование

Вопросы установки базы данных мы оставим в стороне, потому что эта тема хорошо описана в документации, которая входит в комплект поставки. К тому же, некоторые партнеры компании 1С, распространяющие 1С:Предприятие, предоставляют бесплатную установку.

Для начала рассмотрим, как можно сконфигурировать программу 1С:Предприятие на работу с MS SQL Server. Запустите программу, и перед вами появится окно выбора конфигурации для загрузки (рис. 6.1). Базовые конфигурации, которые могут быть настроены в вашей системе (установлены вместе с дистрибутивом), скорее всего, работают с локальной версией и файлами DBF, CDX. Чтобы настроить 1С на работу через MS SQL Server, необходимо создать новую конфигурацию.

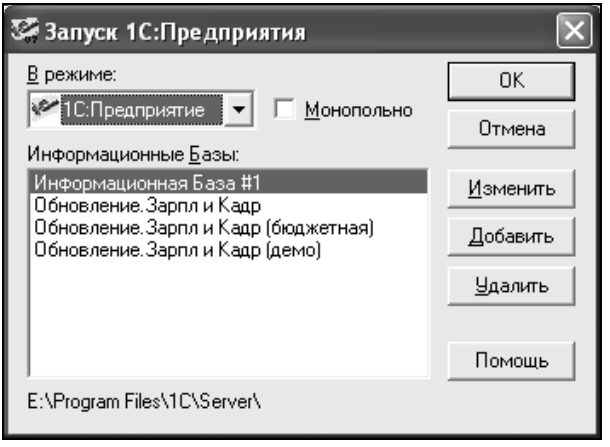


Рис. 6.1. Окно выбора конфигурации для загрузки

Конфигурация в 1С хранится в файле с расширением md. Создайте новый каталог, в котором будет храниться новая конфигурация, и скопируйте в него файл с расширением md (1cv7.md) из любой базовой конфигурации. Желательно, чтобы эта конфигурация выполняла необходимые вам действия, чтобы при программировании не пришлось переписывать все с нуля, и имелась хоть какая-то отправная точка. Я для иллюстрации выбрал конфигурацию "Зарплата и Кадры".

Теперь возвращаемся в окно выбора конфигурации. Нажмите кнопку **Добавить**, чтобы установить новую конфигурацию. Перед вами появится окно, в котором нужно указать путь к md-файлу и имя конфигурации.

На следующем этапе появится окно, в котором нужно выбрать тип хранения данных (рис. 6.2). Так как нас интересует MS SQL Server, то выбираем соответствующий переключатель и нажимаем кнопку **ОК**.

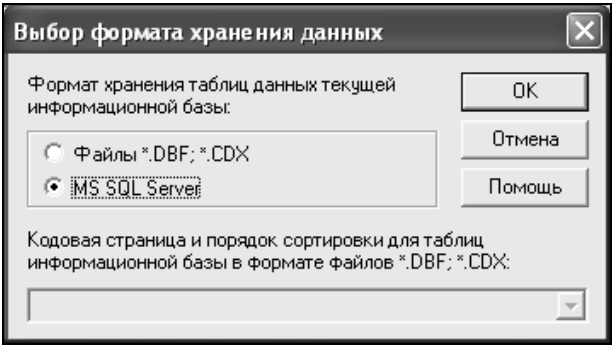


Рис. 6.2. Окно выбора формата хранения данных

Новая запись о конфигурации создана, и мы снова возвращаемся в окно выбора конфигурации (рис. 6.1). Выберите новую конфигурацию, затем в выпадающем списке **В режиме** выберите пункт **Конфигуратор** и нажмите кнопку **ОК** для загрузки.

После завершения загрузки вы оказываетесь в окне конфигуратора. Я не стал вносить какие-либо изменения в стандартную конфигурацию, потому что программирование в 1С выходит за рамки данной книги. Наша тема — MS SQL Server, и именно о нем мы будем говорить далее.

Пока что конфигурация загружена из md-файла, и мы еще ничего не сделали для того, чтобы она отобразилась в SQL-сервере. Давайте завершим настройку соединения. Для этого нам понадобится пользователь, с правами которого будет происходить подключение, и база данных, в которую будет перенесена выбранная конфигурация.

Начнем с пользователя. Вот тут проявляется первый недостаток 1С — подключение должно происходить с правами владельца базы (dbo). У такого пользователя слишком много прав, а ведь программа использует эти возможности на 1%, и в этом мы убедимся дальше. Но эти проблемы не критичны, и с ними можно бороться.

Будьте осторожны в выборе пользователя. Можно войти с администраторским паролем или как пользователь sa, но я рекомендую создать новую запись с необходимыми правами. Для этого можно взять учетную запись Windows или MS SQL Server. Давайте создадим учетную запись MS SQL Server и будем работать с ее правами:

```
EXEC sp_addlogin 'OneCUser', 'qwerty'
```

В реальных условиях лучше использовать учетную запись Windows, как более безопасную, регулярно менять пароль, а авторизацию MS SQL Server желательно вообще запретить.

Конфигуратор требует исключительных прав, потому что он будет создавать таблицы. Чтобы решить эту проблему, придется добавить нового пользователя в роль sysadmin. Для этого выполним следующую команду:

```
EXEC sp_addsrvrolemember 'OneCUser', 'sysadmin'
```

Теперь переходим к созданию базы данных. Вы должны с особой тщательностью подойти к вопросу выбора параметров базы, особенно к первоначальному размеру и размеру приращения файлов данных и журнала. Рассмотрим пример:

```
CREATE DATABASE OneC
ON
( NAME = OneC_data,
   FILENAME = 'c:\data\OneC.mdf',
   SIZE = 30MB,
   MAXSIZE = 100GB,
   FILEGROWTH = 5MB )
LOG ON
( NAME = 'OneC_log',
   FILENAME = 'c:\data\OneC.ldf',
   SIZE = 20MB,
   MAXSIZE = 10GB,
   FILEGROWTH = 5% )
```

Начнем с имени базы. Желательно выбрать такое имя, которое будет состоять только из букв, иначе могут возникнуть проблемы. Например, у меня программа категорически отказалась соединяться с базой по имени 1C.

Теперь о первоначальном размере базы. Сама конфигурация может занимать определенное место на диске. Чтобы не было лишних увеличений файлов данных, желательно выбрать достаточный размер. У меня файл md занимает 9 Мбайт. Конечно же, это не значит, что на сервере нужно столько же (реально понадобится меньше), но это может послужить хоть какой-то отправной точкой. Желательно увеличить это значение хотя бы в два раза. Я для своего файла данных выбрал 30 Мбайт.

Размер журнала также должен быть достаточным, чтобы уменьшить количество автоматических увеличений. Я провел следующий тест — создал новую базу с начальным размером в 1 Мбайт для журнала и данных и сохранил в нем конфигурацию. В результате после сохранения оба файла вместе стали занимать 4 Мбайт. Как видите, на начальном этапе нагрузка на журнал транзакций достаточно большая, поэтому для журнала выбран размер 20 Мбайт.

Чтобы файлы базы данных не переполнили все пространство на диске, желательно ограничить их максимальный размер. Да, теперь придется следить за тем, чтобы этот максимум не был достигнут, но это достаточно просто сделать, если создать оповещение *(см. разд. 3.7)*, которое будет отслеживать заполнение дискового пространства файлами данных и журнала и сообщит о критической ситуации оператору.

Дальнейшую нагрузку подсчитать достаточно сложно, все зависит от конкретной базы данных, количества одновременно работающих пользователей и количества вводимой и обновляемой информации. Тут надо действовать опытным путем и исходить из реальной работы базы данных в течение определенного промежутка времени.

В таблицы, содержащие информацию о работниках, изменения вносятся не так часто. Рассмотрим пример. Допустим, текучка кадров на предприятии не более 10% в месяц. Это значит, что если ваша организация состоит из 1000 работников, то в месяц будет уволено и принято на работу не более 100 человек. Сто операций вставки и удаления в месяц — это не так уж и много для MS SQL Server. Да, операций будет больше, потому что для добавления информации об одном человеке придется обновить несколько таблиц, но даже тысяча операций не даст слишком большой нагрузки на журнал транзакций. Поэтому я выбрал в качестве приращения для файла данных приращение 5 Мбайт, а для журнала транзакций 10%.

Самая большая нагрузка в конфигурации "Зарплата и Кадры" происходит во время расчета заработной платы. Но я не думаю, что к базе происходит достаточно много обращений.

А вот если вы создаете складскую или бухгалтерскую конфигурацию, то во избежание частых увеличений желательно указать значение приращения

побольше, потому что здесь операции добавления и обновления записей происходят ежедневно. При этом каждый день могут происходить тысячи, а то и миллионы операций обновления базы, что будет достаточно накладно и для файла журнала, и для файла данных. Тут приращение выбирается опытным путем на основе наблюдений за конкретной базой.

На начальном этапе можно порекомендовать указать в обоих приращениях значение 10%. Тогда по мере роста базы будет увеличиваться и приращение, а значит, оно будет происходить все реже и реже. Только в этом случае нужно больше уделять внимания занимаемому файлами пространству. Например, если база данных вырастет до 10 Гбайт, то для 10%-ного приращения понадобится целый гигабайт. А если такого количества свободного места на диске не окажется? Диск переполнится, и дальнейшая работа с базой станет невозможной. Но это уже проблемы обслуживания, о которых мы еще поговорим в *разд. 6.2*.

Если на вашем сервере установлено несколько жестких дисков, то стоит подумать о том, чтобы разнести файл данных и файл журнала по разным дискам. Мы уже говорили о том, что это может повысить производительность сервера за счет того, что работа с журналом и данными сможет происходить параллельно. Для достижения максимального выигрыша желательно, чтобы диски были подключены к разным контроллерам, т. е. они не должны "висеть" на одном шлейфе.

Все, о чем мы сейчас говорим, уже было рассмотрено в *главе 1*. Вспомните это, и, возможно, вы найдете еще несколько способов повышения производительности базы и сервера в целом. Могу только напомнить о существовании файловых групп. Советую рассмотреть вариант с созданием пользовательской файловой группы и сделать ее группой по умолчанию. Это позволит вам оставить в основной группе (`PRIMARY`) только системные таблицы, а все пользовательские таблицы будут попадать в пользовательскую группу.

После создания базы данных в нее необходимо перенести конфигурацию. Для этого вернитесь в Конфигуратор 1С и настройте соединение с новой базой данных. Выберите меню **Администрирование | Параметры базы данных SQL**, и перед вами откроется окно, как на рис. 6.3. Здесь четыре поля ввода для указания имени сервера, базы данных, пользователя и пароля. Заполните эти поля и нажмите кнопку **OK**. Если все данные введены правильно, то окно удачно закроется, иначе появится сообщение об ошибке подключения.

Чтобы загрузить конфигурацию, можно внести в нее изменение и выбрать меню **Файл | Сохранить** или просто выбрать меню **Администрирование | Тестирование и изменение ИБ**. Во втором случае перед вами появится окно, в котором можно указать параметры загрузки, а также выбрать, что при этом нужно проверять на корректность (рис. 6.4).

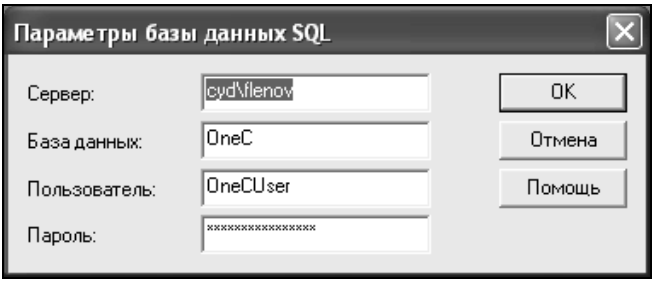


Рис. 6.3. Окно настройки соединения с сервером

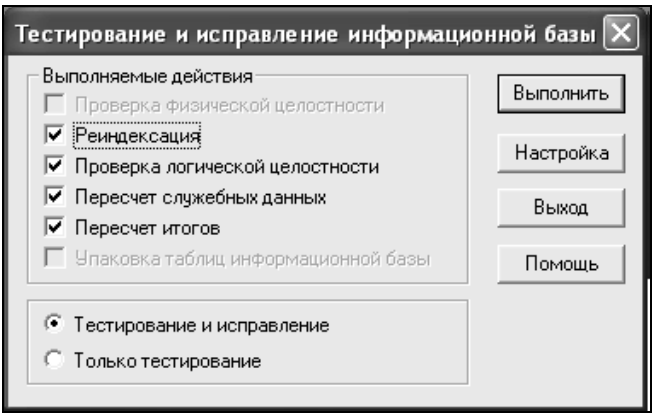


Рис. 6.4. Окно настройки загрузки конфигурации

Перед загрузкой программа 1С "честно признается", что вам необходимо самостоятельно создать резервную копию базы данных, потому что данные могут быть разрушены. Это действительно задача администратора. Вы сами должны заботиться о сохранении резервных копий. Делайте это перед каждым обновлением, чтобы не потерять данные.

Теперь база данных готова к использованию. Вы можете загрузить 1С:Предприятие и начать работу. Рекомендую ввести какие-либо данные, потому что по умолчанию база данных пустая, и последующие запросы, которые мы будем выполнять, ничего не вернут, а значит, будет утрачена наглядность.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter6\1C.sql на компакт-диске, прилагаемом к книге.

6.2. Обслуживание базы данных

В 1С:Предприятии нет никаких программных средств, упрощающих управление базой данных. А как мы уже знаем, создать базу данных мало, ею надо еще и управлять — наблюдать за ее работой, создавать резервную копию и при необходимости уметь восстанавливать базу.

6.2.1. Настройка базы данных

Создать базу недостаточно. Необходимо настроить ее, и желательно сделать это до первого использования или хотя бы до начала заполнения данными.

Начнем с модели восстановления. Дело в том, что модель, которую установил сервер по умолчанию, может не устроить вас и не позволит решить задачу обеспечения целостности данных. Давайте установим полную модель восстановления:

```
ALTER DATABASE OneC
SET RECOVERY FULL
```

Возможно, вы выберете другую модель резервного копирования/восстановления. Для принятия правильного решения вы должны вспомнить все, о чем мы говорили в *главах 1 и 4*, а также четко представлять себе структуру данных и частоту их использования в вашей базе. Главное — это настроить модель до начала работы с базой, чтобы к моменту необходимости создания первой копии у вас уже была выбрана подходящая модель. Если вы не сделаете этого, а сервер установит простую модель восстановления, то при резервном копировании журнала транзакций возникнут серьезные проблемы.

Теперь позаботимся об автоматическом создании и обновлении статистики. Для этого выполним следующие команды:

```
EXEC sp_dboption 'FlenovSQLBook', 'auto create statistics', 'TRUE'

EXEC sp_dboption 'FlenovSQLBook', 'auto update statistics', 'TRUE'
```

В первой команде мы включаем автоматическое создание статистики. Если сервер определит, что где-то не хватает статистики, то она будет создана. Во второй команде мы включаем автоматическое обновление, ведь устаревшая статистика не поможет оптимизатору, а то и помешает.

Это основные настройки, которые желательно произвести в базе данных, чтобы оптимизировать производительность и управляемость.

6.2.2. Резервное копирование

Теперь поговорим о резервном копировании. Мы уже знаем, что в случае с конфигурацией "Зарплата и Кадры" данные изменяются не так часто, а именно изменения нужно резервировать. Для такой конфигурации я бы использовал ежедневное полное резервное копирование в конце рабочего дня.

Полное копирование сделать проще всего. Для этого достаточно выполнить следующую команду:

```
BACKUP DATABASE FlenovSQLBook
TO DISK='c:\Backup\1cbackup.bak'
```

Выполнить восстановление также достаточно просто:

```
RESTORE DATABASE FlenovSQLBook
FROM DISK='c:\Backup\1cbackup.bak'
```

В случае нарушения целостности данных или работоспособности сервера единственное, что мы потеряем, — данные за один день. Так как их немного, то восстановить потерянную информацию можно достаточно быстро.

Если вам предстоит обслуживать складскую или бухгалтерскую конфигурацию, то тут уже изменения происходят постоянно. Количество изменений можно приблизительно определить даже по количеству операторов, работающих с базой.

Когда изменений много, то потеря информации даже одного дня может оказаться ощутимой. Придется заново вводить информацию за весь день, и хорошо, если будет, откуда ее вводить. А если нет первоисточника (например, бумажного документа), то восстановление станет невозможным, а точно вспомнить все изменения не сможет уже никто.

В таких случаях помимо ежедневного полного копирования базы данных полезно периодически (например, каждый час) создавать резервную копию журнала транзакций. Следующий запрос выполняет полное копирование с учетом того, что мы будем делать резервное копирование журнала:

```
BACKUP DATABASE OneC
TO  DISK = N'c:\backup\fullcopy.bak'
WITH  INIT,  NOUNLOAD,  NOSKIP,  STATS = 10,  NOFORMAT
```

В этом примере мы установили следующие параметры:

- `INIT` — все существующие в указанном файле резервные копии должны быть уничтожены. Незачем хранить много копий, потому что они займут лишнее место на жестком диске и носителе информации, куда будет скопирована резервная копия. Например, если база данных занимает 1 Гбайт, то при повторном резервном копировании в один и тот же файл без указания

этого параметра в файле будет две копии с разными датами, а значит, для его хранения потребуется 2 Гбайта, что не всегда приемлемо. Если необходимо хранить несколько полных копий за разные дни, то это желательно делать вручную. Например, копировать каждый файл резервной копии на отдельный носитель;

- `NOUNLOAD` — указывает, что ленточный носитель не должен быть автоматически извлечен из устройства чтения после завершения резервного копирования;
- `NOSKIP` — прежде чем перезаписывать существующие резервные копии, будут проверены даты действия, и перезапись произойдет, только если резервные копии устарели. Это необходимо, чтобы не нарушить цепочку резервного копирования и восстановления журналов транзакций;
- `STATS` — отображать статистику после выполнения каждых 10% всей команды;
- `NOFORMAT` — указывает, что не надо писать заголовок в каждый том резервной копии.

Если вы хотите сохранять историю резервных копий, т. е. не затирать предыдущие, то советую хранить каждую копию в своем файле. В этом случае удобно в качестве имени использовать дату создания резервной копии. Следующий пример показывает, как это реализовать на языке Transact-SQL:

```
declare @path varchar(50)
SET @path='c:\backup\'+cast(GETDATE() AS varchar(50))+'.bak'

BACKUP DATABASE OneC
TO  DISK = @path
WITH  INIT,  NOUNLOAD,  NOSKIP,  STATS = 10,  NOFORMAT
```

В первой строчке объявляется переменная `@path` для хранения пути к создаваемой резервной копии. Во второй строчке генерируется сам путь. Для этого мы складываем три значения:

- путь к каталогу, где будет храниться резервная копия;
- текущая дата, приведенная к строковому типу, которая будет являться именем файла;
- расширение файла.

Теперь достаточно создать резервную копию по указанному пути. Выполните этот сценарий и найдите созданный файл резервной копии. Я специально использовал преобразование `cast`, чтобы показать вам его недостаток. Дата будет в формате:

День Месяц Число Время

При этом для указания времени используется в качестве разделителя двоеточие, которое запрещено в именах файлов, поэтому имя файла будет содержать только часы создания, например, Aug 30 2005 10. Такая резервная копия сделана 30 августа 2005 года в 10 часов. Минуты и расширение будут обрезаны из-за недопустимого символа двоеточия. Это серьезная проблема, потому что у всех копий, созданных с 10 до 11 часов, будут одинаковые имена, так что новые копии просто перезапишут предыдущие.

Чтобы решить эту проблему, можно определить текущую дату по частям с помощью функции `DATEPART()` и установить те разделители, которые разрешены, но более простым способом будет замена символов двоеточия, например, на тире. Для этого можно воспользоваться функцией `REPLACE()` *(см. разд. 2.25.8)*. Итак, код резервного копирования будет выглядеть следующим образом:

```
declare @path varchar(50)
SET @path='c:\backup\'+
    REPLACE(cast(GETDATE() AS varchar(50)), ':', '-')+
    '.bak'

BACKUP DATABASE OneC
TO  DISK = @path
WITH  INIT,  NOUNLOAD,  NOSKIP,  STATS = 10,  NOFORMAT
```

После выполнения этого сценария имя файла будет следующим: Aug 30 2005 10-07AM.bak. Теперь только нельзя создавать две резервных копии за одну минуту. Я думаю, что это не проблема. Ведь если у вас большая база данных и сценарий выполняется больше минуты, то создать две копии за 1 минуту просто не удастся.

Рассмотрим сценарий, который можно использовать для резервного копирования журнала транзакций:

```
BACKUP LOG OneC
TO  DISK = N'c:\backup\logcopy.bak'
WITH  NOUNLOAD,  NOSKIP, STATS = 10,  NOFORMAT
```

Параметры оператора `BACKUP LOG` такие же, как и у оператора полного резервного копирования базы данных. Единственное отличие: здесь нет параметра `INIT`, чтобы все копии в файле сохранялись. Дело в том, что журнал должен сохраняться по накопительной схеме, и никакие предыдущие копии с момента полного резервирования не должны быть уничтожены. В противном случае восстановить данные не удастся, потому что цепочка будет нарушена.

Если резервные копии журнала будут достаточно большими, то лучше не накапливать их в одном файле, а сохранять каждую в своем. Для этого необходимо добавить опцию INIT, а в качестве имени файла использовать дату, но для того, чтобы файл можно было отличить от полной резервной копии, добавляем в начало имени logcopy и расширение log:

```
declare @path varchar(50)
SET @path='c:\backup\logcopy'+
    REPLACE(cast(GETDATE() AS varchar(50)), ':', '-')+
    '.log'

BACKUP LOG OneC
TO  DISK = @path
WITH  INIT, NOUNLOAD,  NOSKIP, STATS = 10,  NOFORMAT
```

6.2.3. Восстановление данных

Теперь переходим к восстановлению данных. Вы должны заранее подготовиться к возможным проблемам и протестировать сценарии создания резервных копий на предмет корректного восстановления. У меня был случай, когда резервная копия создавалась так, что не позволяла восстанавливать журналы без наличия данной копии. Такая проблема может возникнуть, если вы не указали параметр NOSKIP. В *разд. 6.2.2* этот параметр использовался, а это гарантирует, что данные будут перезаписаны только в том случае, если они устарели.

Я рекомендую после того, как вы определитесь с сценариями создания резервных копий, протестировать восстановление на тестовом сервере, т. е. взять рабочие резервные копии и восстановить их на компьютере, где раньше даже не было этой базы. Если все пройдет успешно, то можно будет гарантировать, что в случае нештатной ситуации вам удастся восстановить свою базу. А опыт, который вы получите во время тестирования, поможет вам произвести восстановление в максимально короткие сроки.

Восстановление цепочки журналов мы рассматривали только в теории, а сейчас познакомимся с этим процессом в практике.

Итак, сначала необходимо восстановить базу данных. Если вы создаете только полные резервные копии, то команда восстановления будет выглядеть достаточно просто:

```
RESTORE DATABASE OneC
FROM DISK='c:\Backup\Aug 30 2005 11-20AM.bak'
```

Здесь мы просто восстанавливаем базу данных OneC из указанного файла. При этом не забывайте, что в момент восстановления база данных должна быть свободна, т. е. к ней не должны быть подключены пользователи, в том числе и вы. Рекомендуется, чтобы вы в момент выполнения команды были подключены к базе данных master.

Если в выбранной вами модели используются журналы, то необходимо добавить опцию NORECOVERY, как показано в следующем примере:

```
RESTORE DATABASE OneC
FROM DISK='c:\Backup\Aug 30 2005 11-20AM.bak'
WITH NORECOVERY
```

После восстановления база данных окажется в режиме "только для чтения", но вы сможете восстанавливать журналы, которые были созданы после того, как была создана полная резервная копия.

Следующий пример показывает, как можно восстановить журнал транзакций:

```
RESTORE LOG OneC
FROM DISK='c:\backup\Aug 30 2005 11-20AM.log'
WITH NORECOVERY
```

Обратите внимание, что в данной команде также присутствует опция NORECOVERY. Это значит, что база все еще недоступна для изменений, и можно восстанавливать следующий журнал.

Журналы нужно восстанавливать в том же порядке, в котором они и создавались. Вы не можете пропустить один файл и восстановить его позже или вообще не восстанавливать. Необходимо четко соблюдать последовательность.

Вот тут проявляется удобство того, что в качестве имен файлов для резервных копий мы выбрали дату создания. Вы легко можете определить последовательность восстановления по имени файла.

Когда настанет очередь восстановления последнего файла резервной копии журнала, то опцию WITH NORECOVERY необходимо удалить:

```
RESTORE LOG OneC
FROM DISK='c:\backup\Aug 30 2005 11-20AM.log'
```

После выполнения этой команды база данных становится доступной для пользователей, но вы больше не сможете восстанавливать журналы транзакций. Если вы забудете указать опцию NORECOVERY, когда еще есть журналы для восстановления, то оставшиеся резервные копии восстановить не удастся. В этом случае придется начать все сначала, т. е. восстановить полную копию с опцией NORECOVERY и затем снова восстанавливать все резервные копии журналов.

6.2.4. Задания

Мы выбрали для нашей базы данных 1С в качестве стратегии резервного копирования и восстановления ежедневное полное копирование и ежечасное резервное копирование журналов. Но это не значит, что такие действия вы всегда должны выполнять вручную. В *разд. 3.5* мы познакомились с очень мощным средством администраторов — заданиями. Давайте создадим задание и поместим его в план выполнения, чтобы автоматизировать процесс создания резервных копий. При этом будем создавать задание так, чтобы оно действительно было эффективным, т. е. с информированием администратора в случае возникновения проблем.

Для начала нам понадобится оператор, который будет получать уведомления о статусе выполнения задания. Сообщения будут передаваться на почтовый ящик и командой `NET SEND`:

```
use msdb
exec sp_add_operator @name = 'Михаил',
   @enabled = 1,
   @email_address ='[SMTP:flenov@mail.ru]',
   @netsend_address ='flenov'
```

Напоминаю, что хранимая процедура `sp_add_operator` находится в базе данных msdb, поэтому сначала нужно подключиться к этой базе, иначе сценарий завершит работу ошибкой.

Теперь можно переходить к созданию заданий. Нам понадобятся два задания: для создания базы и для создания журнала. Каждое задание будет запланировано на выполнение шагов с различной периодичностью: полное резервное копирование будет происходить один раз в день, а резервное копирование журнала — каждый час.

Пример задания для создания полной копии может выглядеть следующим образом:

```
USE msdb
EXEC sp_add_job @job_name = 'Резервное копирование базы данных',
   @enabled = 1,
   @description = 'Это задание предназначено для создания полной
                   ежедневной резервной копии',
   @notify_level_eventlog = 3,
   @notify_level_email = 3,
   @notify_level_netsend = 2,
   @notify_email_operator_name = 'Михаил',
   @notify_netsend_operator_name = 'Михаил'
```

Задание сразу же делаем активным, поэтому параметр `@enabled` равен 1. Параметр `@description` содержит чисто информационное описание, которое может быть произвольным. Конечно же, лучше будет, если описание будет соответствовать выполняемым заданием действиям, иначе нет смысла его создавать. В нашем случае описание соответствует действиям.

Очень важно правильно настроить систему информирования оператора о статусе выполнения команд, чтобы вы могли максимально быстро прореагировать на нештатные ситуации. В журнал сообщений и на почтовый ящик будут направляться абсолютно все сообщения о ходе выполнения задания. С помощью команды `NET SEND` будут передаваться сообщения об ошибках, потому что эта команда доставляет сообщение мгновенно, и администратор сразу же увидит сообщение об ошибке на экране монитора.

Доверять команде `NET SEND` не стоит, потому что она не гарантирует доставку данных. Если администратор в этот момент перезагружает компьютер, то сообщение не будет доставлено. Для отправки критических сообщений об ошибках намного эффективнее было бы использовать интернет-пейджер, потому что он более надежен.

В качестве оператора, который будет получать сообщения, в данном примере выбран созданный ранее `Михаил`.

Задание, которое будет создавать резервную копию журнала транзакций, можно создать с теми же параметрами, что и задание резервного копирования базы данных, только нужно указать другое имя и соответствующее описание выполняемых действий в параметре `@description`:

```
USE msdb
EXEC sp_add_job @job_name = 'Резервное копирование журнала',
   @enabled = 1,
   @description = 'Это задание предназначено для
       ежечасного резервного копирования журнала',
   @notify_level_eventlog = 3,
   @notify_level_email = 3,
   @notify_level_netsend = 1,
   @notify_email_operator_name = 'Михаил',
   @notify_netsend_operator_name = 'Михаил'
```

Следующим этапом необходимо привязать задания к серверу. Для этого необходимо воспользоваться хранимой процедурой `sp_apply_job_to_targets`:

```
-- Связываем задание резервного копирования базы
EXEC sp_apply_job_to_targets
 @job_name = 'Резервное копирование базы данных',
```

```
 @target_servers='bigserver\flenov',
 @operation=APPLY

-- Связываем задание резервного копирования журнала
EXEC sp_apply_job_to_targets
  @job_name = 'Резервное копирование журнала',
  @target_servers='bigserver\flenov',
  @operation=APPLY
```

В данном случае привязка происходит к экземпляру базы данных 'flenov' на сервере 'bigserver'.

Теперь переходим к добавлению шагов, которые будут непосредственно заниматься резервным копированием базы данных и журнала. Следующий пример создает полную резервную копию:

```
EXEC sp_add_jobstep
 @job_name = 'Резервное копирование базы данных',
 @step_name = 'Резервное копирование',
 @subsystem = 'TSQL',
 @command = 'declare @path varchar(50)
     SET @path=''c:\backup\''+
         REPLACE(cast(GETDATE() AS varchar(50)), '':'', ''-'')
         +''.bak''
     BACKUP DATABASE OneC
     TO  DISK = @path
     WITH  INIT,  NOUNLOAD,  NOSKIP,  STATS = 10,  NOFORMAT',
 @database_name = 'FlenovSQLBook',
 @on_success_action = 1,
 @on_fail_action = 2
```

В параметре `@command` я указал созданный ранее сценарий полного резервного копирования базы данных. Если вы выбрали другую команду, то измените свой сценарий. Не забывайте, что внутри команды, которая записывается в параметр `@command`, нужно продублировать все одинарные кавычки и заменить одинарную кавычку на две одинарных.

В случае удачного завершения выполняемой команды просто завершаем выполняемое задание, поэтому в параметре `@on_success_action` указываем действие 1 (завершить выполнение задания с положительным результатом). Если во время выполнения задания произошла ошибка, то не будет лишним проинформировать администратора об этом, поэтому в параметр `@on_fail_action` помещаем значение 2 (выход с ошибкой). В обоих случаях происходит

выход, потому что наше задание будет состоять только из одного шага и больше никаких действий не предусмотрено.

Теперь добавим шаг резервирования журнала транзакций, который будет выглядеть схожим образом:

```
EXEC sp_add_jobstep
 @job_name = 'Резервирование журнала',
 @step_name = 'Резервирование',
 @subsystem = 'TSQL',
 @command = 'declare @path varchar(50)
     SET @path=''c:\backup\''+
         REPLACE(cast(GETDATE() AS varchar(50)), '':'', ''-'')
         +''.log''
     BACKUP LOG OneC
     TO  DISK = @path
     WITH  INIT, NOUNLOAD,  NOSKIP, STATS = 10,  NOFORMAT',
 @database_name = 'FlenovSQLBook',
 @on_success_action = 1,
 @on_fail_action = 2
```

Задание полностью готово, и теперь необходимо настроить планировщик задач так, чтобы задание выполнялось в указанное время. Именно с планировщиком возникают основные проблемы у начинающих программистов, и не зря, ведь здесь компания Microsoft изрядно намудрила с параметрами, и дополнительные примеры нам не помешают. Рекомендую сейчас найти *разд. 3.6.1*, где мы рассматривали хранимую процедуру `sp_add_jobschedule`, и сделать на него закладку, чтобы при необходимости вы могли взглянуть на описание параметров этой процедуры.

Первый план, который нам понадобится, должен выполняться ежедневно в конце рабочего дня, а точнее, в 18:30. Следующий код создает соответствующий планировщик:

```
EXEC sp_add_jobschedule
   @job_name = 'Резервное копирование базы данных',
   @name = 'План выполнения полного резервного копирования базы',
   @freq_type = 32,
   @freq_interval = 9,
   @active_start_date = 20050101,
   @active_start_time = 183000
```

Рассмотрим подробно каждый параметр и указанные нами значения:

- `@job_name` — имя задания, которое должен выполнять планировщик;

- `@name` — имя планировщика;
- `@freq_type` — частота выполнения. Значение 32 соответствует частоте, указанной в параметре `@freq_interval`;
- `@freq_interval` — интервал выполнения. При `@freq_type`, равном 32, число 9 в параметре `@freq_interval` означает выполнение задания в рабочие дни;
- `@active_start_date` — дата, когда задание должно стать активным;
- `@active_start_time` — время выполнения задания.

Теперь создадим задание резервного копирования журнала. Оно должно выполняться ежечасно, поэтому необходимо немного изменить код создания планировщика:

```
EXEC sp_add_jobschedule
   @job_name = 'Резервное копирование базы данных',
   @name = 'План выполнения полного резервного копирования базы',
   @freq_type = 32,
   @freq_interval = 9,
   @freq_subday_type=8,
   @freq_subday_interval=1,
   @active_start_date = 20050101,
   @active_start_time = 080000
   @active_end_time = 180000
```

И снова подробно рассмотрим каждый параметр и указанные нами значения:

- `@job_name` — имя задания, которое должен выполнять планировщик;
- `@name` — имя планировщика;
- `@freq_type` — частота выполнения. Значение 32 соответствует частоте, указанной в параметре `@freq_interval`;
- `@freq_interval` — интервал выполнения. При `@freq_type`, равном 32, число 9 в параметре `@freq_interval` означает выполнение задания в рабочие дни;
- `@freq_subday_type` — число 8 в этом параметре означает, что в параметре `@freq_subday_interval` указано время в часах между выполнениями заданий;
- `@freq_subday_interval` — число 1 в этом параметре означает ежечасное выполнение;
- `@active_start_date` — дата, когда задание должно стать активным;
- `@active_start_time` — время начала выполнения задания;
- `@active_end_time` — время окончания выполнения задания.

Надеюсь, что эти примеры помогут вам лучше понять принцип работы хранимой процедуры `sp_add_jobschedule`, которая создает план выполнения заданий.

Теперь у нас есть план резервного копирования, который позволит свести к минимуму потери данных в случае возникновения нештатной ситуации. При этом все необходимые действия по обеспечению целостности данных происходят автоматически.

Но это не значит, что можно сидеть, сложа руки, весь рабочий день, дел еще предостаточно. Иногда возникает необходимость создавать резервные копии внепланово, например, при работе с 1С я бы делал это перед каждым обновлением конфигурации, которая изменяет структуру базы данных. Из-за неправильного конфигурирования вы можете случайно уничтожить важную таблицу с данными, для восстановления которой может потребоваться слишком много времени.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter6\1Cmanage.sql на компакт-диске, прилагаемом к книге.

6.3. Выборка данных

Вот мы и подошли к самому интересному — выборке данных из базы данных 1С. Но прежде чем писать какие-то запросы, необходимо познакомиться со структурой базы данных и увидеть, с чем же работает 1С:Предприятие. Здесь мы будем рассматривать 1С версии 7.7. Версию 8.0 программы 1С:Предприятие я еще не видел, но, насколько мне удалось выяснить, изменений в структуре там не так уж и много.

Трудно описать ощущения, которые я испытал, когда первый раз взглянул на структуру этой базы данных. Это были и ужас, и восторг. Ужас охватил меня вот почему:

- имена таблиц не являются информативными, и понять их назначение без переводчика невозможно, а переводчиком явно служит файл md;
- все связи между таблицами явно реализуются на стороне клиента, потому что у базы данных нет ни одного внешнего ключа. То есть программа не использует возможности сервера по обеспечению целостности данных, а надеется только на себя, и это при том, что в моей конфигурации около 120 таблиц. Такая самоуверенность заставляет восхищаться программистами;

- нет никаких хранимых функций или представлений;
- пользовательские хранимые процедуры выполняют минимум задач — определение максимального идентификатора и выборка определенного значения.

А самое удивительное — все работает, и достаточно неплохо, и это, повторюсь, вызывает восхищение программистами. Я представляю, какой объем работы они выполнили хотя бы для обеспечения целостности данных. Будем надеяться, что программисты реализовали необходимую защиту. А ведь бóльшую часть работы можно было возложить на MS SQL Server. Вообще говоря, непонятно, зачем они выбрали такой мощный сервер баз данных, когда можно было взять что-нибудь простое и бесплатное, например, MySQL или Interbase.

Чтобы самостоятельно просмотреть объекты базы данных, выполните следующий запрос:

```
USE OneC
EXEC sp_help
```

В результате вы увидите таблицу из трех полей:

- `Name` — имя объекта;
- `Owner` — владелец;
- `Object type` — тип объекта.

Для пользовательских таблиц в поле `Object type` можно увидеть значение `user table`.

Но это все мелочи. Самый сильный ужас я испытал, когда запустил Profiler, который позволяет увидеть запросы, выполняемые сервером. Вот тут я ощутил только ужас, потому что все запросы были просты, и для выборки определенных данных выполнялось по несколько элементарных запросов, хотя 90% из них можно было оптимизировать или вообще выкинуть.

Да, 1С:Предприятие — универсальная программа, которая может решать множество различных задач, но все же можно было бы оптимизировать некоторые запросы, или хотя бы более полно использовать возможности MS SQL Server. Производительность программы возросла бы в разы.

Нет, я не пытаюсь критиковать разработчиков и уважаю их за такой замечательный продукт. Главное, что программа решает поставленные перед ней задачи. Это универсальная разработка, и она всегда будет работать медленнее, чем специализированная программа, направленная на решение узкоспециальной задачи. К тому же, универсальное решение написать намного сложнее, особенно со встроенным языком программирования, генератором отчетов и настраиваемым интерфейсом.

На этом закончим вводную часть и перейдем к практическим занятиям и попробуем написать запросы к базе данных.

Как я уже сказал, все имена объектов в базе не несут в себе никакой смысловой нагрузки. Это усложняет нашу задачу, потому что для определения смысла таблицы приходится просматривать содержимое и догадываться обо всем самостоятельно. Все исследования базы данных происходят методом "научного тыка".

Первая таблица, которая бросилась мне в глаза, — `_1SUSERS`. Это одна из немногих таблиц, имя которой несет хоть какую-то смысловую нагрузку. Посмотрим на ее содержимое:

```
SELECT *
FROM _1SUSERS
```

Здесь у нас два поля `USRSCNT` и `NETCHGCN`. Назначение первого поля достаточно просто определить даже по имени — это количество пользователей, которые сейчас подключены к базе через программу 1С. Во время загрузки программа явно увеличивает счетчик из этого поля на единицу.

Следующая таблица — `_1SCONST`, определяет значения констант, описанных в текущей конфигурации. Посмотрим ее содержимое:

```
SELECT *
FROM _1SCONST
```

В этой таблице находятся только значения констант, а имена нужно искать в конфигурационном файле. Но это не проблема. Имена можно определить, если посмотреть их в программе и сопоставить записи по значению. Да, это неудобно, но что делать, если вам понадобится.

Дальнейшие исследования ведут уже к таблицам, имена которых начинаются с префикса `sc`, а дальше идет номер. Тут трудно дать определенные рекомендации по поводу конкретных таблиц, потому что их имена могут зависеть от конкретной конфигурации и от конфигуратора 1С. Но мне удалось определить наиболее часто используемые справочники в различных конфигурациях:

- `sc148` — отделы;
- `sc16` — работники;
- `sc177` — зарплаты;
- `sc193` — должности;
- `sc2005` — контрагенты;
- `sc339` — страны;
- `sc467` — валюта;

- `sc474` — типы выплат;
- `sc491` — удержания;
- `sc525` — документы;
- `sc546` — области.

Надеюсь, что это вам поможет. Рассматривать запросы `SELECT` к этим таблицам мы не будем по двум причинам:

1. Эту тему мы достаточно подробно разобрали в *главе 2*.
2. Имена таблиц и их содержимое зависят от конфигурации, а не от нас.

Рекомендую исследовать вашу собственную базу данных на предмет содержащихся в них данных. Если у вас возникнет необходимость быстро получить простой отчет, и вы будете точно знать таблицу, где находятся необходимые данные, то прямой запрос решит вашу проблему намного быстрее. Как я уже сказал, запросы 1С явно неэффективны из-за своей универсальности, но жертвовать этой универсальностью тоже нельзя, иначе это будет уже не 1С.

Как видите, многого из базы данных 1С не выжмешь. Основная проблема в том, что база данных используется не на все 100%.

Примечание

Примеры запросов из этого раздела можно найти в файле \Chapter6\1Cquery.sql на компакт-диске, прилагаемом к книге.

Заключение

Я надеюсь, что книга помогла вам изучить язык структурированных запросов SQL, а примеры пригодятся вам в реальной работе с MS SQL Server. Напоследок хочется дать некоторые заключительные рекомендации из личной практики.

Разрабатывая приложения "клиент-сервер", старайтесь помещать логику на сервер в виде встроенных процедур, функций и представлений. Без особой надобности не храните запросы на стороне клиента. Если запросы хранятся в клиентской программе, то в случае изменения логики придется изменять все клиентские программы, что может оказаться достаточно сложным, если у вас есть удаленные офисы.

Если логика находится на сервере, то достаточно изменить только базу данных или сценарий (процедуру, функцию или представление), и клиенты уже готовы работать с новой логикой. При этом в большинстве случаев даже не потребуется перезагрузка клиентских программ. Все произойдет на ходу.

По возможности нужно использовать ANSI SQL без расширения Transact-SQL. Таким образом, программы будут максимально переносимыми и независящими от базы данных.

Очень часто производители программного обеспечения "клиент-сервер" создают облегченные (тестовые) версии, которые работают с MS Access. Это упрощает установку тестовой программы, но от некоторых возможностей в любом случае придется отказаться. База данных MS Access не поддерживает триггеры и не может этого делать по определению.

Мы не зря так тщательно изучали возможности Transact-SQL. Несмотря на то, что MS SQL Server обладает мощными визуальными средствами моделирования баз данных и таблиц, лучше всего это делать с помощью сценариев. Недаром Enterprise Manager позволяет сохранять сценарии, которые использовались для превращения действий, выполненных визуально, в реальную базу или таблицу.

Имея сценарии, можно легко восстанавливать пустую структуру базы данных или модифицированную версию. Опять же пример из личного опыта. Однажды я разрабатывал программу для производственных целей, которая считывала параметры с оборудования и сохраняла их в базе данных. Один компьютер и одна база данных обслуживали до 8 производственных линий. Почему именно так? В цехах не было сети, и тем более не было сети между цехами.

Сначала был внедрен один сервер. Потом нужно было создать второй, в котором структура базы данных была немного другой. У меня не было сценариев Transact-SQL, поэтому пришлось брать копию уже работающей базы, чистить ее и изменять.

Нередко в одной базе данных существуют таблицы, имеющие схожую структуру. Например, для все той же промышленной базы данных для удобства были созданы 8 одинаковых таблиц. Каждая производственная линия сохраняла данные в свою таблицу. С одной стороны — это неэффективно, а с другой — так требовала задача. Для этой задачи такое решение было оправданным, потому что каждая таблица находилась в своей файловой группе, и удобно было создавать их резервные копии.

Если у вас в базе данных есть таблицы со схожей структурой, и при этом они будут связываться, а данные будут выбираться из нескольких таблиц, я рекомендую именовать поля уникально для всей базы данных. Это позволит избежать ошибки двусмысленного имени поля. Как мы знаем, эта ошибка возникает, когда выбирается поле, имя которого есть в нескольких таблицах, без четкого указания таблицы. Если вы не хотите лишних проблем, то лучше делать имена полей уникальными.

Именование объектов должно подчиняться определенным правилам. Не забывайте об информативности, т. е. вы должны уже по именам объектов понимать их предназначение. В *главе 1* мы не особо подчинялись каким-либо законам, но для тестовой базы данных я хотел показать вам свои собственные правила, выработанные в результате долгой практики.

ПРИЛОЖЕНИЯ

Приложение 1. Типы данных в SQL Server 2000

Приложение 2. Описание компакт-диска

Приложение 1

Типы данных в SQL Server 2000

Числа

- `bigint` — целое число. Может принимать значения от -2^{63} (–9 223 372 036 854 775 808) до $2^{63}-1$ (9 223 372 036 854 775 807).
- `int` — целое число. Может принимать значения от -2^{31} (–2 147 483 648) до $2^{31}-1$ (2 147 483 647).
- `smallint` — целое число. Может принимать значения от 2^{15} (–32 768) до $2^{15}-1$ (32 767).
- `tinyint` — целое число от 0 до 255. Отрицательных значений не может быть.
- `bit` — этот тип называют также логическим. Хотя реально в базе данных он представлен целым числом, в поле этого типа можно записать 1 или 0.
- `decimal` — число с заданной точностью (дробное число). Возможные значения от $-10^{38}+1$ до $10^{38}-1$. Удобство заключается в том, что вы можете указать необходимую точность данных (число знаков после запятой). Для объявления этого типа нужно указать после имени в круглых скобках количество знаков до и после запятой, например, `decimal(10, 1)` позволяет записывать числа с 10 знаками до запятой и 1 знаком после.
- `numeric` — этот тип по функциональности соответствует `decimal` и используется для совместимости со стандартом и другими базами данных.

Числа с плавающей точкой

- `float` — число с плавающей точкой. Может принимать значения от –1.79E+308 до 1.79E+308.
- `real` — число с плавающей точкой. Может принимать значения от –3.40E+38 до 3.40E+38.

Денежные типы

- `money` — денежный тип, который удобно использовать для полей, в колонках которых вводятся цены или денежные суммы. Может принимать значения от –922 337 203 685 477.5808 до 922 337 203 685 477.5807. Значение хранится с точностью до 4 знаков после запятой. Это значит, что в поле этого типа можно записать число 10,1111 рубля.
- `smallmoney` — денежный тип. Может принимать значения от –214 748.3648 до 214 748.3647. Как видите, после запятой также может быть 4 знака.

Дата и время

- `datetime` — тип для хранения даты и времени. Может принимать значения от 1 января 1753 года до 31 декабря 9999 года, с точностью до 3.33 миллисекунд. Необходимо помнить, что даты по умолчанию на сервере хранятся в формате мм/дд/гггг, т. е. сначала идет месяц, потом число и потом год. Это немного непривычно для нас, потому что в России принято сначала указывать число и только потом месяц.
- `smalldatetime` — тип для хранения даты и времени. Может принимать значения от 1 января 1900 года до 6 июня 2079 года с точностью до 1 минуты.

Строки

- `char` — строка фиксированной длины для хранения данных в кодировке, отличной от Unicode. Максимальная длина 8000 символов. Это значит, что если для хранения строки выделено 10 символов, а вы записали в поле строку длиной 5 символов, в базе данных оставшиеся 5 символов будут дополнены пробелами. Для объявления этого типа нужно указать после имени в круглых скобках длину поля, например, `char(10)`.
- `varchar` — строка переменной длины для хранения данных в кодировке, отличной от Unicode. Максимальная длина 8000 символов. В базе данных такая строка занимает ровно столько символов, сколько записано в поле. Для объявления этого типа нужно указать после имени в круглых скобках длину поля, например, `varchar(10)`.
- `text` — строка переменной длины для хранения данных в кодировке, отличной от Unicode. Максимальная длина 2 147 483 647 символов.
- `nchar` — Unicode-строка фиксированной длины. Максимальная длина 4000 символов (Unicode-аналог типа `char`).

- `nvarchar` — Unicode-аналог типа `varchar`. Максимальная длина 4000 символов.
- `ntext` — Unicode-вариант типа `text`. Максимальная длина строки 1 073 741 823 символов.

Бинарные данные

- `binary` — тип для хранения бинарных данных фиксированного размера. Максимальный размер 8000 байт.
- `varbinary` — тип для хранения бинарных данных переменного размера. Максимальный размер 8000 байт.
- `image` — тип для хранения бинарных данных переменного размера. Максимальный размер 2 147 483 647 байт.

Другие типы данных

- `cursor` — ссылка на курсор.
- `sql_variant` — тип данных, который может принимать любые поддерживаемые сервером MS SQL Server типы, за исключением: `text`, `ntext`, `timestamp` и `sql_variant`.
- `table` — специальный тип для хранения таблиц. Чаще всего используется для хранения временного результата запроса для использования при дальнейших расчетах.
- `timestamp` — уникальное число на уровне базы данных, которое обновляется каждый раз, когда обновляется строка в базе данных.
- `uniqueidentifier` — тип для хранения глобального уникального идентификатора GUID.

Приложение 2

Описание компакт-диска

Папки	Описание
\Chapter1	Файлы сценариев, использованные в главе 1, и резервная копия базы данных для тестирования сценариев последующих глав
\Chapter2	Файлы сценариев, использованные в главе 2
\Chapter3	Файлы сценариев, использованные в главе 3
\Chapter4	Файлы сценариев, использованные в главе 4
\Chapter5	Файлы сценариев, использованные в главе 5
\Chapter6	Файлы сценариев, использованные в главе 6
\Doc	Дополнительная документация. Первые три главы бесплатной книги по .NET и JBuilder
\Programs	Условно бесплатные и бесплатные программы от CyD Softaware Labs

Предметный указатель

A

ABS(), функция 236
ACOS(), функция 239
ADD CONSTRAINT, оператор 102
ADD FILE, оператор 94
ADD FILEGROUP, оператор 94
ADD, оператор 100
ALL, оператор 171
ALTER COLUMN, оператор 104
ALTER DATABASE, оператор 33, 88
ALTER FUNCTION, оператор 277
ALTER PROCEDURE, оператор 264
ALTER TABLE, оператор 98
ALTER TRIGGER, оператор 283
ALTER VIEW, оператор 252
AND, оператор 133
ANSI_DEFAULTS, параметр 349
ANSI_NULL_DFLT, параметр 349
ANSI_NULLS, параметр 350
ANSI_PADDING, параметр 350
ANSI_WARNINGS, параметр 351
ANY, оператор 170
ASIN(), функция 240
ATAN(), функция 240
AVG(), функция 154

B

BACKUP DATABASE, оператор 410, 531
BACKUP LOG, оператор 419, 533
BEGIN TRANSACTION, оператор 189
BETWEEN, оператор 137
BREAK, оператор 217
BULK_LOGGED, параметр 96

C

CASE, оператор 214
CAST(), функция 200
CEILING(), функция 237
CHECK, оператор 51
CLOSE, оператор 505
COLLATE, параметр 21
COLUMNPROPERTY(), функция 355
COMMIT TRANSACTION, оператор 189, 341
CONSTRAINT, оператор 53
CONTAINS, оператор 517
CONTINUE, оператор 218
CONVERT(), функция 202
COS(), функция 240
COUNT(), функция 154
CREATE DATABASE, оператор 18
CREATE DEFAULT, оператор 112
CREATE FUNCTION, оператор 268
CREATE INDEX, оператор 77
CREATE PROCEDURE, оператор 257
CREATE RULE, оператор 113
CREATE STATISTICS, оператор 381
CREATE TRIGGER, оператор 280
CREATE VIEW, оператор 246
CUBE, оператор 486
CURSOR_CLOSE_ON_COMMIT, параметр 96, 348
CyD SQL Factory, программа 9

D

DATABASEPROPERTYEX(), функция 353
DATEADD(), функция 206
DATEDIFF(), функция 207
DATEFIRST, параметр 346
DATEFORMAT, параметр 347
DATENAME(), функция 208
DATEPART(), функция 208
DB_ID(), функция 352
DB_NAME(), функция 352
DBCC SHOW_STATISTICS, оператор 384
DBCC SHOWCONTIG, оператор 375
DBCC SQLPERF, оператор 358
DBCC USEROPTIONS, оператор 363
DDL 17
DECLARE CURSOR, оператор 499
DECLARE, оператор 197
DELETE, оператор 183, 287
DENY, оператор 400
DROP COLUMN, оператор 101
DROP CONSTRAINT, оператор 102
DROP DATABASE, оператор 23
DROP DEFAULT, оператор 113
DROP FUNCTION, оператор 279
DROP INDEX, оператор 77
DROP PROCEDURE, оператор 259
DROP RULE, оператор 113
DROP TABLE, оператор 42
DROP TRIGGER, оператор 285
DROP VIEW, оператор 253

E

ELSE, оператор 211
ENCRYPTION, параметр 266
EXECUTE, оператор 259
EXISTS, оператор 169
EXP(), функция 239

F

FETCH, оператор 501
FILEGROUP, оператор 31
FILEGROWTH, параметр 21
FILENAME, параметр 20
FLOOR(), функция 238
FORCEPLAN, параметр 348
FORMSOF(), функция 520
FROM, оператор 122

G

GETDATE(), функция 205
GO, оператор 121, 122
GRANT, оператор 396
GROUP BY, оператор 123, 156
GUID 37, 87, 221

H

HAVING, оператор 123

I

IDENT_INCR(), функция 359
IDENT_SEED(), функция 359
IDENTITY, свойство 43, 359
IF, оператор 211
IMPLICIT_TRANSACTIONS, оператор 341
IN, оператор 136
INNER JOIN, оператор 150
INSERT, оператор 63, 172, 285
INTO, оператор 122, 139
IS, оператор 138
ISABOUT(), функция 520
ISNULL(), функция 220

L

LEFT(), функция 228
LEN(), функция 229
LIKE, оператор 55, 131, 140
LOG(), функция 239
LTRIM(), функция 230

M

MAX(), функция 154
MAXSIZE, параметр 21
MIN(), функция 154
MODIFY NAME, оператор 95
MOVE, оператор 425

N

NOT, оператор 134
NULL, значение 50, 138, 347, 349

O

OBJECT_ID(), функция 355
OPEN, оператор 501
OPENDATASOURCE(), функция 446
OPENROWSET(), функция 444
OR, оператор 134
ORDER BY, оператор 123, 155
OUTER JOIN, оператор 150

P

PATINDEX(), функция 231
PI(), функция 239
PRIMARY KEY, оператор 58
PRIMARY, параметр 20
PRINT, оператор 198

Q

Query Analyzer 13

R

RAISERROR(), функция 366
RAND(), функция 239
READ_ONLY, параметр 96
READ_WRITE, параметр 96
READTEXT, оператор 493
RECOMPILE, параметр 266
RECOVERY FULL, параметр 96
REMOVE FILE, оператор 94
REMOVE FILEGROUP, оператор 95
REPLACE(), функция 231
REPLICATE(), функция 232
RESTORE DATABASE,
оператор 423, 534
RESTORE FILELISTONLY,
оператор 430
RESTORE LOG, оператор 432, 535
RETURN, оператор 218
REVERSE(), функция 233
REVOKE, оператор 402
ROLLBACK TRANSACTION,
оператор 189, 281
ROLLUP, оператор 489
ROUND(), функция 238
RTRIM(), функция 230

S

SAVE TRANSACTION, оператор 194
SCOPE_IDENTITY(), функция 359
SELECT, оператор 122
SET, оператор 95, 174, 339
SHOWPLAN_ALL, оператор 458
SHRINKDATABASE, оператор 438
SHRINKFILE, оператор 438
SIGN(), функция 237
SIMPLE, параметр 96
SIN(), функция 240
SINGLE_USER, параметр 95, 96
SIZE, параметр 20
sp_add_alert, процедура 331
sp_add_job, процедура 300, 536
sp_add_jobschedule,
процедура 324, 539
sp_add_jobstep, процедура 306, 538
sp_add_notification, процедура 336
sp_add_operator, процедура 302
sp_addlinkedserver, процедура 448
sp_addlinkedsrvlogin, процедура 450
sp_addlogin, процедура 388
sp_addmessage, процедура 330, 369
sp_addrole, процедура 394
sp_addrolemember, процедура 394
sp_addsrvrolemember, процедура 394
sp_addtype, процедура 372
sp_addumpdevice, процедура 412
sp_adduser, процедура 390
sp_apply_job_to_targets,
процедура 313, 537
sp_attach_db, процедура 441
sp_autostats, процедура 386
sp_bindefault, процедура 112
sp_bindrule, процедура 113
sp_changegroup, процедура 395
sp_configure, процедура 296, 365
sp_dboption, процедура 361

sp_delete_alert, процедура 335
sp_delete_job, процедура 320
sp_delete_jobschedule, процедура 329
sp_delete_notification, процедура 338
sp_delete_operator, процедура 306
sp_denylogin, процедура 390
sp_depends, процедура 297
sp_detach_db, процедура 439
sp_droplogin, процедура 389
sp_dropmessage, процедура 331
sp_droprole, процедура 394
sp_droprolemember, процедура 394
sp_dropuser, процедура 391
sp_executesql, процедура 471
sp_fulltext_catalog, процедура 511
sp_fulltext_column, процедура 514
sp_fulltext_database, процедура 511
sp_fulltext_table, процедура 512
sp_grantdbaccess, процедура 391
sp_grantlogin, процедура 389
sp_help, процедура 74, 360
sp_help_alert, процедура 335
sp_help_fulltext_catalogs, процедура 516
sp_help_fulltext_tables, процедура 516
sp_help_job, процедура 315
sp_help_jobschedule, процедура 329
sp_help_jobstep, процедура 318
sp_help_operator, процедура 305
sp_helpdb, процедура 91, 361
sp_helpindex, процедура 374
sp_helprole, процедура 395
sp_helpserver, процедура 362
sp_helptext, процедура 298
sp_helptrigger, процедура 297
sp_helpuser, процедура 392
sp_password, процедура 389
sp_revokelogin, процедура 390
sp_settriggerorder, процедура 294
sp_start_job, процедура 314
sp_tableoption, процедура 39
sp_unbindefault, процедура 112
sp_unbinrule, процедура 113
sp_update_alert, процедура 335
sp_update_job, процедура 319
sp_update_jobschedule, процедура 328
sp_update_jobstep, процедура 320
sp_update_notification, процедура 338
sp_update_operator, процедура 304
sp_updatestats, процедура 386
SPACE(), функция 234
SQL, язык
  вложенный 6
  интерактивный 6
  стандарт 4
SQRT(), функция 240
SQUARE(), функция 240
STATISTICS IO, оператор 455
STATISTICS PROFILE, оператор 461
STR(), функция 234
STUFF(), функция 235
SUBSTRING(), функция 227
SUM(), функция 154

T

TAN(), функция 240
TOP, оператор 128
TRUNCATE TABLE, оператор 186

U

UNION, оператор 160
UPDATE, оператор 178, 253, 289, 383
UPDATETEXT, оператор 495
UPPER(), функция 229
USE, оператор 121

W

WAITFOR, оператор 220
WHERE, оператор 123, 130
WHILE, оператор 216
WRITETEXT, оператор 495

X

XML, язык 370
xp_cmdshell, процедура 464
xp_enumgroups, процедура 468
xp_logevent, процедура 470
xp_logininfo, процедура 466
xp_msver, процедура 468

А

Автоматическое увеличение 43
Авторизация 387

Б

База данных:
  восстановление 423, 534
  отключение 439
  переименование 95
  подключение 440
  резервное копирование 410, 531
  свойства 95
  создание 18
  удаление 23
Блокировка 342
  мертвая 344
Булевы операторы 133

В

Внешнее объединение 145
Внешний ключ 60
Внутреннее объединение 145, 150
Восстановление 409
  базы данных 423, 534
  журнала транзакций 432, 535
  файлов и файловых групп 431
  части базы данных 427
Временная таблица 85, 139
Выполнение процедуры 259
Вычисляемое поле 82

Г

Глобальная переменная 199
Группа файловая 28
Группировка 156

Д

Данные:
  получение 122, 542
  удаление 183
  фрагментация 374
Дата 203, 346
Дефрагментация данных 377
Добавление:
  записи 221
  ограничения 102
  поля 100
  учетной записи 388, 390
  файла 94
  файловой группы 94

Ж

Журнал транзакций 24, 358

З

Задание:
  обновление 319
  создание 300, 536
  удаление 320
Запись:
  добавление 221
  удаление 63, 183
Значение по умолчанию 47
  создание 112
  удаление 113

И

Изменение:
  данных 178
  процедуры 264
  триггера 283
  функции 277
Имя:
  базы данных 23
  глобальной переменной 199
  логическое 25
  локальной переменной 197
  объекта 355
  пользователя 354
  правила 23
  представления 249
  приложения 354
  таблицы 125
  физическое 25

Индекс 69, 373
кластерный 71
некластерный 71
создание 76
составной 78
удаление 77
Информация:
о базе данных 361
о пользователях 392
о правах доступа 403
о ролях 395
о сервере 362
о таблице 360
об индексах 374
об опциях базы данных 361

К

Кластерный индекс 71
Ключ:
внешний 60
первичный 58
Ключевое поле 37
Кодировка 34
Критерии поиска 130
Курсор 497
закрытие 505
объявление 499
открытие 501
тип 498
чтение 501
Куча 69
Кэш 256

Л

Локальная переменная 197

М

Математические операции 153
Мертвая блокировка 344

Н

Некластерный индекс 71

О

Обновление:
задания 319
оператора 304
оповещения 335
планировщика 328
уведомления 338
шага задания 320
Объединение:
внешнее 145
внутреннее 145, 150
ортогональное 144, 150
Ограничение 50
добавление 102
удаление 102
Оператор:
ADD 100
ADD CONSTRAINT 102
ADD FILE 94
ADD FILEGROUP 94
ALL 171
ALTER COLUMN 104
ALTER DATABASE 33, 88
ALTER FUNCTION 277
ALTER PROCEDURE 264
ALTER TABLE 98
ALTER TRIGGER 283
ALTER VIEW 252
AND 133
ANY 170
BACKUP DATABASE 410, 531
BACKUP LOG 419, 533
BEGIN TRANSACTION 189
BETWEEN 137
BREAK 217
CASE 214
CHECK 51
CLOSE 505
COMMIT TRANSACTION 189, 341
CONSTRAINT 53
CONTAINS 517
CONTINUE 218
CREATE DATABASE 18

CREATE DEFAULT 112
CREATE FUNCTION 268
CREATE INDEX 77
CREATE PROCEDURE 257
CREATE RULE 113
CREATE STATISTICS 381
CREATE TABLE 39
CREATE TRIGGER 280
CREATE VIEW 246
CUBE 486
DBCC SHOW_STATISTICS 384
DBCC SHOWCONTIG 375
DBCC SQLPERF 358
DBCC USEROPTIONS 363
DECLARE 197, 499
DELETE 183, 287
DENY 400
DROP COLUMN 101
DROP CONSTRAINT 102
DROP DATABASE 23
DROP DEFAULT 113
DROP FUNCTION 279
DROP INDEX 77
DROP PROCEDURE 259
DROP RULE 113
DROP TABLE 42
DROP TRIGGER 285
DROP VIEW 253
ELSE 211
EXECUTE 259
EXISTS 169
FETCH 501
FILEGROUP 31
FROM 122
GO 121, 122
GRANT 396
GROUP BY 123, 156
HAVING 123
IF 211
IMPLICIT_TRANSACTIONS 341
IN 136
INNER JOIN 150
INSERT 63, 172, 285
INTO 122, 139
IS 138
LIKE 55, 131, 140
MODIFY NAME 95
MOVE 425
NOT 134
OPEN 501
OR 134
ORDER BY 123, 155
OUTER JOIN 150
PRIMARY KEY 58
PRINT 198
READTEXT 493
REMOVE FILE 94
REMOVE FILEGROUP 95
RESTORE
  DATABASE 423, 534
RESTORE FILELISTONLY 430
RESTORE LOG 432, 535
RETURN 218
REVOKE 402
ROLLBACK
  TRANSACTION 189, 281
ROLLUP 489
SAVE TRANSACTION 194
SELECT 122
SET 95, 174, 339
SHOWPLAN_ALL 458
SHRINKDATABASE 438
SHRINKFILE 438
STATISTICS IO 455
STATISTICS PROFILE 461
TOP 128
TRUNCATE TABLE 186
UNION 160
UPDATE 178, 253, 289, 383
UPDATETEXT 495
USE 121
WAITFOR 220
WHERE 123, 130
WHILE 216
WRITETEXT 495
Операции математические 153
Оповещение:
  обновление 335
  создание 331
  удаление 335

Опция:
FILLFACTOR 80
PAD_INDEX 81
Ортогональное объединение 144, 150
Отключение базы данных 439
Ошибка 366

П

Параметр:
ALL 127
ANSI_DEFAULTS 349
ANSI_NULL_DFLT 349
ANSI_NULLS 350
ANSI_PADDING 350
ANSI_WARNINGS 351
BULK_LOGGED 96
COLLATE 21, 34
CURSOR_CLOSE_ON_COMMIT 96, 348
DATEFIRST 346
DATEFORMAT 347
DISTINCT 127
ENCRYPTION 266
FILEGROWTH 21
FILENAME 20
FORCEPLAN 348
MAXSIZE 21
MULTI_USER 96
OFFLINE 96
ONLINE 96
PRIMARY 20
READ_ONLY 96
READ_WRITE 96
RECOMPILE 266
RECOVERY FULL 96
RESTRICTED_USER 96
SIMPLE 96
SINGLE_USER 95
SIZE 20
Первичный ключ 58
Переименование базы данных 95
Переменная:
глобальная 199
локальная 197
Планировщик:
обновление 328
создание 324, 539
удаление 329
Подзапрос 162
Подключение базы данных 33, 440
Поиск 509
Поле:
вычисляемое 82
добавление 100
ключевое 37
псевдоним 128
удаление 101
Права доступа 396
Правило:
создание 113
удаление 113
Представление 246
имя 249
создание 246
удаление 253
Приращение файла 21, 26
Пространство 22
Процедура:
sp_add_alert 331
sp_add_job 300, 536
sp_add_jobschedule 324, 539
sp_add_jobstep 306, 538
sp_add_notification 336
sp_add_operator 302
sp_addlinkedserver 448
sp_addlinkedsrvlogin 450
sp_addlogin 388
sp_addmessage 330, 369
sp_addrole 394
sp_addrolemember 394
sp_addsrvrolemember 394
sp_addtype 372
sp_addumpdevice 412
sp_adduser 390
sp_apply_job_to_targets 313, 537
sp_attach_db 441
sp_autostats 386
sp_bindefault 112
sp_bindrule 113

sp_changegroup 395
sp_configure 296, 365
sp_dboption 361
sp_delete_alert 335
sp_delete_job 320
sp_delete_jobschedule 329
sp_delete_notification 338
sp_delete_operator 306
sp_denylogin 390
sp_depends 297
sp_detach_db 439
sp_droplogin 389
sp_dropmessage 331
sp_droprole 394
sp_droprolemember 394
sp_dropuser 391
sp_executesql 471
sp_fulltext_catalog 511
sp_fulltext_column 514
sp_fulltext_database 511
sp_fulltext_table 512
sp_grantdbaccess 391
sp_grantlogin 389
sp_help 74, 360
sp_help_alert 335
sp_help_fulltext_catalogs 516
sp_help_fulltext_tables 516
sp_help_job 315
sp_help_jobschedule 329
sp_help_jobstep 318
sp_help_operator 305
sp_helpdb 91, 361
sp_helpindex 374
sp_helprole 395
sp_helpserver 362
sp_helptext 298
sp_helptrigger 297
sp_helpuser 392
sp_password 389
sp_revokelogin 390
sp_settriggerorder 294
sp_start_job 314
sp_tableoption 39
sp_unbindefault 112
sp_unbinrule 113
sp_update_alert 335
sp_update_job 319
sp_update_jobschedule 328
sp_update_jobstep 320
sp_update_notification 338
sp_update_operator 304
sp_updatestats 386
xp_cmdshell 464
xp_enumgroups 468
xp_logevent 470
xp_logininfo 466
xp_msver 468
расширенная 464
хранимая 255
Псевдоним 128, 144

Р

Расширенная процедура 464
Регистрация таблиц 512
Резервное копирование 405
базы данных 410, 531
журнала транзакций 419, 533
файлов и файловых групп 416
Репликация 45
Роль 392
добавление пользователя 394
информация 395
серверная 392
создание 394
удаление 394
удаление пользователя 395

С

Свойства базы данных 95
Связь между таблицами 142
многие-ко-многим 66
один-к-одному 65
один-ко-многим 60
Соединение с сервером 9, 13
Создание:
базы данных 18
задания 300, 536
значения по умолчанию 112
(см. окончание рубрики на стр. 564)

Создание *(окончание)*:
индекса 76
каталога 511
оператора 302
оповещения 331
планировщика 324, 539
правила 113
представления 246
процедуры 257
роли 394
связанного сервера 448
таблицы 35
триггера 280
уведомления 336
функции 268
шага задания 306, 538
Сопоставление 21, 34
Сортировка 155
Составной индекс 78
Страница 22

Т

Таблица:
временная 85, 139
имя 125
регистрация 512
создание 35
удаление 42
Типы данных 372, 549
Транзакция 187
Триггер 279
изменение 283
создание 280
удаление 285

У

Уведомление:
обновление 338
создание 336
удаление 338
Увеличение автоматическое 43
Удаление:
базы данных 23
задания 320
записи 63, 183
значения по умолчанию 113
индекса 77
ограничения 102
оператора 306
оповещения 335
планировщика 329
поля 101
правила 113
представления 253
процедуры 259
роли 394
таблицы 42
триггера 285
уведомления 338
учетной записи 389, 390
файла 94
файловой группы 95
функции 279
Уникальность 37
Учетная запись:
добавление 388, 390
удаление 389, 390

Ф

Файл:
добавление 94
приращение 21, 26
удаление 94
Файловая группа 28
добавление 94
удаление 95
Формат даты 203, 204
Фрагментация 374
Функция 267
ABS() 236
ACOS() 239
ASIN() 240
ATAN() 240
AVG() 154
CAST() 200
CEILING() 237
COLUMNPROPERTY() 355
CONVERT() 202
COS() 240

COUNT() 154
DATABASEPROPERTYEX() 353
DATEDIFF() 207
DATENAME() 208
DATEPART() 208
DB_ID() 352
DB_NAME() 352
EXP() 239
FLOOR() 238
FORMSOF() 520
GETDATE() 205, 206
IDENT_INCR() 359
IDENT_SEED() 359
ISABOUT() 520
ISNULL() 220
LEFT() 228
LEN() 229
LOG() 239
LTRIM() 230
MAX() 154
MIN() 154
OBJECT_ID() 355
OPENDATASOURCE() 446
OPENROWSET() 444
PATINDEX() 231
PI() 239
RAISERROR() 366
RAND() 239
REPLACE() 231
REPLICATE() 232
REVERSE() 233
ROUND() 238
RTRIM() 230
SCOPE_IDENTITY() 359
SIGN() 237
SIN() 240
SPACE() 234
SQRT() 240
SQUARE() 240
STR() 234
STUFF() 235
SUBSTRING() 227
SUM() 154
TAN() 240
UPPER() 229
возвращающая таблицу 268, 272
изменение 277
многооператорная 269, 274
скалярная 268, 270
создание 268
удаление 279

Х

Хранимая процедура 255
выполнение 259
изменение 264
создание 257
удаление 259

Ц

Целостность данных 105, 293

Ш

Шаблон 140
Шаг задания:
обновление 320
создание 306, 538
Шифрование 267