

Microsoft SQL Server 2012



- Установка, настройка, администрирование и разработка
- Описание синтаксиса и семантики языка Transact-SQL в нотациях Бэкуса — Наура и при помощи R-графов
- Практические рекомендации по созданию баз данных
- Типы данных, включая XML, пространственные и пользовательские
- Манипулирование данными и управление транзакциями



Материалы
на www.bhv.ru

Наиболее
полное
руководство

В ПОДЛИННИКЕ®

Александр Бондарь

Microsoft SQL Server 2012

Санкт-Петербург
«БХВ-Петербург»
2013

УДК 004.65

ББК 32.973.26-018.2

Б81

Бондарь А. Г.

Б81 Microsoft SQL Server 2012. — СПб.: БХВ-Петербург, 2013. — 608 с.: ил. —
(В подлиннике)

ISBN 978-5-9775-0501-7

Книга посвящена установке, настройке, администрированию и разработке баз данных с помощью СУБД SQL Server 2012. Рассмотрено создание базы данных и основных ее объектов: таблиц, индексов, представлений, хранимых процедур и функций, триггеров и др. Показана работа средств отображения объектов и их характеристики. Описаны типы данных, включая XML, пространственные и пользовательские данные. Приведены синтаксис и семантика языка Transact-SQL в нотациях Бэкуса — Наура и при помощи R-графов. Подробно рассмотрены характеристики и взаимодействия транзакций. Уделено внимание средствам копирования и восстановления базы данных. В ходе создания учебной базы данных описаны примеры использования операторов манипулирования данными, триггеров, хранимых процедур и др. Исходные коды примеров размещены на сайте издательства.

Для программистов

УДК 004.65

ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Екатерина Капалыгина</i>
Редактор	<i>Юрий Рожко</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Марины Дамбиевой</i>

Подписано в печать 29.12.12.

Формат 70×100¹/16. Печать офсетная. Усл. печ. л. 49,02.

Тираж 1400 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

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

ISBN 978-5-9775-0501-7

© Бондарь А. Г., 2013

© Оформление, издательство "БХВ-Петербург", 2013

Оглавление

Введение.....	1
Организация книги	3
Благодарности	6
Дополнительные материалы	7
Глава 1. Инсталляция MS SQL Server 2012.....	9
Глава 2. Общие сведения о SQL Server 2012	27
2.1. Реляционные базы данных	27
2.1.1. Таблицы	28
2.1.1.1. Основные свойства и характеристики таблиц.....	28
2.1.1.2. Ключи в таблицах	31
2.1.2. Представления	33
2.1.3. Хранимые процедуры и триггеры	34
2.1.4. Пользователи, привилегии и роли базы данных	35
2.1.5. Задание первичных ключей таблиц.....	35
2.1.6. Транзакции	37
2.1.7. 12 правил Кодда.....	37
2.2. Реализация отношений в реляционной модели	38
2.2.1. Отношение "один к одному"	38
2.2.2. Отношение "один ко многим"	39
2.2.3. Отношение "многие ко многим"	40
2.3. Нормализация таблиц	41
2.3.1. Цель нормализации таблиц.....	41
2.3.2. Первая нормальная форма	41
2.3.3. Вторая нормальная форма	43
2.3.4. Третья нормальная форма	43
2.3.5. Другие нормальные формы	44
2.3.6. Денормализация таблиц	45
2.4. Проектирование баз данных	45
2.5. Язык Transact-SQL	46
2.5.1. Синтаксис	47
2.5.2. Основные сведения о составе языка Transact-SQL.....	56
Что будет дальше?	57

Глава 3. Работа с базами данных	59
3.1. Запуск и останов экземпляра сервера	60
3.1.1. Запуск на выполнение экземпляра сервера	60
3.1.2. Останов экземпляра сервера	65
3.2. Что собой представляет база данных в SQL Server.....	65
3.2.1. Системные базы данных	67
3.2.2. Базы данных пользователей.....	69
3.2.3. Некоторые характеристики базы данных	70
3.2.3.1. Владелец базы данных (Owner).....	70
3.2.3.2. Порядок сортировки (collation)	70
3.2.3.3. Возможность изменения данных базы данных	71
3.2.3.4. Состояние базы данных (Database State)	71
3.2.4. Некоторые характеристики файлов базы данных	72
3.2.4.1. Основные характеристики файлов базы данных	72
3.2.4.2. Состояния файлов базы данных	72
3.3. Получение сведений о базах данных и их файлах в текущем экземпляре сервера	73
3.3.1. Системное представление <i>sys.databases</i>	73
3.3.2. Системное представление <i>sys.master_files</i>	74
3.3.3. Системное представление <i>sys.database_files</i>	75
3.3.4. Системное представление <i>sys.filegroups</i>	77
3.3.5. Другие средства получения сведений об объектах базы данных	77
3.3.5.1. Системные представления	78
3.3.5.2. Системные хранимые процедуры	79
3.3.5.3. Системные функции	79
3.4. Создание и удаление базы данных	80
3.4.1. Использование операторов Transact-SQL для создания, отображения и удаления баз данных.....	80
3.4.1.1. Оператор создания базы данных	80
3.4.1.2. Оператор удаления базы данных.....	90
3.4.1.3. Создание и отображение баз данных в командной строке.....	91
3.4.1.4. Создание и отображение баз данных в Management Studio	113
3.4.2. Создание базы данных с использованием диалоговых средств Management Studio	123
3.5. Изменение базы данных	127
3.5.1. Изменение базы данных в языке Transact-SQL	128
3.5.1.1. Изменение имени базы данных	128
3.5.1.2. Изменение порядка сортировки	129
3.5.1.3. Изменение файлов базы данных	132
3.5.1.4. Изменение файловых групп.....	137
3.5.1.5. Изменение других характеристик базы данных.....	139
3.5.2. Изменение базы данных диалоговыми средствами Management Studio	141
3.5.2.1. Изменение имени базы данных	141
3.5.2.2. Изменение файлов базы данных	141
3.5.2.3. Изменение файловых групп базы данных	145
3.5.2.4. Изменение других характеристик базы данных	147
3.5.2.5. Отображение отчета использования дискового пространства базы данных	149
3.5.3. Удаление базы данных диалоговыми средствами Management Studio	150

3.6. Создание автономной базы данных.....	150
3.6.1. Установка допустимости автономных баз данных	150
3.6.2. Создание автономной базы данных и пользователя средствами языка Transact-SQL.....	152
3.6.3. Создание автономной базы данных диалоговыми средствами Management Studio	153
3.6.4. Создание автономного пользователя в Management Studio	153
3.6.5. Соединение с автономной базой данных в Management Studio	155
3.7. Присоединение базы данных	156
3.7.1. Присоединение базы данных с использованием Transact-SQL	156
3.7.2. Присоединение базы данных с использованием диалоговых средств Management Studio	159
3.7.3. Отсоединение базы данных	161
3.8. Создание мгновенных снимков базы данных.....	162
3.9. Схемы базы данных	164
3.9.1. Работа со схемами в Transact-SQL.....	164
3.9.2. Работа со схемами в Management Studio.....	168
3.10. Средства копирования и восстановления баз данных	170
3.10.1. Использование операторов копирования/восстановления базы данных	171
3.10.2. Использование диалоговых средств Management Studio для копирования/восстановления базы данных	172
3.11. Домашнее задание	176
Что будет дальше?	176

Глава 4. Типы данных	177
4.1. Классификация типов данных в SQL Server	178
4.2. Объявление локальных переменных	180
4.3. Числовые типы данных	181
4.3.1. Тип данных <i>BIT</i>	183
4.3.2. Целочисленные типы данных <i>TINYINT</i> , <i>SMALLINT</i> , <i>INT</i> , <i>BIGINT</i>	186
4.3.3. Дробные числа <i>NUMERIC</i> , <i>DECIMAL</i> , <i>SMALLMONEY</i> , <i>MONEY</i>	188
4.3.4. Числа с плавающей точкой <i>FLOAT</i> , <i>REAL</i>	192
4.3.5. Функции для работы с числовыми данными.....	192
4.4. Символьные данные	197
4.4.1. Символьные строки <i>CHAR</i> , <i>VARCHAR</i>	198
4.4.2. Символьные строки <i>NCHAR</i> , <i>NVARCHAR</i>	199
4.4.3. Типы данных <i>VARCHAR(MAX)</i> , <i>NVARCHAR(MAX)</i> , <i>VARBINARY(MAX)</i>	200
4.4.4. Строковые функции.....	200
4.5. Типы данных даты и времени	212
4.5.1. Описание типов данных даты и времени.....	212
4.5.2. Действия с датами и временем	213
4.6. Двоичные данные.....	224
4.7. Пространственные типы данных	225
4.7.1. Тип данных <i>GEOMETRY</i>	226
4.7.1.1. Точка.....	226
4.7.1.2. Ломаная линия	231
4.7.1.3. Полигон	235
4.7.1.4. Другие геометрические объекты.....	237
4.7.2. Тип данных <i>GEOGRAPHY</i>	238

4.8. Другие типы данных	243
4.8.1. Тип данных <i>SQL_VARIANT</i>	243
4.8.2. Тип данных <i>HIERARCHYID</i>	247
4.8.3. Тип данных <i>UNIQUEIDENTIFIER</i>	252
4.8.4. Тип данных <i>CURSOR</i>	254
4.8.5. Тип данных <i>TABLE</i>	261
4.8.6. Тип данных <i>XML</i>	262
4.9. Создание и удаление пользовательских типов данных	276
4.9.1. Синтаксис оператора создания пользовательского типа данных	276
4.9.2. Создание псевдонима средствами Transact-SQL	280
4.9.3. Создание псевдонима в диалоговых средствах Management Studio	280
4.9.4. Создание пользовательского табличного типа данных средствами Transact-SQL	281
4.9.5. Создание пользовательского табличного типа данных диалоговыми средствами Management Studio	285
4.9.6. Удаление пользовательского типа данных	286
Что будет дальше?	288

Глава 5. Работа с таблицами 289

5.1. Синтаксис оператора создания таблицы	290
5.1.1. Общие характеристики таблицы	291
5.1.1.1. Идентификатор таблицы	291
5.1.1.2. Предложение <i>AS FileTable</i>	292
5.1.1.3. Определение столбца, вычисляемого столбца, набора столбцов	292
5.1.1.4. Предложение <i>ON</i>	292
5.1.1.5. Предложение <i>TEXTIMAGE_ON</i>	293
5.1.1.6. Предложение <i>FILESTREAM_ON</i>	293
5.1.1.7. Предложение <i>WITH</i>	293
5.1.2. Определение столбца	295
5.1.2.1. Имя столбца	296
5.1.2.2. Тип данных	296
5.1.2.3. Ключевое слово <i>FILESTREAM</i>	296
5.1.2.4. Предложение <i>COLLATE</i>	296
5.1.2.5. Ключевые слова <i>NULL / NOT NULL</i>	296
5.1.2.6. Предложение <i>DEFAULT</i>	296
5.1.2.7. Ключевое слово <i>IDENTITY</i>	297
5.1.2.8. Ключевое слово <i>ROWGUIDCOL</i>	298
5.1.2.9. Ключевое слово <i>SPARSE</i>	298
5.1.3. Ограничения столбца и ограничения таблицы	298
5.1.3.1. Имя ограничения	299
5.1.3.2. Ограничения первичного и уникального ключа	299
5.1.3.3. Ограничение внешнего ключа	302
5.1.3.4. Ограничение <i>CHECK</i>	306
5.1.4. Вычисляемые столбцы	307
5.1.5. Набор столбцов	309
5.2. Простые примеры таблиц	310
5.3. Создание секционированных таблиц	322
5.3.1. Синтаксические конструкции	323
5.3.2. Пример создания секционированной таблицы	327

5.3.3. Отображение результатов создания таблицы.....	336
5.3.4. Изменение характеристик секционированной таблицы	340
5.4. Создание таблиц диалоговыми средствами.....	342
5.4.1. Создание таблицы секционирования	342
5.4.2. Создание таблицы секционирования, схемы секционирования и функции секционирования	352
5.5. Отображение состояния секционированных таблиц	358
5.6. Файловые потоки	358
5.7. Удаление таблиц	364
5.7.1. Определение зависимостей таблицы	364
5.7.2. Удаление таблицы оператором <i>DROP TABLE</i>	368
5.7.3. Удаление таблицы диалоговыми средствами Management Studio.....	368
5.8. Изменение характеристик таблиц.....	371
5.8.1. Изменение таблиц при использовании оператора Transact-SQL.....	371
5.8.1.1. Имя таблицы	374
5.8.1.2. Изменение столбца	374
5.8.1.3. Изменение типа данных	375
5.8.1.4. Изменение порядка сортировки	377
5.8.1.5. Добавление нового столбца (обычного или вычисляемого).....	377
5.8.1.6. Добавление ограничения	377
5.8.1.7. Удаление столбца	377
5.8.1.8. Удаление ограничения	378
5.8.2. Изменение таблиц средствами Management Studio.....	378
5.8.2.1. Изменение имени таблицы	378
5.8.2.2. Изменение столбца	378
5.8.2.3. Изменение типа данных	380
5.8.2.4. Изменение порядка сортировки	384
5.8.2.5. Изменение формулы для вычисляемого столбца.....	385
5.8.2.6. Добавление нового столбца	385
5.8.2.7. Добавление и изменение ограничений	385
5.8.2.8. Удаление столбца	399
5.8.2.9. Удаление ограничений	402
5.9. Файловые таблицы.....	405
Что будет дальше?	408
Глава 6. Индексы	409
6.1. Отображение индексов	410
6.2. Работа с индексами средствами Transact-SQL	411
6.2.1. Создание обычного (реляционного) индекса	411
6.2.2. Создание индекса для представлений	420
6.2.3. Создание индекса columnstore	420
6.2.4. Создание индекса для столбца XML	422
6.2.5. Создание пространственного индекса	428
6.2.6. Удаление индекса	433
6.2.7. Изменение индекса	435
6.3. Работа с индексами с помощью диалоговых средств Management Studio	438
6.3.1. Создание индекса в Management Studio	438
6.3.2. Удаление индекса в Management Studio.....	443
6.3.3. Изменение индекса в Management Studio	443
Что будет дальше?	443

Глава 7. Добавление, изменение и удаление данных.....	445
7.1. Обобщенное табличное выражение	445
7.2. Добавление данных (оператор <i>INSERT</i>).....	446
7.3. Изменение данных (оператор <i>UPDATE</i>).....	453
7.4. Удаление данных (оператор <i>DELETE</i>).....	457
7.5. Удаление строк таблицы (оператор <i>TRUNCATE TABLE</i>).....	459
7.6. Добавление, изменение или удаление строк таблицы (оператор <i>MERGE</i>).....	460
Что будет дальше?	467
Глава 8. Выборка данных	469
8.1. Оператор <i>SELECT</i>	469
8.2. Оператор <i>UNION</i>	480
8.3. Операторы <i>EXCEPT, INTERSECT</i>	481
8.4. Примеры выборки данных	481
8.4.1. Список выбора	481
8.4.2. Упорядочение результата (<i>ORDER BY</i>)	484
8.4.3. Условие выборки данных (<i>WHERE</i>).....	485
8.4.3.1. Использование операторов сравнения	485
8.4.3.2. Использование варианта <i>LIKE</i>	488
8.4.3.3. Использование варианта <i>BETWEEN</i>	489
8.4.3.4. Использование варианта <i>IN</i>	489
8.4.3.5. Использование функций <i>ALL, SOME, ANY, EXISTS</i>	491
8.4.4. Соединение таблиц	493
Внутреннее соединение.....	500
8.4.5. Группировка результатов выборки (<i>GROUP BY, HAVING</i>)	500
8.5. Использование операторов <i>UNION, EXCEPT, INTERSECT</i>	505
Что будет дальше?	507
Глава 9. Представления.....	509
9.1. Синтаксис операторов для представлений	510
9.1.1. Создание представления	510
9.1.2. Изменение представления	511
9.1.3. Удаление представления	512
9.2. Создание представлений в Transact-SQL	512
9.3. Создание представлений диалоговыми средствами Management Studio.....	515
Что будет дальше?	516
Глава 10. Транзакции.....	517
10.1. Понятие и характеристики транзакций	517
10.2. Операторы работы с транзакциями	518
10.3. Уровни изоляции транзакции	520
Что будет дальше?	522
Глава 11. Хранимые процедуры, функции, определенные пользователем, триггеры	523
11.1. Язык хранимых процедур и триггеров.....	524
11.2. Хранимые процедуры	528
11.2.1. Создание хранимой процедуры	528
11.2.2. Изменение хранимой процедуры	530

11.2.3. Удаление хранимой процедуры.....	531
11.2.4. Использование хранимых процедур	532
11.3. Функции, определенные пользователем	537
11.3.1. Создание функции	538
11.3.2. Изменение функций.....	539
11.3.3. Удаление функций.....	540
11.3.4. Использование функций.....	540
11.4. Триггеры.....	541
11.4.1. Создание триггеров	542
11.4.2. Изменение триггеров.....	545
11.4.3. Удаление триггеров	547
11.4.4. Использование триггеров.....	547
Приложение 1. 12 правил Кодда	551
Приложение 2. Зарезервированные слова Transact-SQL.....	555
Приложение 3. Утилита командной строки <i>sqlcmd</i>	561
Приложение 4. Характеристики базы данных.....	565
П4.1. Параметры <i>Auto</i> (в Management Studio — группа <i>Automatic</i>).....	568
П4.2. Параметры доступности базы данных (<i>Availability</i>)	569
П4.3. Параметры автономной базы данных (<i>Containment</i>)	571
П4.4. Параметры курсора (<i>Cursor</i>)	572
П4.5. Параметры восстановления (<i>Recovery, Recovery model</i>).....	572
П4.6. Общие параметры SQL (<i>Miscellaneous</i>)	573
П4.7. Параметры внешнего доступа (<i>External Access</i>).....	577
П4.8. Параметры компонента Service Broker	578
П4.9. Параметры изоляции транзакций для мгновенных снимков (<i>SNAPSHOT</i>).....	578
Приложение 5. Языки, представленные в SQL Server	581
Приложение 6. Описание электронного архива	589
Предметный указатель	591

Введение

Надо сказать, что MS SQL Server версии 2012 (да и многие предыдущие версии) является весьма сложной системой, имеющей огромные возможности. Большое количество программных компонентов, представлений просмотра каталогов, системных процедур, функций и других средств может сбить с толку. Кроме того, получить нужный вам результат можно множеством способов, разными путями, используя различные средства, существующие в системе. Я покажу те способы, которыми можно эффективно и без лишних затрат времени и интеллекта пользоваться для получения конкретного результата. В основном это те средства, которыми пользуюсь лично я или более достойные люди, очень хорошие специалисты в данной области.

При написании этой книги я в первую очередь хочу пообщаться с теми, кто никогда не работал ни с какой версией MS SQL Server, а может быть даже и вообще ни с какой системой управления базами данных — ни с реляционной, ни с сетевой, ни даже с иерархической, не говоря уж и о совсем простеньких ("настольных") системах управления данными. По этой причине здесь вы не найдете подробных сравнений настоящей версии сервера с предыдущими, детальных описаний того, что нового появилось в SQL Server 2012. Если же вам встретятся какие-то сравнения, то знайте, что это у меня вырвалось совершенно случайно.

Разработка баз данных и создание программ, использующих базы данных, включает *проектирование* баз данных, *создание* самой базы данных и всех необходимых для ее эффективного использования объектов (таблицы, индексы, хранимые процедуры, триггеры, функции), *поддержание базы данных* в работоспособном состоянии и, в конце концов, *создание программ* для так называемого "конечного пользователя" (end user), которому было бы комфортно работать с созданными вами базами данных. Специалистов, обслуживающих программные системы (системных администраторов или в более узком смысле администраторов баз данных, АБД), которые используют в своей работе базы данных SQL Server, я также постараюсь не обойти здесь вниманием. Вот только самим конечным пользователям любой программной системы данная книга, я полагаю, не нужна.

В соответствии с этим и структура книги отличается от других книг, посвященных подобным системам управления базами данных. Здесь в соответствующем порядке

описываются те действия, которые будет выполнять нормальный человек, которому нужно спроектировать, создать базу данных, затем заполнить эту базу своими данными, изменять, удалять данные и, наконец, отыскивать нужные ему данные. Следовательно, и в книге эти действия описываются в том же порядке: создание базы данных, создание таблиц, добавление, изменение, удаление, выборка данных. Это начало книги, затем идут описания других необходимых действий по работе с базами данных и соответствующие средства, представленные в системе, которых, надо сказать, огромное множество. Разумеется, все описать в одной книге просто невозможно. Я старался дать здесь те сведения, которые будут необходимы и достаточны для вашей эффективной работы в очень большом диапазоне задач, которые нам с необыкновенной щедростью поставляет жизнь и реальные потребности в обработке данных из различных предметных областей.

Есть у предлагаемой книги еще одна полезная особенность. Книга содержит необходимый материал для того, чтобы вам было проще подготовиться к соответствующим экзаменам, чтобы стать сертифицированным специалистом корпорации Microsoft в области работы с SQL Server, и именно SQL Server 2012. Однако структура книги отличается и от той структуры учебных материалов, которые предлагаются специалистами Microsoft. Надеюсь, в лучшую сторону.

Часто при описании достоинств какой-либо литературы по программированию говорится о том, что никаких предварительных особых знаний и умений от читателя не требуется, чтобы прочесть и понять соответствующую книгу. Про данную книгу я такого сказать (по причине врожденной честности) не могу. Для того чтобы эта книга была действительно полезна читателю, у него, у читателя, должен быть определенный запас знаний и умений, имеющих прямое отношение к программированию, обработке данных. На начальном этапе знакомства с подобными сложными системами мог бы порекомендовать прочитать любую подходящую книгу из серии "step-by-step" (шаг за шагом) или "for dummies" (в нашей переводной литературе называют "для чайников").

Однако и здесь бывают интересные исключения. Есть люди, настолько мотивированные на получение знаний в конкретной области человеческой деятельности, что готовы сломя голову броситься в изучение предмета, им мало знакомого, не начиная с чтения самых простых руководств. Я испытываю глубокое уважение к таким людям, тем более что знаю нескольких таких; это бывшие мои студенты, которые на сегодняшний день во многих областях программистской деятельности достигли впечатляющих результатов. По этой причине я все-таки взял на себя смелость в главе 2 чуть более подробно описать многие базовые моменты, связанные с системами управления базами данных — и касающиеся только реляционных баз данных. Теперь с некоторыми основаниями можно произнести такую фразу: "Если вы можете включить компьютер и запустить на выполнение указанную программу, то вы при желании с блеском освоите материал этой книги". Пожалуй, это не будет слишком большим преувеличением.

Если все эти относительно подробно описываемые основы реляционных баз данных вам хорошо знакомы (то, что все данные представлены в таблицах, наличие нормальных форм, способы нормализации таблиц, средства для описания синтакси-

сиса, назначение языка SQL и др.), не обижайтесь на меня за излишние, казалось бы, подробности, а просто пропустите ненужные вам описания и рассуждения. Хотя можете и бегло их просмотреть, не тратя слишком много времени.

В книге дается множество, может быть и чрезмерное количество, примеров для иллюстрации использования различных средств SQL Server. Иногда мне становится даже немного обидно за то, что я все за вас делаю (далеко не всегда, может быть, и лучшим образом).

Все действия я выполнял в операционной системе Windows 7.

Новая версия MS SQL Server 2012 может выполняться в следующих операционных системах:

- ◆ Windows 7, service pack 1 и выше;
- ◆ Windows Vista, service pack 2 и выше;
- ◆ Windows Server 2008, service pack 2 и выше;
- ◆ Windows Server 2008 R2, service pack 1 и выше.

Приведу краткую историю, точнее, годы создания и развития MS SQL Server.

- ◆ 1992 г. — SQL Server 4.2.
- ◆ 1993 г. — SQL Server 4.21.
- ◆ 1995 г. — SQL Server 6.0.
- ◆ 1996 г. — SQL Server 6.5.
- ◆ 1999 г. — SQL Server 7.0.
- ◆ 2000 г. — SQL Server 2000.
- ◆ 2005 г. — SQL Server 2005.
- ◆ 2008 г. — SQL Server 2008.
- ◆ 2010 г. — SQL Server 2008 R2.
- ◆ 2012 г. — SQL Server 2012.

Организация книги

Книга состоит из 11 глав и 6 приложений.

- ◆ *Глава 1 "Инсталляция MS SQL Server 2012"* посвящена описанию инсталляции SQL Server. Вроде бы и нет особых сложностей при инсталляции системы, в особенности для тех, кто неоднократно выполнял эти действия, однако некоторые моменты требуют пояснений, что я и попытался сделать при описании установки сервера баз данных на компьютере.
- ◆ *Глава 2 "Общие сведения о SQL Server 2012"*

Мне заранее неизвестно, насколько человек, читающий эту книгу, знаком с основами реляционных баз данных, с принятymi средствами описания синтаксиса формальных языков. Поэтому данная глава посвящена основным понятиям реляционных баз данных. Здесь кратко, наверное, очень кратко, описывается хранение данных в базе данных, нормальные формы, нормализация таблиц. Рас-

смотрены способы реализации отношений между данными в реляционных базах данных ("один к одному", "один ко многим", "многие ко многим"). Приводятся простые, но взятые из реальной жизни примеры реализации этих отношений, построенные на связке "внешний/первичный (уникальный) ключ".

В этой же главе описываются чуть измененные нотации Бэкуса — Наура, которые очень эффективно используются во всем цивилизованном мире для описания синтаксиса любых формальных языков, в том числе языков программирования и языка SQL — причем для любых серверов баз данных, будь это SQL Server, Oracle, IBM DB2, InterBase, Firebird, Sybase, PostgreSQL или даже MySQL (простите, если кого-то не упомянул). Я предлагаю и еще один удобный графический способ описания синтаксиса — R-графы. Здесь же даются описания базовых синтаксических конструкций — это идентификаторы (обычные и с разделителями), числа, строковые константы.

Кратко описываются объекты базы данных: таблицы, индексы, пользовательские типы данных, представления, хранимые процедуры, триггеры.

◆ Глава 3 "*Работа с базами данных*"

Поскольку разработчику баз данных нужно в первую очередь создать базу данных, которую он хочет заполнять данными и использовать эти данные для решения задач его предметной области, то эта глава посвящена именно вопросам создания, отображения, удаления и изменения баз данных. MS SQL Server является весьма сложной системой. Сами базы данных в ней имеют множество свойств, характеристик. Так как разработчику баз данных на начальных этапах своей деятельности нет острой необходимости вникать во все тонкости и детали организации данных, то в этой главе я не стал описывать сами базы данных слишком подробно. Однако у того же разработчика могут появиться потребности более детально разобраться с некоторыми характеристиками, которые позволяют более эффективно использовать вычислительные ресурсы и смогут повысить производительность системы. Поэтому достаточно подробное описание характеристик баз данных я поместил в *приложение 4*. В этой же главе, кроме того, описываются и файловые группы — в основном создание, изменение, удаление. Эффект использования файловых групп проявляется позже, когда в базу данных помещаются таблицы, начинается их заполнение и осуществляется выборка данных.

◆ Глава 4 "*Типы данных*"

Тип данных — важнейшее понятие в программировании вообще и в системах управления базами данных в частности. Поэтому в этой главе подробно (надеюсь, очень подробно) описываются все типы данных SQL Server. Приводятся операции над типами данных, допустимые преобразования данных, применяемые функции. Даётся синтаксис оператора создания пользовательских типов данных. Материал этой главы будет использоваться на протяжении всей книги.

◆ Глава 5 "*Работа с таблицами*"

Важнейший объект реляционной базы данных — таблица. В этой главе приводится синтаксис оператора создания таблицы. Даётся множество примеров соз-

дания таблиц из демонстрационной базы данных BestDatabase, которая на самом деле является упрощенным фрагментом промышленной базы данных. Подробно описывается задание столбцов таблицы и — важнейший момент в любой реляционной базе данных — задание ограничений для отдельных столбцов и для таблицы в целом. В некоторых случаях я мог бы поспорить с используемой в фирменной документации терминологией и наверняка не удержусь от критических замечаний, однако терминология эта устоявшаяся, и мы с вами будем следовать в русле "линии партии и правительства".

- ◆ *Глава 6 "Индексы"* рассматривает синтаксис операторов создания, изменения и удаления индексов, кластерные индексы, индексы для представлений. Приводятся примеры.
- ◆ *Глава 7 "Добавление, изменение и удаление данных"* описывает синтаксис и назначение операторов INSERT, UPDATE, DELETE, TRUNCATE TABLE, MERGE. Даются примеры их использования.
- ◆ *Глава 8 "Выборка данных"*

В главе подробно рассматривается самый, пожалуй, сложный оператор SELECT, позволяющий выбирать данные из одной или более таблиц. Приводятся примеры использования средств определения условий выборки данных, группирования результатов, выполнения соединения таблиц (внешних и внутреннего). Так же рассматриваются операторы UNION, EXCEPT, INTERSECT.

- ◆ *Глава 9 "Представления"* посвящена созданию, изменению и удалению представлений. Кроме того, рассмотрено назначение представлений, индексированные представления.

◆ *Глава 10 "Транзакции"*

В главе дается понятие транзакции. Описываются операторы для старта, подтверждения и отмены транзакции. Рассматриваются все уровни изоляции транзакций, используемые в MS SQL Server 2012.

- ◆ *Глава 11 "Хранимые процедуры, функции, определенные пользователем, триггеры"* посвящена описанию языковых средств Transact-SQL для создания и использования программных компонентов MS SQL Server — хранимых процедур, пользовательских функций и триггеров.

◆ *Приложение 1*

В приложении описываются 12 правил Кодда, которые, скорее всего, больше нужны разработчику СУБД, чем человеку, использующему систему.

◆ *Приложение 2*

В приложении приводится список зарезервированных слов языка Transact-SQL, которые нельзя использовать в обычных идентификаторах, а также не слишком рекомендуется применять и в идентификаторах с разделителями.

◆ *Приложение 3*

Это приложение исключительно для любителей работы с командной строкой, здесь кратко описываются параметры утилиты sqlcmd.

◆ Приложение 4

В приложении описывается множество характеристик базы данных, которые можно установить при первоначальном создании базы данных или при ее изменении.

◆ Приложение 5

Здесь описывается то очень большое количество языков, которые поддерживаются в MS SQL Server с некоторыми их характеристиками.

◆ Приложение 6

В этом приложении приводится описание электронного архива тех дополнительных материалов, которые потребуются для работы с книгой.

Теперь несколько слов о принятой в книге терминологии.

Название языка SQL почти все американские программисты (и не только программисты, и не только американские) произносят как "сиквел", но не как "эс-кью-эль". Понятно, это жаргон, жаргон профессиональный. Если в устном общении вы хотите быть ими поняты или желаете легко воспринимать их устную речь, то можете смело использовать тот же вариант произношения. Если вам повезет услышать речь американских специалистов по базам данных, вы также поймете, о чем они говорят. Собственно, я говорю о таком произношении лишь для того, чтобы никто не упрекнул меня в неправильном использовании *русского языка*. Вы видите, например, что *глава 2* называется "Общие сведения о SQL Server 2012". В случае иного произношения следовало бы писать не "о SQL", а "об SQL". Если вы обратите внимание и на англоязычную литературу по этой тематике, то также можете заметить, что перед аббревиатурой SQL всегда стоит неопределенный artikel **a**, а не **an**. Любопытно, что в ранней документации шведской фирмы AB по MySQL (пока эта фирма не отдала MySQL фирме Oracle) ставится все же artikel **an**. Похоже, они мало общались с американскими специалистами в отличие от нас с вами.

Второй терминологический момент связан с переводом на русский язык отдельных английских терминов. Здесь у меня встречаются некоторые расхождения с переводом, выполненным специалистами из Microsoft. Я имею в виду в первую очередь русскоязычный вариант Books Online. Например, слово "statement" они переводят как "инструкция", я же использую принятый во всей отечественной литературе по базам данных вариант "оператор". Подобных отличий не так уж много, но они существуют.

Благодарности

Автор благодарен специалистам, которые просматривали отдельные главы этой книги и высказали довольно большое количество замечаний, дали немало советов, как избежать неточностей и улучшить содержание. Это Алексей Шуленин (корпорация Майкрософт, Москва), Дмитрий Артемов (корпорация Майкрософт, Москва), Олег Меркулов (ОАО "ВНИПИГаздобыча", Саратов). Со многими вопросами помогал мне разбираться Денис Резник (компания Digital Cloud Technologies, Украина, Харьков).

Особая благодарность Дмитрию Ермишину (корпорация Майкрософт, Саратов) за ту техническую поддержку, которую он постоянно мне оказывал и оказывает.

Дополнительные материалы

Скрипты для создания учебной базы данных, используемой в этой книге, и заполнения ее данными можно скачать с сайта издательства "БХВ-Петербург":

ftp://ftp.bhv.ru/9785977505017.zip

Описание скриптов находится в *приложении 6*. Имя каждого скрипта начинается с его порядкового номера. Для создания базы данных BestDatabase, всех ее объектов и для заполнения таблиц данными нужно выполнять скрипты именно в этом порядке.



ГЛАВА 1

Инсталляция MS SQL Server 2012

В многочисленных литературных источниках вы можете найти описание различных версий, точнее реализаций SQL Server. Там описываются возможности, стоимость различных вариантов. Если для вашей деятельности нужна система с конкретными возможностями и не превышающая по стоимости определенной суммы, то обратитесь к соответствующей литературе, а лучше, непосредственно на сайт корпорации Microsoft.

Здесь же мы будем использовать "trialную" (от англ. *trial*), т. е. пробную, версию сервера базы данных, которая позволит выполнить все необходимые действия по ее освоению в течение 180 дней с момента ее инсталляции.

ВНИМАНИЕ!

Если у вас на компьютере были установлены предыдущие версии SQL Server и вы выполнили их деинсталляцию, то после деинсталляции программ в каталоге `c:\Program Files\Microsoft SQL Server\` останутся файлы, которые могут помешать новой установке. Следует удалить их вручную. Иначе при новой инсталляции вы можете получить неприятные сообщения об ошибках.

Для установки на вашем компьютере SQL Server 2012 запустите на выполнение программу установки `setup.exe`.

Вначале появится окно-заставка (рис. 1.1). Это окно будет регулярно появляться на вашем мониторе в процессе инсталляции.



Рис. 1.1. Окно-заставка SQL Server 2012

Через некоторое время появится основное окно инсталляции — **SQL Server Installation Center** (рис. 1.2).



Рис. 1.2. Основное окно инсталляции. Вкладка **Planning**

В этом окне нужную вкладку можно выбрать в левой части окна. Текущей является вкладка **Planning** (планирование). В окне она выделена полужирным шрифтом.

Чтобы выполнить проверку, насколько оборудование вашего компьютера и установленные программные средства соответствуют тем требованиям, которые предъявляются для SQL Server 2012, нужно мышью на вкладке **Planning** щелкнуть по строке **System Configuration Checker** (проверка конфигурирования системы).

После выполнения проверки появится окно **Setup Support Rules** (правила поддержки установки), содержащее итоговые результаты (рис. 1.3). Если детали проверки не видны, то нужно щелкнуть по кнопке **Show details >>** (показать детали, подробности). Чтобы можно было начинать процесс инсталляции, следует убедиться, что количество ошибок (Failed) является нулевым. Количество предупреждающих сообщений (Warning) позволяет выполнять инсталляцию, однако в дальнейшем эти предупреждения могут оказаться на качестве работы и функциональности сервера базы данных или операционной системы.

После щелчка по кнопке **OK** опять появится основное окно инсталляции (см. рис. 1.2). Для выбора следующей вкладки в левой верхней части окна щелкните мышью по строке **Installation** (инсталляция). В результате откроется вкладка **Installation** (рис. 1.4).

В этом окне в правой его части нужно щелкнуть мышью по самой первой строке — **New SQL Server stand-alone installation or add features to an existing installation** (новая инсталляция или добавление новых возможностей к существующей инсталляции). Вначале будут выполнены необходимые проверки установленных на компьютере программных средств, результат которых будет отображен в окне **Setup Support Rules** (правила поддержки установки) (рис. 1.5).

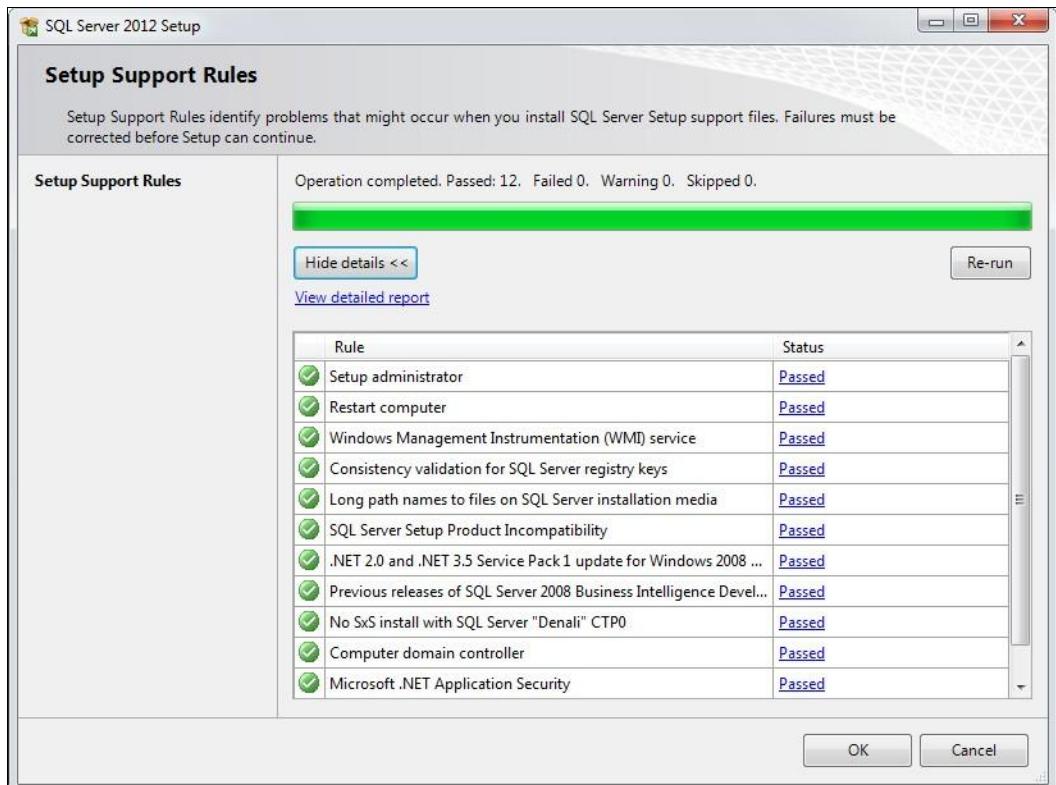


Рис. 1.3. Результат проверки конфигурации системы



Рис. 1.4. Основное окно инсталляции. Вкладка Installation

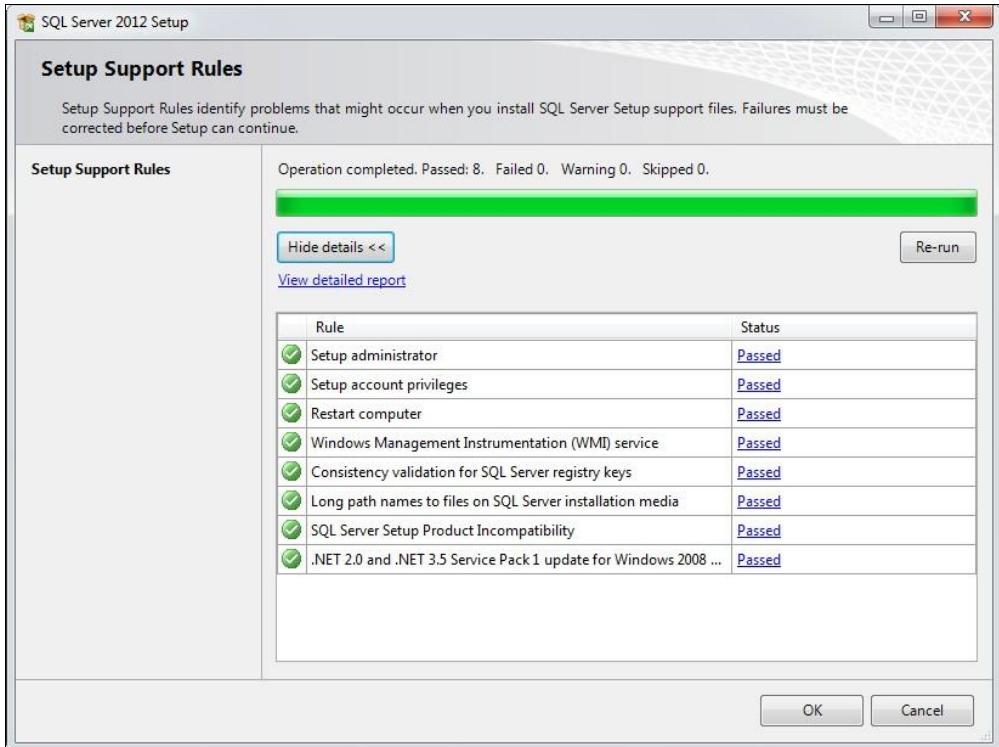


Рис. 1.5. Результат выполненных проверок в окне **Setup Support Rules**

Если количество несоответствий (Failed) равно нулю, то инсталляция может быть продолжена. Щелкните по кнопке **OK**.

В следующем окне, **Product Key** (ключ продукта), нужно выбрать переключатель **Specify a free edition** (задать бесплатную версию) (рис. 1.6). В раскрывающемся списке нужно оставить **Enterprise Evaluation**, чтобы устанавливалась пробная полная версия, работоспособная в течение 180 дней. Кстати, если захочется, из этого раскрывающегося списка вы можете выбрать и бесплатную версию **Express Edition**.

Если же у вас есть лицензионный ключ, то нужно выбрать переключатель **Enter the product key** и в следующем далее поле ввести значение этого ключа.

Щелкните по кнопке **Next** (далее). Следующим будет окно лицензионного соглашения **License Terms** (условия лицензии) (рис. 1.7).

Я настоятельно рекомендую его все-таки прочесть. После этого отметьте флажок **I accept the license terms** (я принимаю условия лицензионного соглашения) и щелкните по кнопке **Next** (далее). Следующим будет окно **Install Setup Files** (установка файлов инсталляции) рис. 1.8.

Здесь от вас никаких действий не требуется. Через некоторое время это окно исчезнет и будет отображено окно **Setup Support Rules** (правила поддержки установки), содержащее сведения о результатах установки правил поддержки (рис. 1.9).

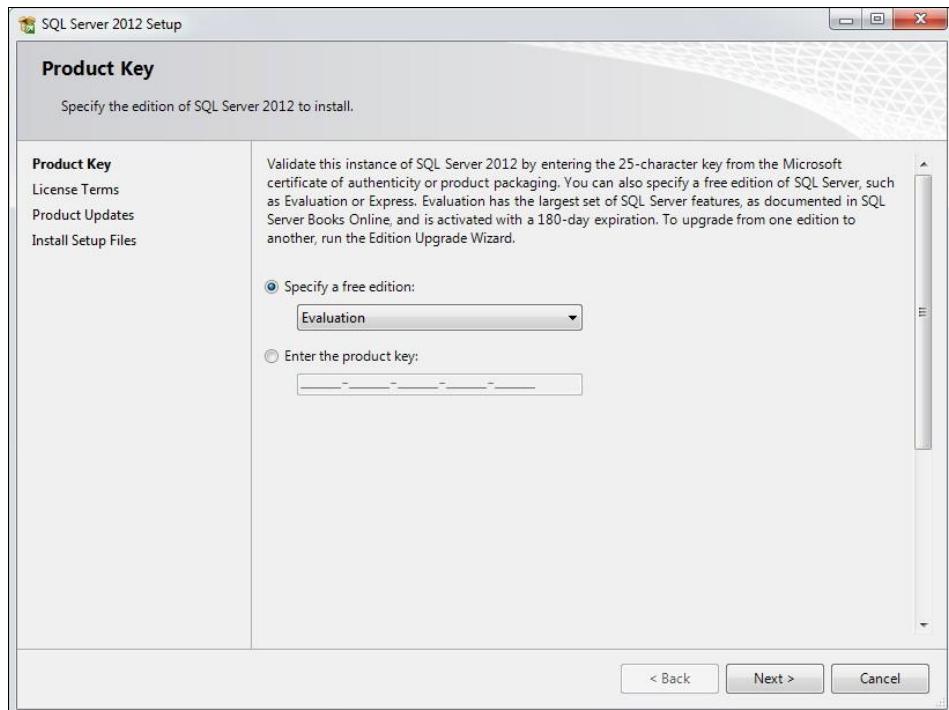


Рис. 1.6. Выбор устанавливаемой версии SQL Server

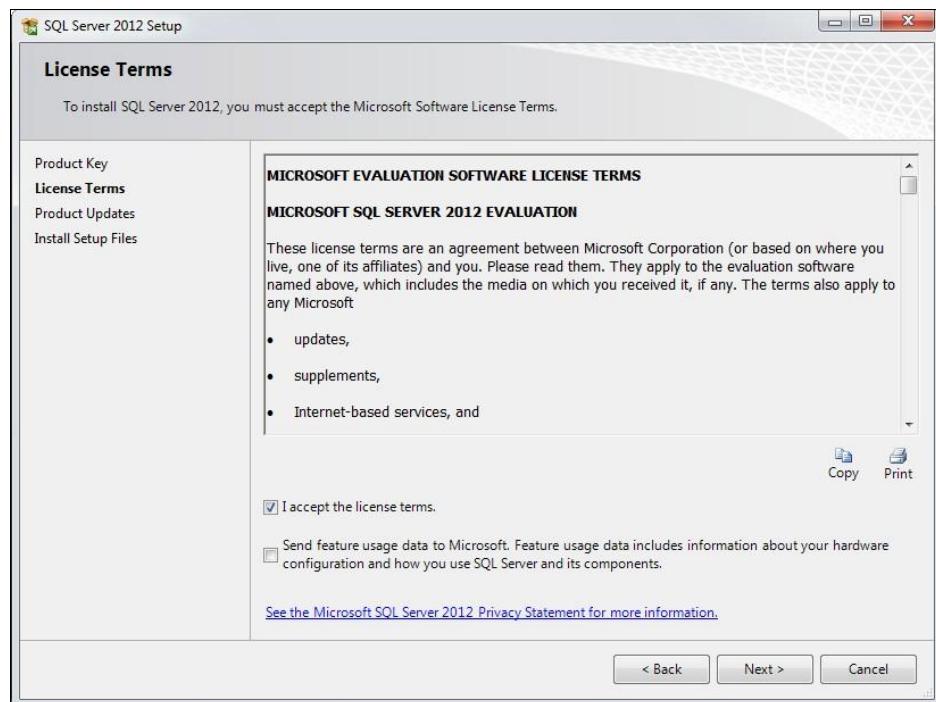


Рис. 1.7. Лицензионное соглашение

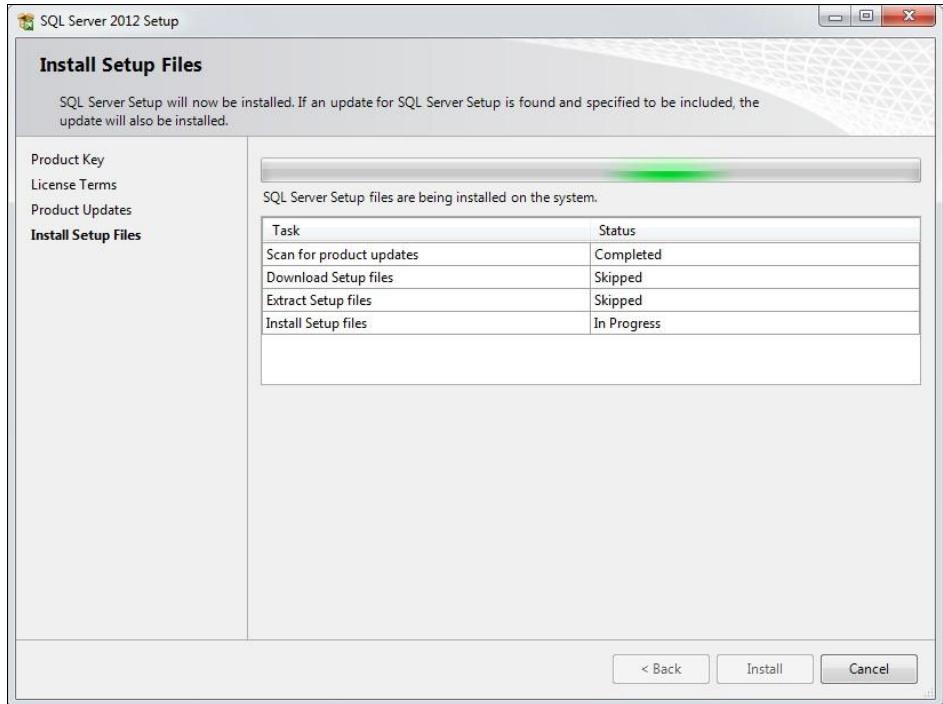


Рис. 1.8. Установка файлов инсталляции

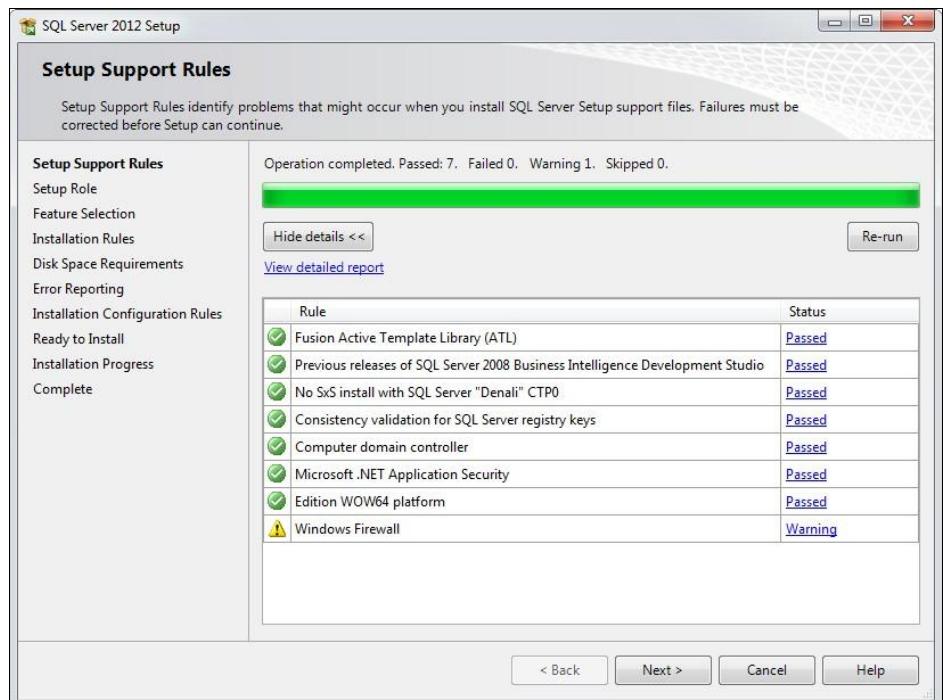


Рис. 1.9. Результат установки правил поддержки

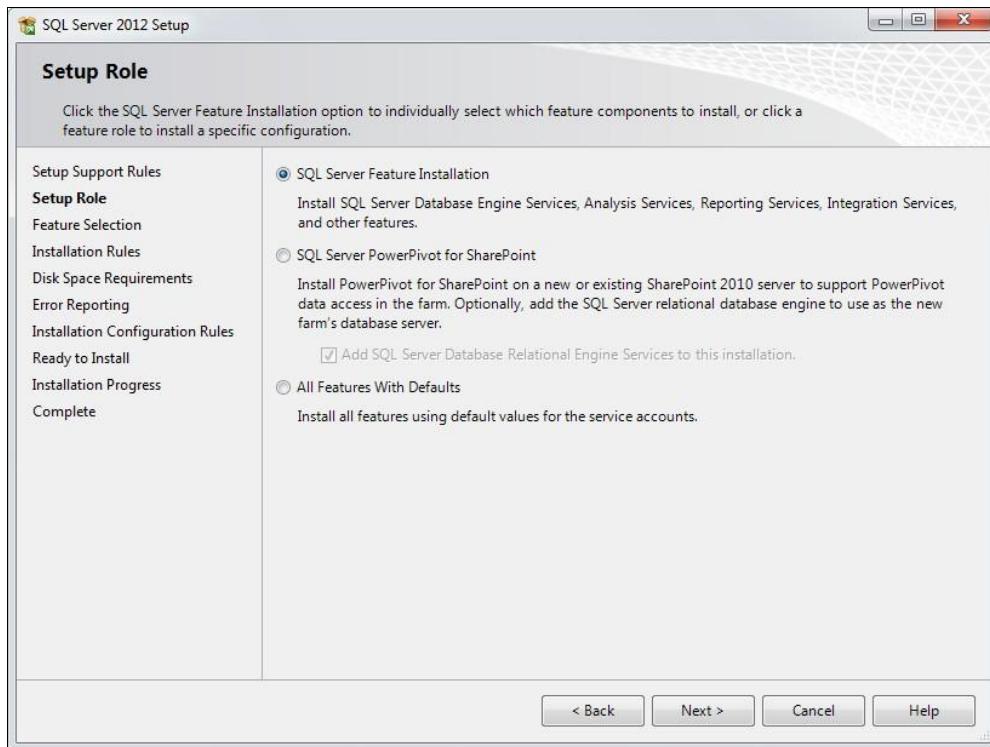


Рис. 1.10. Вид установки

Щелкните по кнопке **Next** (далее). В следующем окне **Setup Role** (роль установки) нужно задать вид установки (рис. 1.10).

Выберите переключатель **SQL Server Feature Installation** (компоненты установки SQL Server). Щелкните по кнопке **Next**. После этого откроется следующее окно **Feature Selection** (выбор компонентов), которое позволит явно указать устанавливаемые компоненты SQL Server (рис. 1.11).

Для выбора установки всех компонентов щелкните по кнопке **Select All** (выбрать все), после чего нажмите кнопку **Next** (далее). После этого будут выполнены очередные проверки состояния программных средств компьютера. Результат отобразится в окне **Installation Rules** (правила установки) (рис. 1.12).

Если не будет обнаружено ошибок, щелкните по кнопке **Next** (далее).

Далее появится окно задания имени экземпляра устанавливаемого сервера **Instance Configuration** (настройка экземпляра) (рис. 1.13).

Выберите переключатель **Default instance** (экземпляр по умолчанию), тогда идентификатором устанавливаемого сервера базы данных (Instance ID) будет **MSSQLSERVER**. Щелкните мышью по кнопке **Next** (далее).

Появится окно, описывающее требования к дисковому пространству компьютера **Disk Space Requirements** (требования к свободному месту на диске) (рис. 1.14).

Здесь ничего выбирать не нужно, просто щелкните по кнопке **Next** (далее).

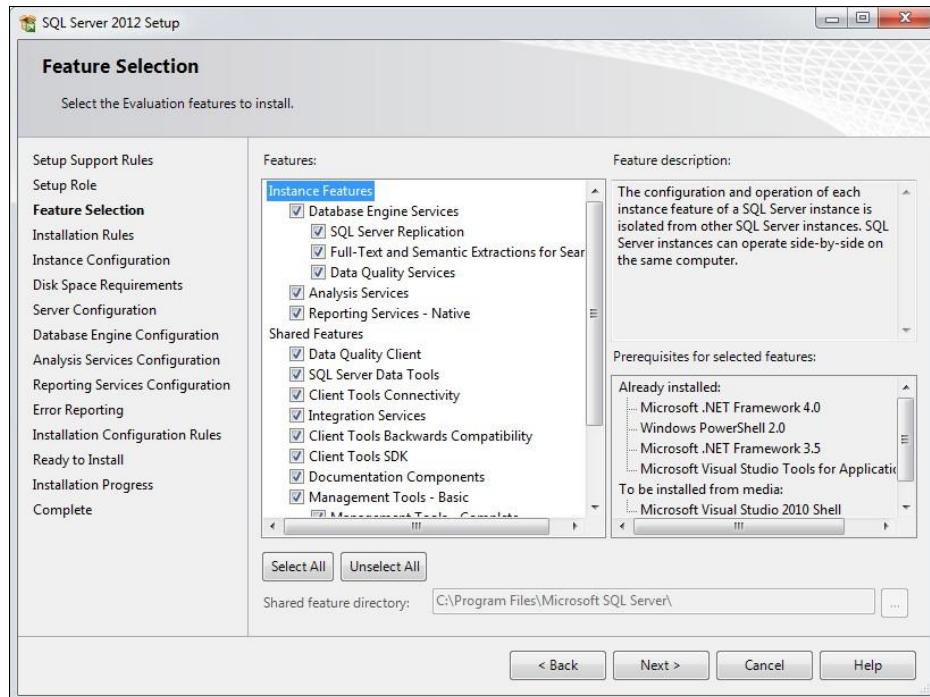


Рис. 1.11. Выбор устанавливаемых компонентов

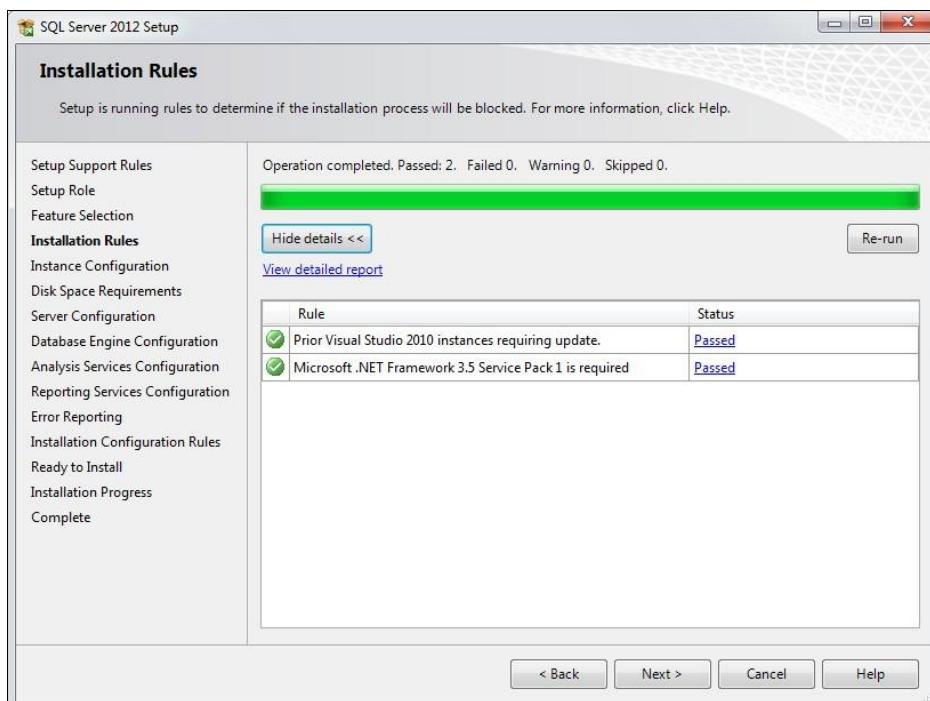


Рис. 1.12. Результат проверки состояния программных средств компьютера

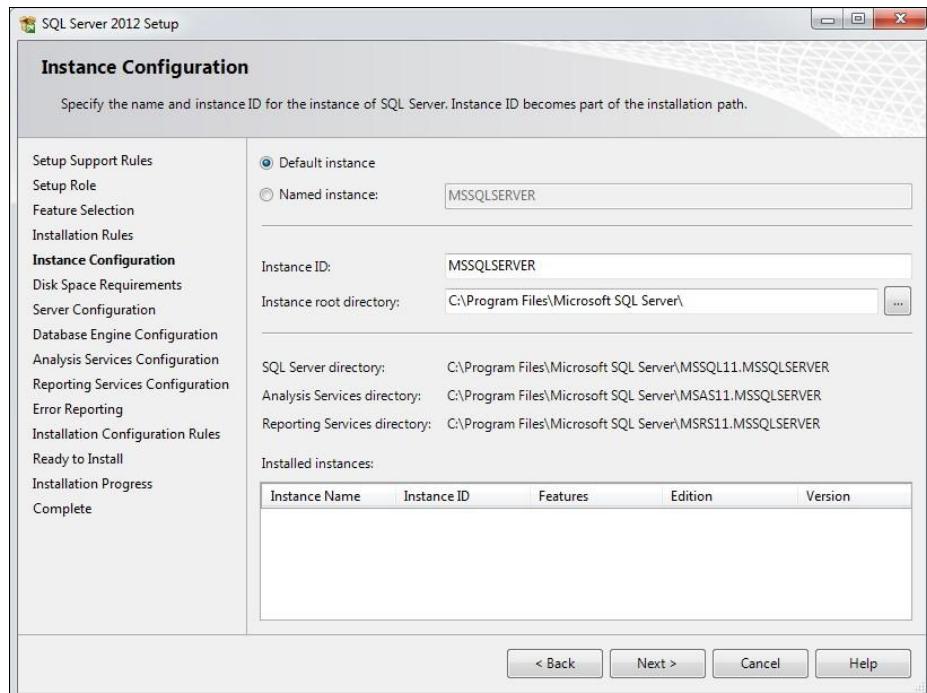


Рис. 1.13. Задание имени экземпляра сервера

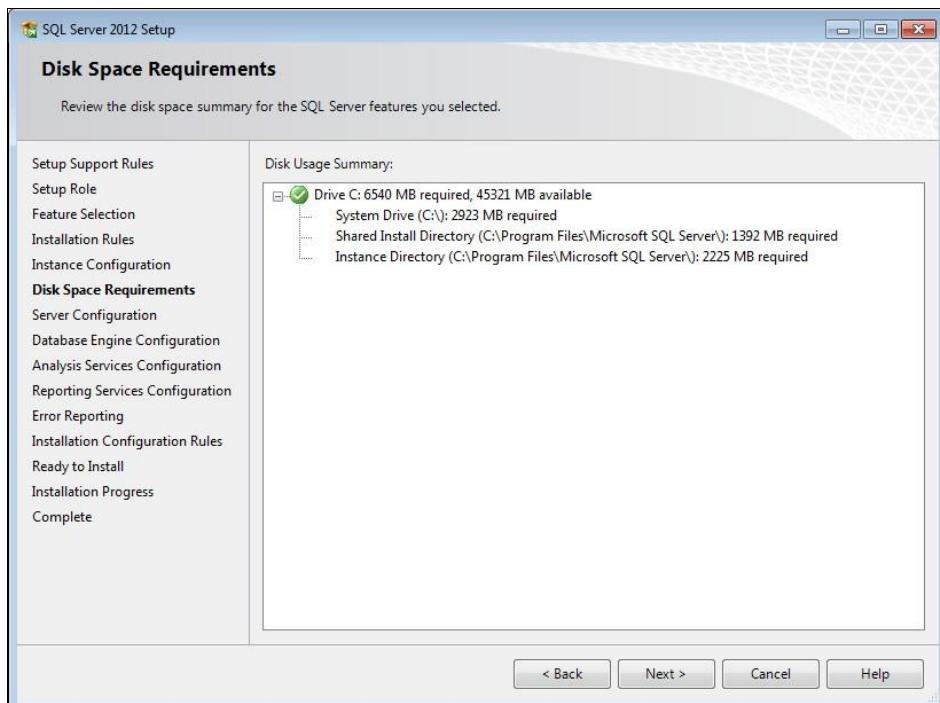


Рис. 1.14. Требования к дисковому пространству

Следующее окно **Server Configuration** (конфигурация сервера) (рис. 1.15) позволяет задать режим запуска на выполнение компонентов SQL Server (ручной, автоматический или запрет на запуск компонента) и пользователя, имеющего право на запуск этих компонентов.

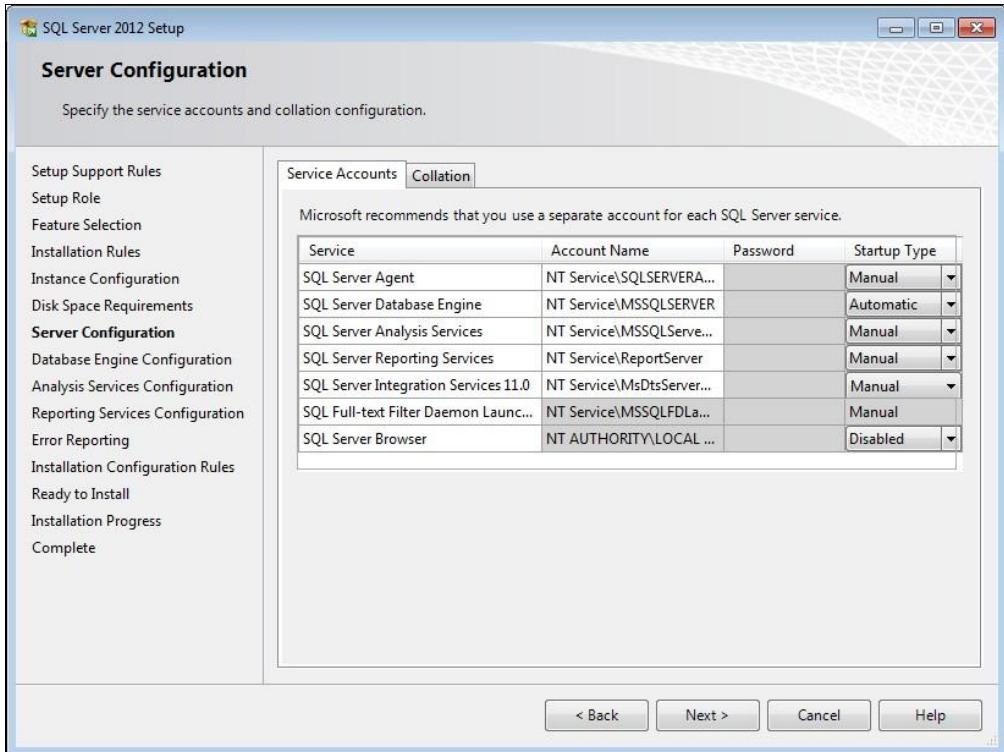


Рис. 1.15. Конфигурирование сервера. Вкладка **Service Accounts**

В этом окне задайте способ запуска на выполнение компонентов (параметр **Startup Type**). Если вы регулярно используете SQL Server, то имеет смысл для компонента движка базы данных (строка **SQL Server Database Engine**) выбрать режим автоматического запуска (т. е. значение **Automatic**). Таким образом SQL Server будет автоматически запускаться на выполнение при загрузке операционной системы. Чтобы понапрасну не занимать ресурсы вашей вычислительной системы, для остальных компонентов, кроме **SQL Server Browser**, лучше выбрать ручной режим запуска (**Manual**). В любой момент времени вы можете запускать на выполнение и останавливать каждый из компонентов.

Перейдите на вкладку **Collation** (параметры сортировки). Если вы что-то неверно установили на предыдущей вкладке, то при попытке этого перехода вы получите сообщение об ошибке. Исправьте указанную ошибку.

На вкладке **Collation** этого окна можно посмотреть и при желании изменить порядок сортировки по умолчанию для экземпляра сервера (рис. 1.16).

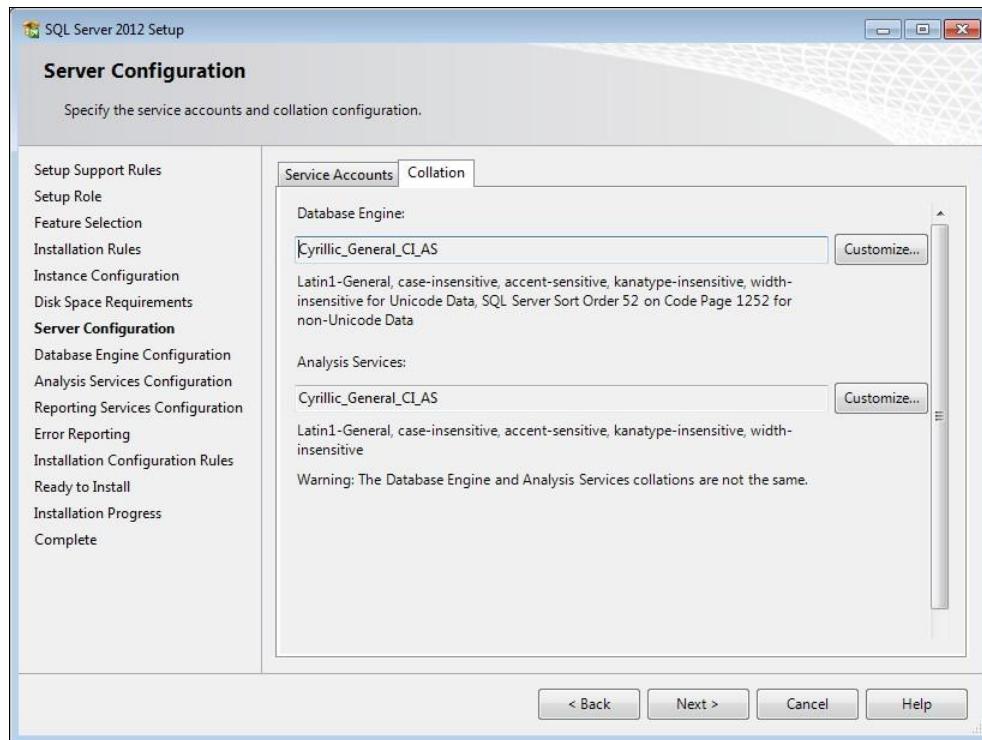


Рис. 1.16. Конфигурирование сервера. Вкладка **Collation**

По умолчанию на этой вкладке задается порядок сортировки Cyrillic_General_CI_AS. Он используется для хранения в строковых типах данных таблиц базы данных помимо латинских букв (и, разумеется, всех дополнительных символов) также и букв кириллицы.

Щелкните по кнопке **Next**. Следующим будет окно **Database Engine Configuration** конфигурирования компонента Database Engine (рис. 1.17).

На вкладке **Server Configuration** (конфигурация сервера) нужно установить режим аутентификации. Выберите переключатель **Mixed Mode** (смешанный режим: аутентификация и SQL Server, и Windows). Щелкните по кнопке **Add Current User** (добавить текущего пользователя), чтобы задать администратора SQL Server. В поля **Enter password** и **Confirm password** введите пароль и подтверждение пароля, например, обычный набор цифр 123456.

На вкладке **Data Directories** (каталоги размещения данных), показанной на рис. 1.18, можно просмотреть и при желании поменять каталоги для размещения объектов базы данных. Рекомендую не поддаваться такому искушению и не изменять каталоги. В моей практике были случаи, когда некоторые программные средства работали неверно при изменении путей к различным программам и данным.

Вкладка **FILESTREAM** (файловые потоки), показанная на рис. 1.19, позволяет задать возможность использования файловых потоков. Со временем мы с вами зайдем и файловыми потоками. Установите все флагки на этой вкладке.

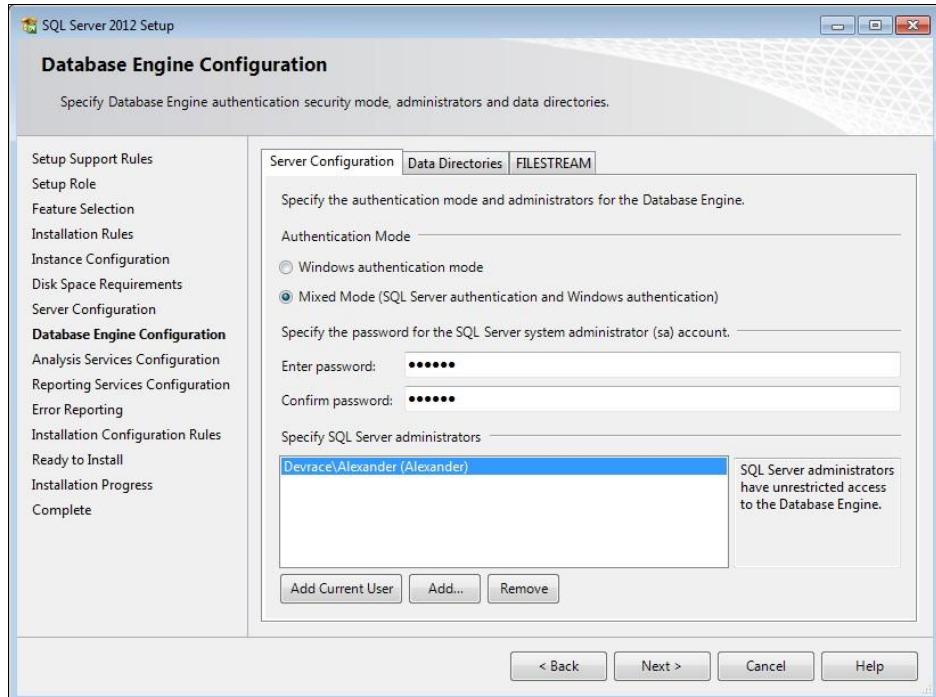


Рис. 1.17. Окно конфигурирования компонента Database Engine. Вкладка Server Configuration

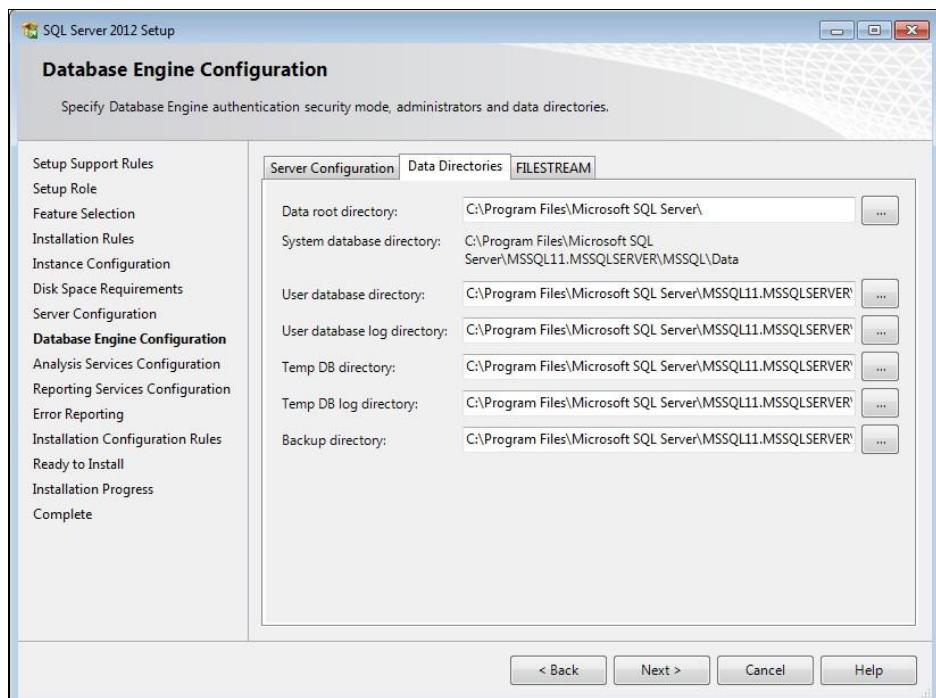


Рис. 1.18. Задание каталогов для размещения компонентов SQL Server

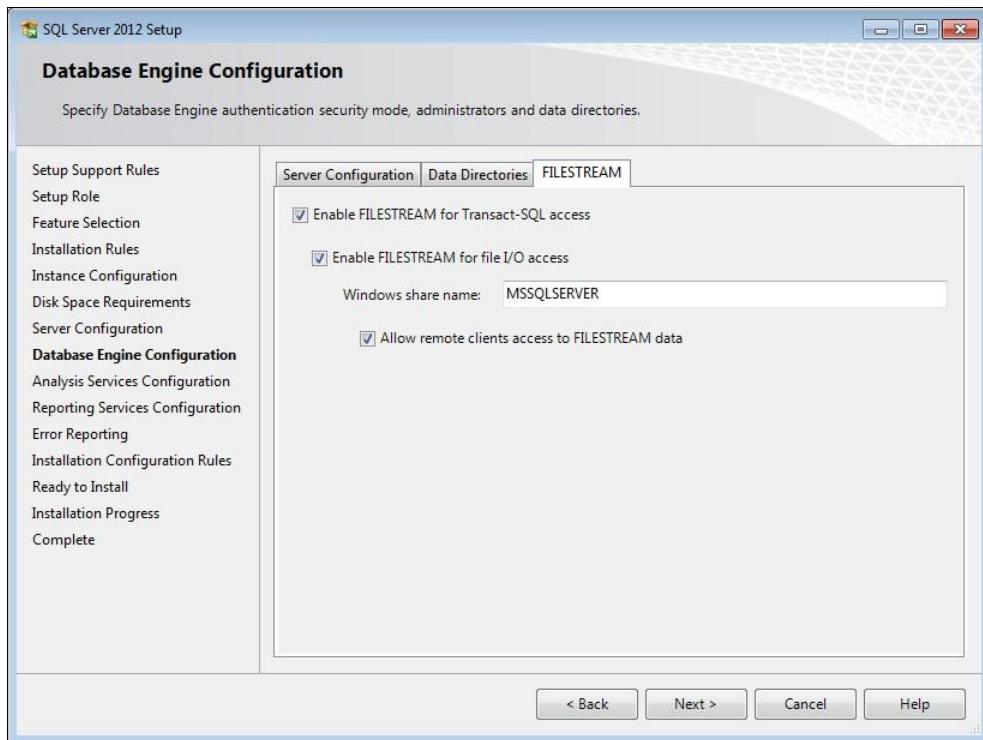


Рис. 1.19. Вкладка FILESTREAM

Щелкните по кнопке **Next**. Следующим будет окно конфигурирования компонента Analysis Services с двумя вкладками.

На вкладке **Server Configuration** (конфигурация сервера) (рис. 1.20) нужно добавить текущего пользователя в качестве администратора компонента Analysis Services. Для этого нужно просто щелкнуть по кнопке **Add Current User** (добавить текущего пользователя).

На вкладке **Data Directories** (каталоги данных) (рис. 1.21) можно просмотреть и изменить пути к файлам, используемым компонентом Analysis Services. Щелкните по кнопке **Next** (далее).

В следующем окне (рис. 1.22) можно выполнить конфигурирование компонента Reporting Services. Собственно конфигурирование заключается лишь в том, чтобы выбрать переключатель **Install and configure** (инсталлировать и конфигурировать) и щелкнуть по кнопке **Next** (далее).

В окне **Error Reporting** (отчет об ошибках) (рис. 1.23) можно задать отправку корпорации Microsoft сообщения об ошибках, созданных SQL Server или операционной системой Windows. Я бы такого делать не стал. Снимите соответствующий флажок, если он был отмечен, и щелкните по кнопке **Next** (далее).

В следующем окне **Installation Configuration Rules** (правила конфигурации установки) (рис. 1.24) выполняются очередные проверки и отображаются результаты этих проверок. Если никаких ошибок не обнаружено, то щелкните по кнопке **Next**.

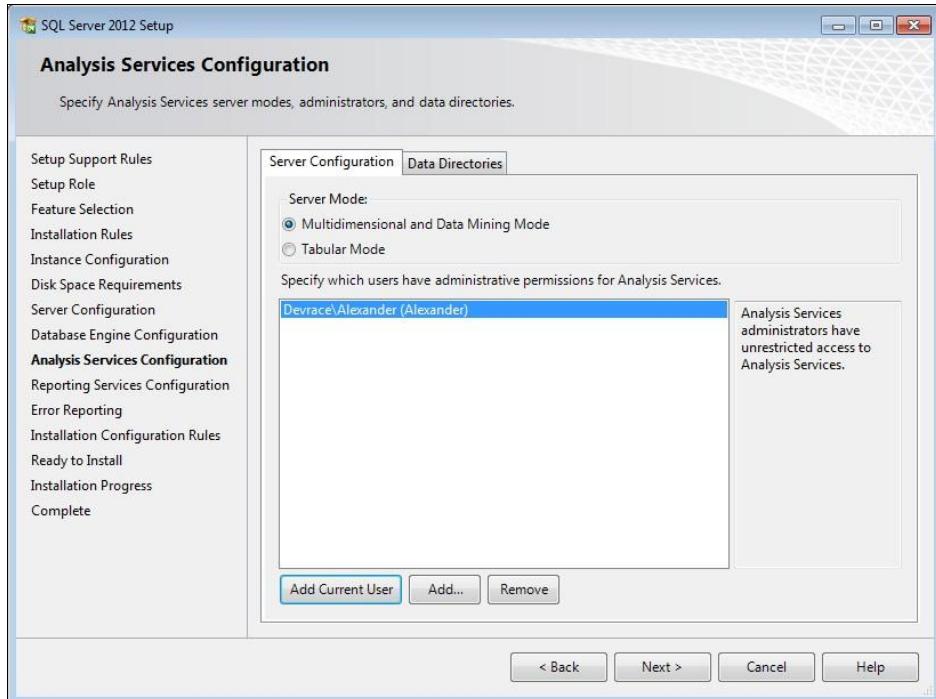


Рис. 1.20. Конфигурирование Analysis Services. Вкладка **Server Configuration**

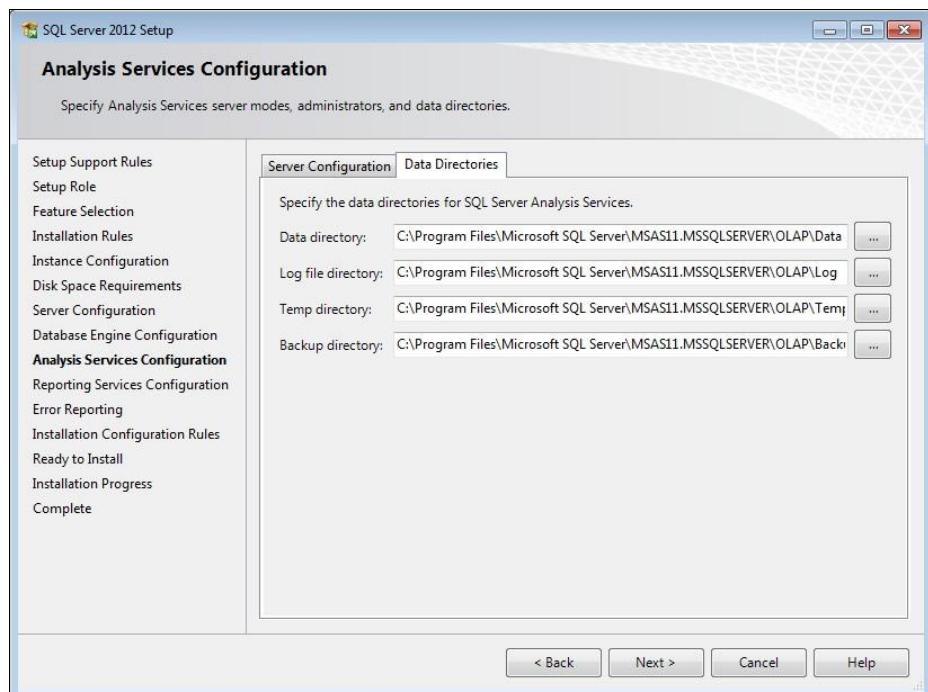


Рис. 1.21. Конфигурирование Analysis Services. Вкладка **Data Directories**

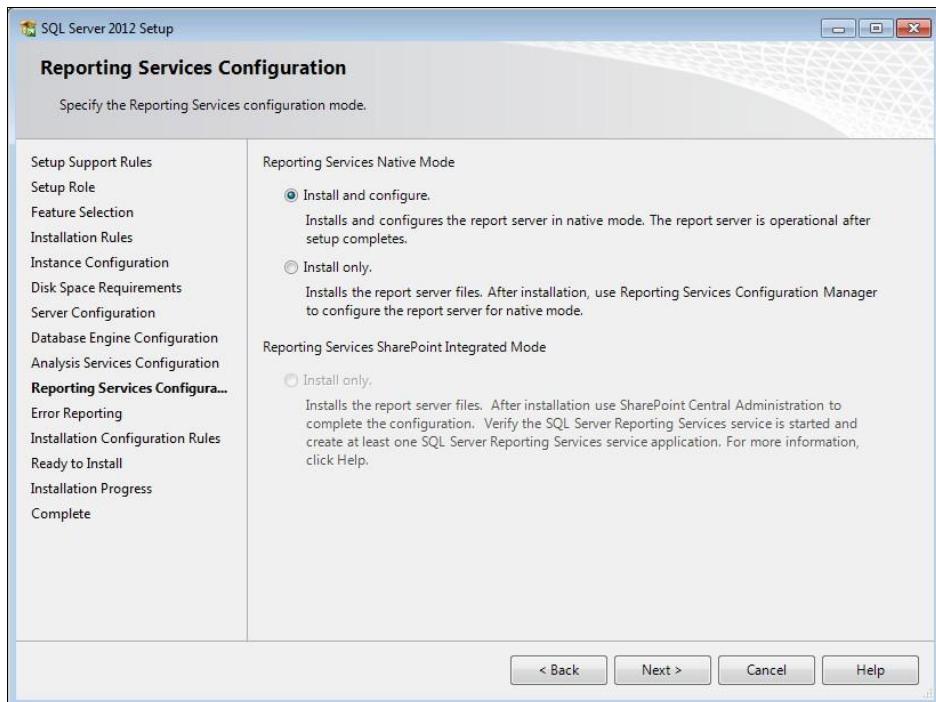


Рис. 1.22. Конфигурирование Reporting Services

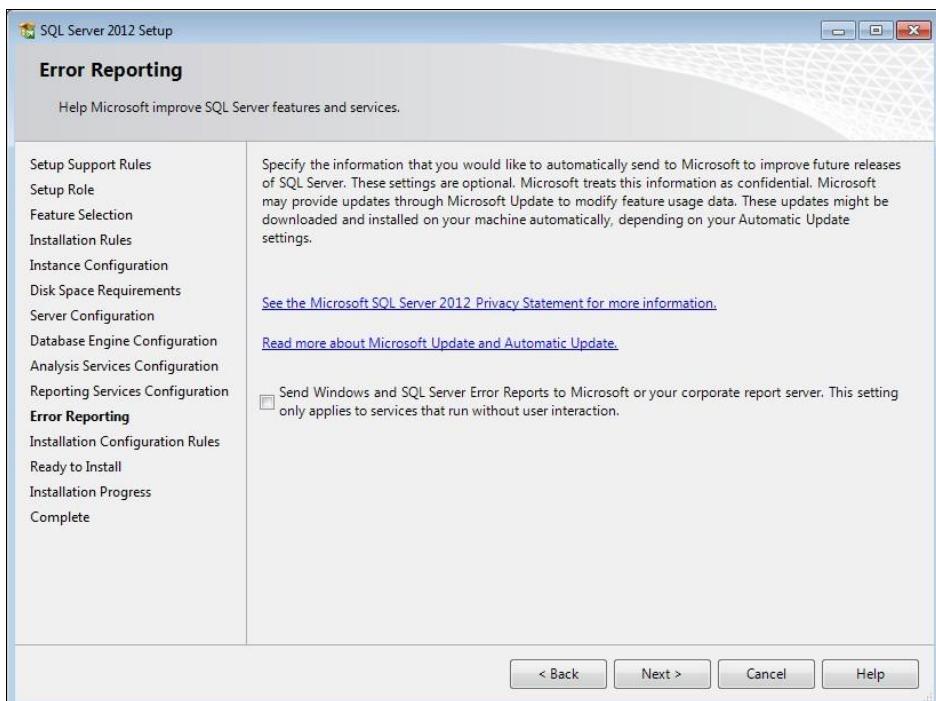


Рис. 1.23. Окно Error Reporting

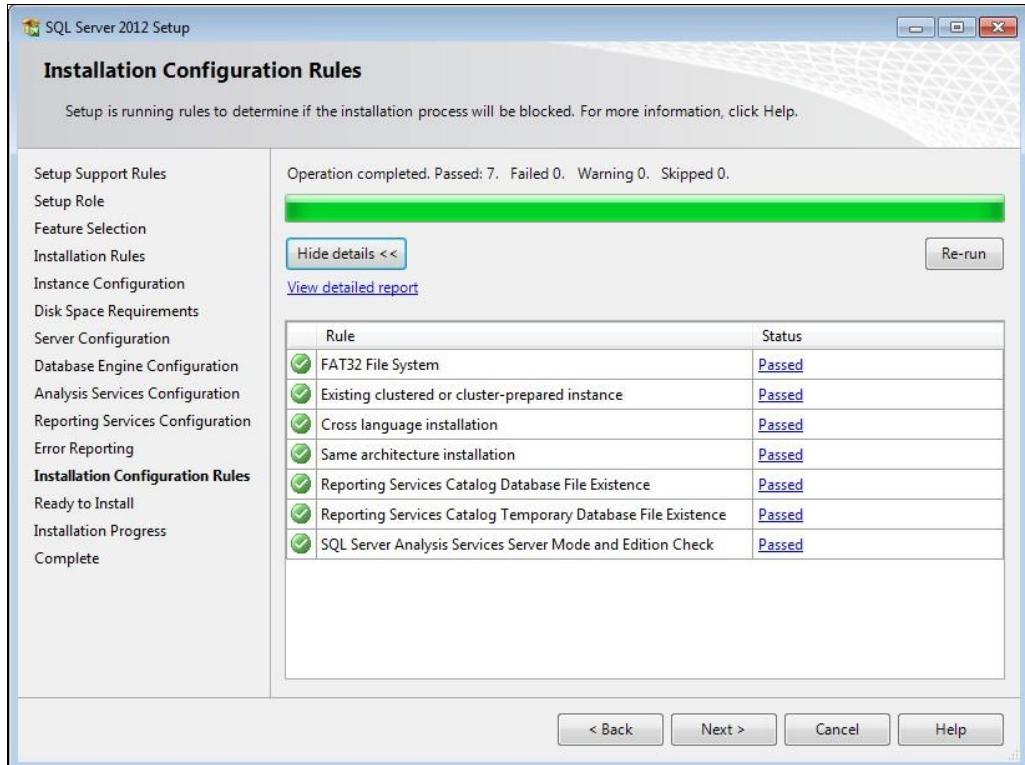


Рис. 1.24. Проверка конфигурации

В окне **Ready to Install** (все готово для установки) (рис. 1.25) даются итоговые сведения об устанавливаемых компонентах и некоторых их характеристиках. Для начала инсталляции щелкните по кнопке **Install** (установить).

Сам процесс инсталляции в зависимости от характеристик вашего компьютера и наличия уже установленных компонентов может занять достаточно много времени, около часа или более. Динамику выполнения инсталляции вы увидите в окне **Installation Progress** (ход выполнения установки) (рис. 1.26).

По завершении этого процесса появится последнее окно (рис. 1.27).

Для окончательного завершения процесса щелкните мышью по кнопке **Close** (закрыть). Появится начальное окно, показанное на рис. 1.4. Его нужно закрыть обычным образом, щелкнув по кнопке закрытия в правом верхнем углу.

Правила хорошего тона предполагают, что вы перезагрузите после этого компьютер.

Как вы понимаете, в новых релизах системы процесс инсталляции может несколько отличаться от того, что мы с вами сейчас рассмотрели.

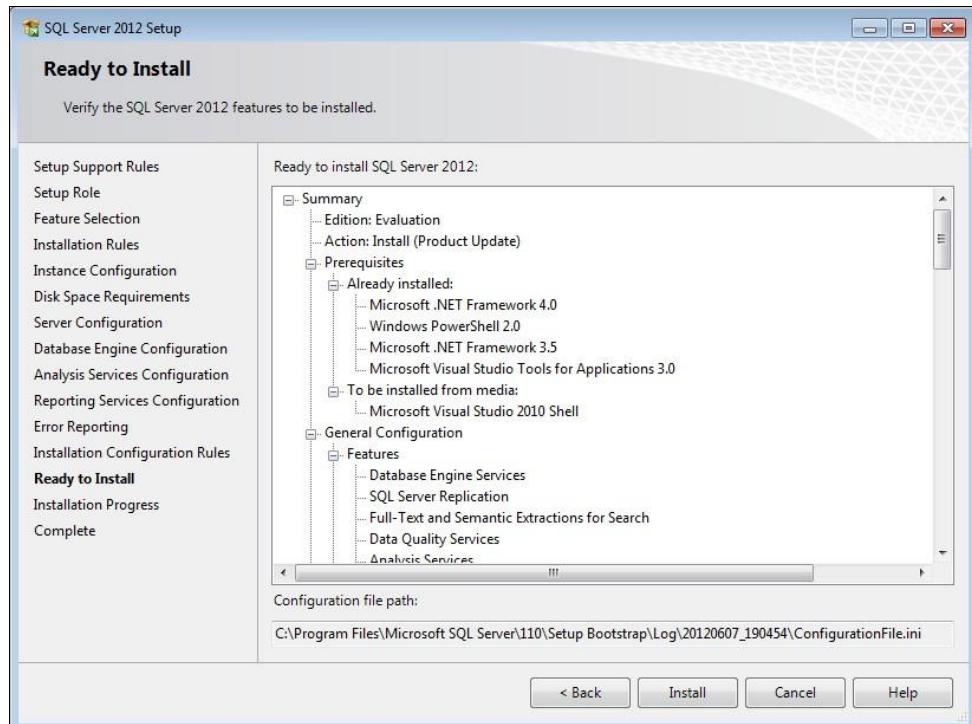


Рис. 1.25. Начало инсталляции. Окно Ready to Install

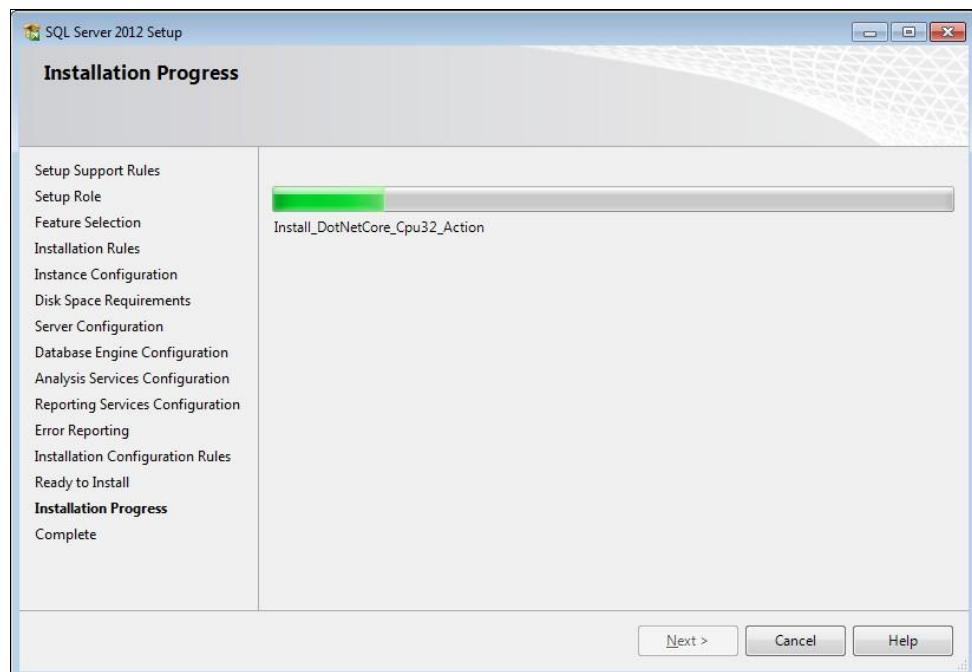


Рис. 1.26. Процесс инсталляции

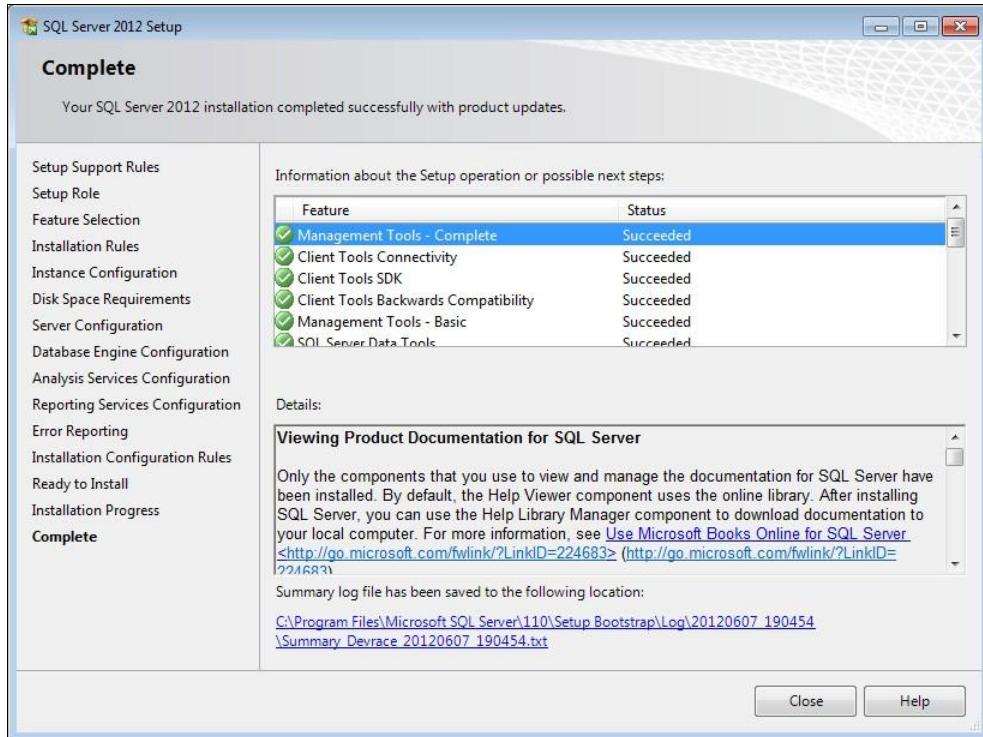


Рис. 1.27. Завершение инсталляции



ГЛАВА 2

Общие сведения о SQL Server 2012

- ◆ Реляционные базы данных
- ◆ Объекты базы данных SQL Server
- ◆ Нормализация таблиц. Нормальные формы
- ◆ Язык Transact-SQL. Основные синтаксические конструкции

В этой главе мы рассмотрим основы реляционных баз данных, их объекты. Главным объектом является, конечно же, таблица. Здесь мы рассмотрим нормальные формы таблиц и примеры приведения таблиц к нормальным формам. Язык, используемый для работы с базами данных в SQL Server, называется *Transact-SQL*. Будут рассмотрены средства описания синтаксиса этого языка. Помимо принятых в документации и в большей части программистской литературы нотаций Бэкуса — Наура для описания синтаксиса здесь будут даны и графические средства — R-графы, которые во многих случаях являются более наглядным способом представления синтаксических конструкций.

Если вы знакомы с основами реляционных баз данных, с объектами базы данных в SQL Server, нормальными формами и процессом нормализации таблиц, то можете смело пропустить первые разделы этой главы или бегло их просмотреть. Только настоятельно рекомендую ознакомиться с теми средствами, которые предлагаются здесь для описания синтаксиса. Графический способ задания синтаксических конструкций не так часто используется в технической литературе. А зря.

2.1. Реляционные базы данных

MS SQL Server 2012 является *реляционной системой управления базами данных*, сокращенно *РСУБД*. Вкратце можно сказать о реляционных системах то, что все данные в таких базах данных хранятся в таблицах. В основе реляционных систем лежит строгий безупречный математический аппарат.

Важной особенностью всех без исключения баз данных является то, что в базе данных помимо самих данных хранятся и описания этих данных — *метаданные*. Это

позволяет сильно уменьшить зависимость программ от данных на логическом уровне.

Главным объектом реляционных баз данных являются таблицы.

2.1.1. Таблицы

2.1.1.1. Основные свойства и характеристики таблиц

Таблица (table) содержит произвольное количество строк (row) или, что то же самое, записей (record). На самом деле максимальное количество строк в таблице ограничивается объемом внешней памяти, доступной для хранения данных базы данных. Разумеется, таблица может быть и пустой, т. е. не содержать ни одной строки.

Все строки одной таблицы имеют одинаковую структуру. Они состоят из столбцов (column). Столбцы иногда называют полями (field). Таблица должна содержать как минимум один столбец.

Данные из таблицы выбираются при помощи оператора `SELECT`. Как правило, выбираются не все строки таблицы, а только те, которые соответствуют условию, заданному в этом операторе. На содержательном уровне это те данные из таблицы, которые реально нужны пользователю для решения им конкретной задачи предметной области. При помощи оператора `SELECT` мы можем выбирать данные не только из одной таблицы, но и из нескольких таблиц, используя операцию соединения (join). Результат выборки называется *набором данных* (dataset).

Типы данных

Основной характеристикой столбца является его тип данных (datatype). Каждый тип данных в SQL имеет имя. Типы данных могут быть предварительно определены в системе (predefined), их иногда называют *системными,строенными* или *базовыми типами данных*. Это также могут быть данные, *определенные пользователем* (user-defined). В некоторых системах есть еще один термин для пользовательских типов данных — *домен* (domain).

Типы данных — числовые (целочисленные, дробные с фиксированной точкой и числа с плавающей точкой), строковые, логические, типы данных даты и времени. Существует тип данных, обычно называемый *двоичным большим объектом* (Binary Large OBject, BLOB), который позволяет хранить любые большие по объему данные — форматированные тексты, изображения, звук, видео. По мере развития программной отрасли в мире программного обеспечения появляются новые типы данных, например XML, или пространственные (spatial) типы данных. Все эти типы данных поддерживаются в системе MS SQL Server.

Тип данных в программировании определяет множество допустимых значений и множество допустимых операций для столбца и вообще для любого элемента данных в программном объекте. Например, для *целочисленных типов данных* множеством допустимых значений является множество целых чисел в определенном диапазоне. Диапазон представления этих чисел определяется количеством байтов, отводимых под целое число. Множеством допустимых операций для всех числовых

типов данных, как целочисленных, так и дробных, являются четыре арифметические операции — сложение, вычитание, умножение и деление.

Для строковых типов данных множеством допустимых значений являются произвольные строки. К ним применяется только одна операция: *конкатенация*, т. е. соединение нескольких строк в одну. Строковые типы данных определенного вида могут хранить, в том числе, и строки в формате Unicode, который дает возможность задать более 65 тысяч различных символов. Для работы со строковыми типами данных существует множество полезных функций, а именно: выделение подстроки, удаление начальных и конечных пробелов, поиск значения в строке и многие другие. Все эти функции мы рассмотрим в *главе 4*.

Для логического типа данных множеством допустимых значений в обычных языках программирования являются два значения: *TRUE* (истина) и *FALSE* (ложь). В реляционных базах данных используется еще одно значение *NULL* (неизвестное значение). Здесь применяется уже не обычная двухзначная, а трехзначная логика.

Для логического типа данных в реляционных базах данных применяются три логические операции: *отрицание* (операция *NOT*), *дизъюнкция* (логическое ИЛИ, *OR*) и *конъюнкция* (логическое И, *AND*).

ЗАМЕЧАНИЕ

Использование трехзначной логики в реляционных базах данных связано с присутствием среди значений столбцов и неизвестного, или неопределенного, значения *NULL*. Об этом значении *NULL* будет сказано чуть позже через несколько абзацев.

При создании таблиц в базе данных можно для каждого столбца подробно описывать все его характеристики — тип данных, значение по умолчанию, допустимые значения и некоторые другие. Характеристики также можно описать и в созданном пользовательском типе данных, а затем ссылаться на это описание при определении столбцов таблицы.

Порядок сортировки

Порядок сортировки (*collation*) используется при помещении данных в строковые столбцы таблицы и при сравнении значений строковых типов данных. Некоторые порядки сортировки помимо "обычных" символов (цифры, буквы латинского алфавита, разделители) содержат и буквы кириллицы, другие позволяют хранить символы практически любых алфавитов, включая разнообразные иероглифы. Перевод символов из одного порядка сортировки в другой называется *транслитерацией*.

Порядок сортировки также задает способ, правила, упорядочения строковых данных. Он определяет не только лексикографический порядок, т. е. упорядочение значений по алфавиту, но и некоторые другие характеристики упорядочения. Например, в нем задается расположение в отсортированном результате разделителей (точка, запятая, двоеточие и др.), порядок для прописных и строчных букв, чувствительность к регистру и ряд других. В стандарте SQL порядок сортировки еще называется *символьным репертуаром* (*character repertoire*).

Порядок сортировки может задаваться на уровне сервера, базы данных и отдельных строковых столбцов таблиц.

ЗАМЕЧАНИЕ

В некоторых системах управления базами данных используется два понятия — *набор символов* (character set) и *порядок сортировки* (collation, collation order). Набор символов определяет, какие символы хранятся в соответствующем элементе данных. Порядок сортировки применим к конкретному набору символов, он определяет, в каком порядке сортируются символы. Один набор символов обычно имеет несколько порядков сортировки.

Неизвестное значение `NULL`

Среди значений, которые может принимать столбец, в реляционных базах данных также используется и пустое или неизвестное значение `NULL`. Не стоит смешивать его с нулевым значением у числового столбца или строкой с нулевой длиной для строкового типа данных. Такое значение присваивается тем столбцам, реальные значения которых нам не известны или которые в принципе неприменимы для конкретного объекта. Примером может служить дата рождения, которая часто требуется при описании какого-либо человека. Иногда бывает так, что эта дата нам просто не известна. При этом большинство задач обработки данных может решаться и при отсутствии таких данных. В этом случае полю присваивается значение `NULL`. Другой пример — серия и номер паспорта человека. Если у него еще нет паспорта, то такое значение будет неопределенным. Еще случай, когда опять же при описании людей в таблице существует столбец отчества. У некоторых национальностей в принципе не бывает отчеств (например, у американцев). Здесь также элементу данных присваивается значение `NULL` (по правде говоря, в этом случае полю можно было бы просто присвоить и пустую строку нулевой длины).

При использовании в базах данных неизвестного значения возникают некоторые вопросы по их применению. Какой результат нужно присвоить операции сравнения, если одна из сравниваемых величин или обе имеют значение `NULL`? Разумным решением будет то, что результат нам также не известен, даже если оба сравниваемых столбца имеют значение `NULL` (не могут два неизвестных значения обязательно быть равными друг другу). В подобных сравнениях результатом не будет ни `TRUE` (истина) и ни `FALSE` (ложь), результатом будет значение `NULL`. Здесь и появляется необходимость в использовании трехзначной логики. Таблицы истинности для операций отрицания, дизъюнкции и конъюнкции в трехзначной логике будут рассмотрены в главе 4.

В языке SQL существует оператор `IS NULL` и функция `ISNULL()`, выполняющие проверку на неизвестное значение. Поскольку языки программирования используют (пока еще) двухзначную логику, то при работе с базами данных при сравнении значений столбцов до применения обычных операций сравнения регулярно используется функция `ISNULL()` или оператор `IS NULL`.

ЗАМЕЧАНИЕ

Вы можете увидеть в литературе критические замечания относительно использования в базах данных и вообще в программировании значения `NULL`. Я считаю, что не следу-

ет принимать близко к сердцу негодящие высказывания по этому поводу, даже если они исходят от известных специалистов в области программирования и баз данных. Использование неизвестного значения в реальных разработках показало, что существует обоснованная потребность в подобном значении.

Индексы

Объект базы данных *индекс* (index) используется для отдельных таблиц. Для каждой таблицы можно создавать один кластерный (см. далее) и до 999 обычных индексов. В таблице выбирается столбец или несколько столбцов, по которым формируется индекс. В результате в базе данных на внешнем носителе создается упорядоченная структура, которая будет содержать значения индексированных столбцов для каждой строки таблицы.

Индексы позволяют ускорить процесс выборки данных из таблицы и процесс упорядочивания выбранных данных. Индексы также могут быть использованы для обеспечения уникальности значений столбцов, входящих в состав индекса. Можно создавать так называемые *кластерные индексы*. Такие индексы в самых нижних узлах своей структуры содержат и строки таблицы. В таблице может быть только один кластерный индекс. Кластерные индексы позволяют увеличить скорость выборки отдельных строк таблицы из базы данных. Не имеющая кластерного индекса таблица называется *кучей* (heap).

Индексы создаются разработчиками базы данных для отдельных таблиц. В некоторых случаях система автоматически создает индексы для ключей таблицы.

Хорошо созданные индексы могут сильно повысить производительность системы. В то же время безобразно спроектированные индексы могут резко снизить производительность.

2.1.1.2. Ключи в таблицах

В таблицах могут присутствовать следующие виды ключей — *первичный ключ* (primary key), *уникальный ключ* (unique), *внешний ключ* (foreign key).

Первичный ключ

Таблица может иметь один, и только один первичный ключ (primary key). *Первичный ключ* — это столбец или группа столбцов, значение которых однозначно определяет конкретную строку таблицы.

Первичный ключ позволяет на основании значения столбцов, входящих в состав этого ключа, отыскать в базе данных ровно одну строку в указанной таблице или установить тот факт, что соответствующей строки в таблице не существует. Основным требованием к первичному ключу является его уникальность. То есть в таблице не должно быть двух различных строк, имеющих одинаковое значение первичного ключа. Ни один столбец, входящий в состав первичного ключа, не может иметь значения NULL (в описании таких столбцов должно, как правило, явно присутствовать предложение NOT NULL).

Первичные ключи часто присутствуют в реализации отношений между таблицами базы данных в связке "внешний ключ/первичный ключ".

Система управления базами данных автоматически создает индекс для первичного ключа таблицы. По умолчанию этот индекс является кластерным.

Стандарты SQL по какой-то причине не требуют обязательного присутствия первичного ключа в каждой таблице базы данных. Однако наличие такого ключа весьма и весьма желательно для каждой таблицы. Это показала практика использования реляционных баз данных. Забегая немного вперед, должен сказать, что в трактовке корпорации Microsoft первая нормальная форма таблиц требует обязательного наличия первичного ключа в каждой таблице. Это не соответствует общепринятой практике, но лично мне нравится.

Уникальный ключ

Каждая таблица может содержать произвольное количество *уникальных ключей* (*unique key*). В состав уникального ключа, как и в случае первичного ключа, может входить один или более столбцов таблицы. В отличие от первичного ключа столбцы уникального ключа могут иметь значение `NULL`. В таблице не может быть двух разных строк, имеющих одинаковое значение уникального ключа. Одним из назначений уникальных ключей является устранение дублирования значений, как и в случае уникальных индексов.

Система управления базами данных автоматически создает индекс для каждого уникального ключа таблицы. Индекс, создаваемый для уникального ключа, может быть кластерным, если для таблицы не существует другого кластерного индекса.

Уникальный ключ может присутствовать в связке таблиц вида "внешний ключ/уникальный ключ". Такая связка сейчас будет рассмотрена.

Внешний ключ

Внешний ключ (*foreign key*) — это столбец или группа столбцов таблицы, которые ссылаются на первичный или уникальный ключ другой или этой же самой таблицы.

Требование к значению столбцов, входящих в состав внешнего ключа, следующее: либо все столбцы внешнего ключа должны иметь значение `NULL`, либо таблица (главная или, иными словами, *родительская*), на первичный или уникальный ключ которой ссылается внешний ключ *подчиненной* (или *дочерней*) таблицы, должна иметь строку со значением первичного или уникального ключа, которое в точности равно значению внешнего ключа дочерней таблицы.

ПРИМЕЧАНИЕ

Здесь требуется маленькое уточнение. Если внешний ключ ссылается на *уникальный ключ* родительской таблицы, то отдельные столбцы (не обязательно все) во внешнем ключе могут иметь значение `NULL`. В родительской таблице в этом случае также должна присутствовать строка, имеющая такую же комбинацию значений в столбцах, входящих в состав уникального ключа.

Сейчас эта фраза, возможно, звучит сухо и непонятно, но дальше мы проиллюстрируем все на многочисленных примерах.

Выражение "ссылается на первичный или уникальный ключ" означает всего лишь то, что в таблице, на которую "ссылается" внешний ключ, должна присутствовать

строка, имеющая первичный или уникальный ключ, значение которого в точности равно значению этого внешнего ключа.

Отношения между таблицами в базе данных

Важнейшими отношениями (связями) в реляционных базах данных являются отношение "внешний ключ/первичный ключ" и отношение "внешний ключ/уникальный ключ". Эти связи (отношения, relationship) между таблицами называются *декларативной целостностью данных* (declarative data integrity). Декларативная целостность обеспечивает и непротиворечивость данных в базе данных — в случае правильного проектирования базы данных.

Декларативная целостность базы данных обеспечивается системой управления базами данных. Система отменяет все попытки добавления и изменения данных, которые нарушают заданную средствами операторов DDL-целостность (т. е. непротиворечивость) данных — в базу данных не может быть помещена строка таблицы, чей внешний ключ не соответствует ни одному значению первичного или уникального ключа родительской таблицы, на который ссылается этот внешний ключ (о DDL читайте далее в этой главе). Нельзя также внести изменение в существующую строку таблицы, если изменяемое значение нарушает целостность данных.

Ограничения таблицы

Первичные, уникальные и внешние ключи таблиц называются *ограничениями* (constraint) таблицы. Кроме них существуют:

- ◆ *ограничение на значения, помещаемые в столбцы таблицы*. Это ограничение CHECK, благодаря которому в таблицу не может быть помещена новая строка или выполнено изменение данных уже существующей в таблице строки, если будет нарушено указанное ограничение. При задании ограничения можно указать довольно сложные условия, которым должно удовлетворять значение одного столбца или значения группы столбцов таблицы;
- ◆ *значение по умолчанию*. Это ограничение DEFAULT. Если при добавлении в таблицу новой строки не было задано значение какого-то столбца, то ему будет присвоено значение по умолчанию. Если при описании столбца не было явно указано значение по умолчанию (ограничение DEFAULT), то этим значением является NULL. Значение по умолчанию используется только при добавлении новой строки в таблицу, но не при изменении значений данных существующей строки;
- ◆ *допустимость для столбца значения NULL*. Предложение NOT NULL в описании столбца запрещает помещать в этот столбец значение NULL.

2.1.2. Представления

Представление (view) — это объект базы данных, при обращении к которому происходит выборка данных из таблицы или из нескольких таблиц базы данных при помощи оператора SELECT или при обращении к хранимой процедуре (см. ниже). Представление позволяет скрыть от пользователя сложный процесс выборки дан-

ных. Кроме того, представление позволяет повысить безопасность данных, предоставляя пользователю только те данные, к которым у него существуют полномочия, за счет выдачи разрешения на представление, а не на базовую таблицу (таблицы). Результатом обращения к представлению, как и в случае обычной выборки данных из таблицы при использовании оператора `SELECT`, является набор данных.

Представления бывают *изменяемые* и *неизменяемые*. Изменяемое представление позволяет вносить изменения в данные, полученные из представления, откуда они автоматически будут распространены в базовые таблицы представления, т. е. в таблицы, к которым обращается это представление. Неизменяемые представления такой возможности не предоставляют.

2.1.3. Хранимые процедуры и триггеры

Язык SQL содержит подмножество языковых средств, называемое языком *хранимых процедур и триггеров* PSQL. В этом подмножестве можно описывать, каким именно образом выбирается очередная запись из базы данных, что нужно сделать с отдельными столбцами этой записи. В языке хранимых процедур и триггеров существует, как и в обычных языках программирования, возможность описания внутренних переменных, оператор присваивания, операторы ветвления, операторы циклов и другие императивные средства. Язык допускает *рекурсию*, т. е. тот случай, когда программа вызывает саму себя.

Этот язык используется при создании хранимых процедур, функций и триггеров. Элементы языка (объявление локальных переменных, операторы ветвления и циклов) также могут быть использованы и в обычных скриптах при работе с базой данных. Такая возможность существует не во всех системах управления базами данных.

Хранимые процедуры (*stored procedure*) являются программами, хранящимися в базе данных и выполняющими различные действия, обычно с данными из базы данных, хотя процедуры могут и не осуществлять никаких обращений к базе. К хранимым процедурам могут обращаться любые программы, работающие с базой данных, к ним также могут обращаться и другие хранимые процедуры и триггеры. Допустима и рекурсия, когда хранимая процедура обращается к самой себе. Хранимые процедуры выполняются на стороне сервера, а не на стороне клиента. Во многих случаях это может резко снизить сетевой трафик при решении различных задач работы с базой данных и повысить производительность системы.

Функции, определенные пользователем (*user defined functions, UDF*). Это программные компоненты, к которым можно обращаться из триггеров, хранимых процедур, из других программных компонентов. Функции выполняют конкретные действия и возвращают ровно одно значение.

Триггеры (*trigger*), так же как и хранимые процедуры, являются программами, выполняющимися на стороне сервера. Однако напрямую обращение к триггерам невозможно. Они автоматически вызываются при наступлении некоторого события базы данных — например, при добавлении, изменении или удалении строк кон-

крайней таблицы. Триггеры могут вызываться при соединении с базой данных, а также в некоторых других случаях.

События базы данных (event). Хранимые процедуры и триггеры могут выдавать события — сообщения о появлении некой ситуации базы данных; такие сообщения могут перехватываться и обрабатываться клиентскими программами. Событиями могут быть ошибки в базе данных, которые выявляются не декларативным, а императивным способом — т. е. не при описании ограничений, таких как связка "внешний ключ/первичный (уникальный) ключ", а при выполнении более сложных проверок на соответствие вводимых данных требованиям предметной области. Часто события создаются при простых действиях с базой данных: при добавлении, изменении или удалении данных из конкретной таблицы. Они дают возможность проинформировать других клиентов о выполненных действиях. Такие события бывают полезными при синхронизации работы нескольких клиентов с одними и теми же данными в базе данных.

2.1.4. Пользователи, привилегии и роли базы данных

Сведения о пользователях, имеющих доступ к базам данных экземпляра сервера, хранятся в самой системе. Местом хранения является внутренний каталог сервера. У пользователя, описанного в системе, есть имя и пароль.

В SQL Server для авторизации пользователей рекомендуется использование средств авторизации операционной системы Windows.

Привилегии (полномочия) к объектам баз данных назначаются пользователям администратором базы данных. Привилегиями могут быть права на выполнение выборки, удаления, добавления и изменения данных конкретной таблицы базы данных, права на выполнение отдельных хранимых процедур, представлений.

Полномочия отдельному пользователю или группе пользователей могут назначаться прямым путем, а могут предоставляться при помощи механизма ролей (role). Роль — это объект базы данных, которому назначаются некоторые полномочия к отдельным объектам базы данных. Затем роль может назначаться различным пользователям. В момент соединения с базой данных при указании роли пользователь получает все полномочия, предоставленные данной роли.

2.1.5. Задание первичных ключей таблиц

Для каждой таблицы желательно использовать первичный ключ. Таблица может иметь только один первичный ключ. Важно правильно выбрать столбец или группу столбцов таблицы, которые войдут в состав первичного ключа. Основное требование к первичному ключу — его уникальность. В таблице не может быть двух разных строк, имеющих одинаковые значения первичного ключа. Второе реальное требование к первичному ключу — его относительно малый размер. Часто первичные ключи принимают участие в связке "внешний ключ/первичный ключ". Для

реализации этого отношения подчиненные, дочерние, таблицы должны включать в свой состав в качестве внешнего ключа столбцы, входящие в состав первичного ключа главной, родительской, таблицы. Кроме того, для первичного ключа система строит индексы. Все это в случае большого по размерам ключа увеличивает объем требуемой внешней памяти и может сильно ухудшить временные характеристики системы.

В процессе проектирования системы обработки данных выбор столбцов, входящих в состав первичного ключа, не всегда является простой задачей. Для осуществления такого выбора в таблице рассматриваются различные столбцы или группы столбцов в качестве *кандидатов* в первичные ключи.

Например, для таблицы, описывающей людей (персонал организации, студентов в учебном заведении), нужно выбрать столбцы, которые войдут в состав первичного ключа. Понятно, что использовать в этом качестве фамилию нельзя: слишком много существует однофамильцев. К фамилии можно добавить имя и отчество. Эти столбцы уже можно рассматривать в качестве кандидатов в первичный ключ. Но это тоже не гарантирует уникальности. Кроме того, размер первичного ключа получается слишком большим (не менее 50 символов), что отрицательно скажется на объеме используемой внешней памяти и на производительности системы.

Для сотрудников организации с этой целью можно использовать табельный номер, если этот номер является уникальным в рамках всей организации. Если табельный номер уникален только в пределах структурного подразделения, то можно сделать первичный ключ, состоящий из двух столбцов — кода структурного подразделения и табельного номера сотрудника внутри этого подразделения. Для студентов учебного заведения можно в качестве первичного ключа выбрать, например, номер студенческого билета.

В общем случае, когда в базе данных нужно хранить различные сведения по людям, не привязываясь ни к каким организациям, учебным структурам, то лучшим решением будет использование *искусственного первичного ключа*. Иногда в литературе можно встретить термин "суррогатный" (surrogate) первичный ключ; такой термин не очень нравится русскоязычным программистам по причине наличия некоторого негативного оттенка в этом слове.

В состав столбцов таблицы в этом случае добавляется целочисленный столбец, который и будет искусственным первичным ключом. В SQL Server такой столбец должен быть описан с атрибутом `IDENTITY`. Столбцы, которым системой автоматически присваивается уникальное значение, в литературе называются *автоинкрементными* (auto increment).

В SQL Server есть еще один способ создания и использования искусственного первичного ключа. Это применение последовательностей (sequence) для получения уникального значения. Их мы тоже рассмотрим в этой книге.

Что касается всевозможных вторичных, вспомогательных, связующих, таблиц, которые вскоре будут нами рассматриваться, то для них обычно используются именно искусственные первичные ключи.

2.1.6. Транзакции

Транзакция является "механизмом" базы данных. Это некоторая законченная, иногда довольно сложная единица работы с данными и/или метаданными базы данных. Все операторы работы с базой данных (как с данными, так и с метаданными) выполняются в рамках — или еще говорится, *в контексте* — какой-либо транзакции. Исключением является оператор `SELECT`, который может выполняться и вне контекста транзакции. В контексте транзакции выполняется, как правило, группа операторов, переводящих базу данных из одного непротиворечивого состояния в другое непротиворечивое состояние.

Все действия операторов одной транзакции могут быть либо подтверждены (оператор `COMMIT`), и тогда выполненные ими изменения будут зафиксированы в базе данных, либо отменены (оператор `ROLLBACK`). После подтверждения транзакции все изменения, выполненные операторами в ее контексте, станут видны другим параллельным процессам (иногда и неподтвержденные изменения бывают видны другим процессам, но об этом потом). Отмененные действия не сохраняются в базе данных.

Транзакциям могут задаваться некоторые характеристики, которые определяют поведение транзакций по отношению к другим параллельным процессам, а также допустимые одновременные действия других процессов.

Транзакции являются важным средством обеспечения одновременной работы с базой данных большого количества клиентских процессов, осуществляющих оперативную обработку данных. Понятно, что если два пользователя будут одновременно менять одну и ту же запись, ничего хорошего не получится. Второй клиент просто отменит изменения, внесенные первым, причем первый ничего про это не узнает. Он будет пребывать в полной уверенности, что сделанные им изменения остались в базе данных. Не узнает о выполненных изменениях также и второй, хотя если бы он знал, что запись уже поменялась, возможно, он захотел бы внести туда совсем другие изменения. SQL Server использует два механизма для разграничения многопользовательской работы. Исторически более ранним является *механизм блокировок*. Когда первый пользователь меняет запись, она блокируется для изменений всеми остальными пользователями. Тогда второй пользователь не сможет внести в нее изменения одновременно с первым. Когда SQL Server "отпустит" запись, второй пользователь сможет прочитать уже ее новое значение и, если захочет ее изменить, сделает это, по крайней мере, осознанно. Существуют различные уровни строгости блокировок, о которых мы поговорим, когда будем рассматривать уровни изоляции транзакций друг от друга. Чтобы уменьшить вероятность появления блокировок при одновременной попытке разных клиентов изменить одни и те же данные, стараются такие транзакции сделать как можно короче.

2.1.7. 12 правил Кодда

Очень часто при описании реляционных систем управления базами данных приводят и известные 12 правил Кодда. Эти правила нужны в первую очередь разработчикам самих СУБД, но нам с вами, специалистам по использованию уже созданных

СУБД, не следует тратить время на изучение этих правил. Тем не менее, я все же поместил эти правила в *приложение 1*, вдруг они все-таки кому-нибудь и понадобятся.

2.2. Реализация отношений в реляционной модели

В базах данных существует три вида отношений: "один к одному", "один ко многим" и "многие ко многим". Отношения в реляционных базах данных чаще всего реализуются связкой "внешний ключ/первичный ключ", реже — связкой "внешний ключ/уникальный ключ". Отношение между двумя таблицами вида "многие ко многим" реализуется добавлением третьей связующей таблицы и двумя связками "внешний ключ/первичный ключ".

Рассмотрим по порядку эти три отношения. Во всех примерах мы будем использовать графическое описание данных. Прямоугольники в этих графических описаниях будут изображать таблицы, линии со стрелками или без стрелок будут представлять отношения между таблицами.

2.2.1. Отношение "один к одному"

Если между двумя таблицами базы данных появляется отношение "один к одному", то лучше объединить эти таблицы в одну. Основной причиной использования этого отношения является экономия памяти и увеличение скорости выполнения запросов. Такое отношение используется в том случае, если связь между двумя таблицами не является обязательной.

Не так часто в реальной жизни встречаются случаи, когда требуется использовать отношение "один к одному".

Рассмотрим связь между человеком и адресом его фактического проживания. Это связь "один к одному". В данном случае используются две таблицы. Одна содержит какие-то сведения о человеке, другая — сведения о его адресе. Такая связь между двумя таблицами представляется на диаграмме линией, не содержащей стрелок (рис. 2.1).



Рис. 2.1. Пример отношения "один к одному"

Если данные об адресе поместить в первую таблицу, то при отсутствии соответствующих сведений это привело бы к некоторому количеству пустых полей в таблице и перерасходу внешней памяти. Надо сказать, что на самом деле на это не потребуется большого количества памяти, поскольку физическое хранение в SQL Server

хорошо продумано, и пустые поля используют минимальное количество внешней памяти.

Реализация такого отношения осуществляется очень просто. Первая таблица (Человек) имеет первичный ключ (скорее всего, искусственный). Во второй таблице (Адрес) нужно сделать такой же искусственный первичный ключ и указать, что он к тому же является и внешним ключом, ссылающимся на первичный ключ первой таблицы.

Можно привести еще некоторое количество примеров отношения "один к одному". Это, в частности, отношение между сотрудником организации (или студентом в учебном заведении) и его личным делом в отделе кадров.

2.2.2. Отношение "один ко многим"

Отношение "один ко многим" между двумя таблицами реализуется связкой "внешний ключ/первичный ключ". Реже применяется связка "внешний ключ/уникальный ключ". Это отношение иногда в литературе называют отношением "многие к одному".

Такое отношение можно назвать *универсальным* — с его помощью можно представить практически любые отношения в базе данных, начиная от простой иерархии до реализации отношения "многие ко многим".

Рассмотрим пример. Пусть есть таблица, содержащая список стран. Вторая таблица содержит список регионов каждой страны (республики, области, края в Российской Федерации; штаты в США, графства в Великобритании).

Первичным ключом таблицы стран будет некоторый код страны (существует международный стандарт для кодов всех стран; эти коды присутствуют и в нашей демонстрационной базе данных). Во вторую таблицу, таблицу регионов, помимо остальных столбцов нужно добавить поле внешнего ключа (код страны), которое будет ссылаться на первичный ключ первой таблицы, таблицы стран. Первичным ключом второй таблицы нужно сделать составной ключ — код страны и код региона.

Отношение "один ко многим" между этими двумя таблицами графически представляется в виде диаграммы, как показано на рис. 2.2.



Рис. 2.2. Пример отношения "один ко многим"

2.2.3. Отношение "многие ко многим"

Хорошим примером такого отношения является отношение между таблицей авторов и таблицей книг. Одна книга может быть написана несколькими авторами, один автор может написать несколько книг (рис. 2.3).

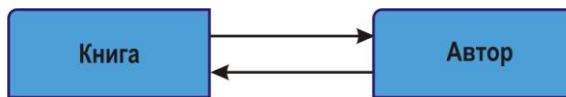


Рис. 2.3. Пример отношения "многие ко многим"

В таблице авторов "код автора" — искусственный (а может быть и обычный, созданный по принятым в издательстве правилам) первичный ключ. Для таблицы книг можно использовать первичный ключ "код книги".

Реализация отношения "многие ко многим" осуществляется добавлением в базу данных третьей, связующей, таблицы и установлением двух необходимых связей "один ко многим".

В данном примере добавляется связующая таблица между таблицей книг и таблицей авторов. Устанавливаются отношения "один ко многим" между книгой и связующей таблицей и "один ко многим" между автором и связующей таблицей (рис. 2.4).

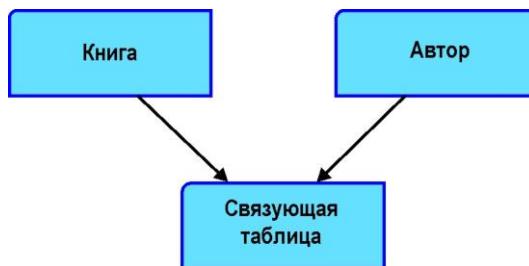


Рис. 2.4. Связующая таблица для реализации отношения "многие ко многим"

Связующая таблица содержит только два столбца — "код книги" и "код автора", связывая книги и авторов.

Столбец "код книги" в связующей таблице является внешним ключом, который ссылается на первичный ключ (код книги) в таблице книг. Столбец "код автора" — внешний ключ, ссылающийся на первичный ключ таблицы авторов.

Первичный ключ связующей таблицы состоит из двух столбцов: "код книги" и "код автора". В данном случае при описании книг и их авторов комбинация значений этих столбцов является уникальной и может быть использована в качестве первичного ключа, однако часто существуют и иные ситуации, где для связующей таблицы приходится вводить свой искусственный первичный ключ.

Надо сказать, что в библиографических информационно-поисковых системах такие отношения ("многие ко многим"), реализуемые при помощи добавления связующих

таблиц, встречаются постоянно. Например, отношение "книга-переводчик", "книга-научный редактор", "книга-ключевое слово". А вот отношение "книга-издательство" является отношением "один к одному".

2.3. Нормализация таблиц

2.3.1. Цель нормализации таблиц

Первая задача при проектировании базы данных — составление списка таблиц и разработка структуры каждой таблицы. В реляционных базах данных есть понятие *нормализации таблиц*. Существует набор стандартов проектирования данных, называемых *нормальными формами*. Нормальные формы определяют правила, которым должны соответствовать структуры таблиц. Общепризнанными являются шесть нормальных форм, хотя в литературе по базам данных можно найти и гораздо большее количество форм нормализации. На практике обычно используется третья нормальная форма.

Нормальные формы используются в таком порядке: первая, вторая, третья, форма Бойса — Кодда, четвертая и пятая. Плюс множество форм, которые мы рассматривать не станем. Каждая последующая форма удовлетворяет требованиям предыдущей. Если данные таблицы удовлетворяют третьему правилу нормализации, то они будут находиться в третьей нормальной форме (а также в первой и во второй формах).

Выполнение правил нормализации для таблицы обычно приводит к разделению таблицы на две или более таблиц с меньшим количеством столбцов. Эти таблицы для наглядности отображения данных за счет связки "внешний ключ/первичный ключ" или "внешний ключ/уникальный ключ" снова могут быть соединены в процессе выборки данных в операторе SELECT при помощи операции соединения (JOIN).

Одним из основных результатов разделения таблиц в соответствии с правилами нормализации является уменьшение избыточности данных. Подробные сведения о каждом объекте (или "сущности", entity) предметной области содержатся ровно один раз в конкретной таблице. В таблицах, где используются эти сущности, осуществляется лишь ссылка на нужную строку в соответствующей таблице.

Опыт проектирования баз данных показывает, что правильное выполнение нормализации таблиц реально создает удобную в эксплуатации базу данных и позволяет осуществлять к такой базе данных запросы, которые вовсе не были предусмотрены при первоначальном создании системы обработки данных. Более того, дальнейшее развитие созданной системы обработки данных проще осуществляется, если таблицы в базе данных были созданы по правилам нормализации.

2.3.2. Первая нормальная форма

Первая нормальная форма (иногда в литературе используется сокращение 1НФ или 1NF — first normal form) требует, чтобы значение любого столбца было единственным, атомарным. Иными словами, в таблице не должно быть повторяющихся

групп. В литературе корпорации Microsoft к первой нормальной форме предъявляется также и требование, чтобы каждая таблица имела первичный ключ.

В стандарте SQL2008 атомарность определяется как значение, которое не может быть разделено на более мелкие части, однако с таким определением можно поспорить. Фактически столбец, например, с типом данных дата (`DATE`) может быть разделен на части — день, месяц и год. Или столбец со строковым типом данных легко может быть разделен на отдельные символы, что очень часто и выполняется в системах обработки данных. Но не будем занудствовать и требовать безупречного с математической точки зрения определения. Нам понятно, что означает это требование.

Пример. Пусть в базе данных имеется справочник стран — таблица `COUNTRY`. На начальном этапе проектирования некоторой системы обработки данных было ясно, что для целей решения задач предметной области нужна не только сама страна, сколько список сразу всех ее регионов. По этой причине для каждой страны в одной строке таблицы в столбце "Центр региона" был задан список всех ее регионов (точнее, названия центров каждого региона). Результат такого проектирования показан в табл. 2.1, где представлен фрагмент одной строки такой таблицы.

Таблица 2.1. Ненормализованная таблица стран

Код страны	Название страны	Центр региона
RUS	Российская Федерация	Брянск
		Владивосток
		Владикавказ
		Владимир
		Волгоград
		...

Первичным ключом здесь является столбец "Код страны".

Таблица содержит повторяющуюся группу: "Центр региона" (этот столбец содержит не одно, а несколько значений), что нарушает требования первой нормальной формы. В данном случае по правилам нормализации нужно из таблицы стран убрать столбец "Центр региона", а все регионы в виде отдельных строк вынести в другую таблицу `REGION`. Для этой новой таблицы для первичного ключа нужно помимо кода страны задать еще и код региона. Несколько строк такой таблицы показано в табл. 2.2.

В этой таблице первичным ключом будет составной ключ: "Код страны" и "Код региона". Столбец "Код страны" будет внешним ключом, который ссылается на код страны таблицы стран. По значению этого внешнего ключа всегда можно будет определить, к какой стране относится данный регион, и при необходимости выбрать для обработки нужные характеристики страны.

Таблица 2.2. Таблица регионов

Код страны	Код региона	Центр региона
RUS	32	Брянск
RUS	25	Владивосток
RUS	15	Владикавказ
RUS	33	Владимир
RUS	34	Волгоград

2.3.3. Вторая нормальная форма

Вторая нормальная форма (2NF) требует, чтобы соблюдались условия первой нормальной формы и чтобы любой неключевой столбец зависел от всего первичного ключа таблицы, а не от его части. Это правило относится только к тому случаю, когда первичный ключ образован из нескольких столбцов.

Пример. Пусть таблица регионов REGION в процессе проектирования приняла следующий вид (табл. 2.3).

Таблица 2.3. Неверно спроектированная таблица регионов

Код страны	Код региона	Центр региона	Страна
RUS	32	Брянск	Россия
RUS	25	Владивосток	Россия
RUS	15	Владикавказ	Россия
RUS	33	Владимир	Россия
RUS	34	Волгоград	Россия

Первичный ключ для этой таблицы состоит из двух полей — "Код страны" и "Код региона". Столбец "Страна" зависит только от части первичного ключа: "Код страны". Этот столбец следует из таблицы просто убрать. Название (да и любые другие характеристики) страны всегда можно будет найти на основании значения внешнего ключа "Код страны".

2.3.4. Третья нормальная форма

Третья нормальная форма (3NF) требует соблюдения условий второй нормальной формы, и чтобы ни один неключевой столбец не зависел от другого неключевого столбца. Для примера рассмотрим таблицу, описывающую отделы организации (табл. 2.4).

Таблица 2.4. Неверно спроектированная таблица отделов

Код отдела	Название отдела	Код руководителя	Фамилия руководителя
01	Продажи	384	Теплов
02	Маркетинг	291	Ожеред
03	Бухгалтерия	124	Майоров

Столбец "Код отдела" в этой таблице является первичным ключом. Столбец "Фамилия руководителя" зависит не от первичного ключа, а от неключевого столбца "Код руководителя". Столбец "Фамилия руководителя" следует убрать из таблицы. В базе данных должна уже существовать или быть вновь создана таблица, описывающая всех сотрудников организации. Первичным ключом такой таблицы должен быть код сотрудника. В таблице отделов через значение столбца "Код руководителя", который является внешним ключом, ссылающимся на первичный ключ таблицы сотрудников, всегда можно найти все необходимые характеристики руководителя отдела.

Реальные системы, как правило, удовлетворяют требованиям третьей нормальной формы. Скорее всего, только в теории существует множество других "нормальных" форм. Рассмотрим вкратце четвертую, пятую формы и форму Бойса — Кодда.

2.3.5. Другие нормальные формы

Нормальная форма Бойса — Кодда (BCNF) является как бы развитием третьей нормальной формы. Она запрещает в качестве столбца, входящего в состав первичного ключа, использовать столбец, который функционально зависит от неключевого столбца, т. е. значение такого столбца можно выбрать из другой таблицы базы данных. Трудно себе представить разработчиков, которые могут создавать таблицы такой изощренной (или просто неразумной) структуры.

Четвертая нормальная форма (4NF) запрещает независимые отношения типа "один ко многим" между ключевыми и неключевыми столбцами. Это требование на представленном обычном языке звучит довольно странно, однако оно очень четко описывается математически в реляционной алгебре.

Пятая нормальная форма (5NF) доводит процесс нормализации до логического финала, разбивая таблицы на минимально возможные части для устранения в них всей избыточности данных. Нормализованная таким образом таблица обычно содержит минимальное количество данных (чаще всего только один столбец), помимо первичного ключа. При этом общий объем данных в базе данных за счет большого количества таблиц сильно увеличивается, что, как правило, ухудшает производительность системы.

В реальной жизни пятая форма практически не используется.

2.3.6. Денормализация таблиц

Нормализованные таблицы в базе данных позволяют уменьшить избыточность данных, в большинстве случаев увеличивают гибкость использования системы, предоставляя возможность выполнять произвольные, сколь угодно сложные запросы, что временами повышает ее производительность. Это в полном объеме относится к оперативным данным, т. е. к данным, регулярно используемым в ежедневном решении оперативных задач предметной области. Такие задачи называются *задачами оперативной обработки транзакций (OLTP — Online Transaction Processing)*.

Однако в реальной жизни существуют ситуации, когда нарушение правил нормализации и при решении оперативных задач дает возможность увеличить производительность системы, уменьшить требуемый объем внешней памяти для хранения данных.

Пусть, например, для решения каких-то метеорологических задач в базе данных нужно хранить почасовые температуры для различных географических точек нашей страны или по всему миру. Пожалуй, лучшим решением в этом случае будет не создание отдельной таблицы, где будет храниться температура конкретного пункта в конкретное время, а использование в основной таблице в качестве столбца массива, содержащего 24 элемента, по одному элементу на каждый час суток. Это нарушает первое правило нормализации, где запрещается использовать повторяющиеся группы, однако такое решение может сильно уменьшить объем требуемой для работы внешней памяти и, соответственно, повысить производительность системы.

Другой пример, когда можно отойти от правил нормализации. Во многих задачах бизнес-аналитики (Business Intelligence), где требуется выполнение большого количества операций с очень большим объемом данных в базе данных, часто осуществляется отход от требований нормализации. При проектировании подобных баз данных учитываются не столько общие правила создания данных, сколько требуемая функциональность — какие именно действия и как часто должны выполняться с теми или иными данными. Как правило, в подобных случаях происходит увеличение объема внешней памяти, возникает дублирование данных. При этом сокращается время решения задач. Такие задачи называются *оперативным анализом данных (OLAP — Online Analytical Processing)*.

2.4. Проектирование баз данных

Существуют различные подходы к проектированию баз данных. В любом случае при проектировании разрабатывается база данных не сама по себе, "вещь в себе", а с учетом тех задач, для решения которых она будет использоваться.

Одним из наиболее часто применяемых подходов является трехуровневое проектирование всей системы обработки данных и, соответственно, базы данных. Это концептуальный (содержательный), логический и физический уровни.

На первом *концептуальном уровне* осуществляется анализ предметной области, для решения задач которой проектируется система обработки данных и база данных. Выявляются и описываются объекты (object) или сущности (entity) предметной области, их свойства, атрибуты (attribute), определяются связи, отношения (relationship) между сущностями, определяется список задач обработки данных, фиксируются требования к временным и иным характеристикам системы.

В результате такого анализа создается *концептуальная (содержательная) модель* базы данных. В этой модели используется содержательная терминология из предметной области. На этом же этапе определяются те реквизиты, которые могут однозначно идентифицировать выделенные объекты, сущности. Велика вероятность, что на следующем этапе именно эти реквизиты могут войти в состав первичных ключей в соответствующих таблицах, которые будут использованы для хранения сведений об этих объектах.

На втором *логическом уровне* создается *логическая модель* базы данных. В первую очередь создаются все таблицы с использованием операторов Transact-SQL. Создаются необходимые триггеры, хранимые процедуры, представления, пользовательские функции. Часто этого бывает достаточно для получения хорошо спроектированной базы данных.

Третий *физический уровень* проектирования используется для настройки физических характеристик базы данных. Возможность влиять на физические аспекты хранения данных может привести к повышению производительности и даже надежности всей системы. MS SQL Server предоставляет средства тонкой настройки физических характеристик базы данных, позволяя для одной базы данных создавать несколько файлов данных, файловые группы, указывая, какие данные и в каком порядке должны размещаться в отдельных файловых группах, в конкретных файлах. Можно создавать секционированные таблицы, дающие возможность повысить производительность системы, ее отказоустойчивость.

На практике при разработке отдельных систем обработки данных то ли по причине дефицита времени, то ли от врожденной лености многие разработчики часто опускают формальную фиксацию результатов концептуального проектирования, ограничиваясь лишь общим описанием содержательных требований к системе, к обрабатываемым данным. Надо сказать, что в большинстве случаев это является оправданным, поскольку разработчики оперируют в своей деятельности понятиями в первую очередь логического уровня (таблицы, столбцы, ограничения и т. д.) и результаты их проектирования удовлетворяют все требования к системе. Правда, своих студентов я заставляю выполнять концептуальное проектирование данных и описывать результаты этого проектирования.

2.5. Язык Transact-SQL

В SQL Server в качестве языка работы с базами данных используется язык, который называется *Transact-SQL*. Этот язык является несколько измененным и расширенным вариантом языка SQL, определенного в международных стандартах. Еще

говорят, что Transact-SQL является *диалектом* стандарта SQL. Операторы языка позволяют выполнять создание, изменение и удаление объектов базы данных и самой базы данных, создавать триггеры, хранимые процедуры, добавлять, изменять, удалять и отыскивать данные в таблицах базы данных.

Основной законченной единицей языка Transact-SQL является оператор (statement), который может состоять из нескольких предложений (clause). При написании операторов и предложений используются константы (литералы) и ключевые слова. Это слова, которые нельзя использовать в качестве имен объектов базы данных. (Вообще-то это не совсем так. Ключевые слова можно использовать в так называемых идентификаторах с разделителями, о которых мы скажем чуть позже в этой главе.)

В любом формальном (в том числе) языке выделяются, в первую очередь, синтаксис и семантика (о прагматике языковых средств мы говорить не будем).

Синтаксис — это способ построения правильных языковых конструкций. Поскольку формальные языки в тысячи раз проще естественных, то всегда можно четко и недвусмысленно описать их синтаксис. Например, можно точно описать синтаксис оператора создания базы данных CREATE DATABASE. Для задания синтаксиса используются довольно простые и удобные средства. Такие средства мы рассмотрим чуть далее.

Семантика — это смысл синтаксически правильно построенных языковых конструкций. Семантика конструкций описывается, объясняется с использованием обычного естественного языка. При описании семантики, например, оператора CREATE DATABASE можно сказать, что этот оператор позволяет создать базу данных для текущего экземпляра сервера базы данных (или подключить базу данных к списку доступных баз данных), а отдельные предложения в этом операторе дают возможность описывать конкретные характеристики создаваемой (подключаемой) базы данных.

2.5.1. Синтаксис

Для четкого описания синтаксических конструкций языка Transact-SQL мы будем применять несколько расширенную систему обозначений (нотации) Бэкуса — Наура, что принято во всем мире при описании синтаксиса большинства формальных языков, используемых в программировании. Кроме того, те же самые конструкции будут описываться и при помощи графических средств — R-графами.

В нотациях Бэкуса — Наура в угловые скобки < > заключают определяемый синтаксический элемент, конструкцию, понятие языка. Например, конструкция <идентификатор> задает понятие идентификатора в Transact-SQL. Такая конструкция называется "нетерминальным символом", т. е. символом или выражением, которое будет определено в "терминальных" символах — в символах, которые, упрощенно говоря, можно ввести с клавиатуры компьютера.

Символы ::= означают "по определению есть". Нетерминальный символ, стоящий слева от этой конструкции, определяется выражением, записанным справа от этой конструкции. Например, следующая формула определяет цифру (десятичную):

```
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Символ вертикальной черты | означает "или", т. е. цифрой является 0 или 1 или 2 и т. д.

В языках программирования существует понятие не только десятичной, но и шестнадцатеричной цифры (не говоря уж о двоичных цифрах; раньше в вычислительной технике использовались и восьмеричные цифры). Шестнадцатеричная цифра определяется следующей синтаксической конструкцией:

```
<шестнадцатеричная цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c
| d | e | f
```

Здесь латинскими буквами от a до f обозначаются шестнадцатеричные цифры, которые соответствуют числам 10, 11, 12, 13, 14 и 15. Эти буквы могут задаваться в любом регистре, т. е. в виде строчных или прописных букв.

Еще пример. Определение буквы:

```
<буква> ::= <буква латинского алфавита> | <буква кириллицы>
```

Здесь один (нетерминальный) символ определяется через другие нетерминальные символы. Такие символы из правой части должны быть в дальнейшем определены через терминальные символы. Дадим эти определения.

```
<буква латинского алфавита> ::= <строчная латинская буква>
| <прописная латинская буква>
```

Здесь все еще нетерминальный символ определяется через другие нетерминальные символы. Наведем окончательный порядок.

```
<строчная латинская буква> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z
```

```
<прописная латинская буква> ::= A | B | C | D | E | F | G | H | I | J | K | L |
M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
```

Аналогичным образом описывается и буква кириллицы:

```
<буква кириллицы> ::= <строчная буква кириллицы>
| <прописная буква кириллицы>
```

```
<строчная буква кириллицы> ::= а | б | в | г | д | е | ё | ж | з | и | й | к | л |
м | н | о | п | р | с | т | у | ф | х | ц | ч | ш | щ | ъ | ы | ь | э | ю | я
```

```
<прописная буква кириллицы> ::= А | Б | В | Г | Д | Е | Ё | Ж | З | И | Й | К | Л |
М | Н | О | П | Р | С | Т | У | Ф | Х | Ц | Ч | Ш | Щ | ъ | ы | ь | э | ю | я
```

Если же список элементов, разделенных символом вертикальной черты |, заключен в фигурные скобки { }, то из этого списка должен быть выбран в точности один элемент. Если же такой список заключен в квадратные скобки [], то из списка можно выбрать один или ни одного элемента. Любая конструкция, заключенная в квадратные скобки, является необязательной, ее можно опустить.

Если в синтаксическом описании языковой конструкции какой-то элемент в списке является подчеркнутым, то этот элемент будет элементом по умолчанию — именно он будет выбран, если не задан ни один элемент из списка. Например, при задании размера файла в операторе CREATE DATABASE синтаксис предложения SIZE записан в следующем виде:

```
SIZE = <целое> [ KB | MB | GB | TB ]
```

Здесь в квадратных скобках указаны единицы измерения — килобайты, мегабайты, гигабайты или терабайты. Если при описании размера не указать единицы измерения, то будет принято значение по умолчанию — мегабайты (МВ), т. к. в синтаксической конструкции это значение подчеркнуто. В этом случае запись

`SIZE = 18 МВ`

эквивалентна записи

`SIZE = 18`

Хочу сказать, что ориентация при написании операторов любого языка на некоторые значения по умолчанию не является таким уж хорошим делом. Бывали случаи, что в новых версиях систем значения по умолчанию изменялись, что иногда приводило к печальным последствиям в виде долгого поиска ошибок при переходе на другую версию. Кроме того, некоторые значения по умолчанию устанавливаются специальными средствами на уровне всей системы или на уровне базы данных. Какие в конкретный момент времени существуют установки, не всегда точно известно. Лучшим решением все-таки является явное задание необходимых значений. Это избавит вас от лишних приключений при переходе на новые версии, а также улучшит документированность ваших скриптов.

В так называемом расширенном варианте нотаций Бэкуса — Наура присутствует и символ многоточия ..., который означает, что предыдущая конструкция может повторяться произвольное количество раз.

Пример. Следующая далее конструкция обычно применяется для описания списка параметров, передаваемых функции или хранимой процедуре. Список параметров заключается в круглые скобки.

`([<параметр> [, <параметр>] ...])`

Эта конструкция означает, что в списке могут отсутствовать параметры (вся синтаксическая конструкция внутри круглых скобок заключена в квадратные скобки), может присутствовать один параметр или произвольное количество параметров, отделенных друг от друга запятыми. Круглые скобки нужны в любом случае.

Целое число

Дадим определение целого числа (или просто целого, как принято говорить в программировании) в нотациях Бэкуса — Наура:

`<целое> ::= <цифра>...`

Целое — это цифра (десятичная цифра), которая может повторяться произвольное количество раз, но не менее одного раза.

В качестве R-графа при описании синтаксиса языковых конструкций используется ориентированный граф.

При помощи R-графа синтаксис целого числа можно представить следующим образом (граф 2.1):



Граф 2.1

Чтение графа выполняется слева направо. Здесь стрелки, соединяющие два узла, задают обход по графу. Если над стрелкой находится какой-либо символ (терминальный или нетерминальный), то в порождаемую синтаксическую конструкцию должен быть добавлен этот символ. Стрелка без задания над ней символа означает просто возможность перехода по графу к следующему узлу. Такой переход является возможным, но не обязательным.

Целое со знаком

Целое со знаком — это целое число, перед которым стоит знак числа: + или - (граф 2.2).

<целое со знаком> ::= {+ | -}<целое>



Граф 2.2

Число с плавающей точкой

Число с плавающей точкой — это число со знаком, содержащее дробное значение и/или показатель степени числа 10, заданный после латинской буквы "Е" (в любом регистре). Показатель степени также может иметь знак числа.

<число с плавающей точкой> ::=

[+ | -] {[<целое1>].<целое2> | <целое1>} [{E | e} [+ | -]<целое3>]

Здесь целое1 — целая часть числа, целое2 — дробная часть, целое3 — показатель степени 10.

Можно заметить, что приведенное определение числа с плавающей точкой не слишком удобно для восприятия, а точнее — очень неудобно. Гораздо более понятным является синтаксис, представленный R-графом (граф 2.3):



Граф 2.3

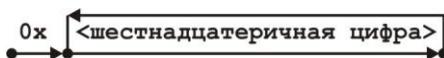
Здесь видно, что числу с плавающей точкой может предшествовать знак числа. Знак может отсутствовать (знаки заключены в квадратные скобки в нотации Бэкуса — Наура, обходящая стрелка сверху над знаками в R-графе). Само число может содержать целую и дробную части, разделенные десятичной точкой, причем любая часть может отсутствовать. После этого может идти латинская буква "Е" или "е", за которой указывается значение порядка числа. Порядок может содержать знак числа: + или -.

Двоичное число

Двоичное число (binary) начинается с символов 0x, за которыми следует не менее одной шестнадцатеричной цифры:

<двоичное число> ::= <шестнадцатеричная цифра>...

R-граф (граф 2.4):



Граф 2.4

ЗАМЕЧАНИЕ

Вообще-то, как вы видите, здесь задается синтаксис шестнадцатеричного числа, однако, следуя принятой терминологии, мы также будем называть такую конструкцию двоичным числом, хотя такие константы не слишком часто используются в базах данных.

Комментарии

В SQL существует возможность в любой позиции, где допустимы пробелы, задавать комментарии. Любой произвольный текст может быть заключен между символами /* и */. Этот текст служит только для объяснения, что конкретно выполняется в соответствующем операторе. Такой комментарий может занимать произвольное количество строк.

Вы также можете задать примечания, набрав подряд два знака минус --. В этом случае комментарий распространяется только до конца текущей строки.

ЗАМЕЧАНИЕ

Часто программисты, имеющие опыт работы с различными языками программирования, пытаются задавать однострочный комментарий, набрав две наклонные черты //. Я и сам иногда так поступаю. Нехорошая привычка.

Строковые константы

Строковые константы можно всегда заключать в апострофы (''). Стока может содержать любые символы. Если в самой строке присутствует символ апострофа, то он должен быть записан дважды. Если значение опции QUOTED_IDENTIFIER установлено в OFF, то строковые константы также можно заключать и в кавычки (""). В этом случае кавычка внутри строки должна быть представлена подряд идущими двумя кавычками.

<строковая константа> ::=
{'<любой символ> | ''}... | {"<любой символ> | ""}..."

Примеры строковых констант:

'MS SQL Server 2012'

'Ann''s daughter'

```
SET QUOTED_IDENTIFIER OFF;
GO
"Ann's daughter"
```

Настоятельно рекомендую всегда строковые константы заключать только в апострофы. В некоторых системах управления базами данных я наблюдал такую картину, когда константы в кавычках в одном предложении оператора воспринимались системой нормально, а в другом предложении этого же оператора вызывали ошибку.

Все операторы Transact-SQL должны завершаться символом точки с запятой (;). Присутствие такого терминатора для оператора также определено и стандартом SQL-92.

ВНИМАНИЕ!

Оператор GO, о котором мы будем говорить в следующей главе, не должен завершаться символом точки с запятой. Наличие этого терминатора вызывает синтаксическую ошибку. Сам оператор GO не является оператором языка Transact-SQL. Это оператор, а точнее команда, утилиты sqlcmd и программы Management Studio.

Идентификатор

Важной (и при этом довольно простой) составной частью синтаксиса любого формального языка, связанного с программированием, является *идентификатор*.

Все объекты базы данных (сама база данных, логические файлы базы данных, таблицы, представления, внутренние переменные, параметры, триггеры и т. д.) должны иметь имена, которые также называют идентификаторами. Большинству объектов вы должны явно присвоить имена. Ограничениям таблицы (PRIMARY KEY, UNIQUE, FOREIGN KEY и CHECK — см. главу 5) вы можете имена явно не указывать — система автоматически присвоит им идентификаторы, которые вам могут и не понравиться. Очень рекомендую всем объектам базы данных (да и всем переменным, объектам классов в программах) присваивать осмысленные имена. Для идентификаторов существуют некоторые ограничения. Обычные идентификаторы должны содержать определенные символы, но они не должны содержать пробелов, специальных символов, в том числе некоторых разделителей. Они не могут быть зарезервированными словами языка Transact-SQL.

В различных языках программирования и в системах управления базами данных правила задания идентификаторов несколько отличаются.

В SQL Server существует два вида идентификаторов — обычные (иногда их называют регулярными, regular) и идентификаторы с разделителями (delimited identifier).

Обычные идентификаторы записываются по правилам, похожим на правила, принятые в нормальных языках программирования. Первым символом должна идти буква (латинская, а если вы при инсталляции SQL Server задали порядок сортировки Cyrillic_General_CI_AS, то допустимо использование и букв кириллицы в любой позиции идентификатора), символ подчеркивания (_), символ @ или #. Последующими символами могут быть буквы (латинские и буквы кириллицы), десятичные цифры, символы _, @, # и \$.

Обычный идентификатор нечувствителен к регистру. Идентификаторы `name1` и `NAME1` являются одинаковыми. Это также верно и для идентификаторов, в которых использованы буквы кириллицы. Например, идентификаторы `Таблица1` и `таблица1` будут рассматриваться системой как одинаковые идентификаторы.

Идентификаторы, начинающиеся с символа `@`, используются для именования локальных переменных или параметров. Не применяйте подобные идентификаторы для именования объектов вашей базы данных.

Идентификаторы, которые начинаются с символа `#`, применяются для имен временных объектов — таблиц или процедур. Такие имена также не следует использовать для обычных объектов базы данных.

Обычный идентификатор не может быть зарезервированным словом языка Transact-SQL. Список зарезервированных слов представлен в *приложении 2*. Тем не менее, зарезервированные слова могут использоваться в качестве имен объектов базы данных при использовании идентификаторов с разделителями.

Идентификатор с разделителями заключается либо в кавычки (`"`), либо в квадратные скобки — в этом случае он размещается между левой (`[`) и правой (`]`) квадратными скобками. Есть две причины использовать идентификаторы с разделителями — применение в качестве идентификатора зарезервированных слов (ну никак не могу одобрить такое их использование, хотя наблюдал случаи, когда необходимость применения этих слов довольно правдоподобно объяснялась разработчиками конкретных баз данных). Во втором случае идентификаторы с разделителями используются для именования объектов базы данных с применением пробелов или с использованием символов, отличных от допустимых в обычных именах. При таком способе именования объектов можно создавать имена, понятные любому не-посвященному во все премудрости программирования человеку.

Идентификаторы с разделителями чувствительны к регистру. Здесь идентификаторы `"name1"` и `"NAME1"`, а также `[name2]` и `[NAME2]` все являются различными идентификаторами. Конечные пробелы в идентификаторах с разделителями всегда отбрасываются.

Ограничители `[` и `]` при объявлении идентификатора с разделителями можно использовать всегда, при любых установках системы. Кавычки в качестве ограничителей для идентификаторов можно использовать только в том случае, когда значение опции `QUOTED_IDENTIFIER` для текущего соединения с сервером базы данных установлено в `ON` (значение по умолчанию). В этом случае строковые константы можно заключать только в апострофы `'`, но не в кавычки `"`. Если же значение опции `QUOTED_IDENTIFIER` задано `OFF`, то использование кавычек для задания идентификатора с разделителями не допускается. В этом случае для идентификаторов с разделителями можно применять только квадратные скобки, а для задания строковых литералов можно использовать как апострофы, так и кавычки. Надо сказать, что правила эти хотя и немного запутанные, но тем не менее довольно разумные, они позволяют избежать двусмысленности при записи операторов Transact-SQL.

Значение опции `QUOTED_IDENTIFIER` для *текущего сеанса* установлено в `ON` по умолчанию. Для того чтобы изменить его значение для *конкретной базы данных*, нужно

выполнить оператор ALTER DATABASE, в котором необходимо установить значение этой опции в ON или OFF. Для текущего сеанса работы с базой данных можно выполнить оператор SET QUOTED_IDENTIFIER.

ВНИМАНИЕ!

Имена переменных и параметров хранимых процедур должны быть только обычными идентификаторами. Идентификаторы с разделителями в этом случае не распознаются в SQL Server.

Если в идентификаторе с разделителями, который заключен в квадратные скобки, нужно задать символ правой (только правой) квадратной скобки], то этот символ нужно записать дважды. Например, для задания имени

Код [Code] вида деятельности

нужно записать:

[Код [Code]] вида деятельности]

Аналогичным образом в идентификаторе, заключенном в кавычки, любой символ кавычки (за исключением обрамляющих кавычек) должен повторяться дважды. Например, для идентификатора

Наименование, "обозначение", расположения

запишем:

"Наименование, ""обозначение""", расположения"

ЗАМЕЧАНИЕ

Чтобы не быть связанными с установками системы, для записи идентификаторов с разделителями в SQL Server лучше использовать квадратные скобки, а для строковых констант — апострофы, хотя применение квадратных скобок и не соответствует принятому стандарту SQL.

Теперь дадим формальное описание синтаксиса для идентификаторов.

```
<идентификатор> ::= <обычный идентификатор>
                  | <идентификатор с разделителями>
<обычный идентификатор> ::= <буква> | _ | @ | #
| <обычный идентификатор><буква>
| <обычный идентификатор><цифра>
| <обычный идентификатор>_
| <обычный идентификатор>@
| <обычный идентификатор>#
| <обычный идентификатор>$
```

Здесь видно, что идентификатор должен начинаться с буквы, символов _, @ или #, за которыми могут следовать символы буква, цифра, _, @, #, \$.

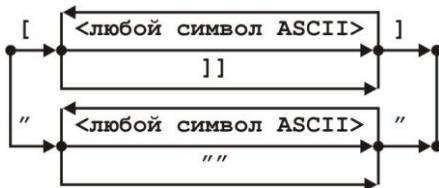
```
<идентификатор с разделителями> ::=
[<любой символ>...] | "<любой символ>..."
```

Эта формула не учитывает необходимости дублирования символов] и ". Давайте внесем небольшое уточнение:

```
<идентификатор с разделителями> ::=  
[ {<любой символ> | ||}... ] | {<любой символ> | ""}..."
```

В последних двух формулах может возникнуть некоторая путаница, поскольку символы квадратных скобок здесь используются не как металингвистические переменные, а как символы языка Transact-SQL. На всякий случай я выделяю их полужирным шрифтом.

R-граф этого синтаксиса не дает такой путаницы (граф 2.5):

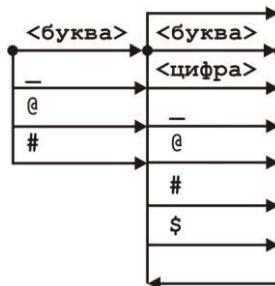


Граф 2.5

Представленное определение обычного идентификатора, хотя является и правильным, и в некотором смысле "классическим", тем не менее, слишком громоздкое. Читаемость такого определения не слишком хорошая. Другой вариант представления этого синтаксиса:

```
<обычный идентификатор> ::= { <буква> | _ | @ | # }  
[ <буква> | <цифра> | _ | @ | # | $ ]...
```

R-граф определения обычного идентификатора (граф 2.6):



Граф 2.6

Теперь все в порядке.

Все идентификаторы (как обычные, так и с разделителями) для "нормальных", чаще всего используемых объектов базы данных, могут содержать не более 128 символов. Количество символов в идентификаторах, используемых для именования временных объектов, не может превышать 116. При подсчете количества символов в идентификаторе не учитываются ограничители (символы " или [и]) для идентификаторов с разделителями.

ЗАМЕЧАНИЕ

На одном из совещаний, относящихся к практическому использованию баз данных одной из любимых мною версий реляционных СУБД, я присутствовал при разговоре, где умные ребята совершенно серьезно обсуждали вопрос, как важно было бы увеличить количество символов, отводимых для именования объектов базы данных. Насколько могу вспомнить, речь шла об увеличении количества символов со 128 до 256. Лишь я сильно сомневаюсь, что большое количество символов в именовании объектов поможет улучшить понимание состава и назначения объектов базы данных. В моей практике не встречалось объектов, размер имени которых превышал бы символов эдак 30.

В SQL Server существует такая возможность, как *расширенные свойства* (extended properties). Здесь можно описать дополнительные характеристики любого объекта базы данных. Кроме того, расширенные свойства являются структурированными, они позволяют не просто дать текстовые описания, но и, например, задать заголовок, который может быть использован в различных программах при отображении значений какого-либо столбца таблицы. Так что для улучшения понимаемости базы данных нет особой необходимости давать объектам очень уж длинные имена.

ЗАМЕЧАНИЕ

В документации и в литературе по SQL Server термины "идентификатор" и "имя" часто используются как синонимы. Временами это приводит к некоторой путанице. Например, каждая база данных в системе имеет имя, построенное по только что рассмотренным правилам. База данных также имеет и идентификатор, который является целым числом. Давайте для синтаксических конструкций, которые именуют объекты базы данных, будем использовать термин "имя".

2.5.2. Основные сведения о составе языка Transact-SQL

Язык SQL и его диалект, используемый в SQL Server, Transact-SQL, можно представить в виде группы подъязыков, частей. По традиции каждый такой подъязык называют языком.

В Transact-SQL выделяются следующие части:

- ◆ язык определения данных (DDL, Data Definition Language);
- ◆ язык манипулирования данными (DML, Data Manipulation Language);
- ◆ язык управления доступом к данным (DCL, Data Control Language);
- ◆ язык управления транзакциями (TCL, Transaction Control Language);
- ◆ язык хранимых процедур и триггеров или процедурное расширение SQL (Stored Procedures and Triggers Language).

DDL применяется для работы с объектами базы данных, с метаданными. Для действий с метаданными используются следующие группы операторов:

- ◆ CREATE. Это операторы, при помощи которых создаются новые объекты базы данных — в первую очередь таблицы, затем пользовательские типы данных, индексы, хранимые процедуры, триггеры, роли и др. При использовании оператора CREATE DATABASE создается и сама база данных.

- ◆ **DROP.** Операторы этого вида удаляют ранее созданные ненужные, как потом выяснилось, объекты базы данных. Те же таблицы, пользовательские типы данных и иные объекты. Оператор позволяет удалить и базу данных.
- ◆ **ALTER.** Это операторы, которые позволяют изменить уже существующие в базе данных ранее созданные объекты и характеристики базы данных.

Для работы с собственно данными в базах данных используются операторы DML, позволяющие создавать, изменять и удалять данные. В состав DML входит и оператор, выполняющий одну из наиболее важных функций в базе данных. Это оператор поиска, выборки данных.

Для данных в базе данных используются четыре основных оператора:

- ◆ добавления данных `INSERT`;
- ◆ изменения существующих данных `UPDATE`;
- ◆ удаления данных `DELETE`;
- ◆ выборки (поиска) данных `SELECT`.

В DML есть и некоторые другие операторы, которые мы с вами рассмотрим в соответствующих главах.

Язык управления доступом к данным DCL содержит операторы, назначающие, отменяющие и удаляющие полномочия к объектам базы данных для пользователей и ролей, а именно:

- ◆ предоставления полномочий к защищаемому объекту `GRANT`;
- ◆ отмены полномочия `DENY`;
- ◆ удаления полномочия `REVOKE`.

Язык управления транзакциями TCL включает в себя операторы, осуществляющие запуск, подтверждение, откат или создание точки сохранения транзакции. Это следующие операторы:

- ◆ операторы старта обычной или распределенной транзакции: `BEGIN TRANSACTION` и `BEGIN DISTRIBUTED TRANSACTION`;
- ◆ операторы подтверждения транзакции: `COMMIT TRANSACTION`, `COMMIT WORK`;
- ◆ операторы отката транзакции: `ROLLBACK TRANSACTION`, `ROLLBACK WORK`;
- ◆ оператор создания точки сохранения `SAVE TRANSACTION`.

Язык хранимых процедур и триггеров содержит операторы, обеспечивающие процедурные императивные средства обработки данных. Язык используется в соответствии с его названием при создании хранимых процедур, функций, определенных пользователем, и триггеров.

Что будет дальше?

В следующей главе, довольно большой по размеру, мы рассмотрим средства создания, изменения и удаления баз данных в SQL Server.



ГЛАВА 3

Работа с базами данных

- ◆ Запуск и останов экземпляра сервера
- ◆ Системные и пользовательские базы данных
- ◆ Характеристики баз данных, файлов и файловых групп
- ◆ Средства получения сведений о характеристиках баз данных и их объектах
- ◆ Создание, изменение, удаление баз данных средствами Transact-SQL и при использовании SQL Server Management Studio
- ◆ Присоединение ранее созданной базы данных
- ◆ Создание мгновенных снимков базы данных
- ◆ Создание схем в базе данных
- ◆ Средства копирования и восстановления баз данных
- ◆ Домашнее задание. Создание реальной базы данных

Прежде чем приступить к созданию объектов в базе данных и начать работать с данными в базе, нужно сначала создать саму базу данных. В ней будут храниться все необходимые для решения задач предметной области данные. Туда же вы можете помещать создаваемые хранимые процедуры, триггеры, пользовательские типы данных, функции, представления.

Здесь мы рассмотрим не только создание, изменение и удаление баз данных. Существующий в языке Transact-SQL оператор `CREATE DATABASE`, который в первую очередь должен создавать базы данных, позволяет также выполнить и еще несколько довольно интересных и полезных функций: присоединение созданной на другом компьютере базы данных к системе и создание мгновенного снимка базы данных (`SNAPSHOT`). Мы рассмотрим ряд системных представлений, позволяющих получить сведения о базах данных, описанных в системе, и об их файлах, а также несколько системных функций, которые пригодятся вам в повседневной жизни.

С базами данных в SQL Server связано понятие схемы (schema). Мы также выясним, что это такое, и создадим парочку схем для нашей базы данных.

Выполнение всех действий проиллюстрируем при использовании операторов языка Transact-SQL. Операторы Transact-SQL будем выполнять как при вызове утилиты `sqlcmd` в командной строке (или в PowerShell), так и в графической среде Management Studio. Создание баз данных и изменение их характеристик выполним также и с применением диалоговых средств системы, представленных в этом наиболее важном и удобном в работе графическом инструменте администрирования и разработки SQL Server — Management Studio. Рекомендую выполнить на вашем компьютере все те примеры, которые здесь описаны. Если же у вас не так много времени на эту деятельность или нет особого желания, то просто внимательно просмотрите предложенные операции и получаемые при их выполнении результаты.

База данных в SQL Server является довольно сложным объектом с множеством характеристик. Все перечисленное ранее мы будем рассматривать по порядку и выполнять необходимые действия по созданию, удалению баз данных, изменению их характеристик, созданию мгновенных снимков и схем, чтобы научиться работать с базами на профессиональном уровне.

3.1. Запуск и останов экземпляра сервера

При первоначальной загрузке вашего компьютера, если вы при инсталляции SQL Server задали автоматический старт компонента Database Engine, то он будет запущен на выполнение. В принципе на одном компьютере может одновременно выполняться несколько экземпляров сервера SQL Server, в том числе и разных версий. Тот сервер базы данных, с которым мы выполняем соединение в одной из наших программ (в первую очередь это будет утилита командной строки `sqlcmd` и программа SQL Server Management Studio), называется *текущим экземпляром сервера*.

Версия SQL Server Enterprise позволяет иметь до 50 экземпляров сервера, версия Express — до 16. Только один экземпляр является экземпляром по умолчанию. Каждый экземпляр содержит свои версии системных баз данных, имеет набор своих характеристик и содержит свой набор пользовательских баз данных.

При запуске и останове экземпляра сервера используется его имя. При инсталляции мы задали имя экземпляра по умолчанию `MSSQLSERVER`, точнее, указали, что он будет экземпляром по умолчанию, а система присвоила ему имя `MSSQLSERVER`.

3.1.1. Запуск на выполнение экземпляра сервера

Если при инсталляции SQL Server не был задан автоматический запуск нужного экземпляра сервера при загрузке операционной системы или вы останавливали его выполнение, то экземпляр нужно запустить вручную из командной строки (или в PowerShell) либо при использовании программ Configuration Manager или Management Studio.

Запуск сервера из командной строки

Для запуска экземпляра сервера из обычной командной строки или из программы PowerShell нужно ввести и выполнить команду `net start`:

```
net start "SQL Server (MSSQLSERVER)"
```

В скобках задается идентификатор сервера. Здесь нужно указать то имя, которое вы задали при инсталляции SQL Server.

Так как мы устанавливали экземпляр сервера как экземпляр по умолчанию, то для запуска сервера можно выполнить команду в более простом виде:

```
net start "MSSQLSERVER"
```

ЗАМЕЧАНИЕ

Для запуска и останова сервера из командной строки существует еще один вариант команды `net start`. В командной строке и в PowerShell такой вариант имеет несколько отличающийся синтаксис. Не станем им пользоваться.

Запуск сервера из программы Configuration Manager

Запустить на выполнение сервер базы данных можно из программы Configuration Manager.

Чтобы вызвать на выполнение эту программу, нужно щелкнуть по кнопке Пуск, выбрать **Все программы | Microsoft SQL Server 2012 | Configuration Tools | SQL Server Configuration Manager**. Появится главное окно программы (рис. 3.1). В правой, основной, части окна будет отображаться список всех сервисов системы SQL Server и их состояний.

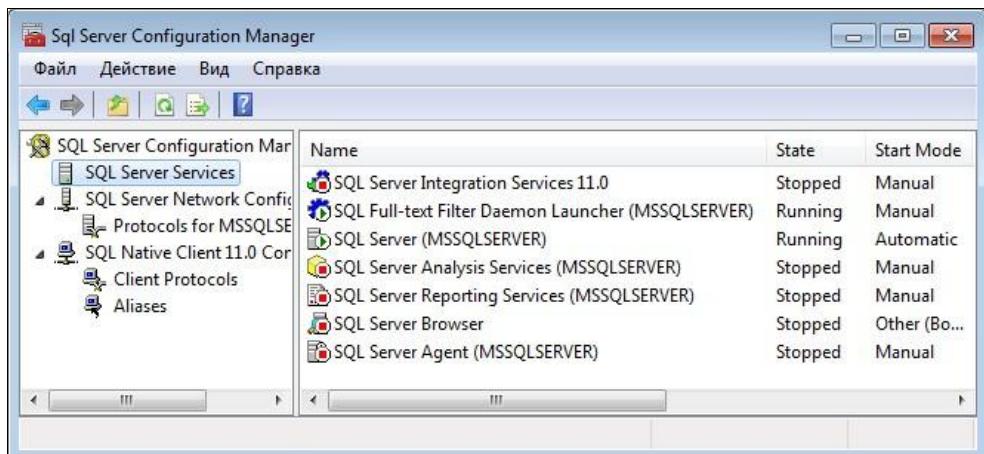


Рис. 3.1. Главное окно программы SQL Server Configuration Manager

В левой панели нужно выполнить двойной щелчок по строке **SQL Server Configuration Manager** и затем выделить мышью строку **SQL Server Services**. В правой панели окна нужно найти соответствующий сервер: строку **SQL Server**

(**MSSQLSERVER**), щелкнуть по ней правой кнопкой мыши и в появившемся контекстном меню выбрать элемент **Start**. Через некоторое время сервер будет запущен.

Одно и то же действие в SQL Server чаще всего может быть выполнено различными способами. В программе SQL Server Configuration Manager можно правой кнопкой мыши щелкнуть по имени сервера **SQL Server (MSSQLSERVER)** и в появившемся контекстном меню выбрать элемент **Свойства** либо дважды щелкнуть мышью по имени сервера. Появится диалоговое окно свойств этого сервера (рис. 3.2).

Для запуска сервера на выполнение в этом окне нужно щелкнуть по кнопке **Start**.

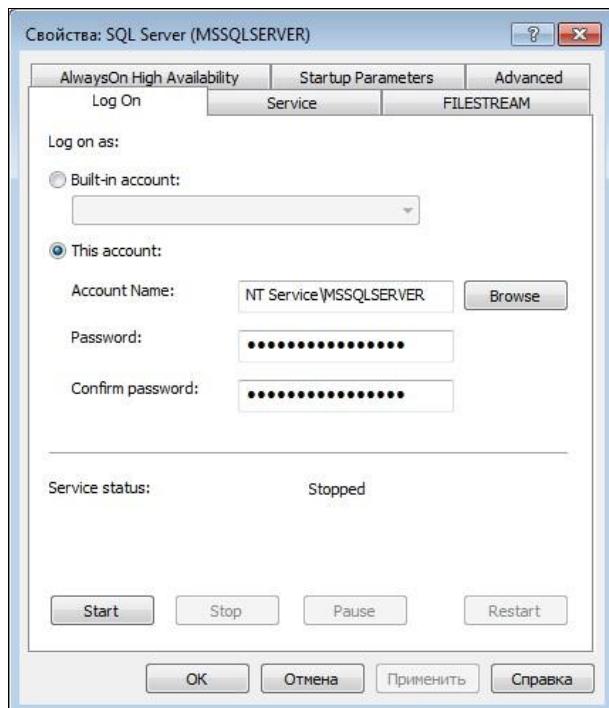


Рис. 3.2. Диалоговое окно свойств сервера

ЗАМЕЧАНИЕ

В этом же диалоговом окне при использовании и других вкладок можно изменить некоторые характеристики сервера, которые были первоначально установлены в процессе инсталляции системы. Будем считать, что все характеристики мы с вами с самого начала задали правильно, и изменять их не будем. Пока.

Запуск сервера с помощью программы Management Studio

Запустить на выполнение сервер можно также при помощи компонента SQL Server Management Studio. Чтобы вызвать на выполнение эту программу, щелкните по кнопке **Пуск**, выберите **Все программы | Microsoft SQL Server 2012 | SQL Server Management Studio**. Появится диалоговое окно соединения с сервером (рис. 3.3).



Рис. 3.3. Диалоговое окно соединения с сервером

Здесь нужно щелкнуть по кнопке **Cancel**, чтобы отменить соединение с сервером, который пока еще не запущен на выполнение, иначе мы получим сообщение об ошибке. Появится главное окно программы Management Studio. Если в окне не видна панель **Registered Servers**, нужно в главном меню выбрать **View | Registered Servers** или нажать комбинацию клавиш **<Ctrl>+<Alt>+G**.

Чтобы получить список экземпляров серверов SQL Server, установленных на вашем компьютере, нужно в левой части окна, в панели **Registered Servers** (зарегистрированные серверы) раскрыть **Database Engine**, дважды щелкнув по нему мышью или щелкнув по символу "+" слева от имени этого элемента, затем таким же способом раскрыть **Local Server Groups** (рис. 3.4). На рисунке видно, что у меня зарегистрирован один экземпляр сервера, и он выполняется.

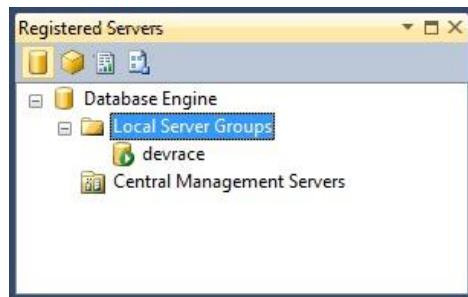


Рис. 3.4. Список зарегистрированных экземпляров серверов SQL Server

Поскольку при инсталляции мы указали экземпляра по умолчанию (см. главу 1), то в списке он представлен только именем компьютера (в моем случае devrake). Иначе для ссылок на сервер, который не является сервером по умолчанию, используется конструкция: имя компьютера\имя сервера, например, для версии Express Edition следовало бы указать: devrake\sqlexpress.

Если в списке зарегистрированных серверов не отображается нужный вам для работы сервер по умолчанию, то следует щелкнуть правой кнопкой мыши по элементу **Local Server Groups** (группа локальных серверов) и в контекстном меню выбрать элемент **New Server Registration** (регистрация нового сервера). Появится окно регистрации нового сервера **New Server Registration** (рис. 3.5).

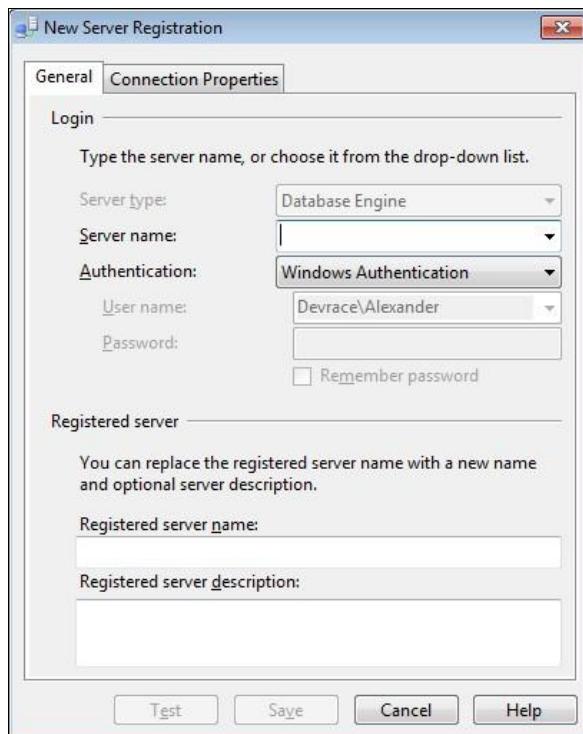


Рис. 3.5. Окно регистрации нового экземпляра сервера

На вкладке **General** из раскрывающегося списка **Server name** нужно выбрать соответствующий элемент. Если же его в списке нет, то следует вручную набрать в этом поле только имя вашего компьютера, не указывая имя сервера (что, напомню, будет означать, что вы регистрируете сервер по умолчанию). Аутентификацию (значение раскрывающегося списка **Authentication**) нужно оставить в виде **Windows Authentication** (аутентификация Windows).

Для проверки правильности регистрации следует щелкнуть по кнопке **Test**. Эта кнопка станет доступной после ввода имени сервера и выбора метода аутентификации. Если параметры заданы правильно, то появится окно **New Server Registration**, сообщающее об успешности проверки соединения с сервером. Щелкните по кнопке **OK**. После этого в окне регистрации сервера щелкните по кнопке **Save**.

В списке зарегистрированных серверов щелкните правой кнопкой мыши по имени сервера (см. рис. 3.4) и в контекстном меню выберите элемент **Service Control** (управление сервисом), в появившемся подменю выберите элемент **Start**. Появится

окно запроса подтверждения, в котором нужно щелкнуть по кнопке **Да**. Через некоторое время сервер будет запущен на выполнение.

3.1.2. Останов экземпляра сервера

Останов сервера из командной строки

Из командной строки или из PowerShell запущенный на выполнение сервер останавливается при выполнении команды:

```
net stop "SQL Server (MSSQLSERVER)"
```

Так как экземпляр сервера был установлен как экземпляр по умолчанию, то можно выполнить команду:

```
net stop "MSSQLSERVER"
```

Останов сервера из программы Configuration Manager

Здесь остановить выполнение сервера можно двумя способами. В главном окне программы Configuration Manager (см. рис. 3.1) нужно в правой основной панели щелкнуть правой кнопкой мыши по имени сервера и в контекстном меню выбрать элемент **Stop**.

Второй вариант: щелкнуть по имени сервера правой кнопкой мыши и в контекстном меню выбрать элемент **Свойства**. Появится диалоговое окно свойств (см. рис. 3.2), где нужно щелкнуть мышью по кнопке **Stop**.

Останов сервера из программы Management Studio

Для останова сервера в левой панели главного окна Management Studio нужно щелкнуть правой кнопкой мыши по имени сервера (см. рис. 3.4), в контекстном меню выбрать элемент **Service Control**, в появившемся подменю выбрать элемент **Stop**. Откроется диалоговое окно подтверждения останова, где нужно щелкнуть по кнопке **Да**.

3.2. Что собой представляет база данных в SQL Server

Сведения обо всех базах данных, относящихся к текущему экземпляру сервера базы данных, хранятся в системной базе данных *master*. Вы всегда при наличии соответствующих полномочий сможете просмотреть всю обширную информацию, касающуюся любой базы данных текущего экземпляра сервера, а также данные относительно всех файлов, входящих в состав каждой базы данных.

В этой книге, несмотря на выражение "текущий экземпляр сервера", мы будем иметь в виду ровно один такой экземпляр — автоматически или вручную запущенный на выполнение в вашей системе Database Engine SQL Server, инсталляцию которого вы выполнили в процессе чтения *главы 1*, а может быть и несколько раньше.

В SQL Server существует два вида баз данных — *системные базы данных и пользовательские базы данных*.

Системные базы данных создаются автоматически при инсталляции SQL Server и изменяются, как правило, системой. Хотя некоторые изменения в отдельных системных базах данных может выполнить и пользователь, но это следует делать только в том случае, если пользователь точно понимает, что он делает и к каким результатам это может привести. Профессиональные программисты SQL Server часто после инсталляции системы вносят нужные им изменения в базу данных *model*, которая используется при создании пользовательских баз данных. Возможность и необходимость таких изменений рассматривается в этой главе и в *приложении 4*.

Пользовательские базы данных создаются, изменяются и удаляются пользователем (профессиональным пользователем, т. е. человеком, знающим очень многое о базах данных). В такие пользовательские базы данных вы помещаете все те данные, которые необходимы вам (вашему заказчику) для решения задач конкретной предметной области. Сюда же можно помещать и другие нужные для работы объекты базы данных — хранимые процедуры, пользовательские типы данных, триггеры, функции и др. Именно этими базами данных мы с вами и будем заниматься на протяжении всей этой книги.

Любая база данных в SQL Server состоит, как минимум, из двух файлов — файла (или нескольких файлов) данных и файла (нескольких файлов) журнала транзакций. Файлы данных (это относится только к файлам данных, но не к файлам журнала транзакций) могут объединяться в файловые группы. Каждому файлу базы данных, будь это файл данных или файл журнала транзакций, присваивается как имя, под которым файл известен в операционной системе, так и логическое имя, по которому к этому файлу можно обращаться в среде SQL Server. Пользовательская база данных также имеет ряд характеристик, которые можно просматривать и при необходимости изменять с использованием различных средств, существующих в системе. Наиболее интересные и важные характеристики баз данных и их файлов мы с вами рассмотрим в этой главе, когда будем создавать и изменять базы данных и когда будем отображать сведения об этих базах данных и об их файлах. Полный, или почти полный перечень характеристик баз данных, а также перечень существующих в системе средств изменения и отображения этих характеристик, представлен в *приложении 4*.

К базе данных можно обратиться по ее логическому имени, задаваемому явно при создании базы данных. При обращении к некоторым функциям используется идентификатор базы данных — целочисленное значение, присваиваемое системой в момент создания базы данных. Существуют функции, позволяющие по имени базы данных находить ее идентификатор (функция `DB_ID()`) и наоборот: по идентификатору — имя базы данных (функция `DB_NAME()`). Такие функции и примеры их полезного использования мы с вами рассмотрим чуть позже в этой главе.

3.2.1. Системные базы данных

Системные базы данных содержат сведения, необходимые для работы SQL Server. Системными базами данных являются:

- ◆ master;
- ◆ model;
- ◆ msdb;
- ◆ tempdb.

Существует еще одна скрытая системная база данных `resource`, которая хранит системные объекты, входящие в состав SQL Server. Эта база данных не отображается в списке системных баз данных, никакие сведения о ней естественными средствами получить невозможно. Напрямую к ней обратиться нельзя, однако существуют средства (системные функции и системные представления), позволяющие получить из нее некоторые данные. Об этой базе данных чуть позже.

База данных *master*

База данных `master` является, пожалуй, наиболее важной системной базой данных в SQL Server. Она содержит все данные, необходимые для работы с СУБД. Она также содержит данные о конфигурации сервера базы данных, сведения обо всех пользовательских базах данных, созданных в экземпляре сервера: характеристики баз данных, характеристики и размещение файлов каждой базы данных. Настоятельно рекомендуется выполнять ее резервное копирование при создании, изменении или удалении любой базы данных пользователя. SQL Server не сможет выполниться, если база данных `master` недоступна.

База данных состоит из двух файлов: из файла данных (логическое имя `master`, имя файла базы данных `master.mdf`) и файла журнала транзакций (логическое имя `mastlog`, имя файла `mastlog.ldf`).

База данных *model*

Основным назначением базы данных `model` является хранение шаблонов для всех вновь создаваемых пользователем баз данных. При создании новой пользовательской базы данных в нее из базы данных `model` копируются типы данных. Создаваемым базам данных присваиваются значения по умолчанию многочисленных характеристик, которые также выбираются из базы данных `model`. Если вы добавите новые объекты в базу данных `model`, то эти объекты будут копироваться во все вновь создаваемые пользовательские базы данных текущего экземпляра сервера.

База данных состоит из двух файлов: файла данных (логическое имя `modeldev`, имя файла `model.mdf`) и журнала транзакций (логическое имя `modellog`, имя файла `modellog.ldf`).

База данных *msdb*

В SQL Server существуют средства создания расписаний (`schedule`) для автоматического выполнения заданий, ведения истории их выполнения и для выдачи преду-

преждающих сообщений (alert). Все это хранится в базе данных msdb. Используется в основном компонентом SQL Server Agent. В этой базе данных также хранится история создания резервных копий, SSIS-пакеты, сведения о репликациях. Используется компонентами Service Broker и database mail.

Эта база данных состоит из двух файлов: файла данных (логическое имя MSDBData, имя файла MSDBData.mdf) и журнала транзакций (логическое имя MSDBLog, имя файла MSDBLog.ldf).

База данных tempdb

В системной базе данных tempdb хранятся временные объекты, создаваемые пользователями (в первую очередь это временные таблицы, которые существуют только на время выполнения соответствующей программы пользователя, где они были созданы), это внутренние объекты, создаваемые сервером базы данных при выполнении запросов, а также ряд других объектов. Во многих случаях использование базы данных tempdb позволяет повысить производительность системы при выполнении различных операций с базами данных.

База данных так же, как и все остальные системные базы данных, использует два файла: файл данных (логическое имя tempdev, имя файла tempdb.mdf) и журнал транзакций (логическое имя templog, имя файла templog.ldf).

База данных resource

В SQL Server существует и такой невидимый обычными средствами объект, как скрытая база данных resource. Эта база данных в схеме sys содержит системные объекты SQL Server (системные хранимые процедуры, представления, функции), которые доступны из любой пользовательской и системной базы данных. Использование ресурсной БД облегчает внесение изменений при установке пакетов исправлений и иных обновлений за счет простой замены на новую версию базы данных.

Эта база данных хранится отдельно от всех других системных баз данных. Несмотря на то, что она сильно засекречена, сообщу вам имена ее файлов: файл данных имеет физическое имя mssqlsystemresource.mdf, файл журнала транзакций — физическое имя mssqlsystemresource.ldf.

* * *

Не следует пытаться вручную вносить изменения в системные базы данных. Разве что нужно будет выполнять их резервное копирование после соответствующих изменений и при необходимости, в случаях сбоев системы, осуществлять восстановление скопированных баз данных. Если уж очень хочется, вы можете внести изменения в системную базу данных model, чтобы вновь создаваемые вами базы данных сразу же автоматически получали необходимые значения параметров, наследовали полезные функции и представления. Лично я стараюсь избегать таких действий. Есть стандартное поведение системы, нравится оно мне или нет, но я его использую по назначению, например, создаю базу данных со всеми значениями по умолчанию, заданными в базе данных model, а уже потом вношу нужные мне изменения. Более того, если хочется сразу при создании базы данных задать множество

нестандартных значений для ее характеристик, то для этого можно использовать и диалоговые средства Management Studio.

Дальше вам решать.

3.2.2. Базы данных пользователей

В созданной пользователем базе данных хранится множество объектов. Главным объектом являются, разумеется, таблицы, в которые вы помещаете все данные, необходимые для решения задач конкретной предметной области. Кроме таблиц база данных хранит пользовательские типы данных, триггеры, хранимые процедуры, индексы и др. База данных содержит как сами данные, хранящиеся в таблицах, так и метаданные, описывающие эти данные. Хранение в базе данных и метаданных является важнейшим принципом, применяемым во всех базах данных. Это позволяет уменьшить зависимость программ от структуры базы данных.

Помимо пользовательских объектов в базе данных хранятся системные объекты — системные таблицы, системные представления (их очень большое количество), системные хранимые процедуры (их еще больше), системные функции (многие из них мы рассмотрим в этой книге), системные типы данных (мы подробно рассмотрим *все* существующие в системе типы данных), а также пользователи, роли, схемы.

Для любой базы данных используется не менее двух файлов операционной системы — файл данных (*data*) для хранения собственно данных и файл журнала транзакций (*transaction log*, иногда этот файл называют *протоколом транзакций*). Каждый из этих файлов может принадлежать только одной базе данных.

База данных может содержать не более 32 767 файлов для хранения данных. Первый или единственный файл данных называется *первичным файлом*. И при вновь созданной, так сказать "пустой", базе данных в первичном файле хранятся системные данные, такие как ссылки на другие, вторичные файлы данных и на файлы журнала транзакций. Начальный размер первичного файла не может быть меньше, чем 3 Мбайта.

При желании вы можете использовать *вторичные файлы* для хранения данных. Во вторичных файлах хранятся только пользовательские данные. Файлы данных могут объединяться в файловые группы. В любой базе данных всегда присутствует первичная файловая группа *PRIMARY*. Если не создано никакой вторичной файловой группы, то все файлы данных принадлежат первичной группе. В некоторых случаях имеет смысл объединять отдельные файлы в файловые группы с целью повышения производительности системы. В файловые группы могут объединяться только файлы данных, но не файлы журнала транзакций.

Все файлы данных имеют страничную организацию. Размер страницы в SQL Server имеет значение 8 Кбайт и не может быть изменен.

Журнал транзакций также может быть представлен несколькими файлами. В журнале хранятся все изменения базы данных, выполненные в контексте каждой транзакции. Прежде чем записать выполненные пользователем изменения в файл данных, система вначале осуществляет необходимые записи в журнал транзакций.

Журнал используется для выполнения операций подтверждения (`COMMIT`) или отката (`ROLLBACK`) транзакций, а также для целей восстановления базы данных на любой заданный момент времени или в случае ее разрушения. Подробнее о транзакциях см. в главе 10. Размер файла журнала транзакций не может быть задан менее чем 512 Кбайт.

Для одного экземпляра сервера базы данных может существовать до 32 767 баз данных. Каждая база данных может содержать не более 32 767 файлов и не более 32 767 файловых групп. Вряд ли вам когда-либо потребуется такое количество баз данных в одном экземпляре сервера и такое количество файлов в базе данных.

3.2.3. Некоторые характеристики базы данных

Каждая база данных имеет множество характеристик. Характеристики базы данных, их значения по умолчанию и средства, используемые для изменения текущих значений, описаны в *приложении 4*.

Рассмотрим некоторые из этих характеристик.

3.2.3.1. Владелец базы данных (Owner)

Владелец базы данных (`owner`) имеет все полномочия к базе данных. Он может изменять характеристики базы, удалять ее, вносить любые изменения в данные и метаданные. Владельцем базы становится пользователь, создавший базу данных.

Владельца пользовательской базы данных можно изменить, используя в языке Transact-SQL системную процедуру `sp_changedbowner`. Вот несколько упрощенный синтаксис обращения к этой процедуре:

```
EXECUTE sp_changedbowner '<имя нового владельца>'
```

Функция возвращает значение 0 при успешной смене владельца или 1 в случае возникновения ошибки. Например, для изменения владельца текущей базы данных можно выполнить следующее обращение к этой процедуре:

```
EXECUTE sp_changedbowner 'anotherowner'
```

Имя нового владельца уже должно быть описано в системе. Чтобы получить список существующих в системе пользователей, можно выполнить системную хранимую процедуру `sp_helplogins`:

```
EXEC sp_helplogins;
```

3.2.3.2. Порядок сортировки (collation)

Порядок сортировки (`collation`) для базы данных определяет допустимый набор символов в строковых типах `CHAR`, `VARCHAR` и правила, по которым будут при необходимости упорядочиваться эти строковые данные. Порядок сортировки определяет, будет ли сортировка происходить по внутреннему коду или в лексикографическом (алфавитном) порядке, в каком порядке будут размещаться строчные и прописные буквы, как распределяются знаки препинания, иные специальные сим-

волы и др. Если для строкового типа данных явно не указан порядок сортировки, то ему будет присвоен порядок, заданный по умолчанию для всей базы данных. Для элемента данных при его описании в базе можно указать любой допустимый порядок сортировки, отличный от порядка сортировки базы данных.

3.2.3.3. Возможность изменения данных базы данных

База данных может находиться в состоянии только для чтения (`READ_ONLY`) или доступна как для чтения, так и для внесения изменений в данные (`READ_WRITE`).

3.2.3.4. Состояние базы данных (Database State)

В каждый момент времени любая база находится в одном конкретном состоянии (*state*). В SQL Server существуют следующие состояния базы данных.

- ◆ **ONLINE.** База данных в доступном состоянии (или, в другой терминологии, находится в *оперативном режиме*). С ней можно выполнять любые действия по изменению данных и метаданных. В этом состоянии средствами операционной системы невозможно удалить или даже скопировать файлы базы данных на другие устройства (при запущенном на выполнение сервере базы данных).
- ◆ **OFFLINE.** База данных в недоступном состоянии (или еще говорят, что она находится в *автономном режиме*). Никакие действия с объектами базы данных в этом состоянии невозможны. Однако средствами операционной системы можно удалить файлы базы данных, чего делать все-таки не стоит, или скопировать их на другой носитель.
- ◆ **RESTORING.** База данных недоступна. В это состояние она переводится, когда выполняется восстановление файлов данных из резервной копии.
- ◆ **RECOVERING.** База данных недоступна, она находится в процессе восстановления. После завершения восстановления база автоматически будет переведена в оперативное состояние (`ONLINE`).
- ◆ **RECOVERY_PENDING.** Это состояние ожидания исправления ошибок восстановления базы данных. База данных недоступна. В процессе восстановления базы произошла ошибка, которая требует вмешательства пользователя. После исправления ошибки пользователь сам должен перевести базу в оперативное состояние.
- ◆ **SUSPECT.** База данных недоступна. Она помечена как подозрительная и может быть поврежденной. Со стороны пользователя требуются действия по устранению ошибок.
- ◆ **EMERGENCY.** База данных повреждена и находится в состоянии только для чтения (`READ_ONLY`). Такое состояние базы используется для ее диагностики и при попытках скопировать неповрежденные данные.

Существует еще множество других менее важных для обычной работы характеристик базы данных, присутствующих в указанных категориях. Некоторые из них мы рассмотрим далее в этой главе.

Здесь мы рассмотрим, какими способами можно отображать и изменять некоторые характеристики базы данных.

3.2.4. Некоторые характеристики файлов базы данных

Каждый файл базы данных (как файл данных, так и файл журнала транзакций) имеет свой набор характеристик.

3.2.4.1. Основные характеристики файлов базы данных

Каждый файл базы данных является либо файлом данных (rows data, строки данных), либо файлом журнала транзакций (log).

У каждого файла помимо имени, известного в операционной системе, существует и логическое имя (logical name), по которому к файлу можно обращаться в операторах Transact-SQL и при использовании различных компонентов SQL Server.

При создании или при изменении характеристик файла ему можно задать начальный размер (initial size). Для файла устанавливается также возможность автоматического увеличения размера и величины приращения — в процентах от начального размера или в абсолютных значениях единиц памяти (в килобайтах, мегабайтах, гигабайтах или терабайтах). Максимальный размер памяти, используемой для хранения файла, можно ограничить конкретной величиной или указать, что размер файла не ограничивается (unlimited).

3.2.4.2. Состояния файлов базы данных

Файлы данных базы данных также могут находиться в различных состояниях, причем состояние файла базы данных поддерживается независимо от состояния самой базы данных.

Файл базы данных может находиться в одном из следующих состояний.

- ◆ **ONLINE.** Файл в доступном состоянии (в *оперативном режиме*). С данными, содержащимися в этом файле, можно выполнять любые действия.
- ◆ **OFFLINE.** Файл в недоступном состоянии (в *автономном режиме*). Перевод файла в автономный и оперативный режим осуществляется пользователем. Обычно файл переводят в состояние OFFLINE, если он поврежден и требует восстановления. После восстановления поврежденного файла пользователь должен явно перевести его в состояние ONLINE.
- ◆ **RESTORING.** Файл находится в процессе восстановления. Другие действия с данными, хранящимися в этом файле, невозможны. После завершения восстановления файл автоматически переводится системой в состояние ONLINE.
- ◆ **RECOVERY_PENDING.** Файл автоматически переводится в это состояние, если при его восстановлении произошла ошибка. Восстановление файла было отложено. Требуется соответствующее действие от пользователя для устранения ошибки. После наведения порядка с файлом пользователь вручную переводит его в оперативное состояние.
- ◆ **SUSPECT.** В процессе оперативного восстановления файла произошла ошибка. База данных также помечается как подозрительная.
- ◆ **DEFUNCT.** Файл был удален (разумеется, когда он не был в состоянии ONLINE).

Если при работе с базой данных отдельные файлы являются недоступными, то все же возможны различные операции с базой данных, если для выполнения этих операций требуются только данные, содержащиеся в файлах, которые находятся в оперативном режиме. Это одно из неоспоримых преимуществ системы MS SQL Server.

ЗАМЕЧАНИЕ

Наверняка при первом знакомстве с SQL Server смысл многих вещей, таких как некоторые состояния базы данных и ее файлов, а также понимание того, что именно в различных ситуациях должен сделать пользователь, часто остается загадкой. Не расстраивайтесь. В процессе приобретения опыта использования системы все станет на свои места, и вы во всем разберетесь. И я вместе с вами, надеюсь, тоже.

3.3. Получение сведений о базах данных и их файлах в текущем экземпляре сервера

Получить сведения о базах данных (как системных, так и пользовательских) и об их файлах можно при использовании множества разнообразных средств, входящих в состав SQL Server, которыми являются:

- ◆ системные представления;
- ◆ системные хранимые процедуры;
- ◆ системные функции;
- ◆ диалоговые средства компонента Management Studio.

3.3.1. Системное представление *sys.databases*

Для того чтобы просмотреть список и характеристики всех баз данных, существующих в текущем экземпляре сервера, как пользовательских, так и системных, мы можем обратиться, пожалуй, к самому важному и информативному системному представлению просмотра каталогов *sys.databases*.

Это представление отображает значения множества характеристик баз данных текущего экземпляра сервера. Представление возвращает одну строку для каждой базы данных. Рассмотрим только некоторые из столбцов, получаемых из этого представления, которые нам будут в первую очередь интересны.

- ◆ *name* — содержит логическое имя базы данных.
- ◆ *database_id* — идентификатор базы данных (целое число), автоматически присваиваемый базе данных системой при ее создании.
- ◆ *create_date* — дата и время создания базы данных. Время указывается с точностью до миллисекунд.
- ◆ *collation_name* — имя порядка сортировки по умолчанию для базы данных.
- ◆ *is_read_only* — определяет, является ли база данных базой только для чтения:
 - 0 — база данных находится в режиме только для чтения (*READ_ONLY*);
 - 1 — база данных в режиме чтения и записи (*READ_WRITE*).

◆ state — состояние базы данных:

- 0 — ONLINE;
- 1 — RESTORING;
- 2 — RECOVERING;
- 3 — RECOVERY_PENDING;
- 4 — SUSPECT;
- 5 — EMERGENCY;
- 6 — OFFLINE.

◆ state_desc — текстовое описание состояния базы данных: ONLINE, RESTORING, RECOVERING, RECOVERY_PENDING, SUSPECT, EMERGENCY, OFFLINE. Эта характеристика является, разумеется, производной от состояния базы данных (`state`).

3.3.2. Системное представление `sys.master_files`

Другое полезное системное представление — `sys.master_files`. Оно позволяет получить детальный список всех баз данных и файлов, входящих в состав каждой базы данных, а также многие интересные характеристики этих файлов.

Наиболее важными для нас столбцами этого представления будут следующие.

- ◆ `database_id` — идентификатор базы данных. Имеет то же значение, что и в представлении `sys.databases`.
- ◆ `file_id` — идентификатор файла (тоже целое число). Первичный файл имеет идентификатор 1.
- ◆ `type` — задает тип файла:
 - 0 — файл данных (`ROWS`);
 - 1 — журнал транзакций (`LOG`);
 - 2 — файловый поток (`FILESTREAM`).
- ◆ `type_desc` — текстовое описание типа файла из столбца `type`: `ROWS` — файл данных, `LOG` — файл журнала транзакций, `FILESTREAM` — файловый поток.
- ◆ `data_space_id` — идентификатор пространства данных (файловой группы), которому принадлежит файл данных. Для всех файлов журнала транзакций идентификатор имеет значение 0.
- ◆ `name` — логическое имя файла, заданное в операторе `CREATE DATABASE` или присвоенное системой по умолчанию, если пользователь не указал логическое имя.
- ◆ `physical_name` — путь к файлу, включая имя дискового устройства, и имя файла в операционной системе.
- ◆ `state` — состояние файла:
 - 0 — ONLINE;
 - 1 — RESTORING;

- 2 — RECOVERING;
 - 3 — RECOVERY_PENDING;
 - 4 — SUSPECT;
 - 6 — OFFLINE;
 - 7 — DEFUNCT.
- ◆ state_desc — текстовое описание состояния файла, производное от состояния файла (state): ONLINE, RESTORING, RECOVERING, RECOVERY_PENDING, SUSPECT, OFFLINE, DEFUNCT.
- ◆ is_read_only — определяет, является ли файл файлом только для чтения:
- 1 — файл READ_ONLY, только для чтения;
 - 0 — файл READ_WRITE, возможны операции чтения, добавления, изменения и удаления данных в этом файле.
- ◆ size — размер файла в страницах. Напомню, что страница в файле данных имеет размер 8 Кбайт или, иными словами, 8192 байта.
- ◆ max_size — указывает три возможных варианта: (1) возможность увеличения размера файла, (2) максимальный размер файла в страницах или (3) неограниченность размера файла. Может принимать следующие значения:
- 0 — увеличение размера файла недопустимо;
 - -1 — файл растет неограниченно, пока он не исчерпает объема всего дискового пространства или пока не достигнет допустимого предела (2 Тбайт для журнала транзакций или 16 Тбайт для файла данных);
 - положительное число — указывает максимальный размер файла в страницах. Число 268435456 может быть указано только для файла журнала транзакций. Означает, что файл может расти до максимального размера в 2 Тбайт.
- ◆ is_percent_growth — указывает, задается ли увеличение размера файла в процентах или в страницах:
- 0 — увеличение размера указано в страницах;
 - 1 — увеличение размера файла указано в процентах от начального размера.
- ◆ growth — указывает, будет ли увеличиваться размер файла:
- 0 — файл не будет увеличиваться в размерах. Указанный размер не может изменяться;
 - 1 и более — размер файла будет при необходимости увеличиваться автоматически в соответствии с заданными параметрами при создании базы данных.

3.3.3. Системное представление `sys.database_files`

Представление `sys.database_files` позволяет получить список файлов и их характеристик только одной текущей базы данных, указанной в операторе `USE`.

Вот некоторые характеристики, которые можно получить при вызове этого представления. Они в точности дублируют значения столбцов представления sys.master_files. Вкратце повторим эти значения.

- ◆ `file_id` — идентификатор файла.
- ◆ `type` — тип файла: 0 — файл данных (`ROWS`), 1 — журнал транзакций (`LOG`), 2 — файловый поток (`FILESTREAM`).
- ◆ `type_desc` — текстовое описание типа файла: `ROWS` — файл данных, `LOG` — файл журнала транзакций, `FILESTREAM` — файловый поток.
- ◆ `data_space_id` — идентификатор файловой группы, которой принадлежит файл.
- ◆ `name` — логическое имя файла, явно заданное в операторе `CREATE DATABASE` или присвоенное системой по умолчанию.
- ◆ `physical_name` — путь к файлу, включая имя дискового устройства, каталоги и имя файла в операционной системе.
- ◆ `state` — состояние файла:
 - 0 — `ONLINE`;
 - 1 — `RESTORING`;
 - 2 — `RECOVERING`;
 - 3 — `RECOVERY_PENDING`;
 - 4 — `SUSPECT`;
 - 6 — `OFFLINE`.
 - 7 — `DEFUNCT`.
- ◆ `state_desc` — текстовое описание состояния файла, производное от состояния файла (`state`): `ONLINE`, `RESTORING`, `RECOVERING`, `RECOVERY_PENDING`, `SUSPECT`, `OFFLINE`, `DEFUNCT`.
- ◆ `is_read_only` — определяет, является ли файл файлом только для чтения:
 - 1 — файл `READ_ONLY`, только для чтения;
 - 0 — файл `READ_WRITE`, возможны операции чтения, удаления и обновления данных в этом файле.
- ◆ `size` — размер файла в страницах.
- ◆ `max_size` — указывает возможность увеличения размера файла, максимальный размер файла в страницах или неограниченность размера файла. Может принимать следующие значения:
 - 0 — увеличение размера файла недопустимо;
 - -1 — файл растет неограниченно, пока не исчерпает всего дискового пространства или не достигнет допустимого предела (2 Тбайта для журнала транзакций или 16 Тбайт для файла данных);
 - положительное число — указывает максимальный размер файла в страницах.

- ◆ `is_percent_growth` — указывает, задается ли увеличение размера файла в процентах или в страницах:
 - 0 — увеличение размера указано в страницах;
 - 1 — увеличение размера файла указано в процентах от начального размера.
- ◆ `growth` — указывает, будет ли увеличиваться размер файла:
 - 0 — файл не будет увеличиваться в размерах. Указанный размер не может изменяться;
 - 1 и более — размер файла будет при необходимости увеличиваться автоматически в соответствии с заданными параметрами при создании базы данных.

3.3.4. Системное представление `sys.filegroups`

Системное представление `sys.filegroups` позволяет получить некоторые данные о файловых группах текущей базы данных, указанной в операторе `USE`. Вот некоторые столбцы этого представления, которые могут быть нам интересны.

- ◆ `name` — название файловой группы. Первой всегда будет первичная файловая группа `PRIMARY`.
- ◆ `type` — тип. Для файловых групп имеет значение `FG`.
- ◆ `type_desc` — текстовое описание типа. Для файловой группы имеет значение `ROWS_FILEGROUP`.
- ◆ `is_read_only` — указывает, является ли файловая группа группой только для чтения:
 - 0 — файловая группа доступна для чтения и записи, т. е. все файлы, входящие в состав этой файловой группы, доступны для чтения и записи;
 - 1 — файловая группа только для чтения.

* * *

Это, пожалуй, наиболее важные и полезные представления, которые вы будете использовать в вашей повседневной жизни. Давайте теперь очень кратко рассмотрим и некоторые другие средства, которые вам могут оказаться полезными при работе с базами данных в SQL Server.

3.3.5. Другие средства получения сведений об объектах базы данных

Мы рассмотрели некоторые средства, позволяющие получить детальные сведения о характеристиках существующих баз данных, их файлах и файловых группах. Имеет смысл сказать несколько слов и о других средствах, которые мы с вами будем использовать в ближайшем будущем.

3.3.5.1. Системные представления

Представления для получения сведений об объектах базы данных.

- ◆ sys.schemas — возвращает сведения о схемах базы данных. Каждая база данных может содержать более двух миллиардов схем. Надо сказать, что "схема" (schema) в SQL Server, это совсем не то, что схема в некоторых других системах управления базами данных. О схемах мы поговорим в разд. 3.8.
- ◆ sys.database_permissions — возвращает сведения о полномочиях в базе данных. Вопросы безопасности в SQL Server решаются довольно жестко и очень разумно. Специалистам по системам безопасности, в том числе и в базах данных, следует этим делам уделить достаточно серьезное внимание.
- ◆ sys.database_principals — возвращает сведения о принципалах (владельцах или, иными словами, участниках доступа к базам данных и их объектам) в базе данных. Это опять же связано с вопросами безопасности в базах данных.
- ◆ sys.database_role_members — возвращает сведения о членах (участниках) ролей в базе данных. И роли базы данных при правильном их использовании могут быть хорошим средством для повышения безопасности баз данных.

Другими системными представлениями просмотра каталогов для объектов, скажем так, "детального уровня", являются следующие далее представления.

- ◆ sys.tables — возвращает сведения о таблицах базы данных. Здесь даются сведения обо всех таблицах текущей базы данных.
- ◆ sys.views — возвращает сведения о представлениях в базе данных. Каждая база данных может содержать множество различных представлений. Представления позволяют упростить задачу пользователя по получению данных из одной или нескольких базовых таблиц. Представления позволяют "скрыть" от не очень профессионального (или слишком ленивого) пользователя все сложности, связанные с составлением запроса к данным базы данных для получения необходимых результатов.
- ◆ sys.indexes — возвращает сведения об индексах базы данных. *Индекс* — это замечательный объект реляционной базы данных, который в одно и то же время может резко ухудшить работу с данными в базе данных, а может при правильном проектировании и сильно повысить производительность при выборке и упорядочении данных базы. Однако добавление к таблицам новых индексов никак не может улучшить временные характеристики при выполнении операций добавления или изменения данных, если в процесс изменения включены столбцы, входящие в состав индекса.
- ◆ sys.events — возвращает сведения о событиях базы данных. *Событие* — это очень интересное и полезное средство при работе с базами данных. События позволяют синхронизировать работу программ, одновременно использующих одну и ту же базу, и иногда избежать некоторых ошибок в работе пользователей с базой данных.

- ◆ `sys.types` — возвращает сведения о системных и пользовательских типах данных.
- ◆ `sys.columns` — возвращает сведения о столбцах таблиц и представлений.

Для обращения к системным представлениям, как и к другим представлениям в базе данных, мы используем оператор `SELECT`, синтаксис которого и варианты применения будем подробно рассматривать в этой главе и в последующих главах.

3.3.5.2. Системные хранимые процедуры

Для обращения к хранимым процедурам, как системным, так и к пользовательским, используется оператор `EXECUTE`. Примеры таких обращений мы вскоре рассмотрим.

Часто используются следующие хранимые процедуры.

- ◆ `sp_databases` — предоставляет список баз данных.
- ◆ `sp_stored_procedures` — возвращает список хранимых процедур.
- ◆ `sp_help` — возвращает список различных объектов базы данных, типов данных, определенных пользователем или поддерживаемых системой SQL Server.
- ◆ `sp_helplogins` — возвращает список регистрационных имен пользователей (`login`).
- ◆ `sp_helptext` — дает возможность получить на языке Transact-SQL тексты, описывающие системные хранимые процедуры, триггеры, вычисляемые столбцы, ограничения `CHECK` для столбцов таблиц.
- ◆ `sp_changedbowner` — позволяет изменить владельца базы данных.
- ◆ `sp_configure` — позволяет изменить некоторые режимы системы.

3.3.5.3. Системные функции

Помимо системных представлений просмотра каталогов и системных хранимых процедур в SQL Server присутствуют системные функции.

Чтобы просмотреть логические имена файлов любой базы данных, системной или пользовательской, можно вызвать системную функцию `FILE_NAME()`.

Для получения идентификатора базы данных, который присваивается ей при ее создании, используется функция `DB_ID()`. Очень удобная и довольно часто применяемая функция. Ее регулярно будем использовать как в этой главе, так и в реальной жизни.

Имя базы данных можно получить при вызове функции `DB_NAME()`, которой передается в качестве параметра идентификатор базы данных.

Функция `FILE_ID()` для текущей базы данных возвращает идентификатор файла базы данных по указанному логическому имени этого файла.

Функция `FILEGROUP_ID()` возвращает идентификатор файловой группы, заданной ее именем.

Функция `FILEGROUP_NAME()`, наоборот, по идентификатору файловой группы возвращает ее имя.

* * *

В нужное время мы обратимся к этим системным представлениям, хранимым процедурам и функциям и станем использовать их по прямому назначению для получения необходимых сведений об объектах и характеристиках базы данных.

3.4. Создание и удаление базы данных

В этом разделе мы рассмотрим всевозможные способы создания пользовательских баз данных при применении как оператора `CREATE DATABASE`, так и диалоговых средств Management Studio. Кроме создания баз данных мы научимся их удалять при помощи простого оператора `DROP DATABASE`, а также при использовании диалоговых средств Management Studio.

Для отображения характеристик и состояния существующей базы данных средствами Transact-SQL можно использовать описанные системные представления и диалоговые средства, существующие в Management Studio. Мы рассмотрим все подходящие варианты.

3.4.1. Использование операторов Transact-SQL для создания, отображения и удаления баз данных

3.4.1.1. Оператор создания базы данных

Для создания новой базы данных используется оператор `CREATE DATABASE`. Его синтаксис (только для целей создания новой базы данных) показан в листинге 3.1 и на графе 3.1. Детализация синтаксических конструкций представлена в графах 3.2—3.7. Основные синтаксические конструкции, как мы ранее договорились, будем представлять и в нотациях Бэкуса — Наура, и при помощи R-графов.

Листинг 3.1. Синтаксис оператора `CREATE DATABASE`. Вариант создания новой базы данных

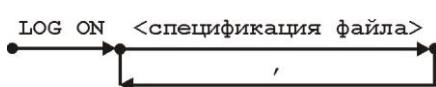
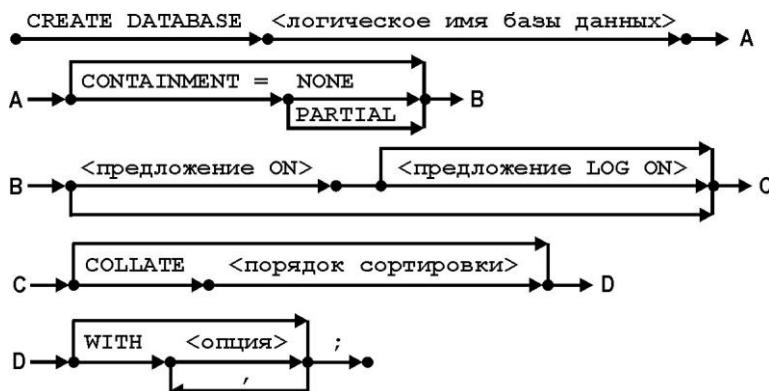
```
CREATE DATABASE <логическое имя базы данных>
[ CONTAINMENT = { NONE | PARTIAL } ]
[ <предложение ON> [ <предложение LOG ON> ] ]
[ COLLATE <порядок сортировки> ]
[ WITH <опция> [, <опция>]... ];
<предложение ON> ::==
ON [ PRIMARY ] <спецификация файла> [, <спецификация файла>]...
[, <файловая группа> [, <файловая группа>] ...]
<предложение LOG ON> ::=
LOG ON <спецификация файла> [, <спецификация файла>]...
```

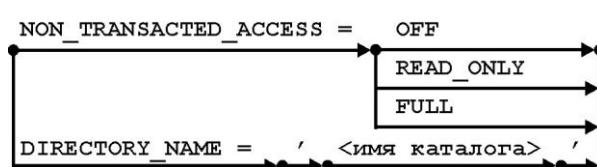
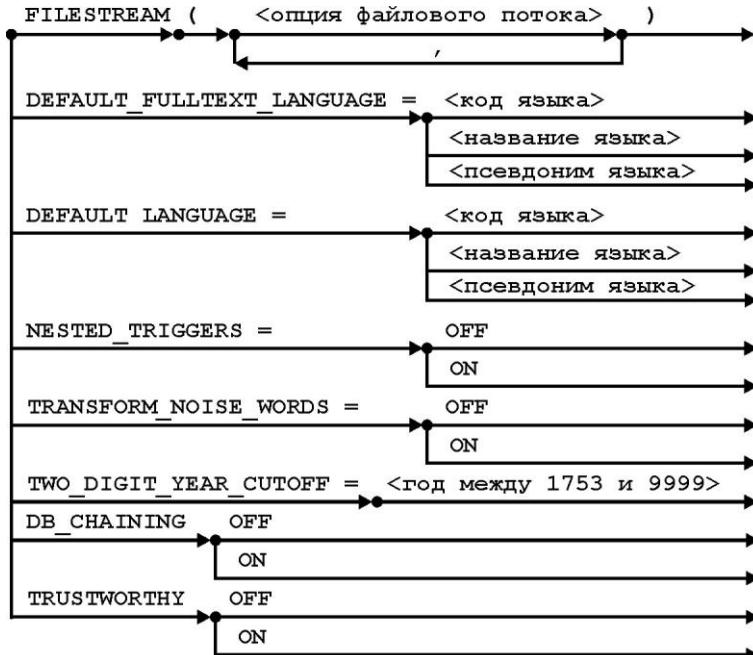
```

<опция> ::==
{ FILESTREAM (<опция файлового потока> [, <опция файлового потока> ]...)
| DEFAULT_FULLTEXT_LANGUAGE =
{ <код языка> | <название языка> | <псевдоним языка> }
| DEFAULT_LANGUAGE =
{ <код языка> | <название языка> | <псевдоним языка> }
| NESTED_TRIGGERS = { OFF | ON }
| TRANSFORM_NOISE_WORDS = { OFF | ON }
| TWO_DIGIT_YEAR_CUTOFF = <год между 1753 и 9999>
| DB_CHAINING { OFF | ON }
| TRUSTWORTHY { OFF | ON }
}

<опция файлового потока> ::=
{ NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }
| DIRECTORY_NAME = '<имя каталога>'
}

```





ЗАМЕЧАНИЕ

Оператор CREATE DATABASE с несколько измененным синтаксисом может также использоваться и для некоторых иных целей: присоединения где-то кем-то когда-то созданной базы данных, возможно, на другом компьютере к списку баз данных текущего экземпляра сервера, а также для создания так называемого мгновенного снимка базы данных (SNAPSHOT). Все это мы рассмотрим чуть позже в данной главе.

В результате успешного создания новой базы данных будут созданы все специфицированные в операторе или сформированные по умолчанию с соответствующими характеристиками файлы базы данных. Самой базе данных будут присвоены заданные явно или по умолчанию значения ее характеристик. Файлы данных могут быть объединены в файловые группы.

Как видно из синтаксиса, при создании базы данных обязательно нужно задать только лишь логическое имя базы данных, которое будет известно в текущем экземпляре сервера. Все остальные конструкции являются необязательными. В этом случае самой базе данных и файлам создаваемой базы данных будут присвоены

значения всех характеристик по умолчанию, которые мы с вами и рассмотрим в ближайшее время.

ЗАМЕЧАНИЕ ПО СИНТАКСИСУ

Во множестве предыдущих версий документации по SQL Server сообщалось, что начиная с данной версии все операторы Transact-SQL должны завершаться символом точки с запятой (;). Это не очень соответствовало реальной действительности. И тогда операторы можно было не завершать этим символом. Лучшим вариантом будет, если вы всегда станете заканчивать любой оператор Transact-SQL символом точки с запятой.

Логическое имя базы данных

Логическое имя базы данных является обязательным параметром. Оно идентифицирует создаваемую базу данных. Имя должно соответствовать правилам задания идентификаторов, обычных или с разделителями (см. главу 2). В обычном идентификаторе здесь также можно использовать и буквы кириллицы, если при инсталляции сервера вы задали порядок сортировки Cyrillic_General_CI_AS. В идентификаторах с разделителями, как вы помните, можно использовать *любые* символы. Такой идентификатор только нужно заключить в квадратные скобки. Имя не должно содержать более 128 символов. Имя должно быть уникальным среди имен всех баз данных текущего экземпляра SQL Server. По этому имени вы обращаетесь к конкретной базе данных экземпляра сервера во всех операторах Transact-SQL, где требуется указать базу данных. Это единственный обязательный параметр в данном операторе.

Предложение CONTAINMENT

```
[ CONTAINMENT = { NONE | PARTIAL } ]
```

Необязательное предложение `CONTAINMENT` определяет степень независимости базы данных от характеристик экземпляра сервера базы данных, в котором создается база данных. Если указано `NONE` (значение по умолчанию), то создается обычная (неавтономная) база данных.

Если же указано значение `PARTIAL`, то создается так называемая "partially contained" база данных, т. е. частично автономная. Пользователь может подключаться к такой базе данных, используя отдельные средства аутентификации, не связанные с уровнем экземпляра сервера. Такую базу данных проще перенести в другой экземпляр сервера базы данных, на другой компьютер. Кроме того, в таких базах данных можно избежать неприятностей, которые иногда возникают при создании временных таблиц, где присутствуют строковые столбцы с порядком сортировки, отличным от принятого по умолчанию.

Подобные базы данных мы позже рассмотрим в примерах.

Предложение ON

В операторе создания базы данных может задаваться первичный файл данных в предложении `ON`. Файл, описанный первым в предложении, а также перечисленные

вслед за ним файлы, если они указаны, помещаются в файловую группу PRIMARY. О вторичных файловых группах см. чуть дальше.

Все описания файлов заключаются в круглые скобки (см. далее синтаксис спецификации файла в разд. "Спецификация файла") и отделяются друг от друга запятыми.

База данных помимо обязательной первичной файловой группы PRIMARY может содержать большое количество других, вторичных, файловых групп. Вторичные файловые группы вместе с принадлежащими им файлами данных описываются после первичной файловой группы.

Предложение LOG ON

Необязательное предложение LOG ON описывает файл (файлы) журнала транзакций. В базе данных может существовать большое количество файлов журналов транзакций. Предложение LOG ON можно опустить. В этом случае файлу журнала транзакций присваиваются значения по умолчанию.

Задание нескольких файлов журналов транзакций имеет смысл только в том случае, если для одного файла не хватает места на внешнем носителе. Тогда администратор БД создает файл (файлы) на другом носителе (других носителях).

ЗАМЕЧАНИЕ

Как видно из приведенного синтаксиса оператора CREATE DATABASE, при создании базы данных можно вообще не задавать никаких файлов данных и файлов журнала транзакций (эти конструкции заключены в описании синтаксиса в обрамляющие квадратные скобки). Если приглядеться внимательно к этому описанию, то можно увидеть, что допустим вариант задания файла (файлов) данных при отсутствии задания файлов журнала транзакций. Однако указать файл (файлы) журнала транзакций при отсутствии описания файла данных нельзя. В таком случае вы получите ошибку системы.

Если при создании новой базы данных вы не укажете файл данных или файл журнала транзакций, то система всем характеристикам этих файлов присвоит значения по умолчанию (см. далее).

Предложение COLLATE

Необязательное предложение COLLATE позволяет задать для создаваемой базы данных порядок сортировки, отличный от того, который был установлен при инсталляции системы для экземпляра сервера. Если предложение не указано, то база данных будет иметь порядок сортировки по умолчанию, заданный при инсталляции системы. Мы с вами при инсталляции SQL Server установили Cyrillic_General_CI_AS.

Предложение WITH

Необязательное предложение WITH позволяет описать некоторые дополнительные характеристики создаваемой базы данных.

```
FILESTREAM <опция файлового потока> [, <опция файлового потока> ]...  
<опция файлового потока> ::=  
{ NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }  
| DIRECTORY_NAME = '<имя каталога>' }
```

После ключевого слова FILESTREAM в скобках перечисляются опции файлового потока.

Опция NON_TRANSACTED_ACCESS определяет возможность доступа к данным файлового потока вне контекста транзакций. Транзакции мы рассмотрим в главе 10. Значениями опции являются:

- ◆ OFF — доступ к данным вне транзакции недопустим;
- ◆ READ_ONLY — к данным файлового потока возможен доступ вне транзакций только для операций чтения;
- ◆ FULL — допустимы все операции к данным файлового потока вне транзакций.

Опция DIRECTORY_NAME задает имя каталога. Это имя должно быть уникальным среди имен каталогов, заданных параметром DIRECTORY_NAME текущего экземпляра сервера. Каталог с этим именем будет создан внутри сетевого каталога экземпляра сервера базы данных. Используется для файловых таблиц (FileTable), которые мы рассмотрим в главе 5.

Опция DEFAULT_FULLTEXT_LANGUAGE (допустимо только для автономной базы данных) задает язык базы данных по умолчанию для полнотекстового поиска в индексированных столбцах. Язык может задаваться в виде кода языка, его названия или псевдонима. Список допустимых языков см. в приложении 5.

Опция DEFAULT_LANGUAGE (допустимо только для автономной базы данных) задает язык по умолчанию для вновь создаваемых регистрационных имен пользователей. Может задаваться в виде кода языка, его названия или псевдонима.

Опция NESTED_TRIGGERS (допустимо только для автономной базы данных) задает возможность использования вложенных триггеров AFTER. Если указано OFF, вложенные триггеры недопустимы. При задании ON может существовать до 32 уровней триггеров. То есть триггер может вызывать (разумеется, неявно) другой триггер, который, в свою очередь, инициирует обращение к триггеру следующего уровня. И так до 32 уровней.

Опция TRANSFORM_NOISE_WORDS (допустимо только для автономной базы данных) задает поведение сервера базы данных в некоторых ситуациях полнотекстового поиска в базе данных. Существует понятие *noise word* или *stop word*. Это слова, которые слишком часто присутствуют в текстах и не имеют особого смысла при выполнении поисковых действий. Чаще всего это предлоги, для некоторых языков артикли, а также множество других часто встречающихся в языке слов.

Значение OFF (по умолчанию) приводит к тому, что если в запросе встречаются такие слова и запрос возвращает нулевое количество строк, то просто выдается предупреждающее сообщение.

Если указано ON, то система выполнит преобразование запроса, удалив из него соответствующие слова.

Опция TWO_DIGIT_YEAR_CUTOFF (допустимо только для автономной базы данных) задает значение года в диапазоне между 1753 и 9999. По умолчанию принимается 2049. Это число используется для интерпретации года, заданного двумя символами.

Если двухсимвольный год меньше или равен последним двум цифрам указанного четырехсимвольного значения, то этот год будет интерпретироваться как год того же столетия. Если больше — то будет использовано столетие, следующее за указанным в операторе столетием. Хорошей практикой является указание во *всех* датах четырехсимвольного значения года.

Опция `DB_CHAINING` определяет, может ли создаваемая база данных использоваться в цепочках связей между несколькими базами данных. `OFF` (по умолчанию) запрещает такое использование, `ON` — разрешает.

Опция `TRUSTWORTHY` задает, могут ли программные компоненты базы данных (хранимые процедуры, представления, созданные пользователем функции) обращаться к ресурсам вне базы данных. `OFF` (по умолчанию) запрещает обращение к внешним ресурсам, `ON` — разрешает.

Спецификация файла

Синтаксис спецификации файла, одинаковый как для файлов данных, так и для журналов транзакций, показан в листинге 3.2 и в соответствующем R-графе (граф 3.6).

Листинг 3.2. Синтаксис спецификации файла

```
<спецификация файла> ::=  
( NAME = <логическое имя файла>,  
  FILENAME = { '<путь к файлу>' | '<путь к файловому потоку>' }  
  [, SIZE = <целое1> [ KB | MB | GB | TB ] ]  
  [, MAXSIZE = { <целое2> [ KB | MB | GB | TB ] | UNLIMITED } ]  
  [, FILEGROWTH = <целое3> [ KB | MB | GB | TB | % ] ]  
)
```



Граф 3.6. Синтаксис спецификации файла

В спецификации файла обязательными являются только предложения `NAME` (логическое имя файла) и `FILENAME` (имя файла в операционной системе). В случае отсутствия любого из предложений `SIZE`, `MAXSIZE` или `FILEGROWTH` соответствующим харак-

теристикам будут присвоены значения по умолчанию, которые мы рассмотрим далее. Эти значения по умолчанию различны для файлов данных и для файлов журнала транзакций.

В R-графе указаны элементы "единица измерения" и "единица измерения2". Второй элемент отличается от первого тем, что в его списке присутствует и знак процента.

Предложение **NAME**

NAME = <логическое имя файла>

Предложение **NAME** задает логическое имя файла. Это имя может использоваться при различных ссылках на данный файл. Оно должно быть уникальным только в этой базе данных.

Если ни один *файл данных* явно при создании базы данных не описывается (отсутствует предложение **ON**), то логическому имени единственного файла данных присваивается имя самой базы данных. Например, если создаваемая база данных имеет имя *Strange*, то и логическое имя файла данных по умолчанию будет *Strange*.

Если не задается ни одного *файла журнала транзакций* (не указано предложение **LOG ON**), то логическому имени единственного файла журнала транзакций присваивается имя, состоящее из имени базы данных, к которому добавляется суффикс *_log*. Для той же базы данных *Strange* при отсутствии явного задания файла журнала транзакций логическое имя этого файла будет *Strange_log*.

Предложение **FILENAME**

FILENAME = { '<путь к файлу>' | '<путь к файловому потоку>' }

Предложение **FILENAME** определяет полный путь к файлу (включая имя внешнего носителя), а также имя самого создаваемого файла. Путь (все каталоги в пути) должен существовать на указанном внешнем носителе, а сам файл должен отсутствовать. Для первичного файла данных принято использовать расширение *mdf*, для вторичных файлов данных — расширение *ndf*, а для журнала транзакций — *ldf*. Такие значения расширений не являются обязательными, однако следование этому правилу опять же повышает читаемость скриптов и удобство в работе с системой. Файл (файлы) журнала транзакций следует размещать на устройствах, отличных от тех, на которых размещаются файлы данных. В случае любых сбоев дисковых носителей это позволит выполнить восстановление базы данных при наличии резервной копии. Кроме того, размещение файлов журнала транзакций на *физических носителях*, отличных от физических носителей для хранения файлов данных, повышает производительность системы, уменьшая количество перемещений головок диска при операциях чтения-записи данных. Однако в случае установок значений по умолчанию эта рекомендация не выполняется. Для достаточно простых баз данных или в целях демонстрационного или исследовательского характера все файлы можно смело размещать на одном и том же носителе и в одном и том же каталоге, что мы в большинстве случаев и делаем в рамках данной книги.

Если файл данных и файл журнала транзакций в операторе CREATE DATABASE явно не описываются, то для их размещения выбираются пути по умолчанию, заданные при инсталляции системы. В нашем случае это будут пути

C:\Program Files\Microsoft SQL Server\MSSQLSERVER\MSSQL\DATA

ЗАМЕЧАНИЕ

Пути по умолчанию для файла данных и файла журнала транзакций можно изменить в Management Studio. Для этого в **Object Explorer** нужно щелкнуть правой кнопкой мыши по имени сервера базы данных и в контекстном меню выбрать **Properties**. В появившемся окне свойств сервера нужно выбрать вкладку **Database Settings** и изменить пути в полях **Data** и **Log** для файлов данных и журналов транзакций соответственно.

Именам файлов присваивается логическое имя файла с расширением **mdf** (файл данных) или **ldf** (журнал транзакций). Например, в приведенном только что примере с созданием базы данных **Strange** файл данных получит имя **Strange.mdf**, а файл журнала транзакций — **Strange.ldf**. Понятно, что по умолчанию всегда будет создаваться лишь один файл данных и только один файл журнала транзакций в одном и том же каталоге на внешнем носителе.

Обратите внимание, что имя файла задается одной строковой константой. Здесь по-прежнему не совсем понятной для меня причине недопустимо использование каких-либо выражений, внутренних переменных, операций. В частности, нельзя даже использовать простую операцию конкатенации строк. Первоначально складывается впечатление, что в пакете на создание базы данных динамически сформировать путь к файлу и имя файла вообще невозможно. Однако это не так. В Transact-SQL можно использовать очень полезный во многих случаях оператор **EXECUTE**, который позволяет динамически создавать в том числе и оператор **CREATE DATABASE**. Использование локальных переменных в таких ситуациях при создании базы данных см. далее в примерах. Кроме того, в утилите командной строки **sqlcmd** существует возможность вызывать на выполнение скрипты, содержащие параметры, значения которых подставляются при вызове этой утилиты. Такой пример мы также вскоре рассмотрим.

В случае, когда спецификация файла задает файлового потока (**filestream**), путь к файлу указывает только имена каталогов. При этом имена всех каталогов, кроме самого нижнего уровня, должны существовать в базе данных. Последний в пути каталог должен отсутствовать. Он будет создан системой.

Предложение SIZE

[**SIZE** = <целое1> [**KB** | **MB** | **GB** | **TB**]]

Необязательное предложение **SIZE** задает начальный размер файла. <целое1> в предложении является целым числом, определяющим размер в указанных следом за ним единицах — в килобайтах (**KB**), мегабайтах (**MB**), гигабайтах (**GB**) или в терабайтах (**TB**). Если единица измерения не указана, то предполагаются мегабайты; обратите внимание, что в описании синтаксиса **MB** подчеркнуто, что, как мы помним, используется для указания значения по умолчанию. Ненавязчиво сообщу, что при создании реальных систем я *всегда* указываю единицу измерения.

Размер любого файла не может быть меньше, чем 512 Кбайт, а первичный файл данных должен иметь размер не менее 3 Мбайт. <целое1> имеет целочисленный тип данных `INTEGER`, его значение не может превышать 2147483647. Здесь речь идет только о числовом значении самого параметра, а не о размере файла. Для задания больших размеров следует указывать соответствующие единицы измерения.

Вы также не сможете задать начальный размер файла, который превышает объем свободного места на выбранном носителе вашего компьютера. При создании базы данных и при размещении ее файлов система выполнит и такую проверку.

Если предложение `SIZE` не задано, то начальному размеру файла присваивается значение по умолчанию, определенное в системной базе данных `model`. В моей версии установленной системы файл данных по умолчанию получает начальный размер 3 Мбайта, а журнал транзакций — 1 Мбайт. Вряд ли эти значения изменятся при переходе к другим версиям системы.

Предложение `MAXSIZE`

```
[ MAXSIZE = { <целое2> [ KB | MB | GB | TB ] | UNLIMITED } ]
```

Необязательное предложение `MAXSIZE` задает максимальный размер, который может получить файл при увеличении количества данных, помещаемых в файл, или указывает, что размер файла не ограничен (параметр `UNLIMITED`). В последнем случае файл будет увеличиваться в размерах на величину, указанную в ключевом слове `FILEGROWTH`, пока не будет исчерпано все свободное пространство носителя. На самом деле это не совсем так. "Неограниченность" размера означает лишь, что файл данных не может превышать 16 терабайт (для редакции Express используется другое ограничение — вся база данных по размеру не может превышать 10 Гбайт), а файл журнала транзакций — 2 Тбайта. Как и в случае задания начального размера файла, в этом предложении максимальный размер может задаваться в килобайтах, мегабайтах, гигабайтах и терабайтах (параметры `KB`, `MB`, `GB` и `TB`, соответственно). Значением единицы измерения по умолчанию также является мегабайт.

Если это предложение не указано, то для файла данных задается неограниченный (`unlimited`) размер.

Предложение `FILEGROWTH`

```
[ FILEGROWTH = <целое3> [ KB | MB | GB | TB | % ] ]
```

Необязательное предложение `FILEGROWTH` позволяет задать значение величины приращения размера файла. Параметр `<целое3>` задает увеличение размера в килобайтах, мегабайтах, гигабайтах, в терабайтах или в процентах от начального размера файла, как указано в этом предложении (`KB`, `MB`, `GB`, `TB`, `%`). Если единица измерения приращения не указана, то принимается мегабайт. Если это предложение вообще не указано, то для файла данных приращение задается в 1 Мбайт, а для журнала транзакций устанавливается приращение в 10% от начального размера файла.

Как уже говорилось, размер страницы файла данных в базе имеет фиксированное значение 8 Кбайт (8192 байта) и не может быть изменен ни при создании базы данных, ни при ее изменении.

Файловая группа

Синтаксис описания файловой группы представлен в листинге 3.3 и в соответствующем R-графе (граф 3.7).

Листинг 3.3. Синтаксис описания файловой группы

```
<файловая группа> ::=  
FILEGROUP <имя файловой группы> [ CONTAINS FILESTREAM ] [DEFAULT]  
<спецификация файла> [, <спецификация файла>] ...
```



Граф 3.7. Синтаксис описания файловой группы

Имя файловой группы должно быть уникальным среди имен файловых групп этой базы данных.

Необязательное предложение **CONTAINS FILESTREAM** означает, что данная файловая группа предназначена только для хранения в файловой системе столбцов указанной таблицы с типом данных **VARBINARY(MAX)**. Примеры использования файловых потоков мы рассмотрим в [главе 5](#).

Ключевое слово **DEFAULT** указывает, что файловая группа является файловой группой по умолчанию в этой базе данных.

Файловой группе должно предшествовать, по меньшей мере, одно описание файла данных первичной группы. В состав файловой группы должен входить хотя бы один файл данных.

ЗАМЕЧАНИЕ ПО СИНТАКСИСУ ОПЕРАТОРА *CREATE DATABASE*

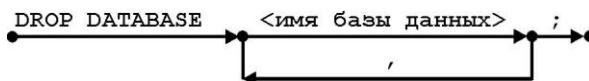
Кажется немного странным, что в языке Transact-SQL синтаксис оператора **CREATE DATABASE** для целей создания новой базы данных (другие варианты использования этого оператора мы рассмотрим ближе к концу главы) не позволяет явно установить начальные значения тому множеству характеристик базы данных, ее файловых групп и файлов, которые существуют в диалоговых средствах SQL Server. Для изменения значений характеристик по умолчанию используется оператор **ALTER DATABASE**. Его мы очень скоро будем рассматривать довольно подробно. В диалоговых средствах Management Studio и в случае первоначального создания базы данных можно задавать значения для большинства характеристик. Полагаю, что такую возможность было бы полезно внести и в оператор **CREATE DATABASE**.

3.4.1.2. Оператор удаления базы данных

Для удаления созданной пользователем и существующей в текущем экземпляре сервера базы данных используется оператор **DROP DATABASE**. Его синтаксис показан в листинге 3.4 и соответствующем R-графе (граф 3.8).

Листинг 3.4. Синтаксис оператора DROP DATABASE

```
DROP DATABASE <имя базы данных> [, <имя базы данных>]... ;
```



Граф 3.8. Синтаксис оператора DROP DATABASE

Оператор `DROP DATABASE` позволяет удалить одну или более указанных пользовательских баз данных или мгновенных снимков базы данных (см. разд. 3.7). Нельзя удалить системную базу данных. Нельзя также удалить базу данных, которая используется в настоящий момент, т. е. ту базу, которую открыл для работы какой-либо пользователь на том же компьютере или в сети. Удаление мгновенного снимка базы данных никак не влияет на базу данных-источник.

При удалении базы она удаляется из списка баз данных экземпляра сервера. Также с внешних носителей физически удаляются все файлы, относящиеся к этой базе данных — все файлы данных (первичный и вторичные) и все файлы журналов транзакций. Однако физическое удаление файлов происходит только в том случае, если база данных в момент удаления находится в состоянии `ONLINE`. Если же база неактивна (находится в состоянии `OFFLINE`), то удаление файлов не происходит.

При удалении мгновенного снимка сведения о нем удаляются из системного каталога и удаляются файлы мгновенного снимка.

* * *

Сейчас мы с вами создадим несколько простых баз данных с использованием оператора `CREATE DATABASE` при помощи утилиты `sqlcmd` и в программе Management Studio, но вначале потренируемся в отображении существующих баз данных, их файлов и интересующих нас характеристик — как характеристик баз данных, так и характеристик соответствующих файлов.

3.4.1.3. Создание и отображение баз данных в командной строке

Независимо от того, собираетесь ли вы когда-нибудь работать с командной строкой, рассмотрите детально следующие примеры. Они вам пригодятся и для работы в нормальной графической среде тоже.

Универсальной утилитой командной строки, позволяющей выполнять операторы Transact-SQL в SQL Server, является `sqlcmd`. Описание наиболее часто используемых параметров этой утилиты см. в *приложении 3*.

Для создания базы данных запустите на выполнение командную строку или Windows PowerShell. На экране появится соответствующее окно и подсказка, например:

```
PS C:\Users\Administrator>
```

В подсказке этого окна введите имя утилиты — sqlcmd. Если на вашем компьютере установлен единственный экземпляр сервера базы данных или вам требуется экземпляр сервера по умолчанию, то при вызове утилиты можно не задавать больше никаких параметров. На моем компьютере установлено два экземпляра SQL Server.

Для вызова утилиты, которая должна будет работать с версией Express Edition, требуется задать параметр -S, в котором нужно указать имя сервера и через обратную наклонную черту имя экземпляра сервера:

```
sqlcmd -S DEVRAVE\SQLEXPRESS
```

Для вызова утилиты, работающей с экземпляром сервера по умолчанию, можно сделать так:

```
sqlcmd -S DEVRAVE
```

Сам экземпляр сервера, компонент Database Engine, должен в момент вызова утилиты уже выполняться.

На экране появится подсказка самой утилиты:

```
1>
```

Давайте вначале отобразим существующие в экземпляре сервера базы данных, используя системное представление sys.databases. Введите в строке подсказки утилиты вначале команду USE, указывающую, что текущей базой данных является системная база данных master. После этой команды следует ввести GO, а затем оператор SELECT, обращающийся к системному представлению sys.databases. Нажмите клавишу <Enter> (пример 3.1).

Пример 3.1. Отображение баз данных текущего экземпляра сервера базы данных в системном представлении sys.databases

```
USE master;
GO
SELECT name, database_id, create_date, collation_name
FROM sys.databases;
```

Однако ничего интересного не произойдет, только на экране появятся введенные строки и следующая строка подсказки утилиты с номером 4:

```
1> USE master;
2> SELECT name, database_id, create_date, collation_name
3>   FROM sys.databases;
4>
```

Для того чтобы был выполнен введенный оператор Transact-SQL (или группа операторов), нужно ввести еще и оператор GO. После ввода этого оператора и нажатия клавиши <Enter> появится список всех баз данных, существующих в экземпляре сервера. Если вы еще не создавали пользовательские базы данных, то список будет содержать только описания четырех системных баз данных: master, tempdb, model и msdb и двух баз данных, относящихся к компоненту Reporting Services и исполь-

зуемых для внутренних целей: ReportServer и ReportServerTempDB. Скрытая база данных resource не отображается в этом списке.

Оператор SELECT, используемый в этом примере, позволяет получить указанные данные из таблицы (таблиц) базы данных или из представления. Он также позволяет просто вывести заданные величины: любые литералы, константы, результаты обращения к различным функциям. В этом случае в операторе не задается предложение FROM.

В примере 3.1 после ключевого слова SELECT мы перечислили в списке выбора этого оператора имена тех характеристик (столбцов) представления, которые хотим отобразить, а в предложении FROM указали имя представления, из которого должны быть получены эти характеристики: sys.databases. В результате выполнения такого оператора мы получим список всех баз данных, существующих в текущем экземпляре сервера базы данных.

ЗАМЕЧАНИЕ

Следует напомнить, что ключевые слова языка Transact-SQL нечувствительны к регистру — их можно вводить как строчными, так и прописными буквами.

Однако полученный результат не производит хорошего впечатления. Сведения по каждой базе данных занимают несколько строк. Это сильно ухудшает читаемость результата. И дело не только в том, что по умолчанию длина строки в PowerShell составляет 120 символов, а в командной строке 80. Размер строки можно изменить, щелкнув правой кнопкой мыши по заголовку окна, выбрав элемент меню **Свойства** и изменив на вкладке **Расположение** размер окна по ширине. Неприятность в том, что размер отображаемых полей слишком велик.

Для улучшения читаемости результата внесем некоторые изменения в наш оператор SELECT, выполнив простые преобразования получаемых данных и задав осмысленные тексты заголовков отображаемых столбцов. В подсказке утилиты введите и выполните несколько измененный оператор выборки данных (пример 3.2):

Пример 3.2. Более правильное отображение в PowerShell баз данных текущего экземпляра сервера базы данных в системном представлении sys.databases

```
USE master;
GO
SELECT CAST(name AS CHAR(20)) AS 'NAME',
       CAST(database_id AS CHAR(4)) AS 'ID',
       create_date AS 'DATE',
       CAST(collation_name AS CHAR(23)) AS 'COLLATION'
FROM sys.databases;
GO
```

Теперь мы получили более симпатичный список баз данных. На моем ноутбуке этот список выглядит следующим образом:

NAME	ID	DATE	COLLATION
master	1	2003-04-08 09:13:36.390	Cyrillic_General_CI_AS
tempdb	2	2012-06-08 16:39:42.653	Cyrillic_General_CI_AS
model	3	2003-04-08 09:13:36.390	Cyrillic_General_CI_AS
msdb	4	2012-02-10 21:02:17.770	Cyrillic_General_CI_AS
ReportServer	5	2012-06-07 23:37:51.727	Latin1_General_CI_AS_KS_WS
ReportServerTempDB	6	2012-06-07 23:37:53.120	Latin1_General_CI_AS_KS_WS

(6 row(s) affected)

В операторе для трех столбцов мы используем весьма простую и очень удобную в работе функцию преобразования данных `CAST()` (которую довольно подробно с многочисленными примерами использования рассмотрим в следующей главе). Синтаксис функции прост:

`CAST(<идентификатор> AS <тип данных>)`

Эта функция позволяет привести тип данных заданной переменной или константы (параметр `идентификатор`) к типу данных, указанному после ключевого слова `as` (параметр `тип данных`). Здесь мы используем эту функцию лишь с целью уменьшения размера поля, отводимого для отображения столбца. Например, имя базы данных может содержать до 128 символов. Если фактическое имя короче, то справа при его отображении система добавляет недостающие пробелы до максимального значения 128. Мы же в операторе при помощи функции `CAST()` сократили этот размер до 20 символов, преобразовав исходный строковый тип данных у столбца `name` (`CHAR(128)`) с количеством символов 128 опять же в строковый, но с другим размером, указав в выходном типе данных 20 символов (`CHAR(20)`).

Мы в операторе `SELECT` только не задали никакого преобразования для столбца, содержащего дату и время, поскольку преобразование, выполняемое по умолчанию при отображении этого типа данных, нас вполне устраивает.

После имени отображаемого столбца в операторе `SELECT` мы можем указать предложение `AS` (не путайте с параметром `AS` в функции преобразования данных `CAST()`) и задать в этом предложении в апострофах текст, который будет отображаться в заголовке соответствующего столбца. Именно это мы и сделали для каждого выбираемого столбца. Разумеется, здесь мы также можем задать и русскоязычные заголовки — 'Имя базы данных', 'Идентификатор', 'Дата и время создания', 'Порядок сортировки', хотя размер таких заголовков будет иметь несколько большую длину. Результирующий размер отображаемых столбцов будет соответствовать большему значению из заданного размера отображаемых данных и размера заголовка.

ЗАМЕЧАНИЕ ПО СИНТАКСИСУ

Если заголовок столбца в предложении `AS` не содержит специальных символов, в частности пробелов, как и в рассмотренном сейчас примере, а является правильным обычным идентификатором (см. главу 2), то его вообще-то можно было бы и не заключать в апострофы. В некоторых информационных и многих учебных материалах корпорации Microsoft, да и в документе Books Online, вы можете увидеть примеры использования такого варианта. Такая свобода задания строковых значений в данном случае в общем-то совершенно понятна. В этом месте ожидается использование

строкового данного, и соответствующий набор знаков воспринимается как строка символов. Наверное не стоит напоминать, что это не является хорошей практикой. Если мы используем строковые константы в наших пакетах, в данном случае при задании заголовков, то всегда следует заключать их в апострофы, чтобы избежать возможной путаницы, повысить читаемость кода и избежать лишних неприятностей при изменениях системы в будущем.

Несколько слов об операторе `GO`. Он не является оператором языка Transact-SQL, по этой причине он не может завершаться символом точка с запятой, наличие этого символа вызовет синтаксическую ошибку. Это служебный оператор утилиты `sqlcmd` и программы Management Studio. В SQL Server существует понятие пакета операторов (`batch`). Пакет содержит группу операторов Transact-SQL; фактическое выполнение группы начинается только после ввода оператора `GO`. Синтаксис оператора `GO`:

```
GO [<количество повторений>]
```

Если количество повторений (которое должно быть положительным целым числом) указано, то весь предыдущий фрагмент пакета выполняется заданное число раз. Если количество повторений не указано, то пакет выполняется один раз. Выполните предыдущий пакет операторов, указав оператор `GO` с любым количеством повторений, большим единицы. Вы получите заданное вами количество одинаковых наборов строк отображения баз данных, существующих в экземпляре сервера.

Теперь немного усложним оператор `SELECT`, указав, что в список вывода должны помещаться только сведения по базе данных `tempdb`. Рассмотрим один из простых вариантов предложения `WHERE` в операторе `SELECT`. Введите и выполните следующие операторы (пример 3.3).

Пример 3.3. Отображение одной базы данных в системном представлении `sys.databases`

```
USE master;
GO
SELECT CAST(name AS CHAR(20)) AS 'NAME',
       CAST(database_id AS CHAR(4)) AS 'ID',
       create_date AS 'DATE',
       CAST(collation_name AS CHAR(23)) AS 'COLLATION'
  FROM sys.databases
 WHERE name = 'tempdb';
GO
```

В предложении `WHERE` задается конкретное требуемое значение поля `name`, определяющее имя единственной отображаемой базы данных. В результат отображения попадет только одна заданная строка, описывающая базу данных `tempdb`:

NAME	ID	DATE	COLLATION
tempdb	2	2012-06-08 17:16:42.653	Cyrillic_General_CI_AS
(1 row(s) affected)			

ЗАМЕЧАНИЕ

В самом начале каждого пакета мы записывали оператор `USE`, который указывает текущую базу данных, т. е. ту базу данных, с которой выполняются все последующие действия. Поскольку мы сейчас выполняем действия при обращении к системному представлению, которое присутствует в любой базе данных, как в системной, так и в пользовательской, то в данном случае не имеет значения, какая именно база данных является текущей. Оператор `USE` в этих примерах можно опустить. Во многих других случаях использования системных средств работы с базами данных имеет значение, какая база данных является текущей. Именно с базой данных, определенной в операторе `USE`, выполняются многие действия при вызове хранимых процедур, представлений или функций. Опять же по правилам хорошего тона следует всегда указывать этот оператор в ваших пакетах. Что лично я, к сожалению, делаю далеко не всегда. Очень рекомендую каждый раз после этого оператора вводить `GO`. В некоторых версиях системы отсутствие `GO` может приводить к неприятным результатам.

* * *

Здесь я хочу сделать небольшое отступление и сказать несколько слов об используемых в работе программных средствах и вообще об удобстве в работе. Утилиту `sqlcmd` можно запускать на выполнение в PowerShell и в обычной командной строке.

Результаты будут весьма похожими. Позже мы сможем сравнить вид этих отображений с тем, что можно получить в программе Management Studio. Надо сказать, что программисты имеют самые различные эстетические пристрастия. Кто-то предпочитает графический интерфейс (таких, разумеется, большинство), но есть люди, искренне любящие командную строку в различных ее вариантах и в принципе не признающие графический интерфейс. Правда, среди наших пользователей (как "юзеров", так и "ламеров") таких я что-то не встречал.

* * *

Более подробную информацию о базах данных и их файлах в текущем экземпляре сервера базы данных можно получить, используя системное представление просмотра каталогов `sys.master_files`. Введите и выполните следующий оператор, отображающий сведения о файлах баз данных (пример 3.4).

Пример 3.4. Отображение баз данных и их файлов в системном представлении `sys.master_files`

```
USE master;
GO
SELECT database_id, file_id, type, type_desc, data_space_id,
       name, physical_name, is_read_only, state, state_desc,
       size, max_size, growth, is_percent_growth
FROM sys.master_files;
GO
```

Вывод опять же будет не слишком наглядным. Здесь просто перечислены все те столбцы, которые мы с вами только что рассмотрели в предыдущем разделе.

Вначале нужно отобрать из представления те столбцы, которые могут быть интересны в первую очередь. Вот более хороший вариант отображения файлов баз данных в командной строке, пример 3.5.

Пример 3.5. Более правильный вариант отображения баз данных и их файлов в системном представлении sys.master_files

```
USE master;
GO
SELECT CAST(database_id AS CHAR(5)) AS 'DB ID',
       CAST(type_desc AS CHAR(6)) AS 'Descr',
       CAST(name AS CHAR(24)) AS 'File Name',
       CAST(state_desc AS CHAR(5)) AS 'State',
       CAST(size AS CHAR(5)) AS 'Size',
       max_size AS 'Max Size',
       CAST(growth AS CHAR(6)) AS 'Growth'
FROM sys.master_files;
GO
```

Результатом будет отображение всех файлов баз данных текущего экземпляра сервера с некоторыми их характеристиками:

DB ID	Descr	File Name	State	Size	Max Size	Growth
1	ROWS	master	ONLIN	624	-1	10
1	LOG	mastlog	ONLIN	224	-1	10
2	ROWS	tempdev	ONLIN	1024	-1	10
2	LOG	templog	ONLIN	64	-1	10
3	ROWS	modeldev	ONLIN	520	-1	128
3	LOG	modellog	ONLIN	128	-1	10
4	ROWS	MSDBData	ONLIN	2136	-1	10
4	LOG	MSDBLog	ONLIN	584	268435456	10
5	ROWS	ReportServer	ONLIN	648	-1	128
5	LOG	ReportServer_log	ONLIN	880	268435456	10
6	ROWS	ReportServerTempDB	ONLIN	520	-1	128
6	LOG	ReportServerTempDB_log	ONLIN	130	268435456	10

(12 row(s) affected)

По поводу красоты мы здесь вопрос вроде бы решили, однако возникают сомнения относительно безупречности полученного результата. Не очень нравятся какие-то числа в первом столбце этого результата: идентификаторы баз данных. Лучше было бы отображать здесь соответствующие имена баз данных.

Изменим обращение к системному представлению sys.master_files следующим образом, используя системную функцию DB_NAME(), позволяющую по идентификатору, получаемому из этого представления, находить имена баз данных. Простоты ради уберем некоторые столбцы. Выполните пример 3.6.

Пример 3.6. Отображение баз данных и их файлов в системном представлении sys.master_files

```
USE master;
GO
SELECT CAST(DB_NAME(database_id) AS CHAR(20)) AS 'DB Name',
       CAST(NAME AS CHAR(24)) AS 'File Name',
       CAST(state_desc AS CHAR(7)) AS 'State',
       CAST(type_desc AS CHAR(6)) AS 'Descr'
FROM sys.master_files
ORDER BY 'DB Name';
GO
```

Результатом будет следующий список:

DB Name	File Name	State	Descr
master	master	ONLINE	ROWS
master	mastlog	ONLINE	LOG
model	modeldev	ONLINE	ROWS
model	modellog	ONLINE	LOG
msdb	MSDBData	ONLINE	ROWS
msdb	MSDBLog	ONLINE	LOG
ReportServer	ReportServer	ONLINE	ROWS
ReportServer	ReportServer_log	ONLINE	LOG
ReportServerTempDB	ReportServerTempDB	ONLINE	ROWS
ReportServerTempDB	ReportServerTempDB_log	ONLINE	LOG
tempdb	tempdev	ONLINE	ROWS
tempdb	templog	ONLINE	LOG

(12 row(s) affected)

Благодаря использованию системной функции DB_NAME(), мы получили имена баз данных. Замечательно и то, что сделать это оказалось необыкновенно просто.

В этом пакете в операторе SELECT мы добавили еще одну возможность. Последней строкой оператора записано ORDER BY 'DB Name'. Это предложение позволяет упорядочить отображаемый список по значению первого поля, указанного в списке выбора оператора, что мы и видим в результате отображения. Причем мы указали не имя столбца, получаемого из системного представления, не его номер (эти варианты тоже возможны в операторе SELECT), а текст заголовка, заданный нами после ключевого слова AS в списке выбора.

Предложение ORDER BY можно задать и в виде ORDER BY 1. Здесь указывается, что упорядочение осуществляется по первому полю из списка выбора. Результат выполнения будет, разумеется, точно таким же.

Другое системное представление просмотра каталогов sys.database_files позволяет просмотреть все файлы одной текущей базы данных, заданной в операторе USE.

Ведите и выполните следующие операторы для отображения файлов базы данных, скажем, master (пример 3.7).

Пример 3.7. Отображение файлов базы данных master в системном представлении sys.database_files

```
USE master;
GO
SELECT CAST(file_id AS CHAR(2)) AS 'ID',
       CAST(type AS CHAR(4)) AS 'Type',
       CAST(type_desc AS CHAR(11)) AS 'Description',
       CAST(name AS CHAR (12)) AS 'Name',
       state AS 'State',
       CAST(state_desc AS CHAR(10)) AS 'State desc',
       CAST(size AS CHAR(5)) AS 'Size'
FROM sys.database_files;
GO
```

Результат будет следующим:

ID	Type	Description	Name	State	State desc	Size
1	0	ROWS	master	0	ONLINE	624
2	1	LOG	mastlog	0	ONLINE	224

(2 row(s) affected)

В точности такой же результат мы можем получить, используя и системное представление sys.master_files при задании в операторе SELECT предложения WHERE, в котором будет указан требуемый идентификатор нужной нам базы данных. Для системной базы данных master, как мы можем видеть из листинга примера 3.2, этот идентификатор равен единице. Выполните операторы примера 3.8:

Пример 3.8. Отображение файлов базы данных master в системном представлении sys.master_files

```
USE master;
GO
SELECT CAST(file_id AS CHAR(2)) AS 'ID',
       CAST(type AS CHAR(4)) AS 'Type',
       CAST(type_desc AS CHAR(11)) AS 'Description',
       CAST(name AS CHAR (12)) AS 'Name',
       state AS 'State',
       CAST(state_desc AS CHAR(10)) AS 'State desc',
       CAST(size AS CHAR(5)) AS 'Size'
FROM sys.master_files
WHERE database_id = 1;
GO
```

В предложении WHERE указывается, что должны отображаться только те строки файлов, для которых идентификатор базы данных (`database_id`) равен единице, т. е. будут отображаться строки файлов, относящиеся к базе данных `master`.

Этот пример выглядит как-то не очень красиво. Получается, что для того чтобы узнать значение идентификатора базы данных `master`, нам нужно выполнить отображение всех баз данных (см. пример 3.2), найти в полученном списке значение идентификатора базы данных `master` и подставить это значение в предложение WHERE.

На самом деле здесь можно и в одном операторе осуществить поиск идентификатора нужной базы данных, используя оператор `SELECT`, который обращается к системному представлению `sys.databases`. Для этого в предыдущем примере предложение WHERE нужно записать в следующем виде:

```
WHERE database_id =  
  ( SELECT database_id  
    FROM sys.databases  
   WHERE name = 'master' );
```

Внутренний оператор `SELECT` в этом предложении возвращает значение идентификатора (столбец `database_id`) базы данных `master`, которая задается при помощи указания имени этой базы данных (столбец `name`). Обратите внимание, что по правилам синтаксиса SQL этот внутренний оператор `SELECT` обязательно должен быть заключен в круглые скобки.

Есть еще более простой способ выполнить нужное нам отображение файлов базы данных `master`, используя системную функцию `DB_ID()`, которая возвращает идентификатор базы данных по ее имени. Эту функцию мы будем еще не один раз использовать в наших скриптах. Синтаксис функции:

```
DB_ID([<имя базы данных>])
```

Если указанная в параметре база данных отсутствует в системе, то функция вернет значение `NULL`.

Ведите и выполните операторы примера 3.9.

Пример 3.9. Лучший вариант отображения файлов базы данных master в системном представлении sys.master_files

```
USE master;  
GO  
SELECT CAST(file_id AS CHAR(2)) AS 'ID',  
       CAST(type AS CHAR(4)) AS 'Type',  
       CAST(type_desc AS CHAR(11)) AS 'Description',  
       CAST(name AS CHAR(12)) AS 'Name',  
       state AS 'State',  
       CAST(state_desc AS CHAR(10)) AS 'State desc',  
       CAST(size AS CHAR(5)) AS 'Size'
```

```
FROM sys.master_files  
WHERE database_id = DB_ID('master');  
GO
```

Если в функции `DB_ID()` не указать необязательный параметр имя базы данных, то она вернет идентификатор текущей базы данных, которая была задана в последнем операторе `USE`.

ЗАМЕЧАНИЕ

Если вам нужно отобразить только сведения по одному из файлов базы данных (это показано в примерах Books Online), то в предложении `WHERE` оператора `SELECT` можно задать имя столбца `name` и после знака равенства в апострофах имя интересующего вас файла, например `name = 'master'`. В этом случае вы получите сведения только по файлу данных базы данных `master`. Однако если в системе (в текущем экземпляре сервера базы данных) у различных баз данных существуют файлы с тем же именем, то вы получите список всех таких файлов. Так что наш с вами вариант отображения файлов конкретной базы данных из примера 3.9 много лучше всех других.

Давайте еще кратко рассмотрим очень простую системную функцию `FILE_NAME()`, которая всего лишь возвращает логическое имя файла базы данных (файла данных или журнала транзакций) по идентификатору этого файла для текущей базы данных. Синтаксис обращения к функции:

```
FILE_NAME(<идентификатор файла>)
```

Идентификатором может быть любое число. Если в текущей базе данных существует файл с таким идентификатором, то функция вернет его логическое имя. Иначе функция возвращает значение `NULL`. В качестве идентификатора вы можете указать и дробное число. В этом случае дробная часть просто отбрасывается (округление не выполняется). Можно задать нулевое значение (напомню — файлы в базе данных нумеруются, начиная с единицы) и даже отрицательное значение; результат, возвращенный функцией при таких значениях параметра, будет `NULL`, ошибка сгенерирована не будет.

Выполните следующие операторы (пример 3.10).

Пример 3.10. Отображение логических имен файлов базы данных master в системной функции FILE_NAME()

```
USE master;  
GO  
SELECT DB_ID() AS 'ID',  
       CAST(FILE_NAME(1) AS CHAR(10)) AS 'Файл 1',  
       CAST(FILE_NAME(2) AS CHAR(10)) AS 'Файл 2',  
       CAST(FILE_NAME(3) AS CHAR(10)) AS 'Ничего';  
GO
```

Результатом будет:

```
ID      Файл 1      Файл 2      Ничего
-----
1       master       mastlog     NULL
(1 row(s) affected)
```

Здесь в операторе `USE` указывается база данных `master`, к которой будут обращаться по умолчанию все следующие операторы. В операторе `SELECT` для имен логических файлов мы также выполняем преобразование данных, чтобы результат поместился в одну строку. По ходу дела в этом операторе мы отображаем идентификатор текущей базы данных. В функции `DB_ID()` мы не указали никакого имени базы данных, поэтому функция вернет идентификатор текущей базы данных, т. е. `master`.

В четвертом столбце задается несуществующий у базы данных номер файла — 3. Результатом, как мы видим, будет значение `NULL`.

Обратите внимание, что в данном примере в операторе `SELECT` не указывается предложение `FROM`, т. е. не говорится, откуда должны получаться результаты — из какой таблицы, из какого представления. Это означает, что на выходе такого запроса будет ровно одна строка, содержащая перечисленные в списке выбора оператора `SELECT` значения, полученные при обращении к функциям.

Все наши операторы мы вводили руками в диалоговом режиме в подсказке утилиты (ну, если уж быть честным, я-то копировал заранее подготовленные мною тексты из электронного варианта этой книги и помещал их в подсказку утилиты). Если вы допустите какую-либо ошибку, то вам придется заново повторять почти все введенные данные. Утилита `sqlcmd` имеет параметр `-i`, который позволяет указать имя файла (файл скрипта), из которого утилита будет читать операторы. Можно поместить пакет операторов в файл, корректировать многократно и выполнять при вызове утилиты `sqlcmd`. Пример использования файла скрипта:

```
sqlcmd -i "D:\Ex3-10.sql"
```

Здесь в параметре `-i` указывается полный путь к файлу и имя файла скрипта.

Еще про один параметр утилиты `sqlcmd`. Если мы хотим, чтобы результат выполнения утилиты выводился не на монитор, а помещался в какой-либо файл, то при вызове утилиты нужно задать параметр `-o`, в котором указывается путь к файлу и имя файла, куда утилита будет выводить все результаты и диагностические сообщения. Имя этого параметра является чувствительным к регистру: вы должны ввести именно строчную букву `o`, а не прописную.

Пример использования этого параметра:

```
sqlcmd -o "D:\Result.txt"
```

Все выходные данные, создаваемые при выполнении утилиты, будут выводиться в файл `Result.txt` в корневом каталоге на диске `D:`. Если файл отсутствует на диске, то он будет создан. Если же файл уже существует, то новые строки будут добавляться в конец файла, не изменяя существующих в файле данных.

Напомню, что описание наиболее полезных параметров утилиты `sqlcmd` содержится в *приложении 2*.

* * *

Теперь, наконец, создадим в утилите `sqlcmd` несколько новых пользовательских баз данных. Потом их поудаляем, чтобы затем опять создать, но уже с использованием Management Studio.

Собственно говоря, различные варианты создания баз данных хорошо описаны в документе Books Online. Сейчас мы с вами выполним похожие действия.

Создадим базу данных, где все, что можно, будем устанавливать по умолчанию. Выполните следующие операторы примера 3.11.

Пример 3.11. Создание и отображение базы данных со всеми значениями по умолчанию

```
USE master;
GO
CREATE DATABASE SimpleDB;
GO
SELECT CAST(file_id AS CHAR(2)) AS 'ID',
       CAST(type AS CHAR(4)) AS 'Type',
       CAST(type_desc AS CHAR(11)) AS 'Description',
       CAST(name AS CHAR (12)) AS 'Name',
       state AS 'State',
       CAST(state_desc AS CHAR(10)) AS 'State desc',
       CAST(size AS CHAR(5)) AS 'Size'
FROM sys.master_files
WHERE database_id = DB_ID('SimpleDB');
GO
```

В результате выполнения оператора `CREATE DATABASE` будет создана база данных `SimpleDB`. Все ее характеристики устанавливаются по умолчанию. При выполнении оператора `SELECT` в данном пакете мы получим следующий список файлов и их характеристик этой базы данных:

ID	Type	Description	Name	State	State desc	Size
1	0	ROWS	SimpleDB	0	ONLINE	520
2	1	LOG	SimpleDB_log	0	ONLINE	130

(2 row(s) affected)

Чаще всего в процессе проектирования баз данных вы будете создавать базу данных и ее объекты, некоторое время с гордостью любоваться результатами вашей деятельности, а затем с грустью замечать, что вы что-то не учли, что-то сделали неверно. Тогда вы начнете вносить изменения в ваши скрипты и вновь запускать их на выполнение. Как правило, база данных пересоздается вами заново. Если вы за-

будете перед этим удалить уже созданную и не совсем правильную базу данных, то получите сообщение об ошибке. Сейчас я повторно ввел эти же самые операторы из примера 3.11 и получил следующее сообщение:

```
Msg 1801, Level 16, State 3, Line 1
Database 'SimpleDB' already exists. Choose a different database name.
```

(Сообщение 1801, уровень 16, состояние 3, строка 1

База данных 'SimpleDB' уже существует. Выберите другое имя базы данных.)

Чтобы избежать таких неприятностей, настоятельно рекомендую использовать функцию `DB_ID()`, которую мы с вами уже применяли для определения идентификатора базы данных по ее имени.

В пакетах SQL Server допустимо использование и операторов ветвления, в частности, оператора `IF`, который, как и в обычных языках программирования, позволяет сделать некоторые проверки и на основании результата таких проверок выполнить различные действия. Сейчас мы его используем для проверки существования нашей базы данных, которую собираемся заново создать. Внесите следующие изменения в ваш пакет создания базы данных `SimpleDB` (пример 3.12).

Пример 3.12. Создание базы данных с удалением существующей "старой" базы данных

```
USE master;
GO
IF DB_ID('SimpleDB') IS NOT NULL
    DROP DATABASE SimpleDB;
GO
CREATE DATABASE SimpleDB;
GO
```

В операторе `IF` мы проверяем при помощи функции `DB_ID()` существование базы данных `SimpleDB`. Если база данных существует, то функция вернет целое число — идентификатор этой базы данных, и тогда будет выполнен оператор удаления базы данных: `DROP DATABASE`.

Если же в системе нет такой базы данных, то функция `DB_ID()` вернет `NULL`. В этом случае оператор удаления не будет выполняться.

ЗАМЕЧАНИЕ

Возможность использования операторов ветвления, операторов циклов и некоторых других в пакетах SQL Server является необыкновенно удобным средством. Не все системы управления базами данных имеют такую возможность. Пользуясь случаем, хочу от имени всего прогрессивного человечества поблагодарить корпорацию Microsoft за такое средство.

Теперь создадим базу данных, с которой мы будем работать на протяжении всей этой книги. Это `BestDatabase`, а имена двух ее файлов будут `Winner` с соответствующими расширениями. При создании базы данных мы явно укажем логические

имена файла данных и файла журнала транзакций и для них зададим все необходимые значения параметров. Выполните операторы примера 3.13.

Пример 3.13. Создание базы данных BestDatabase

```
USE master;
GO
IF DB_ID('BestDatabase') IS NOT NULL
    DROP DATABASE BestDatabase;
GO
CREATE DATABASE BestDatabase
ON PRIMARY (NAME = BestDatabase_dat,
    FILENAME = 'D:\BestDatabase\Winner.mdf',
    SIZE = 5 MB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 1 MB)
LOG ON (NAME = BestDatabase_log,
    FILENAME = 'D:\BestDatabase\Winner.ldf',
    SIZE = 2 MB,
    MAXSIZE = 30 MB,
    FILEGROWTH = 1 MB);
GO
```

Оба файла базы данных — файл данных и файл журнала транзакций — располагаются на диске D: в каталоге BestDatabase. Напомню, что на соответствующем диске каталог с этим именем уже должен существовать, иначе при выполнении оператора создания базы данных вы получите сообщение об ошибке. Сами же файлы с такими именами должны отсутствовать в указанном каталоге.

Для файла данных в создаваемой базе данных установлен начальный размер 5 Мбайт, приращение указано 1 Мбайт, максимальный размер не ограничивается, следовательно, файл может расти до исчерпания объема дискового пространства или до 16 Тбайт.

Начальный размер файла журнала транзакций задается 2 Мбайта, максимальный размер 30 Мбайт, а квант увеличения размера — 1 Мбайт.

МАЛЕНЬКОЕ ЗАМЕЧАНИЕ ПО СИНТАКСИСУ

При указании размеров в операторе CREATE DATABASE единицы измерения могут записываться сразу же после числа, а могут помещаться и через один или более пробелов. Во втором случае запись выглядит, как мне кажется, много лучше. Надеюсь, вы обратили внимание, что во всех примерах единицы измерения заданы явно — там, где можно было бы опустить указание мегабайтов, ключевое слово MB все равно присутствует.

Если у вас еще остаются смутные сомнения в необходимости явного задания величин, для которых можно было бы использовать значения по умолчанию, попробуйте разобраться в скриптах, написанных для других систем управления базами данных, где использованы принятые именно там значения по умолчанию.

Следующий пример в принципе повторяет предыдущий, однако здесь мне хочется рассмотреть некоторые дополнительные полезные средства, используемые в пакетах SQL Server, — локальные переменные и оператор EXECUTE. Выполните следующий пакет (пример 3.14).

Пример 3.14. Создание базы данных BestDatabase, другой вариант

```
USE master;
GO
IF DB_ID('BestDatabase') IS NOT NULL
    DROP DATABASE BestDatabase;
GO

DECLARE @path AS VARCHAR(255),
        @path_data AS VARCHAR(255),
        @path_log AS VARCHAR(255);
SET @path = 'D:\BestDatabase\' ;
SET @path_data = @path + 'Winner.mdf';
SET @path_log = @path + 'Winner.ldf';

EXECUTE (
    'CREATE DATABASE BestDatabase
    ON PRIMARY (NAME = BestDatabase_dat,
        FILENAME = ''' + @path_data + ''',
        SIZE = 5 MB,
        MAXSIZE = UNLIMITED,
        FILEGROWTH = 1 MB)
    LOG ON (NAME = BestDatabase_log,
        FILENAME = ''' + @path_log + ''',
        SIZE = 2 MB,
        MAXSIZE = 30 MB,
        FILEGROWTH = 1 MB);');
GO
```

В этом примере мы в операторе DECLARE объявляем три локальные переменные, т. е. переменные, используемые только в данном пакете: @path, @path_data и @path_log, указав для них строковый тип данных переменной длины до 255 символов (AS VARCHAR(255)). Имена локальных переменных должны начинаться с символа @. Затем операторами SET мы присваиваем этим переменным значения, причем для переменных @path_data и @path_log мы используем операцию конкатенации (соединения строк), которая задается символом плюс (+). В результате эти две локальные переменные будут иметь значение полного пути к файлу данных и к файлу журнала транзакций соответственно.

ЗАМЕЧАНИЕ

Существование объявленных локальных переменных ограничено оператором GO. После выполнения этого оператора система уже ничего "не знает" про любые объявлен-

ные локальные переменные. Далее в скрипте можно объявлять переменные с теми же именами и с любыми иными типами данных.

Собственно для создания базы данных мы выполняем оператор EXECUTE (для него можно также использовать и сокращение EXEC), которому в качестве параметра передаем строку, которую создаем опять же при выполнении конкатенации строковых констант и значений локальных параметров @path_data и @path_log. Вся строка заключается в апострофы, а параметр оператора помещен в круглые скобки.

Обратите внимание, как здесь определяются имена файлов данных и журнала транзакций:

```
FILENAME = ''' + @path_data + ''',
...
FILENAME = ''' + @path_log + ''',
...  
...
```

После знака равенства подряд идут три апострофа. Первые два задают апостроф *внутри* предыдущей части строки (вы помните, что для представления одного апострофа в строке, заключенной в апострофы, нужно записать подряд два апострофа). Третий апостроф завершает предыдущую строку.

Похожим образом в следующей группе из трех апострофов первый начинает строку, а другие два задают апостроф *внутри* этой строки. В результате выполнения всех этих действий мы получаем пути к файлам, которые по правилам синтаксиса должны быть заключены в апострофы:

```
'D:\BestDatabase\Winner.mdf'
```

и

```
'D:\BestDatabase\Winner.ldf'
```

Вся созданная таким образом строка является правильным оператором CREATE DATABASE, который задает создание новой базы данных.

Для того чтобы просмотреть и проверить правильность результата формирования такой строки, достаточно в этом пакете только лишь заменить оператор EXECUTE на SELECT. Стока будет отображена на мониторе. Здесь довольно легко можно найти и исправить ошибки. Что я и сделал при написании предыдущего примера, поскольку вначале при создании этой строки допустил ошибку.

Пример 3.14 является не просто демонстрацией некоторых из тех возможностей, которые существуют в SQL Server. Этот прием позволяет в программных пакетах на основании каких-то условий динамически создавать операторы Transact-SQL. Есть еще один способ динамического создания операторов с использованием утилиты sqlcmd, который мы рассмотрим чуть позже.

Теперь создадим базу данных, содержащую два файла данных и два файла журнала транзакций. По правде сказать, практически ничего нового мы здесь не увидим (пример 3.15). Напомню только, что перед выполнением примера на диске D: нужно создать каталог Multy.

Пример 3.15. Создание многофайловой базы данных

```
USE master;
GO
IF DB_ID('Multy') IS NOT NULL
    DROP DATABASE Multy;
GO
CREATE DATABASE Multy
ON
PRIMARY
( NAME = Multy1,
  FILENAME = 'D:\Multy\Multy1.mdf'),
( NAME = Multy2,
  FILENAME = 'D:\Multy\Multy2.ndf')
LOG ON
( NAME = MultyL1,
  FILENAME = 'D:\Multy\MultyL1.ldf'),
( NAME = MultyL2,
  FILENAME = 'D:\Multy\MultyL2.ldf');
GO
```

Думаю, здесь нам с вами все понятно. Все заданные характеристики в операторе создания базы данных нам уже известны. Следует только напомнить, что создание нескольких файлов журнала транзакций на одном и том же носителе не является осмысленным занятием. Более одного журнала транзакций следует создавать на различных носителях, если есть проблемы с доступным объемом внешней памяти.

* * *

Во всех предыдущих примерах создаваемые базы данных имели только одну файловую группу — первичную (`PRIMARY`), которая обязательно присутствует для каждой базы данных. База данных помимо первичной файловой группы может содержать и произвольное количество других файловых групп, называемых пользовательскими или вторичными файловыми группами.

Теперь создадим базу данных, в которой будет две файловые группы, каждая из которых содержит, скажем, по два файла данных.

Рассмотрим фрагмент синтаксиса оператора `CREATE DATABASE`, предложение `FILEGROUP`, относящееся к созданию файловых групп:

```
<файловая группа> ::=  
FILEGROUP <имя файловой группы> [ CONTAINS FILESTREAM ] [DEFAULT]  
<спецификация файла> [, <спецификация файла>]...
```

Для файловой группы указывается имя, которое должно быть уникальным в текущей базе данных, после чего следует описание как минимум одного файла данных.

Для создания такой базы данных с двумя файловыми группами выполните операторы примера 3.16.

Пример 3.16. Создание базы данных с двумя файловыми группами

```
USE master;
GO
IF DB_ID('MultyGroup') IS NOT NULL
    DROP DATABASE MultyGroup;
GO
CREATE DATABASE MultyGroup
ON
PRIMARY
( NAME = MultyGroup1,
  FILENAME = 'D:\MultyGroup\MultyGroup1.mdf'),
( NAME = MultyGroup2,
  FILENAME = 'D:\MultyGroup\MultyGroup2.ndf'),
FILEGROUP MultyGroup2
( NAME = MultyGroup3,
  FILENAME = 'D:\MultyGroup\MultyGroup3.ndf'),
( NAME = MultyGroup4,
  FILENAME = 'D:\MultyGroup\MultyGroup4.ndf')
LOG ON
( NAME = MultyGroupLog1,
  FILENAME = 'D:\MultyGroup\MultyGroupLog1.ldf'),
( NAME = MultyGroupLog2,
  FILENAME = 'D:\MultyGroup\MultyGroupLog2.ldf');
GO
```

Вообще-то все основные характеристики файлов данных и журнала транзакций взяты с некоторыми изменениями из примера 3.15. Здесь только добавлена вторичная файловая группа с именем MultyGroup2.

Теперь посмотрим, что у нас в результате получилось. Сначала отобразим файлы созданной базы данных. Обратимся к уже хорошо знакомому нам системному представлению sys.database_files. Выполните оператор примера 3.17.

Пример 3.17. Отображение состояния базы данных с двумя файловыми группами

```
USE MultyGroup;
GO
SELECT CAST(file_id AS CHAR(2)) AS 'ID',
       CAST(type AS CHAR(4)) AS 'Type',
       CAST(type_desc AS CHAR(11)) AS 'Description',
       CAST(name AS CHAR (16)) AS 'Name',
       state AS 'State',
       CAST(state_desc AS CHAR(10)) AS 'State desc'
FROM sys.database_files;
GO
```

Будет получен следующий результат:

ID	Type	Description	Name	State	State desc
1	0	ROWS	MultyGroup1	0	ONLINE
2	1	LOG	MultyGroupLog1	0	ONLINE
3	0	ROWS	MultyGroup2	0	ONLINE
4	0	ROWS	MultyGroup3	0	ONLINE
5	0	ROWS	MultyGroup4	0	ONLINE
6	1	LOG	MultyGroupLog2	0	ONLINE

(6 row(s) affected)

Похожий результат можно получить, как вы помните, и при использовании системного представления `sys.master_files`.

Теперь отобразите сведения только по файловым группам, используя системное представление `sys.filegroups`, как это показано в примере 3.18.

Пример 3.18. Отображение файловых групп базы данных MultyGroup

```
USE MultyGroup;
GO
SELECT CAST(name AS CHAR(12)) AS 'Name',
       CAST(type AS CHAR(2)) AS 'Type',
       CAST(type_desc AS CHAR(16)) AS 'Descriprion',
       CAST(is_read_only AS CHAR(1)) AS 'Read-only'
  FROM sys.filegroups;
GO
```

Результат:

Name	Type	Descriprion	Read-only
PRIMARY	FG	ROWS_FILEGROUP	0
MultyGroup2	FG	ROWS_FILEGROUP	0

(2 row(s) affected)

Результат, надо сказать, малоинформативный. Здесь мы можем увидеть только имена файловых групп. Впрочем, чаще всего и этого бывает вполне достаточно.

В завершение использования утилиты командной строки `sqlcmd` мне хочется показать вам одну возможность применения параметров при выполнении в этой утилите заранее подготовленного скрипта.

Ранее мы сказали несколько слов о том, что при вызове утилиты можно не вводить в диалоговом режиме все нужные операторы, а поместить пакет соответствующих операторов в файл скрипта и при вызове утилиты указать имя этого скрипта при использовании параметра утилиты `-i`.

Скрипт может содержать параметры, значения которым присваиваются при вызове утилиты. Структура имени параметра следующая:

`$ (<имя параметра>)`

Задание значений параметрам в скрипте выполняется при использовании параметра (или, иными словами, переключателя) утилиты `-v`. Здесь после имени переключателя указывается имя параметра, знак равенства и в кавычках значение параметра.

Рассмотрим пример скрипта с параметрами. Подготовим скрипт, содержащий оператор создания простой базы данных. Имена базы данных, файлов и каталогов для хранения файлов базы данных зададим при помощи параметра `$ (DBNM)`.

Создайте, например, в Блокноте следующий скрипт:

Пример 3.19. Скрипт создания базы данных, содержащий параметр

```
USE master;
GO
IF DB_ID('$(DBNM)') IS NOT NULL
    DROP DATABASE $(DBNM);
GO
CREATE DATABASE $(DBNM)
ON PRIMARY (NAME = $(DBNM)_dat,
    FILENAME = 'D:\$(DBNM)\$(DBNM).mdf',
    SIZE = 5 MB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 1 MB)
LOG ON (NAME = $(DBNM)_log,
    FILENAME = 'D:\$(DBNM)\$(DBNM).ldf',
    SIZE = 2 MB,
    MAXSIZE = 30 MB,
    FILEGROWTH = 1 MB);
GO
SELECT CAST(file_id AS CHAR(2)) AS 'ID',
    CAST(type AS CHAR(4)) AS 'Type',
    CAST(type_desc AS CHAR(11)) AS 'Description',
    CAST(name AS CHAR (12)) AS 'Name',
    CAST(size AS CHAR(5)) AS 'Size'
FROM sys.master_files
WHERE database_id = DB_ID('$(DBNM)');
GO
```

Сохраните этот скрипт в корневом каталоге на диске D: с именем, например, CreateDBParam.sql.

Здесь параметр `$ (DBNM)` используется для задания имени базы данных, имени каталога на диске D: для размещения файлов создаваемой базы данных, для задания логических имен файла данных и файла журнала транзакций и для задания имен

файлов в операционной системе. При задании путей к файлам, логических и физических имен файлов используется конкатенация, соединение нескольких строк. Причем никаких знаков операции для конкатенации в этом случае не требуется. Параметр `$(DBNM)` просто будет заменен заданным при вызове утилиты значением.

Создайте на диске D: каталог с именем DBParam. Выполните утилиту в командной строке (предварительно завершив выполнение предыдущего сеанса утилиты, введя оператор `quit`):

```
sqlcmd -i "D:\CreateDBParam.sql" -v DBNM="DBParam"
```

В результате будет создана новая база данных. На мониторе отображается результат выполнения скрипта:

ID	Type	Description	Name	Size
1	ROWS		DBParam_dat	640
2	LOG		DBParam_log	256

```
(2 row(s) affected)
```

Посмотрите, как выполняется подстановка указанного при запуске утилиты значения параметра. Вот условный оператор в скрипте, осуществляющий проверку существования базы данных и при необходимости ее удаление:

```
IF DB_ID('$(DBNM)') IS NOT NULL
    DROP DATABASE $(DBNM);
```

В процессе выполнения скрипта этот оператор после подстановки значения параметра будет выглядеть следующим образом:

```
IF DB_ID('DBParam') IS NOT NULL
    DROP DATABASE DBParam;
```

Аналогичным образом будут выполнены подстановки значения параметра и в предложениях других операторов. Вот как будет выполнена конкатенация значения параметра со строками в предложении FILENAME, задающем имя файла данных. Исходное предложение:

```
FILENAME = 'D:\$(DBNM)\$(DBNM).mdf',
```

После подстановки значения эта строка примет вид:

```
FILENAME = 'D:\DBParam\DBParam.mdf',
```

Скрипт может содержать произвольное количество параметров. Синтаксис задания им значений при вызове утилиты является довольно свободным. Значения параметрам задаются после переключателя (иногда переключатель называют параметром — здесь не смешивайте терминологию) утилиты `-v`. Присваивание значений параметрам может следовать после этого переключателя, отделяясь друг от друга пробелами, либо перед каждым присваиванием можно указывать переключатель утилиты `-v`.

Например, если в скрипте присутствует три параметра: \$(P1), \$(P2) и \$(P3), то значения им можно задать в виде

```
sqlcmd ... -v P1="V1" P2="V2" P3="V3"
```

или в следующей форме:

```
sqlcmd ... -v P1="V1" -v P2="V2" -v P3="V3"
```

В любом случае при вызове утилиты значения должны быть заданы *всем* параметрам, присутствующим в выполняемом скрипте.

* * *

Надеюсь, вы получили соответствующее удовольствие от работы с утилитой командной строки. В дальнейшем я буду описывать работу с операторами языка Transact-SQL, не привязываясь к средствам реализации. Где вы их будете применять — это ваше решение. Совершенно одинаково (или почти одинаково) эти операторы можно выполнять как при вызове утилиты sqlcmd в командной строке или в PowerShell, так и в компоненте системы с более мощными и удобными в работе возможностями: Management Studio. В Management Studio вам, скорее всего, не потребуется выполнение преобразований CAST(), как мы это делали в предыдущих примерах, отображая сведения о базах данных и их файлах, поскольку этот компонент автоматически определяет размер каждого столбца. Если такие размеры вас не устраивают, то вы легко с помощью мыши можете их изменять.

Сейчас мы перейдем к рассмотрению средств, существующих в Management Studio.

3.4.1.4. Создание и отображение баз данных в Management Studio

Программа Management Studio является очень мощным и удобным средством работы с базами данных в SQL Server.

Запустите на выполнение Management Studio. Щелкните мышью по кнопке **Пуск**, выберите **Все программы**, щелкните по строке **Microsoft SQL Server 2012** и затем по элементу **SQL Server Management Studio**. Появится диалоговое окно соединения с сервером. В окне **Connect to Server** можно выбрать тип сервера **Server type**, имя сервера **Server name** (имя одного из экземпляров сервера, установленных на компьютере — если существует несколько серверов), вид аутентификации **Authentication** (выберите из раскрывающегося списка **Windows Authentication**) и ввести имя пользователя **User name** (рис. 3.6).

Если на компьютере установлен только один экземпляр сервера базы данных, то в этом окне нужно, ничего ни изменяя и не выбирая, просто щелкнуть по кнопке **Connect** (соединиться). Произойдет соединение с текущим экземпляром Database Engine и следом появится главное окно Management Studio, показанное на рис. 3.7.

Однако если на вашем компьютере установлено несколько серверов базы данных, то вам нужно из раскрывающегося списка **Server name** выбрать соответствующий сервер. У меня установлено два сервера, и каждый раз при запуске Management Studio я стараюсь вспомнить, для какого сервера и для каких целей я это делаю.



Рис. 3.6. Соединение с сервером

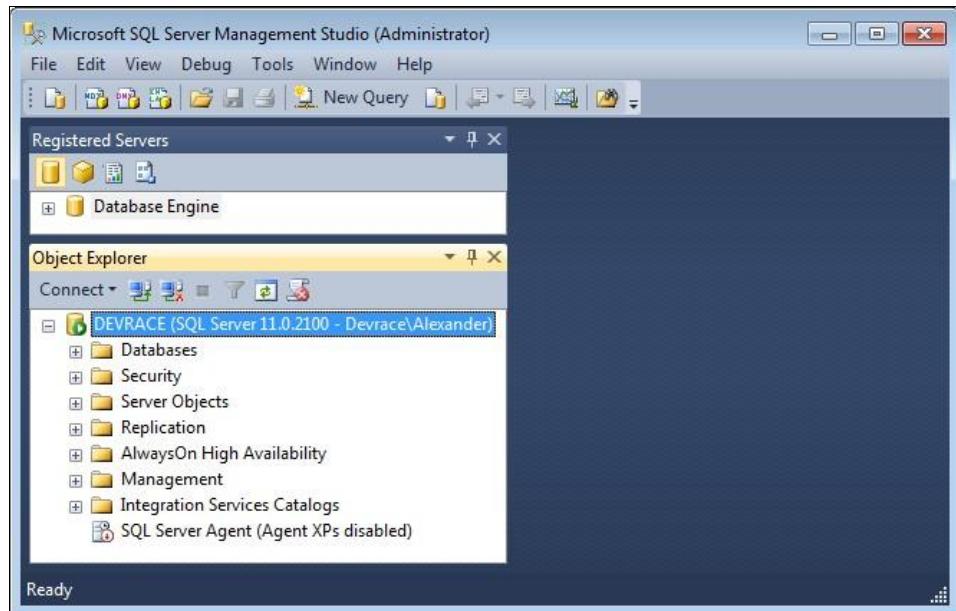


Рис. 3.7. Главное окно Management Studio. Панель Object Explorer

Если в окне не виден **Object Explorer**, выберите в меню **View | Object Explorer** или нажмите клавишу **<F8>**.

Как обычно, в верхней части окна располагается главное меню, ниже присутствует панель инструментов с кнопками быстрого доступа к функциям некоторых элементов главного меню. В левой части находится Инспектор объектов (**Object Explorer**), при помощи которого вы можете получить быстрый доступ ко многим полезным возможностям Management Studio, большинство из которых мы вскоре рассмотрим.

Чтобы выполнять в Management Studio пакеты операторов Transact-SQL, необходимо вызвать окно выполнения запросов. Для этого на панели инструментов нужно мышью щелкнуть по кнопке **New Query** (новый запрос).

Для вызова этого окна можно также щелкнуть мышью по элементу меню **File** (файл), выбрать элемент **New** (новый), а затем элемент **Query with Current Connection** (запрос в текущем соединении). В дальнейшем подобный вызов элемента в многоуровневом меню мы будем изображать в тексте чуть короче в следующем виде: "Выберите меню **File | New | Query with Current Connection**".

Наконец, для вызова окна запросов можно просто нажать клавиши <Ctrl>+<N>.

В центральной части главного окна появится пустое окно запросов, где можно вводить, изменять и выполнять операторы Transact-SQL. Это окно показано на рис. 3.8.



Рис. 3.8. Окно ввода операторов Transact-SQL

Вы можете создать произвольное количество окон запросов, в которых могут содержаться любые операторы работы с базами данных. Окна запросов создаются в виде вкладок. Переключение между этими вкладками выполняется щелчком мыши по заголовкам вкладок.

Тренировки ради предлагаю повторить некоторые скрипты, которые вы вводили в утилите командной строки чуть раньше в этой главе. Например, повторите создание базы данных SimpleDB, введя следующие операторы (пример 3.20).

Пример 3.20. Создание и отображение базы данных со значениями по умолчанию

```
USE master;
GO
IF DB_ID('SimpleDB') IS NOT NULL
    DROP DATABASE SimpleDB;
CREATE DATABASE SimpleDB;
GO
SELECT file_id AS 'ID',
       type AS 'Type',
       type_desc AS 'Description',
       name AS 'Name',
```

```

state AS 'State',
state_desc AS 'State desc',
size AS 'Size'
FROM sys.master_files
WHERE database_id = DB_ID('SimpleDB');
GO

```

Программа Management Studio при вводе операторов SQL выполняет выделение цветом ключевых слов, констант, подчеркивает красной волнистой линией неверные выражения. Иными словами, помогает вам создать правильный текст. Вы сможете выявить ошибки еще до того, как запустите скрипт на выполнение. Правда, бывают и такие случаи, когда красной волнистой линией подчеркиваются как бы ошибочные тексты, которые на самом деле созданы правильно.

Для выполнения этих введенных операторов щелкните мышью по кнопке **Execute** (выполнить) на панели инструментов, выберите в меню **Query** (запрос) | **Execute** (выполнить) либо нажмите клавишу <F5> или комбинацию клавиш <Ctrl>+<E>.

Результат выполнения пакета примера 3.20 показан на рис. 3.9. Это окно появится в нижней части окна выполнения запросов.

The screenshot shows the Management Studio interface with the 'Results' tab selected. A table is displayed with the following data:

ID	Type	Description	Name	State	State desc	Size
1	1	0	ROWS	0	ONLINE	520
2	2	1	LOG	0	ONLINE	130

Below the table, a status bar indicates: 'Query executed successfully.' followed by 'DEVRACE (11.0 RTM)', 'Devrave\Alexander (52)', 'master', '00:00:01', and '2 rows'.

Рис. 3.9. Результат создания простой базы данных

Заметьте, что в отличие от примера 3.9, где эта база данных создавалась и отображалась в командной строке, нам в данном случае при отображении результата, показанного на рис. 3.9, не нужно рассчитывать размер и количество символов, которое уместится на выходе. Нет необходимости выполнять преобразования отображаемых столбцов с использованием функции `CAST()`. В полученном окне результата мы всегда легко с помощью мыши можем уменьшить или расширить поле, отводимое для отображения любого столбца, если система не даст нам приличного варианта. Для этого нужно курсор мыши подвести к границе двух столбцов в заголовке и, нажав левую кнопку, изменить требуемый размер. Как правило, программа Management Studio с самого начала предоставляет хорошо читаемый вариант.

Вы можете заранее любыми средствами (пусть даже в программе Блокнот) создать файл скрипта, который будет содержать все необходимые операторы. Принято таким файлам давать расширение `sql`, хотя это также не является обязательным требованием к скриптам. Чтобы загрузить такой файл в окно запросов Management Studio, нужно выбрать в меню **File** | **Open** | **File**. Можно нажать клавиши <Ctrl>+<O> или щелкнуть мышью по кнопке открытия на панели инструментов.

Появится обычное окно открытия файла, в котором вы выбираете нужный для работы скрипт. Программа создаст новую вкладку и поместит туда выбранный текст.

Если вы создавали скрипт в Management Studio или вносили изменения в существующий скрипт, то вы можете сохранить эти изменения, выбрав в меню **File | Save (имя скрипта)**, нажав комбинацию клавиш **<Ctrl>+<S>** или щелкнув мышью по кнопке сохранения на инструментальной панели.

Если вы создавали новый скрипт или хотите сохранить существующий скрипт на диске с другим именем, то для этого следует выбрать в меню **File | Save [существующее имя скрипта] As ...** и в появившемся диалоговом окне сохранения файла выбрать каталог размещения и новое имя скрипта.

Программа Management Studio предоставляет еще одну удобную возможность. Если в окне существует множество операторов, а вам нужно выполнить только некоторые из них, то достаточно при помощи мыши или клавиатуры выделить требуемую последовательную группу операторов и запустить на выполнение только их. Если вы когда-либо присутствовали на мероприятиях, где специалисты Microsoft рассказывали что-нибудь интересное о возможностях системы с демонстрацией этих возможностей при использовании Management Studio, то вы, конечно же, видели, что они постоянно используют этот прием.

Теперь в Management Studio создадим базу данных **BestDatabase**, которую мы будем использовать в дальнейшей работе. Введите следующие операторы (пример 3.21).

Пример 3.21. Создание базы данных BestDatabase в Management Studio

```
USE master;
GO
IF DB_ID('BestDatabase') IS NOT NULL
    DROP DATABASE BestDatabase;
GO
CREATE DATABASE BestDatabase
ON PRIMARY (NAME = BestDatabase_dat,
    FILENAME = 'D:\BestDatabase\Winner.mdf',
    SIZE = 5 MB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 1 MB)
LOG ON (NAME = BestDatabase_log,
    FILENAME = 'D:\BestDatabase\Winner.ldf',
    SIZE = 2 MB,
    MAXSIZE = 30 MB,
    FILEGROWTH = 1 MB);
GO
```

Выполните операторы.

А для отображения созданной базы данных в этой среде в отличие от утилиты `sqlcmd` можно поступить несколько иначе, нет необходимости использовать только оператор `SELECT` и системное представление `sys.master_files`. Щелкните мышью по символу (+) слева от строки **Databases** в окне **Object Explorer**. Раскроется список баз данных, определенных в текущем экземпляре сервера базы данных (рис. 3.10). Имена баз данных в списке упорядочены по алфавиту. Только хочу напомнить, что те средства, которые вы использовали в утилите `sqlcmd` для отображения характеристик баз данных и их файлов, можно с тем же успехом использовать в Management Studio.

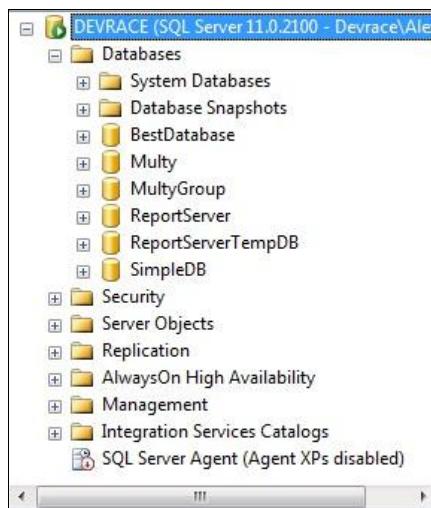


Рис. 3.10. Список баз данных в **Object Explorer**

Мы видим в этом списке и созданную только что базу данных `BestDatabase`.

ЗАМЕЧАНИЕ

Если вновь созданная база данных в окне **Object Explorer** не видна (а скорее всего так и будет сразу после создания новой базы данных), то следует обновить список объектов, щелкнув правой кнопкой мыши по строке **Databases** или по имени сервера базы данных и выбрав в появившемся контекстном меню элемент **Refresh** (обновить). Для обновления списка также можно, как и в любом другом приложении Windows, просто нажать клавишу `<F5>`. Не забудьте только в этом случае фокус перевести именно на **Object Explorer**, щелкнув мышью по заголовку этого окна, и выделить мышью строку **Databases** или строку сервера (самую первую строку в списке). Иначе у вас просто запустится на выполнение скрипт из текущего окна запросов.

Чтобы просмотреть подробнейшие сведения о только что созданной базе данных `BestDatabase` и при желании внести некоторые изменения, щелкните правой кнопкой мыши по строке **BestDatabase** и в появившемся контекстном меню выберите элемент **Properties** (свойства). Откроется окно свойств выбранной базы данных, где текущей будет вкладка **General** (общие) (рис. 3.11).

Здесь мы видим общие свойства базы данных: имя базы, дату создания, порядок сортировки, владельца и ряд других свойств. На этой вкладке не допускается внесение каких-либо изменений.

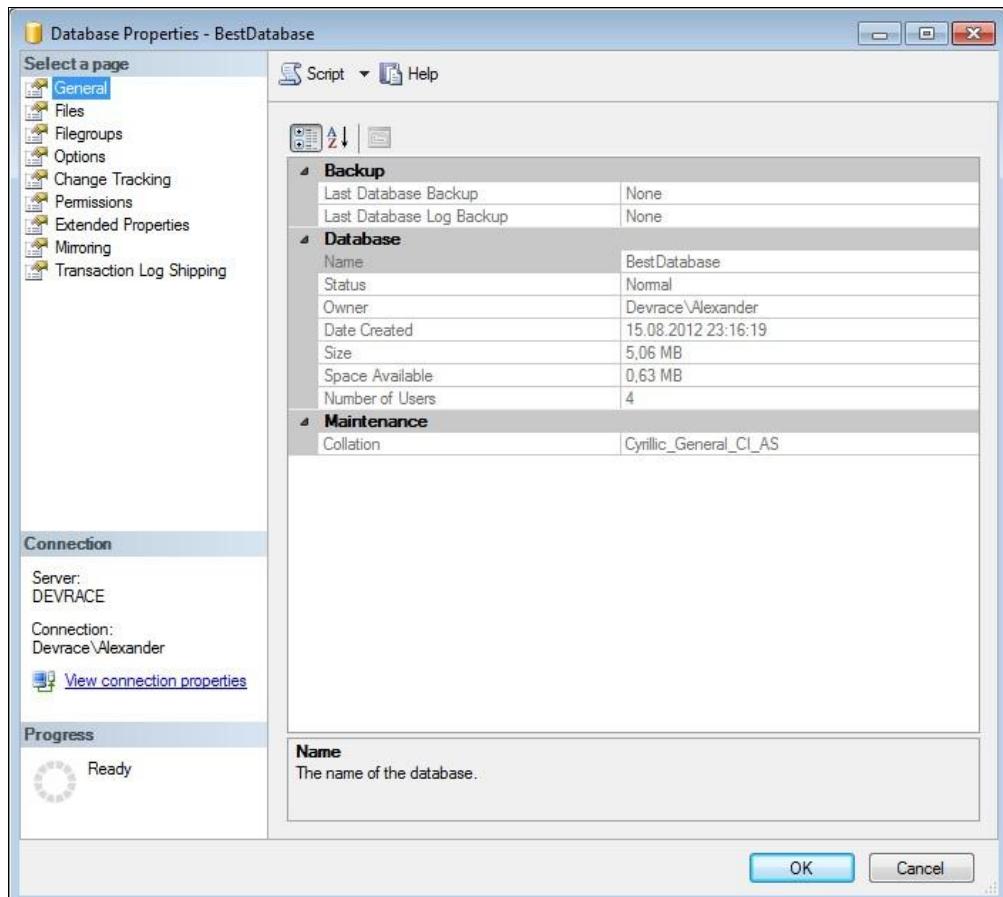


Рис. 3.11. Свойства базы данных BestDatabase. Вкладка General

В левой верхней части главного окна, в панели **Select a page** (выбор страницы) щелкните мышью по строке **Files** (файлы).

Откроется более интересная вкладка **Files**, где присутствует описание характеристик всех файлов, входящих в состав этой базы данных. Вкладка показана на рис. 3.12.

Здесь мы видим характеристики двух созданных файлов базы данных — файла данных с логическим именем `BestDatabase_dat` и файла журнала транзакций с логическим именем `BestDatabase_log`. Указаны их типы: `ROWS Data` (файл данных, словно — строки данных) и `Log` (журнал транзакций). Для них представлены начальный размер в мегабайтах, порядок увеличения размера, путь к файлу и имя файла (на этом рисунке не видно, однако можно просмотреть, воспользовавшись в нижней части окна полосой прокрутки). Более подробно отображаемые характеристики файлов при полностью развернутом окне свойств базы данных показаны на рис. 3.13.

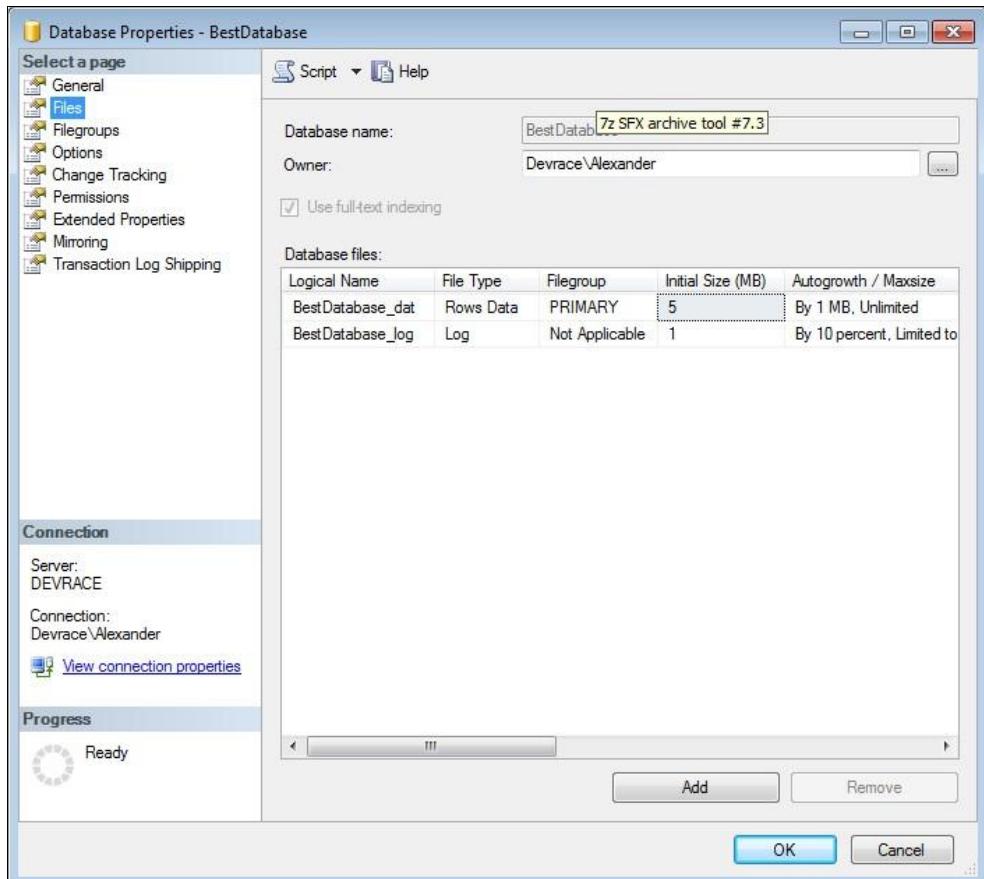


Рис. 3.12. Свойства базы данных BestDatabase. Вкладка Files

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize	Path
BestDatabase_dat	Rows Data	PRIMARY	5	By 1 MB, Unlimited	[...]
BestDatabase_log	Log	Not Applicable	1	By 10 percent, Limited to 209...	[...]

Рис. 3.13. Характеристики файлов базы данных BestDatabase

Давайте подробнее рассмотрим этот список.

В столбце **Logical Name** (логическое имя) мы видим логические имена двух созданных файлов — файла данных (`BestDatabase_dat`) и файла журнала транзакций (`BestDatabase_log`).

В столбце **File Type** (тип файла) указывается именно тип файла — файл данных (`Rows Data`) или файл журнала транзакций (`Log`).

Столбец **Filegroup** содержит имя файловой группы, которой принадлежит соответствующий файл. Для файлов данных в этом столбце указывается имя файловой группы. Здесь для первичного файла данных указано `PRIMARY`, т. е. файл относится

к первичной файловой группе. Для файлов же журнала транзакций здесь содержитя текст Not Applicable (не применимо), поскольку для файлов журнала транзакций не применяется распределение по файловым группам.

Столбец **Initial Size (MB)** указывает начальный размер файла в мегабайтах.

В столбце **Autogrowth** (автоматическое увеличение размера) описывается единица увеличения размера памяти (единица измерения или проценты от начального значения) и до какой величины может увеличиваться размер памяти, отводимой под файл, либо указывается, что размер не ограничивается (*unrestricted growth*).

Столбец **Path** (путь) содержит полный путь к файлу.

В столбце **File Name** содержится имя физического файла в операционной системе.

При выборе на панели **Select a page** вкладки **Options** (свойства) появится окно свойств базы данных (рис. 3.14).

Здесь перечисляется множество общих свойств базы данных. Эти свойства, характеристики и возможность изменения отдельных значений мы рассмотрим чуть позже в этой главе.

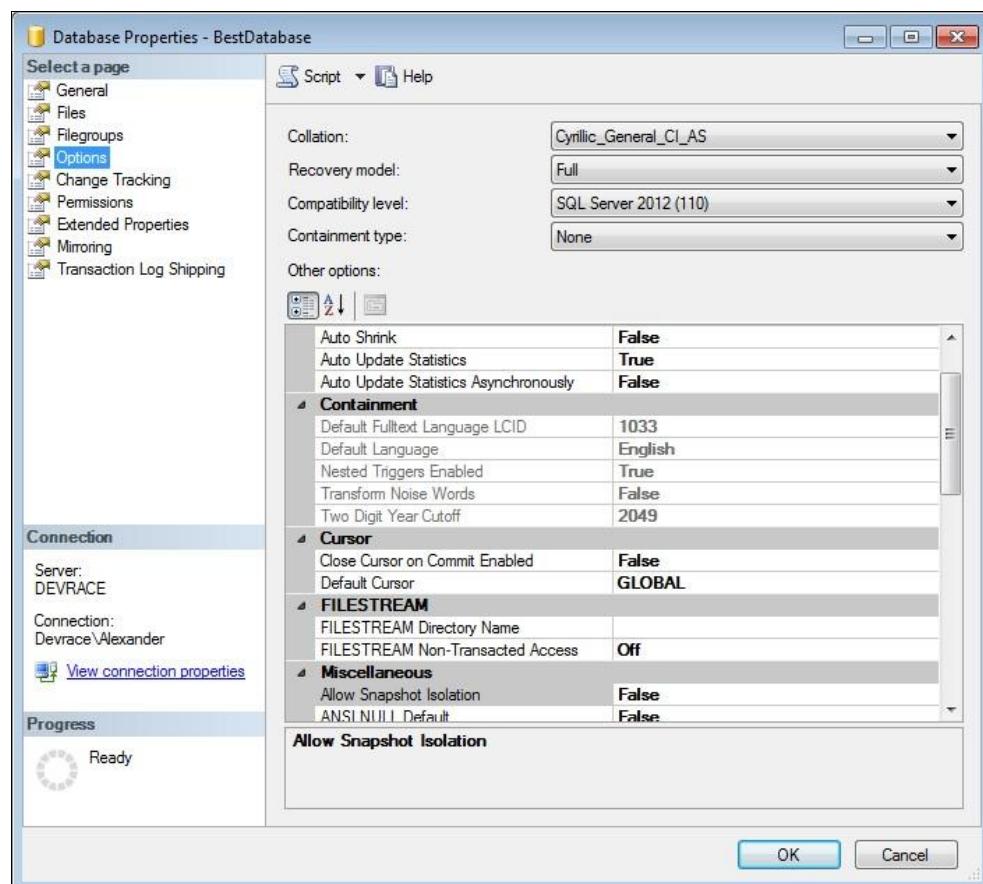


Рис. 3.14. Свойства базы данных BestDatabase. Вкладка Options

Теперь посмотрим на характеристики базы данных **MultyGroup**, которая содержит не одну, а две файловые группы.

Закройте диалоговое окно просмотра свойств базы данных **BestDatabase**, щелкнув мышью по кнопке **Cancel** в любой вкладке этого окна. Щелкните правой кнопкой мыши по имени базы данных **MultyGroup** в **Object Explorer** и в появившемся контекстном меню выберите строку **Properties**. Откроется окно просмотра свойств этой базы данных.

Выберите вкладку **Files** для просмотра файлов базы данных. Список файлов показан на рис. 3.15.

Database files:						
Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize	Path	File Name
MultyGroup1	Rows Data	PRIMARY	5	By 1 MB, Unlimited	[...]	D:\MultyGroup\ MultyGroup1.mdf
MultyGroup3	Rows Data	MultyGroup2	1	By 1 MB, Unlimited	[...]	D:\MultyGroup\ MultyGroup3.ndf
MultyGroup4	Rows Data	MultyGroup2	1	By 1 MB, Unlimited	[...]	D:\MultyGroup\ MultyGroup4.ndf
MultyGroup2	Rows Data	PRIMARY	1	By 1 MB, Unlimited	[...]	D:\MultyGroup\ MultyGroup2.ndf
MultyGroupLog1	Log	Not Applicable	1	By 10 percent, Limited to 209...	[...]	D:\MultyGroup\ MultyGroupLog1.ldf
MultyGroupLog2	Log	Not Applicable	1	By 10 percent, Limited to 209...	[...]	D:\MultyGroup\ MultyGroupLog2.ldf

Рис. 3.15. Список файлов базы данных **MultyGroup**

Главное, что здесь можно увидеть интересного в отличие от списка файлов базы данных **BestDatabase**, это распределение файлов по файловым группам. Мы видим, что файлы данных **MultyGroup1** и **MultyGroup2** принадлежат первичной файловой группе **PRIMARY**, а вторичной файловой группе **MultyGroup2** принадлежат файлы данных **MultyGroup3** и **MultyGroup4**.

Теперь на панели **Select a page** выберите вкладку **Filegroups** (файловые группы). Данные на этой вкладке весьма скромные. Впрочем, на большее и рассчитывать-то не стоило. Что особенного можно сказать про файловые группы?

В верхней части окна указаны обе файловые группы этой базы данных и представлено количество файлов данных, входящих в каждую группу. Для файловой группы **PRIMARY** стоит отметка в столбце **Default**, это означает, что она является первичной файловой группой, файловой группой по умолчанию.

К вкладкам свойств базы данных мы будем неоднократно возвращаться, в том числе и в этой главе, когда станем изменять характеристики существующих баз данных.

Сейчас же рассмотрим диалоговые средства **Management Studio**, используемые для создания новых баз данных. Закройте диалоговое окно просмотра свойств базы данных, щелкнув по кнопке **Cancel**.

Замечания по использованию **Management Studio**

В **Management Studio** есть очень удобная возможность в окне **New Query** (создать запрос) вводить произвольное количество операторов, а для выполнения одного оператора или группы операторов нужно выделить необходимую группу строк и нажать клавишу **<F5>** или комбинацию клавиш **<Ctrl>+<E>**.

Для упрощения получения нужных результатов в подходящем виде в Management Studio есть еще ряд полезных возможностей. Мы выводили все результаты выполнения запросов в окне **New Query** в табличном виде или в так называемую сетку (Grid). Существует возможность выводить результаты в текстовом виде, практически так же, как они отображаются в утилите sqlcmd при использовании командной строки (PowerShell). Для этого перед выполнением группы операторов нужно в меню выбрать **Query | Results To | Results to Text** или нажать клавиши <Ctrl>+<T>.

Чтобы вернуться к более привычной форме отображения результатов в табличном виде, нужно выбрать в меню **Query | Results To | Results to Grid** или нажать клавиши <Ctrl>+<D>.

Кроме этих возможностей в Management Studio существует еще множество дополнительных настроек. Для получения доступа к многочисленным настройкам выберите в меню **Query | Query Options** или щелкните мышью по кнопке **Query Options** на инструментальной панели. Появится окно **Query Options**, в котором на нескольких вкладках можно установить большое количество характеристик, используемых при выполнении программы. Например, на вкладке **Results** (результаты) можно указать необходимость включения текста запроса в результат его выполнения, можно задать отображение результата выполнения запроса в отдельной таблице. Вместо оператора `GO`, разделяющего выполняемые фрагменты пакета, на вкладке **General** можно указать другой оператор, просто введя его текст в соответствующее поле, чего, конечно же, никак нельзя порекомендовать.

Можно указать, что в окне запроса должны отображаться номера строк. Для этого нужно выбрать в меню **Tools | Options**. Появится диалоговое окно задания режимов. В левой части окна нужно раскрыть элемент **Transact-SQL** и щелкнуть мышью по элементу **General**. В правой части окна в разделе **Display** нужно установить флажок **Line numbers** и щелкнуть по кнопке **OK**.

После этой установки в левой части окна запроса будут выводиться номера строк. Это полезная возможность, особенно тогда, когда вы отлаживаете достаточно "длинный" запрос, содержащий большое количество строк, например, добавление многих строк в таблицы базы данных. В случае ошибок система выдает соответствующее сообщение с указанием номера строки и номера позиции, где была обнаружена ошибка.

Если вы вводите неправильную конструкцию, то система подчеркивает неверные символы красной волнистой линией. Подведя к ошибочному тексту курсор мыши, вы получите краткую подсказку, что именно не так вы сделали в операторе. Правда, бывают случаи, когда подчеркиваются и совершенно верные тексты. Но это случается достаточно редко.

3.4.2. Создание базы данных с использованием диалоговых средств Management Studio

Базу данных можно создавать не только с использованием оператора `CREATE DATABASE`. В Management Studio существуют диалоговые средства, позволяющие задать все необходимые характеристики создаваемой базы данных.

Щелкните правой кнопкой мыши в **Object Explorer** по строке **Databases** и в появившемся контекстном меню выберите элемент **New Database** (можно также щелкнуть правой кнопкой мыши по имени любой пользовательской базы данных и выбрать строку **New Database**). Появится окно создания новой базы данных **New Database** (рис. 3.16), где можно создавать базу данных, ее файлы со всеми необходимыми характеристиками. Текущей будет вкладка **General**.

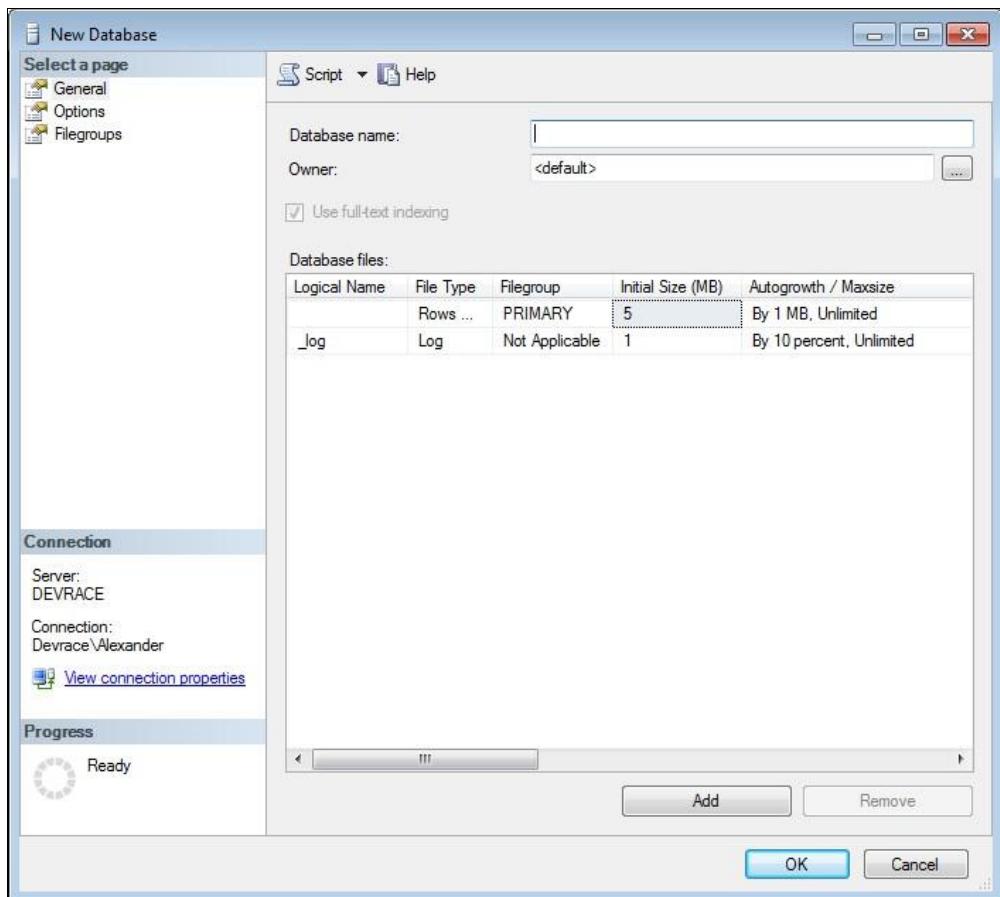


Рис. 3.16. Окно создания новой базы данных **New Database**

Вначале зададим имя базы данных. Пусть это будет база данных с простым и понятным именем **NewDatabase**. Введите это имя в поле **Database name** в верхней части окна. В процессе ввода имени в строках столбца **Logical Name** появляются соответствующие имена **NewDatabase** (файл данных) и **NewDatabase_log** (журнал транзакций). Оставим все как есть, только к имени файла данных добавим еще суффикс **_dat**.

В столбце **Path** (пути к файлам базы данных) программа предлагает нам некоторый вариант размещения файлов создаваемой базы данных. В правой части строки в этом поле присутствует кнопка с многоточием. Для выбора другого размещения

файлов щелкните по этой кнопке. Откроется окно выбора папки — **Locate Folder** (рис. 3.17). Выберите по очереди для каждого из файлов тот же самый каталог, что мы использовали для базы данных BestDatabase: на диске D:\BestDatabase.

Щелкните в этом окне по кнопке **OK**.



Рис. 3.17. Выбор папки для размещения файла базы данных

Каталог для размещения файлов можно задать и путем ввода в поля столбца **Path** нужного пути к файлам, просто набрав с клавиатуры текст D:\BestDatabase.

В столбце **File Name** введите имена файлов, соответственно для файла данных и файла журнала транзакций: NewDatabase.mdf и NewDatabase.ldf. В столбце **Logical Name** введите логические имена файлов: NewDatabase_dat и NewDatabase_log.

Список файлов будет выглядеть так, как на рис. 3.18.

Database files:						
Logical Name	File Type	Filegroup	Initial ...	Autogrowth / Maxsize	Path	File Name
NewDatabase_dat	Rows Data	PRIMARY	5	By 1 MB, Unlimited	<input type="button" value="..."/> D:\BestDatabase	<input type="button" value="..."/> NewDatabase.mdf
NewDatabase_log	Log	Not Applicable	1	By 10 percent, Unlimited	<input type="button" value="..."/> D:\BestDatabase	<input type="button" value="..."/> NewDatabase.ldf

Рис. 3.18. Файлы создаваемой базы данных

Для создания этой базы данных нужно щелкнуть по кнопке **OK**, однако давайте пока повременим с этим и рассмотрим еще возможности добавления вторичных файловых групп и новых файлов.

Если для создаваемой базы данных вам нужно задать несколько файлов данных и/или несколько файлов журнала транзакций, то вы можете щелкнуть мышью по кнопке **Add** в нижней части окна и добавить любое количество файлов данных или файлов журналов транзакций.

Для этой базы данных создадим еще один файл данных с именем NewDatabase2. Щелкните по кнопке **Add**. В окне появится новая строка с заданными значениями некоторых характеристик. В качестве логического имени нового файла введите NewDatabase2, выберите или введите с клавиатуры тот же путь к файлу (**Path**), что и для других файлов этой базы данных, задайте для него и имя физического файла NewDatabase2.ndf.

Тип файла (**File Type**) по умолчанию устанавливается как файл данных (**Rows Data**). Если вы собираетесь создавать файл журнала транзакций, то нужно в поле **File Type** щелкнуть мышью справа от значения этого поля со стрелкой вниз.

Появится раскрывающийся список, в котором вы в этом случае должны были бы выбрать значение `Log`. Сейчас оставим для типа файла значение по умолчанию.

Аналогичным образом вы можете выбрать файловую группу, которой будет принадлежать создаваемый файл — разумеется, только если вы создаете файл данных. Для этого нужно щелкнуть по кнопке со стрелкой вниз в правой части поля **Filegroup** и из раскрывающегося списка выбрать нужную файловую группу. Поскольку никаких вторичных файловых групп мы пока не создавали, список будет содержать только две строки: `PRIMARY` и `<new filegroup>` (новая файловая группа).

Создадим новую файловую группу этим способом. Другой путь создания новой файловой группы мы используем при внесении изменения в эту базу данных.

Выберите в этом раскрывающемся списке строку `<new filegroup>`. Появится окно создания новой файловой группы **New Filegroup**. Введите в поле **Name** имя файловой группы, которую также без затей и назовем: `NewGroup` (рис. 3.19).

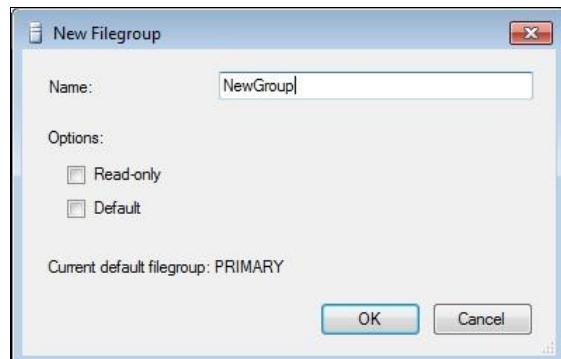


Рис. 3.19. Создание новой файловой группы

Щелкните по кнопке **OK**. Будет создана новая файловая группа, которой будет принадлежать и вновь создаваемый файл `NewDatabase2`.

Установите для файла начальный размер (столбец **Initial Size**) в 2 Мбайта, введя с клавиатуры число 2 в соответствующей строке этого столбца или установив значение, используя кнопку с изображением стрелки вверх (для увеличения значения на единицу) или стрелки вниз (для уменьшения значения на единицу), как это показано на рис. 3.20.



Рис. 3.20. Задание начального размера файла

Для задания характеристик роста размера файла щелкните мышью по кнопке с многоточием в правой части строки столбца **Autogrowth**. Появится диалоговое окно **Change Autogrowth**. В этом окне установите следующие характеристики.

- ◆ В группе переключателей **File Growth** (увеличение размера файла) выберите **In Megabytes** (в мегабайтах), чтобы приращение указывалось в мегабайтах, и за-

дайте величину приращения 2, введя это значение вручную или с использованием кнопок с изображением стрелки вверх или стрелки вниз в правой части соответствующего счетчика.

- ◆ В группе переключателей **Maximum File Size** (максимальный размер файла) выберите **Limited to (МВ)** (ограничение на рост размера файла). Укажите максимальный размер файла в 20 Мбайт, введя значение в счетчик вручную или используя кнопки с изображением стрелки вверх и стрелки вниз.

Выбранные значения показаны на рис. 3.21.

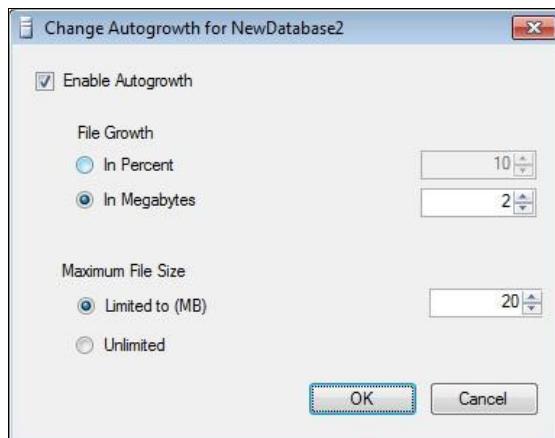


Рис. 3.21. Изменение характеристик увеличения размера файла

Щелкните мышью по кнопке **OK**. Будут сформированы все заданные значения для второго файла данных. Список файлов создаваемой базы данных теперь будет выглядеть, как показано на рис. 3.22.

Для завершения создания базы данных щелкните по кнопке **OK**.

Database files:							
Logical Name	File Type	Filegroup	Initial ...	Autogrowth / Maxsize	Path	File Name	
NewDatabase_dat	Rows Data	PRIMARY	5	By 1 MB, Unlimited	[...]	D:\BestDatabase	[...] NewDatabase.mdf
NewDatabase_log	Log	Not Applicable	1	By 10 percent, Unlimited	[...]	D:\BestDatabase	[...] NewDatabase.ldf
NewDatabase2	Rows Data	NewGroup	2	By 2 MB, Limited to 20 MB	[...]	D:\BestDatabase	[...] NewDatabase2.ndf

Рис. 3.22. Новый список файлов базы данных

3.5. Изменение базы данных

Изменять характеристики существующей в экземпляре сервера пользовательской базы данных можно оператором Transact-SQL `ALTER DATABASE` или при использовании диалоговых средств Management Studio. Рассмотрим оба варианта.

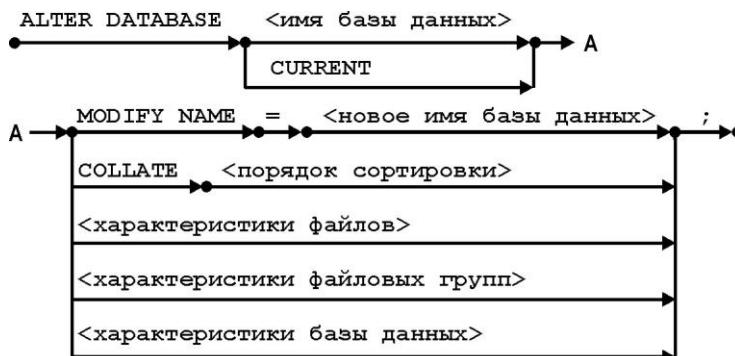
Следует помнить, что вносить изменения можно в ту базу данных, с которой в этот момент не соединены другие пользователи.

3.5.1. Изменение базы данных в языке Transact-SQL

Для изменения существующей пользовательской базы данных в языке Transact-SQL используется оператор ALTER DATABASE. Его синтаксис представлен в листинге 3.5 и соответствующем R-графе (граф 3.9).

Листинг 3.5. Синтаксис оператора ALTER DATABASE

```
ALTER DATABASE { <имя базы данных> | CURRENT }
{ MODIFY NAME = <новое имя базы данных>
| COLLATE <порядок сортировки>
| <характеристики файлов>
| <характеристики файловых групп>
| <характеристики базы данных>
};
```



Граф 3.9. Синтаксис оператора ALTER DATABASE

Изменения будут выполняться либо для базы данных, указанной в операторе по имени, либо для текущей базы данных, заданной в последнем операторе USE, в случае указания ключевого слова CURRENT.

Как видно из синтаксиса, за одно выполнение оператора можно либо переименовать базу данных, либо изменить порядок сортировки, либо изменить характеристики файлов, файловых групп или всей базы данных. Для выполнения нескольких действий нужно использовать соответствующее количество операторов ALTER DATABASE.

3.5.1.1. Изменение имени базы данных

Самое простое действие — изменение имени базы данных. Для этого используется предложение MODIFY NAME, где указывается новое имя базы данных. Это имя должно быть, естественно, уникальным в данном экземпляре сервера.

Измените имя базы данных BestDatabase на NewBest. Выполните операторы из примера 3.22 в утилите sqlcmd или в окне выполнения запросов Management Studio.

Пример 3.22. Изменение имени базы данных BestDatabase

```
USE master;
GO
ALTER DATABASE BestDatabase
    MODIFY NAME = NewBest;
GO
```

В строке утилиты `sqlcmd` или на вкладке **Messages** компонента Management Studio (в зависимости от того, каким средством вы сейчас пользовались) появится сообщение об изменении имени:

The database name 'NewBest' has been set.

Отобразите список имен и порядков сортировки баз данных (пример 3.23).

Пример 3.23. Отображение списка баз данных и порядков сортировки

```
SELECT CAST(name AS CHAR(20)) AS 'NAME',
       CAST(collation_name AS CHAR(23)) AS 'COLLATION'
FROM sys.databases;
GO
```

Видно, что имя базы данных было изменено. Верните нашей базе данных ее родное имя `BestDatabase`, выполнив операторы:

```
USE master;
GO
ALTER DATABASE NewBest
    MODIFY NAME = BestDatabase;
GO
```

3.5.1.2. Изменение порядка сортировки

Для базы данных также весьма просто можно изменить порядок сортировки в операторе `ALTER DATABASE`. Выполните пакет из примера 3.24 (одна моя знакомая программерша уехала во Францию и перед отъездом проявила живейший интерес к порядку сортировки из данного примера).

Пример 3.24. Изменение порядка сортировки базы данных BestDatabase

```
USE master;
GO
ALTER DATABASE BestDatabase
    COLLATE French_CI_AI;
GO
```

В текущем состоянии базы данных `BestDatabase` все пройдет нормально, однако если вы создавали в этой базе данных таблицы (что мы выполним в одной из сле-

дующих глав), то можете получить сообщения об ошибке. В частности, изменение порядка сортировки невозможно, если отдельные строковые столбцы, для которых не задана сортировка, отличная от сортировки по умолчанию, присутствуют в ограничениях CHECK или включены в состав вычисляемых столбцов.

Если изменение существующего порядка сортировки по умолчанию возможно для базы данных, то новое значение будет влиять только лишь на вновь создаваемые столбцы существующих или новых таблиц. Чтобы сохранить существующие данные, потребуется выполнить действия по загрузке/выгрузке данных. Если изменения касаются столбцов, входящих в состав индексов, то нужно будет и перестроить соответствующие индексы.

Отобразите опять список баз данных (см. пример 3.23) и убедитесь, что порядок сортировки нашей базы данных изменился.

При помощи оператора ALTER DATABASE верните порядок сортировки базы данных BestDatabase к значению по умолчанию, принятому в текущем экземпляре сервера — Cyrillic_General_CI_AS.

Пример 3.25. Обратное изменение порядка сортировки базы данных BestDatabase

```
USE master;
GO
ALTER DATABASE BestDatabase
    COLLATE Cyrillic_General_CI_AS;
GO
```

Чтобы просмотреть все существующие в текущем экземпляре сервера базы данных порядки сортировки, используйте функцию fn_helpcollations(). Для каждого порядка сортировки функция возвращает два столбца: Name, содержащий имя порядка сортировки, и Description, в котором хранится текстовое описание.

Для получения списка порядков сортировки выполните следующий оператор:

```
SELECT Name, Description FROM fn_helpcollations();
```

Вы получите более двух тысяч строк. Чтобы выделить из списка только те порядки сортировки, которые связаны с кириллицей, добавьте в оператор SELECT предложение WHERE (пример 3.26).

Пример 3.26. Отображение списка порядков сортировки системы

```
USE master;
GO
SELECT Name, Description FROM fn_helpcollations()
    WHERE name LIKE 'Cyrillic%';
GO
```

Здесь в результирующий набор данных попадут лишь те порядки сортировки, имеющие на которых начинаются с символов 'Cyrillic'. На момент написания данной главы это будут 52 строки:

```
Cyrillic_General_BIN  
Cyrillic_General_BIN2  
Cyrillic_General_CI_AI  
Cyrillic_General_CI_AI_WS  
Cyrillic_General_CI_AI_KS  
Cyrillic_General_CI_AI_KS_WS  
Cyrillic_General_CI_AS  
Cyrillic_General_CI_AS_WS  
Cyrillic_General_CI_AS_KS  
Cyrillic_General_CI_AS_KS_WS  
...
```

В поле `Description` обычным текстом (разумеется, на английском языке) описываются характеристики порядка сортировки, в первую очередь, чувствительность к регистру. Подробнее возможные конструкции в предложении `WHERE` оператора `SELECT` мы будем рассматривать позже. В следующей главе при рассмотрении строковых типов данных мы исследуем их чувствительность к регистру.

Чтобы завершить обсуждение вопросов сортировки, давайте еще кратко рассмотрим функцию `COLLATIONPROPERTY()`. Ее синтаксис следующий:

```
COLLATIONPROPERTY(<имя порядка сортировки>, <свойство>)
```

Первый передаваемый функции параметр — имя порядка сортировки. Второй параметр — интересующее нас свойство. Функция возвращает значение требуемого свойства указанного порядка сортировки. Если функции в качестве параметра будет передано неверное имя порядка сортировки или имя свойства, не предусмотренное в функции, то она вернет значение `NULL`.

Именами свойств являются следующие:

- ◆ `CodePage` — кодовая страница порядка сортировки;
- ◆ `LCID` — код языка в Windows;
- ◆ `ComparisonStyle` — стиль сравнения Windows;
- ◆ `Version` — версия порядка сортировки. Число 0, 1 или 2.

Например, чтобы получить значение кодовой страницы и кода языка для порядка сортировки `Cyrillic_General_CI_AS`, нужно выполнить следующий оператор `SELECT` (пример 3.27).

Пример 3.27. Отображение характеристик порядка сортировки

```
USE master;  
GO  
SELECT COLLATIONPROPERTY('Cyrillic_General_CI_AS', 'CodePage'),  
       COLLATIONPROPERTY('Cyrillic_General_CI_AS', 'LCID');  
GO
```

Результатом будут два значения: 1251 (кодовая страница) и 1049 (код языка).

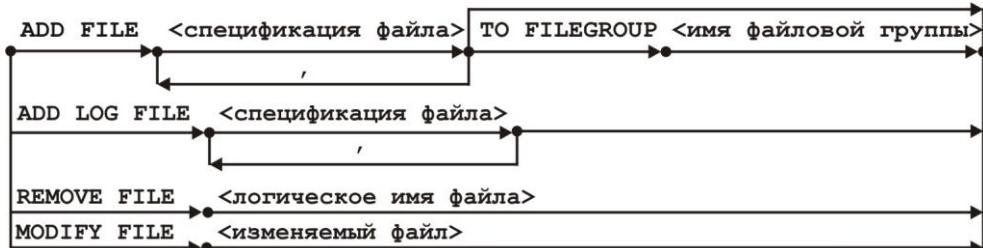
Если есть желание и необходимость, можете поэкспериментировать и с другими порядками сортировки.

3.5.1.3. Изменение файлов базы данных

В общем синтаксисе оператора `ALTER DATABASE` (см. листинг 3.5) указана конструкция "характеристики файлов". Несколько упрощенный синтаксис этой конструкции представлен в листинге 3.6 и соответствующем R-графе (граф 3.10).

Листинг 3.6. Синтаксис предложения изменения файлов базы данных в операторе `ALTER DATABASE`

```
<характеристики файлов> ::=  
{ ADD FILE <спецификация файла> [, <спецификация файла>  
      [ TO FILEGROUP <имя файловой группы> ]  
    | ADD LOG FILE <спецификация файла> [, <спецификация файла>]  
    | REMOVE FILE <логическое имя файла>  
    | MODIFY FILE <изменяемый файл>  
}
```



Граф 3.10. Синтаксис изменения файлов базы данных

Видно, что в одном операторе можно выполнить только одно из действий — добавление, изменение или удаление файла базы данных. Для выполнения множества изменений нужно каждый раз выполнять отдельный оператор `ALTER DATABASE`.

Добавление нового файла

В существующую базу данных мы можем добавлять файлы данных и файлы журнала транзакций. Для этого используются предложения `ADD FILE` и `ADD LOG FILE` соответственно. Синтаксис спецификации добавляемого файла данных или файла журнала транзакций в точности соответствует синтаксису в операторе `CREATE DATABASE`. Чтобы вам лишний раз не перелистывать книгу в поисках описания, я здесь повторю этот синтаксис:

```
<спецификация файла> ::=  
( NAME = <логическое имя файла>,  
  FILENAME = { '<путь к файлу>' | '<путь к файловому потоку>' }  
  [, SIZE = <целое1> [ KB | MB | GB | TB ] ]
```

```
[, MAXSIZE = { <целое2> [ KB | MB | GB | TB ] | UNLIMITED } ]
[, FILEGROWTH = <целое3> [ KB | MB | GB | TB | % ] ]
)
```

Поскольку мы с вами прекрасно умеем создавать новую базу со всеми ее файлами, то здесь нам все понятно.

Удаление существующего файла

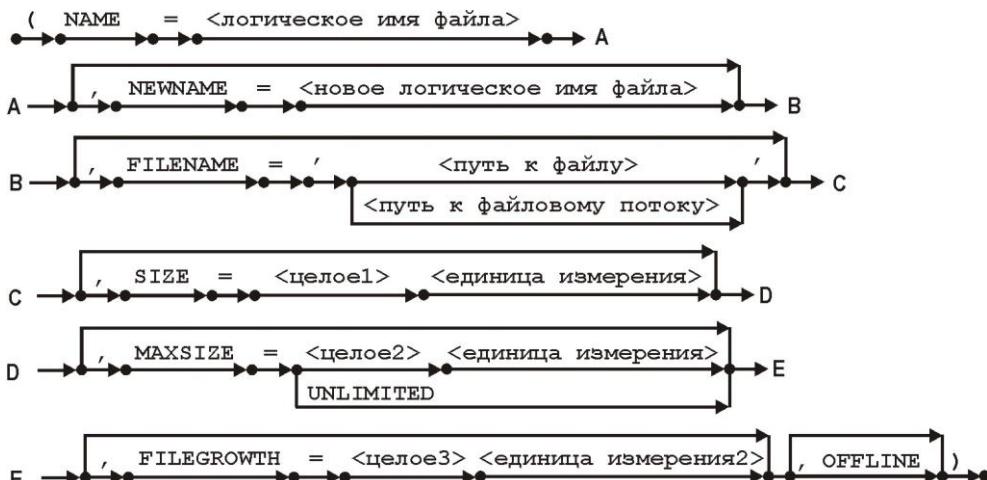
Чтобы удалить существующий в базе файл данных или файл журнала транзакций, используется предложение `REMOVE FILE`, в котором нужно указать логическое имя файла. Нельзя удалить единственный в базе данных файл данных или единственный файл журнала транзакций. Нельзя также удалить файл, содержащий данные.

Изменение характеристик файла

Для переименования или изменения характеристик существующего файла данных или файла журнала транзакций используется предложение `MODIFY FILE`. Синтаксическая конструкция "изменяемый файл" в этом предложении очень похожа на обычную спецификацию файла (см. листинг 3.7 и граф 3.11).

Листинг 3.7. Синтаксис задания изменяемого файла

```
<изменяемый файл> ::=  
( NAME = <логическое имя файла>  
[ , NEWNAME = <новое логическое имя файла> ]  
[ , FILENAME = { '<путь к файлу>' | '<путь к файловому потоку>' } ]  
[ , SIZE = <целое1> [ KB | MB | GB | TB ] ]  
[ , MAXSIZE = { <целое2> [ KB | MB | GB | TB ] | UNLIMITED } ]  
[ , FILEGROWTH = <целое3> [ KB | MB | GB | TB | % ] ]  
[ , OFFLINE ]  
)
```



Граф 3.11. Синтаксис задания изменяемого файла

ЗАМЕЧАНИЕ

Последнюю опцию в этом фрагменте синтаксиса, OFFLINE, следует использовать только в случае разрушения файла. Перевести обратно в активное состояние такой файл можно лишь при восстановлении файла из резервной копии.

NAME задает логическое имя изменяемого файла.

Вариант NEWNAME позволяет задать новое логическое имя для файла. Новое имя должно быть уникальным в этой базе данных.

Здесь также можно изменить значение начального размера файла, максимального размера, а также величины приращения.

Можно переместить файл в другое место на внешнем носителе, задав новый путь в варианте FILENAME. Можно переименовать файл в том же самом каталоге или переместив его в любой другой каталог и на другой внешний носитель. Однако не так все просто. Чтобы переименовать файл данных или файл журнала транзакций и/или переместить его в другой каталог или на другой носитель, нужно выполнить ряд действий, и не только с использованием операторов Transact-SQL.

Рассмотрим на примере нашей базы данных BestDatabase переименование файла данных, скажем, в WinnerNew.mdf. Вначале нужно будет перевести базу данных в неактивное состояние OFFLINE. Затем нужно выполнить оператор ALTER DATABASE для переименования файла, оставив его в том же каталоге. Это можно сделать в том же пакете. Выполните следующие операторы (пример 3.28).

Пример 3.28. Переименование файла данных базы данных BestDatabase

```
USE master;
GO
ALTER DATABASE BestDatabase SET OFFLINE;
ALTER DATABASE BestDatabase
MODIFY FILE (NAME = BestDatabase_dat,
    FILENAME = 'D:\BestDatabase\WinnerNew.mdf');
GO
```

Мы получим следующее информационное сообщение системы:

```
The file "BestDatabase_dat" has been modified in the system catalog.
The new path will be used the next time the database is started.
```

Имя нашего файла было изменено в системном каталоге, а новый путь будет использован только после "перезапуска" нашей базы данных, т. е. после перевода ее опять в активное состояние.

Оператор ALTER DATABASE переименовывает файл базы данных *только в каталоге* сервера базы данных, но не на внешнем носителе. Средствами операционной системы, например, при помощи программы Компьютер переименуйте файл. После этого базу данных можно перевести в оперативное состояние ONLINE, выполнив следующий оператор:

```
ALTER DATABASE BestDatabase SET ONLINE;
```

Аналогичным образом выполняется и перемещение файла на другой носитель или в другой каталог. Вначале база данных переводится в состояние OFFLINE, затем при помощи оператора ALTER DATABASE изменяется путь к файлу. Любыми доступными средствами вы переписываете файл на нужный новый носитель в указанный каталог. После этого переводите базу данных в состояние ONLINE.

За одно выполнение оператора ALTER DATABASE можно изменить имя или положение только одного файла базы данных. Если, например, вы хотите переименовать и файл данных, и файл журнала транзакций, то вам нужно дважды выполнить оператор ALTER DATABASE (не считая оператор перевода базы данных в неактивное состояние). Выполните пакет операторов в примере 3.29.

Пример 3.29. Переименование обоих файлов базы данных BestDatabase

```
USE master;
GO
ALTER DATABASE BestDatabase SET OFFLINE;
ALTER DATABASE BestDatabase
MODIFY FILE (NAME = BestDatabase_dat,
  FILENAME = 'D:\BestDatabase\WinnerNew.mdf');
ALTER DATABASE BestDatabase
MODIFY FILE (NAME = BestDatabase_log,
  FILENAME = 'D:\BestDatabase\WinnerNew.ldf');
GO
```

Не забудьте после переименования файлов на внешнем носителе вернуть базу данных в состояние ONLINE.

Теперь к нашей базе данных добавим еще два файла — файл данных и файл журнала транзакций. Выполните операторы, записанные в примере 3.30.

Пример 3.30. Добавление файлов к базе данных

```
USE master;
GO
ALTER DATABASE BestDatabase
ADD FILE (NAME = BestDatabase_dat2,
  FILENAME = 'D:\BestDatabase\Winner2.mdf',
  SIZE = 5 MB,
  MAXSIZE = UNLIMITED,
  FILEGROWTH = 1 MB);
ALTER DATABASE BestDatabase
ADD LOG FILE (NAME = BestDatabase_log2,
  FILENAME = 'D:\BestDatabase\Winner2.ldf',
  SIZE = 2 MB,
  MAXSIZE = 30 MB,
  FILEGROWTH = 1 MB);
GO
```

Проверьте результат выполнения пакета. Вы можете увидеть, что к базе данных добавлены два новых файла, а на внешнем носителе созданы файлы с соответствующими характеристиками.

Рассмотрим еще пример, в котором изменяется начальный и максимальный размер первого файла данных. Оператор представлен в листинге примера 3.31.

Пример 3.31. Изменение размера файла данных

```
USE master;
GO
ALTER DATABASE BestDatabase
MODIFY FILE (NAME = BestDatabase_dat,
    SIZE = 100 MB,
    MAXSIZE = 1000 MB);
GO
```

Новое значение начального размера файла не должно быть равным или меньшим, чем текущее значение. Что любопытно — система не проверяет, является ли максимальное значение размера большим, чем начальное.

Выполним удаление файлов базы данных. Попробуйте удалить первичный файл данных (пример 3.32).

Пример 3.32. Попытка удаления первичного файла данных

```
USE master;
GO
ALTER DATABASE BestDatabase
REMOVE FILE BestDatabase_dat;
GO
```

Вы получите следующее сообщение:

The primary data or log file cannot be removed from a database.

Нам напоминают, что из базы данных нельзя удалить первичный файл данных или файл журнала транзакций.

А вот вторичные файлы, если они не заполнены данными, можно легко удалить (пример 3.33).

Пример 3.33. Удаление файла данных

```
USE master;
GO
ALTER DATABASE BestDatabase
REMOVE FILE BestDatabase_dat2;
GO
```

Эта операция проходит нормально. Система удаляет описание логического файла из системного каталога и также удаляет физический файл с внешнего носителя.

* * *

Подведем маленький итог по работе с файлами базы данных.

Когда мы создаем базу данных или добавляем в существующую базу новые файлы, система не только добавляет сведения о файлах в системный каталог, но и физически создает файлы с заданными характеристиками на внешних носителях в указанных каталогах. Каталоги в пути к файлам должны уже существовать на диске.

Аналогично, при удалении какого-либо файла из базы система корректирует системный каталог и физически удаляет с внешнего носителя соответствующий файл.

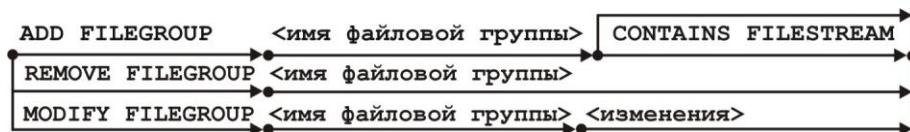
Однако если вы изменяете имя любого файла базы данных или перемещаете его в другой каталог или на другой внешний носитель, вам, во-первых, нужно перевести базу данных в состояние `OFFLINE` и, во-вторых, требуется физически средствами операционной системы изменить имя или переместить файл в другое место. После этого нужно перевести базу данных в состояние `ONLINE`.

3.5.1.4. Изменение файловых групп

В синтаксисе оператора `ALTER DATABASE` (см. ранее листинг 3.5) указана конструкция "характеристики файловых групп". Синтаксис, несколько упрощенный, этой конструкции представлен в листинге 3.8 и в R-графах (графы 3.12 и 3.13).

Листинг 3.8. Синтаксис изменения файловых групп

```
<характеристики файловых групп> ::==
{   ADD FILEGROUP <имя файловой группы> [ CONTAINS FILESTREAM ]
    | REMOVE FILEGROUP <имя файловой группы>
    | MODIFY FILEGROUP <имя файловой группы>
    {
        { READ_ONLY | READ_WRITE }
        | DEFAULT
        | NAME = <имя новой файловой группы>
    }
}
```



Граф 3.12. Синтаксис изменения файловых групп



Граф 3.13. Синтаксис нетерминального символа "изменения"

Здесь можно добавить новую файловую группу (`ADD FILEGROUP`), удалить любую группу, кроме первичной (`REMOVE FILEGROUP`), или изменить характеристики существующей файловой группы (`MODIFY FILEGROUP`).

При добавлении новой файловой группы можно указать ключевые слова `CONTAINS FILESTREAM`. Это означает, что файловая группа будет использоваться только для хранения так называемых файловых потоков, значений столбца таблицы, имеющего тип данных `VARBINARY (MAX)`. О файловых потоках мы поговорим в главе 5.

При изменении файловой группы можно перевести ее в состояние только для чтения (`READ_ONLY`) или в состояние чтения и записи (`READ_WRITE`), можно сделать ее группой по умолчанию (`DEFAULT`), можно изменить ее имя.

Выполните следующий пакет, в котором для базы данных `MultyGroup` переименовывается существующая файловая группа `MultyGroup2` в `MultyGroup0` и добавляется еще одна с именем только что измененной группы `MultyGroup2` (пример 3.34).

Пример 3.34. Внесение изменений в файловые группы базы данных

```

USE master;
GO
ALTER DATABASE MultyGroup
    MODIFY FILEGROUP MultyGroup2
        NAME = MultyGroup0;
GO

ALTER DATABASE MultyGroup
    ADD FILEGROUP MultyGroup2;
GO

```

Проверьте результат выполнения этого пакета. В Management Studio на панели **Object Explorer** щелкните правой кнопкой мыши по имени базы данных **MultyGroup** и в появившемся контекстном меню выберите элемент **Refresh**, чтобы отображались результаты изменения базы данных. Затем в том же контекстном меню выберите **Properties**. В появившемся окне в левой верхней части щелкните по строке **Filegroups**. В основной части окна можно увидеть, что изменения выполнены (рис. 3.23).

В окне видно, что новая файловая группа `MultyGroup2` не содержит файлов. Чтобы во вновь созданную файловую группу добавить один или более файлов, нужно соответствующее количество раз выполнить оператор `ALTER DATABASE`.

Name	Files	Read-Only	Default
PRIMARY	2		<input checked="" type="checkbox"/>
MultyGroup0	2	<input type="checkbox"/>	<input type="checkbox"/>
MultyGroup2	0	<input type="checkbox"/>	<input type="checkbox"/>

Рис. 3.23. Измененный список файловых групп

3.5.1.5. Изменение других характеристик базы данных

В описании синтаксиса оператора ALTER DATABASE (см. листинг 3.5) последней строкой присутствует нетерминальный символ "характеристики базы данных". Эти характеристики задаются предложением SET в операторе ALTER DATABASE (листинг 3.9).

Листинг 3.9. Синтаксис изменения характеристик базы данных

```
ALTER DATABASE { <имя базы данных> | CURRENT }
SET <характеристика> [, <характеристика>]...;
```

Вкратце перечислим существующие характеристики. Достаточно подробно они описаны в *приложении 4*.

- ◆ AUTO_CLOSE { ON | OFF } — задает автоматическое закрытие базы данных после отключения от нее последнего пользователя.
- ◆ AUTO_CREATE_STATISTICS { ON | OFF } — задает или отключает автоматическое создание статистики по базе данных, требуемой для оптимизации запроса.
- ◆ AUTO_UPDATE_STATISTICS { ON | OFF } — задает или отключает автоматическое обновление статистики по базе данных, требуемой для оптимизации запроса.
- ◆ AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF } — влияет на выполнение запросов, которые требуют обновления статистики по базе данных.
- ◆ AUTO_SHRINK { ON | OFF } — задает или отключает режим автоматического сжатия файлов базы данных.
- ◆ Состояние базы данных: { ONLINE | OFFLINE | EMERGENCY }.
- ◆ Вид базы данных: обычная, частично автономная:
CONTAINMENT = { NONE | PARTIAL }.
- ◆ Возможность изменения данных базы данных: { READ_ONLY | READ_WRITE }.
- ◆ Допустимое количество подключаемых к базе данных пользователей:
{ SINGLE_USER | RESTRICTED_USER | MULTI_USER }.
- ◆ CURSOR_CLOSE_ON_COMMIT { ON | OFF } — задает автоматическое закрытие курсора при подтверждении транзакции.

- ◆ CURSOR_DEFAULT { GLOBAL | LOCAL } — задает область действия курсора.
- ◆ RECOVERY { FULL | BULK_LOGGED | SIMPLE } — определяет стратегии создания резервных копий и восстановления поврежденной базы данных.
- ◆ PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE } — задает способ обнаружения поврежденных страниц базы данных.
- ◆ ANSI_NULL_DEFAULT { ON | OFF } — определяет значение по умолчанию для столбцов таблиц: NULL или NOT NULL.
- ◆ ANSI_NULLS { ON | OFF } — задает соответствие стандарту ANSI результат сравнения любых значений со значением NULL.
- ◆ ANSI_PADDING { ON | OFF } — влияет на добавление конечных пробелов в столбцы типов данных VARCHAR и NVARCHAR, а также на конечные нули в двоичных значениях VARBINARY.
- ◆ ANSI_WARNINGS { ON | OFF } — влияет на появление предупреждающих сообщений или сообщений об ошибке при появлении значения NULL в агрегатных функциях или при делении на ноль.
- ◆ ARITHABORT { ON | OFF } — определяет реакцию системы на арифметическое переполнение или при делении на ноль: прекращение выполнения запроса или выдача сообщения и продолжение выполнения запроса.
- ◆ COMPATIBILITY_LEVEL = { 80 | 90 | 100 | 110 } — уровень совместимости с предыдущими версиями SQL Server.
- ◆ CONCAT_NULL_YIELDS_NULL { ON | OFF } — определяет результат конкатенации двух строковых данных, когда одно из строковых значений имеет значение NULL.
- ◆ DATE_CORRELATION_OPTIMIZATION { ON | OFF } — определяет поддержание статистики между таблицами, связанными ограничением FOREIGN KEY и содержащими столбцы типа данных datetime.
- ◆ NUMERIC_ROUNDABORT { ON | OFF } — задает возможность появления ошибки при потере точности в процессе вычисления числового значения.
- ◆ PARAMETERIZATION { SIMPLE | FORCED } — условие выполнения параметризации запросов.
- ◆ QUOTED_IDENTIFIER { ON | OFF } — определяет допустимость использования кавычек при задании идентификаторов с разделителями.
- ◆ RECURSIVE_TRIGGERS { ON | OFF } — определяет допустимость срабатывания рекурсивных триггеров.
- ◆ DB_CHAINING { ON | OFF } — задает, может ли база данных находиться в межбазовой цепочке владения или быть источником в такой цепочке.
- ◆ TRUSTWORTHY { ON | OFF } — определяет, могут ли модули базы данных получать доступ к ресурсам вне базы данных.
- ◆ ALLOW_SNAPSHOT_ISOLATION { ON | OFF } — определяет допустимость использования уровня изоляции транзакции SNAPSHOT.

- ◆ READ_COMMITTED_SNAPSHOT { ON | OFF } — определяет поведение транзакции с уровнем изоляции READ COMMITTED.

Давайте из праздного любопытства изменим значения первых пяти перечисленных характеристик базы данных BestDatabase (пример 3.35). Мы установим для этих характеристик значения, отличные от значений по умолчанию, присваиваемых базе данных при ее создании. Рассматривайте этот пример только как исследование. Совсем не обязательно для реальной базы данных, например, отменять создание статистических данных, которые позволяют сильно повысить производительность системы при обработке запросов.

Пример 3.35. Изменение некоторых характеристик базы данных BestDatabase

```
USE master;
GO
ALTER DATABASE BestDatabase
SET AUTO_CLOSE ON,
    AUTO_CREATE_STATISTICS OFF,
    AUTO_UPDATE_STATISTICS OFF,
    AUTO_UPDATE_STATISTICS_ASYNC ON,
    AUTO_SHRINK ON;
GO
```

3.5.2. Изменение базы данных диалоговыми средствами Management Studio

В Management Studio можно изменять несколько меньше характеристик базы данных, чем при использовании оператора ALTER DATABASE.

3.5.2.1. Изменение имени базы данных

Базу данных очень легко переименовать. Для этого нужно щелкнуть правой кнопкой по имени базы данных и в появившемся контекстном меню выбрать пункт **Rename** (переименовать). Имя базы данных станет доступным для изменения непосредственно в панели **Object Explorer**. Такой же эффект можно получить, если аккуратно щелкнуть мышью по самому имени базы данных в **Object Explorer**.

Чтобы отобразить окно свойств базы данных, нужно в **Object Explorer** щелкнуть правой кнопкой мыши по имени базы данных и в контекстном меню выбрать элемент **Properties**. Появится окно свойств.

3.5.2.2. Изменение файлов базы данных

Если в окне свойств базы данных BestDatabase выбрать вкладку **Files**, то появится список файлов этой базы, как это показано на рис. 3.24.

На этой вкладке мы можем не только изменять характеристики существующих файлов, но также добавлять и удалять файлы.

Logical Name	File Type	Filegroup	Initi...	Autogrowth / Maxsize	Path	File Name
BestDatabase_dat	Rows ...	PRIMARY	5	By 1 MB, Unlimited	D:\BestDatabase	Winner.mdf
BestDatabase_log	Log	Not Applicable	1	By 10 percent, Limited to 2097152 MB	D:\BestDatabase	Winner.ldf

Рис. 3.24. Характеристики файлов базы данных BestDatabase

Здесь можно изменить логическое имя файла (столбец **Logical Name**), начальный размер файла (**Initial Size (MB)**) и характеристики увеличения размера (**Autogrowth**). А вот изменить имя физического файла или путь к физическому файлу в диалоговом режиме Management Studio нельзя, поскольку такие изменения требуют перевода базы данных в **OFFLINE**, а в этом состоянии базы невозможен просмотр ее свойств.

Сейчас удалить ни один из существующих файлов базы BestDatabase мы не можем, потому что она содержит минимальное количество требуемых файлов — один файл данных и один файл журнала транзакций. По этой причине кнопка **Remove** (удалить) является неактивной.

Для изменения логического имени файла (данных или журнала транзакций) нужно щелкнуть мышью по имени соответствующего файла. Поле станет доступным для изменения. После внесения изменений нужно нажать клавишу **<Enter>**. Надо сказать, что система не проверяет вводимые символы, вы можете помещать в имя пробелы, вообще любые символы, что одобрить, как вы понимаете, нельзя.

Для изменения начального размера файла нужно мышью выделить соответствующий элемент (заголовок столбца **Initial Size (MB)**). В правой части строки появится управляющий элемент, позволяющий изменять размер на 1 Мбайт в сторону увеличения или уменьшения (рис. 3.25). Размер также можно менять, вводя с клавиатуры в этом поле нужное значение.



Рис. 3.25. Изменение начального размера файла

После внесения изменения следует перевести фокус ввода на любой другой элемент.

Для изменения характеристики увеличения размера (столбец **Autogrowth**) щелкните мышью по кнопке с многоточием в правой части этого поля у нужного файла. Появится диалоговое окно, позволяющее изменить характеристики увеличения размера файла. На рис. 3.26 показано такое диалоговое окно для файла журнала транзакций базы данных BestDatabase.

Здесь установлен флажок допустимости автоматического увеличения размера (**Enable Autogrowth**). Если увеличение размера возможно, то в группе переключателей **File Growth** (увеличение размера файла) можно указать, в каких единицах размер файла должен увеличиваться: в процентах (**In Percent**) или в мегабайтах (**In Megabytes**).

Можно задать максимальный размер файла в группе переключателей **Maximum File Size**: максимальный (ограниченный) размер файла в мегабайтах (**Restricted File Growth (MB)**) или неограниченный размер файла (**Unrestricted File Growth**).

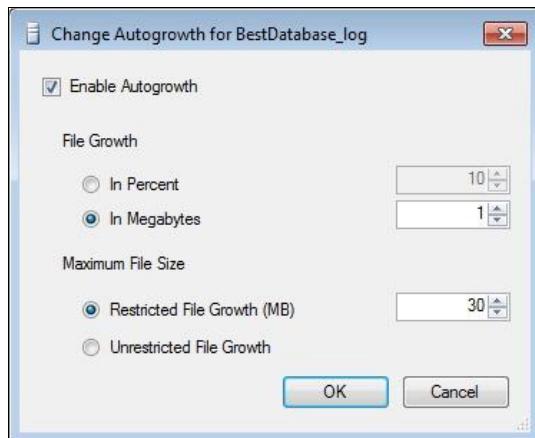


Рис. 3.26. Изменение характеристик увеличения размера файла

Для добавления нового файла (данных или журнала транзакций) нужно на вкладке **Files** щелкнуть по кнопке **Add** (добавить). В список файлов будет добавлена новая строка, где отдельные столбцы будут содержать пустые значения, а для других установлены значения по умолчанию (рис. 3.27).

Logical Name	File Type	Filegroup	Initi...	Autogrowth / Maxsize	Path	File Name
BestDatabase_dat	Rows Data	PRIMARY	5	By 1 MB, Unlimited	...	D:\BestDatabase\Winner.mdf
BestDatabase_log	Log	Not Applicable	1	By 10 percent, Limited to 2097152 MB	...	D:\BestDatabase\Winner.ldf
	Rows Data	PRIMARY	5	By 1 MB, Unlimited	...	C:\Program File... [...]

Рис. 3.27. Добавление нового файла в базу данных BestDatabase

Вначале добавим файл данных. Зададим ему логическое имя (**Logical Name**). Щелкните мышью по пустому значению этого поля и введите логическое имя файла, например, **B2_dat**. Пусть это будет второй файл данных в нашей базе данных **BestDatabase**. По этой причине поле **File Type** (тип файла) мы не меняем, а оставляем значение, установленное по умолчанию, — **Rows Data** (строки данных, т. е. файл данных). Значение поля **Filegroup** (файловая группа), заданное как **PRIMARY**, не изменяем. Начальный размер файла, установленный в 5 Мбайт по умолчанию (поле **Initial Size (МВ)**), также оставим без изменения. Не станем изменять и величину автоматического увеличения размера файла (**Autogrowth**). А вот новый путь к файлу, отличный от пути по умолчанию, зададим. Это можно сделать двумя способами. Первый — для любителей все вводить руками. Нужно щелкнуть мышью по полю **Path** и с клавиатуры набрать полный путь к файлу, не забыв указать и имя внешнего устройства. Напомню, что все каталоги в этом пути уже должны существовать на выбранном вами носителе.

В случае использования диалоговых средств нужно просто щелкнуть мышью по кнопке с многоточием справа от заданного пути по умолчанию в поле **Path**. Появится окно выбора пути для размещения этого файла (рис. 3.28). Выберите диск D: и уже существующий наш каталог **BestDatabase** и щелкните по кнопке **OK**.

В поле имени файла (**File Name**) введем имя создаваемого файла **Winner2.ndf**.



Рис. 3.28. Выбор папки для размещения нового файла базы данных

Второй файл данных для нашей базы данных BestDatabase мы создали. Теперь создадим второй файл журнала транзакций.

Щелкните в окне по кнопке **Add**. В списке файлов появится новая строка. Задайте логическое имя файла `b2_log`. Щелкните по полю типа файла (**File Type**) и из раскрывающегося списка выберите значение `Log`.

Выберите каталог для размещения файла `D:\BestDatabase`. Имя файла введите `Winner2.ldf`.

Теперь новый список файлов этой базы данных будет выглядеть следующим образом (рис. 3.29).

Logical Name	File Type	Filegroup	Initi...	Autogrowth / Maxsize	Path	File Name
BestDatabase_dat	Rows Data	PRIMARY	5	By 1 MB, Unlimited	[...]	D:\BestDatabase\Winner.mdf
BestDatabase_log	Log	Not Applicable	1	By 10 percent, Limited to 2097152 MB	[...]	D:\BestDatabase\Winner.ldf
B2_dat	Rows Data	PRIMARY	5	By 1 MB, Unlimited	[...]	D:\BestDatabase\Winner2.mdf
B2_log	Log	Not Applicable	1	By 10 percent, Unlimited	[...]	D:\BestDatabase\Winner2.ldf

Рис. 3.29. Добавленные два файла в базу данных BestDatabase

Чтобы созданные в предыдущих шагах добавления были фактически выполнены для текущей базы данных, щелкните по кнопке **OK**. Если при создании любого из добавляемых файлов вы допустили ошибки, то сообщения о них появятся только сейчас. Например, если вы задали имя для второго файла журнала транзакций такое же, как и для первого файла, то после щелчка по кнопке **OK** вы получите такое сообщение, как на рис. 3.30.

Здесь говорится о том, что файл '`D:\BestDatabase\Winner2.ldf`' не может быть перезаписан, поскольку он используется базой данных `BestDatabase`.

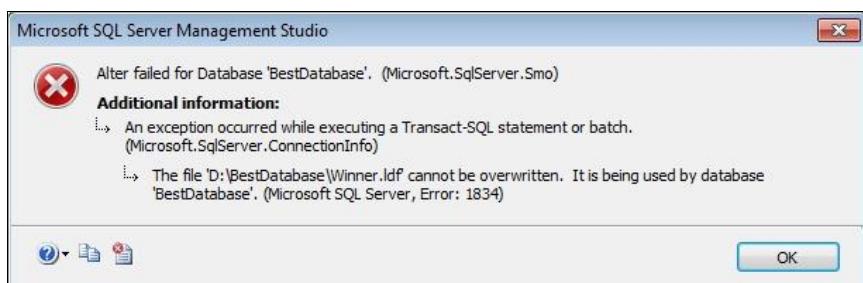


Рис. 3.30. Сообщение о дублировании имен файлов

Для удаления любого файла базы данных (кроме первичного) нужно на вкладке **Files** щелкнуть мышью по кнопке **Remove**, расположенной в нижней части окна. Текущий (выделенный в окне) файл будет удален.

ВНИМАНИЕ!

При щелчке по кнопке удаления система не запрашивает подтверждения необходимости удаления текущего файла. Однако если вам на самом деле не нужно удалять этот файл, вы просто в окне просмотра свойств базы данных можете щелкнуть по кнопке **Cancel** (отмена всех выполненных изменений базы данных).

Если вы выделяете на форме файл, который не может быть удален, то кнопка **Remove** будет в недоступном состоянии.

3.5.2.3. Изменение файловых групп базы данных

Для изменения файловых групп базы BestDatabase щелкните мышью по строке **Filegroups** в панели **Select a Page**. Появится список файловых групп (рис. 3.31).

Name	Files	Read-Only	Default
PRIMARY	3		<input checked="" type="checkbox"/>

Рис. 3.31. Список файловых групп базы данных BestDatabase

Пока база данных содержит только одну первичную (**PRIMARY**) файловую группу. Чтобы добавить вторую файловую группу, нужно щелкнуть мышью по кнопке **Add** в нижней части этого окна. В списке появится новая строка. В поле имени (**Name**) введите имя файловой группы, например, **SECOND** (рис. 3.32).

Name	Files	Read-Only	Default
PRIMARY	3		<input checked="" type="checkbox"/>
SECOND	0	<input type="checkbox"/>	<input type="checkbox"/>

Рис. 3.32. Измененный список файловых групп базы данных BestDatabase

Видно, что количество файлов, принадлежащих этой группе (поле **Files**), равно нулю. Чтобы переместить только что созданный новый файл данных **B2_dat** в эту файловую группу, щелкните мышью по строке **Files** в панели **Select a Page**.

В столбце **Filegroup** для файла B2_dat из раскрывающегося списка выберите имя файловой группы SECOND.

Этот файл станет принадлежать файловой группе SECOND.

ВНИМАНИЕ!

Поместить во вновь созданную файловую группу можно только созданные в этой же сессии файлы, только те файлы, которые были созданы до щелчка по кнопке **OK**.

Давайте для порядка создадим еще один файл данных и одновременно с этим для него новую файловую группу.

На вкладке просмотра файлов щелкните по кнопке **Add**. Создайте новый файл данных с любыми характеристиками (мы его потом удалим вместе с новой файловой группой). При задании файловой группы (столбец **Filegroup**) из раскрывающегося списка выберите последнюю строку <new filegroup>. Это означает, что для файла вы собираетесь создать новую файловую группу. Появится диалоговое окно задания характеристик файловой группы (рис. 3.33).

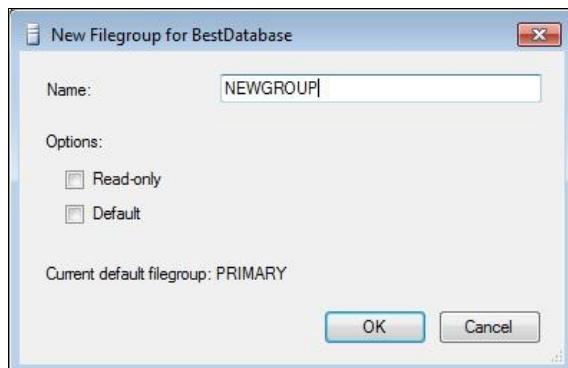


Рис. 3.33. Создание новой файловой группы

В поле **Name** введите имя группы, например NEWGROUP, и щелкните по кнопке **OK**, чтобы создать эту группу.

В этом диалоговом окне существует еще два флажка — **Read-only** (только для чтения) и **Default** (файловая группа по умолчанию). Ни тот, ни другой мы устанавливать не будем. Во-первых, мы предполагаем, что для файлов, включаемых в эту группу, будут допустимы не только операции чтения, но и записи, во-вторых, мы не собираемся менять файловую группу по умолчанию; такой у нас является группа PRIMARY.

Перейдите во вкладку **Filegroups**. Вы увидите в списке файловых групп и только что созданную группу.

Чтобы удалить любую файловую группу, кроме первичной, нужно выделить ее в списке мышью и щелкнуть по кнопке **Remove**. Будет удалена файловая группа, а все файлы, принадлежащие этой группе, будут автоматически перемещены в первичную файловую группу.

Удалите только что созданную группу. Вы увидите, что принадлежащий ей файл будет перемещен в группу PRIMARY.

3.5.2.4. Изменение других характеристик базы данных

Общие характеристики базы данных можно изменять, если в окне просмотра свойств этой базы данных в панели Select a Page выбрать вкладку Options (рис. 3.34).

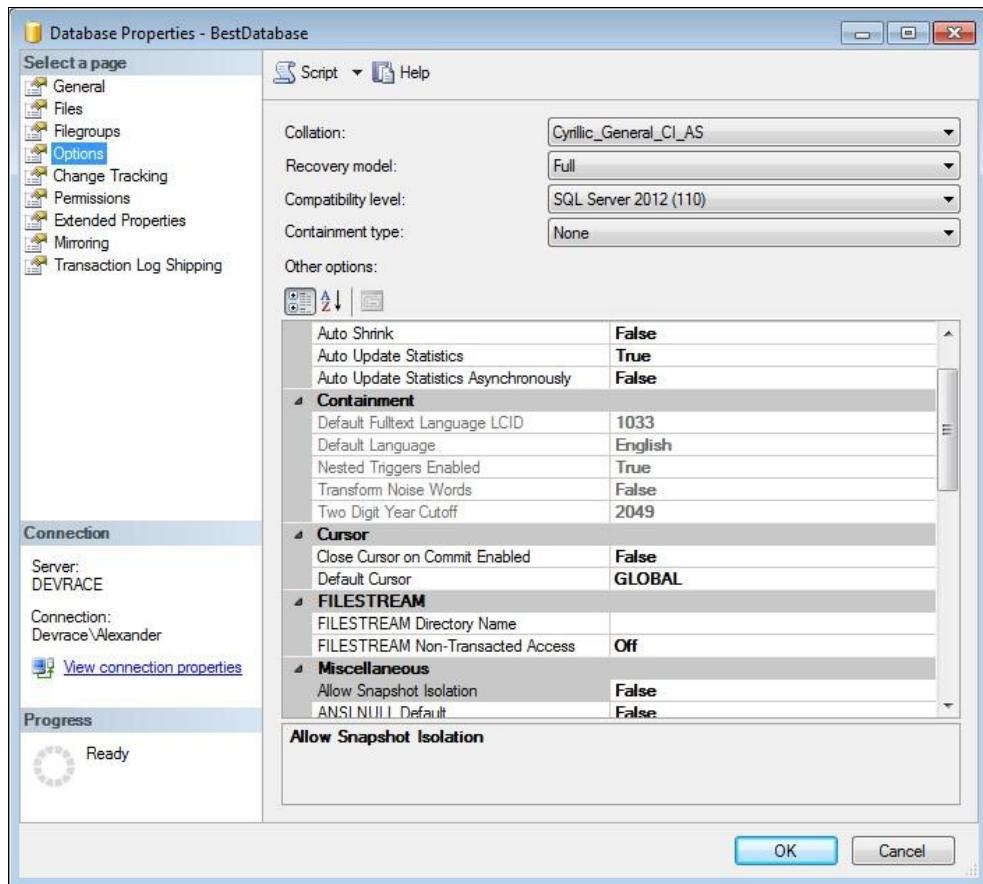


Рис. 3.34. Вкладка свойств базы данных

Во-первых, здесь можно изменить набор символов для базы данных, выбрав в раскрывающемся списке **Collation** нужный набор. Заменим установленный нами по умолчанию при создании базы данных порядок сортировки Cyrillic_General_CI_AS на порядок сортировки, позволяющий вводить символы, присутствующие, скажем, во французском языке. Помимо обычных латинских букв в этом языке также существуют буквы ç, é, è и ряд других. Щелкните по текущему элементу списка **Collation**. Появится раскрывающийся список доступных порядков сортировки. Выберите из списка French_CI_AI.

Можно изменить модель восстановления. Для этого нужно в раскрывающемся списке **Recovery model** выбрать требуемую модель: **Full**, **Bulk-logged**, **Simple**.

Существует понятие уровня совместимости с предыдущими версиями: **Compatibility level**. Для вновь создаваемых систем обработки данных следует оставить для уровня совместимости значение по умолчанию.

В основном списке режимов (рис. 3.35) присутствует множество характеристик базы данных в целом. Те характеристики, которые можно изменять на этой форме, представлены шрифтом обычного цвета. Если характеристику изменять нельзя, то соответствующая строка имеет серый шрифт.

Automatic	
Auto Close	True
Auto Create Statistics	False
Auto Shrink	True
Auto Update Statistics	False
Auto Update Statistics Asynchronously	True
Containment	
Default Fulltext Language LCID	1033
Default Language	English
Nested Triggers Enabled	True
Transform Noise Words	False
Two Digit Year Cutoff	2049
Cursor	
Close Cursor on Commit Enabled	False
Default Cursor	GLOBAL
FILESTREAM	
FILESTREAM Directory Name	
FILESTREAM Non-Transacted Access	Off
Miscellaneous	
Allow Snapshot Isolation	False
ANSI NULL Default	False
ANSI NULLS Enabled	False
ANSI Padding Enabled	False
ANSI Warnings Enabled	False
Arithmetic Abort Enabled	False
Concatenate Null Yields Null	False

Рис. 3.35. Общие свойства базы данных

Чтобы изменить значение характеристики, нужно щелкнуть мышью по этой строке; в правой части значения характеристики появится стрелка, позволяющая вызвать соответствующий раскрывающийся список. Чаще всего в списке присутствует только два значения — истина и ложь (**True** и **False**). На рис. 3.36 показан пример появления раскрывающегося списка с двумя значениями **True** и **False** для свойства **Database Read-Only**.

Если этому свойству задать значение **True**, то база данных становится базой только для чтения (база данных **READ_ONLY**). Если значение установлено в **False** (значение

Database Read-Only	False
Database State	True
Encryption Enabled	False

Рис. 3.36. Выбор значения для свойства базы данных **Database Read-Only**

при создании базы данных по умолчанию), то для базы допустимы как операции чтения, так и записи, т. е. добавление, изменение и удаление данных (база данных `READ_WRITE`).

ВНИМАНИЕ!

Еще раз напомню. Чтобы проделанная вами работа по изменению любых характеристик базы, ее файлов и файловых групп действительно была зафиксирована в базе данных, необходимо в окне просмотра свойств базы в завершение всех действий щелкнуть по кнопке **OK**. Только тогда изменения будут применены к базе данных. Одна моя знакомая (вы не поверите — она натуральная блондинка) жаловалась мне, что не может в Management Studio изменить нужные ей характеристики. Оказалось, что после выполнения изменений она просто закрывала окно, щелкая по кнопке закрытия в правом верхнем углу этого окна.

В этой грустной истории есть один полезный для нас с вами смысл. Вы можете вносить любые изменения в параметры базы данных. Потом, если вдруг замечаете, что сделали совсем не то, что было нужно, вы спокойно отмените все эти безобразия, щелкнув по кнопке **Cancel** или просто закрыв окно.

В Management Studio можно очень просто переводить базу данных в неактивное (`OFFLINE`) и активное (`ONLINE`) состояние.

Если вам понадобится скопировать базу данных на другой носитель, то вы не сможете этого сделать, если при запущенном на выполнении сервере базы данных Database Engine база находится в состоянии `ONLINE`. Для этих целей базу нужно перевести в состояние `OFFLINE`.

Чтобы перевести базу данных в неактивное состояние, нужно в панели **Object Explorer** щелкнуть правой кнопкой мыши по имени базы, в контекстном меню выбрать элемент **Tasks** (задачи) и в открывшемся подменю **Take Offline**. Для перевода в активное состояние в этом подменю нужно выбрать элемент **Bring Online**.

3.5.2.5. Отображение отчета использования дискового пространства базы данных

Здесь хочу сказать два слова о тех "красивостях", которые можно получать, работая с SQL Server. В Management Studio можно создавать различные отчеты, отображающие состояние базы данных и ее объектов. Давайте сейчас создадим отчет, отображающий использование дискового пространства файлами базы данных `BestDatabase`. На панели **Object Explorer** щелкните правой кнопкой мыши по имени базы данных, в контекстном меню выберите элемент **Reports**, затем элементы **Standard Reports** и **Disk Usage**. В результате будет создан наглядный отчет, отображающий использование базой данных внешнего пространства (рис. 3.37).

В процессе заполнения базы данных необходимыми данными можно периодически создавать такой отчет, который будет отображать текущее положение с использованием дискового пространства.

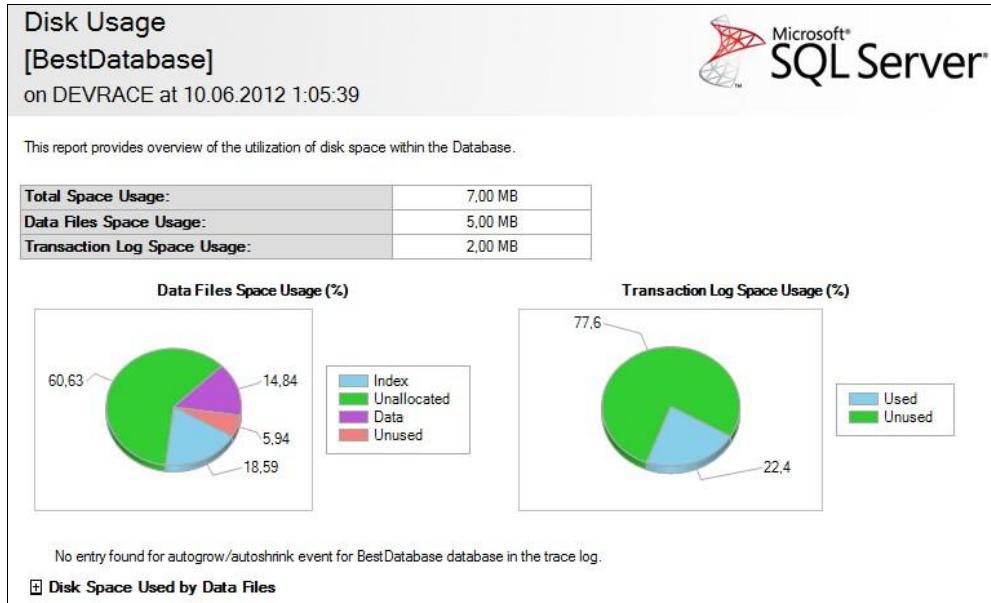


Рис. 3.37. Отчет использования дискового пространства файлами базы данных

3.5.3. Удаление базы данных диалоговыми средствами Management Studio

Для удаления базы данных в Management Studio нужно в **Object Explorer** щелкнуть правой кнопкой мыши по имени базы данных и в контекстном меню выбрать **Delete**.

3.6. Создание автономной базы данных

Автономные базы данных (*contained*) не имеют внешних зависимостей, т. е. ссылок на какие-либо объекты, описанные в экземпляре сервера. По этой причине их очень просто перемещать между различными экземплярами сервера базы данных, переносить на другие компьютеры.

Созданную "обычную" базу данных можно также перевести в автономную.

Рассмотрим вкратце все действия, необходимые для создания автономной базы данных и соответствующего пользователя.

Для выполнения следующих действий на диске D: создайте новый каталог с именем *ContainedDatabase*.

3.6.1. Установка допустимости автономных баз данных

Вначале нужно установить допустимость для экземпляра сервера таких баз данных. Это можно сделать при выполнении хранимой процедуры *sp_configure* и при использовании диалоговых средств Management Studio.

При использовании хранимой процедуры `sp_configure` в утилите командной строки или в Management Studio выполните операторы примера 3.36.

Пример 3.36. Задание допустимости автономных баз данных

```
USE master;
GO
sp_configure 'show advanced options', 1;
RECONFIGURE WITH OVERRIDE;
GO
sp_configure 'contained database authentication', 1;
RECONFIGURE WITH OVERRIDE;
GO
sp_configure 'show advanced options', 0;
RECONFIGURE WITH OVERRIDE;
GO
```

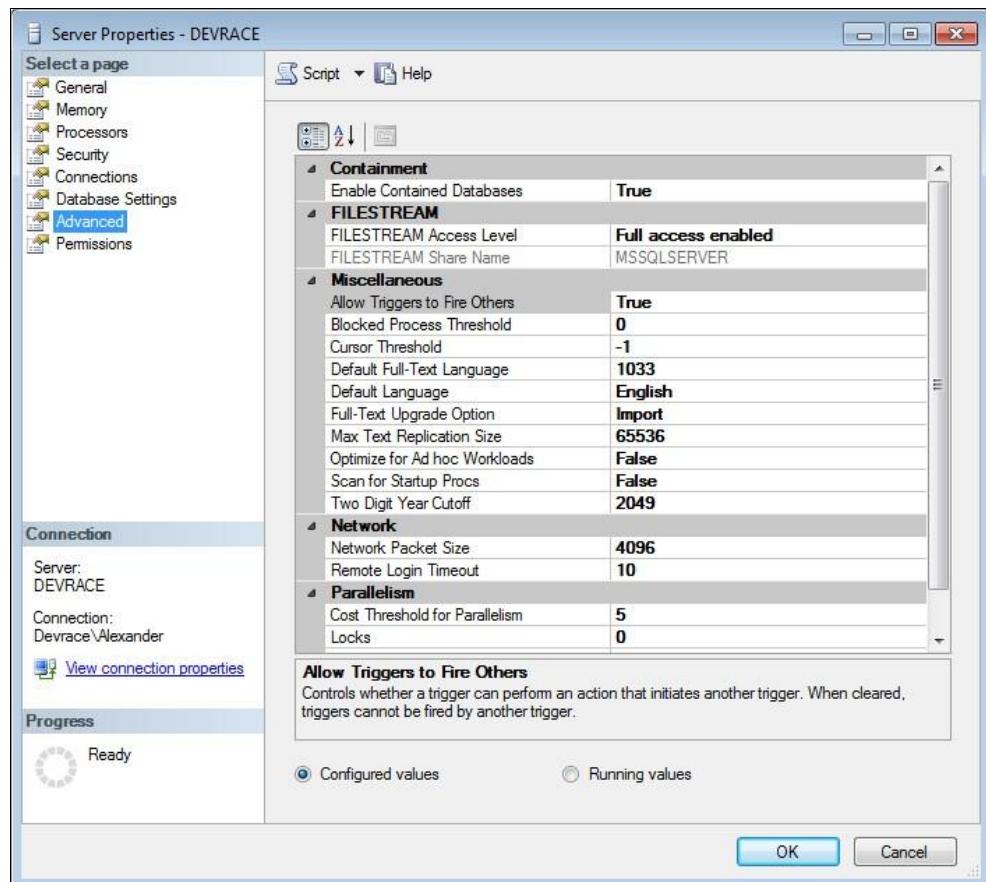


Рис. 3.38. Вкладка **Advanced** свойств сервера

Здесь значение опции contained database authentication устанавливается в 1, что позволяет создавать автономные базы данных. Команда RECONFIGURE нужна, чтобы изменения вступили в силу.

При использовании диалоговых средств Management Studio в окне **Object Explorer** щелкните правой кнопкой мыши по имени сервера базы данных и в контекстном меню выберите **Properties**. В появившемся окне свойств сервера **Server Properties** в левой части выберите вкладку **Advanced** (рис. 3.38).

В основной части окна из раскрывающегося списка поля **Enable Contained Databases** выберите **True**. Закройте окно свойств сервера, щелкнув по кнопке **OK**.

Чтобы изменения вступили в силу, нужно перезапустить сервер. В окне **Object Explorer** щелкните правой кнопкой мыши по имени сервера базы данных и в контекстном меню выберите **Restart**. В появившемся окне подтверждения перезапуска сервера щелкните по кнопке **Yes**. Через некоторое время сервер будет заново запущен, изменения будут приняты.

3.6.2. Создание автономной базы данных и пользователя средствами языка Transact-SQL

Выполните в утилите командной строки или в Management Studio следующий скрипт по созданию автономной базы данных ContainedDatabase1 и ее пользователя с паролем (пример 3.37).

Пример 3.37. Создание автономной базы данных ContainedDatabase1 и ее пользователя

```
USE master;
GO
IF DB_ID('ContainedDatabase1') IS NOT NULL
    DROP DATABASE ContainedDatabase1;
GO
CREATE DATABASE ContainedDatabase1
CONTAINMENT = PARTIAL
ON PRIMARY (NAME = ContainedDatabase1_dat,
    FILENAME = 'D:\ContainedDatabase\ContainedDatabase1.mdf')
LOG ON (NAME = ContainedDatabase1_log,
    FILENAME = 'D:\ContainedDatabase\ContainedDatabase1.ldf')
WITH
    FILESTREAM (NON_TRANSACTED_ACCESS = READ_ONLY,
        DIRECTORY_NAME = 'dir'),
    DEFAULT_FULLTEXT_LANGUAGE = Russian,
    DEFAULT_LANGUAGE = English,
    NESTED_TRIGGERS = OFF,
    TRANSFORM_NOISE_WORDS = OFF,
    TWO_DIGIT_YEAR_CUTOFF = 1990,
    DB_CHAINING OFF,
```

```

TRUSTWORTHY OFF;
GO

USE ContainedDatabase1;
GO

CREATE USER ContainedUser
    WITH PASSWORD = 'ContainedPassword';
GO

```

Здесь создается база данных **ContainedDatabase1** и пользователь этой базы данных **ContainedUser** с паролем **ContainedPassword**.

3.6.3. Создание автономной базы данных диалоговыми средствами Management Studio

Для создания автономной базы данных в окне **Object Explorer** щелкните правой кнопкой мыши по папке **Databases** и в контекстном меню выберите **New Database**. В окне **New Database** в поле **Database name** введите имя базы данных: **ContainedDatabase2**. В поле **Path** введите один и тот же путь к обоим файлам базы данных (файлу данных и файлу журнала транзакций): **D:\ContainedDatabase**. В поле **File Name** укажите имена файлов базы данных (рис. 3.39).

Database files:						
Logical Name	File Type	Filegroup	Initial...	Autogrowth / Maxsize	Path	File Name
ContainedDatabase2	Rows Data	PRIMARY	5	By 1 MB, Unlimited	[...] D:\ContainedDatabase	[...] ContainedDatabase2.mdf
ContainedDatabase2_log	Log	Not Applicable	1	By 10 percent, Unlimited	[...] D:\ContainedDatabase	[...] ContainedDatabase2ldf

Рис. 3.39. Создаваемая база данных contained

На вкладке **Options** в поле **Containment type** выберите из раскрывающегося списка значение **Partial**. Для завершения создания базы данных щелкните по кнопке **OK**.

ВНИМАНИЕ!

Чтобы все действия по созданию автономной базы данных и ее пользователя выполнились правильно, вам следует запустить Management Studio с правами (от имени) администратора.

3.6.4. Создание автономного пользователя в Management Studio

Чтобы в базе данных **ContainedDatabase2** создать нового автономного пользователя с паролем, в **Object Explorer** раскройте узел **Databases**, затем **ContainedDatabase2** и **Security**. Щелкните правой кнопкой мыши по строке **Users** и выберите в контекстном меню строку **New User**. Появится диалоговое окно **Database User — New**. Из раскрывающегося списка **User type** выберите **SQL user with password**. В поле **User**

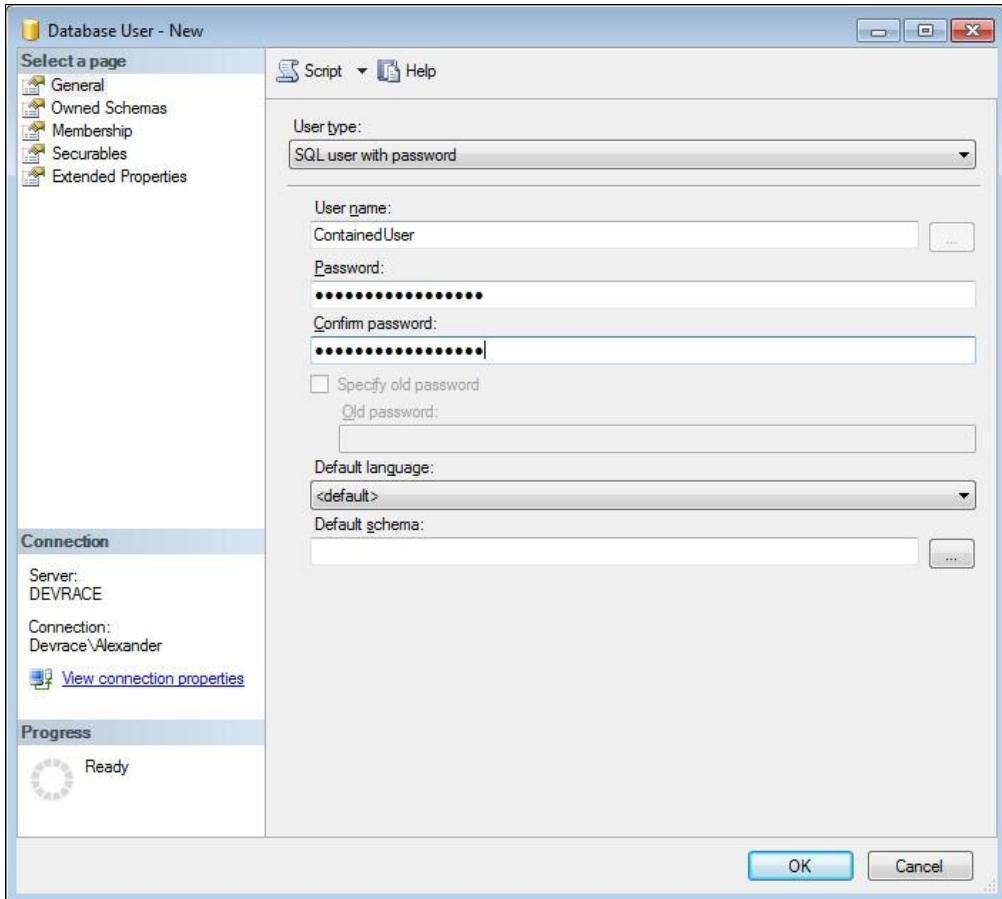


Рис. 3.40. Задание пользователя базы данных. Вкладка General

name введите имя пользователя: ContainedUser, а в поля **Password** и **Confirm password** введите: ContainedPassword (рис. 3.40).

В левой части окна выберите вкладку **Membership**. Установите флажок в поле **db_owner** (рис. 3.41). Щелкните по кнопке **OK**.



Рис. 3.41. Задание пользователя базы данных.
Вкладка Membership

3.6.5. Соединение с автономной базой данных в Management Studio

Для соединения с автономной базой данных под созданным пользователем при запуске на выполнение Management Studio в окне соединения с сервером нужно щелкнуть мышью по кнопке **Options**. В окне появятся три вкладки. Текущей вкладкой будет **Connection Properties** (рис. 3.42).

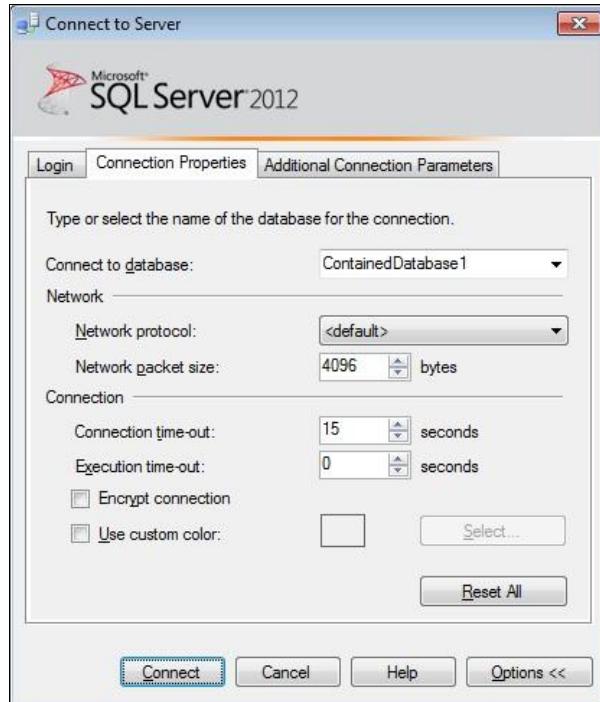


Рис. 3.42. Соединение с базой данных. Вкладка **Connection Properties**

В раскрывающемся списке **Connect to database** нужно выбрать базу данных **ContainedDatabase1** или набрать с клавиатуры это имя.

Выберите вкладку **Login** (рис. 3.43). Из раскрывающегося списка **Authentication** выберите **SQL Server Authentication**, в поле **Login** введите имя пользователя **ContainedUser**, в поле **Password** — пароль **ContainedPassword**.

Щелкните мышью по кнопке **Connect**. Произойдет соединение с этой базой данных. В списке баз данных в **Object Explorer** будет видна только **ContainedDatabase1**.

ЗАМЕЧАНИЕ

В той версии SQL Server 2012, которая была у меня на момент написания этих строк, нужно было вернуться на вкладку **Connection Properties** и повторно указать базу данных, с которой производится соединение. Иначе возникали ошибки соединения. Такое же поведение было и у кандидата в релизы.



Рис. 3.43. Соединение с базой данных. Вкладка Login

3.7. Присоединение базы данных

Если какая-либо база данных была создана на другом компьютере и перенесена на ваш компьютер, то ее можно присоединить к базам данных текущего экземпляра сервера. Присоединить такую базу можно средствами языка Transact-SQL и при использовании диалоговых средств компонента Management Studio.

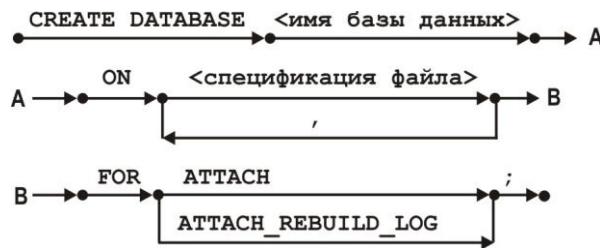
3.7.1. Присоединение базы данных с использованием Transact-SQL

В Transact-SQL для присоединения существующей базы данных используется тот же оператор `CREATE DATABASE`, но с несколько иным синтаксисом. Синтаксис оператора, используемого для присоединения базы, представлен в листинге 3.10 и в соответствующем R-графе (граф 3.14).

Листинг 3.10. Синтаксис оператора `CREATE DATABASE`.

Вариант присоединения существующей базы данных

```
CREATE DATABASE <имя базы данных>
ON <спецификация файла> [, <спецификация файла>]...
FOR { ATTACH
      | ATTACH_REBUILD_LOG };
```



При использовании предложения FOR ATTACH все файлы присоединяемой базы данных, как файлы данных (первичные mdf и вторичные ndf), так и файлы журнала транзакций, должны быть доступны. В случае использования предложения FOR ATTACH_REBUILD_LOG файлы журнала транзакций могут отсутствовать. Новый файл (файлы) журнала транзакций будет создан в процессе присоединения базы данных.

Для иллюстрации использования этого варианта оператора CREATE DATABASE создадим для начала следующую базу данных (пример 3.38). Заодно потренируемся в создании многофайловой базы данных с двумя файловыми группами. Этот опыт нам сможет пригодиться в дальнейшей деятельности.

Перед выполнением этого скрипта на диске D: нужно создать каталог Attach и в нем два подкаталога — A1 и A2.

Пример 3.38. Создание базы данных с множеством файлов и двумя файловыми группами

```

USE master;
GO
IF DB_ID('ForAttach1') IS NOT NULL
    DROP DATABASE ForAttach1;
GO

CREATE DATABASE ForAttach1
ON PRIMARY (NAME = ForAttach0,
    FILENAME = 'D:\Attach\A1\ForAttach0.mdf'),
    (NAME = ForAttach1,
    FILENAME = 'D:\Attach\A1\ForAttach1.ndf'),
FILEGROUP SECOND
(NAME = ForAttach2,
    FILENAME = 'D:\Attach\A2\ForAttach2.ndf'),
    (NAME = ForAttach3,
    FILENAME = 'D:\Attach\A2\ForAttach3.ndf')
LOG ON (NAME = ForAttach1_log,
    FILENAME = 'D:\Attach\ForAttach1.ldf'),
    (NAME = ForAttach2_log,
    FILENAME = 'D:\Attach\ForAttach2.ldf');
GO
    
```

Выполните скрипт в утилите sqlcmd или в Management Studio. В указанных каталогах будут созданы четыре файла данных и два файла журнала транзакций. Чтобы в Management Studio увидеть вновь созданную базу данных, нужно правой кнопкой мыши щелкнуть по строке **Databases** в **Object Explorer** и в контекстном меню выбрать элемент **Refresh**.

Теперь переведите базу данных ForAttach1 в состояние OFFLINE, щелкнув по ней правой кнопкой мыши и выбрав в контекстном меню **Tasks | Take Offline**. Скопируйте все файлы базы данных средствами операционной системы на диск D: в каталог AttachNew с каталогами A1 и A2. Удалите базу данных ForAttach1 средствами Management Studio. Вы можете убедиться, что база данных удаляется только из каталога сервера базы данных; сами же файлы остаются нетронутыми.

Чтобы присоединить эту скопированную в другой каталог базу данных к текущему экземпляру сервера с использованием варианта FOR ATTACH, выполните следующий скрипт (пример 3.39).

Пример 3.39. Присоединение базы данных

```
USE master;
GO
CREATE DATABASE ForAttach1
ON (NAME = ForAttach0,
    FILENAME = 'D:\AttachNew\A1\ForAttach0.mdf'),
    (NAME = ForAttach1,
    FILENAME = 'D:\AttachNew\A1\ForAttach1.ndf'),
    (NAME = ForAttach2,
    FILENAME = 'D:\AttachNew\A2\ForAttach2.ndf'),
    (NAME = ForAttach3,
    FILENAME = 'D:\AttachNew\A2\ForAttach3.ndf'),
    (NAME = ForAttach1_log,
    FILENAME = 'D:\AttachNew\ForAttach1.ldf'),
    (NAME = ForAttach2_log,
    FILENAME = 'D:\AttachNew\ForAttach2.ldf')
FOR ATTACH;
GO
```

База данных со всеми ее файлами будет присоединена к базам данных сервера.

В среде SQL Server Management Studio переведите базу данных в состояние OFFLINE и удалите базу данных. Удалите или переименуйте оба файла журнала транзакций: ForAttach1.ldf и ForAttach2.ldf.

Выполним присоединение базы данных в варианте FOR ATTACH_REBUILD_LOG. Введите и выполните скрипт, как показано в примере 3.40.

Пример 3.40. Присоединение базы данных с пересозданием журнала транзакций

```
USE master;
GO
CREATE DATABASE ForAttach2
ON (NAME = ForAttach0,
    FILENAME = 'D:\AttachNew\A1\ForAttach0.mdf'),
    (NAME = ForAttach1,
    FILENAME = 'D:\AttachNew\A1\ForAttach1.ndf'),
    (NAME = ForAttach2,
    FILENAME = 'D:\AttachNew\A2\ForAttach2.ndf'),
    (NAME = ForAttach3,
    FILENAME = 'D:\AttachNew\A2\ForAttach3.ndf')
FOR ATTACH_REBUILD_LOG;
GO
```

В результате выполнения скрипта будет присоединена база данных со всеми файлами данных и будут выданы предупреждающие сообщения об отсутствии двух файлов журнала транзакций. В последнем сообщении говорится, что создан файл журнала транзакций 'D:\AttachNew\ForAttach2_log.LDF'.

ВНИМАНИЕ!

Вариант присоединения базы данных FOR ATTACH REBUILD LOG рекомендуется использовать для баз данных, к которым будут применяться только операции чтения. Дело в том, что в этом случае происходит разрыв цепочек связанных резервных копий журнала.

3.7.2. Присоединение базы данных с использованием диалоговых средств Management Studio

Средствами Management Studio можно выполнить присоединение базы данных только при наличии всех файлов базы данных, как данных, так и журналов транзакций. Здесь не проходит вариант присоединения базы данных с пересозданием журнала транзакций.

Сейчас в Management Studio можно отключить базу данных ForAttach2 и заново присоединить ее к базам данных экземпляра сервера.

Переведите базу данных в состояние OFFLINE. В **Object Explorer** щелкните правой кнопкой мыши по имени базы **ForAttach2** и в контекстном меню выберите **Tasks | Take Offline**. Появится окно подтверждения того, что база данных переводится в состояние OFFLINE. Щелкните по кнопке **Close**.

Удалите базу данных ForAttach2, щелкнув правой кнопкой мыши по ее имени в **Object Explorer** и выбрав в контекстном меню элемент **Delete**. Появится диалоговое окно удаления объекта, где нужно щелкнуть мышью по кнопке **OK**.

База данных будет удалена из каталога текущего экземпляра сервера, но сами файлы сохранятся на диске, поскольку в момент удаления база данных находилась в состоянии OFFLINE.

Затем нужно щелкнуть правой кнопкой мыши в окне **Object Explorer** по элементу **Databases** и выбрать в контекстном меню элемент **Attach**. Появится окно присоединения базы данных **Attach Databases** (рис. 3.44).

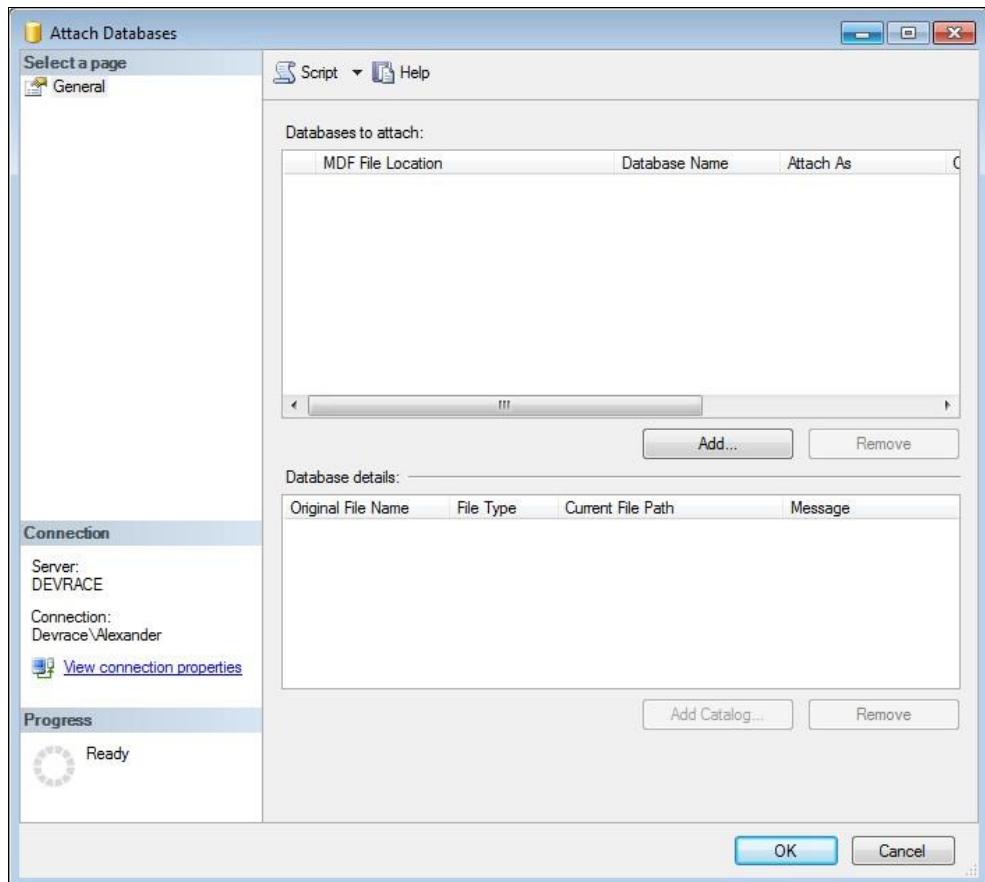


Рис. 3.44. Окно присоединения базы данных

Здесь нужно щелкнуть по кнопке **Add** и в диалоговом окне выбора первичного файла базы данных выбрать на диске D: в каталоге **AttachNew\A1** файл **ForAttach0.mdf**.

После щелчка по кнопке **OK** появится список всех файлов присоединяемой базы данных (рис. 3.45).

После щелчка по кнопке **OK** база данных будет присоединена к списку существующих баз данных текущего экземпляра сервера. Чтобы увидеть ее в списке баз данных в **Object Explorer**, нужно щелкнуть правой кнопкой мыши по строке **Databases** и в контекстном меню выбрать элемент **Refresh**.

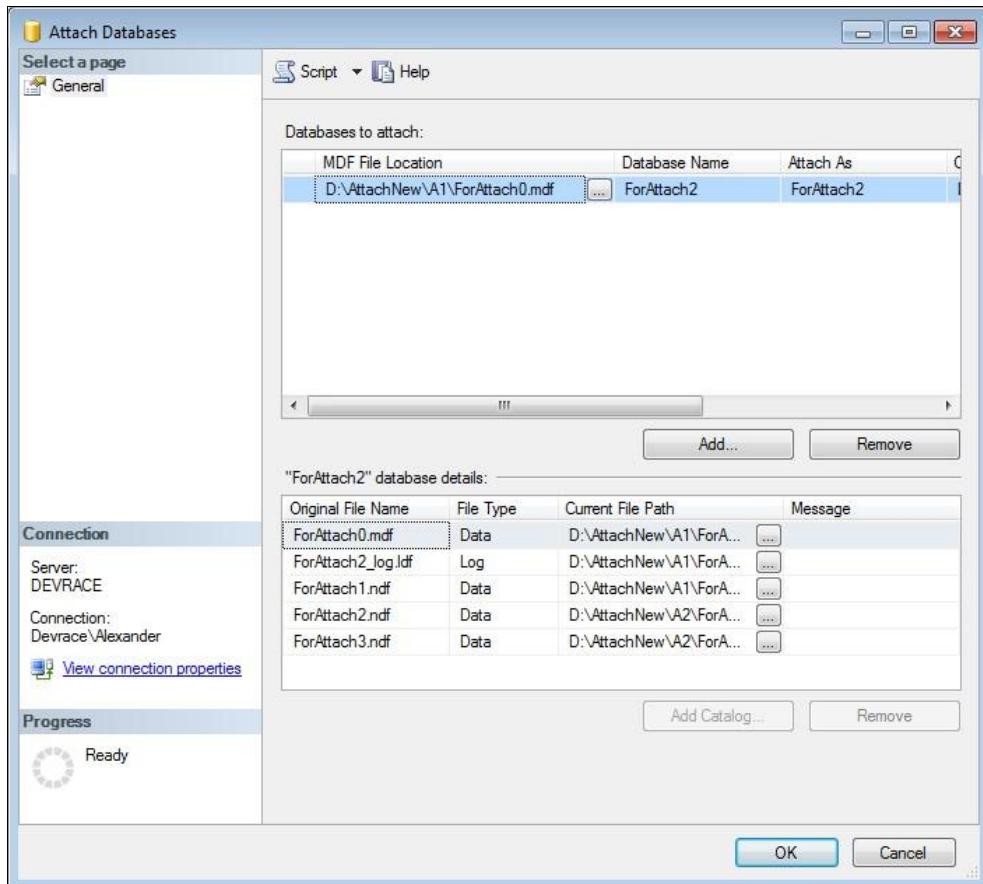


Рис. 3.45. Список файлов присоединяемой базы данных

3.7.3. Отсоединение базы данных

Базу данных можно отсоединить от экземпляра сервера при вызове системной процедуры `sp_detach_db` или в диалоговых средствах Management Studio. База данных отсоединяется от текущего экземпляра сервера базы данных, а файлы базы данных не удаляются с внешнего носителя.

Хранимая процедура `sp_detach_db` может быть вызвана в двух вариантах: когда параметры задаются позиционно или когда используются ключевые параметры.

При позиционном задании параметров синтаксис вызова процедуры выглядит следующим образом:

```
EXECUTE sp_detach_db '<имя базы данных>' [, '<статистика>'];
```

Имя базы данных задает логическое имя отсоединяемой базы данных.

Параметр `<статистика>` указывает, нужно ли обновить статистические данные для оптимизации запросов к базе данных. Может иметь значение `true` (значение по

умолчанию), чтобы были выполнены действия по перерасчету статистики. Рекомендуется использовать для баз данных, которые будут перемещаться на носители только для чтения. Значение `false` не производит автоматического запуска расчета статистики.

В случае ключевых параметров перед соответствующим значением параметра используется ключевое слово, за которым будет следовать знак равенства.

- ◆ `@dbname` — имя отсоединяемой базы данных.
- ◆ `@skipcheckks` — необходимость обновления статистики.

ЗАМЕЧАНИЕ

В предыдущих версиях SQL Server присутствовал и третий параметр процедуры, `@keepfulltextindexfile`. В документации сообщается, что этот параметр будет удален в следующих версиях системы.

Процедура возвращает значение 0 при успешном отсоединении и 1 при ошибках.

Отсоедините базу данных `ForAttach2`, выполнив следующий оператор:

```
EXECUTE sp_detach_db 'ForAttach2', 'true';
GO
```

Для отсоединения базы данных в диалоговых средствах Management Studio нужно щелкнуть правой кнопкой мыши по имени базы данных в **Object Explorer** и в меню выбрать **Tasks | Detach**. Появится окно отсоединения базы данных. На рис. 3.46 показаны простые сведения о базе данных. Здесь можно отметить флажок **Update Statistics** для выполнения перерасчета статистики.

В этом окне нужно щелкнуть мышью по кнопке **OK**.

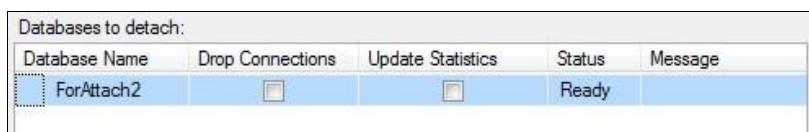


Рис. 3.46. Отсоединение базы данных

3.8. Создание мгновенных снимков базы данных

В SQL Server существует возможность создания так называемых *мгновенных* (иногда говорят *моментальных*) *снимков* (*snapshot*) базы данных.

Мгновенные снимки — это своеобразное средство фиксации состояния базы данных (базы данных-источника) на конкретный момент времени, причем они не требуют выполнения объемного копирования текущего состояния данных, а позволяют сохранять существовавшие данные в страницах файлов до внесения в эти данные изменений.

Основным назначением мгновенных снимков является составление отчетов на конкретный момент времени и восстановление базы данных в случае возникновения ошибок пользователя.

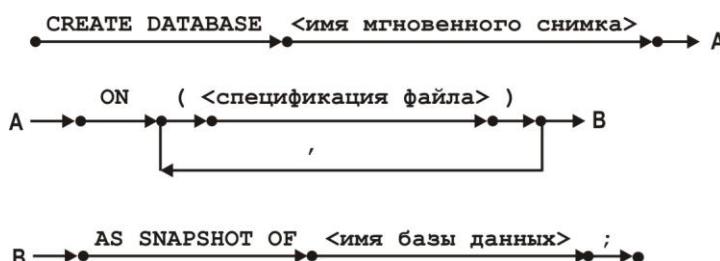
Мгновенные снимки работают на уровне страниц. В момент создания мгновенного снимка в его файлы ничего не записывается. Когда в базе данных-источнике выполняется изменение данных, в мгновенный снимок записывается страница до внесения этих изменений. Очень симпатичные цветные иллюстрации использования мгновенных снимков представлены в документе Books Online. Если хотите ими полюбоваться, то в строке поиска панели **Look for:** наберите "CREATE DATABASE, инструкция" (для русскоязычного варианта), найдите в тексте гиперссылку "моментальный снимок" и пару раз перейдите к нужному кадру. Чтобы получить подобную информацию, также можно в Интернете перейти на соответствующую страницу.

Для создания мгновенного снимка существующей базы данных используется следующий вариант оператора CREATE DATABASE (листинг 3.11 и граф 3.15).

Листинг 3.11. Синтаксис оператора CREATE DATABASE.

Вариант создания мгновенного снимка базы данных

```
CREATE DATABASE <имя мгновенного снимка>
    ON (<спецификация файла>) [, (<спецификация файла>)] ...
    AS SNAPSHOT OF <имя базы данных>;
```



Граф 3.15. Синтаксис оператора CREATE DATABASE в варианте создания мгновенного снимка

В операторе задается имя мгновенного снимка, имя базы данных, для которой создается мгновенный снимок, и спецификации файлов.

В спецификации файла здесь можно указывать два предложения: NAME, которое задает имя логического файла базы данных-источника, и предложение FILENAME, задающее путь и имя физического файла для мгновенного снимка.

Выполните создание мгновенного снимка для базы данных BestDatabase, как это показано в примере 3.41.

Пример 3.41. Создание мгновенного снимка для базы данных BestDatabase

```
USE master;
GO
CREATE DATABASE SnapshotForBestDatabase
```

```

ON (NAME = BestDatabase_dat,
    FILENAME = 'D:\BestDatabase\Snapshot.mdf')
AS SNAPSHOT OF BestDatabase;
GO

```

В результате выполнения этого скрипта в каталоге BestDatabase будет создан файл мгновенного снимка Snapshot.mdf. В панели **Object Explorer** в папке **Database Snapshots** появится папка только что созданного мгновенного снимка **Snapshot-ForBestDatabase**. Чтобы просмотреть файлы мгновенного снимка, нужно как обычно щелкнуть правой кнопкой мыши по этой строке и в контекстном меню выбрать элемент **Properties**.

Вкладка **Files** мгновенного снимка показана на рис. 3.47.

Database files:					
Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize	Path
BestDatabase_dat	Rows ...	PRIMARY	5	By 1 MB, Unlimited	D:\BestDatabase

Рис. 3.47. Список файлов мгновенного снимка базы данных BestDatabase

3.9. Схемы базы данных

Физически база данных представлена в одном или более файлах данных. Логически база состоит из произвольного количества *схем* (schema). В схемах хранятся объекты базы данных, такие как таблицы, представления. Каждый объект базы данных принадлежит одной и только одной схеме. Если при создании объекта базы данных явно не указывается, какой схеме он принадлежит, то он помещается в схему по умолчанию, обычно **dbo** (не путайте с пользователем, который также имеет имя **dbo**). Схему можно рассматривать как пространство имен (namespace). Имена объектов одного типа в схеме должны быть, разумеется, уникальными. При этом объекты с одинаковыми именами могут присутствовать в различных схемах одной и той же базы данных.

Основным назначением схемы является повышение уровня безопасности данных с тем, чтобы предоставить права на использование объектов базы только указанным группам пользователей. Таких пользователей называют *принципалами* (principal). Схема также позволяет выполнить "упаковку" групп объектов, что существенно облегчает манипулирование привилегиями.

При создании базы данных в ней автоматически создается не менее дюжины схем. Схемы можно создавать, используя операторы Transact-SQL или диалоговые средства Management Studio.

3.9.1. Работа со схемами в Transact-SQL

Для создания схемы используется оператор `CREATE SCHEMA`. Для изменения схемы применяется оператор `ALTER SCHEMA`. Изменение схемы — это всего лишь переме-

щение объектов из одной схемы в другую. Удаляется схема оператором `DROP SCHEMA`. Удалить можно только схему, которая не содержит объектов.

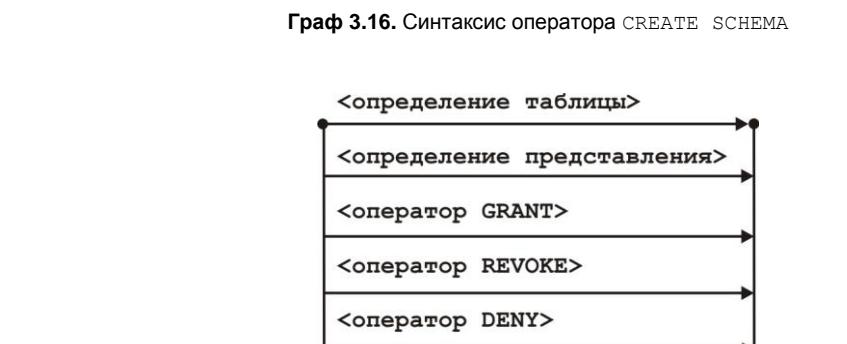
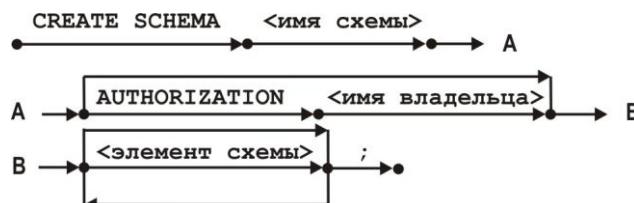
Создание схемы

Синтаксис оператора создания схемы `CREATE SCHEMA` представлен в листинге 3.12 и в соответствующем R-графе (графы 3.16 и 3.17).

Листинг 3.12. Синтаксис оператора `CREATE SCHEMA`

```
CREATE SCHEMA <имя схемы>
    [AUTHORIZATION <имя владельца>]
    [<элемент схемы> ...] ;
```

```
<элемент схемы> ::= 
{   <определение таблицы>
| <определение представления>
| <оператор GRANT>
| <оператор REVOKE>
| <оператор DENY>
}
```



Имя схемы должно быть уникальным в текущей базе данных. Необязательное предложение `AUTHORIZATION` задает владельца схемы.

Сразу в процессе создания схемы в ней можно сформировать объекты (таблицы, представления). Можно также предоставить полномочия (оператор `GRANT`), удалить

полномочия (оператор REVOKE) или запретить наследование полномочий (оператор DENY).

Все операторы, используемые при создании схемы, выполняются как единое целое. Если в процессе создания схемы возникает ошибка, то действия всех операторов отменяются.

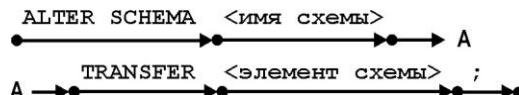
Признаком завершения группы операторов, относящихся к созданию одной схемы, является оператор GO.

Изменение схемы

Перемещение объектов между схемами одной базы данных осуществляется с помощью оператора ALTER SCHEMA. Синтаксис оператора см. в листинге 3.13 и в соответствующем R-графе (граф 3.18).

Листинг 3.13. Синтаксис оператора ALTER SCHEMA

```
ALTER SCHEMA <имя схемы>
    TRANSFER <элемент схемы>;
```



Имя перемещаемого элемента из другой схемы должно состоять из имени схемы и имени объекта, которые разделены точкой.

Удаление схемы

Удаление пустой схемы выполняется оператором DROP SCHEMA (листинг 3.14 и граф 3.19).

Листинг 3.14. Синтаксис оператора DROP SCHEMA

```
DROP SCHEMA <имя схемы>;
```



Граф 3.19. Синтаксис оператора DROP SCHEMA

Пример создания схем

Создайте в базе данных BestDatabase две схемы, выполнив операторы примера 3.42.

Пример 3.42. Создание двух схем в базе данных BestDatabase

```
USE BestDatabase;
GO
CREATE SCHEMA SmartPersons;
GO
CREATE SCHEMA StupidPersons;
GO
```

Обратите внимание, что между двумя операторами CREATE SCHEMA присутствует оператор GO. Если вы его не укажете, то получите ошибку.

Чтобы отобразить список схем базы данных, используется системное представление sys.schemas. Оно возвращает значения трех столбцов: name — имя схемы, schema_id — идентификатор схемы и principal_id — идентификатор принципала, владельца схемы. Отобразите схемы базы данных BestDatabase (пример 3.43).

Пример 3.43. Отображение списка схем базы данных BestDatabase

```
USE BestDatabase;
GO
SELECT CAST(name AS VARCHAR(19)) AS 'Schema Name',
       schema_id AS 'Schema ID',
       principal_id AS 'Principal ID'
FROM sys.schemas;
GO
```

Результат отображения списка схем базы данных BestDatabase:

Schema Name	Schema ID	Principal ID
dbo	1	1
guest	2	2
INFORMATION_SCHEMA	3	3
sys	4	4
SmartPersons	5	1
StupidPersons	6	1
db_owner	16384	16384
db_accessadmin	16385	16385
db_securityadmin	16386	16386
db_ddladmin	16387	16387
db_backupoperator	16389	16389
db_datareader	16390	16390
db_datawriter	16391	16391
db_denydatareader	16392	16392
db_denydatawriter	16393	16393

(15 row(s) affected)

Теперь удалите созданные две схемы (пример 3.44).

Пример 3.44. Удаление ранее созданных схем в базе данных BestDatabase

```
USE BestDatabase;
GO
DROP SCHEMA SmartPersons;
DROP SCHEMA StupidPersons;
GO
```

Отобразите опять список схем базы данных BestDatabase. Можно увидеть, что эти две схемы из списка исчезли.

В следующем подразделе мы опять создадим и удалим эти две схемы, но уже с использованием диалоговых средств Management Studio.

3.9.2. Работа со схемами в Management Studio

Отображение схем базы данных

Чтобы в Management Studio просмотреть список схем базы данных BestDatabase, нужно в **Object Explorer** раскрыть список **Databases**, раскрыть базу данных BestDatabase, папку **Security** и папку **Schemas**. Появится список схем базы данных.

Создание схемы

Чтобы создать новую схему, нужно в окне **Object Explorer** щелкнуть правой кнопкой мыши по элементу **Schemas** в базе данных и в контекстном меню выбрать строку **New Schema**. Появится окно создания новой схемы (рис. 3.48). В поле **Schema name** (имя схемы) введите имя создаваемой схемы: **SmartPersons**.

Имя владельца схемы (поле **Schema owner**) можно не вводить. Туда будет подставлено имя пользователя по умолчанию — **dbo**. Щелкните по кнопке **OK**.

Аналогичным образом создайте вторую схему **StupidPersons**.

Чтобы увидеть вновь созданные схемы в **Object Explorer**, нужно щелкнуть правой кнопкой мыши по элементу **Schemas** и в контекстном меню выбрать строку **Refresh**.

Изменение существующей схемы

Диалоговыми средствами Management Studio можно изменить владельца существующей схемы. Для этого в **Object Explorer** дважды щелкните мышью по схеме или щелкните правой кнопкой мыши по имени схемы и в контекстном меню выберите элемент **Properties**. Появится окно, похожее на окно создания новой схемы.

Чтобы изменить владельца схемы, в поле **Schema owner** щелкните по кнопке **Search**. Появится диалоговое окно выбора владельца схемы — пользователя или роли (рис. 3.49).

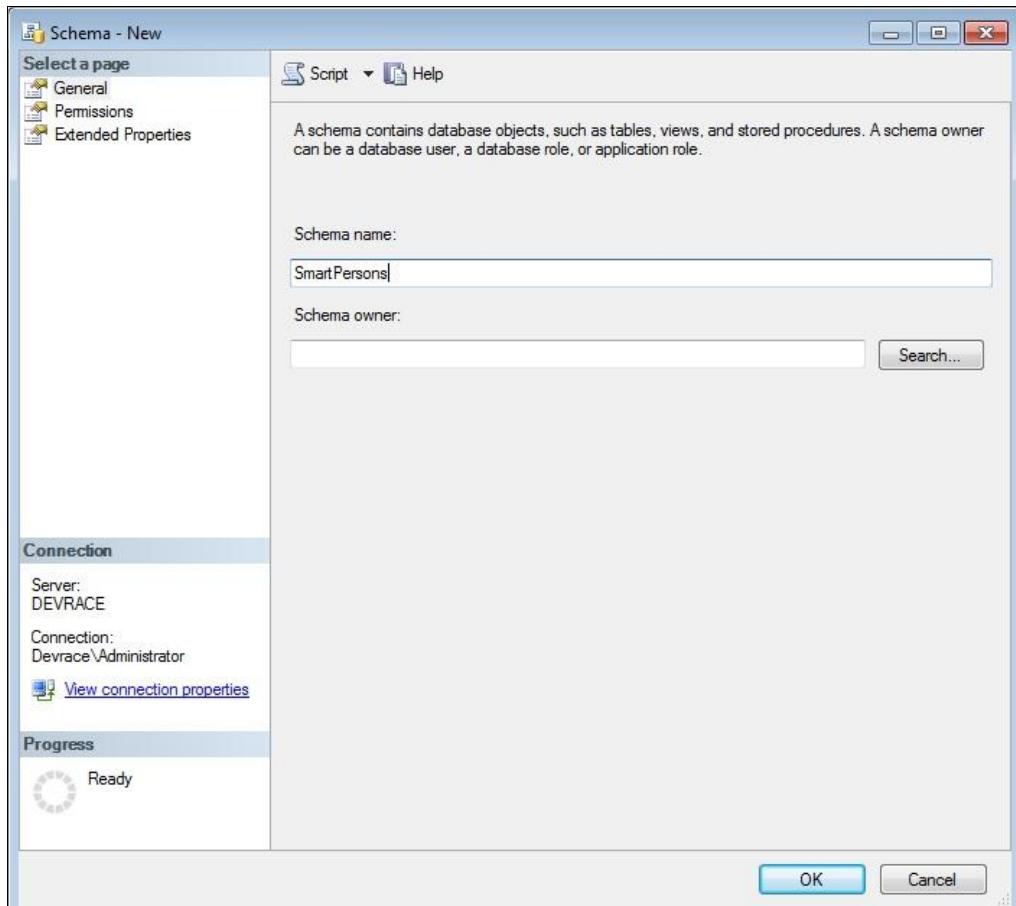


Рис. 3.48. Создание новой схемы в базе данных BestDatabase



Рис. 3.49. Окно выбора владельца схемы

Щелкните по кнопке **Browse** (просмотр). Появится окно просмотра объектов базы данных, которые могут быть использованы в качестве владельцев схемы. Отметьте флажок в строке [public] и щелкните по кнопке **OK** (рис. 3.50).

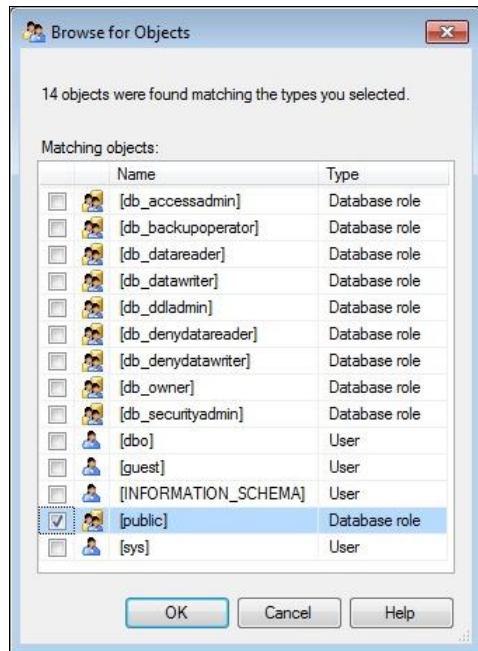


Рис. 3.50. Список возможных владельцев схемы

В окне выбора владельца схемы щелкните по кнопке **OK**. В результате для схемы будет выбран новый владелец **public**. Чтобы сохранить изменения, в окне просмотра свойств схемы щелкните по кнопке **OK**.

Удаление схемы

Удалите поочередно обе созданные схемы. Для этого в **Object Explorer** щелкните по имени удаляемой схемы правой кнопкой мыши и в контекстном меню выберите элемент **Delete** или нажмите клавишу <Delete>. В окне подтверждения удаления схемы щелкните мышью по кнопке **OK**. Схема будет удалена.

3.10. Средства копирования и восстановления баз данных

При интенсивном изменении данных в базе данных эту базу данных нужно регулярно копировать, чтобы в случае сбоев в данных и в оборудовании было можно восстановить данные. Рекомендуется также выполнять резервное копирование базы данных **master** после создания новых баз данных в экземпляре сервера.

Для копирования и восстановления базы данных можно использовать операторы языка Transact-SQL **BACKUP** и **RESTORE** или диалоговые средства Management Studio. Мы рассмотрим только варианты полного копирования и восстановления.

3.10.1. Использование операторов копирования/восстановления базы данных

Для копирования всей базы данных на конкретный носитель используется оператор `BACKUP`. Его синтаксис (только для полного копирования всей базы данных) показан в листинге 3.15 и в соответствующем R-графе (граф 3.20).

Листинг 3.15. Синтаксис оператора `BACKUP`

```
BACKUP DATABASE <имя базы данных>
TO DISK = '<спецификация файла>' [, DISK = '<спецификация файла>'] ...;
```



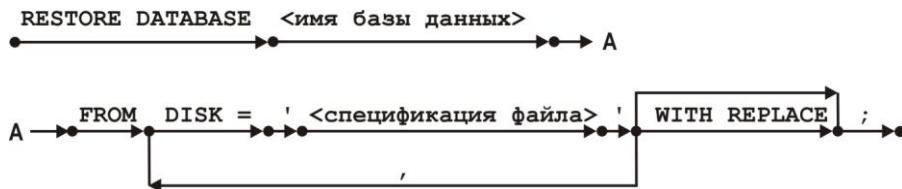
Граф 3.20. Синтаксис оператора `BACKUP`

Этот оператор задает копирование всех файлов базы данных в файлы, заданные после ключевого слова `DISK`. В спецификации файла нужно указать имя диска, каталог и имя файла копии. Имя файла по традиции должно иметь расширение `bak`, хотя это и не обязательно.

Для восстановления базы данных из резервных копий используется оператор `RESTORE`. Его упрощенный синтаксис для полного восстановления базы данных показан в листинге 3.16 и в соответствующем R-графе (граф 3.21).

Листинг 3.16. Синтаксис оператора `RESTORE`

```
RESTORE DATABASE <имя базы данных>
FROM DISK = '<спецификация файла>'
[, DISK = '<спецификация файла>']... [ WITH REPLACE ];
```



Граф 3.21. Синтаксис оператора `RESTORE`

В предложении `FROM` нужно указать имена всех файлов созданной копии и пути к этим файлам.

Опция **WITH REPLACE** указывает, что при восстановлении будут заменяться существующие файлы базы данных.

Важно!

Необходимо иметь в виду, что средства копирования и восстановления в SQL Server 2012 несовместимы с более ранними версиями.

3.10.2. Использование диалоговых средств Management Studio для копирования/восстановления базы данных

Для копирования базы данных **BestDatabase** в **Object Explorer** щелкните правой кнопкой мыши по имени базы данных, в контекстном меню выберите **Tasks | Back Up**. Появится окно **Back Up Database** (рис. 3.51).

Здесь нужно указать путь и имя файла (имена файлов), куда нужно выполнять копирование. Система предлагает путь и имя файла, которые нам не подходят. Поэтому щелкните по кнопке **Remove**, чтобы удалить этот вариант.

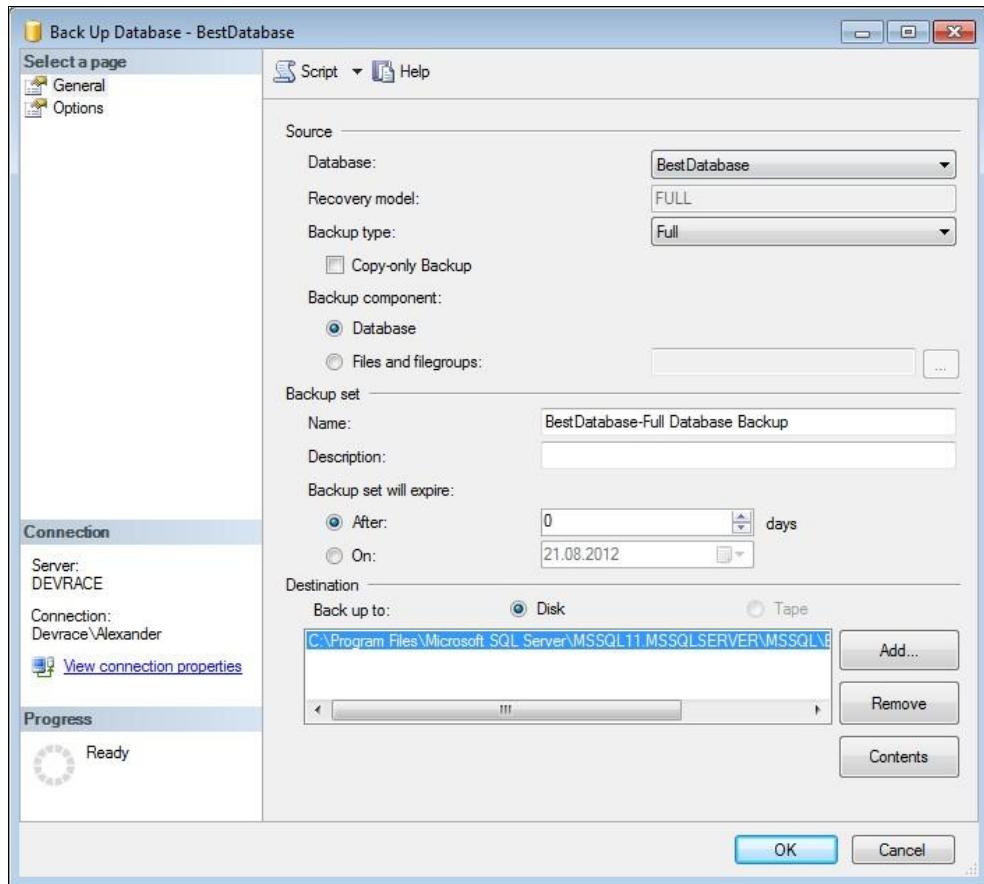


Рис. 3.51. Окно копирования базы данных **Back Up Database**

Затем щелкните по кнопке **Add**, чтобы задать свой диск и свой файл. Появится окно **Select Backup Destination** (рис. 3.52).

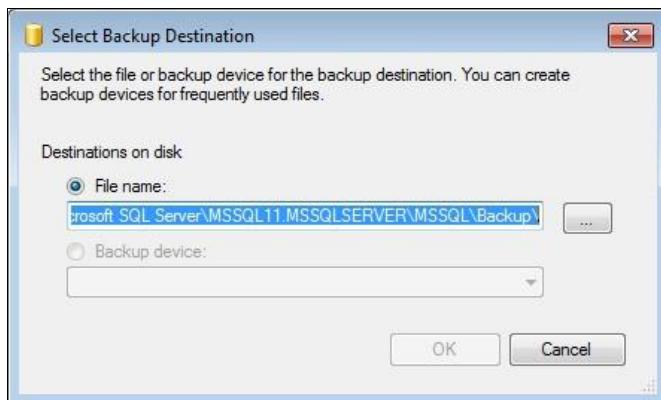


Рис. 3.52. Окно выбора файлов копии для базы данных **Select Backup Destination**

Здесь нужно щелкнуть по кнопке с многоточием. Появится окно **Locate Database Files**, в котором надо задать размещение и имя файла копии. Диск и каталог выбираются из списка, как это показано на рис. 3.53.

Имя файла копии вводится в нижней части окна (рис. 3.54).

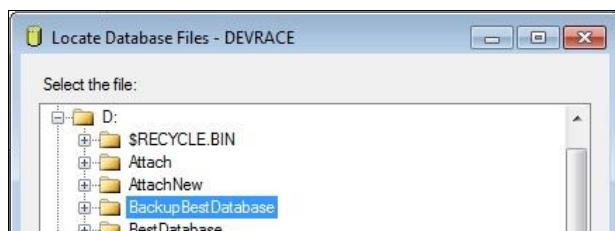


Рис. 3.53. Окно выбора размещения файлов копии **Locate Database Files**

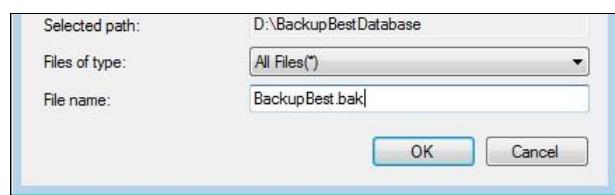


Рис. 3.54. Задание имени файла копии

После щелчка по кнопке **OK** будет выполнено копирование и появится сообщение об успешном завершении копирования (рис. 3.55).

Для восстановления базы данных **BestDatabase** в **Object Explorer** щелкните правой кнопкой мыши по имени базы данных, в контекстном меню выберите **Tasks | Restore | Database**. Появится окно **Restore Database**. Здесь нужно перейти на

вкладку **Options**, на которой установить флажок **Overwrite the existing database (WITH REPLACE)** и снять флажок в **Take tail-log backup before restore** (рис. 3.56).

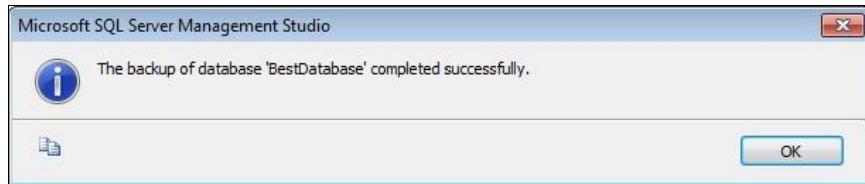


Рис. 3.55. Сообщение об успешном завершении создания копии

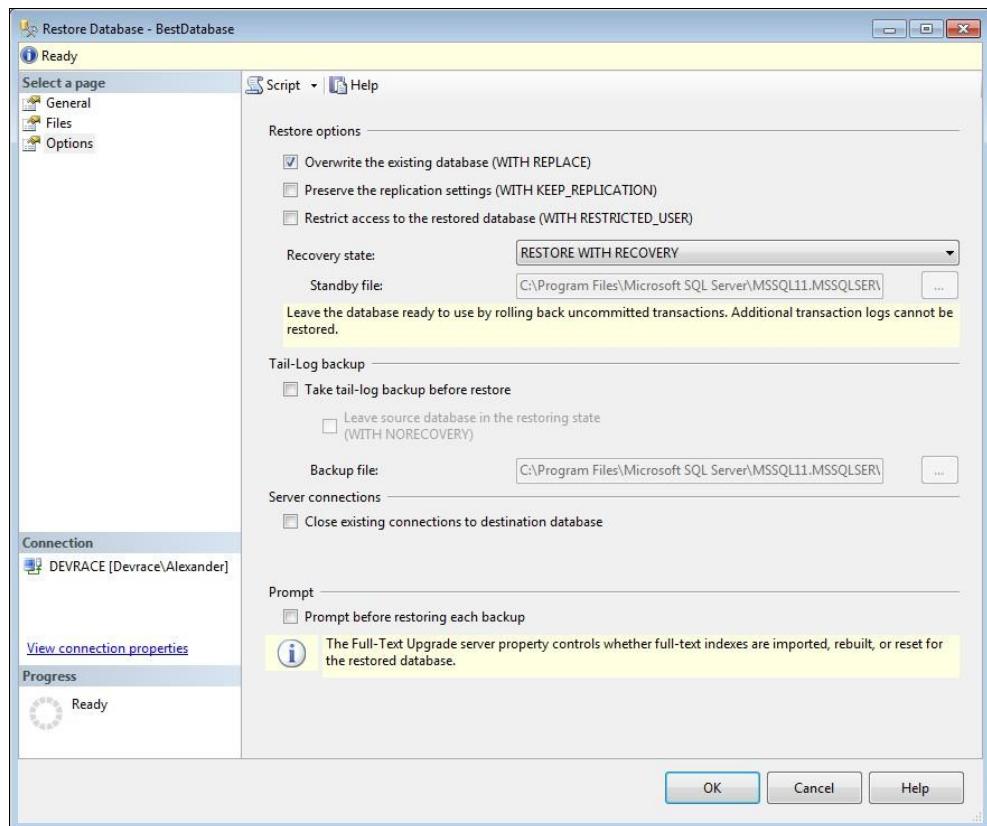


Рис. 3.56. Восстановление базы данных. Вкладка Options

На вкладке **General** нужно выбрать переключатель **Database** и из раскрывающегося списка выбрать имя восстанавливаемой базы данных (рис. 3.57).

На вкладке **Files** (рис. 3.58) можно изменить пути и имена физических файлов базы данных в поле **Restore As**.

После щелчка по кнопке **OK** будет выполнено восстановление и появится соответствующее информационное сообщение (рис. 3.59).

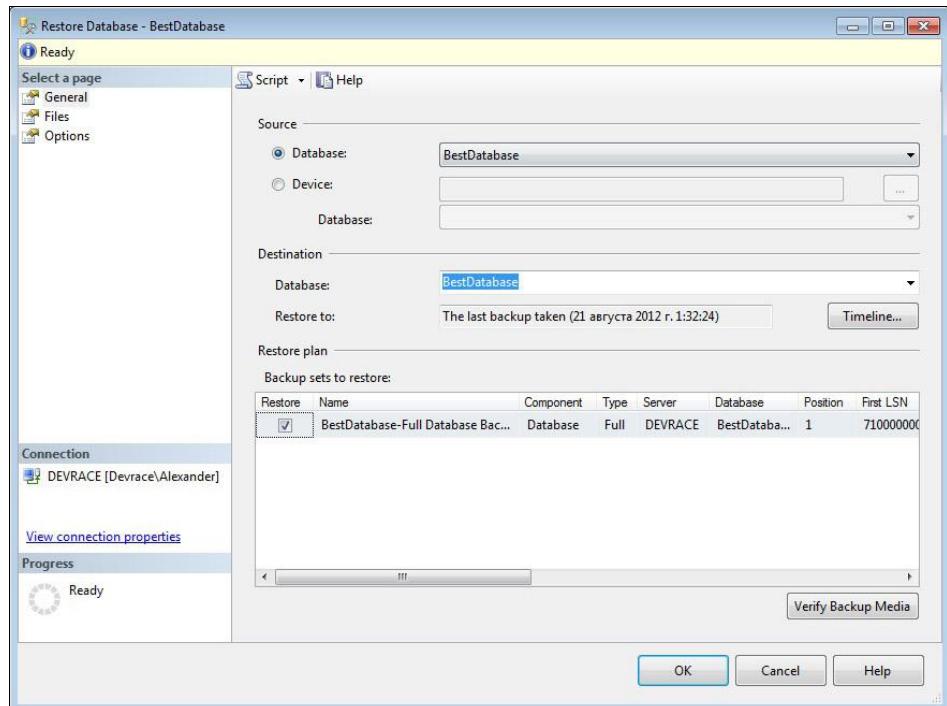


Рис. 3.57. Восстановление базы данных. Вкладка General

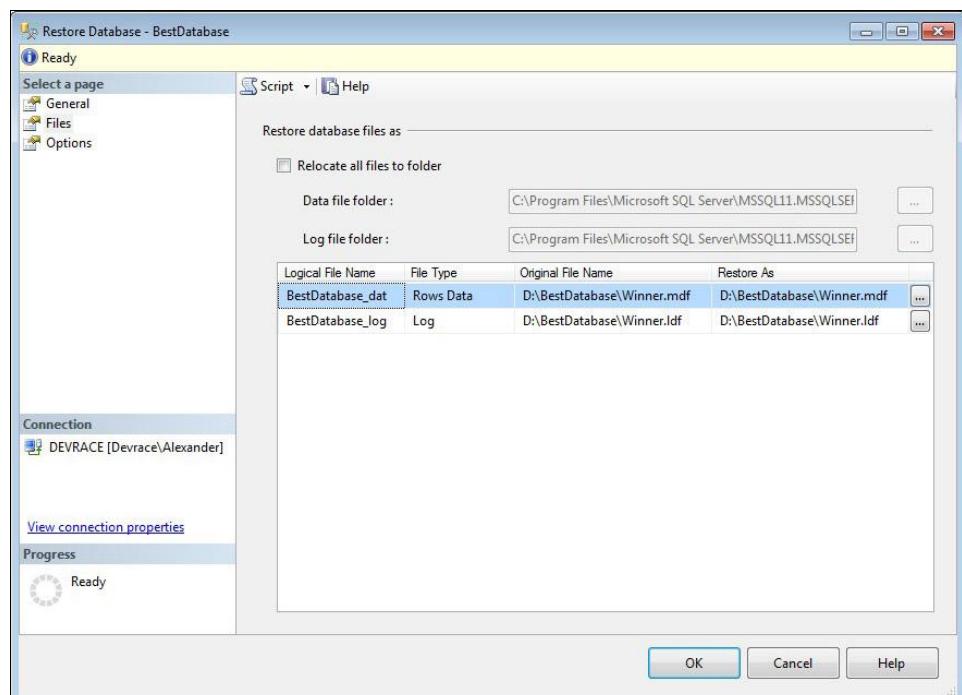


Рис. 3.58. Восстановление базы данных. Вкладка Files



Рис. 3.59. Сообщение об успешном восстановлении базы данных

3.11. Домашнее задание

Сейчас предлагаю выполнить ряд действий по созданию базы, которая может вам пригодиться в реальной жизни.

Если у вас есть хоть немного свободного времени, то выполните ближе к ночи на вашем компьютере следующую работу.

Создайте базу данных с именем, скажем, *MyDatabase*, которая содержит три файловые группы, в каждой из которых будут располагаться по два файла данных. Если у вас на компьютере есть несколько дисков, то поместите файлы этих файловых групп на разные диски. По вашему усмотрению задайте размеры файлов и параметры увеличения их размеров. Создайте для базы данных два файла журнала транзакций, разместив их по возможности на устройствах, отличных от устройств, на которых находятся файлы данных.

Для вновь созданной базы данных создайте две схемы.

Не спеша выполните такие действия с использованием операторов Transact-SQL и диалоговых средств Management Studio.

Пришлите мне операторы создания всех этих объектов. Я с благодарностью использую их в своих будущих разработках.

Что будет дальше?

Самым важным и сложным объектом базы данных являются таблицы. Однако прежде чем приступить к их созданию, мы подробнейшим образом рассмотрим фундаментальное понятие программирования — типы данных.



ГЛАВА 4

Типы данных

- ◆ Классификация типов данных в SQL Server
- ◆ Использование типов данных. Примеры работы с данными различных типов
- ◆ Объявление локальных переменных, курсоров, табличного типа данных
- ◆ Системные функции работы с данными
- ◆ Создание пользовательских типов данных

Тип данных — важнейшее понятие в программировании, он определяет главную характеристику элемента данных, будь то локальная переменная, параметр в хранимой процедуре или в программе или столбец таблицы в базе данных. Тип данных определяет множество допустимых значений для элемента данных и множество допустимых операций, применимых к элементу данных. Кроме того, тип данных определяет и способ внутреннего хранения соответствующего элемента, т. е. физический аспект.

В данной главе описываются все системные типы данных SQL Server. Приводятся допустимые операции над типами данных, преобразования данных, применяемые функции. Дается синтаксис операторов создания и удаления пользовательских типов данных.

Некоторые типы данных с целью совместимости со стандартом SQL имеют синонимы. Эти синонимы также приводятся для соответствующих типов данных.

Материала в данной главе достаточно много. Если вы впервые знакомитесь с типами данных в реляционных базах данных, то при первом (или единственном) прочтении этой главы имеет смысл подробно рассмотреть "классические" типы данных — числовые, строковые и, возможно, дату и время. А к таким экзотическим как пространственные типы или XML можно будет обратиться потом, если появится реальная в них потребность.

4.1. Классификация типов данных в SQL Server

Типы данных SQL можно сгруппировать. В документации по SQL Server принято типы данных объединять в следующие группы:

- ◆ *Числовые данные* (Numeric). Можно разделить на две подгруппы:
 - *точные числа*, или *числа с фиксированной точностью* (Exact Numerics). Сюда включены типы данных:
 - BIT;
 - TINYINT;
 - SMALLINT;
 - INT;
 - BIGINT;
 - NUMERIC;
 - DECIMAL;
 - SMALLMONEY;
 - MONEY.
 - *приближенные числа*, или *числа с плавающей точкой* (Approximate Numerics, Float); включают типы данных:
 - FLOAT;
 - REAL.
- ◆ *Символьные данные* (Character). Включает две подгруппы:
 - *обычные символьные строки* (Character Strings). Сюда включены следующие типы данных:
 - CHAR;
 - VARCHAR;
 - TEXT.
 - *символьные строки в Юникоде* (Unicode Character Strings). Типы данных:
 - NCHAR;
 - NVARCHAR;
 - NTEXT.
- ◆ *Дата и время* (Date and Time). Содержит следующие типы данных:
 - DATETIME;
 - SMALLDATETIME;
 - DATE;
 - TIME;
 - DATETIMEOFFSET;
 - DATETIME2.

◆ *Двоичные строки* (Binary Strings). В эту группу включены следующие типы данных:

- BINARY;
- VARBINARY;
- IMAGE.

◆ *Пространственные типы данных* (Spatial Data Types). Относятся следующие типы данных:

- GEOMETRY;
- GEOGRAPHY.

◆ *Другие типы данных* (Other Data Types). Сюда включены типы данных, не очень подходящие для других категорий. В эту группу включены такие типы данных:

- SQL_VARIANT;
- TIMESTAMP;
- UNIQUEIDENTIFIER;
- HIERARCHYID;
- CURSOR;
- TABLE;
- XML.

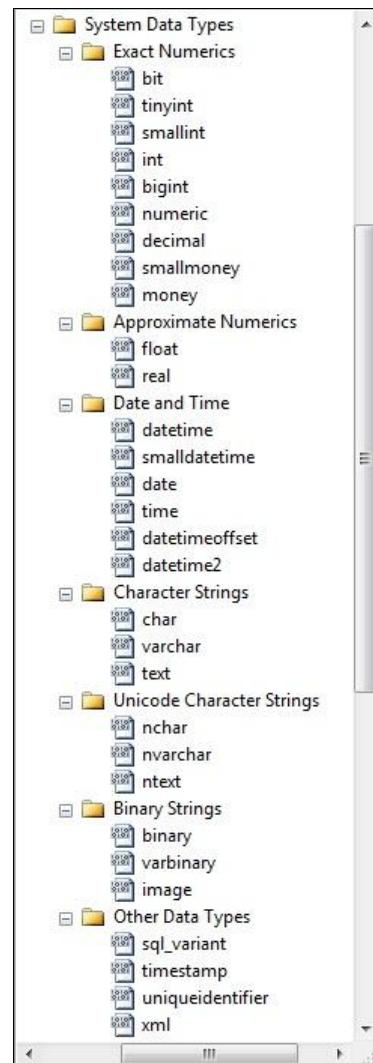


Рис. 4.1. Список типов данных, отображаемых в Management Studio

ЗАМЕЧАНИЕ

Приведенная здесь классификация типов данных в основном соответствует тому, что указано в документации по SQL Server. Однако с моей точки зрения она не является безупречной. Например, тип данных `BIT`, я считаю, следовало бы выделить в отдельную группу или отнести его к группе *другие типы данных* (что, кстати, я как-то увидел в одном из документов Microsoft, где описывались типы данных SQL Server).

Список системных типов данных можно просмотреть и в компоненте Management Studio в окне **Object Explorer**. Для этого нужно раскрыть папку **Databases**, любую пользовательскую базу данных, например **BestDatabase**, далее **Programmability**, **Types** и, наконец, **System Data Types**.

Основная часть списка типов данных при раскрытых подгруппах **System Data Types** показана на рис. 4.1.

Классификация типов в Management Studio, как мы можем заметить, несколько отличается от только что рассмотренной.

Если к строке любого типа данных в этом окне подвести курсор мыши, то рядом с курсором появится подсказка (hint), где помимо названия типа данных указывается также диапазон значений, которые может принимать элемент этого типа.

Для данных возможны различные преобразования из одного типа в другой. Для них существуют разрешенные операции. Рассмотрим подробно системные типы данных, их возможные преобразования и допустимые операции.

Вначале же рассмотрим способы объявления локальных переменных, которые можно использовать в операциях в пакетах работы с базами данных.

4.2. Объявление локальных переменных

При выполнении примеров этой главы, да и многих примеров в последующих главах, вам понадобится объявлять локальные переменные. Для этого используется оператор `DECLARE`.

Оператор `DECLARE` позволяет объявить:

- ◆ обычную локальную переменную;
- ◆ курсор;
- ◆ табличную локальную переменную.

Синтаксис оператора `DECLARE` только для объявления локальных переменных показан в листинге 4.1 и в соответствующем R-графе (граф 4.1).

Листинг 4.1. Синтаксис оператора `DECLARE` для объявления локальной переменной

```
DECLARE <имя переменной> [AS] <тип данных> [ = <значение> ]
[ , <имя переменной> [AS] <тип данных> [ = <значение> ] ] ... ;
```



Граф 4.1. Синтаксис оператора `DECLARE` объявления локальной переменной

В операторе можно объявить одну или более локальных переменных. Объявления отделяются запятыми. Весь оператор, как обычно, завершается символом ; (точка с запятой).

Имя локальной переменной должно начинаться с символа @. Не следует начинать имена локальных переменных с двух символов @@, потому что с этих символов начинаются некоторые системные функции. В объявлении должен быть указан тип данных после необязательного ключевого слова `AS`. Это может быть системный тип данных или тип данных, определенный пользователем.

В операторе объявления можно задать начальное значение переменной, которое присваивается переменной после ее объявления, указав после знака равенства литерал (константу).

Присваивание другого, нового значения локальной переменной в тексте пакета можно сделать при выполнении оператора `SET`. Его несколько упрощенный синтаксис (только для локальных переменных):

```
SET { <имя переменной> = <выражение>
      | <имя переменной>
        {+= | -= | *= | /= | %= | &= | ^= | |= } <выражение>
    };
```

В этом операторе переменной можно присвоить конкретное значение, которое возвращает указанное в операторе выражение. Разумеется, тип данных значения, возвращаемого выражением, должен соответствовать типу данных локальной переменной. Кроме того, как и в языке C++, здесь допустимо использование арифметических, а также логических побитовых операций в процессе присваивания значения, когда для получения конечного результата используется текущее значение переменной.

Как вы помните, операция `+=` означает, что текущее значение локальной переменной суммируется со значением заданного выражения и результат присваивается переменной. Например, оператор:

```
SET @local_number += 1;
```

увеличивает значение числовой локальной переменной `@local_number` на единицу.

ЗАМЕЧАНИЕ

Локальной переменной можно присвоить новое значение и при использовании оператора `SELECT ... INTO`. Об операторе `SELECT` см. в главе 8.

Аналогично выполняются и другие арифметические операции.

Побитовые логические операции присваивания `&=`, `^=` и `|=` означают, соответственно, конъюнкцию (`AND`), исключающую дизъюнкцию (`XOR`) и обычную дизъюнкцию (`OR`). Они выполняются для каждого бита *внутреннего* представления переменной.

Объявление курсоров и табличных переменных мы рассмотрим далее в этой главе, по мере необходимости.

4.3. Числовые типы данных

Давайте восстановим справедливость и предложим следующую классификацию типов данных, традиционно входящих в SQL Server в эту группу Numeric. Используем следующие подгруппы.

◆ *Логический тип данных* `BIT`. Лучше всего было бы назвать его условно логическим типом, т. к. хотя множество его значений и множество операций и соответствует этому типу, однако не всегда его поведение в других ситуациях соответствует принятым в программировании правилам. Например, этому типу данных

нельзя присвоить логическое значение, являющееся результатом вычисления логического выражения. Или невозможно в функции `CAST()` или `CONVERT()` выполнить преобразование логического выражения к типу данных `BIT`. Этот тип данных также нельзя использовать как логическую переменную в условии поиска в предложении `WHERE` оператора `SELECT`. Правда, это претензии не к самому типу данных, а скорее всего к реализации системы.

- ◆ *Целочисленные типы данных:* `TINYINT`, `SMALLINT`, `INT`, `BIGINT`.
- ◆ *Дробные числа:* `NUMERIC`, `DECIMAL`, `SMALLMONEY`, `MONEY` (последние два иногда выделяют в отдельную группу — денежную, *monetary*).
- ◆ *Числа с плавающей точкой:* `FLOAT`, `REAL`.

Теперь справедливость восторжествовала. В три последние подгруппы входят типы данных, которые мы назовем истинно числовыми данными.

Для числовых типов данных (кроме, разумеется, типа данных `BIT`) допустимы все арифметические операции: *сложение* (+), *вычитание* (-), *умножение* (*) и *деление* (/). Операция получения остатка от деления (%) допустима для целых и дробных чисел, т. е. для точных чисел, но не для чисел с плавающей точкой. Все эти типы данных также можно преобразовывать в строки при использовании функции `CAST()` или `CONVERT()`. В числовые типы данных с некоторыми ограничениями, которые мы рассмотрим чуть позже, можно преобразовывать "правильные" строки, т. е. строки, содержащие в нужной последовательности цифры, знак числа, признак порядка (E или e) и порядок числа, десятичную точку. В битовый тип данных можно преобразовать строки 'TRUE' и 'FALSE', указанные в любом регистре.

Для числовых типов данных также существуют и две унарные операции: + и -. Унарной операции плюс (+) фактически не существует, поскольку она не выполняет никаких действий с соответствующим числом или числовым выражением. А унарная операция минус (-) изменяет знак числа (числового выражения) на противоположный: т. е. отрицательное число переводит в положительное, а положительное — в отрицательное.

Для всех числовых типов данных, включая и тип данных `BIT`, поддерживаются следующие операции сравнения:

- ◆ = (равно);
- ◆ !=, <> (не равно);
- ◆ < (меньше);
- ◆ > (больше);
- ◆ <=, !> (меньше или равно, не больше);
- ◆ >=, !< (больше или равно, не меньше).

Для типа данных `BIT` значение "истина" больше чем значение "ложь".

Результатом сравнения является значение "истина" (`TRUE`) или "ложь" (`FALSE`). Если один или оба сравниваемых операнда имеют значение `NULL`, то результат сравнения

не дает ни значения TRUE, ни значения FALSE. Результатом будет тот же NULL. Это — одно из основных правил реляционных баз данных.

К выражениям, возвращающим логические значения, можно применять *операции дизъюнкции* (логическое или, OR), *конъюнкции* (логическое и, AND) и *отрицание* (НЕ, NOT). Эти операции можно применять только в условиях соответствующих операторов. В языке Transact-SQL помимо операторов IF и CASE существуют две логические функции: IIF() и CHOOSE(). Они появились в версии SQL Server 2012.

IIF (<логическое выражение>, <значение для истины>, <значение для лжи>)

Функция возвращает "значение для истины", если логическое выражение истинно, и "значение для лжи", если значение ложное. Здесь "логическое выражение" не может иметь значение NULL. Нельзя также указать литералы 0, 1, 'TRUE' или 'FALSE'.

CHOOSE(<индекс>, <значение> [, <значение>] ...)

Функция имеет переменное количество параметров. По-хорошему, параметр "индекс" должен иметь целочисленное значение. Однако если он представлен дробным числом, то дробная часть просто отбрасывается без округления. Функция возвращает то значение из списка, номер которого равен числу в индексе. Первое значение имеет номер 1. Если значение индекса превышает количество элементов в списке значений или является нулем или отрицательным числом, то функция вернет NULL.

4.3.1. Тип данных BIT

Множеством допустимых значений для битового типа данных BIT являются 0, 1 и NULL. Множество допустимых операций — это три логические побитовые операции: отрицание (~), дизъюнкция (|) и конъюнкция (&). Никакие арифметические операции для этого типа данных невозможны.

Допустимо преобразование при использовании функции CAST() или CONVERT() строковых констант 'True' и 'False' в тип данных BIT. Эти две константы не чувствительны к регистру, их можно записать в любом регистре. Такие преобразования дадут, соответственно, 1 и 0. Значение 1 соответствует в этом типе данных значению "истина", а 0 — значению "ложь".

Для проверки поведения типа данных BIT при выполнении различных операций в командной строке, в PowerShell или в SQL Server Management Studio выполните следующие далее операции (пример 4.1).

Пример 4.1. Операции с битовым типом данных

```
USE master;
GO
SELECT CAST('True' AS BIT) AS 'True', CAST('False' AS BIT) AS 'False',
       CAST(1 AS BIT) & CAST(0 AS BIT) AS 'Conjunction',
       CAST(1 AS BIT) & ~CAST('False' AS BIT) AS '~Conjunction',
       CAST(1 AS BIT) | CAST(0 AS BIT) AS 'Disjunction';
GO
```

Вы получите следующий результат:

```
True  False Conjunction ~Conjunction Disjunction
-----
1      0      0           1           1
(1 row(s) affected)
```

Надеюсь, вы помните, что простая функция `CAST()` выполняет преобразование константы или выражения к типу данных, указанному после ключевого слова `AS`.

В первых двух полях оператора `SELECT` выполняется преобразование строковых констант 'True' и 'False' к типу данных `BIT`. Далее содержится конъюнкция, логическое и, значений 1 (истина) и 0 (ложь). Результат, как и следовало ожидать, получился 0 (ложь). В следующем используется отрицание ложного значения. Последний элемент иллюстрирует дизъюнкцию, логическое или.

Таблицы истинности

В табл. 4.1—4.3 приводятся таблицы истинности для трех логических операций — отрицания, дизъюнкции и конъюнкции при использовании в качестве operandов как логических значений 1 (истина) и 0 (ложь), так и значения `NULL`.

Таблица 4.1. Таблица истинности для отрицания

Операнд	~Операнд
0	1
1	0
NULL	NULL

Таблица 4.2. Таблица истинности для дизъюнкции двух operandов

Операнд 1	Операнд 2	Операнд 1 Операнд 2
0	0	0
0	1	1
1	0	1
1	1	1
1	NULL	NULL
0	NULL	NULL
NULL	1	NULL
NULL	0	NULL
NULL	NULL	NULL

Таблица 4.3. Таблица истинности для конъюнкции двух операндов

Операнд 1	Операнд 2	Операнд 1 & Операнд 2
0	0	0
0	1	0
1	0	0
1	1	1
1	NULL	NULL
0	NULL	NULL
NULL	1	NULL
NULL	0	NULL
NULL	NULL	NULL

Еще одно маленько критическое замечание исключительно для знатоков и любителей математической логики (или просто логики высказываний) и для тех, кто в своих разработках часто использует логические операции.

Посмотрите на приведенные таблицы истинности для дизъюнкции и конъюнкции. Возможно, вы привыкли к тому, что дизъюнкция, когда один из операндов имеет истинное значение (1 или TRUE), всегда дает истину, независимо от значения второго операнда. Это следует из определения дизъюнкции. В реальных системах обработки данных существует множество алгоритмов, где для ускорения логических вычислений проверяется только одно значение, и в подобной ситуации, когда в операции дизъюнкции один операнд является истинным, не нужно выполнять других расчетов. Часто это резко сокращает время получения результата, особенно когда операндами являются не отдельные значения, а довольно сложные логические выражения. Здесь же на подобное поведение системы не стоит рассчитывать. Если один из операндов NULL, то результатом является также NULL, даже если другой операнд истинный.

Похожая ситуация и при использовании конъюнкции. По-хорошему, при значении одного из операндов ложь, операция также должна вернуть ложное значение. Но если значение второго операнда NULL, значением результата будет NULL.

В своих разработках следует учитывать это поведение системы. При этом результат использования логических операций конъюнкции и дизъюнкции применительно к обычным логическим выражениям полностью соответствует общепринятым правилам (см. главу 8).

* * *

В базе данных столбцы таблицы с типом данных `BIT` хранятся по возможности компактно. Понятно, что для хранения одного столбца этого типа нужен один бит. Если в одной таблице присутствует 8 или менее столбцов такого типа данных, то они размещаются в одном байте. Большее количество столбцов в таблице типа данных `BIT` потребует уже нескольких байтов.

4.3.2. Целочисленные типы данных *TINYINT, SMALLINT, INT, BIGINT*

Это типы данных, позволяющие хранить целые числа. Они отличаются количеством байтов, отводимых для их хранения в базе данных, и, соответственно, диапазоном значений чисел, которые могут в них храниться.

Для целочисленных типов данных допустимы четыре классические арифметические операции: сложение (+), вычитание (-), умножение (*) и деление (/) и операция получения остатка от деления (%). Остатком от деления всегда будет целое число. Кроме того, к ним применимы агрегатные функции, о которых будет сказано несколько позже в этой главе. Применимы также унарные операции + и -.

В табл. 4.4 содержится описание целочисленных типов данных с указанием размера используемой памяти и диапазона допустимых значений.

Таблица 4.4. Целочисленные типы данных

Тип данных	Размер (байт)	Диапазон значений
TINYINT	1	от 0 до 255
SMALLINT	2	от -2^{15} до $2^{15} - 1$ или от -32 768 до 32 767
INT (INTEGER)	4	от -2^{31} до $2^{31} - 1$ или от -2 147 483 648 до 2 147 483 647
BIGINT	8	от -2^{63} до $2^{63} - 1$ или от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

Для типа данных INT допустимо использование синонима INTEGER. Синонимы для некоторых типов данных вводятся для соответствия стандарту SQL.

Если при выполнении деления целых чисел в результате получается дробное число, то дробные знаки просто отбрасываются, округление не проводится. Такое же поведение будет и при преобразовании дробного числа в целое в случае использования функции CAST(). При желании можете выполнить соответствующие проверки, например:

```
SELECT 53 / 8;
```

Вообще-то обычное деление этих чисел дает 6.625. При округлении результата можно было бы получить число 7. Здесь же результатом будет целое число 6.

При преобразовании дробного числа в целое также не выполняется округление. Например, при выполнении следующего преобразования

```
SELECT CAST(25.9 AS TINYINT);
```

мы получим число 25. А вот такое преобразование, как

```
SELECT CAST(259 AS TINYINT);
```

даст ошибку — арифметическое переполнение (arithmetic overflow), поскольку тип данных TINYINT дает возможность представлять только положительные числа (точ-

нее неотрицательные, поскольку сюда входит и число 0) до величины 255. Тот же результат с возвратом ошибки будет и при попытке преобразовать к этому типу данных отрицательное число.

Преобразуемое число (здесь мы говорим про числовую константу, но не про строковую константу, записанную в виде числа) может быть записано в виде целого числа, дробного числа или в виде числа с плавающей точкой, когда помимо мантиссы задается и порядок числа. Во всех случаях преобразование выполняется одинаковым образом: дробная часть отбрасывается без округления числа. Например, допустимо такое преобразование:

```
SELECT CAST(25.98E+1 AS INT);
```

Результатом, как можно было и ожидать, будет число 259.

Если дробные числа можно преобразовать к целому числу при соответствии исходного числа размеру целого, отводимого для хранения значения, то вот преобразование строки, хотя бы и содержащей правильное представление дробного числа, к целому числу даст ошибку. Мы видели, что можно преобразовать число 25.9 к типу данных TINYINT. При этом строку вида '25.9' преобразовать к любому целочисленному виду невозможно. Проверьте это, попытавшись выполнить оператор

```
SELECT CAST('25.9' AS TINYINT);
```

Вы получите сообщение, что невозможно преобразовать указанную строку переменной длины в тип данных TINYINT. Иными словами, строка, преобразуемая в целое число, должна быть и записана как целое число без десятичной точки, а также без указания порядка числа (E или e). В преобразуемой строковой константе помимо десятичных цифр можно лишь указать знак числа — плюс или минус.

Выскажу свое скромное мнение (IMHO — In My Humble Opinion), что такое неравноправное поведение системы по отношению к исходным числам и строкам в преобразовании не есть хорошее решение.

Чтобы выполнить преобразование такой "неправильной" строки к целому типу, нужно вначале преобразовать строку к дробному числовому типу данных, например, к DECIMAL, а затем уже преобразовывать результат к целочисленному типу данных. Наше предыдущее преобразование строки '25.9' к целому типу правильно выполняется при вложенном преобразовании типов:

```
SELECT CAST(CAST('25.9' AS DEC(3, 1)) AS TINYINT);
```

Результатом будет число 25.

Похожим образом можно выполнить преобразование строки, содержащей правильно записанное число с плавающей точкой. Здесь промежуточным типом данных может быть FLOAT или DOUBLE PRECISION:

```
SELECT CAST(CAST('25.98E+1' AS FLOAT) AS INT);
```

Результат — 259.

ЗАМЕЧАНИЕ

Некоторые только что описанные действия, например преобразование строковой константы '25.98E+1' к типу данных FLOAT, выглядят довольно странно и неразумно. Правда было бы сразу эту константу записать как число с плавающей точкой, просто убрав апострофы. На самом деле это наглядная иллюстрация того, что можно сделать в случае, когда мы имеем дело не с константами, а со строковыми переменными, которые содержат соответствующие данные.

Преобразование значения NULL к любому целочисленному типу данных, как и к любому другому типу данных, числовому или нечисловому, также даст значение NULL.

Для внутреннего хранения констант система выбирает характеристики, минимально возможные для каждой конкретной константы. Например, если числовая константа является неотрицательной и имеет значение до 255, то ей присваивается тип данных TINYINT.

4.3.3. Дробные числа

NUMERIC, DECIMAL, SMALLMONEY, MONEY

Это точные числа, которые имеют фиксированную дробную часть. Напомню, что точными (в отличие от приближенных) называются целые числа и дробные числа с фиксированной точкой.

Для этих типов данных также допустимы четыре классические арифметические операции: сложение (+), вычитание (-), умножение (*) и деление (/) и операция получения остатка от деления (%). Здесь, в отличие от целых чисел, остатком от деления может быть не только целое, но и дробное число, которое имеет количество дробных знаков, равное максимальному количеству дробных знаков у двух операндов. К этим типам данных применимы агрегатные функции и унарные операции.

Характеристики дробных чисел представлены в табл. 4.5.

Таблица 4.5. Дробные числовые типы данных

Тип данных	Размер (байт)	Диапазон значений
NUMERIC(<i>n, m</i>), DECIMAL(<i>n, m</i>)	5, 9, 13, 17	Зависит от значения точности (<i>n</i>). Максимальный диапазон от $-10^{38} + 1$ до $10^{38} - 1$
SMALLMONEY	2	От -214 748.3648 до 214 748.3647
MONEY	4	От -922 337 203 685 477.5808 до 922 337 203 685 477.5807

Типы данных NUMERIC и DECIMAL

Типы данных NUMERIC и DECIMAL являются одинаковыми, т. е. фактически это один и тот же тип данных. Для типа данных DECIMAL существует синоним DEC.

Данные этого типа позволяют хранить числа с фиксированной десятичной точкой — количество дробных знаков у них задается при определении элемента данных и не подлежит изменению.

Вот синтаксис задания дробного типа данных с фиксированной точкой:

```
{ NUMERIC | DECIMAL }[(n[, m])]
```

При задании этого типа данных можно указать *точность* (параметр *n*) — общее количество десятичных цифр, включающее количество знаков целой и дробной части, — и *масштаб* (параметр *m*) — количество дробных цифр после десятичной точки. Можно вообще не указывать ни точности, ни масштаба, тогда им присваивается значение по умолчанию. Можно указать точность, не указывая масштаба, масштабу в этом случае также присваивается значение по умолчанию.

Точность может быть указана в диапазоне от 1 до 38. Если точность явно не задана, то по умолчанию устанавливается значение 18.

Масштаб может принимать значение в диапазоне от 0 до установленного (явно или неявно) значения точности. По умолчанию ему присваивается значение 0.

Если не указаны ни точность, ни масштаб, то по умолчанию для определяемого объекта будут установлены значения (18, 0).

То есть, выражение

```
DECIMAL
```

соответствует выражению

```
DECIMAL(18, 0)
```

Если вы зададите

```
DECIMAL(20)
```

то это означает задание

```
DECIMAL(20, 0)
```

Количество знаков целой части в числе может быть и нулевым. Например, можно задать такое число:

```
DECIMAL(10, 10)
```

Количество байтов, отводимых для хранения данных этого типа, зависит только от значения точности, как показано в табл. 4.6.

Таблица 4.6. Количество байтов для хранения типов данных NUMERIC и DECIMAL

Точность	Байтов
1—10	5
11—21	9
21—29	13
30—38	17

Типы данных **SMALLMONEY** и **MONEY**

В принципе это тот же тип данных NUMERIC (или DECIMAL) с предустановленными значениями точности и масштаба.

Задание типа данных SMALLMONEY соответствует NUMERIC(10, 4), он занимает 4 байта памяти, а MONEY — NUMERIC(19, 4), занимает 8 байтов.

Эти типы данных используются для работы с денежными (валютными) величинами.

Одним из отличий этих типов данных от типов данных NUMERIC и DECIMAL является то, что при присваивании значений соответствующим элементам данных и в функции CAST(), которая преобразует строку к денежному типу данных, значение в исходной строке может начинаться с символа денежной единицы. В частности, допустимы символы ₽ (просто символ любой денежной единицы), \$ (доллар), € (евро), £ (фунт стерлингов), ¢ (цент "настоящий"), ¥ (иена) и ряд других.

* * *

При выполнении арифметических операций над дробными точными числами результату присваиваются соответствующие характеристики, которые в большинстве случаев позволяют получить ожидаемый результат. Такими характеристиками являются точность и масштаб.

Любопытно, что для операций сложения и вычитания точных дробных чисел расчет точности и масштаба является довольно сложным. Впрочем, характеристики остатка от деления тоже вычисляются не слишком просто.

Сложение и вычитание

Для операций сложения (+) и вычитания (-) двух точных чисел точность результата (общее количество символов числа) вычисляется суммированием максимального значения масштаба чисел с максимальным значением количества знаков целой части, плюс единица. В качестве масштаба результата выбирается максимальное значение масштаба исходных чисел.

Для более понятного описания характеристик результата все же лучше использовать математические формулы, как это сделано в Books Online. Введем следующие обозначения: точность первого операнда обозначим p_1 (от англ. *precision*), второго — p_2 , масштабы, соответственно, будем указывать как s_1 и s_2 (от англ. *scale*).

При принятых обозначениях точность результата (p_3) операций сложения и вычитания рассчитывается следующим образом:

$$p_3 = \max(s_1, s_2) + \max(p_1 - s_1, p_2 - s_2) + 1$$

Масштаб результата (s_3):

$$s_3 = \max(s_1, s_2)$$

Умножение

Для операции умножения (*) точность результата определяется суммой точностей операндов, плюс единица. Масштаб результата — сумма масштабов умножаемых чисел, или в принятых обозначениях:

$$p_3 = p_1 + p_2 + 1$$

$$s_3 = s_1 + s_2$$

Деление

Точность и масштаб результата деления двух чисел (/) определяются следующими формулами:

$$\begin{aligned} p3 &= p1 - s1 + s2 + \max(6, s1 + p2 + 1) \\ s3 &= \max(6, s1 + p2 + 1) \end{aligned}$$

Как видно из приведенных формул, масштаб частного от деления чисел будет иметь значение не менее шести. Однако здесь проявляется еще одна дискриминация, на этот раз к целым числам. Например, при делении двух целых 1 / 3 мы получим 0. Никаких обещанных шести знаков масштаба нам не дадут. (Если уж быть совсем честными, то следует сказать, что шести знаков масштаба нам с вами никто и не обещал в этом случае. Все операции с целыми всегда дают целое число.)

Остаток от деления

Наконец, характеристики результата операции получения остатка от деления (%) определяются следующими формулами:

$$\begin{aligned} p3 &= \min(p1 - s1, p2 - s2) + \max(s1, s2) \\ s3 &= \max(s1, s2) \end{aligned}$$

Характеристики результатов арифметических операций задаются таким образом, что мы всегда можем определить, во что выльются наши арифметические действия с операндами.

Все арифметические операции при необходимости выполняют округление результата — если в полученном числе больше дробных знаков, чем допустимо для столбца таблицы или переменной, куда должен помещаться результат.

ЗАМЕЧАНИЕ

Если очень постараться, то при выполнении арифметических операций над точными дробными числами все-таки можно получить и ошибку типа переполнения. В частности, такую ошибку можно получить при вычислении остатка от деления чисел с очень большим количеством знаков. При этом максимальная точность для таких чисел и только что рассмотренные правила определения характеристик результата арифметических операций сводят вероятность подобной ошибки практически к нулю.

Однако если попытаться выполнить операцию

12345678901234567890123456789012345678 % 3.3

то будет получена долгожданная ошибка переполнения:

Msg 8115, Level 16, State 2, Line 1

Arithmetic overflow error converting expression to data type numeric.

Преобразования в точные числа

Преобразования в дробные числа с использованием функций CAST() и CONVERT() выполняются по правилам, отличающимся от преобразований целых чисел.

Во-первых, при преобразовании выполняется округление числа. Во-вторых, никаких противовестественных ограничений на представление преобразуемой строки не

накладывается: в строке можно указать целое число, точное дробное число или число, заданное по правилам представления чисел с плавающей точкой.

4.3.4. Числа с плавающей точкой *FLOAT*, *REAL*

В Transact-SQL предусмотрено два типа приближенных чисел: *FLOAT* и *REAL*. Характеристики чисел с плавающей точкой представлены в табл. 4.7.

Таблица 4.7. Числовые типы данных с плавающей точкой

Тип данных	Размер (байт)	Диапазон значений
<i>FLOAT</i> (<i>n</i>)	4 или 8. Зависит от значения <i>n</i>	От -1.79×10^{-308} до -2.23×10^{-308} , 0 и от 2.23×10^{-308} до 1.79×10^{308}
<i>REAL</i>	4	От -3.40×10^{-38} до -1.18×10^{-38} , 0 и от 1.18×10^{-38} до 3.4079×10^{38}

После ключевого слова *FLOAT* можно в скобках указать количество битов, отводимых для хранения мантиссы числа. Допустимо указание значения от 1 до 53. Если значение не указано, то принимается 53.

Если значение задано в диапазоне от 1 до 24, то для хранения числа отводится 4 байта. Количество значащих цифр будет 7.

При задании диапазона от 25 до 53 числу отводится 8 байтов. Количество значащих цифр — 15.

Для типа данных *FLOAT* (53) есть синоним *DOUBLE PRECISION*, что соответствует стандарту SQL.

Типы данных для чисел с плавающей точкой позволяют на первый взгляд представлять числа в довольно большом диапазоне. На самом деле не следует сильно обольщаться на этот счет. Помните, что число такого типа может иметь не более 15 значащих цифр.

Если к очень большому числу прибавить очень маленькое число (или вычесть из него такое число), то результат никак не будет отличаться от начального большого числа.

Поскольку числа с плавающей точкой не являются точными, рекомендуется не использовать в операторах обращения к базе данных операций "равно" или "неравно". Часто результат сравнения в этих случаях не будет соответствовать вашим ожиданиям.

4.3.5. Функции для работы с числовыми данными

Для числовых типов данных (истинных числовых данных) в Transact-SQL существуют различные математические функции.

Агрегатные функции

К числовым данным могут применяться агрегатные функции. Такие функции применяются к некоторому множеству числовых значений, обычно к одному столбцу из нескольких строк таблицы. К этим функциям относятся:

- ◆ `SUM()` — суммирует все значения списка.
- ◆ `MIN()` — отыскивает минимальное значение в списке.
- ◆ `MAX()` — отыскивает максимальное значение в списке.
- ◆ `AVG()` — находит среднее арифметическое значение чисел заданного списка.
- ◆ `STDEV()` и `STDEVP()` — возвращают статистическое стандартное и среднеквадратичное отклонение соответственно.
- ◆ `VAR()` и `VARP()` — возвращают статистическую дисперсию значений.

ЗАМЕЧАНИЕ

К агрегатным функциям также иногда относят и некоторые другие функции, например функцию `COUNT()`. Эта функция не обрабатывает числовых данных, а осуществляет подсчет количества повторений экземпляров некоторого объекта базы данных, обычно строк таблицы, отвечающих заданным условиям.

Тригонометрические функции

В языке Transact-SQL существует полный джентльменский набор тригонометрических функций, как прямых, так и обратных. Это следующие функции.

- ◆ `ACOS(<арифметическое выражение>)` — арккосинус числа типа `FLOAT`. Возвращает значение угла в радианах. Как вы помните, значение передаваемого параметра должно находиться в диапазоне от -1 до $+1$.
- ◆ `ASIN(<арифметическое выражение>)` — арксинус числа типа `FLOAT`. Возвращает значение угла в радианах. Значение передаваемого параметра также должно находиться в диапазоне от -1 до $+1$.
- ◆ `ATAN(<арифметическое выражение>)` — арктангенс числа типа `FLOAT`. Возвращает значение угла в радианах.
- ◆ `ATN2(<арифметическое выражение 1>, <арифметическое выражение 2>)` — возвращает значение угла в радианах между положительным направлением оси X и лучом, проведенным из начала координат в точку, задаваемую координатами, указанными в качестве параметров функции. В своей нелегкой программистской жизни такую функцию я никогда не использовал.
- ◆ `COS(<арифметическое выражение>)` — косинус угла типа `FLOAT`, заданного в радианах.
- ◆ `COT(<арифметическое выражение>)` — котангенс угла типа `FLOAT`, заданного в радианах.
- ◆ `SIN(<арифметическое выражение>)` — синус угла типа `FLOAT`, заданного в радианах.
- ◆ `TAN(<арифметическое выражение>)` — тангенс угла типа `FLOAT`, заданного в радианах.

- ◆ RADIANS(<арифметическое выражение>) — возвращает значение угла в радианах для параметра, указанного в градусах.
- ◆ DEGREES(<арифметическое выражение>) — возвращает значение угла в градусах для параметра, указанного в радианах.

Традиционно к этой группе функций относится и функция PI(), которую более естественно было бы назвать константой. Функция, разумеется, возвращает значение числа π . Точность (15, 14), т. е. возвращается десятичное число 3.14159265358979. При обращении к этой функции, как и ко всем остальным, всегда нужно указывать и круглые скобки.

Использование некоторых тригонометрических функций и функций преобразования градусов в радианы и радианов в градусы показано в примере 4.2.

Пример 4.2. Примеры обращения к тригонометрическим функциям

```
USE master;
GO
SELECT 'PI = ' + CAST(DEGREES(PI()) AS VARCHAR(10)) + '°'
+ ', PI/2 = ' + CAST(DEGREES(PI() / 2) AS VARCHAR(12)) + '°'
+ ', -45.0° = ' + CAST(RADIANS(-45.0) AS VARCHAR(22)) + ' RADIAN';
SELECT 'SIN(PI) = ' + CAST(SIN(PI()) AS VARCHAR(20))
+ ', SIN(PI/2) = ' + CAST(SIN(PI()/2) AS VARCHAR(20));
SELECT 'COS(PI) = ' + CAST(COS(PI()) AS VARCHAR(20))
+ ', COS(PI/2) = ' + CAST(COS(PI()/2) AS VARCHAR(20));
SELECT 'TAN(PI) = ' + CAST(TAN(PI()) AS VARCHAR(20))
+ ', TAN(PI/2) = ' + CAST(TAN(PI()/2) AS VARCHAR(20));
SELECT 'COT(PI) = ' + CAST(COT(PI()) AS VARCHAR(20))
+ ', COT(PI/2) = ' + CAST(COT(PI()/2) AS VARCHAR(20));
GO
```

Результат выполнения операторов:

PI = 180°, PI/2 = 90°, -45.0° = -0.785398163397448300 RADIAN

SIN(PI) = 1.22465e-016, SIN(PI/2) = 1

COS(PI) = -1, COS(PI/2) = 6.12323e-017

TAN(PI) = -1.22465e-016, TAN(PI/2) = 1.63312e+016

COT(PI) = -8.16562e+015, COT(PI/2) = 6.12323e-017

В первом операторе выполняется преобразование радианов в градусы и градусов в радианы. Следующие операторы отображают результаты вызова функций получения синуса, косинуса, тангенса и котангенса двух различных углов — π и $\pi/2$.

Логарифмические функции, возвведение в степень, извлечение корня

Полным набором таких функций являются следующие.

- ◆ EXP (<арифметическое выражение>) — возвращает экспоненту заданного числа.
- ◆ LOG (<арифметическое выражение>) — возвращает натуральный логарифм для заданного числового параметра.
- ◆ LOG10 (<арифметическое выражение>) — возвращает десятичный логарифм для заданного параметра.
- ◆ POWER (<арифметическое выражение>, <степень>) — возвращает значение числового параметра, возведенного в указанную степень. Степень может быть представлена не только целым, но и дробным числом — положительным или отрицательным.
- ◆ SQUARE (<арифметическое выражение>) — возвращает квадрат заданного числового параметра.
- ◆ SQRT (<арифметическое выражение>) — возвращает квадратный корень заданного числового параметра.

Соответствующие проверки использования функций приведены в примере 4.3.

Пример 4.3. Примеры обращения к логарифмическим функциям, функциям возвведения в степень, извлечения корня

```
USE master;
GO
SELECT 'e = ' + CAST(EXP(1) AS VARCHAR(20));
SELECT 'LOG(10) = ' + CAST(LOG(10) AS VARCHAR(20))
    + ', LOG10(10) = ' + CAST(LOG10(10) AS VARCHAR(20));
SELECT 'LOG(e) = ' + CAST(LOG(EXP(1)) AS VARCHAR(20))
    + ', LOG10(e) = ' + CAST(LOG10(EXP(1)) AS VARCHAR(20));
SELECT '6.5 ^ 3 = ' + CAST(POWER(6.5, 3) AS VARCHAR(20))
    + ', 6.5 ^ 2 = ' + CAST(SQUARE(6.5) AS VARCHAR(20));
SELECT 'SQRT(42.25) = ' + CAST(SQRT(42.25) AS VARCHAR(20))
    + ', V(3)(274.6) = ' + CAST(POWER(274.6, 1.0 / 3) AS VARCHAR(20));
GO
```

Результат:

e = 2.71828

LOG(10) = 2.30259, LOG10(10) = 1

LOG(e) = 1, LOG10(e) = 0.434294

6.5 ^ 3 = 274.6, 6.5 ^ 2 = 42.25

SQRT(42.25) = 6.5, V(3)(274.6) = 6.5

Первый оператор возвращает значение числа e , поскольку экспонента единицы, EXP(1), дает именно это число.

Следующий оператор возвращает натуральный и десятичный логарифм числа 10, после этого оператор вычисляет натуральный и десятичный логарифм числа e .

Затем число 6.5 возводится в третью степень и в квадрат. Последний оператор находит квадратный корень из числа 42.25 и кубический корень из числа 274.6. Извлечение корня третьей степени я здесь изобразил в виде $\sqrt[3]{(274.6)}$. Как вы помните, чтобы извлечь из числа кубический корень, нужно возвести это число в степень $1/3$, что и сделано в обращении к функции POWER() в последней строке кода. Обратите внимание, что степень $1/3$ задана в виде отношения $1.0 / 3$, чтобы при делении целых чисел в результате мы не получили 0.

Другие математические функции

Вот другие математические функции, которые не включены ни в одну из перечисленных групп.

- ◆ ABS(<арифметическое выражение>) — возвращает абсолютное значение заданного параметра, т. е. положительное, точнее неотрицательное, число.
- ◆ CEILING(<арифметическое выражение>) — возвращает наименьшее целое число, большее или равное значению заданного параметра. Другое название функции — округление в большую сторону.
- ◆ FLOOR(<арифметическое выражение>) — возвращает наибольшее целое число, меньшее или равное значению заданного параметра. Другое название — округление в меньшую сторону.
- ◆ ROUND(<арифметическое выражение>, <точность> [, <признак>]) — выполняет округление числа с указанной точностью. Если параметр "признак" равен нулю или отсутствует, то выполняется обычное округление. Если этот параметр имеет положительное значение, то результат округляется до соответствующего количества знаков после десятичной точки. Если параметр отрицательный, то указанное количество знаков целого справа обнуляется, точнее, происходит округление (но не усечение, как сказано в Books Online, в ее русскоязычной версии) числа до знака *признак* + 1. Чтобы понять, что я здесь написал об этой функции, давайте чуть позже рассмотрим пример 4.4.
- ◆ RAND([<начальное значение>]) — возвращает псевдослучайное число в диапазоне от 0 до 1. Если начальное значение не задано, то используется случайное начальное значение. Функция при одном и том же заданном начальном значении всегда будет возвращать одинаковый результат в одном соединении с сервером.
- ◆ SIGN(<арифметическое выражение>) — возвращает знак арифметического выражения: число +1, если выражение положительное, -1, если отрицательное, и 0, если выражение равно нулю.

В примере 4.4 даны операторы обращения к двум функциям этой группы. Пожалуй, более интересными из всех перечисленных являются функции ROUND() и RAND().

Пример 4.4. Обращение к математическим функциям RAND и ROUND

```
USE master;
GO
SET NOCOUNT ON;
SELECT RAND(10) AS 'RAND(10)',
       RAND(10) AS 'RAND(10)',
       RAND() AS 'RAND()', 
       RAND() AS 'RAND()';
SELECT ROUND(12345.6789, 0) AS 'ROUND(0)', 
       ROUND(12345.6789, 1) AS 'ROUND(1)', 
       ROUND(12345.6789, -1) AS 'ROUND(-1)', 
       ROUND(12345.6789, -3) AS 'ROUND(-3)';
GO
```

Результат выполнения:

RAND(10)	RAND(10)	RAND()	RAND()
0,713759689954247	0,713759689954247	0,182458908613686	0,586642279446948
ROUND(0)	ROUND(1)	ROUND(-1)	ROUND(-3)
12346.0000	12345.7000	12350.0000	12000.0000

В первом операторе происходят обращения к функции `RAND()`. Если функции передается одно и то же значение параметра, то функция возвращает одинаковые значения. Это видно по первым двум результатам. Если же функции не передается никаких параметров, то возвращаемые значения различны — см. два последних значения.

В следующем операторе при обращении к функции `ROUND()` выполняется округление одного и того же числа 12345.6789 с различной точностью. Здесь точность задается и отрицательными значениями. Посмотрите, какие получаются результаты.

В первом случае выполняется округление до целого числа по принятым правилам округления. Похожим образом выполняется правильное округление с любой заданной точностью. Если параметр точности задать положительным числом, равным или большим чем количество дробных знаков числа, то никакого изменения исходного числа выполнено не будет.

4.4. Символьные данные

Символьные типы данных позволяют хранить строки символов. Символьные данные (Character) представлены двумя подгруппами.

Обычные символьные строки (Character Strings). Включают типы данных `CHAR`, `VARCHAR` и `TEXT`. Тип данных `TEXT` в ближайшее время будет удален из системы, поэтому его рассматривать мы не будем.

Символьные строки в Юникоде (Unicode Character Strings). Типы данных NCHAR, NVARCHAR и NTEXT. Тип данных NTEXT также будет удален, рассматривать его мы не станем.

Для строковых данных допустимы все операции сравнения: =, !=, <, <=, !<, >, >=, !=>. Сравнения выполняются в так называемом лексикографическом порядке, при котором учитывается положение буквы в алфавите. При этом прописная буква и соответствующая ей строчная буква считаются равными. По крайней мере, такое равенство соблюдается при используемом нами порядке сортировки. Поведение операций сравнения для других порядков сортировки можно определить, посмотрев описание (поле `Description`) в строке отображения порядка сортировки при использовании функции `fn_helpcollations()`. Присутствие в поле описания текста "case-insensitive" означает отсутствие чувствительности к регистру, т. е. символы "A" и "a" считаются равными. Напротив, текст "case-sensitive" означает, что символы чувствительны к регистру. Например, при использовании порядка сортировки `Cyrillic_General_CI_KS_WS` строчные и прописные буквы будут рассматриваться как различные. Как можно отобразить список существующих в системе порядков сортировки, показано в главе 3, в примере 3.26.

4.4.1. Символьные строки `CHAR`, `VARCHAR`

Эти типы данных позволяют хранить обычные, не Юникод, строки. Тип данных `CHAR` имеет фиксированную длину, `VARCHAR` — переменную. Каждый символ занимает один байт.

Синтаксис задания типов данных:

```
CHAR [ (<целое>) ]  
VARCHAR [ (<целое> | max) ]
```

Для типа данных `VARCHAR` могут использоваться такие синонимы, как `CHAR VARYING` и `CHARACTER VARYING`.

Если при задании любого из указанных строковых типов данных параметр "целое" опущен, то значение устанавливается в 1. Кстати, если при преобразовании значения в строковый тип в функции `CAST()` для `CHAR` или `VARCHAR` не указать размер, то по умолчанию он устанавливается в 30. При этом следует помнить, что в случае типа данных `VARCHAR` конечные пробелы отбрасываются, если параметр `ANSI_PADDING` установлен в значение `OFF` — это значение по умолчанию.

Параметр "целое" задает количество байтов, которое отводится для хранения строки. Точнее, это максимальное количество символов, которое может содержать строка. Параметр может принимать значение от 1 до 8000. Для типа данных `VARCHAR` размер увеличивается на два байта.

Если при задании столбца указать `VARCHAR(max)`, то в нем можно хранить объем данных до 2 Гбайт. Данные `VARCHAR(max)` хранятся отдельно от данных таблицы. Для такого типа данных допустимы и обычные строковые операции.

Переменным строковых типов данных назначается порядок сортировки (collate), установленный по умолчанию для текущей базы данных, если им при объявлении явно не задается иной порядок в предложении `COLLATE`.

Если строковой переменной или строковому столбцу присваивать значение, по длине превышающее допустимое количество символов, то ничего страшного не произойдет. Не будет сгенерировано никакой ошибки; просто молча будет выполнено усечение значения строки до нужного размера. В некоторых системах управления базами данных в таком случае выдается ошибка переполнения.

Строковые константы, как мы помним, заключаются в апострофы. Если внутри строки должен присутствовать апостроф, то его нужно записать дважды, чтобы отличить от завершающего константу апострофа.

ПРИМЕЧАНИЕ

Еще раз хочу напомнить, что при определенных установках системы строковые константы могут заключаться и в кавычки. Но как мы выяснили, использование для этих целей апострофов избавит нас от ненужных неприятностей в случае изменения установок системы.

Для строковых типов данных применима операция конкатенации, объединяющая две строки в одну. Операция задается символом плюс (+).

Пример. Результатом конкатенации двух строк

```
'Smart ' + 'students'
```

будет строка

```
'Smart students'
```

Чуть позже мы рассмотрим и множество функций, используемых для работы со строковыми данными.

4.4.2. Символьные строки `NCHAR`, `NVARCHAR`

Типы данных `NCHAR` и `NVARCHAR` позволяют хранить строки в кодировке Юникод. Тип данных `NCHAR` имеет фиксированную длину, `NVARCHAR` — переменную. Каждый символ занимает два байта. Такие данные используют набор символов UCS-2. Набор символов UCS (Universal Character Set, универсальный набор символов) определен на основании стандарта ISO 10646. Он позволяет кодировать до 65 536 символов. Включает в себя письменность практически всех существующих "живых" языков, за исключением некоторых совсем уж экзотических. В частности, содержит буквы еврейского алфавита, используемые в языках иврит и идиш, иероглифы Хань, применяемые в китайском и японском языках, фонетическое представление типа катакана и хираганы (используются в Японии).

Синтаксис задания типов данных:

```
NCHAR [ (<целое>) ]
```

```
NVARCHAR [ (<целое> | max) ]
```

Для типа данных NCHAR используются синонимы NATIONAL CHAR и NATIONAL CHARACTER. Для типа данных NVARCHAR могут использоваться синонимы NATIONAL CHAR VARYING и NATIONAL CHARACTER VARYING.

Если при задании строкового типа данных параметр "целое" опущен, то значение устанавливается в 1. При отсутствии указания размера в функции CAST() так же, как и для обычных строковых данных, принимается значение 30.

Параметр "целое" задает количество символов, которое может содержать строка. Параметр может принимать значение от 1 до 4000. Для типа данных NVARCHAR размер увеличивается на два байта.

Если при задании столбца указать вариант NVARCHAR(max), то, как и в случае типа данных VARCHAR(max), в нем можно хранить большой объем данных.

В случае присваивания значения полю с данными Юникод используются строковые константы, заключенные в апострофы. Чтобы выполнить преобразование строки в формат Юникод перед константой, нужно поставить символ N, например:

```
SET @string = N'αβγδεζηθω ';
```

4.4.3. Типы данных

VARCHAR(MAX), NVARCHAR(MAX), VARBINARY(MAX)

Столбцы таблиц базы данных с типами данных VARCHAR(MAX), NVARCHAR(MAX) и VARBINARY(MAX) могут хранить довольно большой объем данных — до $2^{31} - 1$ или 2147483647 байтов, т. е. около 2 Гбайтов. Если это одна книга в формате PDF, то это книга с десятками тысяч страниц. Если это фильм довольно хорошего качества, то это фильм часа на полтора-два. При этом следует отметить, что для типа данных VARBINARY(MAX) в таблицах существует возможность использовать атрибут FILESTREAM, что позволяет снять и такое ограничение на объем хранимых данных. Файловые потоки мы рассмотрим в следующей главе 5.

К данным этих типов могут применяться и некоторые строковые функции.

4.4.4. Строковые функции

Для строковых типов данных существует ряд полезных функций, которые, как правило, присутствуют и в обычных языках программирования. Здесь приведено большинство из них, которые по смыслу я объединил в небольшие группы.

Для строковых функций приводится немало примеров. Если в вашей профессиональной деятельности вам приходится обрабатывать различные строковые данные, использовать разные строковые функции, то в процессе исследования существующих строковых функций SQL Server и дальнейшего их использования вы получите истинное удовольствие.

Функции определения размера DATALENGTH(), LEN()

Для определения длины (количество байтов) строки или строкового выражения можно использовать функцию DATALENGTH(), которой передается в качестве параметра

метра исходное выражение. Другая функция, `LEN()`, возвращает количество символов строки, за исключением конечных пробелов.

Еще раз подчеркну отличие этих функций. Функция `LEN()` возвращает количество символов, а функция `DATALENGTH()` — количество байтов. Иногда они отличаются друг от друга по значению.

Для иллюстрации использования строковых функций `LEN()` и `DATALENGTH()` по отношению к различным строковым типам данных в Management Studio выполните операторы примера 4.5.

Пример 4.5. Использование строковых функций

```
USE master;
GO
SET NOCOUNT ON;
-- 1
DECLARE @STR CHAR(20);
SET @STR = ' 123 ';
SELECT LEN(@STR) AS 'LEN', DATALENGTH(@STR) AS 'DATALENGTH';
-- 2
DECLARE @VARSTR VARCHAR(20);
SET @VARSTR = ' 123 ';
SELECT LEN(@VARSTR) AS 'LEN', DATALENGTH(@VARSTR) AS 'DATALENGTH';
-- 3
DECLARE @NVARSTR NVARCHAR(20);
SET @NVARSTR = ' 123 ';
SELECT LEN(@NVARSTR) AS 'LEN', DATALENGTH(@NVARSTR) AS 'DATALENGTH';
-- 4
DECLARE @NSTR NCHAR(20);
SET @NSTR = ' 123 ';
SELECT LEN(@NSTR) AS 'LEN', DATALENGTH(@NSTR) AS 'DATALENGTH';
GO
SET NOCOUNT OFF;
```

Оператор `SET NOCOUNT ON` отменяет значение по умолчанию — вывод сообщения о количестве обработанных строк. В конце скрипта оператором `SET NOCOUNT OFF` восстанавливается это значение по умолчанию.

Оператор `DECLARE` объявляет локальную переменную. Имя такой переменной по правилам Transact-SQL должно начинаться с символа @. В операторе также указывается тип данных переменной. Оператор `SET` присваивает переменной строковое значение.

Во всех четырех вариантах переменным различных типов строковых данных присваивается одно и то же значение, состоящее из пяти символов. Первый и последний символ — пробел. Здесь мы сможем увидеть поведение функций по отношению и к пробелам.

Чтобы результаты отображались в текстовом виде, нужно в меню выбрать **Query | Results To | Results to Text** или нажать комбинацию клавиш <Ctrl>+<T>.

Результат выполнения операторов:

		DATALENGTH
LEN	20	DATALENGTH
LEN	5	DATALENGTH
LEN	10	DATALENGTH
LEN	40	LEN

`LEN()` действительно независимо от конкретного типа данных строковой переменной возвращает количество символов, а не байтов, и не учитывает конечных пробелов, но учитывает начальные пробелы.

Функция же `DATALENGTH()` возвращает количество байтов в строке. Для строковых типов данных *переменной* длины учитывается только фактическое количество символов, включая и конечные пробелы. Для данных *фиксированной* длины всегда возвращается количество символов, заданных при определении переменной.

Функции выделения подстроки *LEFT()*, *RIGHT()*, *SUBSTRING()*

Три функции позволяют выделить из исходной строки подстроку: `LEFT()`, `RIGHT()`, `SUBSTRING()`. Исходная строка может быть как локальной переменной или столбцом таблицы, так и строковым выражением.

Функция `LEFT()` возвращает указанное количество первых (левых) символов строки. Синтаксис функции:

`LEFT(<исходная строка>, <количество символов>)`

Функция `RIGHT()` возвращает указанное количество правых символов строки. Синтаксис функции:

`RIGHT(<исходная строка>, <количество символов>)`

Функция `SUBSTRING()` позволяет выделить любую часть строки. Ее синтаксис:

`SUBSTRING(<исходная строка>, <начальная позиция>, <количество символов>)`

Здесь из исходной строки выделяется указанное количество символов, начиная с заданной позиции. Символы в исходной строке (строковом выражении) нумеруются, начиная с единицы. Выделение подстрок показано в примере 4.6.

Пример 4.6. Использование функций выделения подстроки

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @string VARCHAR(20);
SET @string = 'Stupid students';
SELECT LEFT(@string, 6) AS 'LEFT',
       RIGHT(@string, 8) AS 'RIGHT',
      SUBSTRING(@string, 11, 99) AS 'SUBSTRING';
GO
SET NOCOUNT OFF;
```

Результат выполнения:

LEFT	RIGHT	SUBSTRING

Stupid students dents		

Функции также прекрасно работают и со строками Юникод (пример 4.7).

Пример 4.7. Использование функций выделения подстроки для данных Юникод

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @NVARSTR NVARCHAR(23);
SET @NVARSTR = N'просите и дастъ сѧ вамъ';
SELECT @NVARSTR AS 'NVARSTR',
       LEFT(@NVARSTR, 7) AS 'LEFT',
      RIGHT(@NVARSTR, 7) AS 'RIGHT',
     SUBSTRING(@NVARSTR, 9, 7) AS 'SUBSTRING';
GO
SET NOCOUNT OFF;
```

Результат выполнения:

NVARSTR	LEFT	RIGHT	SUBSTRING

просите и дастъ сѧ вамъ просите сѧ вамъ и дастъ			

Функции удаления пробелов *LTRIM()*, *RTRIM()*

Функция *LTRIM()* удаляет в заданной строке начальные (левые, left) пробелы, *RTRIM()* — конечные (правые, right). А вот функции *TRIM()*, которая бы одновременно удаляла как начальные, так и конечные пробелы, в языке Transact-SQL нет. Чтобы удалить начальные и конечные пробелы, нужно к строке применить обе функции *LTRIM()* и *RTRIM()*. Применение этих функций можно, разумеется, выполнять в любом порядке. Использование этих функций см. в примере 4.8.

Пример 4.8. Использование функций удаления пробелов

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @string char(12);
SET @string = ' New Moon '
SELECT '    ' + @string + '    ' AS 'Original',
      '    ' + LTRIM(@string) + '    ' AS 'LTRIM',
      '    ' + RTRIM(@string) + '    ' AS 'RTRIM',
      '    ' + LTRIM(RTRIM(@string)) + '    ' AS 'Full TRIM 1',
      '    ' + RTRIM(LTRIM(@string)) + '    ' AS 'Full TRIM 2'
GO
```

Результат:

Original	LTRIM	RTRIM	Full TRIM 1	Full TRIM 2
' New Moon '	'New Moon'	' New Moon'	'New Moon'	'New Moon'

Как вы уже догадались, эти функции работают корректно и с данными Юникод.

Функции преобразования символов

ASCII(), STR(), CHAR(), NCHAR(), UNICODE()

Эта группа функций позволяет получить код конкретного символа строки (число) или, наоборот, получить символ из заданного числа.

Функция **ASCII(<строковое выражение>)** возвращает код (целое число) заданного первого символа в строке.

Функция **STR()** выполняет преобразование числа с плавающей точкой в строку символов. Синтаксис вызова функции:

STR(<числовое выражение> [, <длина> [, <количество десятичных знаков>]])

<числовое выражение> — преобразуемое число (числовое выражение). Это может быть целое, дробное число или число с плавающей точкой. <длина> — задает количество символов, отводимых под результат. Учитывает цифры, десятичную точку, знак числа. Если длина не указана, то предполагается 10. Последний параметр, <количество десятичных знаков>, задает количество символов, отводимых для размещения дробных знаков преобразованного числа. Не может превышать 16.

Функция **CHAR(<числовое выражение>)**. Параметром является целочисленное выражение. Преобразует число (целое) в символ.

Функция **NCHAR(<числовое выражение>)**. Преобразует число (целое) в символ Юникода.

Функция **UNICODE(<строковое выражение>)**. Возвращает код (целое число), соответствующего стандарту Юникода, первого символа в строке.

Использование функций ASCII(), CHAR(), NCHAR(), UNICODE() и STR() см. в примере 4.9.

Пример 4.9. Использование функций преобразования символов

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @nstring nchar(13);
DECLARE @string char(13);
DECLARE @position int;
SET @nstring = N'i азъ глаголюх';
SET @string = 'I am speaking';
SET @position = 1;
WHILE @position <= DATALENGTH(@string)
BEGIN
    SELECT ASCII(SUBSTRING(@string, @position, 1)),
           CHAR(ASCII(SUBSTRING(@string, @position, 1))),
           UNICODE(SUBSTRING(@nstring, @position, 1)),
           NCHAR(UNICODE(SUBSTRING(@nstring, @position, 1)));
    SET @position = @position + 1;
END;
-- Использование функции STR()
DECLARE @num FLOAT = 0.314E1;
DECLARE @DEC DECIMAL(3,2) = 3.14;
SELECT STR(@num, 5, 3), STR(@DEC, 4, 2), STR(PI(), 10, 8);
SET NOCOUNT OFF;
GO
```

Результат:

73	I	1111	i
32		32	
97	a	1072	а
109	m	1079	з
32		1098	ъ
115	s	32	
112	p	1075	г
101	e	1083	л
97	a	1072	а

```

----- -----
107      k    1075      г
----- -----
105      i    1086      о
----- -----
110      n    1083      л
----- -----
103      г    1133      як
----- -----
3.140 3.14 3.14159265

```

Вот этот пример давайте рассмотрим достаточно подробно, поскольку в нем содержатся некоторые новые для кого-то из нас средства, операторы.

Вначале привычным для нас с вами способом объявляются три переменные, затем им присваиваются значения. Переменным можно присваивать начальные значения и в операторе объявления переменной. Например, можно объявить переменную @nstring и присвоить ей значение следующим образом:

```
DECLARE @nstring nchar(13) = N'ї азъ глаголяк';
```

В скриптах SQL Server можно использовать императивные средства. Оператор ветвления `IF` мы уже неоднократно использовали. Здесь же мы применяем оператор цикла `WHILE`.

```

WHILE @position <= DATALENGTH(@string)
BEGIN
    SELECT ASCII(SUBSTRING(@string, @position, 1)),
           CHAR(ASCII(SUBSTRING(@string, @position, 1))),
           UNICODE(SUBSTRING(@nstring, @position, 1)),
           NCHAR(UNICODE(SUBSTRING(@nstring, @position, 1)));
    SET @position = @position + 1;
END;

```

Он работает таким же образом, как и аналогичный оператор в большинстве языков программирования. Вначале задается условие продолжения цикла. Если оно истинно, то выполняется тело цикла, иначе происходит выход из цикла.

Поскольку сам цикл (тело цикла) содержит два оператора, то их надо заключить в операторные скобки `BEGIN` и `END`. В цикле осуществляется последовательный просмотр символов исходных строк и выводятся данные об очередном символе каждой из двух исследуемых строк, обычной и в кодировке Юникод. Затем значение параметра цикла увеличивается на единицу. А перед выполнением цикла значение этого параметра, как водится, устанавливается в 1.

Так как обе строки у нас имеют одинаковое количество символов, то в скрипте задано следующее условие продолжения цикла, при котором значение параметра цикла сравнивается с размером в байтах обычной строки `@string`:

```
WHILE @position <= DATALENGTH(@string)
```

Условие можно было бы применить также и к строке Юникод. В этом случае вместо функции `DATALENGTH()` мы должны будем использовать функцию `LEN()`:

```
WHILE @position <= LEN(@nstring)
```

Результаты использования функций `ASCII()`, `CHAR()`, `NCHAR()` и `UNICODE()` можно видеть в списке выведенных строк.

ЗАМЕЧАНИЕ

Знатокам старославянского и современного английского языков — пожалуйста, не рассматривайте две строковые переменные этого примера как попытку перевода с одного языка на другой. Здесь я подбирал фразы по размеру с целью простой демонстрации использования функций. Ну и смысл их оказался близким.

В последних строках пакета демонстрируется использование функции `STR()`:

```
-- Использование функции STR()
DECLARE @num FLOAT = 0.314E1;
DECLARE @DEC DECIMAL(3,2) = 3.14;
SELECT STR(@num, 5, 3), STR(@DEC, 4, 2), STR(PI(), 10, 8);
```

Здесь функция применяется к локальным переменным с плавающей точкой и десятичного типа данных, а также к системной функции `PI()`. Получены следующие результаты:

```
-----  
3.140 3.14 3.14159265
```

Функции изменения регистра символов `UPPER()`, `LOWER()`

Функция `UPPER(<строковое выражение>)` переводит все буквы строки в верхний регистр, т. е. преобразует все буквы в прописные или, как еще говорят, в заглавные, "большие". Функция применима как к буквам латинского алфавита, так и к буквам кириллицы. Она одинаково правильно работает для обычных строк и для строк в кодировке Юникод.

Аналогично, функция `LOWER(<строковое выражение>)` переводит буквы в нижний регистр, преобразует их в строчные, "маленькие". Применяется к обычным строкам и к кодировке Юникод, для латинских букв и для кириллицы.

На символы, отличные от букв, эти функции не оказывают никакого влияния.

Если вам интересно и у вас есть достаточно времени, то для исследований можете в предыдущий пример добавить использование этих функций. Мне было интересно, и я выполнил для себя такие проверки.

Функция преобразования строки к идентификатору `QUOTENAME()`

Функция `QUOTENAME(<строковое выражение> [, <ограничитель>])` позволяет преобразовать строку (имя, идентификатор) в кодировке Юникод в правильную строку с разделителями. Иными словами, функция позволяет получить из исходной строки правильный идентификатор с разделителями. Смысл, а точнее техника преобразования, заключается в добавлении ограничителей (квадратных скобок, кавычек или

апострофов) и при необходимости в дублировании внутри строки символы, являющиеся признаком завершения такой строки-идентификатора. Если параметр "ограничитель" отсутствует, то в качестве ограничителя для имени выбираются по умолчанию квадратные скобки.

Например, выражение

```
QUOTENAME('asd[]fghj')
```

вернет значение

```
[asd[]]fghj
```

Здесь по умолчанию в качестве разделителя используются квадратные скобки. В созданном идентификаторе правая квадратная скобка, являющаяся частью идентификатора, продублирована.

В следующей строке содержится кавычка. Этот же символ выбран в качестве ограничителя.

```
QUOTENAME('asd"fghj', '')
```

Результатом будет строка:

```
"asd""fghj"
```

Функции поиска данных и изменения строки

REPLICATE(), REVERSE(), REPLACE(), CHARINDEX()

Функция **REPLICATE**(<строковое выражение>, <целое>) повторяет исходную строку указанное количество раз. Если исходное строковое выражение не является строкой с типом данных **VARCHAR(MAX)** или **NVARCHAR(MAX)**, то при необходимости результат усекается до размера 8000 байтов.

Функция **REVERSE**(<строковое выражение>) переставляет символы исходной строки в обратном порядке.

Функция **REPLACE**(<строковое выражение>, <заменяемое строковое выражение>, <заменяющее строковое выражение>) выполняет замену в исходном строковом выражении все вхождения конкретной подстроки (<заменяемое строковое выражение>) на новое значение (<заменяющее строковое выражение>).

Функция **CHARINDEX**(<отыскиваемое строковое выражение>, <исходное строковое выражение> [, <начальное значение>]) выполняет поиск подстроки в исходном строковом выражении. Поиск начинается с позиции, заданной параметром <начальное значение>. Если этот параметр отсутствует, то поиск начинается с первого символа.

Функция возвращает номер позиции в строке, с которой начинается первое вхождение искомой подстроки. Если заданный текст не найден, функция возвращает значение 0. Функция одинаково работает с обычными строками и со строками в кодировке Юникод. Поведение функции в отношении строчных и прописных букв связано с используемым порядком сортировки — если порядок сортировки не является чувствительным к регистру, функция рассматривает строчную и прописную букву как равные, а иначе, как различные.

Использование этих функций показано в примере 4.10.

Пример 4.10. Использование функций поиска и изменения данных

```
USE master;
GO
SET NOCOUNT ON;
SELECT REPLICATE('Учиться, ', 2) AS 'REPLICATE',
       REVERSE('А роза упала на лапу Азора') AS 'REVERSE',
       REPLACE(REPLICATE('Учиться, ', 2), 'Учиться','Отдыхать')
              AS 'REPLACE',
       CHARINDEX('Что-то', 'Там этого нет') AS 'CHARIND';

DECLARE @string1 char(1) = 'o';
DECLARE @string2 varchar(25) = ' Одинокий прохожий спешит';
SELECT CHARINDEX(@string1, @string2) AS 'Default collation',
       CHARINDEX('o' COLLATE Cyrillic_General_CS_AI_KS_WS,
                  ' Одинокий прохожий спешит'
                  COLLATE Cyrillic_General_CS_AI_KS_WS)
              AS 'Collation Cyrillic_General_CS_AI_KS_WS';
GO
```

Результат:

REPLICATE	REVERSE	REPLACE	CHARIND
Учиться, Учиться, арозА упал ан алапу азор А Отдыхать, Отдыхать,			0
Default collation	Collation Cyrillic_General_CS_AI_KS_WS		
2	6		

Здесь выполняется повторение текстовой строки (`REPLICATE`), инверсия (`REVERSE`), замена символов в повторяющейся строке (использование функции `REPLICATE()` в качестве входного параметра функции `REPLACE()`) и поиск несуществующей подстроки (`CHARINDEX`). В четвертом элементе выбора оператора `SELECT` функция `CHARINDEX()` возвращает значение 0, т. к. искомый текст отсутствует.

Во второй части примера проверяется наличие чувствительности к регистру у функции `CHARINDEX()` при использовании соответствующего порядка сортировки. Объявляются две локальные переменные, им присваиваются строковые значения. В последующем операторе `SELECT` дважды выполняется функция `CHARINDEX()` с одними и теми же данными, но с различными порядками сортировки. В первом случае отсутствует чувствительность к регистру, и мы получаем результат 2, порядковый номер прописной буквы "O". Во втором случае используется порядок сортировки `Cyrillic_General_CS_AI_KS_WS`, являющийся чувствительным к регистру. Результатом будет число 6. Это номер первой в строке строчной буквы "o".

Кстати, фраза из А. Фета, используемая в этом примере в операторе `REVERSE()`, фраза-перевертыш, которая одинаково читается как слева направо, так и справа налево, на всякий случай напомню, называется *палиндромом*.

Функция поиска данных по шаблону **PATINDEX()**

Функция **PATINDEX()** позволяет задать весьма сложные условия поиска данных в строке по указанному шаблону. Сейчас мы относительно подробно рассмотрим эту функцию и используемые формы шаблонов. Дело в том, что эту функцию можно использовать и в операторе сложной выборки данных из одной или более таблиц базы данных, в операторе **SELECT**, в предложении **WHERE**, в конструкции **LIKE**.

Синтаксис-то функции простой:

```
PATINDEX(<шаблон>, <строковое выражение>)
```

Функция нечувствительна к регистру (в нашем порядке сортировки), она правильно выполняется для обычных строк и для строк в кодировке Юникод.

Основное богатство функции (и ее сложность) заключается в шаблоне, который помимо обычных символов, используемых для поиска в строке, содержит и специальные шаблонные символы, которые задают дополнительные условия выборки данных.

Используемые шаблонные символы в функции **PATINDEX()** и в конструкции **LIKE** предложений **WHERE** оператора **SELECT** перечислены в табл. 4.8.

Таблица 4.8. Шаблонные символы

Символ	Значение
%	Произвольная строка, содержащая ноль или любое количество символов
_	Любой один символ
[]	Задание диапазона допустимых значений. Подходит любой символ из указанного диапазона. Символы диапазона указываются через дефис, например [a-z]. Это означает, что допустимыми являются все буквы латинского алфавита
[^]	Задание диапазона недопустимых значений. Исключается любой символ из указанного диапазона. Указание [^a-z] означает, что все буквы латинского алфавита являются недопустимыми. В варианте [^abc] недопустимыми будут только перечисленные символы — a, b и c

Для этой функции в задачах реального поиска практически в любом шаблоне должен присутствовать шаблонный символ %. Рассмотрим варианты использования функции **PATINDEX()** (пример 4.11).

Пример 4.11. Использование функции поиска данных по шаблону

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @S VARCHAR(100);
SET @S =
'И начинанья, вознесшиеся мощно, сворачивая
в сторону свой ход, теряют имя действия';
```

```
select PATINDEX('%о_о%', @S),
PATINDEX('и нач%', @S),
PATINDEX('нач%', @S),
PATINDEX('%в_я%', @S),
PATINDEX('%в[А-Я]я%', @S),
PATINDEX('%в[^а]я%', @S),
PATINDEX('%в[^а-я]я%', @S);
```

```
GO
```

Результат:

```
-----  
27      1      0      40      40      84      0
```

Обратите внимание, что строковую константу можно представлять и на нескольких строках, как сделано в этом примере.

В первом обращении к функции `PATINDEX()` осуществляется поиск по шаблону '`%о_о%`'. В этом шаблоне подряд идут два знака подчеркивания (не очень хорошо видно в приведенных текстах), что означает допустимость двух произвольных символов. Результатом такой выборки данных является число 27. Это номер первой буквы "о" в слове "мощно". Здесь мы также видим, что поиск по шаблону не чувствителен к регистру — в шаблоне обе буквы "О" прописные.

В следующем обращении к функции шаблон имеет вид '`и нач%`'. Результат 1. Здесь мы лишний раз убеждаемся, что шаблон нечувствителен к регистру. Шаблон не начинается с символа процента, следовательно, по нему выполняется проверка только самых первых пяти символов исходной строки.

Чтобы проверить наличие проверки с самого начала строки при подобном задании шаблона, в следующем вызове функции шаблон представлен как '`нач%`'. Результатом будет 0, т. е. подстроки, соответствующей этому шаблону (начинающейся с этих символов), не существует в исходной строке.

Шаблону '`%в_я%`' в следующем обращении к функции соответствует две подстроки: в слове "сворачивая" и в "действия". Мы получаем число 40, т. е. первого слева появления нужной подстроки.

В следующем обращении к функции задан шаблон '`%в[А-Я]я%`', который, как и в предыдущем случае, задает поиск трех символов. Для второго, центрального, символа задан диапазон допустимых значений — любая буква кириллицы. Результат будет 40, как и при предыдущем обращении к функции.

Шаблон '`%в[^а]я%`' задает поиск подстроки из трех же букв, только вторая буква не должна быть буквой "а". Этому шаблону соответствует окончание последнего слова в строке "действия". Функция вернет число 84.

Последнему шаблону '`%в[^а-я]я%`' не соответствует ни одна подстрока в исходной строке. Функция возвращает 0.

Наиболее впечатляющие результаты использования функции `PATINDEX()` можно получить в операторе `SELECT`. Там функция применяется не к одной фиксированной строке, а к множеству строковых данных из различных записей таблицы.

* * *

На этом рассмотрение строковых функций закончим. На самом деле мы проигнорировали лишь две функции, которые связаны со звуковым преобразованием строк — SOUNDEX() и DIFFERENCE(), о которых вы при желании сможете прочесть и в Books Online.

4.5. Типы данных даты и времени

4.5.1. Описание типов данных даты и времени

К этой группе относятся следующие типы данных:

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

Характеристики этих типов данных представлены в табл. 4.9.

Таблица 4.9. Типы данных даты и времени

Тип данных	Размер (байт)	Диапазон значений
DATE	3	Дата от 1 января 0001 года до 31 декабря 9999 года. Формат отображения: гггг-мм-дд, имеет 10 символов
TIME	3, 4 или 5	От 00:00:00.0000000 до 23:59:99.9999999
DATETIME	8	Объединяет дату и время. Дата от 1 января 1753 года до 31 декабря 9999 года. Время в диапазоне от 00:00:00 до 23:59:59.997
SMALLDATETIME	4	Объединяет дату и время. Дата от 1 января 1900 года до 6 июня 2079 года. Время в диапазоне от 00:00:00 до 23:59:59
DATETIME2	6, 7, 8	Дата от 1 января 0001 года до 31 декабря 9999 года. Время от 00:00:00.0000000 до 23:59:99.9999999
DATETIMEOFFSET	8, 9, 10	Дата от 1 января 0001 года до 31 декабря 9999 года. Время от 00:00:00.0000000 до 23:59:99.9999999. Учитывает часовой пояс

Тип данных DATE позволяет хранить данные в очень большом диапазоне.

Тип данных TIME хранит время с высокой точностью. Синтаксис задания типа данных TIME:

TIME [(<масштаб>)]

Здесь масштаб — число от 0 до 7. Масштаб задает количество дробных знаков секунд или, как сказано в документации, количество сотен наносекунд. Если масштаб не указан, то предполагается 7.

Тип данных `DATETIME` является объединением типов данных `DATE` и `TIME` с масштабом, который приблизительно можно назвать 3.

Тип данных `SIMILARDATETIME` позволяет в очень компактном виде хранить дату и время, которые вполне могут вас устроить при обработке текущих данных, не связанных с большими промежутками времени.

Тип `DATETIME2` является расширенным вариантом типа данных `DATETIME`. Синтаксис задания `DATETIME2`:

`DATETIME2 [(<масштаб>)]`

`<масштаб>` — число от 0 до 7. Задает количество дробных знаков секунд. Если не указан, то устанавливается 7.

Тип данных `DATETIMEOFFSET` похож на тип данных `DATETIME2`, при этом он еще позволяет учитывать и часовой пояс. Синтаксис задания типа данных `DATETIMEOFFSET`:

`DATETIMEOFFSET [(<точность>, <масштаб>)]`

`<точность>` — целое число от 26 до 34. Задает общее количество знаков; `<масштаб>` — число от 0 до 7. Задает количество дробных знаков секунд. Если точность и масштаб не указаны, то предполагается (34, 7). Помимо даты и времени этот тип данных также содержит величину смещения часового пояса — от -14:00 до +14:00.

Все типы данных, содержащие время, представляют время в 24-часовом формате.

Для работы с датами и временем существует несколько полезных функций.

4.5.2. Действия с датами и временем

Задание даты и времени

Существует много способов задания даты.

В литерале, задающем дату, разделителями между номером дня, номером месяца и годом может служить точка (.), знак минуса (-) или наклонная черта (/). Эти разделители являются взаимозаменяемыми. В одном литерале можно использовать любой набор из разделителей, хотя этого одобрить уж никак нельзя.

Проблема при задании литерала заключается лишь в определении, является ли двузначное число в тексте номером дня или номером месяца. Для задания формата даты используется команда `SET DATEFORMAT`. Параметром команды может быть `mdy`, `dmy`, `ymd`, `ydm`, `udm`, `tud` и `fut`. Набор этих символов определяет порядок, в котором в литерале размещаются элементы даты.

Здесь `d` указывает номер дня, `m` — номер месяца и `y` — год. Например, выполнение команды

```
SET DATEFORMAT dmy;
```

означает, что в литералах вначале записывается день месяца, затем номер месяца и под конец год. Такая форма задания даты является наиболее привычной для русскоязычных пользователей системы.

По умолчанию дата отображается в формате *гггг-мм-дд*, где *гггг* — год, *мм* — номер месяца и *дд* — номер дня в месяце. В системе существует довольно сложная функция преобразования данных `CONVERT()`. Для преобразования даты в нужный формат используется следующий синтаксис этой функции:

`CONVERT(VARCHAR, <преобразуемое выражение>, <стиль>)`

Третий параметр функции `<стиль>` задает формат отображения. Вот некоторые варианты указания стиля:

- ◆ 20, 120 — формат *гггг-мм-дд*;
- ◆ 4 — формат *мм.дд.гг*;
- ◆ 104 — формат *мм.дд.гггг*. Отличается от предыдущего указанием четырехсимвольного года;
- ◆ 1 — формат *мм/дд/гг*;
- ◆ 101 — формат *мм/дд/гггг*.

Существует и ряд других стилей. Для нас наиболее естественным является стиль 4 и стиль 104.

Задать дату можно также и при использовании функции, которая появилась в версии SQL Server 2012, `DATEFROMPARTS()`. Ее синтаксис:

`DATEFROMPARTS(<год>, <месяц>, <день>)`

Варианты задания одной и той же даты, "14 января 2012 года", представлены в примере 4.12. Тут же даны и варианты использования функции `CONVERT()` для форматирования отображаемой даты.

Пример 4.12. Задание даты

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @D1 date;
DECLARE @D2 date;
DECLARE @D3 date;
SET DATEFORMAT dmy;
SET @D1 = '14.01.2012';
SET DATEFORMAT mdy;
SET @D2 = '01-14-2012';
SET DATEFORMAT ymd;
SET @D3 = '2012/01/14';
SELECT CONVERT(CHAR(12), @D1, 20) AS 'Стиль 20',
       CONVERT(CHAR(12), @D1, 4)   AS 'Стиль 4',
```

```

    CONVERT(CHAR(12), @D1, 104) AS 'Стиль 104',
    DATEFROMPARTS(2012, 01, 14) AS 'DATEFROMPARTS';
SELECT CONVERT(CHAR(12), @D1, 1)   AS 'Стиль 1',
    CONVERT(CHAR(12), @D1, 101) AS 'Стиль 101',
    @D1 AS 'По умолчанию';
GO

```

Выполните операторы примера. Результат будет следующим:

Стиль 20	Стиль 4	Стиль 104	DATEFROMPARTS
2012-01-14	14.01.12	14.01.2012	2012-01-14
<hr/>			
Стиль 1	Стиль 101	По умолчанию	
<hr/>			
01/14/12	01/14/2012	2012-01-14	

Время всегда задается одинаковым образом в виде: `чч:мм:сс.ннннннн`. Здесь `чч` — часы, `мм` — минуты, `сс` — секунды, `ннннннн` — дробная часть секунд.

Если в одном литерале задается и дата, и время, то время отделяется от даты пробелом, причем можно ввести произвольное количество пробелов. Например, чтобы указать дату 29 марта 2012 года и время 12 часов 16 минут 53 секунды, нужно написать:

```
'29.03.2012 12:16:53.0000000'
```

Помимо литерала дату и время можно задать также и при помощи функции `DATETIMEFROMPARTS()`. Ее синтаксис:

```
DATETIMEFROMPARTS(<год>, <месяц>, <день>,
                    <часы>, <минуты>, <миллисекунды>)
```

Функция `DATETIME2FROMPARTS()` выполняет похожее создание даты и времени.

```
DATETIME2FROMPARTS(<год>, <месяц>, <день>,
                     <часы>, <минуты>, <секунды>, <дробь>, <точность>)
```

Параметр `<точность>` определяет количество знаков после десятичной точки в дробной части времени. Не может быть меньше, чем количество значащих цифр (отличных от нуля), указанных в параметре `<дробь>`.

Выполните пример 4.13.

Пример 4.13. Задание даты и времени

```

USE master;
GO
SET NOCOUNT ON;
DECLARE @D1 datetime2;
SET DATEFORMAT dmy;
SET @D1 = '29.03.2012 12:16:53.0000021';

```

```

SELECT CAST(@D1 AS VARCHAR(28)) AS 'Дата и время';
SELECT DATETIMEFROMPARTS(2012, 03, 29, 12, 16, 53, 0000021)
    AS 'DATETIMEFROMPARTS';
SELECT DATETIME2FROMPARTS(2012, 03, 29, 12, 16, 53, 000021, 3)
    AS 'DATETIME2FROMPARTS';
GO

```

Результат:

```

Дата и время
-----
2012-03-29 12:16:53.0000021

DATETIMEFROMPARTS
-----
2012-03-29 12:16:53.0000021

DATETIME2FROMPARTS
-----
2012-03-29 12:16:53.021

```

Здесь используется тип данных DATETIME2, который позволяет хранить и дату, и время. Обратите внимание, что для полноценного отображения результата в скрипте используется явное преобразование даты и времени в тип данных VARCHAR(28). Если не выполнить преобразование, то при отображении времени количество дробных знаков будет урезано до двух. Для функций DATETIMEFROMPARTS() и DATETIME2FROMPARTS() такое преобразование не требуется.

Для типа данных DATETIMEOFFSET помимо даты и времени в литерале через один или более пробелов указывается еще и смещение часового пояса. В примере 4.14 показано использование такого типа данных и литерала, задающего смещение в четыре часа, что соответствует смещению московского времени относительно Гринвича.

Пример 4.14. Задание даты, времени и смещения часового пояса

```

USE master;
GO
SET NOCOUNT ON;
DECLARE @D0 DATETIMEOFFSET;
SET @D0 = '29.03.2012 12:16:53.1234567 +04:00';
SELECT @D0 AS 'Дата, время, смещение';
GO

```

Результат:

```

Дата, время, смещение
-----
2012-03-29 12:16:53.1234567 +04:00

```

Функции, возвращающие значение текущей даты, времени и смещения часового пояса

Существуют следующие функции, позволяющие получить текущее значение (установленное на компьютере, где запущен на выполнение SQL Server) даты, времени, смещения.

- ◆ `SYSDATETIME()` — возвращает текущую дату и время, установленные на компьютере.
- ◆ `SYSDATETIMEOFFSET()` — возвращает текущую дату, время и смещение часового пояса данного компьютера.
- ◆ `SYSUTCDATETIME()` — возвращает текущую дату и время.
- ◆ `CURRENT_TIMESTAMP` — возвращает текущую дату и время. Стого говоря, это не функция, а так называемая *контекстная переменная*. После контекстной переменной, в отличие от функций, круглые скобки не ставятся.
- ◆ `GETDATE()` — возвращает текущую дату и время.
- ◆ `GETUTCDATE()` — возвращает текущую дату и время.

Для задания времени можно использовать не только литерал, но и функцию `TIMEFROMPARTS()`:

```
TIMEFROMPARTS(<часы>, <минуты>, <секунды>, <дробь>, <точность>)
```

Параметр `<точность>`, как и в функции `DATETIME2FROMPARTS()`, определяет количество знаков после десятичной точки в дробной части времени. Не может быть меньше, чем количество значащих цифр (отличных от нуля), указанных в параметре `<дробь>`.

В примере 4.15 выполняется вызов некоторых из перечисленных функций и обращение к контекстной переменной. Помимо отображения результатов этих вызовов, выполняется преобразование результатов к строковым данным переменной длины.

Пример 4.15. Получение текущей даты, времени и смещения часового пояса

```
USE master;
GO
SET NOCOUNT ON;
SELECT SYSDATETIME() AS 'SYSDATETIME',
       CAST(SYSDATETIME() AS VARCHAR(30)) AS 'VARCHAR';
SELECT SYSDATETIMEOFFSET() AS 'SYSDATETIMEOFFSET',
       CAST(SYSDATETIMEOFFSET() AS VARCHAR(35)) AS 'VARCHAR';
SELECT SYSUTCDATETIME() AS 'SYSUTCDATETIME',
       CAST(SYSUTCDATETIME() AS VARCHAR(40)) AS 'VARCHAR';
SELECT CURRENT_TIMESTAMP AS 'CURRENT_TIMESTAMP',
       CAST(CURRENT_TIMESTAMP AS VARCHAR(40)) AS 'VARCHAR';
SELECT GETDATE() AS 'GETDATE',
       CAST(GETDATE() AS VARCHAR(40)) AS 'VARCHAR';
SELECT GETUTCDATE() AS 'GETUTCDATE',
       CAST(GETUTCDATE() AS VARCHAR(40)) AS 'VARCHAR';
SELECT TIMEFROMPARTS(12, 16, 53, 21, 4) AS 'TIMEFROMPARTS',
       GO
```

Результат выполнения операторов:

SYSDATETIME	VARCHAR			

2011-11-03 17:28:13.17	2011-11-03 17:28:13.1770806			
SYSDATETIMEOFFSET	VARCHAR			

2011-11-03 17:28:13.1770806	+04:00	2011-11-03 17:28:13.1770806	+04:00	
SYSUTCDATETIME	VARCHAR			

2011-11-03 13:28:13.17	2011-11-03 13:28:13.1770806			
CURRENT_TIMESTAMP	VARCHAR			

2011-11-03 17:28:13.177	Nov	3	2011	5:28PM
GETDATE	VARCHAR			

2011-11-03 17:28:13.177	Nov	3	2011	5:28PM
GETUTCDATE	VARCHAR			

2011-11-03 13:28:13.177	Nov	3	2011	1:28PM
TIMEFROMPARTS				

12:16:53.0021				

Здесь видны отличия в возвращаемых функциями результатах. Если в вашей работе часто приходится работать с датой и временем, не поленитесь, подробно рассмотрите эти примеры.

Функции преобразования и выделения части даты и времени

Существующий в системе оператор `SET LANGUAGE` позволяет установить текущий язык системы. Эта установка в нашем случае влияет на результаты выполнения отдельных функций, которые возвращают названия — функция `DATENAME()`.

По умолчанию в системе используется английский язык. Чтобы задать русский язык, на котором будут выводиться тексты, нужно выполнить оператор

```
SET LANGUAGE N'Russian';
```

Можно выполнить оператор и в таком виде:

```
SET LANGUAGE N'русский';
```

В первом случае для указания языка используется псевдоним, алиас языка — англоязычное его название (от англ. *alias*). Во втором примере используется название языка на этом же самом языке. Такое название записывается в кодировке Юникод. В первом операторе язык тоже указывается как строка Юникод, хотя этого можно и

не делать — псевдоним можно задать и обычной строкой, поскольку он содержит "обычные" символы.

Список всех поддерживаемых в SQL Server языков с их характеристиками (краткие и полные названия месяцев, названия дней недели и ряд других) представлен в *приложении 5*.

Существуют глобальные системные переменные, которые хранят идентификатор и название текущего языка системы. Это переменная `@@LANGUAGE`, которая содержит название языка, и глобальная переменная `@@LANGID`, в которой хранится идентификатор текущего языка. В примере 4.16 после каждого изменения текущего языка отображаются и его характеристики.

Функция `DATENAME()` позволяет выделить из заданной даты отдельные элементы. Синтаксис обращения к функции:

`DATENAME(<выделяемая часть>, <дата>)`

Первый параметр, "выделяемая часть", определяет, какая часть данных будетозвращена функцией и в каком виде. Значения параметра перечислены в табл. 4.10.

Таблица 4.10. Значение параметра "выделяемая часть" функции `DATENAME()`

Параметр	Сокращения	Возвращаемое значение
<code>year</code>	<code>YY, yyyy</code>	Четырехзначное значение года
<code>quarter</code>	<code>qq, q</code>	Номер квартала
<code>month</code>	<code>mm, m</code>	Название месяца
<code>dayofyear</code>	<code>dy, y</code>	Номер дня в году
<code>day</code>	<code>dd, d</code>	День в месяце
<code>week</code>	<code>wk, ww</code>	Номер недели в году
<code>weekday</code>	<code>dw</code>	Название дня недели
<code>hour</code>	<code>hh</code>	Часы
<code>minute</code>	<code>mi, n</code>	Минуты
<code>second</code>	<code>ss, s</code>	Секунды
<code>millisecond</code>	<code>ms</code>	Миллисекунды
<code>microsecond</code>	<code>mcs</code>	Микросекунды
<code>nanosecond</code>	<code>ns</code>	Наносекунды
<code>TZoffset</code>	<code>tz</code>	Смещение часового пояса

При вызове функции вы можете использовать как полное название выделяемой части, так и сокращение.

Функция `DATENAME()` позволяет получить любой фрагмент даты и времени. Есть функция `DATEPART()`, которая позволяет похожим образом выделять фрагменты даты и времени. Синтаксис функции такой же, как и у `DATENAME()`:

`DATEPART(<выделяемая часть>, <дата>)`

Значения параметра функции <выделяемая часть> представлены в табл. 4.11.

Таблица 4.11. Значение параметра "выделяемая часть" функции DATEPART()

Параметр	Сокращения	Возвращаемое значение
year	yy, yyyy	Четырехзначное значение года
quarter	qq, q	Номер квартала
month	mm, m	Номер месяца в году
dayofyear	dy, y	Номер дня в году
day	dd, d	День в месяце
week	wk, ww	Номер недели в году
weekday	dw	Номер дня в неделе
hour	hh	Часы
minute	mi, n	Минуты
second	ss, s	Секунды
millisecond	ms	Миллисекунды
microsecond	mcs	Микросекунды
nanosecond	ns	Наносекунды
TZoffset	tz	Смещение часового пояса

Параметры и результаты мало отличаются от функции DATENAME(). Видно, что параметр month позволяет получить не название, а номер месяца в году.

В примере 4.16 показаны различные варианты обращения к функциям DATENAME() и DATEPART() для получения разных элементов даты на нескольких языках.

Пример 4.16. Получение элементов даты при использовании функций DATENAME() и DATEPART()

```
USE master;
GO
SET NOCOUNT ON;
SET DATEFORMAT dmy;
DECLARE @D DATE;
SET @D = '23.03.2012';
-- English
SET LANGUAGE 'English';
SELECT @@LANGID AS 'LANGID',
      @@LANGUAGE AS 'LANGUAGE';
-- Использование функции DATENAME
SELECT DATENAME(day, @D) AS 'Day',
       DATENAME(year, @D) AS 'Year';
```

```

SELECT DATENAME(quarter, @D) AS 'Quarter',
       DATENAME(dayofyear, @D) AS 'Day of Year',
       DATENAME(week, @D) AS 'Week';
SELECT DATENAME(month, @D) AS 'Month Name',
       DATENAME(weekday, @D) AS 'Weekday';
-- Russian
SET LANGUAGE 'Russian';
SELECT @@LANGID AS 'LANGID',
      @@LANGUAGE AS 'LANGUAGE';
SELECT DATENAME(month, @D) AS 'Месяц',
       DATENAME(weekday, @D) AS 'День недели';
-- Japanese
SET LANGUAGE 'Japanese';
SELECT @@LANGID AS 'LANGID',
      @@LANGUAGE AS 'LANGUAGE';
SELECT DATENAME(weekday, @D) AS 'День недели';

DECLARE @D1 DATETIMEOFFSET;
SET @D1 = SYSDATETIMEOFFSET();
SELECT DATENAME(TZoffset, @D1) AS 'Смещение часового пояса';
-- Использование функции DATEPART
SET LANGUAGE 'English';
SELECT DATEPART(day, @D) AS 'Day',
      DATEPART(year, @D) AS 'Year';
SELECT DATEPART(quarter, @D) AS 'Quarter',
      DATEPART(dayofyear, @D) AS 'Day of Year',
      DATEPART(week, @D) AS 'Week';
SELECT DATEPART(month, @D) AS 'Month Number',
      DATEPART(weekday, @D) AS 'Weekday';
GO

```

Результат выполнения операторов:

Changed language setting to us_english.

LANGID LANGUAGE

0 us_english

Day	Year
-----	------

23	2012
----	------

Quarter	Day of Year	Week
---------	-------------	------

1	83	12
---	----	----

Month Name	Weekday
------------	---------

March	Friday
-------	--------

Параметры языка изменены на "русский".

LANGID LANGUAGE

21	русский
----	---------

Месяц	День недели
-------	-------------

Март	пятница
------	---------

言語設定が 日本語 に変更されました。

LANGID LANGUAGE

3	日本語
---	-----

День недели

金曜日

Смещение часового пояса

+04:00

Changed language setting to us_english.

Day	Year
-----	------

23	2012
----	------

Quarter	Day of Year	Week
---------	-------------	------

1	83	12
---	----	----

Month Number Weekday

3	6
---	---

Мы видим, что в зависимости от используемого языка название месяца и дня недели выводятся соответствующими текстами. После изменения текущего языка системы мы отображаем и его характеристики — идентификатор и название.

Похожие результаты получены и при использовании функции DATEPART().

В примере 4.17 показаны обращения к функциям DATENAME() и DATEPART() для получения элементов времени.

Пример 4.17. Получение элементов времени при использовании функций DATENAME () и DATEPART ()

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @D DATETIMEOFFSET;
SET @D = SYSDATETIMEOFFSET();

SELECT DATENAME(hour, @D) AS 'Hour',
       DATENAME(minute, @D) AS 'Minute',
       DATENAME(second, @D) AS 'Second';
SELECT DATENAME(millisecond, @D) AS 'Millisecond',
       DATENAME(microsecond, @D) AS 'Microsecond',
       DATENAME(nanosecond, @D) AS 'Nanosecond';

SELECT DATEPART(hour, @D) AS 'Hour',
       DATEPART(minute, @D) AS 'Minute',
       DATEPART(second, @D) AS 'Second';
SELECT DATEPART(millisecond, @D) AS 'Millisecond',
       DATEPART(microsecond, @D) AS 'Microsecond',
       DATEPART(nanosecond, @D) AS 'Nanosecond';
GO
```

Результат:

Hour	Minute	Second
23	34	47
Millisecond	Microsecond	Nanosecond
496	496271	496271000
Hour	Minute	Second
23	34	47
Millisecond	Microsecond	Nanosecond
496	496271	496271000

Видно, что результаты выделения элементов времени, полученные при помощи этих двух функций, совершенно одинаковы.

Другие функции для даты и времени

Существуют и другие функции для типов данных даты и времени. Вот некоторые из них.

- ◆ DATETIMEOFFSETFROMPARTS (<год>, <месяц>, <день>, <часы>, <минуты>, <секунды>, <дробь>, <смещение часов>, <смещение минут>, <точность>)

Функция возвращает тип данных DATETIMEOFFSET. Помимо собственно даты и времени дает смещение часов и минут.

◆ **SMALLDATETIMEFROMPARTS (<год>, <месяц>, <день>, <часы>, <минуты>)**

Аналогична функции DATETIMEFROMPARTS(), но возвращает значение даты и времени типа SMALLDATETIME.

◆ **DATEADD(<часть даты>, <интервал>, <дата>)**

Для заданной даты указанная часть даты суммируется с интервалом. Полученная дата возвращается функцией. <часть даты> задается фиксированным текстом в диапазоне от года до наносекунд. Подробности см. в Books Online.

◆ **EOMONTH(<начальная дата> [, <добавляемые месяцы>])**

Функция возвращает последний номер дня и время для заданного месяца. Если задан только первый параметр, то возвращается дата и время последнего дня указанного месяца. Например, для февраля будет возвращаться 28 либо 29, если год високосный. Второй параметр задает число, на которое нужно увеличить (или уменьшить, если число отрицательное) номер указанного месяца.

4.6. Двоичные данные

Двоичные типы данных содержат произвольные (именно двоичные) данные. Существует три типа таких данных: BINARY, VARBINARY и IMAGE. Тип данных IMAGE будет удален из системы в какой-нибудь из следующих версий. Вместо него следует использовать тип данных VARBINARY (MAX).

Наибольший интерес представляет тип данных VARBINARY. Его синоним — BINARY VARYING. Синтаксис задания типа данных:

`VARBINARY [({<размер> | max})]`

Размер задает количество байтов, которое отводится для хранения данных. Он может изменяться от 1 до 8000. Если задано ключевое слово max, то данное может занимать до $2^{31} - 1$ байтов или около 2 Гбайт. Несколько слов про такой вариант использования двоичных данных мы уже сказали чуть ранее в разд. 4.4.3. Если для хранения таких данных будут использоваться файловые потоки, которые мы рассмотрим в следующей главе, то и этот размер может быть увеличен.

Этот тип данных может использоваться для хранения любых данных: текстов в различных форматах, изображений, звука, видео, выполняемых файлов.

Напомню, что двоичная константа должна начинаться с символов 0x, за которыми следуют шестнадцатеричные цифры — цифры от 0 до 9 и буквы латинского алфавита от A до F в любом регистре.

Пример простого использования типа данных VARBINARY мы рассмотрим, когда будем говорить о пространственных типах данных в следующем разделе.

4.7. Пространственные типы данных

Пространственные (spatial) типы данных появились еще в SQL Server 2008. Международным консорциумом Open Geospatial Consortium (OGC, открытый картографический консорциум) разработаны спецификации OpenGIS (Open Geographic Information Systems, открытые географические информационные системы), определяющие характер интерфейса и функциональность в работе с пространственными данными. Эти спецификации положены в основу работы с подобными данными в SQL Server.

К пространственным типам данных в SQL Server относятся GEOMETRY и GEOGRAPHY. С их помощью можно создавать различные геометрические фигуры — точки, линии, многоугольники или, как их еще называют в англоязычной литературе, *полигоны*. Эти типы данных позволяют задать собственно фигуру, ее внешний вид и ее местоположение в некоторой системе координат. В базе данных оба типа данных хранятся в одинаковом формате — в виде потока двоичных данных. Размер поля, отводимого для хранения таких данных, является переменным.

Тип данных GEOMETRY применяется к плоским фигурам, имеет прямое отношение к Евклидовой геометрии, где все объекты располагаются на плоской поверхности.

Тип данных GEOGRAPHY используется для задания фигур, объектов, определения расстояний в условиях поверхности Земли, т. е. с учетом формы Земли, которая, как известно, является приплюснутым сфероидом. Здесь важны не только размеры фигур, но и их расположение на земной поверхности. Для объектов этого типа данных существует одно ограничение — размер объекта не должен выходить за пределы полусфера (именно половины сферы) Земли, независимо от того, где расположен объект. Это касается только расстояния между любыми точками географического объекта и никак не связано с тем, в каком, например, полушарии располагается объект.

Замечательной особенностью поддержки пространственных типов данных в SQL Server является возможность создания индексов для этих типов данных.

В SQL Server пространственные типы данных реализованы как системные типы данных Microsoft .NET Framework CLR. Эти типы данных также распространяются в виде отдельной бесплатной библиотеки в составе SQL Server Feature Pack, что позволяет использовать их в любом приложении, не устанавливая SQL Server.

Система .NET Framework является программной средой, которая устанавливается на компьютере вместе с SQL Server.

Для внешнего представления пространственных типов данных SQL Server использует так называемый текстовый формат WKT (well-known text), определенный консорциумом OGC. В базе данных такие данные хранятся в двоичном виде. Форма внутреннего хранения также определена в стандартах OGC. Это двоичный формат WKB (well-known binary).

Вот пространственные объекты, определенные OGC и поддерживаемые SQL Server:

- ◆ Point — точка;
- ◆ LineString — ломаная линия. Простейший вариант ломаной — это линия, соединяющая две точки;
- ◆ Polygon — многоугольник, полигон. Его можно рассматривать как замкнутый объект LineString.

Это базовые геометрические объекты. Из них можно построить любые пространственные данные. Существуют и другие объекты.

- ◆ GeomCollection — коллекция экземпляров объектов GEOMETRY или GEOGRAPHY: набор линий, точек, многоугольников.
- ◆ MultiPoint — коллекция точек.
- ◆ MultiLineString — коллекция экземпляров типа LineString.
- ◆ MultiPolygon — коллекция экземпляров типа Polygon.

При работе с геопространственными данными (GEOGRAPHY) используется понятие пространственной ссылки (spatial reference, SR) и идентификатора пространственной ссылки SRID. Эта система позволяет с определенной степенью точности характеризовать размеры и поверхность Земли. Более детально рассмотрим ее чуть позже при описании типа данных GEOGRAPHY.

Для задания значения столбцам таблиц и локальным переменным пространственных типов данных используются многочисленные методы из пространства имен .NET CLR `geometry` или `geography`.

4.7.1. Тип данных **GEOMETRY**

Этот тип данных представляет геометрические объекты на двумерной плоской поверхности. Характеристиками объекта являются его координаты в плоской системе координат, уровень, мера и идентификатор пространственной ссылки (SRID, Spatial Reference Identifier). Значение SRID может указываться в диапазоне от 0 до 999 999. В этом типе данных SRID никакого отношения к форме Земли не имеет. Его можно рассматривать, как указатель на конкретную плоскость, на которой располагаются геометрические объекты. Плоскости с различными значениями SRID нигде не "соприкасаются": любые операции над геометрическими объектами с разными SRID всегда дадут значение `NULL`. По умолчанию для объектов при их создании устанавливается SRID в значение 0.

Рассмотрим объекты этого типа данных.

4.7.1.1. Точка

Точка, `Point`, задается с указанием ее координат *X* и *Y*. При задании точки также можно указать уровень *Z* и меру *M*. Уровень и меру я не хочу здесь рассматривать, поскольку они не слишком уж нужны при работе с геометрическими типами данных.

ВНИМАНИЕ!

Следует учитывать то, что имена всех свойств и методов чувствительны к регистру. Вы должны соблюдать правила ввода имен, указывая прописные буквы там, где это задано в синтаксисе.

Свойства

Следующие свойства типа данных GEOMETRY позволяют получить характеристики объекта точка:

- ◆ STX — координата X;
- ◆ STY — координата Y;
- ◆ STSrid — идентификатор пространственной ссылки SRID;
- ◆ z — уровень;
- ◆ m — мера;
- ◆ IsNull — возвращает TRUE, если экземпляр NULL, т. е. не имеет никакого объекта. Это свойство может применяться к любому объекту любого пространственного типа данных.

Методы

Методы для работы с объектом точка можно разбить на следующие группы:

- ◆ методы отображения характеристик объекта точка;
- ◆ методы создания объекта;
- ◆ дополнительные методы.

Отображение характеристик

Вот некоторые методы, позволяющие получить характеристики объекта точка.

- ◆ Метод `STGeometryType()` — возвращает текст, отражающий тип геометрического объекта. Его можно применять для любого пространственного объекта. Для точки будет возвращена строка "Point".
- ◆ Метод `ToString()` — позволяет отобразить текст, соответствующий геометрическому объекту, в текстовом формате WKT.
- ◆ Метод `STAsText()` — дает такой же результат, что и `ToString()`.
- ◆ Метод `STAsBinary()` — позволяет отобразить текст, соответствующий геометрическому объекту, в двоичном формате WKB.

Создание объекта

Для создания точки можно использовать следующие статические геометрические методы из пространства имен .NET `geometry`:

- ◆ `STPointFromText();`
- ◆ `STGeomFromText();`
- ◆ `Point();`

- ◆ Parse();
- ◆ STPointFromWKB();
- ◆ STGeomFromWKB().

ЗАМЕЧАНИЕ

Пожалуй, выражение "создание объекта" в этом контексте не очень-то правильное. Здесь речь идет не о создании нового объекта (он создается в виде локальной переменной в операторе DECLARE или в виде столбца таблицы в операторе CREATE TABLE). При помощи перечисленных методов формируется значение, которое присваивается соответствующему геометрическому объекту при использовании операторов SET для локальной переменной или INSERT для столбца таблицы. Однако такая терминология постоянно используется в русскоязычной литературе и в русском варианте Books Online. Продолжим эту традицию.

Методы STPointFromText(), Point() и STPointFromWKB() позволяют получить только точку. Если параметр, передаваемый такому методу, описывает любой другой объект, то будет сгенерировано сообщение об ошибке.

Например, попытка выполнить следующий оператор присваивания значения полигона локальной переменной @P1, которая была описана как точка, вызовет ошибку.

```
SET @P1 = geometry::STPointFromText
      ('POLYGON((0 0, 3 0, 3 3, 0 3, 0 0), (2 2, 2 1, 1 1, 1 2, 2 2))', 1);
```

Другие три метода позволяют создавать любой геометрический объект.

Синтаксические конструкции этих шести методов представлены в листинге 4.2. При обращении к любому из перечисленных методов имени метода должен предшествовать текст geometry:: для указания пространства имен, где определен метод.

Листинг 4.2. Синтаксис операторов создания точки

```
STPointFromText('<текст WKT>', SRID)
STGeomFromText('<текст WKT>', SRID)
Point(<координата X>, <координата Y>, SRID)
Parse('POINT(<координата X> <координата Y> SRID)')
STPointFromWKB(<двоичный литерал>, SRID)
STGeomFromWKB(<двоичный литерал>, SRID)
```

Обратите внимание, что в большинстве методов значения параметров отделяются друг от друга запятыми. В методе Parse() параметры отделяются пробелами.

В этих конструкциях текст в формате WKT должен содержать функцию задания геометрической точки. Текст представляется в виде

```
'POINT(<координата X>, <координата Y>)'
```

Идентификатор пространственной ссылки SRID обычно указывается 0.

В примере 4.18 показано использование всех перечисленных методов создания точек и дан вариант отображения их характеристик.

Пример 4.18. Создание точек и отображение их характеристик

```
USE master;
GO
SET NOCOUNT ON;

DECLARE @P1 geometry;
DECLARE @P2 geometry;
DECLARE @P3 geometry;
DECLARE @P4 geometry;
DECLARE @P5 geometry;
DECLARE @P6 geometry;

SET @P1 = geometry::STPointFromText('POINT (3 4)', 0);
SET @P2 = geometry::STGeomFromText('POINT (3 4)', 0);
SET @P3 = geometry::Point(3, 4, 0);
SET @P4 = geometry::Parse('POINT(3 4 0)');
SET @P5 =
geometry::STPointFromWKB(0x010100000000000000000000000000840000000000000001040, 0);
SET @P6 = geometry::STGeomFromWKB(0x01010000000000000000000000000084000000000000001040, 0);

SELECT CAST(@P1.STX AS CHAR(3)) AS 'STX',
      CAST(@P1.STY AS CHAR(3)) AS 'STY',
      CAST(@P1.ToString() AS CHAR(11)) AS 'ToString',
      CAST(@P1.STAsText() AS CHAR(11)) AS 'STAsText',
      CAST(@P1.STSrid AS CHAR(6)) AS 'STSrid';
SELECT @P1.STAsBinary() AS 'STAsBinary';
SELECT CAST(@P2.STX AS CHAR(3)) AS 'STX',
      CAST(@P2.STY AS CHAR(3)) AS 'STY',
      CAST(@P2.ToString() AS CHAR(11)) AS 'ToString',
      CAST(@P2.STAsText() AS CHAR(11)) AS 'STAsText',
      CAST(@P2.STSrid AS CHAR(6)) AS 'STSrid';
SELECT @P2.STAsBinary() AS 'STAsBinary';
SELECT CAST(@P3.STX AS CHAR(3)) AS 'STX',
      CAST(@P3.STY AS CHAR(3)) AS 'STY',
      CAST(@P3.ToString() AS CHAR(11)) AS 'ToString',
      CAST(@P3.STAsText() AS CHAR(11)) AS 'STAsText',
      CAST(@P3.STSrid AS CHAR(6)) AS 'STSrid';
SELECT @P3.STAsBinary() AS 'STAsBinary';
SELECT CAST(@P4.STX AS CHAR(3)) AS 'STX',
      CAST(@P4.STY AS CHAR(3)) AS 'STY',
      CAST(@P4.ToString() AS CHAR(11)) AS 'ToString',
      CAST(@P4.STAsText() AS CHAR(11)) AS 'STAsText',
      CAST(@P4.STSrid AS CHAR(6)) AS 'STSrid';
SELECT @P4.STAsBinary() AS 'STAsBinary';
SELECT CAST(@P5.STX AS CHAR(3)) AS 'STX',
      CAST(@P5.STY AS CHAR(3)) AS 'STY',
      CAST(@P5.ToString() AS CHAR(11)) AS 'ToString',
```

```
        CAST(@P5.STAsText() AS CHAR(11)) AS 'STAsText',
        CAST(@P5.STSrid AS CHAR(6)) AS 'STSrid';
SELECT @P5.STAsBinary() AS 'STAsBinary';
SELECT CAST(@P6.STX AS CHAR(3)) AS 'STX',
       CAST(@P6.STY AS CHAR(3)) AS 'STY',
       CAST(@P6.ToString() AS CHAR(11)) AS 'ToString',
       CAST(@P6.STAsText() AS CHAR(11)) AS 'STAsText',
       CAST(@P6.STSrid AS CHAR(6)) AS 'STSrid';
SELECT @P6.STAsBinary() AS 'STAsBinary';
GO
```

Здесь создается шесть локальных переменных типа данных GEOMETRY. С использованием шести описанных методов им присваивается одно и то же значение геометрического объекта точки с координатами $X = 3$, $Y = 4$. Последующие операторы SELECT отображают набор характеристик каждого из шести объектов. Это координаты X и Y , два варианта отображения текста в формате WKT (метод `ToString()` и метод `STAsText()`) и двоичное представление объекта в формате WKB.

ЗАМЕЧАНИЕ

При отображении строк в операторе SELECT использована функция преобразования CAST() исключительно для того, чтобы данные умещались на одной строке.

Для каждой из локальных переменных будет получено одинаковое отображение ее характеристик:

STX	STY	ToString	STAsText	STSrId
3	4	POINT (3 4)	POINT (3 4)	0

STAsBinary
0x01010000000000000000000000000008400000000000000000000001040

ЗАМЕЧАНИЕ

Как вы, наверное, заметили, последняя строка в результатах этого примера содержит двоичные данные, которые и были использованы при присваивании значений локальным переменным `@P5` и `@P6`.

Дополнительные методы

Рассмотрим еще несколько методов, применимых к геометрическим точкам.

- ◆ Метод `ST_Equals()` — сравнивает два геометрических объекта и возвращает значение 1, если оба объекта идентичны, т. е. совпадают их формы и местоположение. Иначе метод возвращает 0. Можно сравнивать только объекты с одинаковым значением идентификатора пространственной ссылки SRID.
 - ◆ Метод `ST_Distance()` — вычисляет расстояние между двумя точками на плоскости. Расстояние можно получить только между объектами с одинаковым значением идентификатора SRID. Этот метод также может быть применен и к гео-

графическим объектам. Чуть позже мы его используем для вычисления расстояния между различными городами на нашей планете.

- ◆ Метод `STLength()` — определяет длину объекта. Для точки длина имеет значение 0. Более осмысленные результаты можно получить для других геометрических объектов.
- ◆ Метод `STDimension()` — возвращает размерность объекта. Для точки это число 0. В примере 4.19 показано использование этих методов.

Пример 4.19. Вычисление расстояния между точками и сравнение двух объектов, определение "длины" точки

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @P1 geometry;
DECLARE @P2 geometry;
DECLARE @ResultOfComparison VARCHAR(5);

SET @P1 = geometry::STGeomFromText('POINT (3 4)', 0);
SET @P2 = geometry::Point(1, 6, 0);
IF @P1.STEquals(@P2) = 1
    SET @ResultOfComparison = 'True'
ELSE
    SET @ResultOfComparison = 'False';

SELECT @P1.STDistance(@P2) AS 'STDistance',
    @ResultOfComparison AS 'Result Of Comparison',
    @P1.STLength() AS 'Length';
GO
```

Здесь объявляются две геометрические локальные переменные, им присваиваются объекты точка. Положение этих точек на поверхности различное.

Для отображения результата сравнения геометрических объектов создается переменная символьного типа данных. Ей в операторе `IF` присваивается соответствующее текстовое значение `True` или `False`.

Оператор `SELECT` отображает расстояние между точками, результат сравнения и длину точки. При выполнении пакета будут выведены следующие данные:

STDistance	Result Of Comparison	Length
2,82842712474619	False	0

4.7.1.2. Ломаная линия

Линия, `LineString` является ломаной линией. Задается в виде последовательности точек, между которыми располагаются соединяющие их прямые линии. Объект

может содержать не менее двух точек. В частном случае объект может быть замкнутым, когда начальная точка совпадает с конечной.

Для создания линии используются следующие методы из пространства имен `geometry`:

- ◆ `STGeomFromText()`;
- ◆ `STLineFromText()`;
- ◆ `Parse()`;
- ◆ `STLineFromWKB()`;
- ◆ `STGeomFromWKB()`.

Текст для создания линии в методах `STGeomFromText()`, `STLineFromText()` и `Parse()` имеет следующий синтаксис:

```
'LINESTRING(<координата X> <координата Y>, <координата X> <координата Y>
[, <координата X> <координата Y>] ...)'
```

Объект должен содержать не менее двух точек. Это видно и по синтаксису. При попытке создать объект, содержащий лишь одну точку, вы получите сообщение об ошибке (что с моей точки зрения не очень уж правильно — можно было бы рассматривать одну точку как граничный случай ломаной линии). Координата *Y* отделяется от координаты *X* пробелом. Описания точек отделяются друг от друга запятыми.

Для отображения характеристик ломаной линии используются те же методы, что и для объекта точка.

- ◆ Метод `STGeometryType()` — возвращает текст, отражающий тип геометрического объекта. В данном случае это строка "LineString".
- ◆ Методы `ToString()` и `STAsText()` — отображают текст, соответствующий геометрическому объекту в формате WKT.
- ◆ Метод `STAsBinary()` — отображает текст в двоичном формате WKB.

Другие методы для *LineString*.

- ◆ Метод `STEquals()` — сравнивает два геометрических объекта. Возвращает 1, если объекты равны, и 0, если не равны.
- ◆ Метод `STDistance()` — вычисляет минимальное расстояние между двумя объектами на плоскости, т. е. между двумя точками объектов, минимально удаленных друг от друга.
- ◆ Метод `STDimension()` — возвращает размерность объекта. Для линии это число 1.
- ◆ Метод `STLength()` — определяет общую длину объекта — сумму длин всех линий.
- ◆ Методы `STStartPoint()` и `STEndPoint()` — возвращают текст в формате WKT, соответственно, начальной и конечной точки линии.

Для получения идентификатора пространственной ссылки здесь также можно использовать свойство STSrid.

В примере 4.20 показано использование этих методов.

Пример 4.20. Использование методов для объектов ломаная линия

```
USE master;
GO
SET NOCOUNT ON;

DECLARE @L1 geometry;
DECLARE @L2 geometry;
DECLARE @L3 geometry;
DECLARE @L4 geometry;
DECLARE @B varbinary(200);
DECLARE @ResultOfComparison VARCHAR(5);

SET @L1 = geometry::STGeomFromText('LINESTRING (1 1, 1 9)', 0);
SET @L2 = geometry::Parse('LINESTRING (2 6, 5 1, 2 1)');
SET @L3 = geometry::STLineFromText('LINESTRING (8 1, 6 8, 12 8, 12 1)', 0);
SET @B =
0x0102000000020000000000000000F03F000000000000F03F000000000000F03F000000000000
02240;
SET @L4 = geometry::STGeomFromWKB(@B, 0);

SELECT CAST(@L3.STStartPoint().ToString() AS CHAR(14)) AS 'StartPoint',
       CAST(@L3.STEndPoint().ToString() AS CHAR(14)) AS 'EndPoint',
       @L3.STLength() AS 'Length',
       @L1.STDistance(@L3) AS 'Distance';
SELECT @L1.STAsBinary() AS 'STAsBinary';
SELECT CAST(@L1.ToString() AS CHAR(29)) AS 'ToString',
       CAST(@L1.STAsText() AS CHAR(29)) AS 'STAsText',
       CAST(@L1.STSrid AS CHAR(6)) AS 'STSrid';

IF @L1.STEquals(@L4) = 1
    SET @ResultOfComparison = 'True'
ELSE
    SET @ResultOfComparison = 'False';

SELECT @ResultOfComparison AS 'Result Of Comparison';
GO
```

Здесь объявляется несколько локальных переменных типа GEOMETRY. Последующие операторы SET демонстрируют различные варианты присваивания им значения ломаной линии. Объявлена также локальная переменная @B типа данных VARBINARY. Она используется для присваивания значения переменной @L4 с помощью метода STGeomFromWKB().

Первой переменной @L1 задается значение двух точек, т. е. переменная будет содержать только одну вертикальную линию.

Переменная @L2 содержит две линии. Третья переменная @L3 — три линии. Переменной @L4 присваивается значение, получаемое из типа данных BINARY. В результате полученный геометрический объект полностью совпадает с объектом, описанным в переменной @L1.

Созданные геометрические объекты представлены на рис. 4.2.

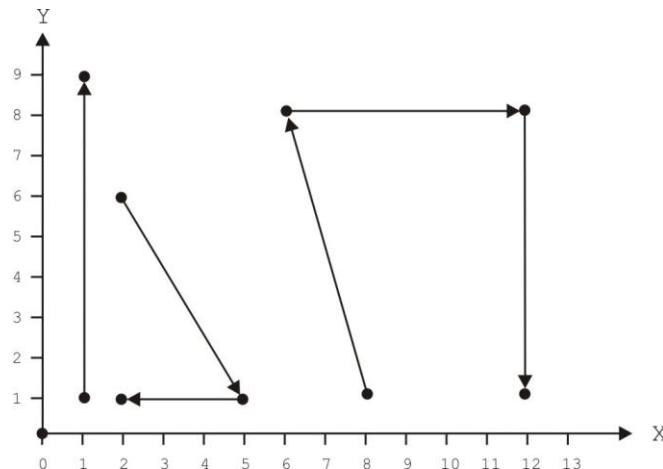


Рис. 4.2. Геометрические объекты ломаная линия

На рисунке стрелки отображают лишь порядок формирования ломаной линии. Объекты на рисунке расположены слева направо, т. е. крайний левый это @L1, затем @L2, потом @L3. Объект @L4 совпадает с @L1.

В результате выполнения операторов пакета будут получены следующие данные:

StartPoint	EndPoint	Length	Distance
POINT (8 1)	POINT (12 1)	20,2801098892805	5
<hr/>			
<hr/>			
STAsBinary			
<hr/>			
0x010200000002000000000000000000F03F000000000000F03F000000000000F03F000000000000			
02240			
<hr/>			
ToString	STAsText	STSrid	
LINESTRING (1 1, 1 9)	LINESTRING (1 1, 1 9)	0	
<hr/>			
Result Of Comparison			
<hr/>			
True			

Здесь также демонстрируется использование методов получения начальной и конечной точки в ломаной линии, общей длины линии и расстояния между геометрическими объектами. Последние строки задают проверку равенства двух объектов, созданных разными способами. Эти объекты действительно одинаковы, что и подтверждает результат сравнения.

4.7.1.3. Полигон

Полигон (Polygon) или многогранник является замкнутым объектом. Он должен содержать не менее трех точек. Объект должен быть замкнутым, т. е. последняя точка в его описании должна совпадать с первой. Полигон может включать и внутренние кольца, полигоны. Внутренние объекты не должны пересекаться, но могут соприкасаться в точках по касательной.

Для создания полигона можно использовать методы `STPolygonFromText()`, `Parse()`, `STGeomFromWKB()` и `STGeomFromText()`. При задании текста в формате WKT используется ключевое слово "POLYGON".

Поскольку полигоны создаются замкнутыми ломаными линиями, для них применима функция вычисления площади. Это выполняется при помощи метода `STArea()`.

Для полигона можно использовать еще ряд методов. Рассмотрим только один из них `STPointOnSurface()` (в переводе на русский язык — точка на поверхности). Этот метод возвращает геометрический объект `Point` — произвольную точку внутри полигона.

В примере 4.21 показаны операторы создания нескольких полигонов и отображения некоторых их характеристик.

Пример 4.21. Использование методов для объектов полигон

```
USE master;
GO
SET NOCOUNT ON;

DECLARE @P1 geometry;
DECLARE @P2 geometry;
DECLARE @P3 geometry;
DECLARE @B varbinary(200);

SET @P1 = geometry::STPolyFromText('POLYGON((1 1, 1 3, 2 1, 1 1))', 0);
SET @P2 = geometry::STGeomFromText('POLYGON((0 0, 0 3, 3 3, 3 0, 0 0),
                                         (1 1, 1 2, 2 1, 1 1))', 0);
SET @B =
0x01030000000100000040000000000000000F03F000000000000F03F000000000000F03F000
00000000084000000000000004000000000000F03F000000000000F03F000000000000F03F
SET @P3 = geometry::STGeomFromWKB(@B, 0);
```

```

SELECT @P1.STAsBinary() AS 'STAsBinary';
SELECT CAST(@P1 AS VARCHAR(32)) AS 'P1';
SELECT CAST(@P2 AS VARCHAR(58)) AS 'P2';
SELECT CAST(@P3 AS VARCHAR(32)) AS 'P3';
SELECT @P1.STArea() AS 'Area P1', @P2.STArea() AS 'Area P2';
SELECT @P1.STPointOnSurface().ToString() AS 'STPointOnSurface';
GO

```

Созданные объекты показаны на рис. 4.3. Крайний слева — полигон @P1. Это прямой треугольник. Он совпадает с полигоном @P3. Второй — полигон @P2, прямой четырехугольник. Стрелки на этом рисунке также отражают только порядок, в котором описывается создание объекта.

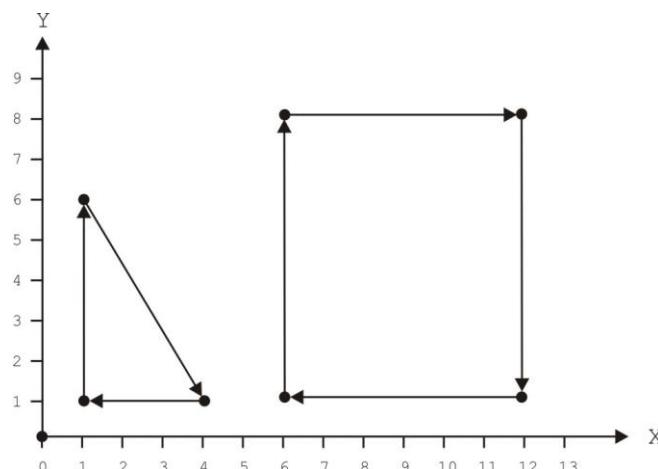


Рис. 4.3. Геометрические объекты полигон

Результат выполнения пакета:

STAsBinary

```
0x010300000001000000040000000000000000F03F000000000000F03F000000000000F03F000
00000000084000000000000004000000000000F03F000000000000F03F000000000000F03F
```

P1

```
POLYGON ((1 1, 1 3, 2 1, 1 1))
```

P2

```
POLYGON ((0 0, 0 3, 3 3, 3 0, 0 0), (1 1, 1 2, 2 1, 1 1))
```

P3

```
POLYGON ((1 1, 1 3, 2 1, 1 1))
```

Area P1	Area P2
1	8,5
STPointOnSurface	
POINT (1.33333333333357 1.666666666666714)	

После создания полигонов отображается в двоичном и текстовом виде их вид. Расчитывается и отображается площадь полигонов @P1 и @P2. К полигону @P1 применяется метод STPointOnSurface().

4.7.1.4. Другие геометрические объекты

Вкратце рассмотрим другие геометрические объекты. Методы для этих объектов также выбираются из пространства имен `geometry`.

Объект `GeomCollection` позволяет хранить коллекцию различных экземпляров геометрических объектов. Для создания объекта используются обычные методы `STGeomFromText()`, `STGeomFromWKB()`, `Parse()` и метод, предназначенный для создания только коллекций `STGeomCollFromText()`.

Объект `MultiPoint` хранит коллекцию точек. Для создания объекта используются методы `STGeomFromText()`, `STGeomFromWKB()`, `Parse()` и `STMPointFromText()`.

В объекте `MultiLineString` хранится коллекция ломаных линий. Для создания объекта используются методы `STGeomFromText()`, `STGeomFromWKB()`, `STMLineFromText()`, `STMLineFromWKB()`, `Parse()` и `STMLineStringFromText()`.

Отображение всех объектов можно выполнить с использованием методов `STAsBinary()`, `STGeometryType()`, `ToString()`, `STAsText()`.

В следующем примере 4.22 показано использование методов для этих геометрических объектов.

Пример 4.22. Использование методов для объектов `GeomCollection`, `MultiPoint`, `MultiLineString`

```
USE master;
GO
SET NOCOUNT ON;

DECLARE @GC1 geometry;
DECLARE @GC2 geometry;
DECLARE @MP geometry;
DECLARE @ML geometry;

SET @GC1 = geometry::STGeomCollFromText
('GEOMETRYCOLLECTION(POINT (3 4), POLYGON((1 1, 1 3, 2 1, 1 1)))', 0);
SET @GC2 = geometry::STGeomFromText
('GEOMETRYCOLLECTION(POINT (3 4), POLYGON((1 1, 1 3, 2 1, 1 1)))', 0);
```

```

SET @ML = geometry::Parse('MULTILINESTRING((0 8, 1 6), (1 0, 4 5))');
SET @MP = geometry::STGeomFromText('MULTIPOINT((2 3), (7 8))', 0);

SELECT CAST(@GC1 AS VARCHAR(67)) AS 'GC1';
SELECT CAST(@GC2 AS VARCHAR(67)) AS 'GC2';
SELECT @GC1.STGeometryType() AS 'GeometryType';
SELECT CAST(@MP.STAsText() AS CHAR(26)) AS 'MP',
       CAST(@ML.STAsText() AS CHAR(41)) AS 'ML';

GO

```

Здесь локальным переменным присваиваются значения различных объектов, а затем отображаются эти значения и некоторые характеристики созданных объектов. Собственно говоря, здесь отображается только одна характеристика — `GeometryType`. Результатом выполнения пакета будет:

```

GC1
-----
GEOMETRYCOLLECTION (POINT (3 4), POLYGON ((1 1, 1 3, 2 1, 1 1)))

GC2
-----
GEOMETRYCOLLECTION (POINT (3 4), POLYGON ((1 1, 1 3, 2 1, 1 1)))

GeometryType
-----
GeometryCollection

MP          ML
-----
MULTIPOINT ((2 3), (7 8)) MULTILINESTRING ((0 8, 1 6), (1 0, 4 5))

```

4.7.2. Тип данных **GEOGRAPHY**

Тип данных `GEOGRAPHY` позволяет описывать объекты на земной поверхности. Тип данных `GEOGRAPHY`, как и `GEOMETRY`, позволяет хранить данные о тех же семи типах объектов: `Point`, `MultiPoint`, `LineString`, `MultiLineString`, `Polygon`, `MultiPolygon` и `GeometryCollection`. С этими объектами можно использовать методы, похожие на методы (совпадающие по именам), что и применяемые для объектов типа данных `GEOMETRY`. Только эти методы принадлежат пространству имен `geography`.

Вот некоторые методы, используемые при работе с типом данных `GEOGRAPHY`:

- ◆ `STArea()` — возвращает общую площадь поверхности экземпляра;
- ◆ `STAsBinary()` — возвращает текст в формате WKB экземпляра;
- ◆ `STAsText()` — возвращает текст в формате WKT экземпляра;
- ◆ `STDifference()` — выполняет теоретико-множественное вычитание элементов (точек) двух множеств (двух земных поверхностей). То есть результат (экземпляр `GEOGRAPHY`) будет содержать все точки первого объекта, которые не принадлежат второму объекту;

- ◆ `STDisjoint()` — возвращает значение 1, если один экземпляр не имеет пространственного перекрытия с другим экземпляром. В противном случае возвращается значение 0;
- ◆ `STDistance()` — наименьшее расстояние между двумя объектами;
- ◆ `STEndpoint()` — конечная точка экземпляра;
- ◆ `STEquals()` — сравнение экземпляров; возвращает 1, если экземпляры равны, иначе 0;
- ◆ `STGeometryType()` — возвращает имя географического типа;
- ◆ `STIntersection()` — выполняет теоретико-множественное пересечение элементов (точек) двух множеств (двух земных поверхностей). Результат (экземпляр `GEOGRAPHY`) будет содержать все точки первого объекта, которые также принадлежат второму объекту;
- ◆ `STIntersects()` — возвращает 1, если экземпляры пересекаются, т. е. имеют общие точки, хотя бы одну общую точку, иначе 0.

Характеристиками объектов этого типа данных являются широта, долгота, уровень, мера и идентификатор пространственной ссылки SRID.

Географические координаты (широта и долгота) в географических науках определяются в градусах, минутах и секундах. В литературе эту систему еще называют DMS (Degree, Minute, Second — градус, минута, секунда). Для использования координат в SQL Server необходимо перевести принятую систему координат в десятичное представление градусов. То есть вместо указания в координате минут и секунд задается дробное значение градуса координаты. Для перевода значения из DMS в десятичное значение используется простая формула:

$$\begin{aligned} \text{Десятичное представление координаты} &= \\ &= \text{градусы} + (\text{минуты} / 60) + (\text{секунды} / 3600) \end{aligned}$$

Поскольку координаты задаются с использованием типа данных `FLOAT`, максимальное количество знаков в десятичном представлении не может превышать 15.

Для Саратова координаты: $51^{\circ} 32' 26''$ северной широты, $46^{\circ} 00' 31''$ восточной долготы. При переводе в десятичное значение: широта будет 51.5405555555556, долгота 46.0086111111111.

Москва: $55^{\circ} 45' 08''$ северной широты, $37^{\circ} 36' 56''$ восточной долготы. Десятичное значение: широта 55.7522222222222, долгота 37.6155555555556.

Земля является сплющенным сферионом, причем очень неправильной формы. Для описания характеристик этого трехмерного геометрического объекта (и описания приближенного) используются такие две характеристики, как размер большой полуоси, расстояние от центра Земли до экватора, а также размер малой полуоси, расстояние от центра Земли до полюса. В географических объектах в программировании для описания Земли используются размер большой полуоси и так называемое *улучшенное инвертированное отношение*, которое вычисляется по формуле:

$$\begin{aligned} \text{Улучшенное инвертированное отношение} &= \\ &= \text{большая полуось} / (\text{большая полуось} - \text{малая полуось}) \end{aligned}$$

Идентификатор пространственной ссылки SRID определяет конкретную систему пространственных ссылок, чьи характеристики используются при описании элемента данных. Эта система также определяет возможность взаимодействия конкретного элемента данных `GEOGRAPHY` с другими пространственными элементами данных того же типа. Для пространственных ссылок существует стандарт `EPSG`. SQL Server поддерживает большое количество различных пространственных ссылок. Их около 400. Они отличаются точностью представления координат и расстояний между объектами в различных областях земной поверхности. Список пространственных ссылок можно получить при обращении в операторе `SELECT` к системному представлению отображения каталогов `sys.spatial_reference_systems`.

Чтобы отобразить отдельные столбцы из этого набора записей, в Management Studio выполните оператор `SELECT`, как это показано в примере 4.23.

Пример 4.23. Отображение списка пространственных ссылок

```
USE master;
GO
SELECT spatial_reference_id,
       well_known_text,
       unit_of_measure,
       unit_conversion_factor
FROM sys.spatial_reference_systems;
GO
```

Значения столбцов, указанных в примере, показано в табл. 4.12.

Таблица 4.12. Некоторые столбцы системного представления отображения каталогов `sys.spatial_reference_systems`

Столбец	Содержание
<code>spatial_reference_id</code>	Идентификатор системы координат
<code>well_known_text</code>	Параметры системы координат в формате WKT (well-known text)
<code>unit_of_measure</code>	Название единицы измерения, используемой в системе координат для задания расстояний. В большинстве систем используется метр
<code>unit_conversion_factor</code>	Коэффициент для перевода используемых единиц измерения в метры. Чаще всего это единица

Список достаточно большой. Нас будут интересовать строки с двумя идентификаторами системы координат: 4200 и 4326.

Для России больше всего подходит система координат с идентификатором 4200. Описание идентификатора представлено столбцом `well_known_text`, который содержит описание в формате WKT. Текст не слишком наглядный. Разобъем его на несколько строк, выполнив необходимые отступы.

```
GEOGCS
  ["Pulkovo 1995",
   DATUM
     ["Pulkovo 1995",
      ELLIPSOID
        ["Krassowsky 1940",
         6378245,
         298.3
        ]
     ],
   PRIMEM["Greenwich", 0],
   UNIT["Degree", 0.0174532925199433]
]
```

В самом начале указывается, что идентификатор относится к географической системе: GEOGCS. Затем указывается название: "Pulkovo 1995". После ключевого слова DATUM указываются параметры. Текст "Pulkovo 1995" задает название. После ключевого слова ELLIPSOID указывается имя используемого эллипсоида: "Krassowsky 1940".

Затем даны числовые характеристики используемого эллипсоида. Число 6378245 определяет размер большой полуоси в метрах. Число 298.3 — величину инвертированного отношения.

После ключевого слова PRIMEM указывается начальная точка отсчета, меридиан, от которого считается долгота географических объектов. Здесь это ["Greenwich", 0], т. е. за точку отсчета выбирается Гринвичский меридиан.

За ключевым словом UNIT следует указание единиц, используемых для измерения углов широты и долготы. Здесь задано "Degree", т. е. единицей измерения является градус. Следом идет коэффициент, используемый для перевода радиан в градусы. Это число 0.0174532925199433 или $\pi / 180$.

По умолчанию в SQL Server для географических объектов используется идентификатор системы координат 4326. Его имя "WGS 84". Описание в формате WKT:

```
GEOGCS
  ["WGS 84",
   DATUM
     ["World Geodetic System 1984",
      ELLIPSOID[
        "WGS 84",
        6378137,
        298.257223563
      ]
     ],
   PRIMEM["Greenwich", 0],
   UNIT["Degree", 0.0174532925199433]
]
```

Основное отличие "WGS 84" от "Pulkovo 1995" в значении используемых величин и коэффициентов.

В примере 4.24 представлен пакет, позволяющий определить кратчайшее расстояние между двумя городами России — Москвой и Саратовом.

Пример 4.24. Определение расстояния между Москвой и Саратовом

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @Moscow AS GEOGRAPHY =
    geography::Point(55.75222222222222, 37.6155555555556, 4200);
DECLARE @Saratov AS GEOGRAPHY =
    geography::Point(51.5405555555556, 46.0086111111111, 4200);
SELECT @Moscow.Lat AS 'Широта Москвы',
    @Moscow.Long AS 'Долгота Москвы';
SELECT @Saratov.Lat AS 'Широта Саратова',
    @Saratov.Long AS 'Долгота Саратова';
SELECT @Moscow.STDistance(@Saratov) AS 'Расстояние Москва – Саратов';
GO
```

Здесь объявляются две локальные переменные типа данных `GEOGRAPHY`: `@Moscow` и `@Saratov`. При помощи метода `Point()` им присваиваются значения точек на поверхности Земли с координатами Москвы и Саратова. В этом методе первый параметр задает широту, второй — долготу. Третьим параметром указывается идентификатор системы координат SRID. Мы задали 4200, т. е. идентификатор "Pulkovo 1995".

Следующие операторы `SELECT` отображают заданные характеристики и расстояние между городами. Широту объекта можно найти при обращении к свойству `Lat` (сокращение от `latitude` — широта), а долготу — к свойству `Long` (`longitude` — долгота).

Результатом выполнения пакета будет:

Широта Москвы	Долгота Москвы
55,752222222222	37,6155555555556

Широта Саратова	Долгота Саратова
51,5405555555556	46,0086111111111

Расстояние Москва – Саратов

725633,115488806

Все расстояния указываются в тех единицах измерения, которые заданы в SRID, а это в нашем случае метры. То есть расстояние между Москвой и Саратовом чуть больше 725 километров.

Вот еще один пример определения расстояния между двумя городами — Парижем и Берлином (пример 4.25). Здесь для представления координат городов используется идентификатор системы координат 4326.

Пример 4.25. Определение расстояния между Парижем и Берлином

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @Paris AS GEOGRAPHY = geography::Point(48.87, 2.33, 4326);
DECLARE @Berlin AS GEOGRAPHY = geography::Point(52.52, 13.4, 4326);
SELECT CAST(@Paris.Lat AS VARCHAR(15)) AS 'Paris Latitude',
       CAST(@Paris.Long AS VARCHAR(15)) AS 'Paris Longitude',
       CAST(@Berlin.Lat AS VARCHAR(15)) AS 'Berlin Latitude',
       CAST(@Berlin.Long AS VARCHAR(15)) AS 'Berlin Longitude';
SELECT @Paris.STDistance(@Berlin) AS 'Distance between Paris and Berlin';
GO
```

Результат:

Paris Latitude	Paris Longitude	Berlin Latitude	Berlin Longitude
48.87	2.33	52.52	13.4

Distance between Paris and Berlin

879989,866996421

Для того чтобы координаты обоих городов при отображении уместились в одной строке, в операторе `SELECT` была использована функция преобразования `CAST()`.

На этом приостановим рассмотрение географического типа данных и его объектов.

4.8. Другие типы данных

К другим типам данных по традиции отнесены типы данных `SQL_VARIANT`, `TIMESTAMP`, `UNIQUEIDENTIFIER`, `HIERARCHYID`, `CURSOR`, `TABLE`, `XML`.

В документации говорится, что тип данных `TIMESTAMP` будет удален в следующих версиях системы. Основным назначением этого типа данных было отслеживание версий строк таблицы (добавление или изменение значений строки). Подобные изменения можно выполнить другими способами. Любопытно, что смысл типа данных `TIMESTAMP` совсем не тот, что определено стандартами SQL.

4.8.1. Тип данных `SQL_VARIANT`

Тип данных `SQL_VARIANT` является типом данных, который может хранить данные различных типов. Это могут быть целочисленные, строковые типы данных, про-

странственные, тип данных XML. Похожие типы данных существуют во многих скриптовых языках. Вот типы данных, которые могут храниться в объекте с типом данных SQL_VARIANT:

- ◆ строки CHAR, VARCHAR, NCHAR, VARCHAR (MAX), NVARCHAR (MAX);
- ◆ числовые типы данных INT, DEC, FLOAT;
- ◆ IMAGE, VARBINARY (MAX);
- ◆ XML;
- ◆ GEOGRAPHY, GEOMETRY;
- ◆ HIERARCHYID;
- ◆ и даже сам SQL_VARIANT.

В столбце таблицы или в переменной с этим типом данных помимо самих данных хранятся также и метаданные — данные, описывающие характеристики хранимых данных. Эти характеристики данных можно отобразить при использовании функции SQL_VARIANT_PROPERTY(). Функции передаются два параметра: имя переменной или столбца таблицы и параметр, задающий вид отображаемых метаданных. Следующие параметры заключаются в апострофы:

- ◆ BaseType — тип данных, хранящихся в настоящий момент в переменной;
- ◆ Precision — количество знаков;
- ◆ Scale — количество знаков после десятичной точки для числовых типов данных. Для всех остальных содержит 0;
- ◆ TotalBytes — количество байтов, используемых для хранения данных и метаданных;
- ◆ Collation — порядок сортировки строковых данных. Для остальных типов данных возвращается значение NULL;
- ◆ MaxLength — максимальное количество байтов, необходимых для хранения собственно данных.

В примере 4.26 показаны некоторые варианты использования типа данных SQL_VARIANT. Здесь также отображаются и характеристики данных с использованием функции SQL_VARIANT_PROPERTY().

Пример 4.26. Использование типа данных SQL_VARIANT

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @D SQL_VARIANT;
SET @D = SYSDATETIMEOFFSET();
SELECT DATENAME(hour, CAST(@D AS DATETIME)) AS 'Hour',
       DATENAME(minute, CAST(@D AS DATETIME)) AS 'Minute',
       DATENAME(second, CAST(@D AS DATETIME)) AS 'Second';
```

```
SELECT CAST(SQL_VARIANT_PROPERTY(@D, 'BaseType') AS VARCHAR(15))
       AS 'Base Type',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Precision') AS VARCHAR(11))
       AS 'Precision',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Scale') AS VARCHAR(11)) AS 'Scale',
       CAST(SQL_VARIANT_PROPERTY(@D, 'TotalBytes') AS VARCHAR(11))
       AS 'TotalBytes',
       CAST(SQL_VARIANT_PROPERTY(@D, 'MaxLength') AS VARCHAR(11))
       AS 'MaxLength',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Collation') AS VARCHAR(24))
       AS 'Collation'

SET @D = 'We always kill the one we love';
SELECT CAST(@D AS VARCHAR(30)) AS 'String';
SELECT CAST(SQL_VARIANT_PROPERTY(@D, 'BaseType') AS VARCHAR(15))
       AS 'Base Type',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Precision') AS VARCHAR(11))
       AS 'Precision',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Scale') AS VARCHAR(11)) AS 'Scale',
       CAST(SQL_VARIANT_PROPERTY(@D, 'TotalBytes') AS VARCHAR(11))
       AS 'TotalBytes',
       CAST(SQL_VARIANT_PROPERTY(@D, 'MaxLength') AS VARCHAR(11))
       AS 'MaxLength',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Collation') AS VARCHAR(24))
       AS 'Collation'

SET @D = 1024;
SELECT CAST(@D AS INTEGER) AS 'Integer';
SELECT CAST(SQL_VARIANT_PROPERTY(@D, 'BaseType') AS VARCHAR(15))
       AS 'Base Type',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Precision') AS VARCHAR(11))
       AS 'Precision',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Scale') AS VARCHAR(11)) AS 'Scale',
       CAST(SQL_VARIANT_PROPERTY(@D, 'TotalBytes') AS VARCHAR(11))
       AS 'TotalBytes',
       CAST(SQL_VARIANT_PROPERTY(@D, 'MaxLength') AS VARCHAR(11)) AS
'MaxLength',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Collation') AS VARCHAR(24)) AS
'Collation'

SET @D = 2.873;
SELECT CAST(@D AS DECIMAL(4, 3)) AS 'Decimal';
SELECT CAST(SQL_VARIANT_PROPERTY(@D, 'BaseType') AS VARCHAR(15))
       AS 'Base Type',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Precision') AS VARCHAR(11))
       AS 'Precision',
       CAST(SQL_VARIANT_PROPERTY(@D, 'Scale') AS VARCHAR(11)) AS 'Scale',
```

```

CAST(SQL_VARIANT_PROPERTY(@D, 'TotalBytes') AS VARCHAR(11))
AS 'TotalBytes',
CAST(SQL_VARIANT_PROPERTY(@D, 'MaxLength') AS VARCHAR(11))
AS 'MaxLength',
CAST(SQL_VARIANT_PROPERTY(@D, 'Collation') AS VARCHAR(24))
AS 'Collation'

SET @D = 2.873E-6;
SELECT CAST(@D AS FLOAT) AS 'Float';
SELECT CAST(SQL_VARIANT_PROPERTY(@D, 'BaseType') AS VARCHAR(15))
AS 'Base Type',
CAST(SQL_VARIANT_PROPERTY(@D, 'Precision') AS VARCHAR(11))
AS 'Precision',
CAST(SQL_VARIANT_PROPERTY(@D, 'Scale') AS VARCHAR(11)) AS 'Scale',
CAST(SQL_VARIANT_PROPERTY(@D, 'TotalBytes') AS VARCHAR(11))
AS 'TotalBytes',
CAST(SQL_VARIANT_PROPERTY(@D, 'MaxLength') AS VARCHAR(11))
AS 'MaxLength',
CAST(SQL_VARIANT_PROPERTY(@D, 'Collation') AS VARCHAR(24))
AS 'Collation';

```

GO

Результат:

Hour	Minute	Second
15	30	42

Base Type	Precision	Scale	TotalBytes	MaxLength	Collation
datetimeoffset	34	7	13	10	NULL

String

```
We always kill the one we love
```

Base Type	Precision	Scale	TotalBytes	MaxLength	Collation
varchar	0	0	38	30	Cyrillic_General_CI_AS

Integer

```
1024
```

Base Type	Precision	Scale	TotalBytes	MaxLength	Collation
int	10	0	6	4	NULL

Decimal

2.873

Base Type	Precision	Scale	TotalBytes	MaxLength	Collation
-----------	-----------	-------	------------	-----------	-----------

numeric	4	3	9	5	NULL
---------	---	---	---	---	------

Float

2,873E-06

Base Type	Precision	Scale	TotalBytes	MaxLength	Collation
-----------	-----------	-------	------------	-----------	-----------

float	53	0	10	8	NULL
-------	----	---	----	---	------

Здесь одной и той же локальной переменной типа данных SQL_VARIANT в одном скрипте поочередно присваиваются значения типов данных DATETIMEOFFSET, VARCHAR (или CHAR), INTEGER, DECIMAL и FLOAT. При отображении значения SQL_VARIANT в операторе SELECT требуется явное преобразование переменной к соответствующему типу данных с помощью функции CAST().

Для каждого типа данных при помощи функции SQL_VARIANT_PROPERTY() отображаются и его характеристики. Просмотрите их для интересующих вас типов данных.

4.8.2. Тип данных *HIERARCHYID*

Тип данных HIERARCHYID используется для представления иерархических данных, т. е. для представления данных в иерархической, древовидной структуре. Здесь можно описывать иерархические отношения между данными в виде отношения родитель-потомок. Иерархия является весьма естественным способом организации данных, описывающих структуру различных предметных областей человеческой деятельности. Например, любая организация по сути своей является иерархической. Файловая система любой, достаточно развитой операционной системы, также иерархична. Административно-территориальное деление стран нашего мира тоже имеет иерархическую структуру. Связь между ключевыми словами (дескрипторами), которые описывают характеристики хранимых сведений о технической литературе, является древовидной, иерархической.

Тип данных HIERARCHYID появился только в SQL Server версии 2008. Для работы с данными такого типа существует ряд полезных функций, выполняющих создание объектов, поиска по уровням иерархии и др.

ЗАМЕЧАНИЕ

Для работы с типами данных GEOGRAPHY, GEOMETRY и HIERARCHYID существует бесплатная библиотека, которая содержит немало полезных функций. Библиотеку можно скачать из состава Feature Pack.

Данные `HIERARCHYID` имеют переменную длину и хранятся очень компактно. Максимальный размер поля этого типа данных 892 байта. По иерархическим данным можно создавать индексы.

При работе с этим типом данных используются перечисленные далее методы.

- ◆ `GetAncestor(<номер уровня>)` — возвращает предка текущего узла в иерархии, уровень которого на указанное значение меньше узла текущего уровня. Параметр "номер уровня" должен быть целым неотрицательным числом. Если указать 0, то будет возвращен сам текущий узел.
- ◆ `GetDescendant(<дочерний узел 1>, <дочерний узел 2>)` — возвращает дочерний узел текущего узла. Что именно возвращается, зависит от значения двух передаваемых методу параметров. Несмотря на простой синтаксис этого метода, поведение его достаточно сложное, но логичное. Оно зависит от наличия значений `NULL` у передаваемых параметров и от положения в иерархии двух узлов, переданных в качестве параметров:
 - если оба параметра имеют значение `NULL`, то метод возвращает идентификатор первого элемента следующего уровня;
 - если значение `NULL` имеет второй параметр, то метод вернет идентификатор следующего после (находящегося правее) дочернего узла 1 элемента иерархии того же уровня;
 - если только первый параметр имеет значение `NULL`, то возвращается идентификатор элемента, находящегося левее второго параметра;
 - если указаны оба параметра одного уровня, имеющие непустое значение, то возвращается идентификатор элемента, расположенного между заданными элементами иерархии. Причем первый элемент должен быть "меньше" второго, т. е. располагаться левее в иерархической структуре. Если дважды выполнить этот метод с одними и теми же значениями параметров, то оба раза будут получены одинаковые результаты. Оба параметра должны быть элементами иерархии одного и того же уровня.

Простой пример использования этого метода мы рассмотрим чуть позже.

- ◆ `GetLevel()` — возвращает номер уровня (или еще говорят "глубину") узла в иерархии. Уровни в иерархии нумеруются, начиная с нуля.
- ◆ `GetRoot()` — возвращает корневой узел иерархии. Тип возвращаемого данного — `HIERARCHYID`.
- ◆ `Parse(<исходная строка>)` — этот метод функционально, т. е. по исходным данным и по получаемым результатам, полностью соответствует выполнению метода `CAST(<исходная строка> AS HIERARCHYID)`. Любопытно, что исходная строка не должна содержать конечных пробелов. В этом случае будет получено сообщение об ошибке. Параметр `<исходная строка>` должен содержать правильную, каноническую строку, описывающую положение элемента в иерархической структуре. Вид правильных строк мы с вами рассмотрим при выполнении нашего примера работы с иерархическими типами данных.

- ◆ GetReparentedValue(<узел 1>, <узел 2>) — заменяет путь <узла 1> на путь (значение) <узла 2>. Пример использования этого метода мы также вскоре рассмотрим.
- ◆ ToString() — применительно к типу данных HIERARCHYID возвращает строковое ("каноническое") представление иерархического данного.

В документах, имеющих отношение к типу данных HIERARCHYID, вы можете встретить еще упоминание функций Read() и Write(). В Transact-SQL эти функции не поддерживаются. Вместо них вы можете с тем же успехом использовать уже хорошо известную функцию CAST().

В примере 4.27 показано использование некоторых функций для иерархического типа данных HIERARCHYID.

Пример 4.27. Использование типа данных HIERARCHYID

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @H1 AS HIERARCHYID;
DECLARE @H21 AS HIERARCHYID;
DECLARE @H22 AS HIERARCHYID;
DECLARE @H23 AS HIERARCHYID;
DECLARE @H24 AS HIERARCHYID;
DECLARE @H25 AS HIERARCHYID;
SET @H1 = hierarchyid::GetRoot();
SET @H21 = @H1.GetDescendant(NULL, NULL);
SET @H22 = @H1.GetDescendant(@H21, NULL);
SET @H23 = @H1.GetDescendant(@H22, NULL);
SET @H24 = @H1.GetDescendant(@H23, NULL);
SET @H25 = @H1.GetDescendant(@H23, @H24);

SELECT @H1.GetLevel() AS 'H1 Level',
       CAST(@H1.ToString() AS VARCHAR(12)) AS 'H1 ToString',
       CAST(@H1 AS VARBINARY(10)) AS 'H1 VARBINARY';
SELECT @H21.GetLevel() AS 'H21 Level',
       CAST(@H21.ToString() AS VARCHAR(12)) AS 'H21 ToString',
       CAST(@H21 AS VARBINARY(6)) AS 'H21 VARBINARY',
       CAST(@H21.GetAncestor(1) AS VARBINARY(6)) AS 'H21 Ancestor Bin',
       CAST(@H21.GetAncestor(1) AS VARCHAR(18)) AS 'H21 Ancestor Char';
SELECT @H22.GetLevel() AS 'H22 Level',
       CAST(@H22.ToString() AS VARCHAR(12)) AS 'H22 ToString',
       CAST(@H22 AS VARBINARY(6)) AS 'H22 VARBINARY',
       CAST(@H22.GetAncestor(1) AS VARBINARY(6)) AS 'H22 Ancestor Bin',
       CAST(@H22.GetAncestor(1) AS VARCHAR(18)) AS 'H22 Ancestor Char';
SELECT @H23.GetLevel() AS 'H23 Level',
       CAST(@H23.ToString() AS VARCHAR(12)) AS 'H23 ToString',
```

```

CAST(@H23 AS VARBINARY(6)) AS 'H22 VARBINARY',
CAST(@H23.GetAncestor(1) AS VARBINARY(6)) AS 'H23 Ancestor Bin',
CAST(@H23.GetAncestor(1) AS VARCHAR(18)) AS 'H23 Ancestor Char';
SELECT @H24.GetLevel() AS 'H24 Level',
CAST(@H24.ToString() AS VARCHAR(12)) AS 'H24 ToString',
CAST(@H24 AS VARBINARY(6)) AS 'H24 VARBINARY',
CAST(@H24.GetAncestor(1) AS VARBINARY(6)) AS 'H24 Ancestor Bin',
CAST(@H24.GetAncestor(1) AS VARCHAR(18)) AS 'H24 Ancestor Char';
SELECT @H25.GetLevel() AS 'H25 Level',
CAST(@H25.ToString() AS VARCHAR(12)) AS 'H25 ToString',
CAST(@H25 AS VARBINARY(6)) AS 'H25 VARBINARY',
CAST(@H25.GetAncestor(1) AS VARBINARY(6)) AS 'H25 Ancestor Bin',
CAST(@H25.GetAncestor(1) AS VARCHAR(18)) AS 'H25 Ancestor Char';
SELECT CAST(@H25.ToString() AS VARCHAR(10)) AS 'Old H25',
CAST(@H24.ToString() AS VARCHAR(10)) AS 'H24',
CAST(@H25.GetReparentedValue(@H25, @H24) AS VARCHAR(10))
AS 'New H25';
GO

```

Результат выполнения этого пакета:

	H1 Level	H1 ToString	H1 VARBINARY	H21 Level	H21 ToString	H21 VARBINARY	H21 Ancestor Bin	H21 Ancestor Char
0	/		0x	1	/1/	0x58	0x	/
	H22 Level	H22 ToString	H22 VARBINARY	H22 Ancestor Bin	H22 Ancestor Char			
1	/2/		0x68	0x	/			
	H23 Level	H23 ToString	H23 VARBINARY	H23 Ancestor Bin	H23 Ancestor Char			
1	/3/		0x78	0x	/			
	H24 Level	H24 ToString	H24 VARBINARY	H24 Ancestor Bin	H24 Ancestor Char			
1	/4/		0x84	0x	/			
	H25 Level	H25 ToString	H25 VARBINARY	H25 Ancestor Bin	H25 Ancestor Char			
1	/3.1/		0x8160	0x	/			
	Old H25	H24	New H25					
	/3.1/	/4/	/4/					

Здесь объявляются полдюжины локальных переменных типа данных HIERARCHYID. Мы создаем простую двухуровневую иерархию.

Переменной @h1 присваивается значение корневого узла при использовании метода GetRoot(). При отображении в первом операторе SELECT значения этой переменной видим, что она имеет уровень 0, текстовое (каноническое) ее представление /, в двоичном виде она имеет значение 0x.

Далее остальным локальным переменным присваиваются значения второго уровня (в терминологии SQL Server — уровня 1). Присваивание осуществляется при обращении к методу GetDescendant(). Первое обращение к методу:

```
SET @H21 = @H1.GetDescendant(NULL, NULL);
```

Оба параметра, передаваемые методу, имеют значение NULL, следовательно, здесь создается первый элемент следующего уровня. Его каноническое представление будет таким: /1/.

Для переменных @h22, @h23, и @h24 создаются последующие элементы текущего уровня:

```
SET @H22 = @H1.GetDescendant(@H21, NULL);
SET @H23 = @H1.GetDescendant(@H22, NULL);
SET @H24 = @H1.GetDescendant(@H23, NULL);
```

Второй параметр, передаваемый методу, имеет значение NULL. Эти переменные в результате будут иметь канонические представления (строковый вид) /2/, /3/, /4/.

Переменной @h25 устанавливается значение, которое должно находиться между элементами @h23 и @h24:

```
SET @H25 = @H1.GetDescendant(@H23, @H24);
```

Этим значением (строковым) будет /3.1/.

Созданная этими операторами иерархия показана на рис. 4.4.

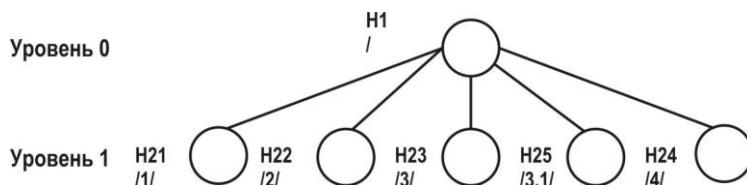


Рис. 4.4. Созданная иерархическая структура

На рисунке для каждого элемента иерархии указано имя локальной переменной (без начального символа @) и каноническое описание элемента.

В последнем операторе SELECT этого пакета демонстрируется использование метода GetReparentedValue() для размещения элемента иерархии между двумя существующими:

```
SELECT CAST(@H25.ToString() AS VARCHAR(10)) AS 'Old H25',
       CAST(@H24.ToString() AS VARCHAR(10)) AS 'H24',
       CAST(@H25.GetReparentedValue(@H25, @H24) AS VARCHAR(10))
              AS 'New H25';
```

Вначале отображаются значения переменных @H25 и @H24. После этого осуществляется изменение значения переменной @H25 на значение, получаемое из переменной @H24. Новое значение тоже выводится в этой строке. Результатом будет:

Old H25	H24	New H25
/3.1/	/4/	/4/

Мы видим, что переменная @H25 получила то же значение, что и переменная @H24.

ЗАМЕЧАНИЕ

В Интернете вы можете найти многочисленные обсуждения этого типа данных. Кто-то с жаром уверяет, что для представления иерархических структур в базах данных не существует ничего лучшего. Другие же считают, что классические способы хранения таких данных более удобны и позволяют получить более высокую производительность системы. Включаясь в это обсуждение, хочу сказать, что лично я предпочитаю так называемые "классические" методы работы с иерархией. Важным недостатком с моей точки зрения типа данных `HIERARCHYID` является фиксированное количество уровней иерархии, которые можно представить в этом типе данных. В моей же практике часто встречаются такие иерархические структуры, где нельзя точно определить корень дерева. Пример подобной структуры мы рассмотрим с вами, когда будем обсуждать вопросы создания таблиц, описывающих людей в их реальной жизни, их родственные связи и средства работы с такими таблицами.

4.8.3. Тип данных `UNIQUEIDENTIFIER`

Тип данных `UNIQUEIDENTIFIER` позволяет получать и хранить в переменной или в столбце таблицы уникальные значения. Его размер 16 байтов. Значения в этом типе данных являются уникальным глобальным идентификатором (Globally Unique Identifier — *GUID*). *GUID* обеспечивает действительно глобальную уникальность данных. Во всем мире (во всей Вселенной) на разных компьютерах никогда не будет получено двух одинаковых значений (на самом деле, некоторые специалисты подсчитали, к какому году будет исчерпан объем разных значений).

Присвоить значение такой переменной в обычном скрипте можно при помощи функции `NEWID()` или с использованием шестнадцатеричной константы, которая может быть задана в символьном или двоичном формате. Формат символьной константы следующий:

```
'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'
```

Каждый элемент `x` в константе является шестнадцатеричным числом — имеет значение от 0 до 9 или от `a` до `f`. Буквы можно задавать как прописные, так и строчные. Для задания константы в двоичном виде нужно ввести, например, следующие шестнадцатеричные цифры:

```
0xff19966f868b11d0b42d00c04fc964ff
```

Для столбцов таблицы с типом данных `UNIQUEIDENTIFIER` в качестве значения по умолчанию (предложение `DEFAULT` в описании столбца) может быть использована функция `NEWSEQUENTIALID()`. Эта функция позволяет создать значение, превышающее любое значение уникального идентификатора, которое было создано на этом компьютере. То есть, значения, полученные при помощи этой функции, будут уникальными на данном компьютере. Недостатком этой функции считается то, что получаемое при ее помощи значение можно предугадать, что отрицательно сказывается на решении вопросов конфиденциальности.

Функция `NEWID()` возвращает глобальное уникальное значение для типа данных `UNIQUEIDENTIFIER`.

В примере 4.28 показаны два способа присваивания значения переменной с типом данных `UNIQUEIDENTIFIER`.

Пример 4.28. Использование типа данных `UNIQUEIDENTIFIER`

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @D UNIQUEIDENTIFIER;
SET @D = NEWID();
SELECT CAST(@D AS VARCHAR(36)) AS 'UNIQUEIDENTIFIER 1';
SET @D = 'AE71E0D7-CE6E-49A7-959A-02D4139AA2D1';
SELECT CAST(@D AS VARCHAR(36)) AS 'UNIQUEIDENTIFIER 2';
SET @D = 0xff19966f868b11d0b42d00c04fc964ff;
SELECT CAST(@D AS VARCHAR(36)) AS 'UNIQUEIDENTIFIER 3';
GO
```

Результат:

UNIQUEIDENTIFIER 1

AB7D1EA8-6078-49DD-8E1B-DA2E54D8FD93

UNIQUEIDENTIFIER 2

AE71E0D7-CE6E-49A7-959A-02D4139AA2D1

UNIQUEIDENTIFIER 3

6F9619FF-8B86-D011-B42D-00C04FC964FF

Если переменная не инициализирована, то она будет иметь, естественно, значение `NULL`. Продолжение исследования типа данных `UNIQUEIDENTIFIER` см. в следующем примере, рассматриваемом через пару страниц.

4.8.4. Тип данных CURSOR

Тип данных CURSOR хранит указатель на набор данных — т. е. на множество строк, полученных из таблицы (таблиц) базы данных в результате выполнения запроса к базе данных. Этот тип данных нельзя использовать при описании столбцов таблиц. Интересно, что само слово "cursor", по крайней мере в этом контексте, было образовано из выражения CURrent Set Of Records (текущий набор записей).

Не следует путать этот термин с тем курсором, который связан с текущим положением указателя мыши на экране или с текущим символом в текстовом окне.

Давайте в этом подразделе рассмотрим основные возможности и языковые средства, связанные с курсорами, чтобы потом к ним уже не возвращаться в этой книге. Сейчас мы забежим несколько вперед в нашем исследовании возможностей SQL Server, а именно в отношении поиска данных. Нам нужно будет рассмотреть возможности и средства, используемые для выборки данных из таблиц базы данных. Если тема курсоров вам интересна, то рекомендую чуть позже, когда мы подробнейшим образом рассмотрим средства получения данных из базы, вернуться к этому подразделу. Здесь мы с вами сможем увидеть основные возможности объявления и использования курсоров для просмотра данных из базы данных.

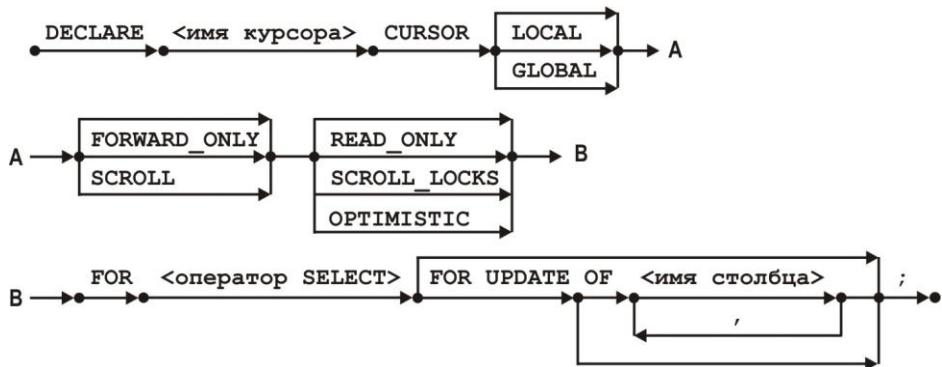
Прежде всего, нам с вами нужно избежать возможной путаницы в терминологии. Мы будем рассматривать две различные сущности, которые называются одним термином *курсор*. Во-первых, курсор — это объект SQL Server, который позволяет выполнять некоторые действия с данными из базы данных. Он создается при помощи оператора `DECLARE CURSOR`. Во-вторых, курсор (`CURSOR`) — это тип данных, который может назначаться локальной переменной, которая будет ссылаться, указывать на объект курсор. Для объявления такой переменной используется обычный оператор `DECLARE`, определяющий локальную переменную типа данных `CURSOR`.

Для объявления курсора как объекта используется оператор `DECLARE CURSOR`. Несколько сокращенный вариант синтаксиса объявления курсора представлен в листинге 4.3 и в соответствующем R-графе (граф 4.2). Следует отметить, что синтаксис этого оператора в SQL Server существенно расширен по сравнению с тем, что предлагает нам международный стандарт SQL. По крайней мере, это относится к последним версиям сервера.

Листинг 4.3. Синтаксис оператора `DECLARE CURSOR`

```
DECLARE <имя курсора> CURSOR [ LOCAL | GLOBAL ]
    [ FORWARD_ONLY | SCROLL ]
    [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
FOR <оператор SELECT>
    [ FOR UPDATE [ OF <имя столбца> [, <имя столбца>]... ] ];
```

Имя курсора, как объекта, в отличие от имен локальных переменных, не должно начинаться с символа @.



Граф 4.2. Синтаксис оператора DECLARE CURSOR

Можно объявить локальный (`LOCAL`) или глобальный (`GLOBAL`) курсор. Локальный курсор известен лишь в том скрипте (пакете), триггере или хранимой процедуре, где он был объявлен. Вариант `GLOBAL` означает, что курсор известен в любом пакете, триггере или хранимой процедуре, которые выполняются в текущем соединении с сервером. Если вид курсора не указан, то ему устанавливается значение по умолчанию, которое зависит от текущих установок базы данных. Для этого используется опция `CURSOR_DEFAULT` предложения `SET` оператора изменения базы данных `ALTER DATABASE`.

Варианты `FORWARD_ONLY` и `SCROLL` указывают, допустим ли при использовании курсора просмотр записей набора данных только вперед (`FORWARD_ONLY`) или возможен возврат и к предыдущим записям (`SCROLL`), т. е. и в обратном порядке. При задании `SCROLL` возможны все варианты навигации по набору данных в операторе `FETCH` — см. синтаксис этого оператора далее в листинге 4.4.

Вариант `READ_ONLY` определяет, что набор данных является набором только для чтения, т. е. никакие изменения в полученных при использовании курсора данных невозможны. Нельзя будет ни изменять значения данных, ни удалять строки. В случае задания `SCROLL_LOCKS` можно выполнять изменение данных соответствующего набора данных за счет блокирования от изменения другими процессами считываемых в набор данных записей. В варианте `OPTIMISTIC` запрещается изменение набора данных, если исходные данные были изменены другими процессами с момента их считывания в набор данных курсора.

В обязательном предложении `FOR` (в первом предложении `FOR`) задается оператор `SELECT`, который и определяет набор данных, с которым будет осуществляться деятельность через создаваемый курсор. Об операторе `SELECT` мы будем очень много говорить в одной из следующих глав. Сейчас же рассмотрим простой пример, в котором будет использована уже известная нам форма этого оператора.

Во втором предложении `FOR` (в предложении `FOR UPDATE`) можно указать столбцы набора данных, для которых возможно выполнение изменений.

При работе с курсорами для навигации по набору данных, с которым связан курсор, используется оператор `FETCH`. Его синтаксис представлен в листинге 4.4 и в соответствующем R-графе (граф 4.3).

Листинг 4.4. Синтаксис оператора `FETCH`

```

FETCH [ NEXT
       | PRIOR
       | FIRST
       | LAST
       | ABSOLUTE <целое>
       | RELATIVE <целое> ]
FROM { <имя курсора> | <имя переменной> }
[ INTO <имя переменной> [, <имя переменной>]... ];

```



Граф 4.3. Синтаксис оператора `FETCH`

Обратите внимание, что в этом операторе (как впрочем и в других операторах работы с курсорами) для выполнения необходимого действия можно указать как сам объект курсор, так и локальную переменную типа данных `CURSOR`, ссылающуюся на реальный курсор. Различные варианты использования этого оператора мы прямо сейчас и рассмотрим.

Навигация по набору данных задается с помощью ключевых слов следующим образом:

- ◆ ключевое слово `NEXT` указывает, что должна быть считана следующая строка или, иными словами, указатель курсора должен быть перемещен на следующую строку;
- ◆ `PRIOR` задает перемещение указателя на предыдущую строку;
- ◆ `FIRST` — переход к первой строке в наборе данных;
- ◆ `LAST` — переход к последней строке в наборе данных;
- ◆ вариант `ABSOLUTE` задает переход к строке в наборе данных с указанным номером;
- ◆ при указании `RELATIVE` осуществляется перемещение на указанное количество строк относительно текущей строки. Если задано положительное число, указа-

тель курсора перемещается дальше по набору, если отрицательное — то в обратном направлении, ближе к началу.

Если ни одно из ключевых слов не задано, то по умолчанию предполагается NEXT (это ключевое слово подчеркнуто в описании синтаксиса).

В предложении FROM задается имя курсора или имя локальной переменной, ссылающейся на ранее определенный курсор.

Необязательное предложение INTO содержит список локальных переменных, в которые будут помещаться элементы выбранной строки набора данных.

Чтобы при помощи курсора был выполнен оператор SELECT, читающий данные в набор данных, необходимо открыть курсор с помощью оператора OPEN. В этом операторе также можно указать собственно курсор или локальную переменную типа данных CURSOR, ссылающуюся на объект курсор.

После завершения использования набора данных, полученного при помощи курсора, необходимо закрыть курсор (используемую локальную переменную) оператором CLOSE и освободить память, занимаемую курсором, выполнив оператор DEALLOCATE.

В следующем примере (пример 4.29) рассматривается вариант обращения к системному представлению каталогов sys.databases для отображения списка баз данных текущего экземпляра сервера и некоторых их характеристик, но уже при использовании курсора. Этот пример дублирует аналогичные действия и результат примера 3.2 из предыдущей главы. В примере используется объект курсор.

Пример 4.29. Использование курсора

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @NAME_U AS CHAR(20);
DECLARE @ID_U AS CHAR(4);
DECLARE @DATE_U AS DATE;
DECLARE @COLLATION_U AS CHAR(23);
DECLARE OurCursor CURSOR SCROLL FOR
    SELECT CAST(name AS CHAR(20)) AS 'NAME',
           CAST(database_id AS CHAR(4)) AS 'ID',
           create_date AS 'DATE',
           CAST(collation_name AS CHAR(23)) AS 'COLLATION'
    FROM sys.databases;
OPEN OurCursor;
FETCH NEXT FROM OurCursor INTO @NAME_U, @ID_U, @DATE_U, @COLLATION_U;
WHILE @@FETCH_STATUS = 0
BEGIN
```

```
SELECT @NAME_U AS 'NAME',
       @ID_U AS 'ID',
       @DATE_U AS 'DATE',
       @COLLATION_U AS 'COLLATION';
      FETCH NEXT FROM OurCursor INTO @NAME_U, @ID_U, @DATE_U, @COLLATION_U;
END
GO
CLOSE OurCursor;
DEALLOCATE OurCursor;
```

В этом скрипте объявляются четыре локальных переменных и один курсор `OurCursor` (наш курсор). Указывается, что курсор будет ссылаться на набор данных, полученный при помощи оператора `SELECT`, который выбирает данные (список баз данных) из системного представления каталогов `sys.databases`. При описании курсора указано ключевое слово `SCROLL`, это означает, что набор данных, полученный при помощи курсора, можно просматривать в любом направлении.

Чтобы были получены соответствующие данные (чтобы был выполнен заданный оператор `SELECT`), необходимо "открыть" курсор с использованием оператора `OPEN`. В процессе открытия курсора выполняется соответствующий оператор `SELECT` и в память загружается набор данных. После открытия курсор указывает на запись (не-существующую), предшествующую первой записи в полученном наборе данных. Для получения ссылки на следующую запись набора данных используется оператор `FETCH`. В необязательном выражении `INTO` этого оператора перечисляются имена локальных переменных, которым присваиваются значения выбранных столбцов из системного представления.

Первый после открытия курсора выполненный оператор `FETCH NEXT` позволяет получить первую строку набора данных. Здесь в нашем пакете из примера 4.29 можно в операторе не задавать ключевое слово `NEXT`, а указать, что оператор должен читать самую первую (`FIRST`) запись в наборе данных:

```
FETCH FIRST FROM OurCursor INTO @NAME_U, @ID_U, @DATE_U, @COLLATION_U;
```

Затем выполняется цикл просмотра полученных строк. Цикл осуществляется при помощи оператора `WHILE`. В операторе задается условие продолжения цикла. Здесь используется системная функция `@@FETCH_STATUS`. Она будет возвращать значение 0, пока не будет достигнут конец списка строк, полученных при открытии курсора. Точнее следует сказать, что эта функция будет возвращать значение 0, пока курсор ссылается на какую-либо существующую запись набора данных. Когда весь список будет просмотрен, функция вернет значение -1, и цикл будет завершен.

В теле цикла, заключенного в операторные скобки `BEGIN` и `END`, выполняется отображение данных текущей строки и осуществляется переход к следующей строке при выполнении оператора `FETCH NEXT`.

Результат может быть приблизительно следующим (если вы не добавляли много новых баз данных в систему, в текущий экземпляр сервера):

NAME	ID	DATE	COLLATION
<hr/>			
master	1	2003-04-08	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
tempdb	2	2011-11-03	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
model	3	2003-04-08	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
msdb	4	2011-06-24	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
ReportServer	5	2011-07-27	Latin1_General_CI_AS_KS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
ReportServerTempDB	6	2011-07-27	Latin1_General_CI_AS_KS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
SimpleDB	7	2011-08-29	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
BestDatabase	8	2011-09-05	NULL
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
Multy	9	2011-08-19	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
MultyGroup	10	2011-08-19	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
DBParam	11	2011-08-19	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
ContainedDatabase1	12	2011-08-24	Cyrillic_General_CI_AS

NAME	ID	DATE	COLLATION
<hr/>			
NewDatabase	13	2011-08-31	Cyrillic_General_CI_AS
<hr/>			
NAME	ID	DATE	COLLATION
<hr/>			
Contained1	14	2011-09-09	Cyrillic_General_CI_AS

Здесь мы просматривали список с начала до конца. Можно выполнить просмотр, начиная с последней записи, передвигаясь к началу списка. Для этого первый оператор `FETCH` следует изменить:

```
FETCH LAST FROM OurCursor INTO @NAME_U, @ID_U, @DATE_U, @COLLATION_U;
```

В этом варианте оператор `FETCH` переводит указатель курсора на последнюю запись набора данных.

Чтобы в теле цикла `WHILE` указатель курсора от последней строки перемещался к началу набора данных, нужно оператор `FETCH` записать в следующем виде:

```
FETCH PRIOR FROM OurCursor INTO @NAME_U, @ID_U, @DATE_U, @COLLATION_U;
```

В этом примере мы работали с самим объектом курсор. Теперь слегка изменим наш пакет, чтобы проиллюстрировать использование локальной переменной типа данных `CURSOR`. Внесите в скрипт следующие изменения, как показано в примере 4.30.

Пример 4.30. Использование локальной переменной типа данных CURSOR

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @NAME_U AS CHAR(20);
DECLARE @ID_U AS CHAR(4);
DECLARE @DATE_U AS DATE;
DECLARE @COLLATION_U AS CHAR(23);
DECLARE @Cursor_Pointer AS CURSOR;
DECLARE OurCursor CURSOR SCROLL FOR
    SELECT CAST(name AS CHAR(20)) AS 'NAME',
           CAST(database_id AS CHAR(4)) AS 'ID',
           create_date AS 'DATE',
           CAST(collation_name AS CHAR(23)) AS 'COLLATION'
      FROM sys.databases;

    SET @Cursor_Pointer = OurCursor;
    OPEN @Cursor_Pointer;

    FETCH NEXT FROM @Cursor_Pointer
        INTO @NAME_U, @ID_U, @DATE_U, @COLLATION_U;
```

```
WHILE @@Fetch_Status = 0
BEGIN
    SELECT @NAME_U AS 'NAME',
        @ID_U AS 'ID',
        @DATE_U AS 'DATE',
        @COLLATION_U AS 'COLLATION';
    FETCH NEXT FROM @Cursor_Pointer
        INTO @NAME_U, @ID_U, @DATE_U, @COLLATION_U;
END;

CLOSE @Cursor_Pointer;
DEALLOCATE @Cursor_Pointer;
```

Отличие этого примера от предыдущего только в том, что здесь объявляется локальная переменная `@Cursor_Pointer` типа данных CURSOR, и все дальнейшие действия выполняются именно с ней, а не с курсором. Сам объект CURSOR остался точно таким же, как и в предыдущем примере. Функциональность данного пакета осталась той же самой.

Для присваивания локальной переменной указателя на ранее объявленный курсор используется оператор SET:

```
SET @Cursor_Pointer = OurCursor;
```

Операторы OPEN, FETCH, CLOSE и DEALLOCATE выполняются теперь с локальной переменной, а не с самим курсором.

4.8.5. Тип данных **TABLE**

Тип данных TABLE позволяет временно хранить в табличном виде результаты обработки данных. Эти данные впоследствии могут быть прочитаны с целью дальнейшей обработки. Тип данных TABLE может быть использован в хранимых процедурах, в функциях и в обычных пакетах (скриптах). Его нельзя использовать при описании столбцов таблиц.

В примере 4.31 показано создание переменной типа данных TABLE. В эту переменную (временную таблицу) помещается несколько строк, которые затемчитываются оператором SELECT.

Пример 4.31. Использование типа данных TABLE

```
USE master;
GO
SET NOCOUNT ON;
DECLARE @T TABLE
(
    ID INTEGER NOT NULL IDENTITY PRIMARY KEY,
    UNIQUEIDENTIFIER_Again UNIQUEIDENTIFIER
        DEFAULT NEWSEQUENTIALID(),
    TXT CHAR(1)
);
```

```

INSERT INTO @T VALUES (NEWID(), 'A');
INSERT INTO @T VALUES (NEWID(), 'B');
INSERT INTO @T (TXT) VALUES ('C');
INSERT INTO @T (TXT) VALUES ('D');
-- Дальнейшая обработка данных
SELECT * FROM @T;
GO

```

Результат выборки данных (на моем компьютере, у вас это будет выглядеть иначе):

ID	UNIQUEIDENTIFIER AGAIN	TXT
1	A0E949B6-5878-45ED-AD0C-B4C0436EF7C8	A
2	B87CA43C-EA91-44A7-B86C-CEA58D371EC9	B
3	35F780C5-5E3F-DF11-89C7-001E3307EB08	C
4	36F780C5-5E3F-DF11-89C7-001E3307EB08	D

В примере объявляется локальная переменная типа TABLE. Эта таблица содержит три столбца. Первый столбец является первичным ключом типа данных INTEGER. Столбец описан с атрибутом IDENTITY. Это означает, что система при добавлении новой строки в таблицу будет автоматически формировать уникальное целочисленное значение для этого столбца. Такой ключ, как мы помним, называется автоинкрементным. В операторах INSERT, добавляющих новые строки в таблицу, не задается значения для первичного ключа. При отображении строк таблицы можно видеть, что система присваивает им последовательные целочисленные значения.

Второй столбец в таблице имеет тип данных UNIQUEIDENTIFIER. Для него указано значение по умолчанию NEWSEQUENTIALID(). Третий столбец имеет символьный тип данных.

Операторы INSERT, добавляющие строки в переменную этого типа данных, позволяют продемонстрировать и генерацию значений первичного ключа, и формирование значения столбца типа данных UNIQUEIDENTIFIER. В этом случае мы можем увидеть поведение системы, когда при добавлении строки не задано значение столбца UNIQUEIDENTIFIER. Это два последних оператора INSERT. Поскольку при описании столбца с типом данных UNIQUEIDENTIFIER указано значение по умолчанию в виде вызова функции NEWSEQUENTIALID(), при добавлении новой строки столбцу будет присвоено значение, сформированное системой. Эти значения можно увидеть при отображении строк таблицы. Значения отличаются друг от друга, т. к. этот тип данных обеспечивает уникальность значений.

Более подробно таблицы мы будем рассматривать в *следующей главе*.

4.8.6. Тип данных XML

Тип данных XML используется для хранения данных в формате XML. Данные в этом элементе должны быть корректными данными формата XML. Этот тип данных появился в SQL Server версии 2005.

Вообще про XML и использование этого типа данных опять же нужно писать отдельную книгу. Рекомендую найти соответствующую литературу и включиться в этот интересный мир. Официальные сведения, описание стандарта можно найти на сайте консорциума W3C: www.w3.org. Здесь же мы рассмотрим только некоторые аспекты этого типа данных и его использование в базах данных SQL Server.

XML является языком разметки текстов. HTML также является языком разметки, однако HTML имеет лишь фиксированный набор тегов, смысл которых заранее определен. В XML не существует заранее определенных тегов. Они создаются пользователем, и смысл тегам также задает пользователь или программа, которая выполняет работу с этими данными.

Рассмотрим пример создания простого документа XML в локальной переменной (пример 4.32).

Пример 4.32. Простой документ XML

```
USE master;
GO
DECLARE @X AS XML;
SET @X =
'<?xml version="1.0"?>
<Countries>
    <Country>
        <ID>RUS</ID>
        <Name>Российская Федерация</Name>
    </Country>
    <Country>
        <ID>USA</ID>
        <Name>United States of America</Name>
    </Country>
</Countries>'
SELECT @X;
GO
```

Здесь объявляется локальная переменная типа данных XML. Ей присваивается значение правильного документа XML. Весь текст заключается в апострофы, как обычная строковая константа.

Этот документ содержит сведения о странах. Для каждой страны задается ее код (идентификатор) и название.

Первая строка содержит указание на версию XML. Это версия 1.0, и есть подозрение, что другой не будет по причине достаточной полноты текущей версии. Эти данные в теге `<?xml?>` не являются обязательными. Далее идет сам текст документа. Все структурные элементы документа должны располагаться внутри тегов. Каждый открывающий тег должен быть "закрытым", т. е. ему должен сопутствовать соответствующий завершающий тег. Например, для задания кода страны используется

начальный тег `<ID>`, затем указывается значение самого кода и завершающий тег `</ID>`. Элементы могут быть вложенными. При этом следует сохранять правильность вложенности. Здесь должна присутствовать строгая иерархия вложенности. Например, следующая конструкция будет неверной:

```
<T1>Некий текст<T2> Текст</T1>...</T2>
```

Существуют и пустые теги, т. е. теги, не содержащие никаких данных. Пустой тег можно указать двумя способами: либо

```
<empty></empty>
```

либо еще проще:

```
<empty />
```

Затем в примере выполняется отображение значения этой локальной переменной. Результатом будет:

```
<Countries><Country><ID>RUS</ID><Name>Российская  
Федерация</Name></Country><Country><ID>USA</ID><Name>United States of  
America</Name></Country></Countries>
```

Документ XML должен начинаться с инструкции `<?xml?>`, где могут присутствовать некоторые параметры, общие для документа. В частности, это может быть номер версии языка, кодовая страница и др.

Важный момент: XML чувствителен к регистру, т. е. строчные и прописные буквы в нем рассматриваются как различные символы.

Элементом в XML является конструкция, состоящая из начального тела, значения и конечного тела. Само "значение" также может быть элементом, т. е. включать в себя начальные теги, значения и конечные теги.

Для помещения в документ примечаний используются начальные символы `<!--` и конечные `-->`. Внутри размещается произвольный текст, который не проверяется анализатором.

Элементы в документе XML могут иметь атрибуты. В начальном теге элемента в этом случае указывается имя атрибута, знак равенства и в кавычках — значение атрибута. Между элементами такой конструкции может добавляться любое количество пробелов. Возможны варианты, когда элемент не имеет значения, все необходимые значения могут быть заданы атрибутами.

Аналогичный по смыслу документ, что и показанный в примере 4.32, можно получить при использовании атрибутов (пример 4.33).

Пример 4.33. Задание документа XML с использованием атрибутов

```
USE master;  
GO  
DECLARE @X AS XML;  
SET @X =  
'<?xml version="1.0"?>
```

```
<Countries>
  <Country ID = "RUS" Name = "Российская Федерация"/>
  <Country ID = "USA" Name = "United States of America"/>
</Countries>';
SELECT @X;
GO
```

Здесь тот самый случай, когда элементы не имеют никаких значений. Атрибуты содержат все необходимые сведения.

Результатом выполнения пакета будет:

```
<Countries><Country ID="RUS" Name="Российская Федерация" /><Country ID="USA"
Name="United States of America" /></Countries>
```

В одном документе можно одновременно использовать как теги с заданными значениями, так и атрибуты. В примере 4.34 показано такое использование.

Пример 4.34. Задание документа XML с использованием атрибутов и тегов

```
USE master;
GO
DECLARE @X AS XML;
SET @X =
  '<?xml version="1.0"?>
<Countries>
  <Country ID = "RUS">
    <Name>Российская Федерация</Name>
  </Country>
  <Country ID = "USA">
    <Name>United States of America</Name>
  </Country>
</Countries>';
SELECT @X;
GO
```

Для представления в тексте документа специальных символов используются так называемые ссылки на сущности. Это пять конструкций:

- ◆ & — задает знак амперсанда;
- ◆ " — задает кавычку;
- ◆ ' — задает апостроф;
- ◆ < — задает знак <;
- ◆ > — задает знак >.

Каждая из перечисленных конструкций заканчивается символом точки с запятой (;). Здесь вы видите полную аналогию с HTML.

Если в тексте документа встречается большое количество перечисленных специальных символов, то для сокращения объема вводимых данных можно использовать секцию CDATA. Синтаксис такой секции выглядит следующим образом:

```
<! [CDATA[ произвольный текст ]]>
```

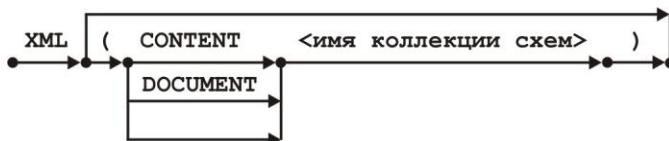
Текст внутри этой секции, `<произвольный текст>`, может содержать амперсанды, кавычки, апострофы, знаки `>` и `<`. Он только лишь не должен содержать подряд идущих трех символов: `]]>`.

Элемент с типом данных XML может быть типизированным или нетипизированным, он может содержать полный документ XML или только его фрагмент.

Синтаксис задания типа данных XML для элемента (столбца таблицы или локальной переменной) показан в листинге 4.5 и в соответствующем R-графе (граф 4.4).

Листинг 4.5. Синтаксис задания типа данных XML

```
XML [ ([CONTENT | DOCUMENT] <имя коллекции схем>) ]
```



Граф 4.4. Синтаксис задания типа данных XML

Ключевое слово **CONTENT** означает, что помещаемые в элемент данные должны быть корректным (well-formed) фрагментом документа XML.

Ключевое слово **DOCUMENT** указывает, что данные должны быть корректным документом XML.

Если ни одно слово не указано, то предполагается **CONTENT**.

Типизированный элемент позволяет выполнить проверку помещаемых в него данных не только с точки зрения правильности формата XML, то также и в плане соответствия описанию, хранящемуся в указанной коллекции схем (о коллекциях схем чуть позже).

Нетипизированный элемент не ссылается ни на какую коллекцию схем. При помещении данных в такой элемент выполняется лишь формальная проверка соответствия синтаксиса данных основным требованиям XML.

Ключевые слова **DOCUMENT** и **CONTENT** применяются только к типизированным документам.

Рассмотрим эти виды элементов XML. Начнем с нетипизированных.

Нетипизированные элементы XML

В примере 4.35 показано создание простой таблицы, содержащей столбец с нетипизированным типом данных XML.

Пример 4.35. Создание таблицы с нетипизированным столбцом XML

```
USE BestDatabase;
GO
CREATE TABLE REFREGXMLU
( CODCTR      CHAR(3) NOT NULL,    /* Код страны */
  REGIONDESCR XML,                  /* Описание регионов */
  CONSTRAINT PK_REFREGXMLU
    PRIMARY KEY (CODCTR)
);
GO
```

В эту таблицу поместим пару строк, которые будут содержать сведения о некоторых регионах России и некоторых штатах США. Выполните операторы примера 4.36.

Пример 4.36. Помещение данных в таблицу с нетипизированным столбцом XML

```
USE BestDatabase;
GO
/* Россия */
insert into REFREGXMLU (CODCTR, REGIONDESCR)
values ('RUS', '<?xml version="1.0"?>
<Regions>
<Region>
<ID>03</ID>
<Name>Краснодарский край</Name>
<Center>Краснодар</Center>
</Region>
<Region>
<ID>04</ID>
<Name>Красноярский край</Name>
<Center>Красноярск</Center>
</Region>
<Region>
<ID>05</ID>
<Name>Приморский край</Name>
<Center>Владивосток</Center>
</Region>
<Region>
<ID>07</ID>
<Name>Ставропольский край</Name>
<Center>Ставрополь</Center>
</Region>
</Regions>
');
```

```

/* Соединенные Штаты Америки */
insert into REFREGXMLU (CODCTR, REGIONDESCR)
    values ('USA', '<?xml version="1.0"?>
<Regions>
<Region>
<ID>AL</ID>
<Name>Alabama</Name>
<Center>MONTGOMERY</Center>
</Region>
<Region>
<ID>AK</ID>
<Name>Alaska</Name>
<Center>JUNEAU</Center>
</Region>
<Region>
<ID>AZ</ID>
<Name>Arizona</Name>
<Center>PHOENIX</Center>
</Region>
<Region>
<ID>AR</ID>
<Name>Arkansas</Name>
<Center>LITTLE ROCK</Center>
</Region>
</Regions>
');

GO

```

Помещаемые в столбец XML данные являются полными документами XML.

Отобразить данные таблицы можно обычным оператором SELECT:

```
SELECT * FROM REFREGXMLU;
```

Если данные выводятся в SQL Server Management Studio на сетку (Grid), то результат будет выглядеть, как показано на рис. 4.5.

The screenshot shows a table grid with two rows. The columns are labeled 'CODCTR' and 'REGIONDESCR'. Row 1 contains 'RUS' and an XML string representing the regions for Russia. Row 2 contains 'USA' and an XML string representing the regions for the United States. A cursor is positioned over the USA XML string. A tooltip 'Click to show in XML Editor' appears above the cursor, and another tooltip 'Click and hold to select this cell' appears below it. The XML content includes nested elements like <Regions>, <Region>, <ID>, <Name>, and <Center>.

CODCTR	REGIONDESCR
1 RUS	<Regions><Region><ID>03</ID><Name>Краснодарский К...</Name><Center>KRAZNO...</Center></Region><Region><ID>04</ID><Name>Северо-Кавказский Ф...</Name><Center>KRAZNO...</Center></Region><Region><ID>05</ID><Name>Сибирь</Name><Center>KRAZNO...</Center></Region><Region><ID>06</ID><Name>Урал</Name><Center>KRAZNO...</Center></Region><Region><ID>07</ID><Name>Дальневосточный Ф...</Name><Center>KRAZNO...</Center></Region><Regions>
2 USA	<Regions><Region><ID>AL</ID><Name>Alabama</Name><Center>MONTGOMERY</Center></Region><Region><ID>AK</ID><Name>Alaska</Name><Center>JUNEAU</Center></Region><Region><ID>AZ</ID><Name>Arizona</Name><Center>PHOENIX</Center></Region><Region><ID>AR</ID><Name>Arkansas</Name><Center>LITTLE ROCK</Center></Region></Regions>

Рис. 4.5. Отображение строк таблицы, содержащей данные XML

Если в этой панели щелкнуть мышью по полю столбца REGIONDESCR, содержащего данные формата XML, то программа в отдельной вкладке даст более удобный для восприятия текст. Например, для России:

```
<Regions>
  <Region>
    <ID>03</ID>
    <Name>Краснодарский край</Name>
    <Center>Краснодар</Center>
  </Region>
  <Region>
    <ID>04</ID>
    <Name>Красноярский край</Name>
    <Center>Красноярск</Center>
  </Region>
  <Region>
    <ID>05</ID>
    <Name>Приморский край</Name>
    <Center>Владивосток</Center>
  </Region>
  <Region>
    <ID>07</ID>
    <Name>Ставропольский край</Name>
    <Center>Ставрополь</Center>
  </Region>
</Regions>
```

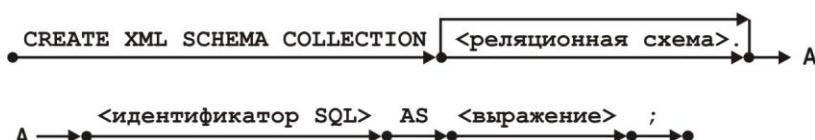
Типизированные элементы XML

Для описания структуры документов XML используются средства XSD (XML Schema Definition, определение схемы XML). Эти описания хранятся в системном каталоге конкретной базы данных. Они называются *коллекцией схем XML* (XML Schema Collection). Аналогом коллекции схем в "обычном" XML является DTD (Document Type Definition, определение типа документа).

Для создания коллекции схем в текущей базе данных используется оператор CREATE XML SCHEMA COLLECTION. Синтаксис оператора представлен в листинге 4.6 и в соответствующем R-графе (граф 4.5).

Листинг 4.6. Синтаксис оператора CREATE XML SCHEMA COLLECTION

```
CREATE XML SCHEMA COLLECTION [<реляционная схема>.]  
  <идентификатор SQL> AS <выражение>;
```



Граф 4.5. Синтаксис оператора CREATE XML SCHEMA COLLECTION

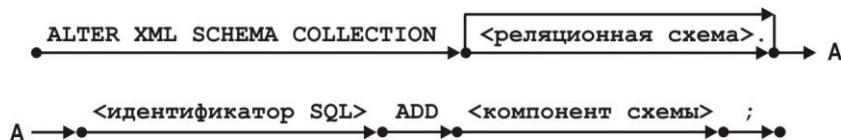
Здесь <выражение> является описанием, созданным средствами XSD.

Для добавления элементов в описание структуры документов XML в существующую коллекцию схем используется оператор ALTER XML SCHEMA COLLECTION. Его синтаксис представлен в листинге 4.7 и в соответствующем R-графе (граф 4.6).

Листинг 4.7. Синтаксис оператора ALTER XML SCHEMA COLLECTION

```
ALTER XML SCHEMA COLLECTION [<реляционная схема>.]
```

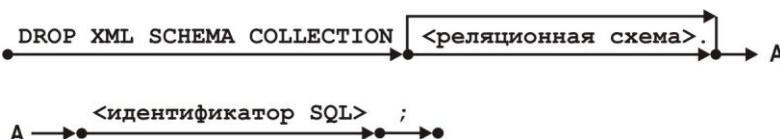
```
<идентификатор SQL> ADD <компонент схемы>;
```



Удалить существующую в базе данных коллекцию схем можно при использовании оператора DROP XML SCHEMA COLLECTION (листинг 4.8 и граф 4.7). Удалить можно лишь ту коллекцию схем, на которую не существует ссылок в элементах (столбцах таблиц) базы данных.

Листинг 4.8. Синтаксис оператора DROP XML SCHEMA COLLECTION

```
DROP XML SCHEMA COLLECTION [<реляционная схема>.]<идентификатор SQL>;
```



Для отображения коллекций схем текущей базы данных используется системное представление каталогов sys.xml_schema_collections. Основными столбцами, возвращаемыми этим представлением, являются:

- ◆ xml_collection_id — идентификатор коллекции схем (целое число). Является уникальным в конкретной базе данных;
- ◆ name — имя коллекции схем;
- ◆ create_date — дата создания;
- ◆ modify_date — дата изменения.

Чтобы создать коллекцию схем для таблицы из примера 4.35, куда должны помещаться данные, как показано в примере 4.36, нужно выполнить операторы, приве-

денные в примере 4.37. Здесь также создается и таблица REFREGXML, содержащая типизированный столбец XML.

Пример 4.37. Создание простой коллекции схем и таблицы с типизированным столбцом XML

```
USE BestDatabase;
GO
CREATE XML SCHEMA COLLECTION TestSchema AS
N'<?xml version="1.0" encoding="UTF-16"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="Region">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ID" type="xsd:string" />
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Center" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>' ;
GO

CREATE TABLE REFREGXML
( CODCTR      CHAR(3) NOT NULL,      /* Код страны */
  REGIONDESCR XML(TestSchema),        /* Описание регионов */
  CONSTRAINT PK_REFREGXML PRIMARY KEY (CODCTR)
);
GO
```

В коллекции схем указано, что в документе хранятся элементы с именами ID, Name и Center, имеющие строковый тип данных (для всех указано type="xsd:string").

При создании таблицы для столбца типа данных XML в скобках указывается имя коллекции схем:

```
REGIONDESCR XML(TestSchema),      /* Описание регионов */
```

Этот столбец является типизированным. По умолчанию ему присваивается характеристика CONTENT, т. е. он может содержать не целый документ XML, а фрагмент такого документа. При помещении новых данных в таблицу система будет проверять соответствие помещаемых данных тому описанию, которое существует в указанной коллекции схем TestSchema.

Теперь создадим в базе данных BestDatabase более интересную коллекцию схем с именем TestSchemaCtr. Выполните операторы примера 4.38.

Пример 4.38. Создание коллекции схем

```
USE BestDatabase;
GO
IF exists(SELECT * FROM sys.xml_schema_collections
    WHERE name = 'TestSchemaCtr')
    DROP XML SCHEMA COLLECTION TestSchemaCtr;

CREATE XML SCHEMA COLLECTION TestSchemaCtr AS
N'<?xml version="1.0" encoding="UTF-16"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
    <xsd:element name="Country">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="IDCTR" type="xsd:string" />
                <xsd:element name="NameCTR" type="xsd:string" />
                <xsd:element name="Regions" type="Region" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
<xsd:complexType name="Region">
    <xsd:choice minOccurs="0" maxOccurs="unbounded" >
        <xsd:sequence>
            <xsd:element name="ID" type="xsd:string" />
            <xsd:element name="Name" type="xsd:string" />
            <xsd:element name="Center" type="xsd:string" />
        </xsd:sequence>
    </xsd:choice>
</xsd:complexType>
</xsd:schema>';
GO
```

Вначале в операторе `IF` проверяется наличие такой же коллекции схем в текущей базе данных. Для этого используется оператор `SELECT`, обращающийся к системному представлению каталогов `sys.xml_schema_collections`. Если коллекция схем уже существует в базе данных (функция `exists` возвращает значение `TRUE`), то она удаляется оператором `DROP XML SCHEMA COLLECTION`.

Далее создается новая схема. Вначале описывается корневой тег с именем `Country`. Указывается, что он состоит из тегов (элементов данных) `IDCTR`, которые должны содержать строковые данные (задано `type="xsd:string"`), тега `NameCTR` тоже строкового типа и элемента `Regions`, который имеет тип `Region`.

После этого описывается объект `Region`, на который выполнялась ссылка в первой части описания. Указывается, что этот объект является сложным типом (`complexType`), может повторяться от нуля до неограниченного количества раз

(minOccurs="0" maxOccurs="unbounded"), состоит из трех элементов: ID, Name и Center, каждый из которых может содержать строковые данные (type="xsd:string").

Содержательно все это означает, что документ должен хранить сведения о стране, о ее коде и названии, а также может хранить сведения о любом количестве ее регионов, где присутствует код, название региона и центр региона.

Отобразите список коллекций схем базы данных, выполнив следующие операторы:

```
USE BestDatabase;
GO
SELECT xml_collection_id, name, create_date, modify_date
FROM sys.xml_schema_collections;
```

Результат выполнения выборки:

xml_collection_id	name	create_date	modify_date
1	sys	2009-04-13 12:59:13.390	2011-06-24 18:27:37.393
65597	TestSchema	2011-11-04 12:40:51.733	2011-11-04 12:40:51.733
65605	TestSchemaCtr	2011-11-04 12:55:29.073	2011-11-04 12:55:29.073

(3 row(s) affected)

Список коллекций схем базы данных также можно получить, используя диалоговые средства SQL Server Management Studio. В окне **Object Explorer** раскройте базу данных **BestDatabase**, затем папку **Programmability**, папку **Types** и папку **XML Schema Collections**. Полученный список показан на рис. 4.6.

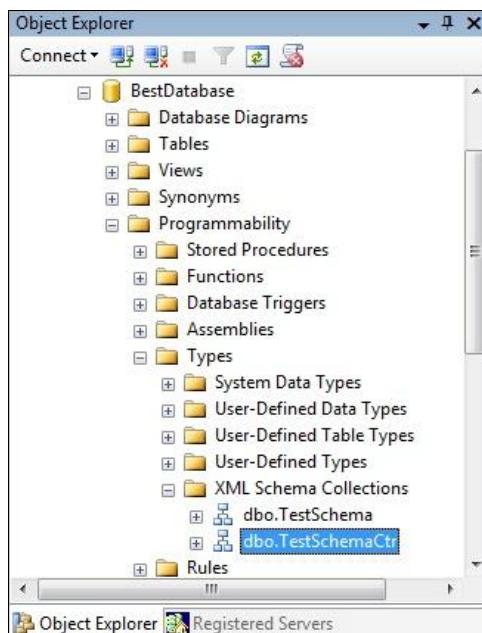


Рис. 4.6. Список коллекций схем

Чтобы просмотреть текст коллекции схем, нужно правой кнопкой мыши щелкнуть по ее имени и в появившемся контекстном меню выбрать элементы **Script XML Schema Collection as | CREATE To | New Query Editor Window**. Появится новое окно, которое будет содержать текст.

Создайте таблицу REFCTRXML, в которой столбец XML ссылается на коллекцию схем TestSchemaCtr (пример 4.39).

Пример 4.39. Создание таблицы, использующей коллекцию схем

```
USE BestDatabase;
GO
CREATE TABLE REFCTRXML
( CODCTR      CHAR(3) NOT NULL,    /* Код страны */
  REGIONDESCR XML(TestSchemaCtr), /* Описание регионов */
  CONSTRAINT PK_REFCTRXML PRIMARY KEY (CODCTR)
);
```

Для загрузки данных в таблицу выполните операторы из примера 4.40.

Пример 4.40. Заполнение таблицы, использующей коллекцию схем

```
USE BestDatabase;
GO
/* Россия */
insert into REFCTRXML (CODCTR, REGIONDESCR)
  values ('RUS', '<?xml version="1.0"?>
<Country>
  <IDCTR>RUS</IDCTR>
  <NameCTR>Российская Федерация</NameCTR>
  <Regions>
    <ID>03</ID>
    <Name>Краснодарский край</Name>
    <Center>Краснодар</Center>
    <!--   <Region> -->
    <ID>04</ID>
    <Name>Красноярский край</Name>
    <Center>Красноярск</Center>
    <!--   <Region> -->
    <ID>05</ID>
    <Name>Приморский край</Name>
    <Center>Владивосток</Center>
    <!--   <Region> -->
    <ID>07</ID>
    <Name>Ставропольский край</Name>
    <Center>Ставрополь</Center>
  </Regions>
</Country>
');
```

```
/* Соединенные Штаты Америки */
insert into REFCTRXML (CODCTR, REGIONDESCR)
    values ('USA', '<?xml version="1.0"?>
<Country>
    <IDCTR>USA</IDCTR>
    <NameCTR>United States of America</NameCTR>
    <Regions>
        <!--      <Region> -->
        <ID>AL</ID>
        <Name>Alabama</Name>
        <Center>MONTGOMERY</Center>
        <!--      <Region> -->
        <ID>AK</ID>
        <Name>Alaska</Name>
        <Center>JUNEAU</Center>
        <!--      <Region> -->
        <ID>AZ</ID>
        <Name>Arizona</Name>
        <Center>PHOENIX</Center>
        <!--      <Region> -->
        <ID>AR</ID>
        <Name>Arkansas</Name>
        <Center>LITTLE ROCK</Center>
    </Regions>
</Country>
') ;
GO
```

Отображение таблиц, содержащих столбцы XML

Для отображения подобных данных мы использовали обычный оператор `SELECT`, который предоставлял данные в исходном, т. е. не очень удобном для восприятия виде. В SQL Server существуют и другие, довольно сложные, возможности отображения данных XML.

У самого типа данных XML существует множество методов (методов объектов класса XML), которые позволяют, в том числе, выполнить отображение данных.

Рассмотрим только один вариант выборки данных из созданной ранее таблицы `REFCTRXML` с использованием метода `value()`.

Выполните операторы примера 4.41.

Пример 4.41. Выборка данных из столбца XML

```
USE BestDatabase;
GO
SELECT REGIONDESCR.value('(/Country/IDCTR)[1]', 'NVARCHAR(3)')
    AS 'Код',
```

```
REGIONDESCR.value('(/Country/NameCTR)[1]', 'NVARCHAR(30)')  
AS 'Название'  
FROM REFCTRXML;  
GO
```

Методу `value()` передается два параметра. Первый содержит указание на отыскиваемый объект с заданием индекса, определяющего номер отображаемого элемента в списке. Второй задает тип данных, в который будет преобразовываться результат выборки.

Результат выполнения запроса:

Код	Название
RUS	Российская Федерация
USA	United States of America

(2 row(s) affected)

В связи с ограниченностью объема книги мы более подробно подобные средства рассматривать не будем.

ЗАМЕЧАНИЕ

В настоящей версии системы в операторе `SELECT` все еще существует предложение `FOR XML`. Использовать его не рекомендуется, потому что это предложение будет удалено в последующих версиях сервера базы данных.

4.9. Создание и удаление пользовательских типов данных

В SQL Server существует возможность создавать пользовательские типы данных (или псевдонимы), основываясь на базовых типах данных, а также "новые" типы данных при помощи классов сборки в среде Microsoft .NET Framework CLR. Типы данных .NET Framework мы здесь обсуждать и рассматривать не будем по причине их довольно большой сложности и ограниченности объема книги.

Пользовательские типы данных могут создаваться средствами Transact-SQL и в среде Management Studio.

4.9.1. Синтаксис оператора создания пользовательского типа данных

Для создания пользовательского типа данных в языке Transact-SQL используется оператор `CREATE TYPE`. Его синтаксис для создания псевдонимов и табличных типов данных представлен в листинге 4.9 и соответствующих R-графах (графы с 4.8 по 4.16).

Листинг 4.9. Синтаксис оператора CREATE TYPE

```
CREATE TYPE [<имя схемы>.]<имя типа данных>
{ <описание псевдонима> | <описание табличного типа данных> };

<описание псевдонима> ::==
    FROM <имя базового типа> [({<точность> [, <масштаб>] | max})]
        [NULL | NOT NULL]

<описание табличного типа данных> ::==
    AS TABLE
    ( <определение столбца>
        [, <определение столбца>
        |, <определение вычисляемого столбца>
        |, <ограничение таблицы>
        ] ...)

<определение столбца> ::= <имя столбца> <тип данных>
    [ COLLATE <порядок сортировки> ]
    [ NULL | NOT NULL ]
    [ DEFAULT <выражение>
        | IDENTITY [(<начальное значение>, <приращение>)]
    ]
    [ ROWGUIDCOL ]
    [ <ограничение столбца> ... ]

<ограничение столбца> ::==
{   { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [ WITH (<параметр индекса> [, <параметр индекса>]...)
    | CHECK (<логическое выражение>)
}

<определение вычисляемого столбца> ::==
<имя столбца> AS <выражение для вычисляемого столбца>
[ PERSISTED [ NOT NULL ] ]
[ { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [ WITH (<параметр индекса> [, <параметр индекса>]...
    | CHECK (<логическое выражение>)
]

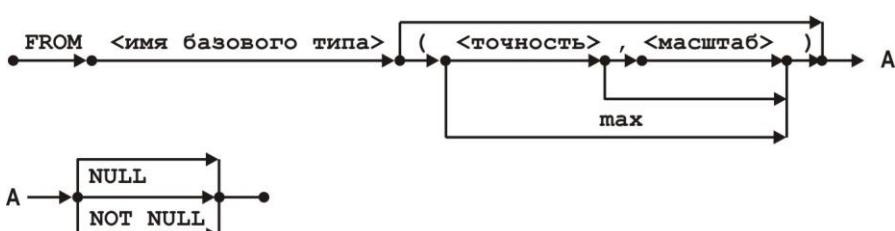
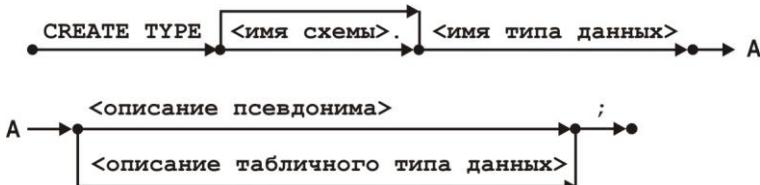
<ограничение таблицы> ::= <первичный или уникальный ключ>
| <ограничение CHECK>

<первичный или уникальный ключ> ::==
{ PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ]
```

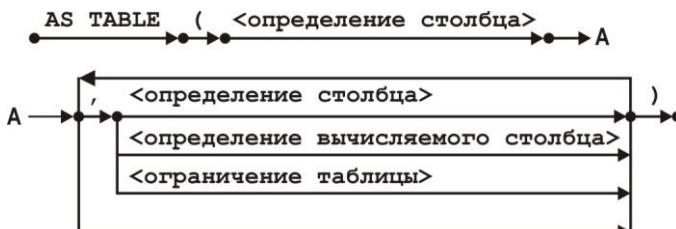
(<имя столбца> [ASC | DESC] [, <имя столбца> [ASC | DESC]] ...) ...)
 [WITH (<параметр индекса> [, <параметр индекса>] ...)]

<ограничение CHECK> ::=

CHECK (<логическое выражение>)



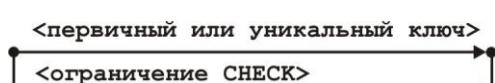
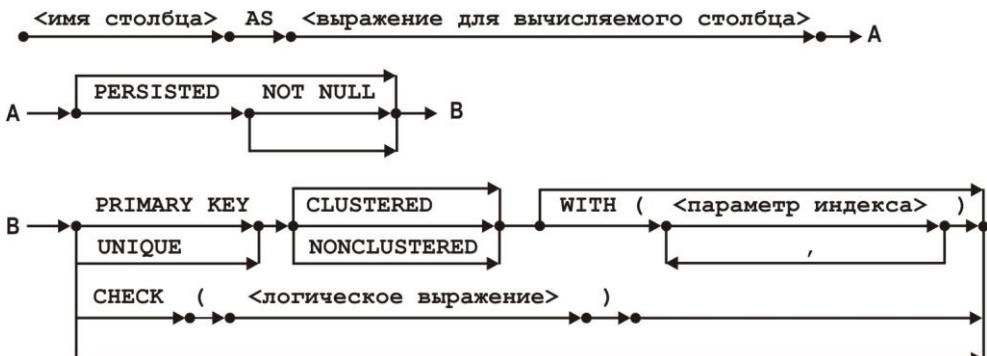
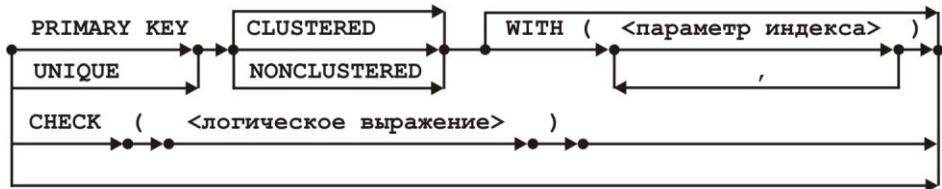
Граф 4.9. Синтаксис описания псевдонима



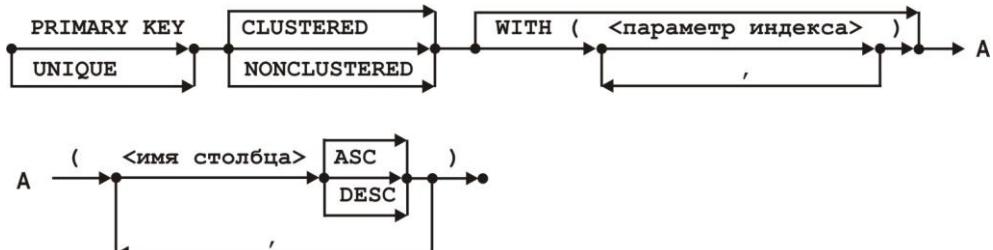
Граф 4.10. Синтаксис описания табличного типа данных



Граф 4.11. Синтаксис определения столбца



Граф 4.14. Синтаксис определения ограничения таблицы



Граф 4.15. Синтаксис определения ограничения первичного или уникального ключа



Граф 4.16. Синтаксис определения ограничения CHECK

Оператор CREATE TYPE в этом варианте позволяет создать простой псевдоним типа данных, основываясь на существующем в системе базовом типе данных, или пользовательский табличный тип данных. Напомню, пользовательский тип данных в среде Microsoft .NET Framework CLR мы не рассматриваем.

Псевдонимы типов данных после их создания в дальнейшем можно использовать в этой базе данных везде, где можно использовать системные типы данных.

Табличные типы данных могут быть использованы в качестве локальных переменных, а также входных и выходных параметров хранимых процедур и функций.

4.9.2. Создание псевдонима средствами Transact-SQL

Имя создаваемого пользовательского типа данных (псевдонима) должно соответствовать правилам задания идентификаторов. В операторе можно указать и имя схемы, которой будет принадлежать псевдоним. Если схема не указана, псевдоним будет принадлежать схеме по умолчанию dbo.

В предложении FROM указывается базовый тип, на основе которого создается псевдоним. Для базовых типов DECIMAL и NUMERIC можно задать значения точности и масштаба. Масштаб (точнее, количество знаков) можно задавать и для строковых типов данных CHAR, NCHAR, VARCHAR и NVARCHAR. Для строковых типов данных также можно указать и значение max. Подчеркну, что базовым типом может быть именно системный тип данных или его синоним. Использовать в качестве базового типа ранее созданный пользовательский тип нельзя.

Можно использовать любой системный тип данных за исключением типов данных XML, HIERARCHYID, CURSOR, TABLE и пространственных типов данных GEOMETRY и GEOGRAPHY.

В операторе можно указать допустимость значения NULL. Эта характеристика задается по умолчанию при стандартных установках системы.

Должен признаться, что у меня серьезные сомнения в целесообразности создания псевдонимов, по крайней мере, в SQL Server. Можно видеть, что псевдоним в отличие от системного типа данных позволяет лишь задать допустимость или недопустимость пустого значения, для типов данных DECIMAL и NUMERIC дает возможность также указать масштаб и точность, а для строковых типов данных — количество хранимых символов. И все.

4.9.3. Создание псевдонима в диалоговых средствах Management Studio

Для создания псевдонима в Management Studio в окне **Object Explorer** щелкните правой кнопкой мыши по элементу **User-Defined Data Types** и в контекстном меню выберите элемент **New User-Defined Data Types**. Появится диалоговое окно. Создадим псевдоним для типа данных **bit** и назовем его **BOOLEAN** (рис. 4.7). Из выпадающего списка **Data Type** выберем тип данных **bit**.

После щелчка по кнопке **OK** в базе данных **BestDatabase** будет создан новый псевдоним.

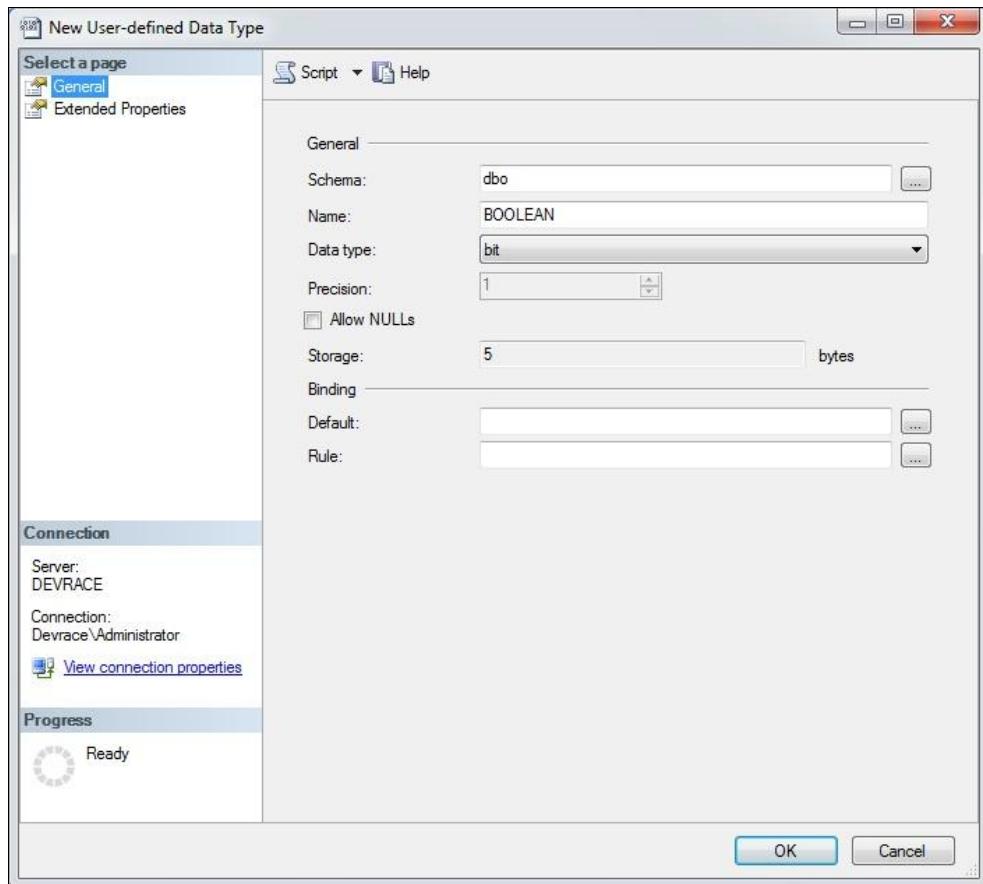


Рис. 4.7. Создание псевдонима BOOLEAN

4.9.4. Создание пользовательского табличного типа данных средствами Transact-SQL

Если выполняется создание пользовательского табличного типа данных, то в операторе CREATE TYPE (см. листинг 4.9) задается предложение AS TABLE. Любая таблица (и, соответственно, любой табличный тип данных) должна содержать не менее одного столбца. Она может содержать произвольное количество вычисляемых столбцов и ограничений таблицы. Элементы описания структуры таблицы отделяются друг от друга запятыми.

Здесь мы относительно кратко рассмотрим создание табличного типа данных, используемые для этого средства. Подробно о таблицах поговорим в следующей главе 5.

Определение столбца

Имя столбца должно быть уникальным среди имен столбцов этого табличного типа данных. Для столбца должен быть указан тип данных. Это может быть систем-

ный или пользовательский тип данных, созданный ранее в этой базе данных. Если столбец является строковым (CHAR или VARCHAR), то для него в предложении COLLATE может быть указан порядок сортировки, отличный от установленного по умолчанию.

Ключевыми словами NULL и NOT NULL задается допустимость для столбца неизвестного значения. Если не указано, то используются установки по умолчанию, обычно NULL.

В предложении DEFAULT можно задать значение по умолчанию для столбца. Это значение будет присваиваться столбцу, если при добавлении новой строки в таблицу (оператор INSERT) для столбца не будет задано никакого значения. В противном случае, при отсутствии указания значения по умолчанию столбцу будет присвоено значение NULL.

Предложение IDENTITY указывает, что столбец каждый раз при добавлении новой строки в таблицу будет получать новое уникальное значение. По умолчанию для первой добавляемой строки значение столбца устанавливается в единицу. При добавлении каждой последующей строки столбец получает значение на единицу большее, чем в предыдущей добавленной строке. Это поведение можно изменить, указав в скобках начальное значение, которое будет присвоено столбцу для первой строки таблицы, и величину приращения.

Предложения DEFAULT и IDENTITY являются взаимоисключающими.

Предложение ROWGUIDCOL указывает, что это столбец GUID, его значение будет уникальным.

Ограничения столбца

Для столбца можно задать так называемые *ограничения*. При описании табличного пользовательского типа данных допустимо использование трех ограничений: первичный ключ, уникальный ключ и проверка значения. Забегая немного вперед, следует напомнить, что для столбцов обычных таблиц (не табличного типа данных) можно использовать и ограничение внешнего ключа.

Ограничение первичного ключа задается словами PRIMARY KEY. В таблице может быть только один первичный ключ.

Ограничение уникального ключа (UNIQUE) указывает, что значение столбца должно быть уникальным среди всех значений в строках таблицы. Таблица может содержать произвольное количество уникальных ключей.

Ключевые слова CLUSTERED и NONCLUSTERED определяют характеристики индексов, которые будут автоматически создаваться для ограничения первичного или уникального ключа — будет ли индекс кластерным.

В предложении CHECK задается логическое условие, которое должно возвращать значение истинны, чтобы в таблицу была записана новая строка (INSERT) или чтобы были выполнены изменения данных строки (UPDATE).

Вычисляемые столбцы

В таблице могут присутствовать вычисляемые столбцы. Это столбцы, значения которых не вводятся пользователем, а получаются ("вычисляются") из значений других столбцов этой таблицы. Способ получения значения такого столбца указывается после ключевого слова `AS`. Выражение для значения вычисляемого столбца может быть любым правильным выражением, содержащим константы, имена столбцов этой таблицы, знаки операции.

Чаще всего значение вычисляемого столбца фактически не хранится в базе данных, а получается, "вычисляется", при выборке данных из таблицы. Однако при задании ключевого слова `PERSISTED` значение такого столбца будет храниться в таблице базы данных.

Для вычисляемых столбцов, как и для обычных столбцов, могут задаваться ограничения. Синтаксис и смысл ограничений точно такой же, как и у обычных столбцов.

Вот простой пример использования вычисляемого столбца. Пусть в таблице, описывающей сотрудников некоторой организации, присутствует столбец, содержащий значение заработной платы конкретного сотрудника:

```
SALARY DECIMAL(8, 2),
```

В этой же таблице может присутствовать столбец, в котором будет находиться выдаваемая сотруднику на руки сумма. Эта сумма с учетом налогов должна быть на 13% меньше заработной платы. Вычисляемый столбец может быть описан следующим образом:

```
NETSALARY AS SALARY * 0.87,
```

Что интересно, вычисляемый столбец может к тому же являться первичным или уникальным ключом.

Ограничения таблицы

Для табличного типа данных (на уровне всей таблицы) могут задаваться ограничения. Они похожи на ограничения столбца, однако если в ограничениях столбца соответствующие описания относились только к одному текущему столбцу, то в ограничениях таблицы могут присутствовать любые столбцы этой таблицы. Например, если первичный ключ состоит из одного столбца, то ограничение `PRIMARY KEY` можно описать в качестве ограничения этого столбца (это ограничение также можно описать и как ограничение таблицы). Если же первичный ключ таблицы в своем составе содержит более одного столбца, то такое ограничение можно описать только на уровне всей таблицы. По этой причине в синтаксисе ограничений таблицы для первичного и уникального ключа мы видим, что в круглые скобки заключен список столбцов, входящих в состав этого ключа.

Среди ограничений на уровне таблицы не может описываться ограничение внешнего ключа. Это относится только к пользовательскому табличному типу данных, но не к настоящим таблицам базы данных.

Пример создания пользовательских типов данных

В примере 4.42 показано создание простых пользовательских типов данных (псевдонимов) и табличного типа данных.

Пример 4.42. Создание пользовательских типов данных

```
USE BestDatabase;
GO
-- Создание псевдонимов
CREATE TYPE D_INT      FROM INT;
CREATE TYPE D_CHAR30   FROM VARCHAR(30);
GO
-- Создание табличного пользовательского типа данных
CREATE TYPE REFPEOPLE AS TABLE
(
    COD          D_INT NOT NULL IDENTITY, /* Код человека */
    NAME1        D_CHAR30,                /* Имя */
    NAME2        D_CHAR30,                /* Отчество */
    NAME3        D_CHAR30,                /* Фамилия */
    SALARY DECIMAL(8, 2),              /* Начисленная зарплата */
    NETSALARY AS SALARY * 0.87,        /* Выдаваемая сумма */
    PRIMARY KEY (COD)
);
GO
```

Здесь вначале создается два строковых псевдонима. Затем создается табличный тип данных. При описании столбцов в качестве их типов данных используются только что созданные псевдонимы.

Тут же мы применили на практике и использование вычисляемого столбца из предыдущего примера.

Последней строкой в описании таблицы задается ограничение первичного ключа.

Обратите внимание, что после операторов создания псевдонимов записан оператор go. Это необходимо сделать, чтобы вновь создаваемые псевдонимы стали "видимыми" для последующих операторов в этом скрипте, чтобы их можно было использовать в создаваемом далее табличном типе данных.

* * *

Чтобы в Management Studio просмотреть список пользовательских типов данных в конкретной базе данных, нужно в **Object Explorer** раскрыть список баз данных (**Databases**), раскрыть нужную базу данных, раскрыть папку **Programmability**, раскрыть **Types** и раскрыть **User-Defined Data Types**. Чтобы увидеть список пользовательских табличных типов данных, нужно в папке **Types** раскрыть еще и папку **User-Defined Table Types**. Если при создании любого пользовательского типа данных не была указана схема, то этот псевдоним будет относиться к схеме по умолчанию **dbo**.

На рис. 4.8 показано, как в окне **Object Explorer** выглядит описание созданных типов данных и табличного типа данных для пустой базы данных.

Для псевдонимов в этом списке помимо имени отображается базовый тип данных и допустимость пустого значения.

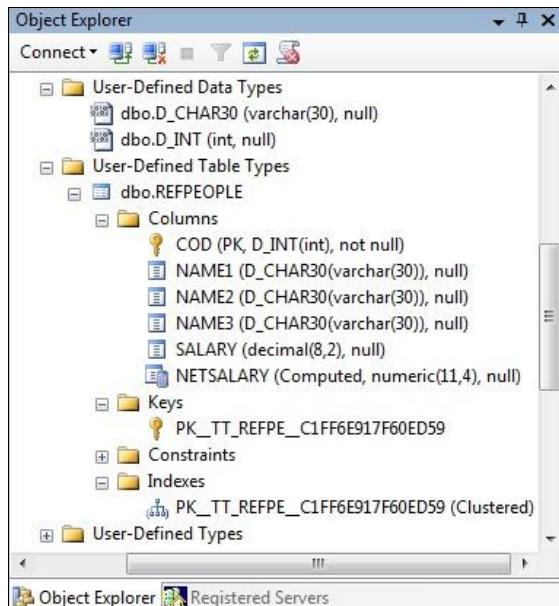


Рис. 4.8. Описание пользовательских типов данных и табличного типа данных

В случае табличного пользовательского типа данных можно раскрыть папку со списком столбцов созданного нами типа данных (**Columns**), папку ключей (**Keys**), папку ограничений (**Constraints**) и папку индексов (**Indexes**). Кстати, на рис. 4.8 мы видим, что для ограничения первичного ключа система по умолчанию создает кластерный (**Clustered**) индекс с длинным и нудным названием. В дальнейшем при создании реальных таблиц нашей базы данных мы будем ограничениям присваивать довольно осмысленные имена, чтобы при различных вариантах отображения сведений о таблицах получать читаемые тексты.

В скриптах создания демонстрационной базы данных существует множество примеров создания псевдонимов. После выполнения полного скрипта создания базы **BestDatabase** и пользовательских типов данных в Management Studio можно увидеть, что все типы принадлежат схеме **dbo**, т. к. они создавались в схеме по умолчанию.

4.9.5. Создание пользовательского табличного типа данных диалоговыми средствами Management Studio

Рассчитывая на то, что можно создать диалоговыми средствами Management Studio и пользовательский табличный тип данных, мы в окне **Object Explorer** раскрываем список баз данных (**Databases**), раскрываем базу данных **BestDatabase**, раскрываем

папку **Programmability**, раскрываем **Types**, щелкаем правой кнопкой мыши по строке **User-Defined Table Types** и выбираем в контекстном меню элемент **New User-Defined Table Type** (новый определяемый пользователем табличный тип данных). Однако вместо того, чтобы увидеть диалоговое окно, позволяющее задать характеристики этого типа данных, мы получаем скрипт, который содержит описание синтаксиса оператора `CREATE TYPE`:

```
-- =====
-- Create User-defined Table Type
-- =====
USE <database_name, sysname, AdventureWorks>
GO

-- Create the data type
CREATE TYPE <schema_name, sysname, dbo>.<type_name, sysname, TVP> AS TABLE
(
    <columns_in_primary_key, , c1> <column1_datatype, , int>
<column1_nullability,, NOT NULL>,
    <column2_name, sysname, c2> <column2_datatype, , char(10)>
<column2_nullability,, NULL>,
    <column3_name, sysname, c3> <column3_datatype, , datetime>
<column3_nullability,, NULL>,
    PRIMARY KEY (<columns_in_primary_key, , c1>)
)
GO
```

Так что для этих целей лучше использовать ваши глубокие познания в области языка Transact-SQL и конкретно оператора `CREATE TYPE`.

ЗАМЕЧАНИЕ

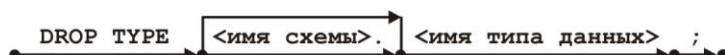
То, что в настоящий момент нет удобных диалоговых средств создания пользовательского табличного типа данных, еще не означает, что такие средства не появятся в ближайших релизах. Создать обычную таблицу достаточно удобными диалоговыми средствами в Management Studio можно, что мы и увидим в следующей главе 5.

4.9.6. Удаление пользовательского типа данных

Для удаления пользовательского типа данных из базы в языке Transact-SQL используется оператор `DROP TYPE`. Его синтаксис показан в листинге 4.10 и в соответствующем R-графе (граф 4.17).

Листинг 4.10. Синтаксис оператора `DROP TYPE`

```
DROP TYPE [<имя схемы>.]<имя типа данных>;
```



Граф 4.17. Синтаксис оператора `DROP TYPE`

Тип данных можно удалить только в том случае, если он не используется в других объектах (локальных переменных, столбцах таблиц, параметрах хранимых процедур) базы данных. Или в иной терминологии: если от него не зависят другие объекты базы данных.

Например, попробуйте оператором `DROP TYPE` удалить псевдоним `D_CHAR30`. Вы получите сообщение, что удалить его невозможно, поскольку на него ссылаются другие объекты. В данном случае это созданный нами табличный тип данных.

Более подробные сведения можно получить при попытке удаления этого псевдонима из среды Management Studio. Щелкните в окне **Object Explorer** правой кнопкой по имени псевдонима и в контекстном меню выберите элемент **Delete**. Появится окно удаления объекта **Delete Object** (рис. 4.9).

Если здесь щелкнуть по кнопке **OK**, то в следующих диалоговых окнах вы сможете увидеть, по какой причине удаление невозможно. Чтобы получить подробную информацию о зависимостях этого псевдонима, нужно в окне **Delete Object** щелкнуть мышью по кнопке **Show Dependencies** (показать зависимости).

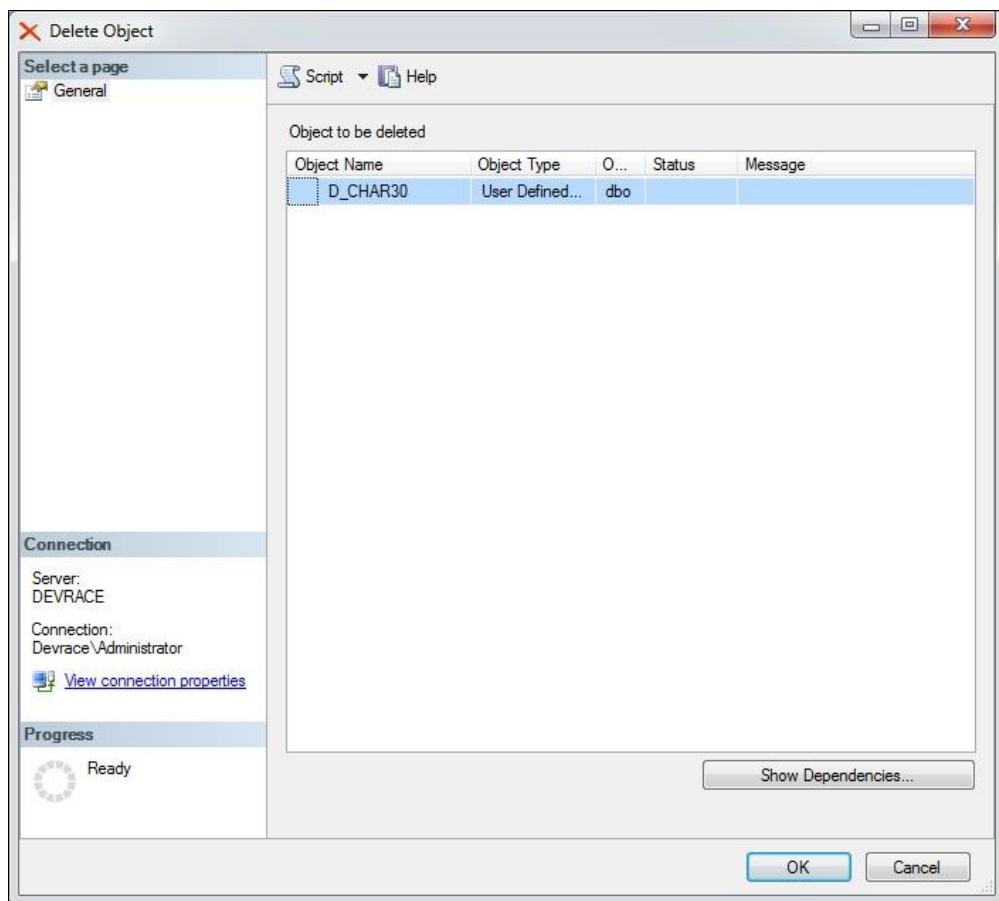


Рис. 4.9. Окно удаления объекта

Появится окно со списком объектов, зависящих от данного псевдонима (рис. 4.10).

Здесь видно, что от псевдонима зависит объект базы данных REFPEOPLE.

В этом же окне можно увидеть и список объектов, от которых зависит наш объект. Для этого в верхней части окна нужно выбрать переключатель **Objects on which [D_CHAR30] depends** (объекты, от которых зависит объект D_CHAR30).

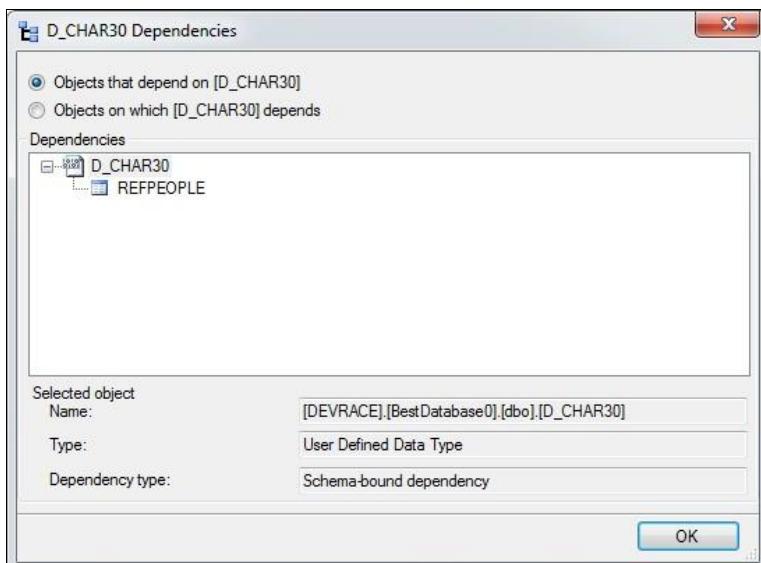


Рис. 4.10. Объекты, зависящие от псевдонима

Чтобы удалить пользовательский тип данных (да и любой другой объект базы данных), нужно устраниТЬ все его зависимости. Для этого следует либо удалить те объекты базы данных, которые его используют, либо в этих объектах устраниТЬ ссылки на удаляемый объект. Например, если какая-либо таблица содержит столбец, ссылающийся на удаляемый псевдоним, то нужно либо удалить этот столбец, либо заменить эту ссылку, установив для столбца системный тип данных.

* * *

На этом мы пока приостановим рассмотрение типов данных. Более подробно все моменты, связанные с табличными типами данных и с "настоящими" таблицами, мы рассмотрим в главе 5.

Что будет дальше?

В следующей главе мы рассмотрим создание таблиц, основных объектов реляционной базы данных, в которых хранятся все обрабатываемые данные.

Некоторые характеристики элементов таблиц мы уже рассмотрели, когда описывали пользовательские табличные типы данных. В следующей главе мы вернемся к этому еще раз и детально разберем все возможности, используемые при создании таблиц. Или почти все.



ГЛАВА 5

Работа с таблицами

- ◆ Создание таблиц в Transact-SQL
- ◆ Диалоговые средства создания таблиц в Management Studio
- ◆ Задание характеристик столбцов
- ◆ Ограничения столбцов и таблиц
- ◆ Вычисляемые столбцы
- ◆ Простые примеры создания таблиц
- ◆ Секционирование таблиц
- ◆ Использование файловых потоков
- ◆ Проверка зависимостей таблиц
- ◆ Удаление таблиц
- ◆ Изменение таблиц в Transact-SQL и средствами Management Studio
- ◆ Добавление, удаление, изменение характеристик обычных и вычисляемых столбцов
- ◆ Добавление, удаление, изменение ограничений

Важнейший объект базы данных — это *таблица*. В таблицах содержатся все обрабатываемые данные исходной предметной области.

С точки зрения пользователя структура таблицы — это некоторое количество столбцов (*column*) или, как их еще называют, *полей* (*field*). Основной характеристикой столбца является его тип данных. В таблице, если это не файловая таблица (*FileTable*), должно быть не менее одного столбца.

В таблице могут описываться так называемые *ограничения* (*constraint*), определяющие некоторые дополнительные характеристики одного или группы столбцов таблицы. К ограничениям относятся ограничение первичного ключа (*PRIMARY KEY*), ограничение уникального ключа (*UNIQUE*), ограничение на помещаемые в столбцы

значения (`CHECK`) и ограничение внешнего ключа (`FOREIGN KEY`). *Ограничения* — это свойства отдельных столбцов или группы столбцов таблицы.

Теоретически таблица может содержать произвольное количество строк. Таблица может быть и пустой. В частности, сразу после создания таблицы она не будет содержать никаких данных.

Таблицу можно создать как при использовании оператора `CREATE TABLE` языка Transact-SQL, так и с помощью диалоговых средств Management Studio.

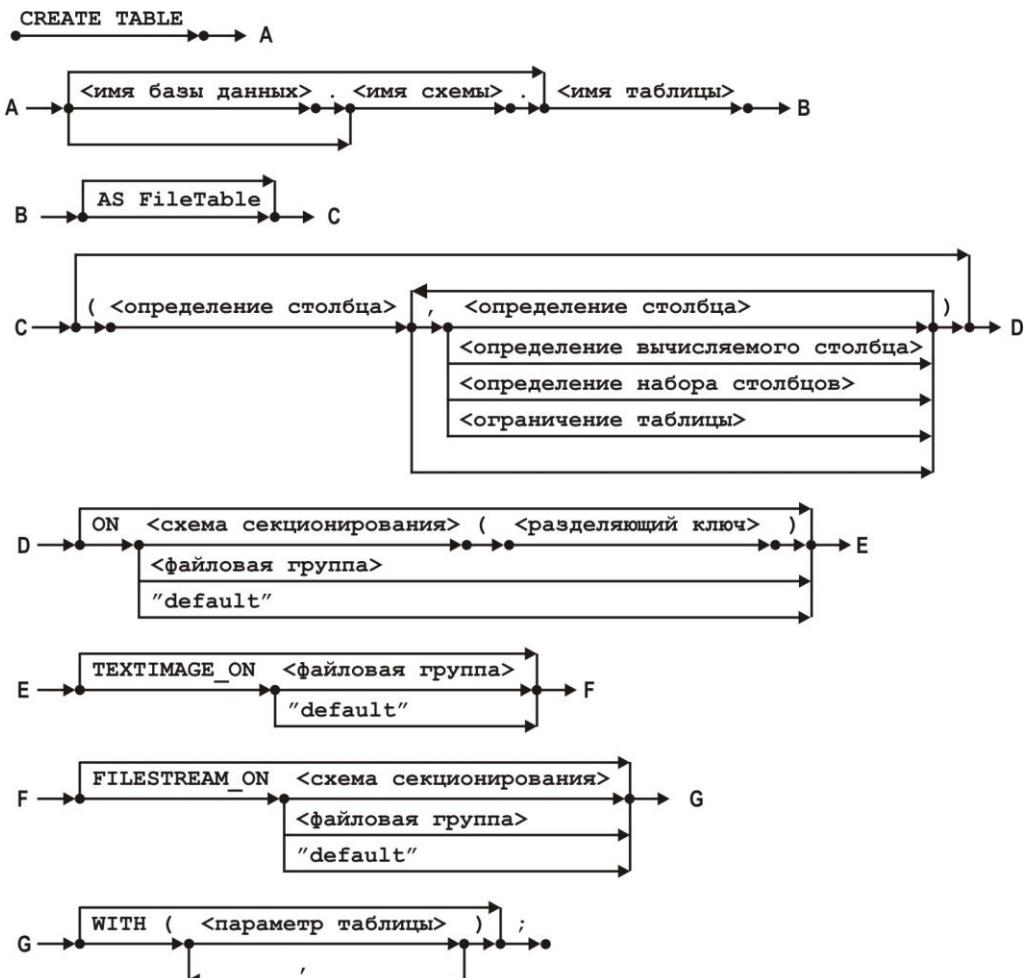
5.1. Синтаксис оператора создания таблицы

В одной базе данных может присутствовать чуть более двух миллиардов различных объектов, включая таблицы.

Для создания таблицы в языке Transact-SQL используется оператор `CREATE TABLE`. Его синтаксис представлен в листинге 5.1 и соответствующем R-графе (граф 5.1).

Листинг 5.1. Синтаксис оператора `CREATE TABLE`

```
CREATE TABLE
    [ [<имя базы данных>.]<имя схемы>.]<имя таблицы>
    [ AS FileTable ]
    [ ( <определение столбца>
        [, <определение столбца>
        |, <определение вычисляемого столбца>
        |, <определение набора столбцов>
        |, <ограничение таблицы>
        ] ... )
    ]
    [ ON { <схема секционирования> (<разделяющий ключ>
        | <файловая группа>
        | "default"
        }
    ]
    [ TEXTIMAGE_ON { <файловая группа>
        | "default"
        }
    ]
    [ FILESTREAM_ON { <схема секционирования>
        | <файловая группа>
        | "default"
        }
    ]
    [ WITH (<параметр таблицы> [, <параметр таблицы>] ...) ] ;
```



Граф 5.1. Синтаксис оператора CREATE TABLE

5.1.1. Общие характеристики таблицы

5.1.1.1. Идентификатор таблицы

В операторе `CREATE TABLE` для идентификации создаваемой таблицы должно быть указано как минимум имя таблицы. Можно также задать имя базы данных и имя схемы, отделяя их символом точки. Если в операторе не указано имя базы данных, то таблица создается в текущей базе данных — в той, которая была установлена в последнем операторе `USE`. Если не указано имя схемы, то таблица создается в схеме базы данных по умолчанию — обычно это схема `dbo`, однако при создании нового пользователя можно указать любую другую схему в качестве схемы по умолчанию.

Имя таблицы должно быть уникальным среди имен таблиц данной схемы. При этом в разных схемах базы данных могут присутствовать таблицы с одним и тем же именем.

Оператор `CREATE TABLE` позволяет создавать как обычные, так и временные таблицы. Временные таблицы могут быть локальными или глобальными. Имя локальной временной таблицы должно начинаться с символа `#`, имя глобальной временной таблицы — с двух символов `##`. Локальные временные таблицы видны только в одном сеансе (в текущем подключении к базе данных), где они создаются. С глобальными временными таблицами могут работать и другие параллельные процессы, выполняемые одновременно с процессом, в котором создается глобальная временная таблица.

Все временные таблицы (локальные и глобальные) удаляются автоматически при завершении сеанса, в котором они были созданы.

В предыдущей главе мы рассматривали переменные типа таблица, `TABLE`. Есть нечто общее между временными таблицами и табличным типом данных. Они существуют только в течение времени, пока активно приложение, их создавшее, и автоматически удаляются по завершении работы этого приложения. Есть между ними и различия.

- ◆ Переменные табличного типа могут быть только локальными. Они известны только в программе, их создавшей.
- ◆ Временные таблицы, в отличие от переменных табличного типа, не могут передаваться в качестве параметров программам и хранимым процедурам.
- ◆ Переменные табличного типа хранятся на клиентском компьютере. Для работы с ними не требуется использование сетевого трафика.

Количество символов в имени таблицы не должно превышать 128. Имя временной таблицы не может содержать более 116 символов.

После идентификации таблицы в круглых скобках перечисляется список столбцов таблицы, список вычисляемых столбцов и произвольное количество ограничений таблицы. Все эти описания отделяются друг от друга запятыми.

5.1.1.2. Предложение AS FileTable

Если в операторе задано предложение `AS FileTable`, то это означает, что создается так называемая *файловая таблица*. Такие таблицы мы позже рассмотрим в *этой главе*.

5.1.1.3. Определение столбца, вычисляемого столбца, набора столбцов

Описание столбцов таблицы может быть довольно серьезным с точки зрения содержания и синтаксиса. Это мы подробно рассмотрим в *разд. 5.1.2*.

5.1.1.4. Предложение ON

Предложение `ON` указывает, что строки таблицы должны размещаться в файловой группе по умолчанию ("default"), в конкретной файловой группе или на основании схемы секционирования. Секционированные таблицы рассмотрены далее в *этой главе*.

5.1.1.5. Предложение *TEXTIMAGE_ON*

Предложение *TEXTIMAGE_ON* указывает, что значения столбцов таблицы с типами данных *TEXT*, *NTEXT*, *IMAGE*, *XML*, *VARCHAR(MAX)*, *NVARCHAR(MAX)*, *VARBINARY(MAX)* и с пользовательскими типами данных будут помещаться в указанную файловую группу.

Если указано "default" (используется по умолчанию), то значения будут помещаться в файловую группу по умолчанию.

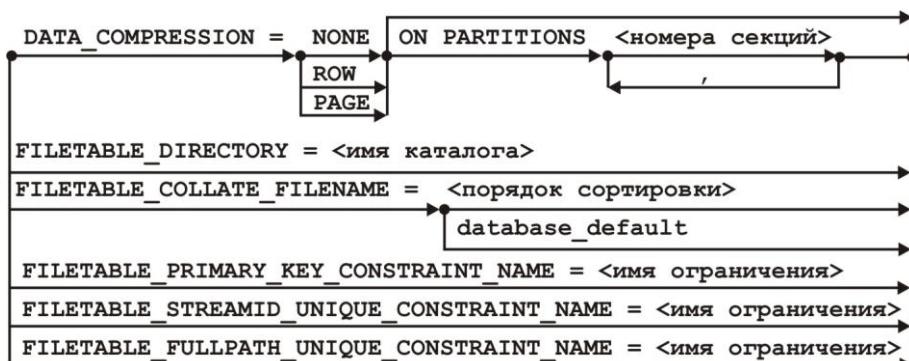
5.1.1.6. Предложение *FILESTREAM_ON*

Для хранения данных файловых потоков (*filestream*) предложение *FILESTREAM_ON* задает имя файловой группы, указывает файловую группу по умолчанию или указывает схему секционирования для распределения данных файловых потоков, если таблица является секционированной. Использование файловых потоков см. в разд. 5.6.

5.1.1.7. Предложение *WITH*

В необязательном предложении *WITH* можно указать параметры таблицы. Большинство параметров появилось в версии SQL Server 2012 (граф 5.2).

```
<параметр таблицы> ::=  
{ DATA_COMPRESSION = { NONE | ROW | PAGE }  
| ON PARTITIONS (<номера секций> [, <номера секций>] ...) ]  
| FILETABLE_DIRECTORY = <имя каталога>  
| FILETABLE_COLLATE_FILENAME =  
{ <порядок сортировки> | database_default }  
| FILETABLE_PRIMARY_KEY_CONSTRAINT_NAME = <имя ограничения>  
| FILETABLE_STREAMID_UNIQUE_CONSTRAINT_NAME = <имя ограничения>  
| FILETABLE_FULLPATH_UNIQUE_CONSTRAINT_NAME = <имя ограничения>  
}
```



Граф 5.2. Синтаксис оператора *CREATE TABLE*

Вариант *DATA_COMPRESSION* задает режим сжатия данных и позволяет указать секции, для которых будет применяться сжатие.

Допустимы следующие режимы:

- ◆ **NONE** — не производится сжатие данных всей таблицы или в указанных секциях;
- ◆ **ROW** — выполняется сжатие строк;
- ◆ **PAGE** — выполняется сжатие страниц.

При сжатии строк для значений столбцов большинства типов данных уменьшается объем внешней памяти, необходимой для хранения конкретного значения. Например, если столбец описан с типом данных **BIGINT**, а текущее его значение в строке таблицы может поместиться в один байт, то для него и отводится один байт.

При сжатии страниц вначале выполняется сжатие строк. После этого следуют этапы, которые называются *сжатием префикса* и *сжатием словаря*. Смысл этих действий в том, что повторяющиеся части значений выносятся в отдельное поле, а в столбцах даются ссылки на такие поля. Более подробные сведения см. в Books Online.

Как все процессы сжатия данных скажутся на экономии места на внешнем носителе и на производительности системы можно определить, скорее всего, опытным путем.

После ключевых слов **ON PARTITIONS** можно указать, к каким секциям хранения данных таблицы применяются средства сжатия. Это относится только к секционированным таблицам.

Секции указываются перечислением их номеров и в конструкции **<секция1> TO <секция2>**, что означает диапазон секций от секция1 до секция2. Например:

ON PARTITIONS (1, 3, 5 TO 8)

Здесь указываются секции 1, 3 и секции с номерами от 5 до 8.

О секционированных таблицах см. далее в *разд. 5.3.*

Следующие варианты параметров таблицы появились с версии SQL Server 2012. Они относятся к файловым таблицам (**FileTable**).

FILETABLE_DIRECTORY = <имя каталога>

Задает имя каталога, где будут храниться данные файловой таблицы. Если параметр не указан, то будет использован каталог с именем самой файловой таблицы.

FILETABLE_COLLATE_FILENAME = { <порядок сортировки> | database_default }

Задает порядок сортировки, который будет использован для столбца **name** файловой таблицы. Порядок сортировки должен быть нечувствительным к регистру.

FILETABLE_PRIMARY_KEY_CONSTRAINT_NAME = <имя ограничения>

Указывает имя, которое будет присвоено ограничению первичного ключа, который автоматически создается для файловой таблицы. Если параметр не указан, система генерирует это имя.

FILETABLE_STREAMID_UNIQUE_CONSTRAINT_NAME = <имя ограничения>

Задает имя, которое будет присвоено ограничению уникального ключа, автоматически создаваемому для столбца **stream_id** файловой таблицы. Если параметр не указан, система генерирует имя.

`FILETABLE_FULLPATH_UNIQUE_CONSTRAINT_NAME = <имя ограничения>`

Задает имя, которое будет присвоено ограничению уникального ключа, автоматически создаваемому для столбцов `parent_path_locator` и `name` файловой таблицы. Если параметр не указан, система генерирует имя.

Файловые таблицы рассмотрены далее в *этой главе*.

5.1.2. Определение столбца

Обычная таблица может содержать до 1024 столбцов. Размер строки таблицы должен быть таким, чтобы строка помещалась на одну страницу базы данных размером 8 Кбайт. При этом для хранения данных влов используются отдельные страницы; размер таких данных может быть очень большим. Для строковых данных переменной длины система также использует эффективную стратегию хранения; во многих случаях на основной странице хранится лишь указатель в 24 байта, сами же данные размещаются в других страницах. Для разреженных (SPARSE) столбцов используется стратегия хранения, сокращающая объем требуемой памяти. Таблицы, содержащие разреженные столбцы, называются "широкими таблицами" (wide table). Такие таблицы могут содержать до 30000 столбцов, однако количество неразреженных столбцов в них не должно превышать 1024.

Синтаксис определения столбца таблицы представлен в листинге 5.2 и соответствующем R-графе (граф 5.3).



Граф 5.3. Синтаксис определения столбца таблицы

Листинг 5.2. Синтаксис определения столбца таблицы

```

<определение столбца> ::= 
  <имя столбца> <тип данных>
  [ FILESTREAM ]
  [ COLLATE <порядок сортировки> ]
  [ NULL | NOT NULL ]
  
```

```
[ DEFAULT <выражение>
  | IDENTITY [(<начальное значение>, <приращение>) ]
  ]
[ ROWGUIDCOL ]
[ <ограничение столбца> ... ]
[ SPARSE ]
```

5.1.2.1. Имя столбца

Имя столбца должно быть уникальным в данной таблице. Другие таблицы этой базы данных могут содержать столбцы с теми же именами.

5.1.2.2. Тип данных

Тип данных — системный или пользовательский тип данных, созданный ранее для этой базы данных. Столбец таблицы не может иметь тип данных TABLE. Подробности о типах данных см. в *главе 4*.

5.1.2.3. Ключевое слово FILESTREAM

Необязательное ключевое слово FILESTREAM допустимо только для типа данных VARBINARY(MAX). Все значения такого столбца будут храниться не непосредственно в самой базе данных, а в контейнере данных в файловой системе Windows. В этом случае таблица также должна содержать еще и столбец с системным типом данных UNIQUEIDENTIFIER с атрибутом ROWGUIDCOL. Этот столбец должен быть описан как уникальный или первичный ключ данной таблицы. Файловые потоки мы подробно рассмотрим чуть позже в *этой главе*.

5.1.2.4. Предложение COLLATE

Необязательное предложение COLLATE позволяет для строкового столбца задать порядок сортировки, отличный от порядка, установленного для всей базы данных.

5.1.2.5. Ключевые слова NULL / NOT NULL

Ключевые слова NULL | NOT NULL определяют допустимость для столбца значения NULL. По умолчанию, при стандартных установках системы, допустимо наличие и значения NULL. Для столбцов, входящих в состав первичного ключа таблицы, рекомендуется всегда явно задавать характеристику NOT NULL независимо от установок системы.

5.1.2.6. Предложение DEFAULT

Предложение DEFAULT позволяет задать для столбца значение по умолчанию. Это значение будет помещаться в столбец, если при добавлении новой строки в таблицу в операторе INSERT не было явно указано его значение. Значение по умолчанию не определяет никакого действия, если при изменении существующей строки таблицы в операторе UPDATE не было задано значение этого столбца. Значение по умолчанию нельзя задавать для столбца IDENTITY.

5.1.2.7. Ключевое слово **IDENTITY**

Ключевое слово **IDENTITY** означает, что столбцу автоматически будет присваиваться уникальное числовое (целочисленное) значение при помещении новой строки в таблицу. Такие столбцы называются *автоинкрементными* (auto increment). В документации Books Online эти столбцы называют *столбцами идентификаторов*. Такие столбцы, как правило, используются для задания искусственных первичных ключей таблицы.

По умолчанию для первой добавляемой строки таблицы этому столбцу будет установлено значение 1. Все последующие присваиваемые значения будут увеличиваться на единицу. Эти характеристики по умолчанию для столбцов **IDENTITY** можно изменить, задав после ключевого слова **IDENTITY** в скобках начальное значение и величину приращения:

IDENTITY (<начальное значение>, <приращение>)

Для подобного автоинкрементного столбца нельзя задавать значение по умолчанию (предложение **DEFAULT**). Столбец с такой характеристикой должен иметь тип данных **TINYINT**, **SMALLINT**, **INT**, **BIGINT**, **DECIMAL** или **NUMERIC**. Если ему задается тип данных **DECIMAL** или **NUMERIC**, то количество дробных знаков должно быть указано нулевым. Столбец типа **IDENTITY** может быть в таблице только один.

В обычных условиях столбцу с этой характеристикой нельзя явно присвоить значение. Имя такого столбца не должно присутствовать в операторах **INSERT** и **UPDATE**. При этом существует оператор **SET IDENTITY_INSERT**, дающий возможность задавать конкретное значение такому столбцу. Его синтаксис приведен в листинге 5.3 и в соответствующем R-графе (граф 5.4).

Листинг 5.3. Синтаксис оператора **SET IDENTITY_INSERT**

```
SET IDENTITY_INSERT [ [<имя базы данных>.]<имя схемы>.]<имя таблицы>
{ ON | OFF };
```



Граф 5.4. Синтаксис оператора **SET IDENTITY_INSERT**

Если для таблицы указано **SET IDENTITY_INSERT ON**, то столбцу **IDENTITY** можно явно присвоить значение. Если вновь присваиваемое значение больше текущего значения, которое автоматически присваивается этому столбцу, то новое значение заменит текущее (автоматически присваиваемое) значение.

В одной сессии соединения с базой данных только для одной таблицы базы данных можно установить такую возможность.

5.1.2.8. Ключевое слово ROWGUIDCOL

Ключевое слово `ROWGUIDCOL` указывает, что столбец является столбцом идентификаторов GUID. Такой столбец должен быть только один в таблице. Его тип данных должен быть `UNIQUEIDENTIFIER`. Некоторые подробности про этот тип данных, используемые для работы с ним функции и примеры см. в предыдущей главе 4. Такой столбец обязательно должен быть описан в таблице при использовании файловых потоков.

5.1.2.9. Ключевое слово SPARSE

Ключевое слово `SPARSE` указывает, что столбец является разреженным столбцом. Такой тип столбцов имеет смысл использовать для экономии внешней памяти, если в строках таблицы присутствует большое количество значений `NULL` для соответствующего столбца. Это ключевое слово может быть указано для любых типов данных за исключением `TEXT`, `NTEXT`, `IMAGE`, `TIMESTAMP`, `GEOMETRY`, `GEOGRAPHY`, а также пользовательских типов данных. Столбец не может иметь атрибутов `NOT NULL` и `FILESTREAM`. В таблице может присутствовать столбец, являющийся набором столбцов `SPARSE` (см. далее разд. 5.1.5).

* * *

Для одного столбца (уровень столбца) или для группы столбцов таблицы (уровень таблицы) можно указать ограничения столбца или ограничения таблицы. Задание ограничения одного столбца можно также выполнить и на уровне таблицы.

5.1.3. Ограничения столбца и ограничения таблицы

В базах данных существует четыре вида ограничений как для одного столбца, так и для группы столбцов таблицы. Это:

- ◆ первичный ключ (`PRIMARY KEY`);
- ◆ уникальный ключ (`UNIQUE`);
- ◆ внешний ключ (`FOREIGN KEY`);
- ◆ ограничение на значение, помещаемое в столбец, в столбцы таблицы (`CHECK`).

Начальный синтаксис ограничения одного столбца и ограничения на уровне таблицы представлен в листинге 5.4 и в соответствующем R-графе (граф 5.5). "Начальный синтаксис" он лишь потому, что содержит только нетерминальные символы и еще не дает представления о реальном содержимом языковых конструкций.

Листинг 5.4. Синтаксис определения ограничения столбца таблицы и ограничения таблицы

```
<ограничение столбца или ограничение таблицы> ::=  
[ CONSTRAINT <имя ограничения> ]  
{   <первичный ключ>
```

```

| <的独特键>
| <外键>
| <约束 CHECK>
}
```



5.1.3.1. Имя ограничения

Любому ограничению вы можете по желанию присвоить конкретное имя, используя предложение **CONSTRAINT**. Можно порекомендовать всем ограничениям присваивать осмысленные имена, иначе система присвоит свое имя, которое вам ничего толкового не скажет, когда вы будете просматривать структуру вашей таблицы, например, в программе Management Studio. Имена ограничений первичного и уникального ключа не должны совпадать с именами других ограничений и индексов в базе данных.

Ограничениям первичного ключа я задаю имя **PK_<имя таблицы>**, для уникальных ключей использую **UK_<номер ключа в таблице>_<имя таблицы>**, хотя уникальные ключи в своих разработках использую как-то не очень часто. Для внешнего ключа применяю **FK_<номер ключа в таблице>_<имя таблицы>** и для ограничения **CHECK** — конструкцию **CK_<номер ограничения>_<имя таблицы>**. Эти же имена будут присвоены и индексам, поддерживающим ограничения первичного и уникального ключа.

5.1.3.2. Ограничения первичного и уникального ключа

Первичный ключ (**PRIMARY KEY**) может быть только один в таблице. Основное свойство первичного ключа — его уникальность. В таблице не может быть двух строк с одинаковыми значениями первичного ключа. Ключ может состоять из одного или нескольких столбцов. Столбцы, входящие в состав первичного ключа, не могут иметь значения **NULL**, для них в описании нужно явно указать **NOT NULL**.

Уникальный ключ (**UNIQUE**) также содержит уникальное значение в пределах таблицы. Таблица может содержать несколько уникальных ключей. Столбцы, входящие в состав уникального ключа, в отличие от столбцов первичного ключа, могут иметь и значение **NULL**.

В случае уникального ключа два значения **NULL** считаются равными друг другу. Иными словами, в таблицу нельзя будет поместить две строки, у которых уникальные ключи имеют значение **NULL**. Будет помещена только первая запись. При по-

пытке поместить такую же вторую запись будет выдано сообщение о дублировании значений. Если уникальный ключ состоит из нескольких столбцов, то любая комбинация значений, включающих и значения `NULL`, должна быть уникальной в таблице.

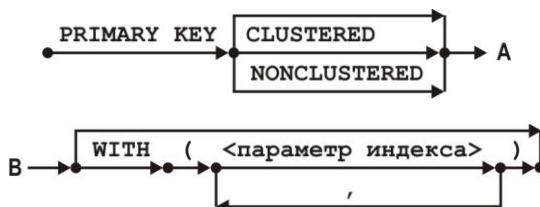
Для первичного и уникального ключа система создает индекс. Если вы указали в предложении `CONSTRAINT` имя, то именно оно будет присвоено и индексу. Иначе система сгенерирует свое имя, одинаковое и для ограничения, и для индекса.

Как первичный, так и уникальный ключи могут принимать участие в связке "внешний ключ/первичный (уникальный) ключ".

Синтаксис определения первичного ключа (`PRIMARY KEY`) на уровне столбца и на уровне таблицы представлен в листингах 5.5 и 5.6, соответственно, и в графах 5.6 и 5.7.

Листинг 5.5. Синтаксис ограничения первичного ключа на уровне столбца таблицы

```
<первичный ключ (столбец)> ::=  
    PRIMARY KEY  
        [ CLUSTERED | NONCLUSTERED ]  
        [ WITH (<параметр индекса> [, <параметр индекса>] ...) ]
```



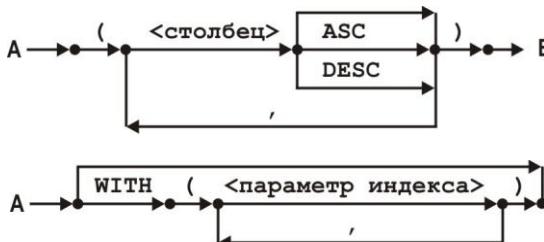
Граф 5.6. Синтаксис ограничения первичного ключа на уровне столбца таблицы

Листинг 5.6. Синтаксис ограничения первичного ключа на уровне таблицы

```
<первичный ключ (таблица)> ::=  
    PRIMARY KEY  
        [ CLUSTERED | NONCLUSTERED ]  
        (<столбец> [ ASC | DESC ] [, <столбец> [ ASC | DESC ] ]...)  
        [ WITH (<параметр индекса> [, <параметр индекса>] ...) ]
```



Граф 5.7. Синтаксис ограничения первичного ключа на уровне таблицы
(см. окончание)



Граф 5.7. Окончание

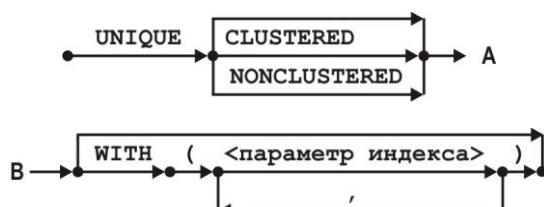
Синтаксические конструкции для отдельного столбца (на уровне столбца) и для группы столбцов, т. е. на уровне таблицы, мало отличаются друг от друга. Основным их отличием является то, что для ограничения таблицы обязательно в круглых скобках должны быть указаны имена столбцов, к которым применяется это ограничение. Для ограничения на уровне столбца такое указание, разумеется, не нужно. Любое ограничение на уровне таблицы может относиться и к одному единственному столбцу. В этом случае в скобках указывается имя этого столбца. Я использую и для отдельных столбцов синтаксические конструкции ограничения для всей таблицы, располагая их в самом конце оператора `CREATE TABLE`.

Для столбцов первичного ключа (только на уровне таблицы, но не на уровне столбца) можно указать упорядоченность в создаваемом для этого ключа индексе. Ключевое слово `ASC` означает упорядочение по возрастанию значений, ключевое слово `DESC` — по убыванию. Причем в рамках одного составного первичного или уникального ключа отдельные столбцы могут упорядочиваться по возрастанию, другие по убыванию, т. е. допустима смешанная упорядоченность. По умолчанию принимается упорядоченность столбцов по возрастанию значений.

Синтаксис задания ограничения уникального ключа (`UNIQUE`) отличается от синтаксиса первичного ключа только лишь названием вида ключа. Соответствующие синтаксические конструкции представлены в листингах 5.7 и 5.8 и в соответствующих R-графах (граф 5.8 и 5.9).

Листинг 5.7. Синтаксис ограничения уникального ключа на уровне столбца таблицы

```
<уникальный ключ (столбец)> ::=  
    UNIQUE  
        [ CLUSTERED | NONCLUSTERED ]  
        [ WITH (<параметр индекса> [, <параметр индекса>] ...) ]
```



Граф 5.8. Синтаксис ограничения уникального ключа на уровне столбца таблицы

Листинг 5.8. Синтаксис ограничения уникального ключа на уровне таблицы

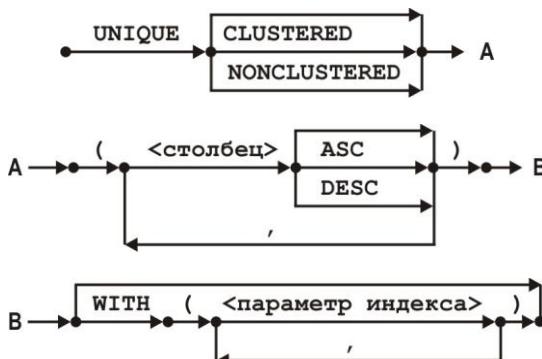
<уникальный ключ (таблица)> ::=

UNIQUE

[CLUSTERED | NONCLUSTERED]

(<столбец> [ASC | DESC] [, <столбец> [ASC | DESC]]...)

[WITH (<параметр индекса> [, <параметр индекса>] ...)]



Граф 5.9. Синтаксис ограничения уникального ключа на уровне таблицы

В рассмотренных синтаксических конструкциях создания первичного и уникального ключа описывается и предложение **WITH**, где в скобках можно перечислить некоторые параметры индекса, создаваемого для поддержания ключа.

Создаваемые индексы

Для первичного и уникального ключа система автоматически создает индекс.

Вы можете указать, что для поддержки ключа будет использоваться *кластерный* (ключевое слово **CLUSTERED**) индекс. Используется еще термин "*кластеризованный*". По умолчанию для первичного ключа создается кластерный индекс, для уникального ключа — *некластерный* (или *некластеризованный*, **NONCLUSTERED**).

Таблица может иметь только один кластерный индекс. В SQL Server индексы на внешних носителях физически представлены в виде сбалансированных деревьев. Они имеют иерархическую структуру. В кластерном индексе самые нижние узлы (узлы последнего уровня) содержат и соответствующие строки данных этой таблицы.

Более подробно об индексах мы поговорим в следующей главе 6. Там же рассмотрим и некоторые параметры индекса, которые можно задавать после ключевого слова **WITH**.

5.1.3.3. Ограничение внешнего ключа

Внешний ключ является важнейшим элементом механизма, обеспечивающего связи между таблицами в базе данных, реализующего отношение "один ко многим".

Это отношение является универсальным отношением, поддерживающим все основные связи в реляционных базах данных.

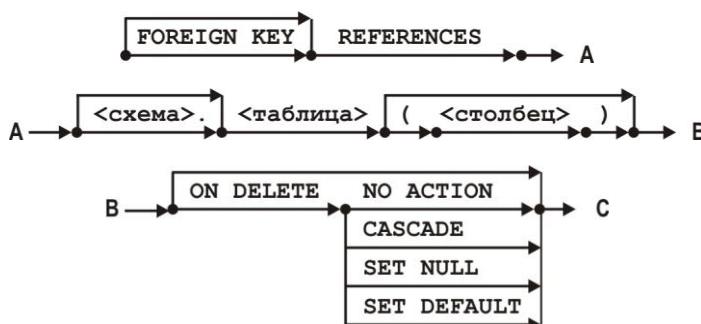
Ограничение внешнего ключа (`FOREIGN KEY`) может описываться на уровне столбца и на уровне таблицы. *Внешний ключ* — это столбец или группа столбцов, значение которых ссылается на значение первичного или уникального ключа в родительской таблице. Родительской таблицей может быть другая или та же самая таблица. Она может располагаться в той же самой или в другой схеме, но обязательно в той же базе данных. В родительской таблице должна присутствовать строка, у которой значение соответствующего первичного или уникального ключа совпадает со значением внешнего ключа подчиненной дочерней таблицы. Если в родительской таблице нет такой строки, то новая строка не будет помещена в дочернюю таблицу (в случае оператора `INSERT`) или не будет выполнено изменение существующей строки дочерней таблицы, если в ней изменяется значение столбцов, входящих в состав внешнего ключа (при использовании оператора `UPDATE`).

Основное правило для внешнего ключа: в родительской таблице должна существовать строка, первичный или уникальный ключ которой равен значению внешнего ключа дочерней таблицы. Таким образом поддерживается так называемая *декларативная целостность базы данных*.

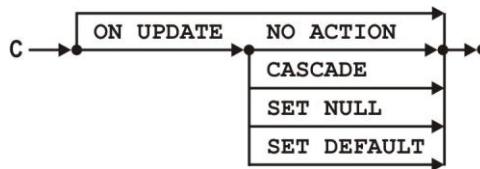
Синтаксис задания внешнего ключа на уровне столбца и на уровне таблицы, опять же с небольшими изменениями, представлен в листингах 5.9 и 5.10 и в соответствующих R-графах (граф 5.10 и 5.11).

Листинг 5.9. Синтаксис ограничения внешнего ключа на уровне столбца

```
<ограничение внешнего ключа (столбец)> ::=  
[ FOREIGN KEY ]  
    REFERENCES [<схема>.]<таблица> [ (<столбец>) ]  
        [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]  
        [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
```



Граф 5.10. Синтаксис ограничения внешнего ключа на уровне столбца
(см. окончание)



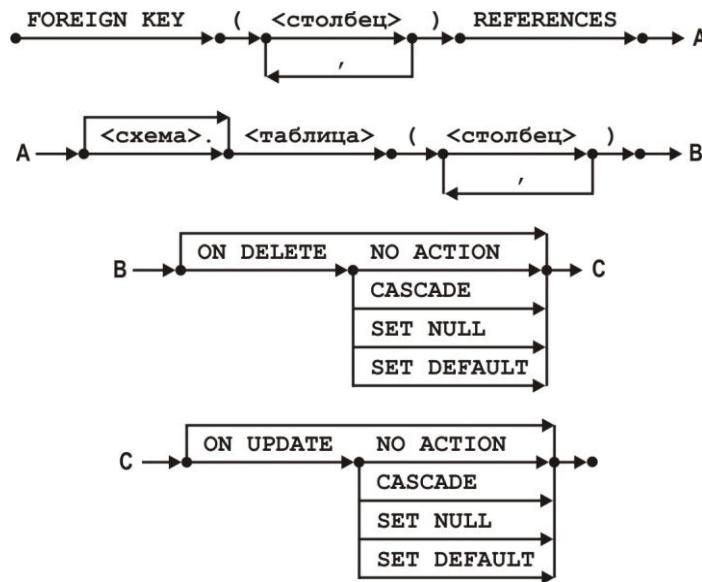
Граф 5.10. Окончание

Листинг 5.10. Синтаксис ограничения внешнего ключа на уровне таблицы

<ограничение внешнего ключа (таблица)> ::=

```

FOREIGN KEY (<столбец> [, <столбец>]...)
    REFERENCES [<схема>.]<таблица> [<столбец> [, <столбец>]...]
        [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
    
```



Граф 5.11. Синтаксис ограничения внешнего ключа на уровне таблицы

Рассмотрим основные отличия этих синтаксических конструкций.

- ◆ Для внешнего ключа на уровне столбца таблицы ключевые слова FOREIGN KEY могут отсутствовать. На уровне таблицы указание их обязательно.
- ◆ На уровне таблицы указание столбца, столбцов, входящих в состав внешнего ключа, является обязательным. В обоих случаях для родительской таблицы, на которую ссылается внешний ключ, можно не указывать список столбцов, если осуществляется ссылка на первичный и только первичный ключ этой таблицы.

Важным является то, что структура внешнего ключа и типы данных столбцов, входящих в его состав, у дочерней таблицы должны полностью совпадать (вплоть до

количества знаков в строковых столбцах) со структурой и типами данных столбцов соответствующего первичного или уникального ключа родительской таблицы.

Предложение **ON DELETE**

Предложение **ON DELETE** определяет, что должно произойти со строками дочерней таблицы (в которой описывается внешний ключ) при удалении соответствующей строки родительской таблицы. Можно задать один из следующих вариантов.

- ◆ **NO ACTION** — означает, что не должно быть выполнено никаких действий. Это значение по умолчанию. Если удаляется строка родительской таблицы, для которой существует некоторое количество подчиненных строк дочерней таблицы (значение внешнего ключа в этих строках совпадает со значением первичного или уникального ключа удаляемой строки родительской таблицы), то произойдет нарушение декларативной целостности данных в базе данных. Внешний ключ в этом варианте будет ссылаться на несуществующую строку родительской таблицы. В этом случае необходимо предусмотреть некоторые дополнительные действия для устранения нарушения целостности данных. Основной инструмент для восстановления целостности данных — триггеры.
- ◆ **CASCADE** — приводит к удалению всех соответствующих подчиненных строк дочерней таблицы, т. е. тех строк, которые имеют значение внешнего ключа, совпадающее со значением первичного/уникального ключа, на который ссылается этот внешний ключ, удаляемой родительской записи. Это, пожалуй, наиболее простой и естественный способ поддержания декларативной целостности.
- ◆ **SET NULL** — требует установить в значение **NULL** все столбцы внешнего ключа дочерней таблицы, совпадающие по значению с ключом удаляемой строки родительской таблицы. Тоже хорошее решение, если, несмотря на отсутствие соответствующей строки родительской таблицы, строки дочерней таблицы должны оставаться в базе данных. Такой пример мы рассмотрим в этой главе далее.
- ◆ **SET DEFAULT** — устанавливает в значение по умолчанию все столбцы внешнего ключа дочерней таблицы. Наблюдал на практике такое тоже довольно разумное решение по поддержанию целостности данных. В одной базе данных всегда существует строка родительской таблицы, у которой значение ключа равно значению по умолчанию для столбцов внешнего ключа дочерней таблицы.

Предложение **ON UPDATE**

Предложение **ON UPDATE** указывает, что происходит со строками дочерней таблицы, когда изменяется значение любого столбца, входящего в состав ключа родительской таблицы, на который ссылается внешний ключ дочерней таблицы. Варианты те же, что и в случае задания предложения **ON DELETE**.

- ◆ **NO ACTION** — означает, что не должно быть выполнено никаких действий. Значение по умолчанию. В этом случае необходимо предусмотреть некоторые дополнительные действия для поддержания целостности данных.
- ◆ **CASCADE** — приводит к изменению значений нужных ключевых реквизитов (реквизитов внешнего ключа) всех соответствующих подчиненных строк дочерней

таблицы, тех строк, которые имеют значение внешнего ключа, совпадающее со значением изменяемого первичного/уникального ключа родительской записи. Наиболее естественный вариант поведения системы.

- ◆ SET NULL — требует установить в значение NULL все столбцы внешнего ключа дочерней таблицы, совпадающие по значению с ключом соответствующей строки родительской таблицы.
- ◆ SET DEFAULT — устанавливает в значение по умолчанию все столбцы внешнего ключа дочерней таблицы.

5.1.3.4. Ограничение CHECK

Ограничение CHECK задает логическое условие, которое должно возвращать значение истина для того, чтобы в таблицу помещалась новая строка или чтобы были выполнены изменения данных существующей строки.

Синтаксис ограничения в первом приближении крайне простой, он один и тот же на уровне столбца и на уровне таблицы. На самом деле логическое выражение может быть достаточно сложным, очень сложным. В листинге 5.11 и соответствующем R-графе (граф 5.12) представлено задание синтаксиса в простом варианте.

Листинг 5.11. Синтаксис определения ограничения CHECK

```
<ограничение CHECK> ::=  
    CHECK (<логическое выражение>)
```



Граф 5.12. Синтаксис определения ограничения CHECK

Вообще говоря, никаких особых формальных требований к содержимому логического выражения не предъявляется. Вы, например, можете задать какое-либо условие, всегда возвращающее истину. Тогда реальных проверок на допустимость значения проводиться не будет. Или наоборот, можно задать тождественно ложное условие. В этом случае ни одна строка в таблицу не будет добавлена.

В реальности же в ограничении CHECK записываются условия, которым должны удовлетворять данные, помещаемые в строку таблицы, или замещающие существующие в строке значения.

В ограничении CHECK на уровне столбца логическое выражение должно относиться к текущему столбцу. В ограничении на уровне таблицы в логическом выражении могут присутствовать имена любых столбцов этой таблицы. Однако система допускает варианты задания ограничений CHECK и на уровне столбца, когда в логическом выражении используются имена любых столбцов данной таблицы.

Средства создания логических выражений в Transact-SQL весьма обширные. Подробно мы их рассмотрим, когда будем описывать предложение WHERE оператора

SELECT. В этой же главе чуть дальше приведем очень простой пример проверки значения столбца таблицы в ограничении CHECK.

5.1.4. Вычисляемые столбцы

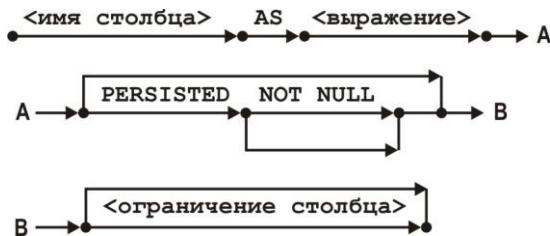
Вычисляемые столбцы в таблице — это столбцы, значения которых пользователь явно не помещает в строки таблицы. Эти значения автоматически вычисляются на основании выражения, заданного при определении столбца.

Сами вычисляемые столбцы и их значения могут храниться в таблице (быть "постоянными", PERSISTED) или их значения могут не присутствовать в базе данных, а вычисляться только при отображении данных таблицы.

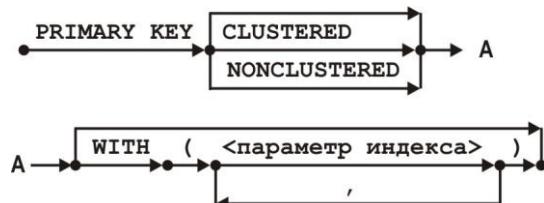
Синтаксис определения вычисляемого столбца показан в листинге 5.12 и в соответствующем R-графе (графы 5.13—5.18).

Листинг 5.12. Синтаксис определения вычисляемого столбца

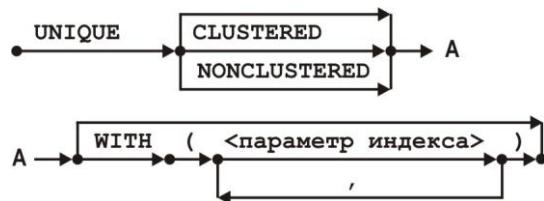
```
<вычисляемый столбец> ::=  
    <имя столбца> AS <выражение>  
    [ PERSISTED [ NOT NULL ] ]  
    [ <ограничение столбца> ]  
  
<ограничение столбца> ::=  
    [ CONSTRAINT <имя ограничения> ]  
    {  
        <первичный ключ>  
        | <уникальный ключ>  
        | <внешний ключ>  
        | <ограничение CHECK>  
    }  
  
<первичный ключ> ::=  
    PRIMARY KEY  
    [ CLUSTERED | NONCLUSTERED ]  
    [ WITH (<параметр индекса> [, <параметр индекса>] ...) ]  
  
<уникальный ключ> ::=  
    UNIQUE  
    [ CLUSTERED | NONCLUSTERED ]  
    [ WITH (<параметр индекса> [, <параметр индекса>] ...) ]  
  
<ограничение внешнего ключа> ::=  
    [ FOREIGN KEY ]  
    REFERENCES [<схема>.]<таблица> [(<столбец>) ]  
    [ ON DELETE { NO ACTION | CASCADE } ]  
    [ ON UPDATE NO ACTION ]  
  
<ограничение CHECK> ::=  
    CHECK (<логическое выражение>)
```



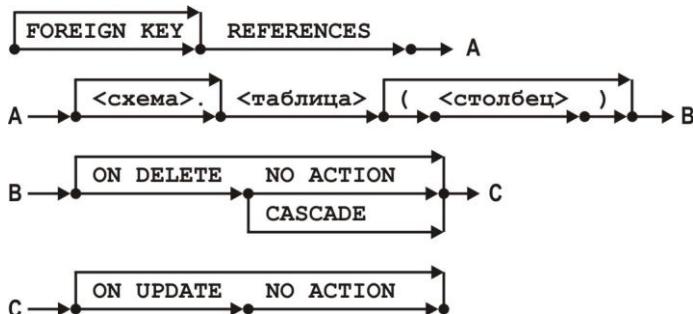
Граф 5.14. Синтаксис ограничения вычисляемого столбца



Граф 5.15. Ограничение первичного ключа вычисляемого столбца



Граф 5.16. Ограничение уникального ключа вычисляемого столбца



Граф 5.17. Ограничение внешнего ключа вычисляемого столбца



Граф 5.18. Ограничение CHECK вычисляемого столбца

Основные свойства вычисляемого столбца весьма похожи на свойства обычных столбцов. Главным моментом является задание способа получения значения такого столбца при помощи выражения после ключевого слова AS. Столбцы, на которые осуществляются ссылки в этом выражении, не должны быть вычисляемыми столбцами. Как правило, такое выражение задает значение столбца на основании значений других столбцов этой же строки таблицы, хотя бывают и более сложные случаи, которые мне не очень хочется здесь рассматривать.

Если вычисляемый столбец объявлен как постоянный (PERSISTED), то его данные будут физически помещаться в строку таблицы при создании новой строки и будут изменяться, если будут меняться значения исходных столбцов, которые принимают участие в формировании значения вычисляемого столбца. Постоянные вычисляемые столбцы могут входить в состав создаваемого индекса для таблицы. Основное требование к PERSISTED вычисляемым столбцам — значение столбца должно быть детерминированным, т. е. это значение не должно вычисляться при использовании функций, возвращающих случайные значения.

В определении внешнего ключа для вычисляемого столбца, как можно заметить, резко сокращены варианты реагирования на изменение и удаление соответствующей строки родительской таблицы. В случае удаления можно указать лишь NO ACTION и CASCADE, а при изменении значения ключевого реквизита родительской таблицы — только NO ACTION. Такое поведение системы понятно. Поскольку мы не можем напрямую изменять значение вычисляемого столбца, то при удалении строки родительской таблицы можно лишь удалить все строки подчиненной таблицы или не выполнять никаких действий. При изменении значения ключа родительской таблицы в принципе невозможны никакие действия со значением вычисляемого столбца дочерней таблицы.

5.1.5. Набор столбцов

Если в таблице присутствуют разреженные (PARSE) столбцы, то можно добавить столбец, который называют "набор столбцов" (column set). Синтаксис описания такого столбца:

```
<имя столбца> XML COLUMN_SET FOR ALL_SPARSE_COLUMNS
```

Такой столбец является "вычисляемым", данные не хранятся в таблице, а создаются при отображении значения столбца в XML-представлении. Таблица может иметь только один набор столбцов.

Использование набора столбцов может повысить производительность системы. Подробную информацию см. в Books Online. Там содержатся рекомендации по использованию и примеры добавления, изменения и выборки данных из наборов столбцов.

* * *

Давайте сейчас пока приостановим дальнейшее рассмотрение других конструкций в синтаксисе оператора CREATE TABLE и разберем примеры создания достаточно простых таблиц.

5.2. Простые примеры таблиц

Один из самых простых вариантов таблицы показан в примере 5.1.

Пример 5.1. Таблица, описывающая страны

```
USE BestDatabase;
GO
     /*** Справочник стран ***/
CREATE TABLE REFCTR
( CODCTR    CHAR(3) NOT NULL,      /* Код страны */
  NAME       VARCHAR(60),          /* Краткое название страны */
  FULLNAME  VARCHAR(65),          /* Полное название страны */
  CAPITAL    VARCHAR(30),          /* Название столицы */
  MAP        VARBINARY(MAX),      /* Карта страны */
  DESCRL     VARBINARY(MAX),      /* Дополнительное описание */
  CONSTRAINT PK_REFCTR
    PRIMARY KEY (CODCTR)
);
GO
```

Здесь код страны представлен строкой символов фиксированной длины. Этот столбец является первичным ключом. Задание ограничения первичного ключа выполняется на уровне таблицы в последней строке оператора:

```
CONSTRAINT PK_REFCTR PRIMARY KEY (CODCTR)
```

Так как в этом случае первичный ключ состоит из одного столбца, то это ограничение можно было бы указать и на уровне столбца:

```
CODCTR    CHAR(3) NOT NULL PRIMARY KEY, /* Код страны */
```

Здесь также можно указать и имя ограничения. Тогда задать первичный ключ на уровне столбца нужно в следующем виде:

```
CODCTR    CHAR(3) NOT NULL
CONSTRAINT PK_REFCTR PRIMARY KEY, /* Код страны */
```

Здесь задается и имя ограничения первичного ключа. Это имя будет присвоено индексу, который автоматически создаст система для первичного ключа.

Для других текстовых столбцов в этой таблице используется тип данных VARCHAR. Два последних столбца, карта страны и дополнительное описание, могут хранить графику и форматированные тексты. Они указаны с типом данных VARBINARY (MAX). Чуть позже в этой главе мы рассмотрим вариант использования файловых потоков

для хранения значений этих двух столбцов в файловой системе Windows, а не непосредственно в самой базе данных.

В следующем примере 5.2 создается таблица для хранения сведений о регионах стран. Здесь мы используем еще один стандартный прием, часто применяемый при создании таблиц. До создания таблицы мы проверим, существует ли такая таблица в нашей базе данных, и в случае ее существования, удалим. Для этих целей используется системное представление каталогов sys.tables, при помощи которого проверяется существование (функция EXISTS()) таблицы с именем REFREG.

Однако если в базе данных есть объекты, зависящие от данной таблицы, то такую таблицу удалить нельзя, пока не будут устраниены существующие зависимости — будет удален зависящий объект или будет удалена зависимость.

В данном случае от таблицы регионов зависит таблица районов (см. далее). В таблице районов внешний ключ ссылается на первичный ключ таблицы регионов.

Более подробно о зависимостях мы поговорим далее в разд. 5.7.1.

Пример 5.2. Таблица регионов

```
USE BestDatabase;
GO
      /*** Справочник регионов ***/
IF EXISTS (SELECT * FROM sys.tables
           WHERE NAME = 'REFREG')
    DROP TABLE REFREG;
GO
CREATE TABLE REFREG
( CODCTR  CHAR(3) NOT NULL,      /* Код страны */
  CODREG   CHAR(2) NOT NULL,      /* Код региона */
  NAME     VARCHAR(110),          /* Название региона */
  CENTER   VARCHAR(25),          /* Название центра региона */
  MAP      VARBINARY(MAX),       /* Карта региона */
  DESCR   VARBINARY(MAX),       /* Дополнительное описание */
  CONSTRAINT PK_REFREG PRIMARY KEY (CODCTR, CODREG),
  CONSTRAINT FK_REFREG
    FOREIGN KEY (CODCTR)
    REFERENCES REFCTR (CODCTR)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
GO
```

Здесь первичный ключ состоит из двух столбцов — кода страны и кода региона. Составной ключ можно определить только на уровне таблицы, но не на уровне столбца.

Внешний ключ, код страны, ссылается на первичный ключ родительской таблицы, справочника стран. Он определен на уровне таблицы.

Его также можно определить и на уровне столбца следующим образом:

```
CODCTR  CHAR(3) NOT NULL /* Код страны */
CONSTRAINT FK_REFREG
FOREIGN KEY REFERENCES REFCTR (CODCTR)
ON DELETE CASCADE
ON UPDATE CASCADE,
```

В обоих случаях после имени родительской таблицы можно было бы не указывать имя столбца, поскольку он является первичным ключом этой таблицы (чего делать, как вы помните, не следует). Предложение `ON DELETE CASCADE` указывает, что при удалении строки родительской таблицы справочника стран будут удалены и все относящиеся к этой стране регионы. Предложение `ON UPDATE` определяет, что при изменении значения первичного ключа в любой строке справочника стран также будут изменены значения внешнего ключа всех соответствующих строк справочника регионов. Здесь также указан вариант `CASCADE`.

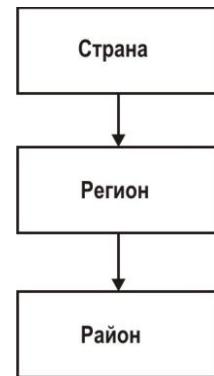
Наконец, для полноты ощущений в примере 5.3 показано создание таблицы районов. Все три таблицы (страны, регионы, районы) дают нам образец чистой трехуровневой иерархии, используемой для представления административно-территориального деления любой страны.

Пример 5.3. Таблица районов

```
USE BestDatabase;
GO
     /*** Справочник районов ***/
IF EXISTS (SELECT * FROM sys.tables
           WHERE NAME = 'REFAREA')
    DROP TABLE REFAREA;
GO
CREATE TABLE REFAREA
( CODCTR  CHAR(3) NOT NULL,      /* Код страны */
  CODREG   CHAR(2) NOT NULL,      /* Код региона */
  CODAREA  CHAR(3) NOT NULL,      /* Код района */
  NAME     VARCHAR(110),          /* Название района */
  CENTER   VARCHAR(50),           /* Название центра района */
  DESCRL   VARBINARY(MAX),        /* Дополнительное описание */
  CONSTRAINT PK_REFAREA
    PRIMARY KEY (CODCTR, CODREG, CODAREA),
  CONSTRAINT FK_REFAREA
    FOREIGN KEY (CODCTR, CODREG)
    REFERENCES REFREG (CODCTR, CODREG)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
GO
```

В этой таблице первичный и внешний ключи являются составными, поэтому их нужно объявлять только на уровне таблицы. Обратите внимание, что состав и характеристики столбцов, входящих во внешний ключ этой дочерней таблицы, полностью соответствуют характеристикам столбцов первичного ключа родительской таблицы.

Созданная иерархия показана на рис. 5.1.



ЗАМЕЧАНИЕ

В некоторых базах данных мне приходилось наблюдать, как разработчики в подобную структуру добавляют еще и связи районов со странами, задав в таблице районов еще один внешний ключ, ссылающийся на первичный ключ таблицы стран. Необходимости в такой дополнительной связи нет. Это никак не повысит производительность системы обработки данных. Скорее наоборот — это может ухудшить временные характеристики системы в первую очередь при добавлении в таблицу районов новых строк.

Следующая демонстрация. В примере 5.4 создается несколько упрощенная по структуре таблица, содержащая список персонала некоторой организации.

Пример 5.4. Таблица сотрудников организации

```

USE BestDatabase;
GO
/** Сотрудники организации */
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'STAFF')
    DROP TABLE STAFF;
GO
CREATE TABLE STAFF
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL,      /* Код сотрудника – первичный ключ */
  NAME1     VARCHAR(15),      /* Имя */
  NAME2     VARCHAR(15),      /* Отчество */
  NAME3     VARCHAR(20),      /* Фамилия */
  DUTIES    VARCHAR(40),      /* Должность */
  SALARY    DECIMAL(8, 2),      /* Оклад */
  FULLNAME AS NAME1 + ' ' +
              NAME2 + ' ' +
              NAME3,
  NET_SALARY AS (SALARY * .87),
  CONSTRAINT PK_STAFF
    PRIMARY KEY (COD)
);
GO
  
```

Если в предыдущих примерах таблицы имели обычные, "естественные", первичные ключи, для которых пользователь значения задавал вручную, то в данной таблице используется искусственный первичный ключ. Он описан с атрибутом `IDENTITY`. После этого ключевого слова в скобках указаны два значения, равные единице. Это означает, что первая помещенная в таблицу строка получит ключ со значением единица (первое значение в скобках), а при добавлении каждой новой строки система автоматически будет увеличивать последнее значение ключа на единицу (второе значение параметра в скобках) и присваивать это значение ключевому реквизиту.

В таблице присутствует два вычисляемых столбца — `FULLNAME` и `NET_SALARY`. Причем их значения не хранятся в базе данных, а "вычисляются", формируются при выборке данных из таблицы (в описании этих столбцов отсутствует ключевое слово `PERSISTED`).

Для столбца `NET_SALARY` вычисляется сумма заработной платы, получаемая сотрудником "на руки". Само вычисление простое. Нужно просто уменьшить размер начисленной суммы (столбец `SALARY`) на 13%. Если же будет введена прогрессивная шкала налогообложения, о чём часто говорят некоторые отечественные экономисты, точнее, политические деятели, то эта формула может оказаться много сложнее.

Столбец `FULLNAME` является конкатенацией (т. е. соединением) имени, отчества и фамилии сотрудника. Чтобы результат был читаемым, а все имена не сливались в единое слово, между элементами полного имени добавляются пробельные символы при использовании той же операции конкатенации.

Следует заметить, что эта таблица не является хорошим решением для большинства систем обработки данных реальной жизни. Здесь мы ее рассматриваем только в качестве простой иллюстрации создания таблиц. Чуть позже приведем примеры и более правильных решений.

Часто в задачах обработки данных нужно описывать различные организации и некоторые их характеристики. В примере 5.5 содержится создание таблицы организаций, которая может быть полезной и в реальных системах. Поскольку в организации задается и ее организационно-правовая форма, то до создания этой таблицы нужно создать справочник организационно-правовых форм.

Пример 5.5. Таблицы организаций и организационно-правовых форм

```
USE BestDatabase;
GO
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'ORGANIZATION')
    DROP TABLE ORGANIZATION;
GO
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'REFFORMORG')
    DROP TABLE REFFORMORG;
GO
```

```

    /*** Справочник организационно-правовых форм REFFORMORG ***/
CREATE TABLE REFFORMORG
( COD CHAR(2) NOT NULL,      /* Код организационно-правовой формы */
  NAME VARCHAR(120),          /* Название организационно-правовой формы */
  CONSTRAINT PK_REFFORMORG
    PRIMARY KEY (COD)
);
GO
    /*** Список организаций ***/
CREATE TABLE ORGANIZATION
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL,      /* Код организации */
  CODCTR      CHAR(3),      /* Код страны */
  CODREG      CHAR(2),      /* Код региона */
  CODAREA     CHAR(3),      /* Код района */
  LOCATION     CHAR(1) DEFAULT '0', /* Признак адреса:
                                         * 0 – региональный центр,
                                         * 1 – районный центр,
                                         * 2 – район. */
  ADDRESS      VARCHAR(60),   /* Адрес */
  NAME        VARCHAR(60),   /* Название организации */
  CODFORMORG CHAR(2) DEFAULT '00', /* Организационно-правовая форма */
  CONSTRAINT PK_ORGANIZATION PRIMARY KEY (COD),
  CONSTRAINT FK1_ORGANIZATION
    FOREIGN KEY (CODCTR, CODREG) REFERENCES REFREG (CODCTR, CODREG)
    ON DELETE SET NULL
    ON UPDATE CASCADE,
  CONSTRAINT FK2_ORGANIZATION
    FOREIGN KEY (CODFORMORG) REFERENCES REFFORMORG (COD)
    ON DELETE SET DEFAULT
    ON UPDATE CASCADE
);
GO

```

В таблице организаций также используется искусственный автоинкрементный первичный ключ при указании атрибута `IDENTITY`. Для кода организационно-правовой формы организации (столбец `CODFORMORG`) указывается значение по умолчанию (предложение `DEFAULT`) '`00`'. Этот столбец является внешним ключом, ссылающимся на справочник организационно-правовых форм. При описании внешнего ключа используется выражение `ON DELETE SET DEFAULT`, т. е. при удалении из справочника организационно-правовых форм соответствующей организационно-правовой формы столбцу будет установлено нулевое (не `NULL!`) значение. В справочнике организационно-правовых форм присутствует строка с этим кодом и названием такой формы "Неопределенная". Это один из способов сохранения целостности данных, о котором мы с вами уже говорили.

Подчеркну тот факт, что здесь присутствует отношение "один ко многим": одна организационно-правовая форма может быть у многих организаций. У одной орга-

низации есть только одна организационно-правовая форма. Напоминаю об этом, потому что по непонятной для меня причине это утверждение постоянно вызывает некоторые сомнения у моих студентов.

Для адресной части организации (или ее центрального офиса) в таблицу добавляются столбцы, содержащие код страны, код региона, код района. Надо сказать, что в некоторых случаях структуризация адреса не получается слишком уж простой. Приходится добавлять столбец `LOCATION`, который определяет уровень адресной части в иерархии административно-территориального деления страны, т. е. уточняет, находится ли организация в региональном центре или в районе. Здесь нельзя указать внешний ключ, который ссылается на таблицу районов, поскольку не все организации располагаются в районах регионов. Столбец `ADDRESS` будет содержать такие реквизиты, как название улицы, номер дома, корпуса, возможно, номер квартиры или офиса.

Обратите внимание, что в самом начале скрипта проверяется существование и удаление сначала таблицы организаций, а затем справочника. Здесь важен именно такой порядок, потому что таблица организаций "зависит" от таблицы справочника.

Теперь рассмотрим пример отношения "многие ко многим", применительно к организациям. Существует такое понятие, как *вид деятельности организации*. Вид деятельности несколько обобщенно определяет то, чем занимается организация. Например, видами деятельности могут быть производство космических ракет, выращивание огурцов, торговля оружием, разработка программ, издательская деятельность и многое, многое другое.

Ясно, что один и тот же вид деятельности может относиться ко многим организациям. В то же время одна организация может иметь несколько видов деятельности. Здесь мы имеем отношение "многие ко многим" в самом чистом виде. Такое отношение, как мы помним, реализуется добавлением в базу данных третьей, связующей, таблицы.

В примере 5.6 показано создание справочника видов деятельности и связующей таблицы для описания видов деятельности организаций. Таблица организаций описана в предыдущем примере.

Пример 5.6. Таблица видов деятельности и связующая таблица для организаций

```
USE BestDatabase;
GO
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'ORGACTIV')
    DROP TABLE ORGACTIV;
GO
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'REFACTIV')
    DROP TABLE REFACTIV;
GO
```

```
/** Справочник видов деятельности REFACTIV **/
CREATE TABLE REFACTIV
( COD      CHAR(4) NOT NULL,      /* Код вида деятельности */
  NAME     VARCHAR(110),          /* Наименование вида деятельности */
  CONSTRAINT PK_REFACTIV
    PRIMARY KEY (COD)           /* Первичный ключ */
);
GO
/** Виды деятельности организации **/
CREATE TABLE ORGACTIV
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL,      /* Код — первичный ключ */
  CODACT     CHAR(4),          /* Код вида деятельности */
  CODORG    INTEGER,          /* Код организации */
  TEXT      VARCHAR(110),        /* Описание вида деятельности */
  CONSTRAINT PK_ORGACTIV
    PRIMARY KEY (COD),          /* Первичный ключ */
  CONSTRAINT FK1_ORGACTIV
    FOREIGN KEY (CODACT) REFERENCES REFACTIV (COD)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  CONSTRAINT FK2_ORGACTIV
    FOREIGN KEY (CODORG) REFERENCES ORGANIZATION (COD)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
GO
```

Надо полагать, нам с вами все здесь понятно. Связующая таблица, помимо искусственного автоинкрементного первичного ключа, имеет два внешних ключа, один из которых ссылается на таблицу организаций, а другой — на таблицу (справочник) видов деятельности. Помимо этого, в ней присутствует также и некоторое дополнительное текстовое описание вида деятельности конкретной организации в виде столбца с типом данных `VARCHAR(110)`. Если требуется обширное описание чего-либо, то следует использовать столбец с типом данных `VARCHAR(MAX)`.

В самом начале скрипта проверяется существование в базе данных таблицы видов деятельности организации и удаление этой таблицы. Потом аналогичные действия выполняются со справочником видов деятельности. Здесь нужен именно такой порядок действий, поскольку от справочника видов деятельности зависит таблица видов деятельности организаций.

Диаграмма, показывающая эти отношения, представлена на рис. 5.2.

Аналогичным образом можно создать справочник товаров и описывать для каждой организации, какие товары она покупает, какие продает. Здесь будут присутствовать две связующие таблицы. Одна задает список покупаемых товаров, другая — продаваемых. Есть еще вариант использования только одной связующей таблицы,

в которой, помимо всего прочего, будет присутствовать признак, является товар покупаемым или продаваемым. В моей практике в различных разработках были использованы оба варианта. Я так и не оценил преимущества применения какого-либо из них.

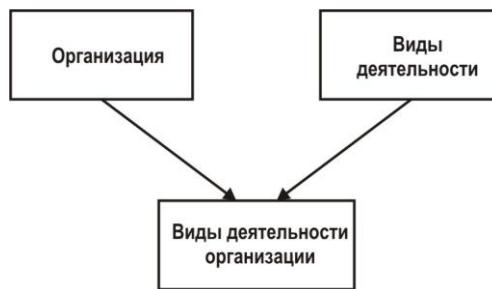


Рис. 5.2. Задание видов деятельности организаций

Теперь рассмотрим некоторые средства описания людей. Речь идет о таблице, хранящей сведения о людях. Ясно, что для такой таблицы нужен искусственный первичный ключ. Далее указываются основные характеристики человека: имя, пол, дата рождения и др.

Для некоторых систем обработки данных нужны сведения о родственных связях каждого человека. Вы не поверите, но это легко решается в рамках одной лишь таблицы, и нет необходимости использовать иерархический тип данных `HIERARCHYID`. Для этого в первую очередь нужны сведения о родителях человека (в реальных системах я еще указываю и сведения о супруге). Здесь не нужны дополнительные связующие таблицы. Все внешние ключи ссылаются на *ту же самую таблицу*, но, разумеется, на разные строки этой таблицы. На основании таких ссылок можно восстановить любые родственные отношения между различными людьми, правда, это потребует определенных действий при выборке данных.

Однако тут начинается одна грустная история. Итак, по порядку.

Несколько упрощенный (и неверный в текущей версии SQL Server) вариант задания таблицы, хранящей сведения о людях, представлен в примере 5.7.

Пример 5.7. Таблица людей (неверная в SQL Server 2012)

```

USE BestDatabase;
GO
**** Список людей ****/
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'PEOPLE')
    DROP TABLE PEOPLE;
GO
CREATE TABLE PEOPLE
( COD      INTEGER IDENTITY(1, 1)
            NOT NULL,      /* Код человека */

```

```
NAME1      VARCHAR (15),          /* Имя */
NAME2      VARCHAR (15),          /* Отчество */
NAME3      VARCHAR (20),          /* Фамилия */
BIRTHDAY   DATE,                /* Дата рождения */
SEX        CHAR(1) DEFAULT '0',  /* Пол: */
                  /* 0 — мужской, */
                  /* 1 — женский. */
FULLNAME   AS                   /* Вычисляемый столбец */
  (NAME3 + ' ' + NAME1 + ' ' + NAME2),
CODMOTHER  INTEGER
                  DEFAULT NULL,    /* Ссылка на мать */
CODEFATHER INTEGER
                  DEFAULT NULL,    /* Ссылка на отца */
CODOTHERHALF INTEGER
                  DEFAULT NULL,    /* Ссылка на супруга */
CONSTRAINT PK_PEOPLE PRIMARY KEY (COD),
CONSTRAINT CH_PEOPLE CHECK (SEX IN ('0', '1')),
CONSTRAINT FK1_PEOPLE
  FOREIGN KEY (CODMOTHER) REFERENCES PEOPLE (COD)
  ON DELETE SET NULL,
CONSTRAINT FK2_PEOPLE
  FOREIGN KEY (CODEFATHER) REFERENCES PEOPLE (COD)
  ON DELETE SET NULL,
CONSTRAINT FK3_PEOPLE
  FOREIGN KEY (CODOTHERHALF) REFERENCES PEOPLE (COD)
  ON DELETE SET NULL
);
GO
```

В принципе средства, используемые при создании этой таблицы, нам с вами хорошо известны. Неприятности возникают при описании внешних ключей, ссылающихся на мать, отца и супруга. Здесь внешние ключи ссылаются на первый ключ *той же самой* таблицы, но на различные ее строки.

Полужирным шрифтом выделены те предложения, которые не нравятся системе. Это конструкции

```
ON DELETE SET NULL
```

Мы получаем сообщение, что подобные указания могут вызвать зацикливание или многочисленные каскадные изменения в системе. Вообще-то наше описание весьма логично и не должно вызывать никакого зацикливания при удалении любой записи. Если из таблицы удаляется, например строка, описывающая мать человека, то самым естественным образом у ребенка (у всех детей) этой матери следовало бы установить в значение `NULL` ссылку на несуществующую уже в базе данных мать. Удалять записи детей нет необходимости, они могут использоваться в дальнейшей работе.

В попытках обмануть систему я явно указал для этих внешних ключей значение по умолчанию NULL (это видно в описании столбцов таблицы, хотя по логике вещей здесь предложение DEFAULT и не нужно) и скорректировал предложение ON DELETE:

```
ON DELETE SET DEFAULT
```

Обман не прошел. Система выдала то же самое сообщение.

Подобная структура у меня давно работает и работает в промышленных вариантах в системах InterBase и Firebird. Я сообщил представителям корпорации Microsoft об этом явно ошибочном поведении системы. В ответном очень добром письме они меня поблагодарили и сказали, что это будет исправлено в одном из ближайших релизах. Однако вышедшие версии SQL Server 2008, 2008 R2 и 2012 все равно выдают ту же самую ошибку.

По этой причине таблицу пришлось изменить, как показано в примере 5.8. Вместо предложения ON DELETE SET NULL задается ON DELETE NO ACTION.

Пример 5.8. Таблица людей (верная в SQL Server 2012)

```
USE BestDatabase;
GO
/** Список людей ***/
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'PEOPLE')
    DROP TABLE PEOPLE;
GO
CREATE TABLE PEOPLE
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL,      /* Код человека */
  NAME1     VARCHAR(15),      /* Имя */
  NAME2     VARCHAR(15),      /* Отчество */
  NAME3     VARCHAR(20),      /* Фамилия */
  BIRTHDAY  DATE,           /* Дата рождения */
  SEX       CHAR(1) DEFAULT '0', /* Пол: */
                /* 0 — мужской, */
                /* 1 — женский. */
  FULLNAME  AS              /* Вычисляемый столбец */
    (NAME3 + ' ' + NAME1 + ' ' + NAME2),
  CODMOTHER INTEGER
    DEFAULT NULL,          /* Ссылка на мать */
  CODEFATHER INTEGER
    DEFAULT NULL,          /* Ссылка на отца */
  CODOTHERHALF INTEGER
    DEFAULT NULL,          /* Ссылка на супруга */
  CONSTRAINT PK_PEOPLE PRIMARY KEY (COD),
  CONSTRAINT CH_PEOPLE CHECK (SEX IN ('0', '1')),
  CONSTRAINT FK1_PEOPLE
    FOREIGN KEY (CODMOTHER) REFERENCES PEOPLE (COD)
    ON DELETE NO ACTION,
```

```
CONSTRAINT FK2_PEOPLE
    FOREIGN KEY (CODFATHER) REFERENCES PEOPLE (COD)
        ON DELETE NO ACTION,
CONSTRAINT FK3_PEOPLE
    FOREIGN KEY (CODOITHERHALF) REFERENCES PEOPLE (COD)
        ON DELETE NO ACTION
);
GO
```

Задача сохранения целостности данных решаема, это можно выполнить при использовании триггеров, о которых нам предстоит разговор в главе 11.

Здесь мы также видим простой пример использования ограничения CHECK:

```
CONSTRAINT CH_PEOPLE CHECK (SEX IN ('0', '1')),
```

В логическом выражении ограничения используется конструкция `IN`. В скобках задается список значений. Выражение будет истинным, если значение, помещаемое в столбец, будет равно одному из значений, указанному в списке. Здесь нужно еще одно маленькое уточнение. Поскольку столбец `SEX` допускает и значение `NULL` (при описании столбца не задано `NOT NULL`), то при добавлении или изменении данных таблицы ему можно задать и значение `NULL`. То есть, столбцу могут быть присвоены значения '`0`', '`1`' и `NULL`.

Теперь давайте рассмотрим, как можно описать сотрудников для многих организаций, исходя из того, что один и тот же человек может одновременно работать (или просто числиться) в нескольких организациях.

В примере 5.5 было показано создание таблицы организаций (`ORGANIZATION`), в примере 5.8 — таблицы людей (`PEOPLE`). Чтобы описать сотрудников любой организации, нам нужна лишь третья связующая таблица. Создание такой таблицы показано в примере 5.9.

Пример 5.9. Таблица персонала организаций

```
USE BestDatabase;
GO
     *** Сотрудники организации ***
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'STAFF')
    DROP TABLE STAFF;
GO
CREATE TABLE STAFF
( COD      INTEGER IDENTITY(1, 1)
            NOT NULL,      /* Код сотрудника – первичный ключ */
    CODEPEOPLE INTEGER,          /* Код человека из списка людей */
    CODORG    INTEGER,          /* Код организации */
    DUTIES    VARCHAR(40),       /* Должность */
    SALARY    DECIMAL(8, 2),      /* Оклад */
```

```

NET_SALARY      AS (SALARY * .87),
CONSTRAINT PK_STAFF PRIMARY KEY (COD),
CONSTRAINT FK1_STAFF
    FOREIGN KEY (CODPEOPLE) REFERENCES PEOPLE (COD)
    ON DELETE CASCADE,
CONSTRAINT FK2_STAFF
    FOREIGN KEY (CODORG) REFERENCES ORGANIZATION (COD)
    ON DELETE CASCADE
);
GO

```

В таблице не описываются "паспортные" данные человека, как было сделано в примере 5.4. Просто для сотрудников организации был создан внешний ключ, который ссылается на таблицу людей, где и содержатся все необходимые подробные данные о человеке.

В результате мы получаем фрагмент базы данных, графическое представление которого можно проиллюстрировать рисунком 5.3.

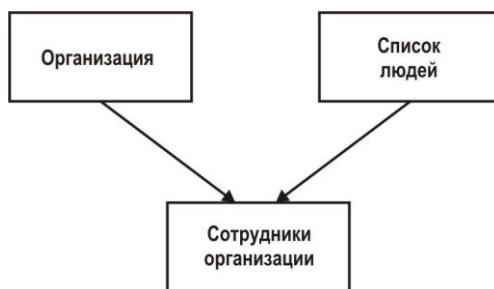


Рис. 5.3. Описание сотрудников организаций

В реальных системах, где нужны довольно подробные сведения о людях, часто используется и указание адресов (адрес регистрации и адрес фактического проживания). В этом случае в таблице людей задается также структурированная форма адресов, аналогичная тому, что помещается и в таблицу организаций, как мы видели в примере 5.5.

5.3. Создание секционированных таблиц

Секционирование таблиц позволяет на основании некоторого критерия размещать отдельные строки таблицы в различных файловых группах. Здесь достигается несколько положительных результатов. Во-первых, в одной файловой группе при правильном проектировании системы будут располагаться тесно связанные между собой данные, что во многих случаях может повысить производительность системы обработки данных.

Во-вторых, если какая-либо файловая группа по различным причинам станет недоступной для работы, функционирование системы все равно будет возможным, если программы будут обращаться лишь к файловым группам, к которым существует доступ. Это резко улучшает характеристику системы, которую обычно и называют *доступностью* (availability).

5.3.1. Синтаксические конструкции

Вернемся к листингу 5.1, где содержится описание синтаксиса оператора CREATE TABLE.

Необязательное предложение ON позволяет задать возможность разделения данных таблицы по разделам (выполнить секционирование таблицы). Напомню, синтаксис этого предложения выглядит следующим образом:

```
ON { <схема секционирования> (<разделяющий ключ>)
    | <файловая группа>
    | "default"
}
```

Это предложение определяет, где должны располагаться конкретные строки таблицы. Существует три варианта. Начнем с конца.

Если задано "default" или предложение ON не присутствует в операторе создания таблицы, то строки таблицы будут сохраняться в файловой группе по умолчанию. Размещение строк лежит полностью на совести системы. Обратите внимание, что здесь используется не ключевое слово языка, а несколько странная конструкция "default".

Если задано имя файловой группы, то все строки данной таблицы будут размещаться в файлах указанной файловой группы. Файловая группа с таким именем должна, разумеется, существовать в этой базе данных.

Более интересным и сложным является вариант, когда задаются схема секционирования (иногда ее называют схемой разделения или разделяющей схемой), функция секционирования (разделяющая функция) и разделяющий ключ.

```
ON <схема секционирования> (<разделяющий ключ>)
```

В этом случае можно указать, что строки таблицы в определенных диапазонах значений разделяющего ключа помещаются в файлы конкретных файловых групп. Критерии размещения задаются значением разделяющего ключа (этот ключ должен быть частью первичного или уникального ключа таблицы), явно заданной функцией секционирования (разделяющей функцией, в этом синтаксисе отсутствует, ее мы увидим позже) и созданной схемой секционирования (разделяющей схемой).

До использования схемы секционирования в операторе создания таблицы эту схему секционирования нужно создать в базе данных. Схемы секционирования не размещаются в отдельных схемах базы данных, они принадлежат всей базе данных.

Синтаксис оператора создания схемы секционирования CREATE PARTITION SCHEME представлен в листинге 5.13 и в соответствующем R-графе (граф 5.19).

Листинг 5.13. Синтаксис оператора создания схемы секционирования

```
CREATE PARTITION SCHEME <имя схемы секционирования>
AS PARTITION <имя функции секционирования>
[ ALL ] TO ( { <файловая группа> | PRIMARY }
[ , { <файловая группа> | PRIMARY } ] ... );
```



Граф 5.19. Синтаксис оператора создания схемы секционирования

Имя схемы секционирования должно быть уникальным в текущей базе данных.

ЗАМЕЧАНИЕ

Обратите внимание, что здесь используется термин *schema*, а не ставшее нам более привычным слово *schema*. Оба термина переводятся на русский язык как "схема". Для создания средств, описывающих секционирование, разделение данных таблицы, используется термин *schema*. Во втором случае речь идет об относительно самостоятельной части базы данных. Вообще слово *schema* было введено в терминологию баз данных комитетом CODASYL где-то в 60-е годы прошлого столетия. Правда, означал он совсем не то, что подразумевается под схемой базы данных в SQL Server.

В предложении **AS PARTITION** указывается имя предварительно созданной функции секционирования, синтаксис которой мы сейчас рассмотрим.

Далее перечисляются файловые группы.

Если задано ключевое слово **ALL**, то это означает, что все строки таблицы будут помещаться в указанную далее в скобках файловую группу (в указанные файловые группы, если в списке задано несколько имен файловых групп). Причем в этом случае можно в списке указать имя лишь одной файловой группы или задать **PRIMARY**, т. е. первичную файловую группу. Такое использование схемы секционирования полностью совпадает с вариантом задания в операторе создания таблицы предложения **ON** в следующем виде:

```
ON <файловая группа>
```

Далее в операторе создания схемы секционирования после ключевого слова **TO** в скобках перечисляются имена существующих в базе данных файловых групп, которые отделяются друг от друга запятыми. Одно и то же имя файловой группы может в списке встречаться несколько раз. Чтобы не создавать себе лишних приключений, следует обеспечить полное соответствие списка файловых групп тем

секциям, которые описываются в соответствующей функции секционирования, на которую указывает ссылка в операторе создания схемы секционирования.

Давайте сразу же рассмотрим средства удаления и изменения схемы секционирования. Чтобы удалить существующую схему секционирования, нужно использовать оператор `DROP PARTITION SCHEME`. Его синтаксис представлен в листинге 5.14 и соответствующем R-графе (граф 5.20).

Листинг 5.14. Синтаксис оператора удаления схемы секционирования

```
DROP PARTITION SCHEME <имя схемы секционирования>;
```



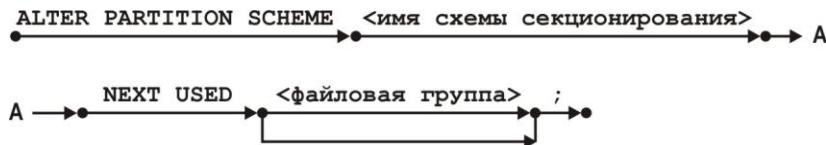
Граф 5.20. Синтаксис оператора удаления схемы секционирования

Чтобы вы могли удалить схему секционирования, на нее не должны ссылаться никакие объекты базы данных. Например, не должно быть таблиц, использующих ее имя в предложении `ON`. Перед удалением такой схемы необходимо устраниить в базе данных все ссылки на нее.

Для внесения изменений в существующую схему секционирования используется оператор `ALTER PARTITION SCHEME`. Его синтаксис показан в листинге 5.15 и соответствующем графе 5.21.

Листинг 5.15. Синтаксис оператора изменения схемы секционирования

```
ALTER PARTITION SCHEME <имя схемы секционирования>
NEXT USED [ <файловая группа> ] ;
```



Граф 5.21. Синтаксис оператора изменения схемы секционирования

Оператор позволяет добавить новую файловую группу в конец списка файловых групп схемы секционирования. Если в операторе не указана файловая группа и в схеме секционирования существует файловая группа с характеристикой `NEXT USED` (следующая), то у этой файловой группы снимается характеристика `NEXT USED`.

Теперь рассмотрим функции секционирования. Для реального выполнения задач секционирования схема секционирования обращается к функции секционирования.

Синтаксис оператора создания функции секционирования в текущей базе данных представлен в листинге 5.16 и в соответствующем R-графе (граф 5.22).

Листинг 5.16. Синтаксис оператора создания функции секционирования

```
CREATE PARTITION FUNCTION <имя функции секционирования>
    (<тип данных входного параметра>) AS RANGE [ LEFT | RIGHT ]
    FOR VALUES ([<значение> [, <значение>] ...]);
```



Граф 5.22. Синтаксис оператора создания функции секционирования

Имя функции должно быть уникальным в базе данных. После имени функции в скобках указывается тип данных входного параметра, параметра, который определяется ключом секционирования, т. е. столбцом, входящим в состав таблицы. Имя столбца (разделяющий ключ) указывается в предложении **ON** оператора создания таблицы. Здесь может быть указан системный тип данных или тип данных, определенный пользователем.

Предложение **AS RANGE** задает, какая операция сравнения будет использована для значения ключа. Значение **LEFT** в этом предложении означает, что будет использована операция "меньше или равно", т. е. в диапазон попадут строки, у которых разделяющий ключ меньше или равен указанной границе и больше предыдущего значения границы. Это значение по умолчанию. **RIGHT** означает, что диапазоны будут определяться операцией "меньше".

Предложение **FOR VALUES** задает список границ значений входного параметра. Если, например, в предложении **AS RANGE** указано **LEFT**, то каждая граница в списке определяет максимальное значение диапазона, которому должно соответствовать значение разделяющего ключа, чтобы попасть в соответствующий диапазон. Количество значений не должно превышать 999. По-хорошему, значения в списке должны быть упорядочены по возрастанию, однако если вы поленились выполнить соответствующее упорядочение, система это сделает за вас. Если список значений отсутствует, то все строки будут помещаться в единственный раздел, независимо от значения разделяющего ключа.

Для удаления функции секционирования используется оператор **DROP PARTITION FUNCTION** (листинг 5.17 и граф 5.23).

Листинг 5.17. Синтаксис оператора удаления функции секционирования

```
DROP PARTITION FUNCTION <имя функции секционирования>;
```



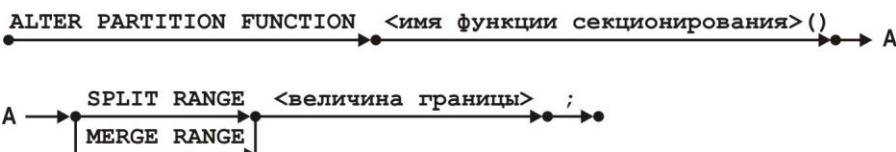
Функция секционирования может быть удалена только в том случае, если на нее не ссылается ни одна схема секционирования.

Для изменения функции секционирования применяется оператор ALTER PARTITION FUNCTION. Его синтаксис представлен в листинге 5.18 и в соответствующем графике 5.24.

Листинг 5.18. Синтаксис оператора изменения функции секционирования

```

ALTER PARTITION FUNCTION <имя функции секционирования>()
{ SPLIT RANGE | MERGE RANGE } <величина границы>;
    
```



Оператор позволяет разделить или объединить границы, применяемые к значениям разделяющего ключа. Обратите внимание, что по правилам синтаксиса после имени функции секционирования в этом операторе обязательно должны идти две круглые скобки () без задания каких-либо значений внутри.

Рассмотрим пример, который даст возможность проиллюстрировать использование всех только что описанных конструкций, и, надеюсь, будет иметь какое-то отношение к реальной жизни, к тем задачам, с которыми вы можете когда-нибудь столкнуться в этой самой реальной жизни.

5.3.2. Пример создания секционированной таблицы

Сейчас мы с вами выполним проектирование, создание и изменение всех необходимых средств для работы с секционированной таблицей.

Пусть существует международная организация, занимающаяся созданием и продажей различного программного продукта и аппаратных средств. Назовем ее, например, Hugehard. Эта организация имеет весьма большое количество филиалов, представительств по всей Земле. В филиалах, не покладая рук, с утра до позднего вечера, практически без выходных, активно трудится очень большое количество сотрудников. С целью упорядочения сведений о филиалах и их сотрудниках фирма создала центральные офисы в различных регионах земного шара, связав их с континентами (удачно или не очень). Центральные офисы располагаются в Америке (сюда входит Северная и Южная Америка), в Европе, в Азии, в Африке и в Австралии. Антарктида пока не присутствует в этом списке.

Каждому центральному офису был присвоен порядковый номер: Америке, разумеется, — 1, Европе — 2, Азии — 3, Африке — 4 и Австралии — 5.

С целью повышения производительности системы обработки данных по персоналу были использованы только что рассмотренные средства секционирования таблицы персонала. Все сведения по персоналу, относящемуся к одному центральному офису (а это множество региональных офисов с большим количеством сотрудников), было решено помещать в отдельную файловую группу. В соответствии с этим необходимо создать, по меньшей мере, пять файловых групп, в которых будут размещаться сведения о сотрудниках.

Давайте для этой организации выполним в правильном порядке все требуемые действия по созданию средств секционирования соответствующих данных, сведений по персоналу. Для начала создадим базу данных, которая будет содержать необходимые для хранения данных по сотрудникам файловые группы. Как мы установили, этих групп должно быть пять (помимо первичной файловой группы). Пусть каждая файловая группа будет содержать по одному файлу.

Сначала на диске D: любыми известными вам средствами создайте каталог для хранения всех файлов базы данных — Hugehard. Затем выполните оператор создания самой базы данных с тем же именем Hugehard. Оператор показан в примере 5.10.

Пример 5.10. Создание базы данных Hugehard

```
USE master;
GO
IF DB_ID('Hugehard') IS NOT NULL
    DROP DATABASE Hugehard;
GO
CREATE DATABASE Hugehard
ON
PRIMARY
( NAME = Hugehard,
  FILENAME = 'D:\Hugehard\Hugehard.mdf'),
FILEGROUP America
( NAME = America,
  FILENAME = 'D:\Hugehard\America.ndf'),
FILEGROUP Europe
( NAME = Europe,
  FILENAME = 'D:\Hugehard\Europe.ndf'),
FILEGROUP Asia
( NAME = Asia,
  FILENAME = 'D:\Hugehard\Asia.ndf'),
FILEGROUP Africa
( NAME = Africa,
  FILENAME = 'D:\Hugehard\Africa.ndf'),
```

```
FILEGROUP Australia
  ( NAME = Australia,
    FILENAME = 'D:\Hugehard\Australia.ndf')
LOG ON
  ( NAME = HugehardLog,
    FILENAME = 'D:\Hugehard\HugehardLog.ldf');
GO
```

Помимо первичной файловой группы PRIMARY, в которой предполагается размещать какие-то общие для всей системы данные, в базе данных создается еще пять вторичных файловых групп, которые будут использоваться только для хранения данных по персоналу организации в соответствии с местоположением офисов. Имена файловых групп, имена логических и физических файлов базы данных имеют, как вы заметили, весьма осмыслиенные значения.

ЗАМЕЧАНИЕ

В приведенном примере все файлы размещаются на одном диске и в одном и том же каталоге. Разумеется, это сделано лишь с целью иллюстрации использования средств секционирования таблицы и с возможностями ваших компьютеров, на которых вы захотите выполнять примеры этой книги. В реальной жизни все эти файловые группы по-хорошему должны размещаться на разных жестких дисках, именно на разных физических, а не только на логических дисках. Это повысит надежность всей системы. Файл первичной файловой группы и файл журнала транзакций также следует разместить на разных дисках.

Все характеристики файловых групп и файлов мы задаем здесь по умолчанию, потому что основная наша задача — лишь разобраться в возможностях секционирования.

Отобразите результаты создания базы данных Hugehard. Нам нужно отобразить характеристики файлов и файловых групп базы данных. В примере 5.11 используется системное представление sys.master_files для отображения некоторых характеристик файлов базы данных.

Пример 5.11. Отображение характеристик файлов базы данных Hugehard

```
USE Hugehard;
GO
SELECT CAST(file_id AS CHAR(2)) AS 'ID',
       CAST(type AS CHAR(5)) AS 'TypeN',
       CAST(type_desc AS CHAR(5)) AS 'Type',
       CAST(name AS CHAR (12)) AS 'Name',
       CAST(state_desc AS CHAR(7)) AS 'State',
       CAST(size AS CHAR(5)) AS 'Size',
       CAST(max_size AS CHAR(11)) AS 'MaxSize'
FROM sys.master_files
WHERE database_id = DB_ID('Hugehard');
GO
```

Результат:

ID	TypeN	Type	Name	State	Size	MaxSize
1	0	ROWS	Hugehard	ONLINE	520	-1
2	1	LOG	HugehardLog	ONLINE	128	268435456
3	0	ROWS	America	ONLINE	128	-1
4	0	ROWS	Europe	ONLINE	128	-1
5	0	ROWS	Asia	ONLINE	128	-1
6	0	ROWS	Africa	ONLINE	128	-1
7	0	ROWS	Australia	ONLINE	128	-1

(7 row(s) affected)

Если вы хотите уточнить, что означают некоторые столбцы этого представления, обратитесь к *главе 3*, где все достаточно подробно расписано, хотя, по-моему, и так все понятно.

В примере 5.12 используется системное представление `sys.filegroups` для отображения файловых групп базы данных `Hugehard`.

Пример 5.12. Отображение файловых групп базы данных Hugehard

```
USE Hugehard;
GO
SELECT CAST(name AS CHAR (10)) AS 'Name',
       CAST(type AS CHAR(4)) AS 'Type',
       CAST(type_desc AS CHAR(15)) AS 'Type Text',
       CAST(is_read_only AS CHAR(7)) AS 'Read Only'
FROM sys.filegroups;
GO
```

Результат:

Name	Type	Type Text	Read Only
PRIMARY	FG	ROWS_FILEGROUP	0
America	FG	ROWS_FILEGROUP	0
Europe	FG	ROWS_FILEGROUP	0
Asia	FG	ROWS_FILEGROUP	0
Africa	FG	ROWS_FILEGROUP	0
Australia	FG	ROWS_FILEGROUP	0

(6 row(s) affected)

Мы видим, что помимо первичной файловой группы в базе данных присутствует еще пять нужных нам файловых групп. Причем все они допускают как операции чтения, так и записи (значение поля `Read Only = 0`).

Теперь вроде бы нужно создать таблицу персонала, однако в реальности нам придется выполнить все необходимые три действия в обратном порядке: вначале создать функцию секционирования, затем схему секционирования и только после этого саму таблицу.

Оператор создания таблицы сотрудников организации показан в примере 5.13. Не выполняйте пока этот оператор.

Пример 5.13. Таблица сотрудников организации Hugehard

```
USE Hugehard;
GO
/** Сотрудники организации Hugehard ***/
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'STAFF')
    DROP TABLE STAFF;
GO
CREATE TABLE STAFF
( COD      INTEGER IDENTITY(1, 1)
    NOT NULL, /* Код сотрудника – первичный ключ */
  REGION   CHAR(1) NOT NULL, /* Ключ секционирования */
  NAME1    VARCHAR(15), /* Имя */
  NAME2    VARCHAR(15), /* Отчество */
  NAME3    VARCHAR(20), /* Фамилия */
  DUTIES   VARCHAR(40), /* Должность */
  SALARY   DECIMAL(8, 2), /* Оклад */
  CONSTRAINT PK_STAFF PRIMARY KEY (REGION, COD),
  CONSTRAINT CH_STAFF
    CHECK (REGION IN ('1', '2', '3', '4', '5'))
)
ON SchemeHugehard (REGION);
GO
```

Ссылка на схему секционирования присутствует в последней строке оператора создания таблицы:

```
ON SchemeHugehard (REGION)
```

При попытке сейчас выполнить этот оператор вы получите сообщение, что не существует схемы секционирования SchemeHugehard.

Как мы с вами ранее установили, такая структура таблицы не является достаточно умной. Однако в данном случае это подходящее решение, потому как та самая фирма категорически запрещает ее сотрудникам работать еще в каких-либо других организациях, так что все характеристики сотрудника можно описывать именно в этой одной таблице.

Здесь в состав первичного ключа нам пришлось ввести как его часть и столбец REGION. Система секционирования требует, чтобы разделяющий ключ был частью

первичного или уникального ключа. Для этого столбца мы задали и ограничение CHECK, которое позволяет помещать в столбец только указанные значения, соответствующие существующим регионам.

Предложение ON ссылается на схему секционирования SchemeHugehard. В качестве разделяющего ключа указывается столбец REGION.

Для создания нужной схемы секционирования можно использовать операторы примера 5.14. Эти операторы пока опять же выполнять нельзя, поскольку еще не создана функция секционирования.

Пример 5.14. Схема секционирования для таблицы персонала организации Hugehard

```
USE Hugehard;
GO
CREATE PARTITION SCHEME SchemeHugehard
    AS PARTITION FunctionHugehard
    TO (America, Europe, Asia, Africa, Australia);
GO
```

Здесь указывается имя функции секционирования FunctionHugehard и перечисляются имена файловых групп, которые участвуют в процессе секционирования строк нашей таблицы.

Наконец, функция секционирования создается при выполнении операторов, показанных в примере 5.15.

Пример 5.15. Функция секционирования для схемы секционирования, распределяющей строки таблицы персонала организации Hugehard

```
USE Hugehard;
GO
CREATE PARTITION FUNCTION FunctionHugehard (CHAR(1))
    AS RANGE LEFT
    FOR VALUES ('1', '2', '3', '4');
GO
```

После имени функции в скобках задается тип данных столбца, который участвует в распределении строк таблицы по файловым группам. В нашей таблице персонала таким столбцом является REGION, который имеет тип данных CHAR(1). Именно этот тип данных и указан при создании функции секционирования.

В предложении AS RANGE LEFT задано, что файловые группы выбираются слева направо в зависимости от указанного значения разделяющего ключа.

В предложении FOR VALUES перечисляются границы значений разделяющего ключа. Здесь давайте остановимся и подробнее рассмотрим всю картину секционирования строк таблицы. В предложении FOR VALUES указано *четыре* значения:

```
FOR VALUES ('1', '2', '3', '4');
```

При создании схемы секционирования было задано пять имен файловых групп:

TO (America, Europe, Asia, Africa, Australia);

В табл. 5.1 показано, где будут сохраняться строки таблицы в зависимости от значения разделяющего ключа (столбец REGION).

Таблица 5.1. Распределение строк таблицы персонала по файловым группам

Значение разделяющего ключа	Целевая файловая группа
REGION <= '1'	America
REGION > '1' AND REGION <= '2'	Europe
REGION > '2' AND REGION <= '3'	Asia
REGION > '3' AND REGION <= '4'	Africa
REGION > '4'	Australia

Теперь, чтобы все это заработало, можно выполнить последовательно создание функции секционирования (см. пример 5.15), создание схемы секционирования (см. пример 5.14) и создание таблицы сотрудников организации (см. пример 5.13).

Пожалуй, более правильный вариант создания всех объектов базы данных показан в примере 5.16. Здесь все выполняется в нужном порядке и при выполнении всех подходящих проверок существования объектов базы данных.

Пример 5.16. Создание всех объектов для секционированной таблицы

```
USE Hugehard;
GO
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'STAFF')
    DROP TABLE STAFF;
GO
IF EXISTS (SELECT * FROM sys.partition_schemes
            WHERE name = 'SchemeHugehard')
    DROP PARTITION SCHEME SchemeHugehard;
GO
IF EXISTS (SELECT * FROM sys.partition_functions
            WHERE name = 'FunctionHugehard')
    DROP PARTITION FUNCTION FunctionHugehard;
GO
CREATE PARTITION FUNCTION FunctionHugehard (CHAR(1))
    AS RANGE LEFT
    FOR VALUES ('1', '2', '3', '4');
GO
CREATE PARTITION SCHEME SchemeHugehard
    AS PARTITION FunctionHugehard
    TO (America, Europe, Asia, Africa, Australia);
```

GO

```

    /*** Сотрудники организации Hugehard ***/
CREATE TABLE STAFF
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL, /* Код сотрудника — первичный ключ */
REGION   CHAR(1) NOT NULL, /* Ключ секционирования */
NAME1    VARCHAR(15),      /* Имя */
NAME2    VARCHAR(15),      /* Отчество */
NAME3    VARCHAR(20),      /* Фамилия */
DUTIES   VARCHAR(40),      /* Должность */
SALARY   DECIMAL(8, 2),    /* Оклад */
CONSTRAINT PK_STAFF PRIMARY KEY (REGION, COD),
CONSTRAINT CH_STAFF
  CHECK (REGION IN ('1', '2', '3', '4', '5'))
)
ON SchemeHugehard (REGION);
GO

```

Посмотрите на операторы примера. Здесь важен порядок, в котором выполняются проверки и удаления существующих объектов. Например, прежде чем удалять схему и функцию секционирования, нужно вначале удалить таблицу, которая их использует.

Для проверки существования в базе данных схемы секционирования используется представление просмотра каталогов `sys.partition_schemes`, а для проверки существования функции секционирования — представление `sys.partition_functions`, при вызове которых указываются имена соответствующих объектов базы данных.

Эти представления вы можете использовать для отображения списка и характеристик схем секционирования и функций секционирования в базе данных.

Операторы примера 5.16 можно выполнять многократно. Каждый раз в нужной последовательности будут удаляться и создаваться требуемые объекты.

Чтобы проверить, как выполняется распределение данных по файловым группам, заполним таблицу сотрудников некоторыми данными. Соответствующий скрипт приведен в примере 5.17.

Пример 5.17. Заполнение данными таблицы сотрудников

```

USE Hugehard;
GO
SET NOCOUNT ON;
DECLARE @N INT = 1;
-- Регион America
WHILE @N <= 20
BEGIN
  INSERT INTO STAFF (REGION, NAME3)
  VALUES ('1', 'America ' + CAST(@N AS CHAR(2)));

```

```
        SET @N += 1;
END;
-- Регион Europe
SET @N = 1;
WHILE @N <= 20
BEGIN
    INSERT INTO STAFF (REGION, NAME3)
    VALUES ('2', 'Europe ' + CAST(@N AS CHAR(2)));
    SET @N += 1;
END;
-- Регион Asia
SET @N = 1;
WHILE @N <= 20
BEGIN
    INSERT INTO STAFF (REGION, NAME3)
    VALUES ('3', 'Asia ' + CAST(@N AS CHAR(2)));
    SET @N += 1;
END;
-- Регион Africa
SET @N = 1;
WHILE @N <= 20
BEGIN
    INSERT INTO STAFF (REGION, NAME3)
    VALUES ('4', 'Africa ' + CAST(@N AS CHAR(2)));
    SET @N += 1;
END;
-- Регион Australia
SET @N = 1;
WHILE @N <= 20
BEGIN
    INSERT INTO STAFF (REGION, NAME3)
    VALUES ('5', 'Australia ' + CAST(@N AS CHAR(2)));
    SET @N += 1;
END;
GO
SELECT COD, REGION, NAME3
    FROM STAFF;
GO
SET NOCOUNT OFF;
```

Здесь объявляется целочисленная локальная переменная @N. Она используется в качестве параметра циклов заполнения таблицы. В одном цикле помещаются строки в отдельную файловую группу. В каждой файловой группе размещается по 20 строк таблицы персонала. Не сильно напрягая воображение, здесь помимо значения региона в таблицу мы помещали значение столбца NAME3, которое состояло из имени региона и порядкового номера сотрудника в этом регионе.

В результате отображения содержимого таблицы мы видим, что она содержит 100 строк. Вот первый десяток отображаемых строк:

COD	REGION	NAME3
1	1	America 1
2	1	America 2
3	1	America 3
4	1	America 4
5	1	America 5
6	1	America 6
7	1	America 7
8	1	America 8
9	1	America 9
10	1	America 10
...		

5.3.3. Отображение результатов создания таблицы

В Management Studio можно просмотреть результаты создания таблиц, схем секционирования, функций секционирования и многое другое.

В окне **Object Explorer** раскройте базу данных **Hugehard**, раскройте папку таблиц (**Tables**), раскройте созданную таблицу **dbo.STAFF**.

Если раскрыть список столбцов этой таблицы (папка **Columns**), то можно увидеть список всех ее столбцов с основными их характеристиками.

Раскрыв папку **Keys**, можно увидеть список всех ключей таблицы. В нашем случае это только первичный ключ **PK_STAFF**.

Если раскрыть и папку **Indexes**, то мы получаем список всех индексов, явно или неявно созданных для данной таблицы. В нашем случае система создала индекс для первичного ключа таблицы.

При раскрытии папки **Constraints** мы видим имя ограничения таблицы. Это ограничение на значение столбца **REGION**.

Если раскрыть папку **Storage**, а затем **Partition Schemes**, то в окне будут показаны схемы секционирования базы данных. Если же раскрыть папку **Partition Functions**, то можно увидеть список функций секционирования.

Некоторые из этих характеристик представлены на рис. 5.4.

ЗАМЕЧАНИЕ

Похожие средства секционирования используются не только для строк таблиц, но также и для индексов.

Здесь же можно получить и текстовое представление о структуре любой из таблиц базы данных. Для этого щелкните правой кнопкой мыши по имени таблицы

dbo.STAFF. В появившемся контекстном меню выберите **Script Table as | CREATE To | New Query Editor Window**. В новом окне появится скрипт создания таблицы **STAFF**. Фрагмент этого скрипта представлен в примере 5.18.

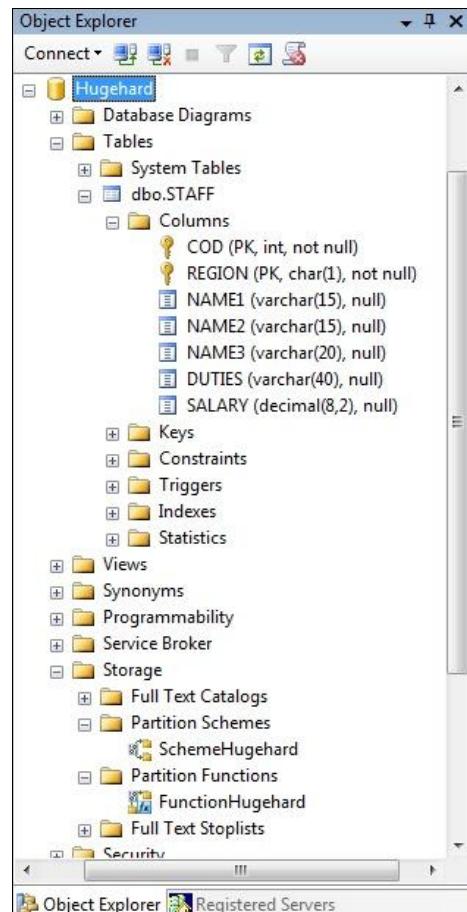


Рис. 5.4. Отображение характеристик таблицы в окне Object Explorer Management Studio

Пример 5.18. Фрагмент скрипта создания таблицы сотрудников

```

CREATE TABLE [dbo].[STAFF] (
    [COD] [int] IDENTITY(1,1) NOT NULL,
    [REGION] [char](1) NOT NULL,
    [NAME1] [varchar](15) NULL,
    [NAME2] [varchar](15) NULL,
    [NAME3] [varchar](20) NULL,
    [DUTIES] [varchar](40) NULL,
    [SALARY] [decimal](8, 2) NULL,
    CONSTRAINT [PK_STAFF] PRIMARY KEY CLUSTERED
(
    [REGION] ASC,
    [COD] ASC
)WITH (PAD_INDEX = OFF,
       STATISTICS_NORECOMPUTE = OFF,
       IGNORE_DUP_KEY = OFF,

```

```
ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON)
)

...
ALTER TABLE [dbo].[STAFF] WITH CHECK ADD CONSTRAINT [CH_STAFF]
CHECK (([REGION]='5' OR
        [REGION]='4' OR
        [REGION]='3' OR
        [REGION]='2' OR
        [REGION]='1'))
...

GO
```

В этом скрипте помимо структуры таблицы подробно описываются и характеристики создаваемого индекса для первичного ключа.

Здесь также можно увидеть, как реализуется ограничение `CHECK`.

При этом тот факт, что таблица является секционированной, тут увидеть нельзя.

Чтобы выяснить и наличие секционированности таблицы, щелкните правой кнопкой по имени таблицы и в контекстном меню выберите строку **Properties**. Появится окно, отображающее некоторые свойства таблицы. В левой верхней части этого окна щелкните мышью по строке **Storage**. Теперь в основном окне будут отображаться характеристики, связанные с хранением таблицы. В частности, указано, что таблица является секционированной (**Table is partitioned** имеет значение **True**), отображено имя схемы секционирования (**Partition scheme**), количество разделов секционирования (**Number of partitions**) и разделяющий ключ (**Partition column**) (рис. 5.5).

Чтобы увидеть список схем секционирования и функций секционирования, в окне **Object Explorer** раскройте папку **Storage**, далее раскройте папки **Partition Schemes** и **Partition Functions** (рис. 5.6).

Чтобы просмотреть исходный текст схемы, нужно щелкнуть правой кнопкой по имени схемы и в контекстном меню выбрать **Script Partition Scheme as | CREATE To | New Query Editor Window**. В новом окне появится набор операторов для создания этой схемы секционирования (показаны в примере 5.19).

Аналогичным образом можно получить текст функции секционирования, если щелкнуть правой кнопкой мыши по имени функции и в контекстном меню выбрать **Script Partition Function as | CREATE To | New Query Editor Window**.

Полученные тексты создания схемы секционирования и функции секционирования, как мы видим, мало чем отличаются от тех скриптов, которые мы выполняли. Посмотрите тексты сгенерированных системой скриптов в примере 5.19 (здесь два скрипта объединены в один).

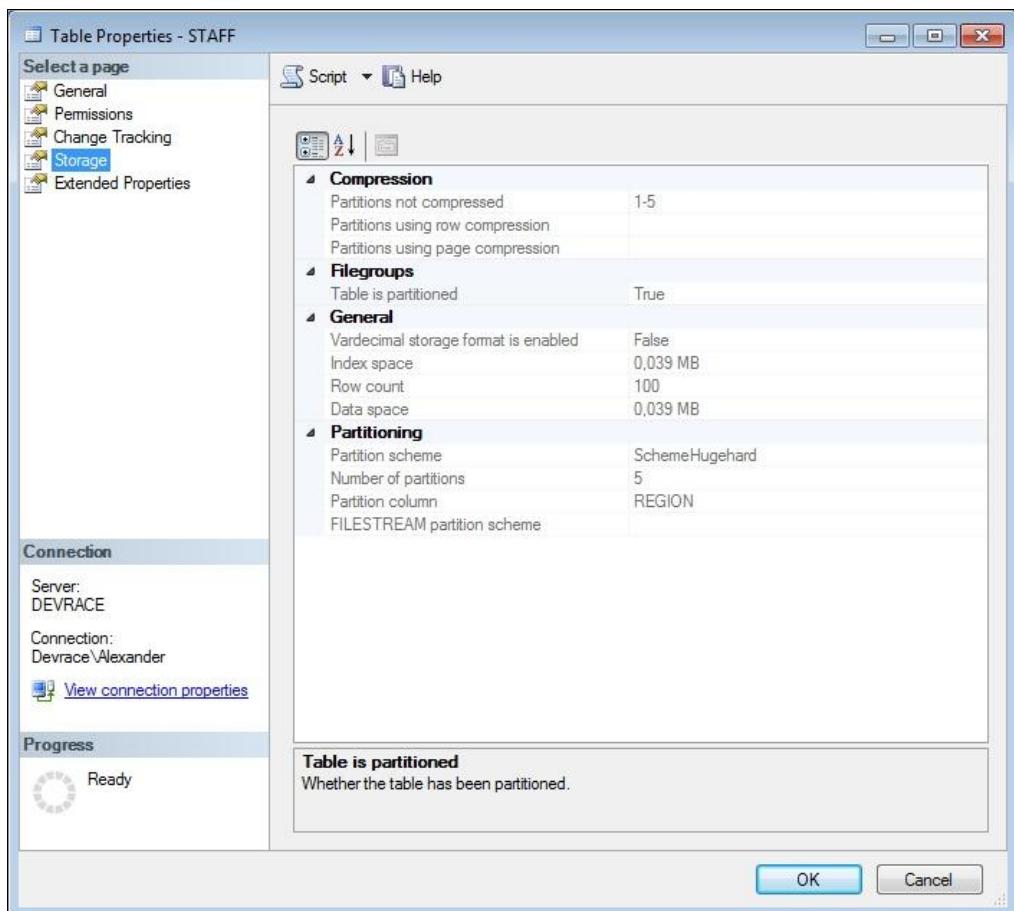


Рис. 5.5. Отображение характеристик секционированной таблицы в Management Studio

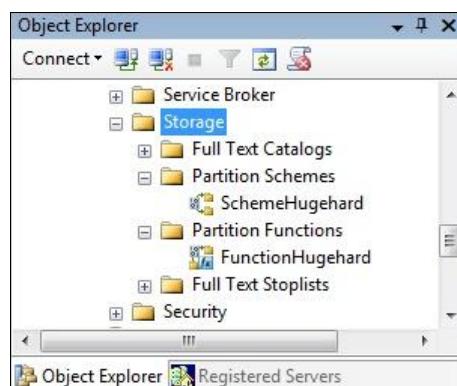


Рис. 5.6. Список схем и функций секционирования в окне Object Explorer

Пример 5.19. Сгенерированные скрипты создания схемы и функции секционирования таблиц

```
-- Функция секционирования
USE [Hugehard]
GO
/***** Object: PartitionFunction [FunctionHugehard]
    Script Date: 12/11/2011 23:06:22 *****/
CREATE PARTITION FUNCTION [FunctionHugehard] (char(1))
    AS RANGE LEFT FOR VALUES (N'1', N'2', N'3', N'4')
GO
-- Схема секционирования
USE [Hugehard]
GO
/***** Object: PartitionScheme [SchemeHugehard]
    Script Date: 12/11/2011 23:05:52 *****/
CREATE PARTITION SCHEME [SchemeHugehard]
    AS PARTITION [FunctionHugehard] TO
    ([America], [Europe], [Asia], [Africa], [Australia])
GO
```

5.3.4. Изменение характеристик секционированной таблицы

Теперь внесем необходимые изменения во все объекты базы данных, связанные с секционированной таблицей STAFF. Дело в том, что организация Hugehard теперь решила открыть представительства и в Антарктиде. По этой причине появилась потребность выполнить некоторые изменения в базе данных.

В первую очередь в базу данных нужно добавить новую файловую группу, в которую поместить один файл. В примере 5.20 представлены операторы, осуществляющие нужные действия. Выполните этот пакет.

Пример 5.20. Создание новой файловой группы и файла для базы данных Hugehard

```
USE master;
GO
ALTER DATABASE Hugehard
    ADD FILEGROUP Antarctica;
GO
ALTER DATABASE Hugehard
    ADD FILE
    ( NAME = Antarctica,
        FILENAME = 'D:\Hugehard\Antarctica.ndf')
        TO FILEGROUP Antarctica;
GO
```

Поскольку в систему был добавлен еще один регион, следует изменить ограничение CHECK в таблице STAFF, добавив в список допустимых значений столбца REGION и строкового значения 6, которое будет соответствовать региону Антарктида. Для внесения изменений в такое ограничение нужно сначала удалить существующее ограничение таблицы, а затем добавить новое. Выполните скрипт из примера 5.21.

Пример 5.21. Изменение ограничения CHECK для таблицы STAFF

```
USE Hugehard;
GO
ALTER TABLE STAFF
    DROP CONSTRAINT [CH_STAFF];

ALTER TABLE STAFF
    ADD CONSTRAINT [CH_STAFF] CHECK
        (([REGION]='6' OR
        [REGION]='5' OR
        [REGION]='4' OR
        [REGION]='3' OR
        [REGION]='2' OR
        [REGION]='1'));
```

GO

Обратите внимание, здесь мы задали ограничение в виде, отличающемся от того, что вначале было использовано для таблицы STAFF. Там мы применяли конструкцию IN. Смысл ограничения остался точно таким же.

Теперь нужно изменить схему секционирования и функцию секционирования. Изменения должны выполняться именно в таком порядке. Выполните скрипт из примера 5.22.

Пример 5.22. Внесение изменений в схему секционирования и функцию секционирования

```
USE Hugehard;
GO
ALTER PARTITION SCHEME SchemeHugehard
    NEXT USED Antarctica;

ALTER PARTITION FUNCTION FunctionHugehard ()
    SPLIT RANGE ('5');

GO
```

В первом операторе в существующую схему секционирования добавляется еще одна файловая группа. Второй оператор добавляет в функцию секционирования новое значение разделяющего ключа.

Добавьте в таблицу персонала и сведения об антарктических ребятах, выполнив операторы примера 5.23.

Пример 5.23. Добавление новых данных в таблицу сотрудников

```
USE Hugehard;
GO
DECLARE @N INT = 1;
-- Регион Antarctica
WHILE @N <= 20
BEGIN
    INSERT INTO STAFF (REGION, NAME3)
    VALUES ('6', 'Antarctica ' + CAST(@N AS CHAR(2)));
    SET @N += 1;
END;
GO
SELECT COD, REGION, NAME3
    FROM STAFF;
GO
```

В результате отображения содержимого таблицы персонала мы видим, что теперь таблица содержит 120 записей.

* * *

Вы заметили мою привычку создавать объекты базы данных и саму базу данных при использовании, в первую очередь, средств языка Transact-SQL. И дело здесь не столько в личных пристрастиях, сколько в прагматических аспектах нелегкой жизни разработчика программного обеспечения. Наличие сохраненных операторов создания этих объектов, тем более когда они еще и документированы при использовании хорошо написанных комментариев, позволяет достаточно легко вносить необходимые в процессе разработки изменения, как в сам процесс разработки, так и в результат документирования этой разработки.

При этом существуют и диалоговые средства создания объектов базы данных, которые можно использовать в процессе работы с Management Studio. Иногда при создании так называемых "макетных" проектов бывает довольно удобным использование именно диалоговых средств для создания небольших по объему объектов базы данных исключительно для целей исследования.

5.4. Создание таблиц диалоговыми средствами

5.4.1. Создание таблицы секционирования

Создадим новую таблицу в базе данных Hugehard. В окне **Object Explorer** раскройте базу данных **Hugehard**. Щелкните правой кнопкой мыши по папке **Tables** и в контекстном меню выберите элемент **New Table**. Появится окно, где можно описывать столбцы создаваемой таблицы и их характеристики (рис. 5.7).

Column Name	Data Type	Allow Nulls
▶		<input type="checkbox"/>

Рис. 5.7. Описание столбцов создаваемой таблицы

Создадим таблицу, аналогичную таблице сотрудников STAFF с полным повторением ее структуры и секционированности и назовем ее STAFF1.

В поле **Column Name** введите имя столбца COD. Из раскрывающегося списка **Data Type** выберите тип данных int. Снимите отметку в поле **Allow Nulls**, чтобы не допустить помещение в столбец значения NULL, поскольку он будет входить в состав первичного ключа создаваемой таблицы.

Теперь нужно установить для столбца свойство **IDENTITY**. В нижней части формы, в окне **Column Properties** (рис. 5.8), раскройте свойство **Identity Specification**, из раскрывающегося списка подсвойства (**Is Identity**) выберите Yes. Начальное значение (поле **Identity Seed**) и величина приращения (поле **Identity Increment**) уже установлены в единицу, как и в случае описания таблицы STAFF, что нас вполне устраивает. Так что здесь ничего менять не будем.

Рис. 5.8. Характеристики столбца COD создаваемой таблицы

Column Name	Data Type	Allow Nulls
COD	int	<input type="checkbox"/>
REGION	char(1)	<input type="checkbox"/>
NAME1	varchar(15)	<input checked="" type="checkbox"/>
NAME2	varchar(15)	<input checked="" type="checkbox"/>
NAME3	varchar(20)	<input checked="" type="checkbox"/>
DUTIES	varchar(40)	<input checked="" type="checkbox"/>
▶ SALARY	decimal(8, 2)	<input checked="" type="checkbox"/>

Рис. 5.9. Список столбцов создаваемой таблицы

Создайте описания остальных столбцов таблицы. Тип данных выбирается из раскрывающегося списка. Там, где нужно указывать размеры (для типов данных CHAR, VARCHAR, DECIMAL), в этом же поле корректируем с клавиатуры соответствующие величины.

Список будет таким, как показано на рис. 5.9.

Теперь нужно задать первичный ключ таблицы. Он должен состоять из двух столбцов: REGION и COD. Щелкните правой кнопкой мыши по столбцу REGION и в контекст-

ном меню выберите элемент **Set Primary Key**. Программа установит этот столбец в качестве первичного ключа таблицы.

Чтобы добавить в состав первичного ключа и столбец `COD`, опять щелкните по столбцу `REGION` правой кнопкой мыши и в появившемся контекстном меню выберите элемент **Indexes/Keys**. Появится окно, содержащее описание всех индексов таблицы (рис. 5.10).

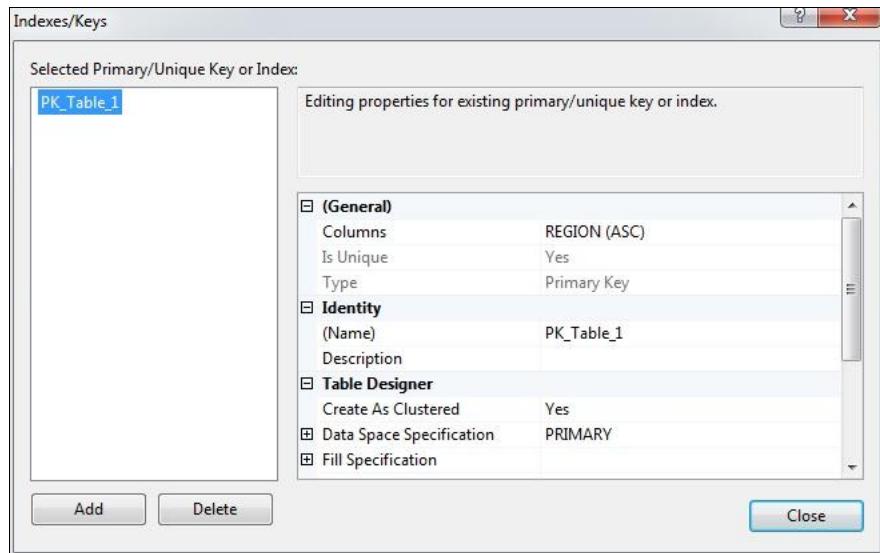


Рис. 5.10. Список индексов создаваемой таблицы

Здесь сразу можно изменить имя индекса первичного ключа, набрав в поле **(Name)** (имя этого поля заключается в скобки) текст `PK_STAFF1`. Чтобы добавить к первичному ключу еще один столбец, нужно щелкнуть мышью по полю **Columns**. Затем в правой части поля — по появившейся кнопке с многоточием `...`.

После этого в следующем появившемся окне **Index Columns** (рис. 5.11) нужно добавить в список столбец `COD`, просто выбрав его из раскрывающегося списка. Для обоих столбцов, входящих в состав индекса, выбрана упорядоченность (**Sort Order**) по возрастанию значений (**Ascending**).

Щелкните в этом окне по кнопке **OK**. Теперь структура индекса будет такой, как показано на рис. 5.12 в поле **Columns**. Здесь изменено имя индекса и добавлен столбец `COD` в состав первичного ключа.

В этом окне щелкните по кнопке **Close**. Список столбцов таблицы теперь будет выглядеть, как показано на рис. 5.13.

Столбцы, входящие в состав первичного ключа, будут отмечены слева соответствующим изображением ключа.

Теперь нужно сохранить созданную таблицу. Щелкните правой кнопкой мыши по заголовку вкладки окна создания структуры таблицы и в контекстном меню выберите элемент **Save Table_1** (рис. 5.14).

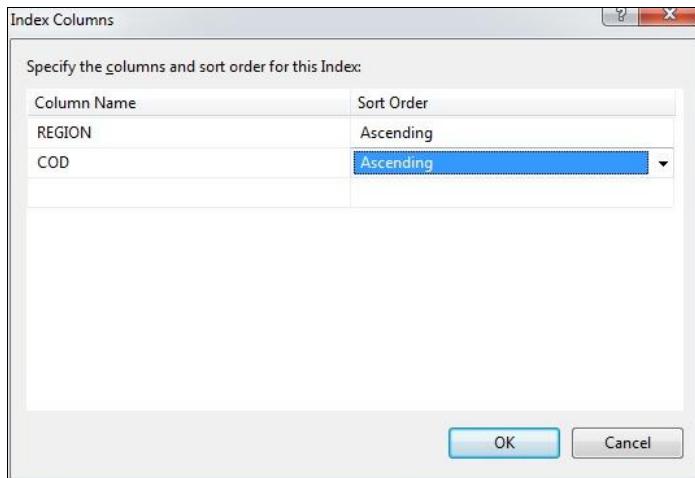


Рис. 5.11. Окно столбцов индекса Index Columns

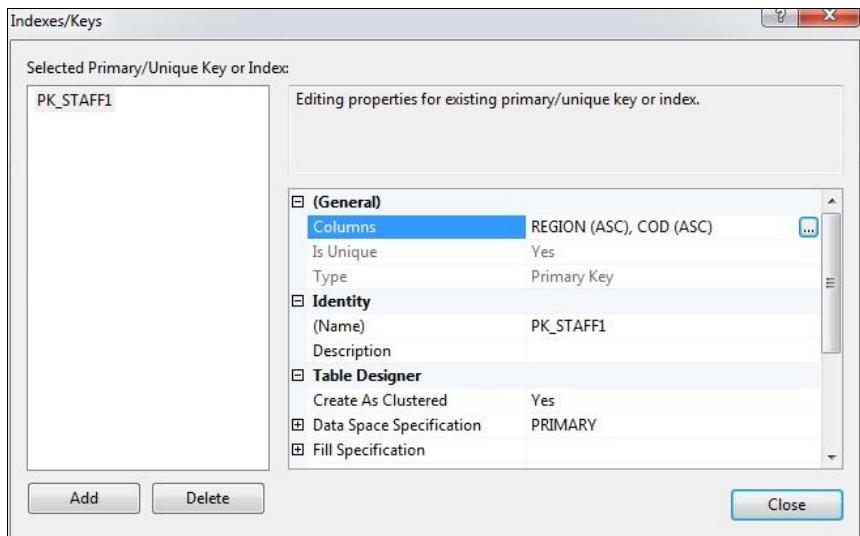


Рис. 5.12. Измененные характеристики индекса

	Column Name	Data Type	Allow Nulls
PK	COD	int	<input checked="" type="checkbox"/>
PK	REGION	char(1)	<input checked="" type="checkbox"/>
	NAME1	varchar(15)	<input checked="" type="checkbox"/>
	NAME2	varchar(15)	<input checked="" type="checkbox"/>
	NAME3	varchar(20)	<input checked="" type="checkbox"/>
	DUTIES	varchar(40)	<input checked="" type="checkbox"/>
	SALARY	decimal(8, 2)	<input checked="" type="checkbox"/>

Рис. 5.13. Окончательный список столбцов создаваемой таблицы

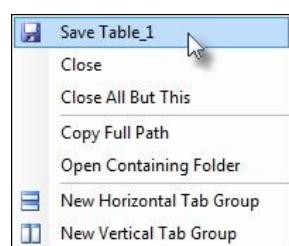


Рис. 5.14. Выбор элемента меню для сохранения таблицы

Появится окно задания имени таблицы (рис. 5.15), где нужно ввести STAFF1 и щелкнуть мышью по кнопке **OK**.

Таблица будет сохранена с этим именем в базе данных.



Рис. 5.15. Ввод имени таблицы

Чтобы задать необходимые характеристики секционирования таблицы, нужно в окне **Object Explorer** раскрыть таблицы базы данных Hugehard (папка **Tables**), щелкнуть правой кнопкой мыши по имени только что созданной таблицы **STAFF1** и выбрать в меню **Storage | Create Partition**. Появится начальное окно Мастера создания секционирования — **Create Partition Wizard** (рис. 5.16).

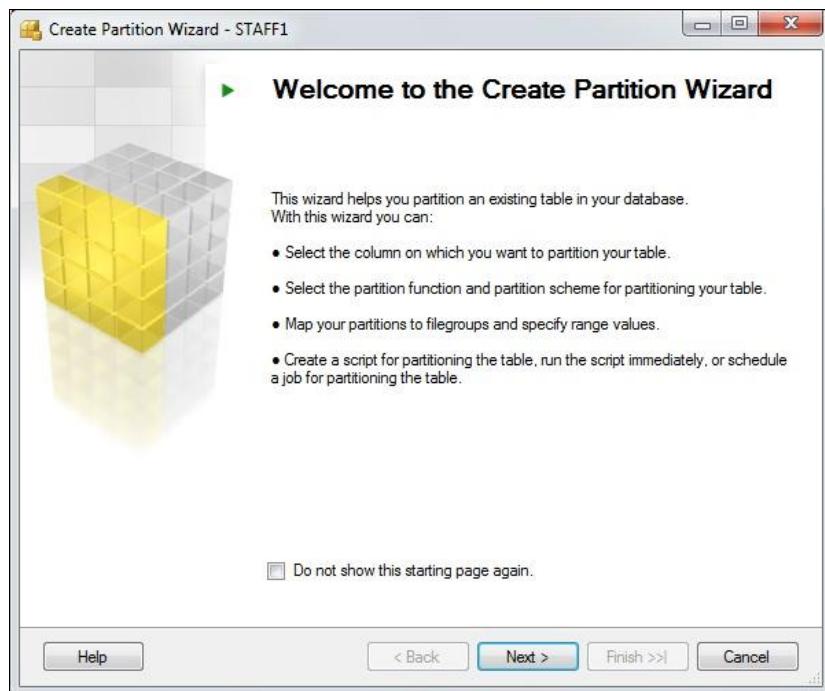


Рис. 5.16. Начальное окно **Create Partition Wizard**

Щелкните по кнопке **Next**.

В следующем окне (рис. 5.17) нужно выбрать столбец, который будет использован в качестве разделяющего ключа.

Отметьте столбец **REGION** и щелкните мышью по кнопке **Next**.

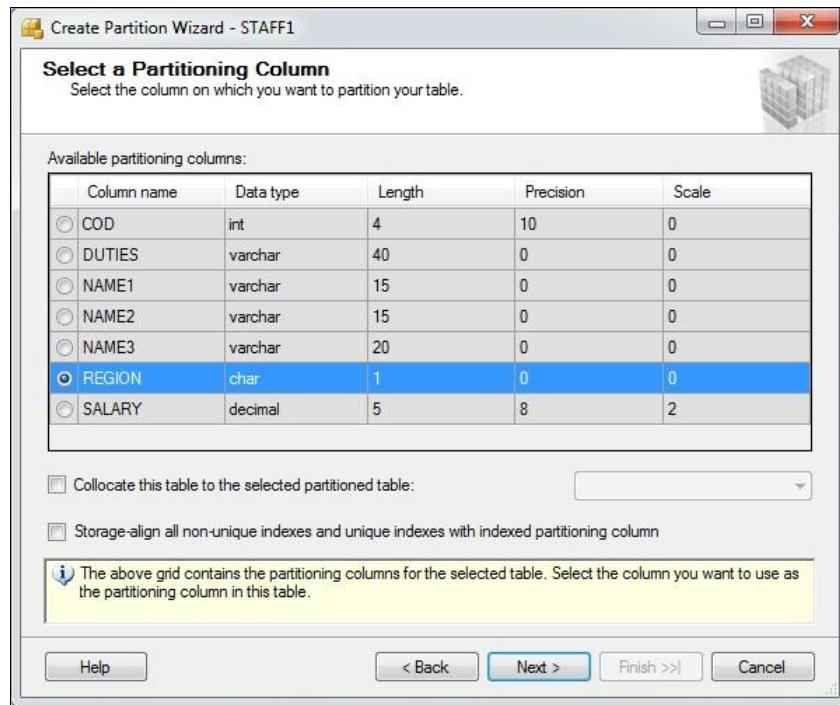


Рис. 5.17. Выбор столбца разделения

В следующем окне (рис. 5.18) нужно выбрать переключатель **Existing partition function:**, поскольку мы собираемся использовать существующую функцию секционирования. В правой части этого поля присутствует и имя функции: FunctionHugehard.

Щелкните мышью по кнопке **Next**.

В следующем окне (рис. 5.19) похожим образом мы выбираем существующую в базе данных схему секционирования, выбрав переключатель **Existing partition scheme** и из раскрывающегося списка справа — нужное имя схемы. На самом деле имя этой единственной схемы уже указано в данном поле.

После щелчка по кнопке **Next** откроется окно, в котором описывается соответствие значений разделяющего ключа именам файловых групп (рис. 5.20).

Здесь ничего менять не нужно, просто щелкните мышью по кнопке **Next**.

Следом появится диалоговое окно (рис. 5.21), в котором можно выбрать действия с полученным скриптом. Его можно выполнить немедленно (**Run immediately**), сохранить в файл (**Script to file**) или поместить в новое окно запросов (**Script to New Query Window**).

Выберем последний вариант и щелкнем мышью по кнопке **Next**.

В следующем окне даются итоговые сведения об используемых объектах базы данных (рис. 5.22).

Здесь нужно только лишь щелкнуть мышью по кнопке **Finish**.

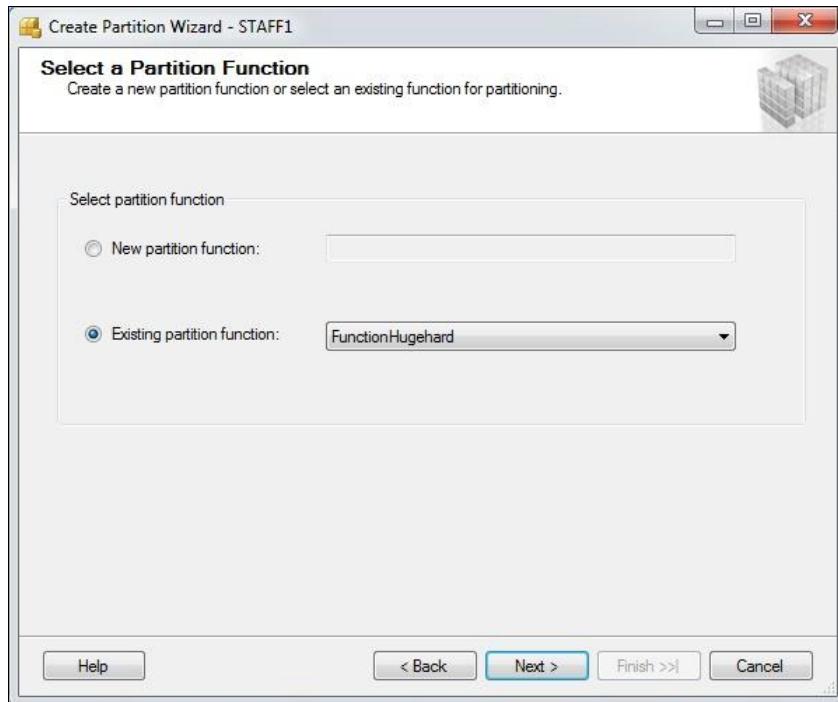


Рис. 5.18. Выбор функции секционирования

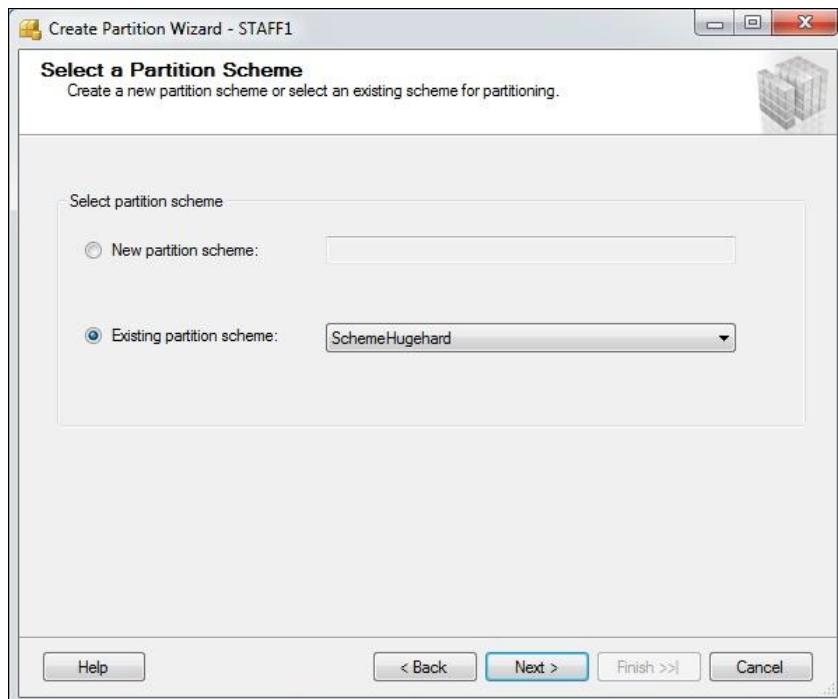


Рис. 5.19. Выбор схемы секционирования

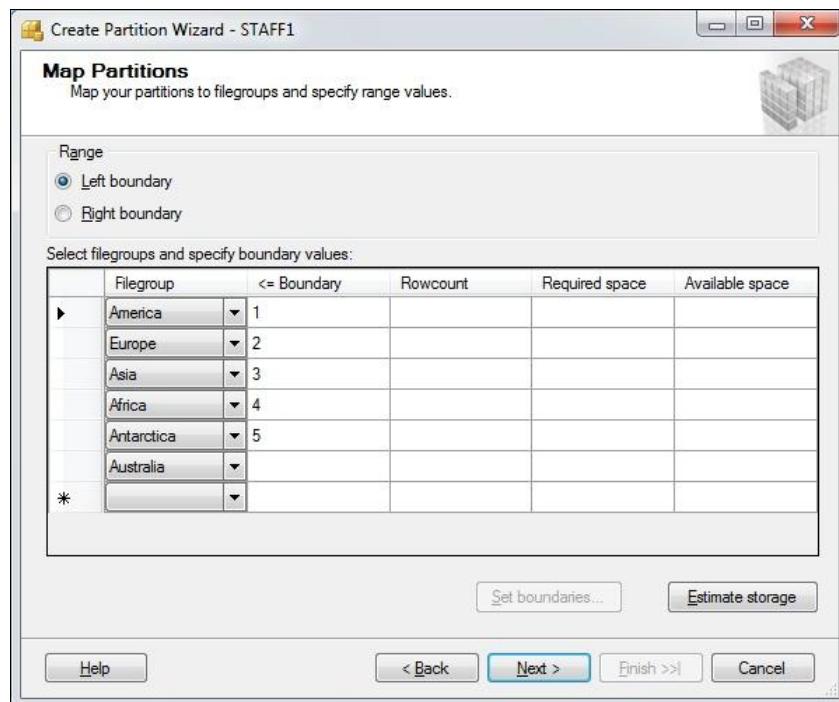


Рис. 5.20. Соответствие значений разделяющего ключа именам файловых групп

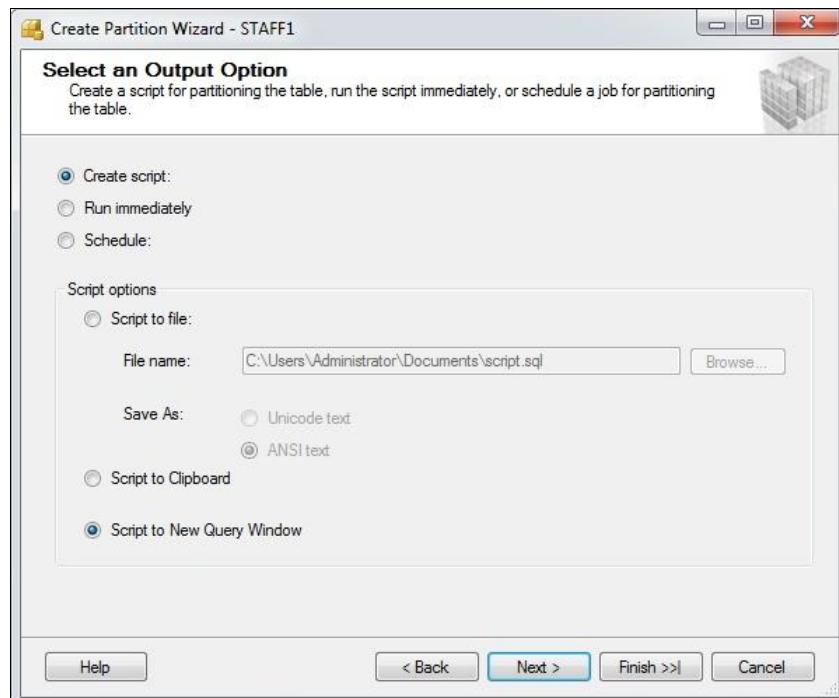


Рис. 5.21. Сохранение созданного скрипта

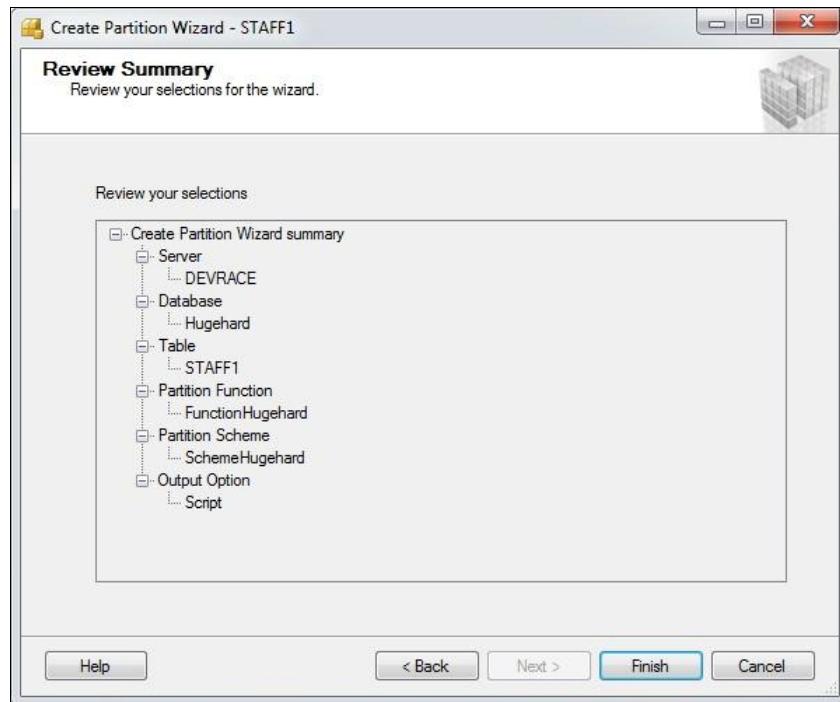


Рис. 5.22. Итоговые данные по выполненной работе

В самом последнем окне сообщается об успешном выполнении действий (рис. 5.23). Щелкните мышью по кнопке **Close**.

Однако это еще не все. Вам нужно выполнить скрипт, который был создан Мастером **Create Partition Wizard**. Этот скрипт, как мы указали, помещается в новое окно запросов. Его текст показан в примере 5.24.

Пример 5.24. Сгенерированный Мастером скрипт задания секционирования таблицы сотрудников

```
USE [Hugehard]
GO
BEGIN TRANSACTION
ALTER TABLE [dbo].[STAFF1] DROP CONSTRAINT [PK_STAFF1]

ALTER TABLE [dbo].[STAFF1] ADD CONSTRAINT [PK_STAFF1] PRIMARY KEY CLUSTERED
(
    [REGION] ASC,
    [COD] ASC
)WITH (PAD_INDEX = OFF,
       STATISTICS_NORECOMPUTE = OFF,
       SORT_IN_TEMPDB = OFF,
       IGNORE_DUP_KEY = OFF,
       ONLINE = OFF,
```

```
ALLOW_ROW_LOCKS = ON,  
ALLOW_PAGE_LOCKS = ON)  
ON [SchemeHugehard] ([REGION])
```

```
COMMIT TRANSACTION
```

Выполните этот скрипт, нажав клавишу <F5>.

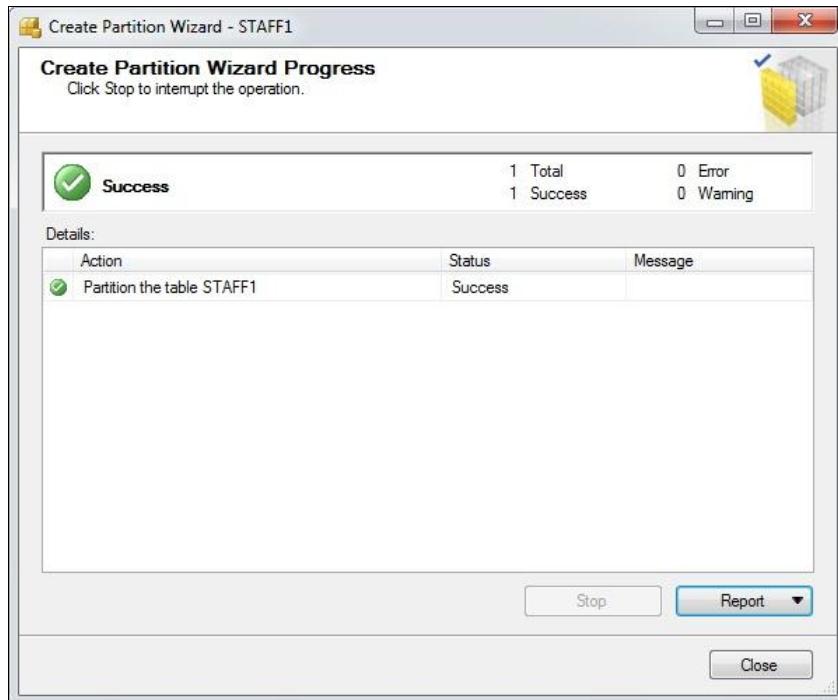


Рис. 5.23. Завершение секционирования таблицы

Первым оператором скрипта является оператор запуска транзакции с характеристиками по умолчанию:

```
BEGIN TRANSACTION
```

Далее в скрипте удаляется первичный ключ таблицы при выполнении первого оператора ALTER TABLE. Затем этот ключ заново создается с большим количеством характеристик соответствующего индекса, а последней строкой во втором операторе изменения таблицы записано нужное нам предложение ON:

```
ON [SchemeHugehard] ([REGION])
```

В завершении скрипта выполняется подтверждение транзакции, в контексте которой выполнялись все изменения:

```
COMMIT TRANSACTION
```

О транзакциях мы еще с вами будем говорить в дальнейшем.

Честно скажу, я утомился, создавая секционированную таблицу в диалоговых средствах Management Studio. Лишний раз убедился, что в подобных действиях намного удобнее использовать средства языка Transact-SQL.

Однако продолжим процесс освоения диалоговых средств Management Studio.

5.4.2. Создание таблицы секционирования, схемы секционирования и функции секционирования

При создании средств секционирования таблицы STAFF1 мы использовали существующую схему секционирования и функцию секционирования. Теперь же давайте создадим новую таблицу и сформируем средства секционирования "с нуля", т. е. создав с помощью диалоговых средств Management Studio схему и функцию секционирования.

Создайте любыми известными вам средствами новую таблицу с именем STAFF2. Она должна иметь ту же структуру, что и таблица STAFF1 (и STAFF). Я, конечно, использовал для этого операторы Transact-SQL.

Для создания средств секционирования в окне **Object Explorer** щелкните правой кнопкой мыши по имени вновь созданной таблицы и выберите в контекстном меню **Storage | Create Partition**. Появится начальное окно Мастера создания секционирования. Его мы уже видели ранее (см. рис. 5.16). После щелчка по кнопке **Next** откроется окно выбора разделяющего ключа (столбца разделения) (рис. 5.24).

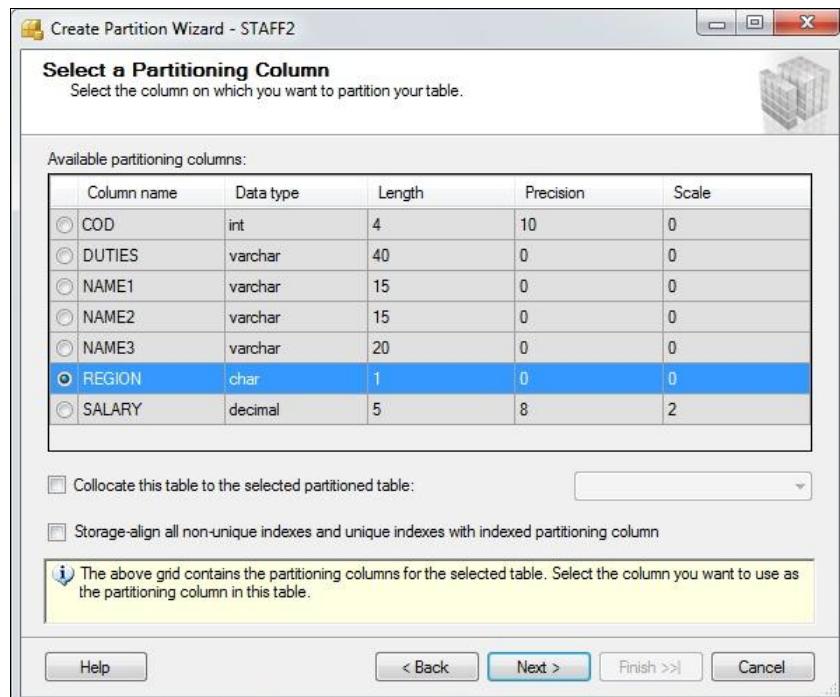


Рис. 5.24. Выбор столбца разделения

Отмечаем столбец **REGION** и щелкаем по кнопке **Next**. В следующем окне (рис. 5.25) указываем, что будем создавать новую функцию секционирования, и вводим в поле **New partition function** имя функции — **FunctionHugehard2**.

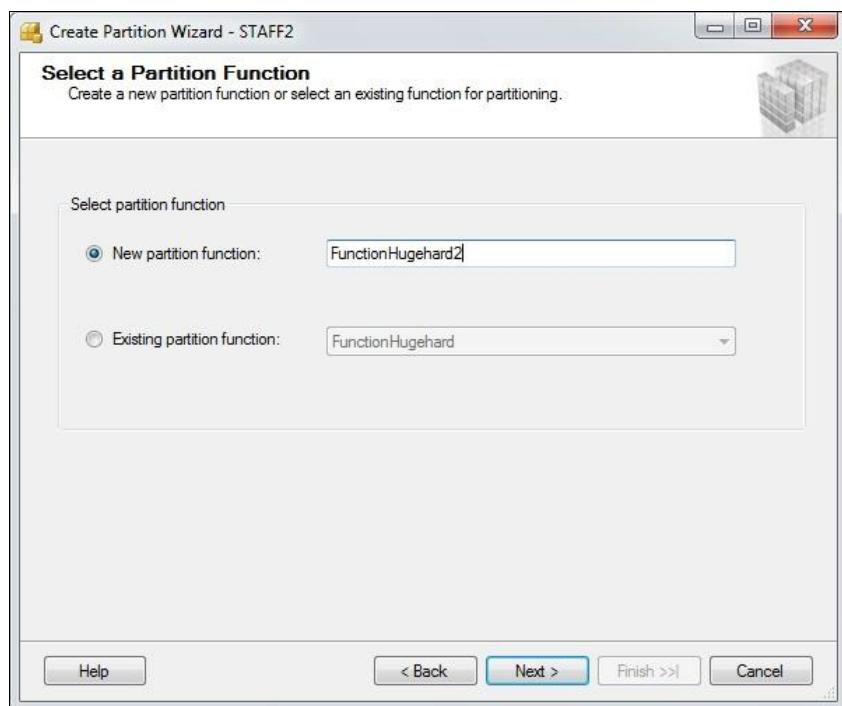


Рис. 5.25. Задание новой функции секционирования

После щелчка по кнопке **Next** в следующем окне (рис. 5.26) мы указываем, что будем создавать новую схему секционирования, выбрав переключатель **New partition scheme**.

В этом поле **New partition scheme** вводим имя схемы **SchemeHugehard2**.

Щелкаем по кнопке **Next**.

В следующем окне (рис. 5.27) нужно создать условия секционирования таблицы.

Здесь в столбце **Filegroup** нужно поочередно для каждой строки из раскрывающегося списка выбрать имена файловых групп: America, Europe, Asia, Africa, Australia и Antarctica.

В следующем столбце задания границ значений **<= Boundary** для каждой новой строки поочередно вводим числа 1, 2, 3, 4, 5. Обратите внимание, что здесь, в отличие от оператора SQL, значения не нужно заключать в апострофы.

Результат показан на рис. 5.28.

Для последней строки, **Antarctica**, значение границы не указывается. Это будут все значения, которые превышают предыдущее значение, 5.

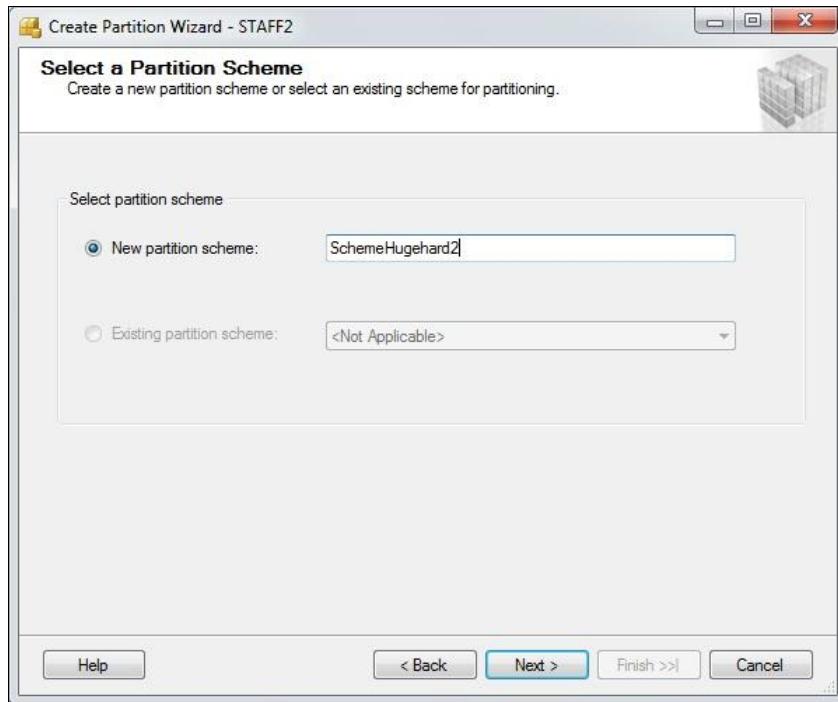


Рис. 5.26. Задание новой схемы секционирования

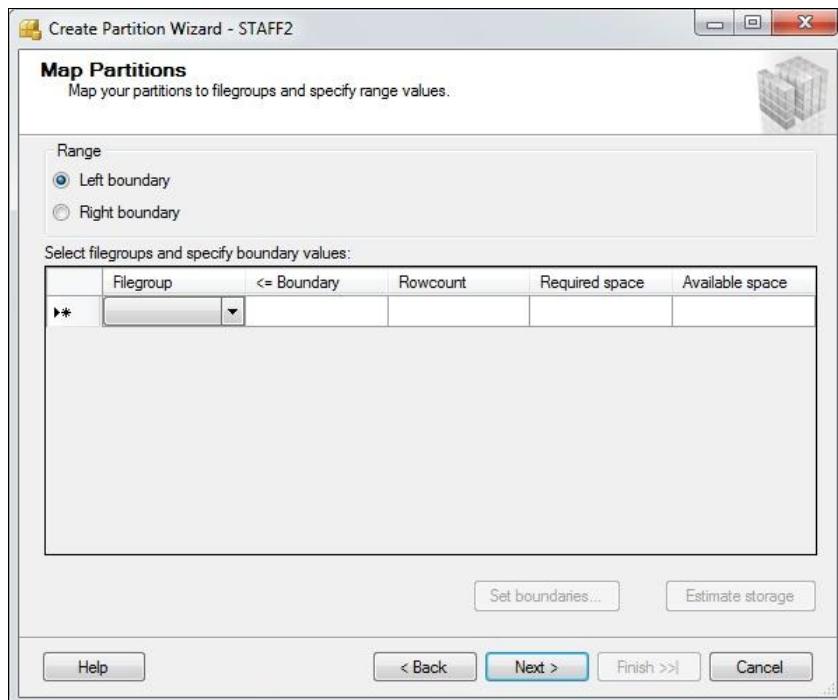


Рис. 5.27. Окно задания условий секционирования таблицы

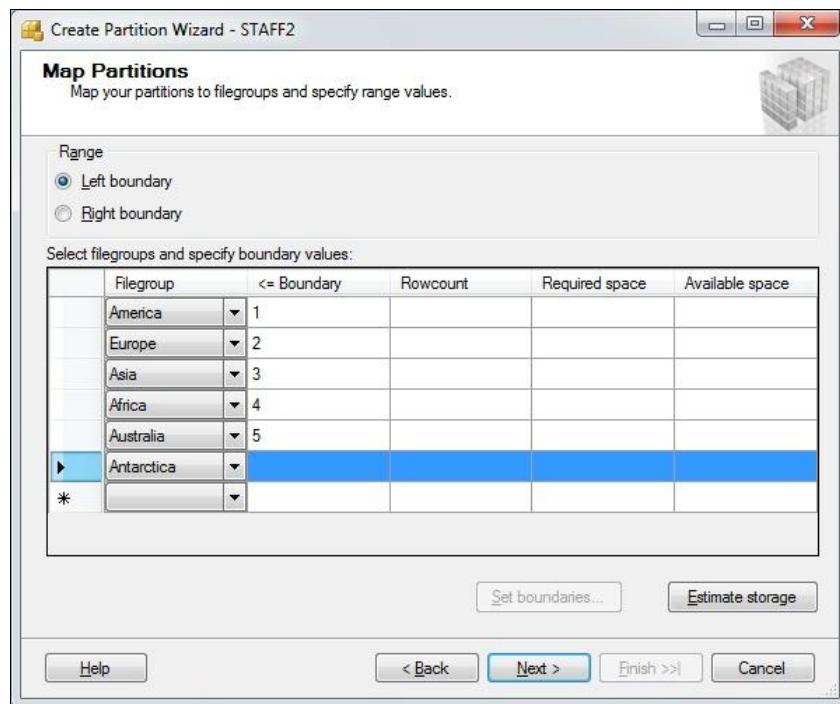


Рис. 5.28. Заданные условия секционирования таблицы

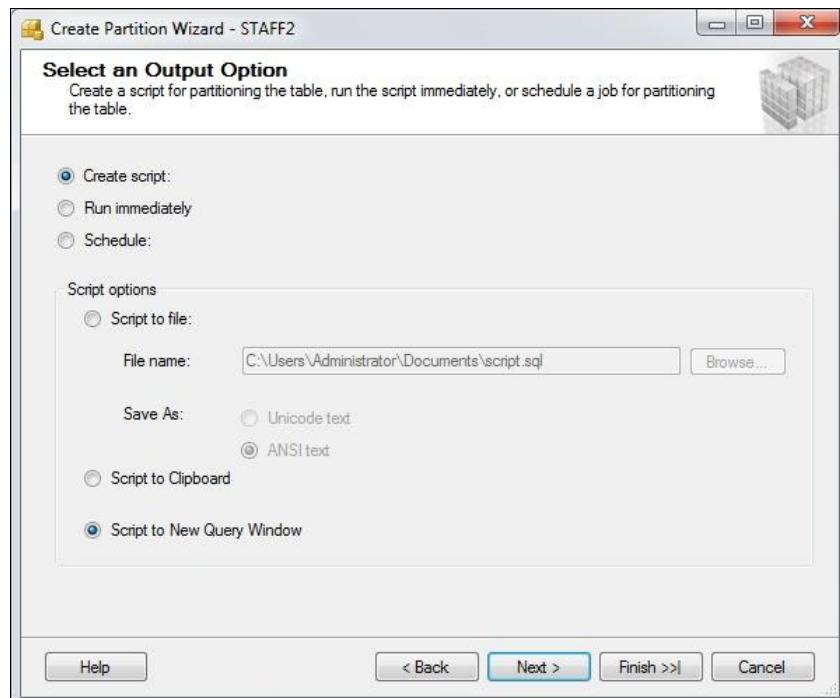


Рис. 5.29. Указание, что необходимо создание скрипта

Щелкаем по кнопке **Next** и в следующем окне (рис. 5.29) отмечаем, что требуется создание скрипта в новом окне запросов (выбираем переключатель **Script to New Query Window**).

После щелчка по кнопке **Next** мы увидим итоговое окно (рис. 5.30), в котором нужно только лишь щелкнуть мышью по кнопке **Finish**.

Последним будет окно завершения работы Мастера с перечнем создаваемых объектов и с указанием успешности выполнения необходимых действий (рис. 5.31).

Закройте это последнее окно, щелкнув мышью по кнопке **Close**.

В этом режиме все, что сделал Мастер создания секционированной таблицы, это лишь сгенерированный скрипт, только при выполнении которого будет создана функция секционирования, схема секционирования и будут внесены небольшие изменения в таблицу **STAFF2**.

Скрипт показан в примере 5.25.

Пример 5.25. Сгенерированный Мастером скрипт задания секционирования таблицы сотрудников

```
USE [Hugehard]
GO
BEGIN TRANSACTION
CREATE PARTITION FUNCTION [FunctionHugehard2] (char(1)) AS RANGE LEFT FOR VALUES
(N'1', N'2', N'3', N'4', N'5')

CREATE PARTITION SCHEME [SchemeHugehard2] AS PARTITION [FunctionHugehard2] TO
([America], [Europe], [Asia], [Africa], [Australia], [Antarctica])

ALTER TABLE [dbo].[STAFF2] DROP CONSTRAINT [PK_STAFF2]

ALTER TABLE [dbo].[STAFF2] ADD CONSTRAINT [PK_STAFF2] PRIMARY KEY CLUSTERED
(
    [REGION] ASC,
    [COD] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,
IGNORE_DUP_KEY = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS =
ON) ON [SchemeHugehard2] ([REGION])

COMMIT TRANSACTION
```

Выполните скрипт. В окне **Object Explorer** можно увидеть, что в базе данных появилась новая функция секционирования и новая схема секционирования. Чтобы их действительно увидеть, необходимо обновить список. Для этого правой кнопкой мыши нужно щелкнуть по имени базы данных **Hugehard** и в появившемся контекстном меню выбрать **Refresh**.

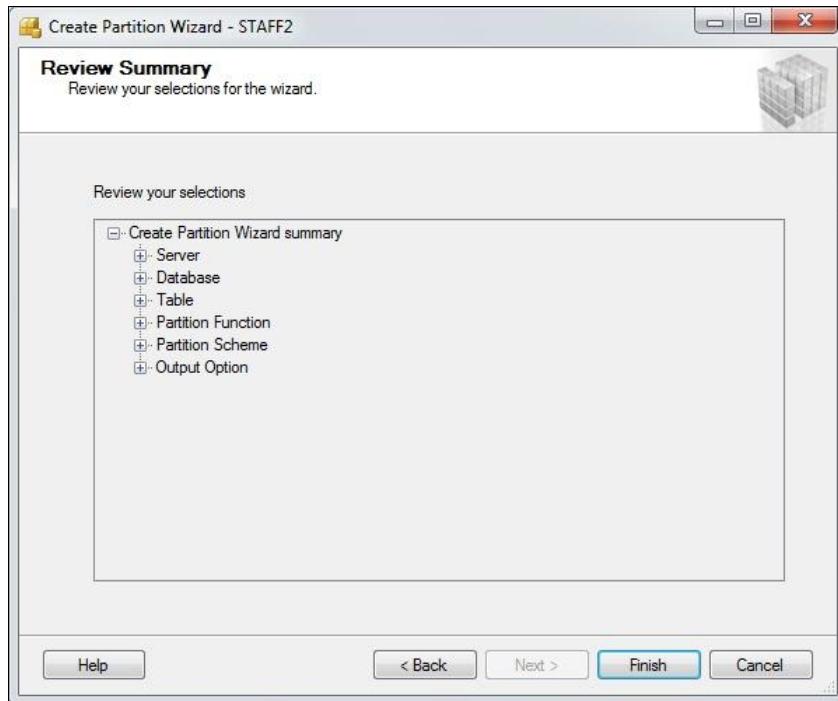


Рис. 5.30. Итоговое окно

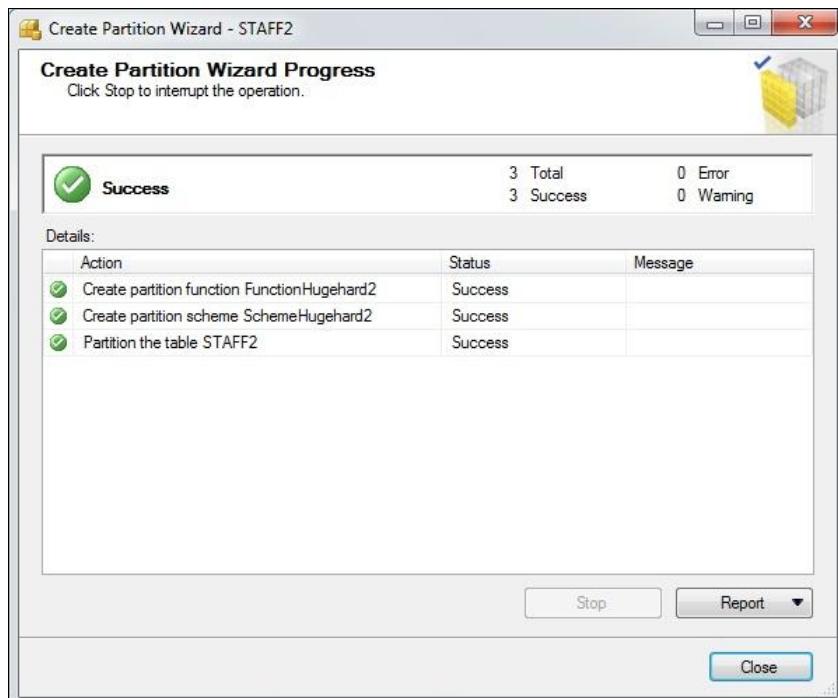


Рис. 5.31. Завершение работы Мастера

5.5. Отображение состояния секционированных таблиц

На сегодняшний день у нас в базе данных Hugehard есть три секционированные таблицы. Для того чтобы посмотреть, как распределены данные между файловыми группами, щелкните правой кнопкой мыши по имени базы данных **Hugehard** и в контекстном меню выберите **Reports | Standard Reports | Disk Usage by Partition**.

В результате будет создан отчет, как показано на рис. 5.32.

Disk Usage by Partition
[Hugehard]
on DEVRACE at 20.01.2012 1:19:25

This report provides detailed data on the utilization of disk space by index and by partitions within the Database.

Table Name	# Records	Reserved (KB)	Used (KB)
dbo.STAFF	120	96	96
Index (PK_STAFF)	120	96	96
Partition No.	# Records	Reserved (KB)	Used (KB)
1	20	16	16
2	20	16	16
3	20	16	16
4	20	16	16
5	20	16	16
6	20	16	16
dbo.STAFF1	0	0	0
Index (PK_STAFF1)	0	0	0
Partition No.	# Records	Reserved (KB)	Used (KB)
1	0	0	0
dbo.STAFF2	0	0	0
Index (PK_STAFF2)	0	0	0

Рис. 5.32. Распределение данных по файловым группам секционированной таблицы

Здесь можно видеть, что в каждом разделе для таблицы STAFF присутствует по 20 записей. В таблицах STAFF1 и STAFF2 пока нет данных, поэтому там указано нулевое количество записей.

5.6. Файловые потоки

Прежде чем приступить к исследованию возможностей файловых потоков, следует проверить, поддерживает ли ваша конфигурация сервера базы данных такие средства. При инсталляции системы мы задали такую поддержку. Тем не менее на всякий случай выполните следующую проверку и при необходимости внесите нужные корректизы.

Запустите на выполнение программу SQL Server Configuration Manager (рис. 5.33).

В правой части окна щелкните правой кнопкой мыши по имени сервера. В моем случае это строка **SQL Server (MSSQLSERVER)**. В контекстном меню выберите

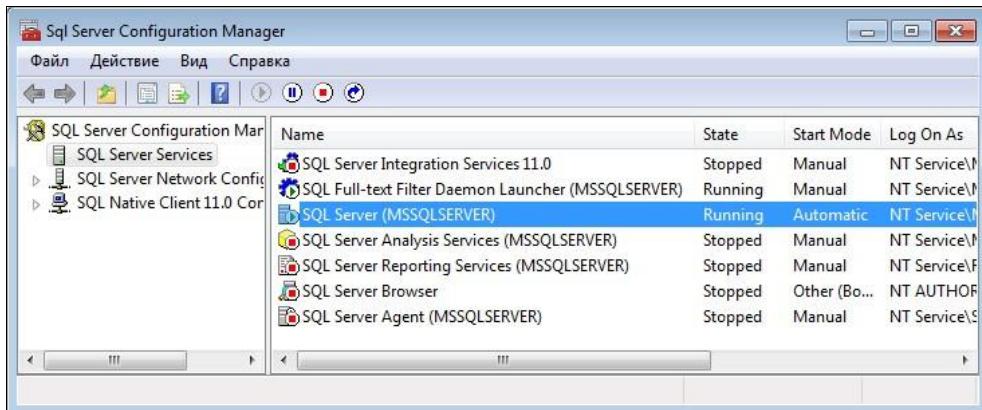


Рис. 5.33. Главное окно программы SQL Server Configuration Manager

элемент **Свойства**. Появится диалоговое окно свойств сервера базы данных. Выберите вкладку **FILESTREAM** (рис. 5.34).

Здесь нужно установить флажок допустимости использования файловых потоков в обычных операторах Transact-SQL: **Enable FILESTREAM for Transact-SQL access**. В этом случае для таблиц, использующих файловые потоки, можно будет применять обычные операторы добавления данных (`INSERT`), изменения существующих данных (`UPDATE`) и удаления строк таблицы (`DELETE`). Все эти операторы Transact-SQL будут выполняться "естественным" образом. То есть при изменении данных файлового потока будут корректироваться и данные в файловой системе.

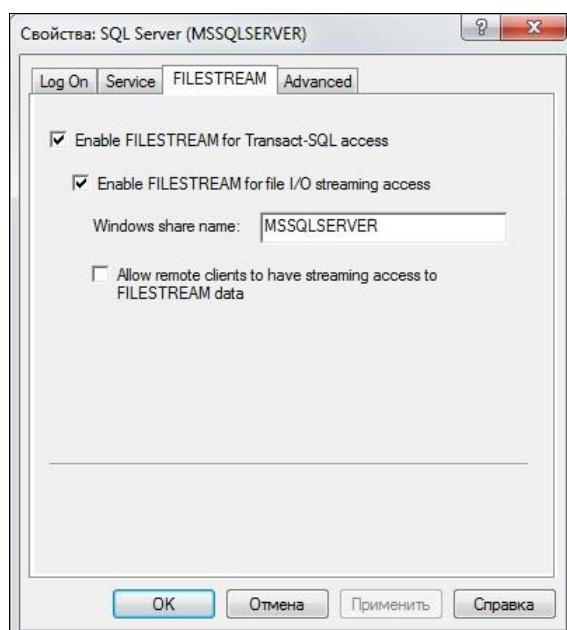


Рис. 5.34. Вкладка FILESTREAM свойств сервера

При удалении строк таблицы данные файлового потока также будут физически удаляться и из файлов файловой системы.

Если нужно, чтобы к файлам потока осуществлялся доступ и средствами операционной системы, установите флажок **Enable FILESTREAM for file I/O streaming access**. При этом в поле **Windows share name** нужно ввести имя сервера базы данных: MSSQLSERVER.

Если вы хотите, чтобы с файловыми потоками могли работать и удаленные клиенты, то следует установить и флажок **Allow remote clients to have streaming access to FILESTREAM data**. На рисунке, как вы видите, это опция не установлена, потому что на моем компьютере нет никаких сетевых подключений и, соответственно, отсутствуют другие клиенты.

Если вы внесли изменения в этом окне, то щелкните по кнопке **Применить**. Для того чтобы эти изменения вступили в силу, необходимо выполнить реконфигурацию при помощи следующего кода, который нужно ввести в командной строке:

```
EXEC sp_configure filestream_access_level, 2  
RECONFIGURE
```

Тип данных VARBINARY(MAX) для столбцов таблиц позволяет хранить произвольные данные. Это могут быть форматированные тексты, рисунки, звуковые файлы, фильмы. Такой тип данных в реляционных системах баз данных называется обычно двоичным большим объектом BLOB (Binary Large OBject) или LOB. Эти данные могут храниться в самой базе данных. Тогда максимальный объем памяти, который они могут занимать, ограничивается размером 2 Гбайта. Размер достаточно большой, однако существуют приложения, требующие еще большего размера.

В SQL Server существуют средства, позволяющие увеличить размер памяти, используемой для хранения таких типов данных. Это так называемые *файловые потоки* (filestream).

Файловый поток в SQL Server — это средство хранения больших двоичных объектов (типов данных VARBINARY (MAX)) в файловой системе Windows. Для конкретного столбца таблицы указывается, что его данные будут храниться не непосредственно в базе данных, а в файловом потоке. В одной таблице может присутствовать несколько таких столбцов. При этом в соответствующей пользовательской базе данных должна существовать определенная файловая группа, предназначенная для хранения именно файловых потоков, и только их.

ЗАМЕЧАНИЕ

Для данных в файловых потоках невозможно выполнить шифрование.

Для создания такой файловой группы можно использовать либо оператор CREATE DATABASE в случае первоначального создания базы данных, либо оператор ALTER DATABASE для помещения в существующую базу данных новой файловой группы.

Давайте для базы данных Hugehard создадим файловую группу для файловых потоков. Используем для этого оператор ALTER DATABASE (пример 5.26).

Пример 5.26. Добавление в базу данных новой файловой группы

```
USE master;
GO
ALTER DATABASE Hugehard
    ADD FILEGROUP GroupStream CONTAINS FILESTREAM;
GO
ALTER DATABASE Hugehard
    ADD FILE
        (NAME = GroupStream,
        FILENAME = 'd:\Hugehard\GroupStream')
    TO FILEGROUP GroupStream;
GO
```

За одно выполнение оператора ALTER DATABASE можно выполнить только одно действие. Первый оператор добавляет в базу данных новую файловую группу с именем GroupStream. Ключевые слова CONTAINS FILESTREAM указывают на то, что файловая группа предназначена для хранения файловых потоков.

Второй оператор добавляет файл в созданную файловую группу. В предложении FILENAME задается путь к файлу. Все каталоги в пути должны существовать на внешнем носителе, за исключением последнего каталога. Он будет создан системой. Внутри этого каталога будет создан файл `filestream.hdr` и каталог `$FSLOG`.

Чтобы просмотреть список файловых групп базы данных, щелкните правой кнопкой мыши по имени базы данных **Hugehard** в окне **Object Explorer**, выберите в контекстном меню элемент **Properties** и в появившемся окне в левом верхнем углу выберите вкладку **Filegroups** (рис. 5.35).

Файловые группы, предназначенные для хранения файловых потоков, отображаются в нижней части окна, в разделе **Filestream**. В данном случае такая файловая группа одна.

Создадим таблицу, в которой будут использоваться файловые потоки. Это уже известная нам таблица стран с небольшими уточнениями. Для создания таблицы выполните операторы примера 5.27.

Пример 5.27. Таблица стран, использующая файловые потоки

```
USE Hugehard;
/** Справочник стран */
CREATE TABLE REFCTR
( CODCTR  CHAR(3) NOT NULL,      /* Код страны */
  NAME      VARCHAR(60),          /* Краткое название страны */
  FULLNAME VARCHAR(65),          /* Полное название страны */
  CAPITAL   VARCHAR(30),          /* Название столицы */
  ID        UNIQUEIDENTIFIER,    /* Идентификатор для файловых потоков */
  ROWGUIDCOL NOT NULL
    DEFAULT NEWID() UNIQUE,
```

```

MAP        VARBINARY (MAX)
FILESTREAM,      /* Кarta страны */
DESCR     VARBINARY (MAX)
FILESTREAM,      /* Дополнительное описание */
CONSTRAINT PK_REFCTR PRIMARY KEY (CODCTR)
) ;
GO

```

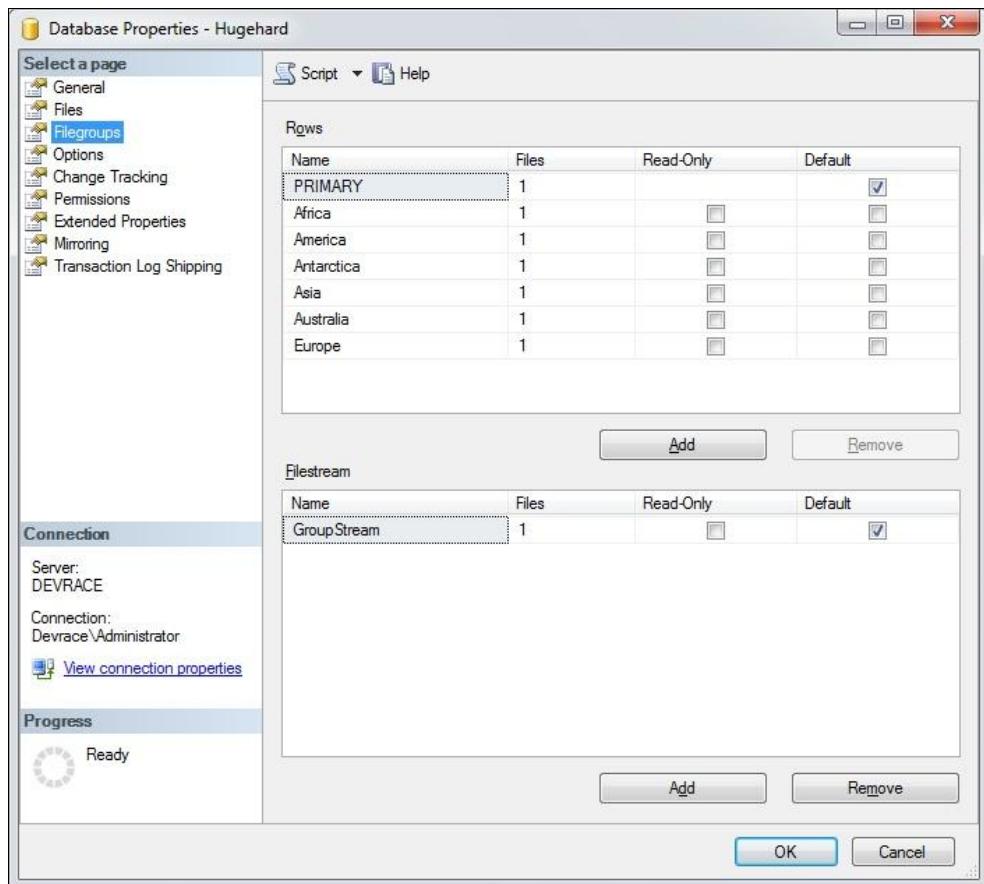


Рис. 5.35. Файловые группы базы данных Hugehard

В этой таблице предполагается хранить в файловом потоке значения двух столбцов — MAP (карта страны) и DESC (некоторое форматированное текстовое описание). Оба они заданы с атрибутом FILESTREAM.

Таблица, где присутствуют столбцы, данные которых будут храниться в файловых потоках, обязательно должна содержать первичный или уникальный ключ с типом данных UNIQUEIDENTIFIER и с атрибутом ROWGUIDCOL. С этой целью в структуру таблицы включен столбец ID с соответствующими характеристиками. Чтобы пользователь не мучился с присваиванием значения этому идентификатору, для столбца

установлено значение по умолчанию: NEWID(). Соответствующее значение будет присваиваться столбцу при помещении в таблицу новой строки.

После создания таблицы со столбцами, значения которых будут храниться в файловом потоке, на внешнем устройстве в каталоге GroupStream системой создается еще ряд каталогов и файлов.

Запишем в эту таблицу пару строк, а затем их отобразим.

Выполните операторы примера 5.28. Здесь в таблицу добавляется две строки.

Пример 5.28. Внесение данных в таблицу, использующую файловые потоки

```
USE Hugehard;
INSERT INTO REFCTR (CODCTR, NAME, DESCRIPTOR)
VALUES ('RUS', 'Россия',
        CAST('Произвольное описание' AS VARBINARY(MAX)));
INSERT INTO REFCTR (CODCTR, NAME, DESCRIPTOR)
VALUES ('USA', 'United States',
        CAST('Тоже некоторое описание' AS VARBINARY(MAX)));
GO
```

Чтобы поместить в столбец типа данных VARBINARY(MAX) строковые данные (да и любые другие данные, отличные от двоичного типа данных), необходимо выполнить явное преобразование строки к типу данных VARBINARY(MAX) при помощи функции CAST(), что мы и сделали в операторах INSERT.

Для отображения введенных данных выполните оператор SELECT, как показано в примере 5.29.

Пример 5.29. Отображение данных таблицы, использующей файловые потоки

```
USE Hugehard;
SELECT CODCTR AS 'Код',
       NAME AS 'Название',
       DESCRIPTOR AS 'Описание'
  FROM REFCTR;
GO
```

Результат будет не очень наглядным. Двоичные данные отображаются именно как двоичные:

Код	Название	Описание
RUS	Россия	0xCFF0EEE8E7E2EEEBCFCEDEEE520EEEFE8F1E0EDE8E5
USA	United States	0xD3EE6E520EDE5EAEEF2EEF0EEE520EEEFE8F1E0EDE8E5

(2 row(s) affected)

Изменим несколько оператор выборки данных, добавив отображение идентификатора ID и выполнив преобразование двоичных данных к строковому типу данных переменной длины (пример 5.30).

Пример 5.30. Другой вариант отображения данных таблицы, использующей файловые потоки

```
USE Hugehard;
SELECT CODCTR AS 'Код',
       NAME AS 'Название',
       ID AS 'Идентификатор',
       CAST(DESCR AS VARCHAR(50)) AS 'Описание'
  FROM REFCTR;
GO
```

Результат:

Код	Название	Идентификатор	Описание
RUS	Россия	3B90370E-6009-4906-B983-1DD80A63C55F	Произвольное описание
USA	United States	AB8DEF3C-4171-4A2F-8A88-5B215AC73484	Тоже некоторое описание

(2 row(s) affected)

Несмотря на то, что данные описания хранятся в отдельном файле, мы можем с ними работать точно таким же образом, как и с любыми данными, хранящимися в самой базе данных.

5.7. Удаление таблиц

Удалить существующую в базе данных таблицу можно при выполнении оператора Transact-SQL `DROP TABLE` или же при использовании диалоговых средств в программе Management Studio.

Нельзя удалить таблицу, у которой существуют так называемые *внешние зависимости*. Это когда на таблицу ссылается хранимая процедура, представление (`VIEW`) или когда на первичный или уникальный ключ таблицы ссылается внешний ключ другой таблицы.

5.7.1. Определение зависимостей таблицы

Проще всего определить зависимости таблицы в Management Studio.

В окне **Object Explorer** раскройте папку **Databases**, **BestDatabase**, папку **Tables**. Щелкните правой кнопкой мыши по имени таблицы `dbo.REFCTR`. Выберите в контекстном меню элемент **View Dependencies** (посмотреть зависимости). Появится окно, в котором при выбранном переключателе **Objects that depend on**

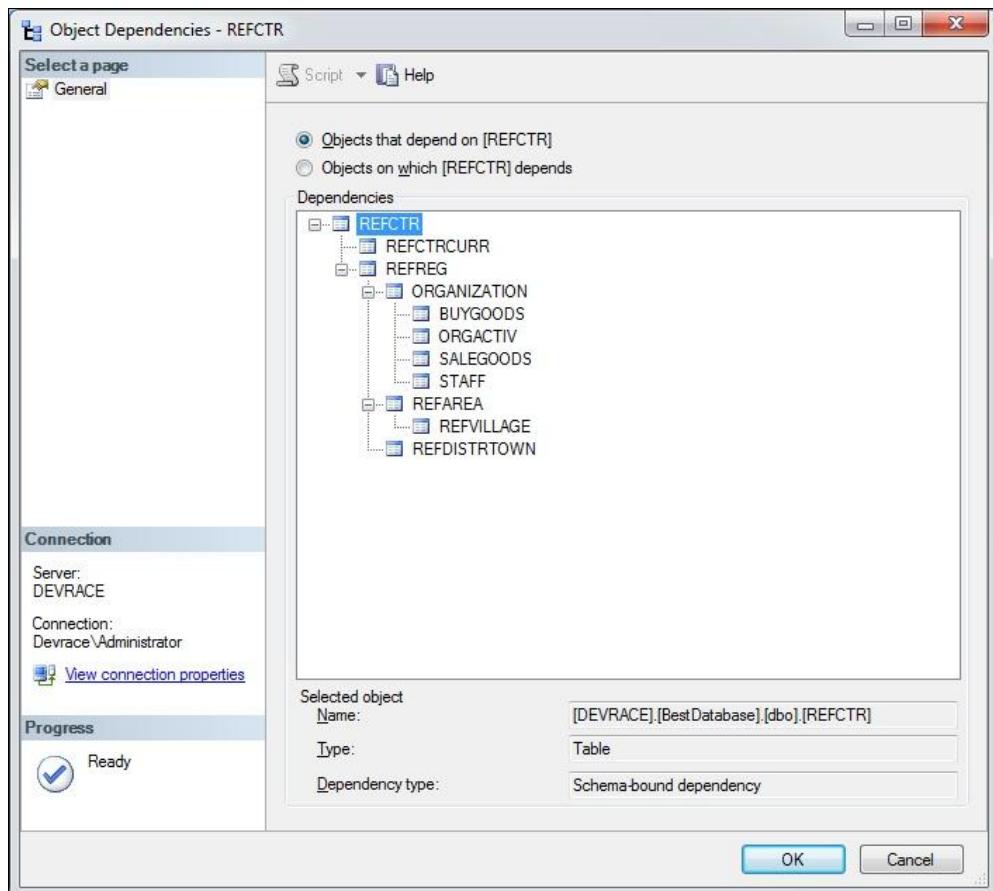


Рис. 5.36. Список объектов, зависящих от таблицы REFCTR

[REFCTR] показаны все объекты базы данных, которые так или иначе ссылаются на таблицу REFCTR (рис. 5.36).

Если раскрыть все элементы дерева, то можно увидеть две таблицы, которые напрямую зависят от справочника стран, а также множество таблиц, зависящих от таблиц, которые зависят от справочника стран.

Если же выбрать второй переключатель, **Objects on which [REFCTR] depends**, то можно увидеть список объектов, от которых зависит эта таблица (рис. 5.37).

Здесь перечислены имена пользовательских типов данных, используемых при описании характеристик столбцов этой таблицы.

Любопытства ради можно посмотреть зависимости таблицы STAFF в базе данных Hugehard. На рис. 5.38 показана зависимость этой таблицы от схемы секционирования SchemeHugehard, которая в свою очередь зависит от функции секционирования FunctionHugehard.

Другие средства определения зависимостей в базе данных не производят хорошего впечатления. За одно обращение к какому-либо представлению нельзя получить

той исчерпывающей картины, которую дает Management Studio. Зато при помощи этих средств можно отобразить в достаточно хорошем виде многие характеристики таблиц.

Например, системная функция `sp_helpconstraint` позволяет отобразить все ограничения таблицы. Следующий пакет позволяет отобразить все ограничения таблицы `REFCTR` базы данных `BestDatabase`. Обращение к функции показано в примере 5.31.

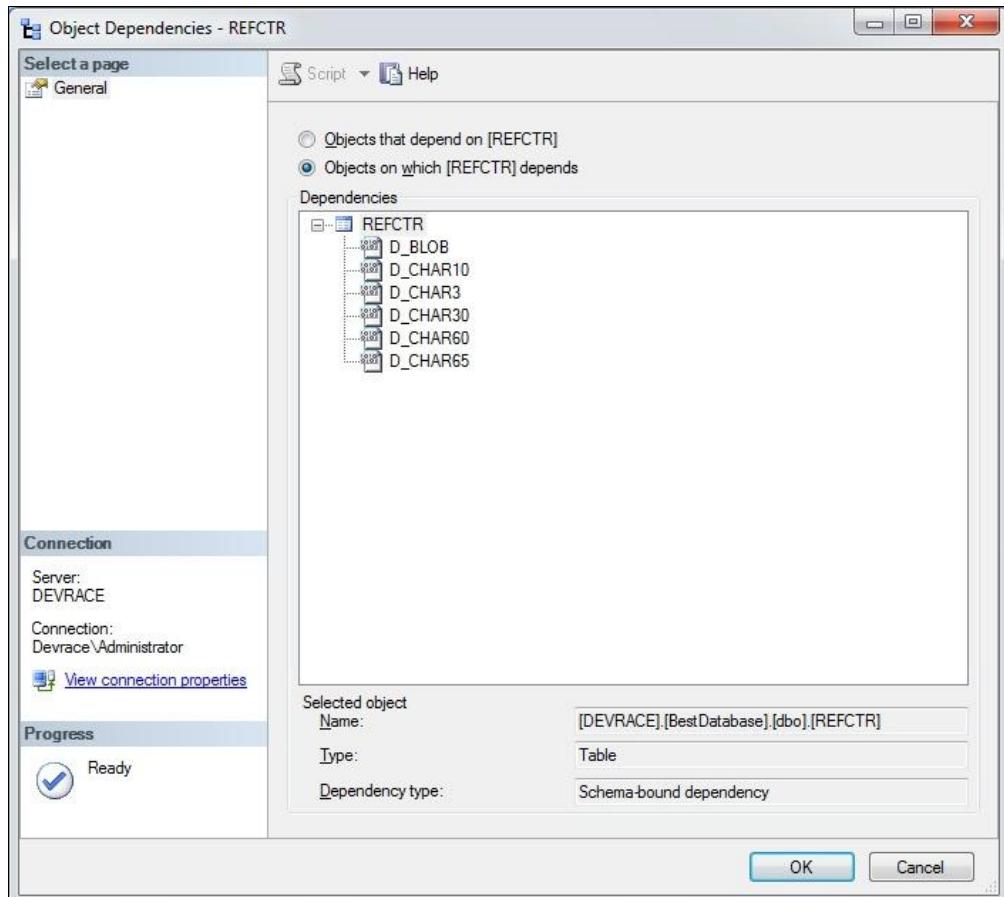


Рис. 5.37. Список объектов, от которых зависит таблица `REFCTR`

Пример 5.31. Отображение ограничений таблицы `REFCTR`

```
USE BestDatabase;
GO
EXEC sp_helpconstraint 'dbo.REFCTR';
GO
```

В несколько сокращенном виде будет получен такой результат:

Object Name

dbo.REFCTR

constraint_type	constraint_name	constraint_keys
PRIMARY KEY (clustered)	PK_REFCTR	CODCTR

Table is referenced by foreign key

BestDatabase.dbo.REFCTRCURR: FK1_REFCTRCURR

BestDatabase.dbo.REFREG: FK_REFREG

Здесь отображаются:

- ◆ имя исходной таблицы: dbo.REFCTR;
- ◆ сведения о первичном ключе таблицы: PRIMARY KEY (clustered) PK_REFCTR CODCTR;

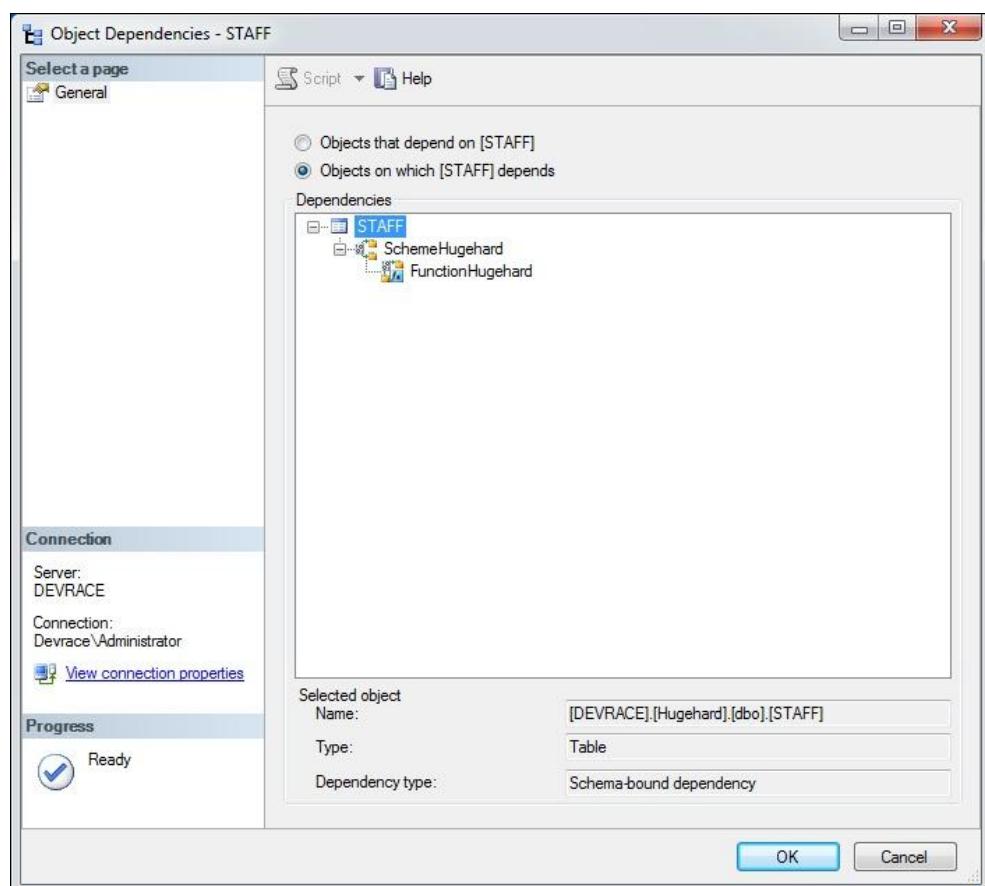


Рис. 5.38. Список объектов, от которых зависит таблица STAFF

◆ внешние ключи таблиц, ссылающихся на этот первичный ключ:

- BestDatabase.dbo.REFCTRCURR: FK1_REFCTRCURR,
- BestDatabase.dbo.REFREG: FK_REFREG.

Можете отобразить ограничения таблицы PEOPLE. Там будут отображены все ограничения, включая FOREIGN KEY и значения по умолчанию.

* * *

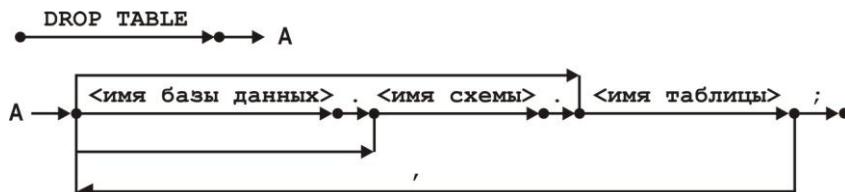
Чтобы удалить таблицу, необходимо либо удалить объекты, ссылающиеся на данную таблицу, либо удалить в этих объектах ссылки на данную таблицу.

5.7.2. Удаление таблицы оператором *DROP TABLE*

Синтаксис оператора *DROP TABLE* представлен в листинге 5.18 и в соответствующем R-графе (граф 5.25).

Листинг 5.18. Синтаксис оператора удаления таблицы *DROP TABLE*

```
DROP TABLE [[<имя базы данных>.]<имя схемы>.<имя таблицы>
[, [[<имя базы данных>.]<имя схемы>.<имя таблицы>] ...;
```



Граф 5.25. Синтаксис оператора удаления таблицы *DROP TABLE*

Как и в случае оператора создания таблицы, здесь также можно задать имя базы данных и имя схемы, отделяя их символом точки. Если в операторе не указано имя базы данных, то таблица удаляется из текущей базы данных — из базы данных, которая была установлена в последнем операторе *USE*. Если не указано имя схемы, то таблица удаляется из схемы базы данных по умолчанию — обычно это схема *dbo*.

В одном операторе можно удалить несколько таблиц, отделяя их имена запятыми.

5.7.3. Удаление таблицы диалоговыми средствами Management Studio

Чтобы удалить таблицу с использованием диалоговых средств Management Studio, нужно в окне **Object Explorer** по имени удаляемой таблицы щелкнуть правой кнопкой мыши и выбрать в контекстном меню элемент **Delete**. Появится окно удаления объекта, как показано на рис. 5.39.

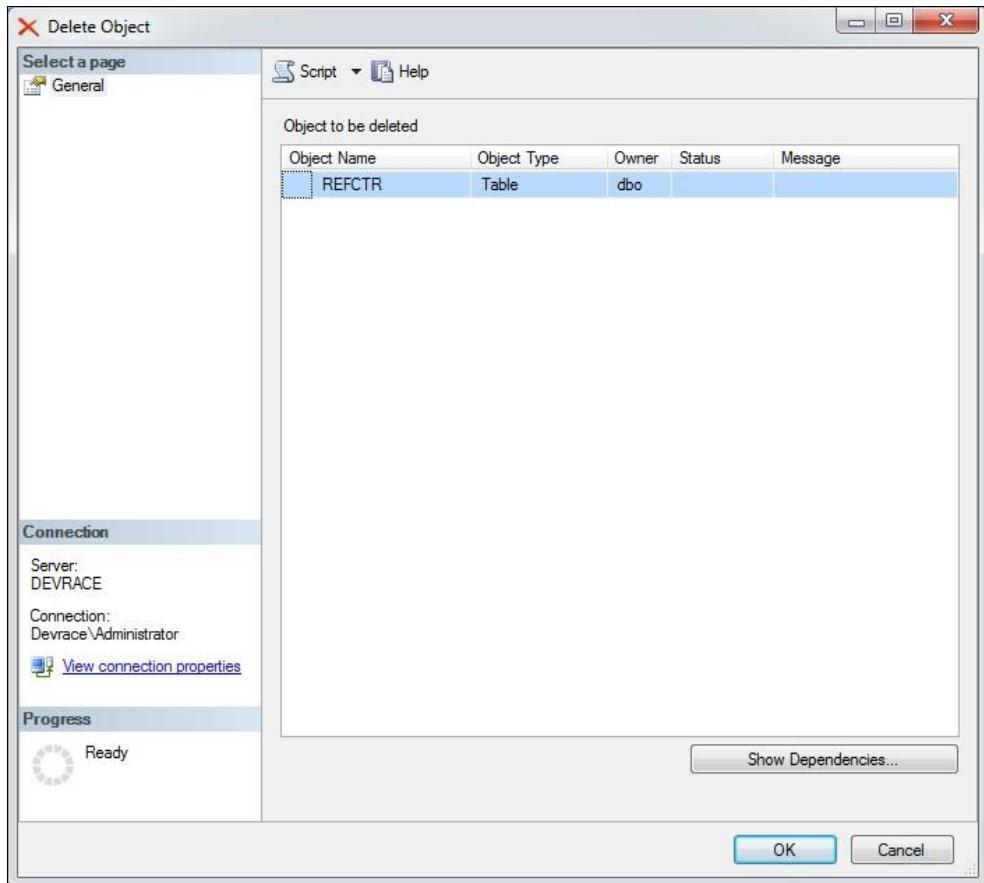


Рис. 5.39. Окно удаления объекта базы данных

Чтобы просмотреть зависимости удаляемой таблицы, нужно щелкнуть по кнопке **Show Dependencies**. Будет отображено окно, которое было показано ранее на рис. 5.36.

Если вы захотите удалить таблицу, для которой существуют внешние зависимости, т. е. в базе данных присутствуют объекты, зависящие от данной таблицы, то вы получите сообщение об ошибке, как это показано на рис. 5.40.

Чтобы просмотреть сообщение о причинах невозможности удаления таблицы, щелкните мышью по гиперссылке в столбце **Message**. Появится сообщение, показанное на рис. 5.41.

Здесь сообщается, что невозможно удалить объект **dbo.REFCTR**, потому что на него ссылается ограничение внешнего ключа.

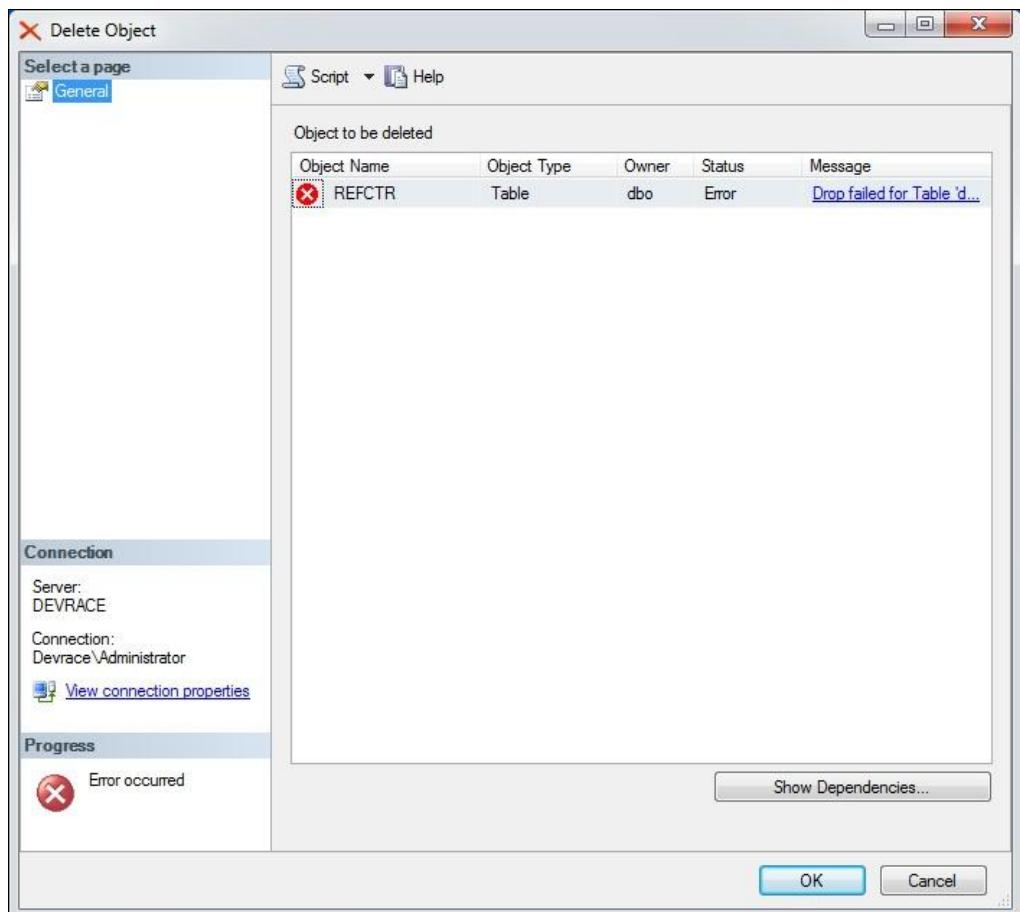


Рис. 5.40. Сообщение об ошибке при попытке удаления объекта базы данных, на который имеются ссылки других объектов базы данных

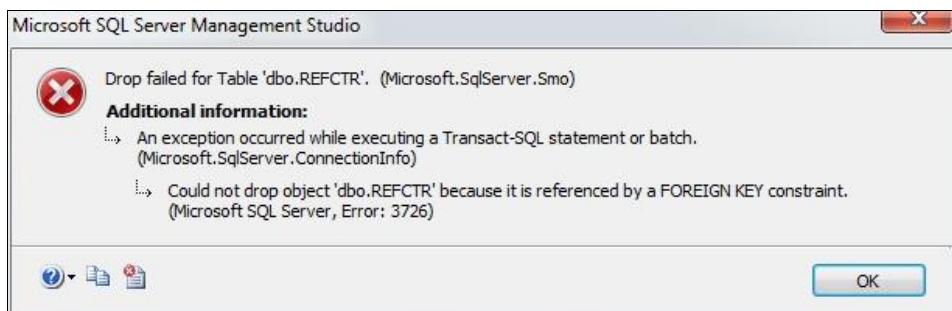


Рис. 5.41. Сообщение о причинах невозможности удаления таблицы

5.8. Изменение характеристик таблиц

Для изменения характеристик таблиц можно использовать оператор `ALTER TABLE` языка Transact-SQL или диалоговые средства Management Studio.

Можно выполнять следующие изменения.

- ◆ *Изменить имя таблицы.* Возможно только в Management Studio. При этом если на данную таблицу ссылаются другие дочерние таблицы, например, посредством внешнего ключа, то в этих таблицах происходит автоматическое изменение имени родительской таблицы.
- ◆ *Удалить столбец таблицы.* Нельзя удалить столбец, который:
 - входит в состав такого ограничения таблицы, как первичный, уникальный, внешний ключ, `CHECK`;
 - используется для получения значения вычисляемого столбца;
 - является столбцом с типом данных `UNIQUEIDENTIFIER` и с атрибутом `ROWGUIDCOL` в таблице, использующей файловые потоки.
- ◆ *Добавить новый столбец.* Здесь никаких ограничений на добавление столбцов нет, если не считать, что количество столбцов одной таблицы не должно превышать 1024 или с учетом разреженных столбцов 30 000.
- ◆ *Изменить имя столбца* — только в Management Studio. Однако такое изменение невозможно, если столбец присутствует в ограничении `CHECK`, в выражении для получения значения вычисляемого столбца. Если же изменяемый столбец входит в состав первичного или уникального ключа, на который ссылаются внешние ключи других или той же самой таблицы, то изменение его имени автоматически осуществляется во всех внешних ключах дочерних таблиц. Это изменение также автоматически учитывается и в самих ограничениях первичного, уникального или внешнего ключа.
- ◆ *Изменить характеристики столбцов.* В принципе есть возможность изменять в определенных пределах тип данных столбца, однако далеко не всегда это бывает возможным. Примеры мы рассмотрим.
- ◆ *Добавлять, изменять и удалять ограничения таблицы.* Если для ограничения `CHECK` нет проблем с изменением или удалением, то нельзя просто удалить ограничение первичного или уникального ключа, если на этот ключ ссылаются внешние ключи другой или той же самой таблицы.

5.8.1. Изменение таблиц при использовании оператора Transact-SQL

Синтаксис оператора `ALTER TABLE` с довольно большими упрощениями представлен в листинге 5.19 и соответствующем R-графе (граф 5.26—5.32).

Листинг 5.19. Синтаксис оператора изменения таблицы ALTER TABLE

```
ALTER TABLE [ [<имя базы данных>.]<имя схемы>.]<имя таблицы>
```

```
{ <изменение столбца>
  | <добавление столбца>
  | <добавление вычисляемого столбца>
  | <добавление ограничения>
  | <удаление столбца или ограничения> }
```

```
<изменение столбца> ::=
```

```
ALTER COLUMN <имя столбца>
[ <тип данных> [ COLLATE <порядок сортировки> ] ]
[ NULL | NOT NULL ] [ SPARSE ]
```

```
<добавление столбца> ::=
```

```
ADD <имя столбца> <тип данных>
[ FILESTREAM ]
[ COLLATE <порядок сортировки> ]
[ NULL | NOT NULL ]
[ DEFAULT <выражение>
  | IDENTITY [ (<начальное значение>, <приращение>) ] ]
[ ROWGUIDCOL ]
[ <ограничение столбца> ... ]
[ SPARSE ]
```

```
<ограничение столбца> ::=
```

```
[ CONSTRAINT <имя ограничения> ]
{ <первичный ключ>
  | <уникальный ключ>
  | <внешний ключ>
  | <ограничение CHECK>
}
```

```
<добавление вычисляемого столбца> ::=
```

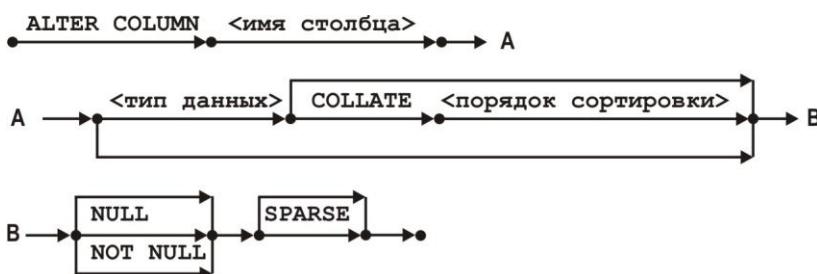
```
ADD <имя столбца> AS <выражение>
[ PERSISTED [ NOT NULL ] ]
[ <ограничение столбца> ]
```

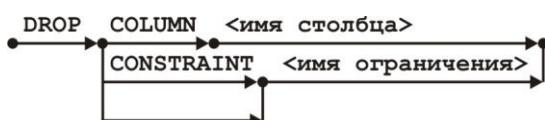
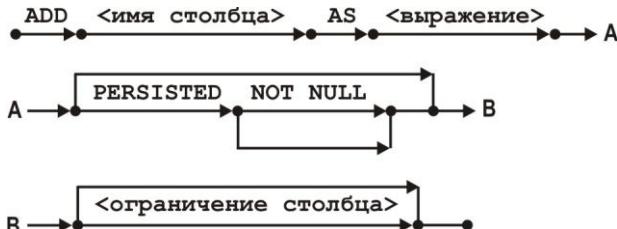
```
<добавление ограничения> ::=
```

```
ADD [ CONSTRAINT <имя ограничения> ]
{ <первичный ключ>
  | <уникальный ключ>
  | <внешний ключ>
  | <ограничение CHECK>
}
```

```
<удаление столбца или ограничения> ::=
```

```
DROP { COLUMN <имя столбца>
  | [ CONSTRAINT ] <имя ограничения> }
```





Многие конструкции нам с вами уже известны по синтаксису оператора создания таблицы CREATE TABLE. Некоторые детали в листинге 5.19 и в графах не показаны. Отсутствуют подробные описания ограничений столбца и таблицы. При необходимости их можно посмотреть в соответствующих листингах в начале этой главы.

5.8.1.1. Имя таблицы

Как и в случаях создания или удаления таблицы, в операторе должно быть указано как минимум имя изменяемой таблицы. Можно также задать имя базы данных и имя схемы, отделяя их символом точки. Если в операторе не указано имя базы данных, то изменяется таблица в текущей базе данных — в той базе, которая была установлена в последнем операторе USE. Если не указано и имя схемы, то изменяемая таблица выбирается в схеме базы данных по умолчанию — чаще всего схема dbo.

5.8.1.2. Изменение столбца

Для изменения столбца в операторе ALTER TABLE используется предложение ALTER COLUMN. Здесь (в принципе) можно изменить тип данных, порядок сортировки (ключевое слово COLLATION) и характеристики NULL / NOT NULL.

5.8.1.3. Изменение типа данных

Нельзя изменить тип данных никоим образом, если столбец входит в состав первичного, уникального ключа, даже независимо от того, ссылаются ли на эти ограничения внешние ключи других таблиц. Нельзя изменить тип данных внешнего ключа, поскольку появится несоответствие между внешним ключом и первичным (уникальным) ключом родительской таблицы. Нельзя менять тип данных столбца, входящего в состав какого-либо индекса. Также нельзя менять размерность у столбцов, которые используются в вычисляемых столбцах. Можно лишь изменять количество символов в типах данных VARCHAR и NVARCHAR, но только в сторону увеличения, если такой столбец не входит в состав ни одного из ключей или индекса или присутствует в выражении для вычисляемого столбца. Также можно изменять в сторону увеличения количество знаков в числовых типах данных, если соответствующие столбцы не входят в состав первичного, уникального или внешнего ключа и не используются для получения значения вычисляемого столбца.

Например, следующие попытки изменения таблиц базы данных BestDatabase вызовут соответствующие ошибки.

Первый пример. Попытка изменить размер даже в сторону увеличения строкового типа данных, являющегося первичным ключом таблицы:

```
USE BestDatabase;
GO
ALTER TABLE REFCTR
    ALTER COLUMN CODCTR CHAR(4);
GO
```

Такие операторы вызывают бурю негодования:

```
Msg 5074, Level 16, State 1, Line 1
The object 'PK_REFCTR' is dependent on column 'CODCTR'.
Msg 5074, Level 16, State 1, Line 1
The object 'FK1_REFCTRCURR' is dependent on column 'CODCTR'.
Msg 5074, Level 16, State 1, Line 1
The object 'FK_REFREG' is dependent on column 'CODCTR'.
Msg 4922, Level 16, State 9, Line 1
ALTER TABLE ALTER COLUMN CODCTR failed because one or more objects access this
column.
```

Здесь сообщается, что от этого столбца зависит объект (ограничение первичного ключа этой же таблицы) PK_REFCTR, объекты FK1_REFCTRCURR (внешний ключ таблицы валют) и FK_REFREG (внешний ключ таблицы регионов).

Другой пример. Попытка изменить в сторону увеличения характеристики числового типа данных DECIMAL для столбца, используемого для получения значения вычисляемого столбца:

```
USE BestDatabase;
GO
ALTER TABLE STAFF
    ALTER COLUMN SALARY DECIMAL(10, 3);
GO
```

Будет выдано сообщение:

```
The column 'NET_SALARY' is dependent on column 'SALARY'.
```

Вычисляемый столбец NET_SALARY зависит от столбца SALARY, для которого осуществляется попытка изменить тип данных. По правде сказать, здесь я не вижу реального ограничения на изменение в сторону увеличения или уменьшения размера числового типа данных, поскольку вычисляемый столбец даже не является постоянным в таблице (PERSISTED).

А вот, например, совершенно спокойно можно увеличить размер строкового типа данных переменной длины, если столбец не входит ни в какие контакты с какими-либо ограничениями или вычисляемыми столбцами. В той же таблице REFCTR можно увеличить размер строкового столбца FULLNAME. Кстати, в свое время мне понадобилось подобное увеличение размера в похожей ситуации. Я использовал тогда другую систему управления базами данных, и такое изменение тогда у меня не прошло.

```
USE BestDatabase;
GO
ALTER TABLE REFCTR
    ALTER COLUMN FULLNAME VARCHAR(70);
GO
```

Здесь увеличивается количество символов в столбце FULLNAME со строковым типом данных переменной длины с 65 до 70.

Однако попытка уменьшить размер строкового данного, сделать его меньше, чем размер существующих уже в таблице данных, даст ошибку. Например, при выполнении такого пакета

```
USE BestDatabase;
GO
ALTER TABLE REFCTR
    ALTER COLUMN FULLNAME VARCHAR(60);
GO
```

вы получите следующее сообщение об ошибке, в котором говорится, что возможно усечение строковых или двоичных данных:

```
String or binary data would be truncated.
The statement has been terminated.
```

Вообще говоря, изменение типа данных с моей точки зрения и на основании моей практики работы с базами данных является весьма нездоровым занятием. Единственный разумный вариант — это изменение количества знаков строковых типов данных в сторону увеличения, если нужные вам тексты не помещаются в существующие заданные размеры, или же увеличение размерности числовых типов данных.

5.8.1.4. Изменение порядка сортировки

При изменении характеристик столбца в предложении ALTER COLUMN в конструкции COLLATE вы можете изменить порядок сортировки строкового столбца таблицы, столбца, имеющего тип данных CHAR, VARCHAR, NCHAR или NVARCHAR. Нельзя изменить порядок сортировки у столбца, для которого указан пользовательский тип данных, пусть и строковый.

Для примера изменения порядка сортировки столбца FULLNAME таблицы REFCTR можно выполнить следующие операторы:

```
USE BestDatabase;
GO
ALTER TABLE REFCTR
    ALTER COLUMN FULLNAME VARCHAR(70) COLLATE French_CI_AI;
GO
```

Если вам хочется посмотреть, к каким неприятным последствиям это привело, отобразите строки таблицы REFCTR. Вы увидите, что тексты полного названия стран стали совершенно нечитаемыми.

5.8.1.5. Добавление нового столбца (обычного или вычисляемого)

При добавлении нового столбца в таблицу нет никаких ограничений, учитывая только тот факт, что количество столбцов в таблице не должно превышать 1024 (30000 в случае присутствия разреженных столбцов).

Каждый новый столбец добавляется в конец таблицы. Не существует способов изменения таблицы, при которых менялся бы установленный порядок существующих или добавляемых в таблицу столбцов. Не думаю, что это нас с вами так уж сильно может огорчить. В плане решения задач предметной области нам совершенно все равно, в каком порядке в таблицах располагаются столбцы.

Добавление обычного или вычисляемого столбца выполняется таким же образом, как и в случае создания таблицы. Добавляемый столбец может содержать ограничения, которые задаются точно так же, как и при создании таблицы.

5.8.1.6. Добавление ограничения

В операторе можно добавить новое ограничение. Синтаксис и виды ограничений мы с вами уже хорошо знаем, поскольку рассматривали все это при использовании оператора создания таблицы. Следует только помнить, что таблица не может содержать более одного ограничения первичного ключа.

5.8.1.7. Удаление столбца

Любой столбец можно удалить, только если он не входит в состав ограничения первичного, уникального или внешнего ключа. Нельзя удалить столбец, если он используется для получения значения вычисляемого столбца. Также нельзя удалить столбец, на который есть ссылки в ограничениях CHECK таблицы.

При удалении столбца, чьи данные хранятся в файловом потоке, эти данные не будут удалены с внешнего носителя до перезапуска сервера базы данных.

5.8.1.8. Удаление ограничения

Можно удалить любое ограничение таблицы (первичный, уникальный, внешний ключ, ограничение CHECK), кроме ограничений первичного или уникального ключа, если на эти ограничения ссылаются внешние ключи других таблиц или той же самой таблицы.

ЗАМЕЧАНИЕ

К моему удивлению, я не нашел средств в операторе `ALTER TABLE` для удаления или изменения значения по умолчанию `DEFAULT` для столбца таблицы. Однако в диалоговых средствах Management Studio это можно выполнить очень просто.

5.8.2. Изменение таблиц средствами Management Studio

Как обычно, диалоговые средства и для изменения таблиц много удобнее в использовании, чем оператор `ALTER TABLE`.

5.8.2.1. Изменение имени таблицы

В окне **Object Explorer** нужно раскрыть базу данных, раскрыть папку **Tables**, щелкнуть правой кнопкой мыши по имени таблицы и в контекстном меню выбрать элемент **Rename**. Имя таблицы станет доступным для изменения. После ввода нового имени нужно нажать клавишу `<Enter>`.

Изменять можно имя у любой пользовательской таблицы, даже той, на которую ссылаются другие объекты базы данных. Единственное естественное здесь требование — в схеме базы данных не должно быть таблицы с тем же именем.

5.8.2.2. Изменение столбца

Для изменения характеристик столбца таблицы нужно щелкнуть правой кнопкой мыши по имени таблицы и в контекстном меню выбрать элемент **Design**. В основной части окна Management Studio появится новая вкладка, вкладка проектирования таблицы, содержащая список столбцов таблицы и некоторых их характеристик. В нижней части главного окна располагается вкладка **Column Properties**, в которой описаны характеристики (свойства) выбранного столбца таблицы. В главном меню программы появится элемент **Table Designer**, содержащий десяток элементов меню, позволяющих установить или удалить первичный ключ, добавить новый столбец, удалить существующий столбец и ряд других. Аналогичные кнопки быстрого доступа появятся и на панели инструментов.

На рис. 5.42 показан список столбцов в режиме **Design** таблицы `REFCTR` нашей базы данных `BestDatabase`.

Column Name	Data Type	Allow Nulls
CODCTR	D_CHAR3:char(3)	<input type="checkbox"/>
NAME	D_CHAR60:varchar(60)	<input checked="" type="checkbox"/>
FULLNAME	D_CHAR65:varchar(65)	<input checked="" type="checkbox"/>
CAPITAL	D_CHAR30:varchar(30)	<input checked="" type="checkbox"/>
TELCODE	D_CHAR10:char(10)	<input checked="" type="checkbox"/>
FLAG	D_BLOB:varchar(MAX)	<input checked="" type="checkbox"/>
FLAG_S	D_BLOB:varchar(MAX)	<input checked="" type="checkbox"/>
EMBLEM	D_BLOB:varchar(MAX)	<input checked="" type="checkbox"/>
EMBLEM_S	D_BLOB:varchar(MAX)	<input checked="" type="checkbox"/>
MAP	D_BLOB:varchar(MAX)	<input checked="" type="checkbox"/>
DESCR	D_BLOB:varchar(MAX)	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

Рис. 5.42. Список столбцов таблицы REFCTR во вкладке проектирования

Для каждого столбца указано имя (столбец **Column Name**), тип данных (**Data Type**) и допустимость значения NULL (**Allow Nulls**). В самом левом столбце значок указывает, входит ли данный столбец таблицы в состав первичного ключа.

На рис. 5.43 показаны характеристики столбца CODCTR, отображаемые на вкладке **Column Properties** в нижней части окна проектирования таблицы.

На этой вкладке мы можем видеть имя текущего (выделенного) столбца, допустимость значения NULL (**Allow Nulls**), тип его данных (**Data Type**) и отдельно размер

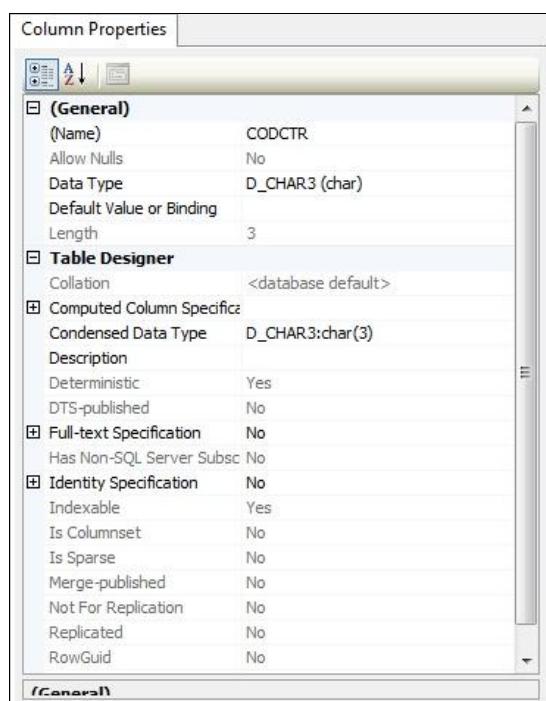


Рис. 5.43. Свойства столбца CODCTR таблицы REFCTR

строкового данного (**Length**), порядок сортировки (**Collation**), значение по умолчанию (**Default Value or Binding**), характеристики `IDENTITY`, если они установлены, формулу для получения значения вычисляемого столбца (в папке **Computed Column Specification**, в поле **Formula**) и ряд других.

5.8.2.3. Изменение типа данных

Изменить тип данных столбца можно на вкладке проектирования таблицы (см. рис. 5.42) или на вкладке просмотра свойств столбца (см. рис. 5.43).

На вкладке проектирования таблицы нужно щелкнуть мышью по полю **Data Type** у соответствующего столбца таблицы. Справа появится кнопка со стрелкой вниз. После щелчка по этой кнопке появится раскрывающийся список всех допустимых в этой базе данных системных и пользовательских типов данных.

На рис. 5.44 показан такой список для столбца `FULLNAME` таблицы `REFCTR`.

Column Name	Data Type	Allow Nulls
CODCTR	D_CHAR3:char(3)	<input type="checkbox"/>
NAME	D_CHAR60:varchar(60)	<input checked="" type="checkbox"/>
► FULLNAME	D_CHAR60:varchar(60)	<input checked="" type="checkbox"/>
CAPITAL	D_CHAR40:varchar(40)	<input checked="" type="checkbox"/>
TELCODE	D_CHAR50:varchar(50)	<input checked="" type="checkbox"/>
FLAG	D_CHAR60:varchar(60)	<input checked="" type="checkbox"/>
FLAG_S	D_CHAR65:varchar(65)	<input checked="" type="checkbox"/>
EMBLEM	D_CHAR110:varchar(1...	<input checked="" type="checkbox"/>
EMBLEM_S	D_CHAR120:varchar(1...	<input checked="" type="checkbox"/>
MAP	D_CHAR1000:varchar(...	<input checked="" type="checkbox"/>
	D_BLOB:varchar(MAX)	<input checked="" type="checkbox"/>

Рис. 5.44. Типы данных для столбца `FULLNAME` таблицы `REFCTR`

В этом списке нужно выбрать соответствующий тип данных (в нашем случае для столбца `FULLNAME` это `varchar(50)`). Здесь можно изменить размер строки. Давайте зададим 70. Это число нужно поместить в скобки после типа данных. Напомню, мы чаще всего можем только увеличить размер строкового поля переменной длины. Хотя существуют и другие варианты поведения системы.

Аналогичным образом можно изменить тип данных на вкладке просмотра свойств столбца **Column Properties**. Щелкните мышью по строке **Data Type**. Справа появится кнопка со стрелкой вниз. При щелчке мышью по этой кнопке появится раскрывающийся список допустимых системных и пользовательских типов данных (рис. 5.45).

Из списка выберите тип данных `varchar`. Систем задаст ему размер 50. Следует установить для него размер 70. Для задания нового значения размера нужно чуть ниже в списке свойств щелкнуть мышью по полю **Length** и изменить заданный там размер поля на 70 и нажать клавишу `<Enter>`.

Все выполненные изменения характеристик столбца на вкладке **Column Properties** сразу будут видны в окне списка столбцов таблицы. Однако эти изменения еще не

внесены в базу данных. Чтобы изменения были зафиксированы, нужно щелкнуть правой кнопкой мыши по заголовку вкладки окна списка столбцов таблицы **(Design)** и в контекстном меню выбрать строку **Save REFCTR**. Однако вместо того чтобы убедиться в том, что изменения сохранены, мы получим следующее сообщение (рис. 5.46).

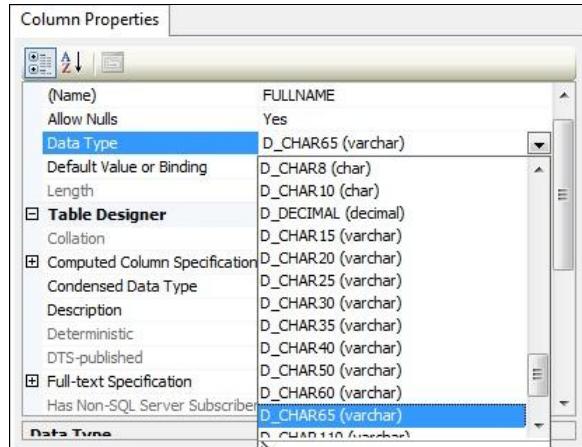


Рис. 5.45. Типы данных для столбца FULLNAME на вкладке Column Properties

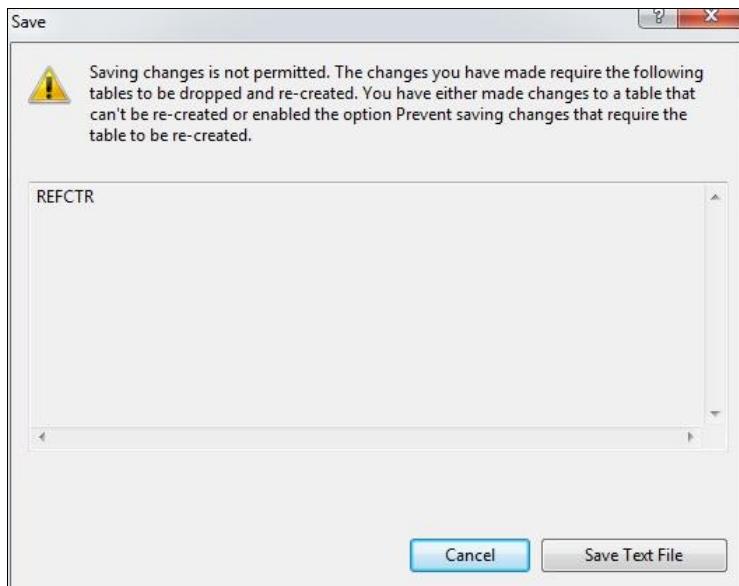


Рис. 5.46. Сообщение о невозможности сохранить изменения характеристик таблицы

Здесь нужно щелкнуть мышью по кнопке **Cancel**, поскольку при существующих установках программы Management Studio (это значения по умолчанию) сохранить выполненные изменения невозможно. Следом появится информационное окно (рис. 5.47).

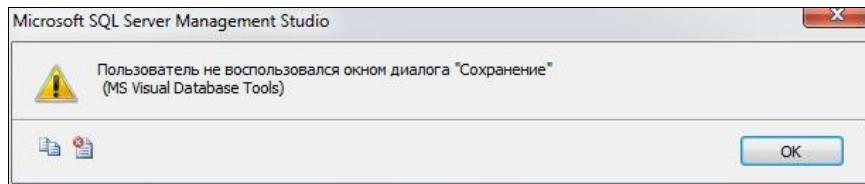


Рис. 5.47. Информационное сообщение

Чтобы действительно было возможно изменять характеристики столбцов таблицы, нужно установить соответствующее свойство. В главном меню программы щелкните мышью по элементу **Tools** и выберите элемент **Options**.

Появится окно задания свойств **Options** (рис. 5.48).

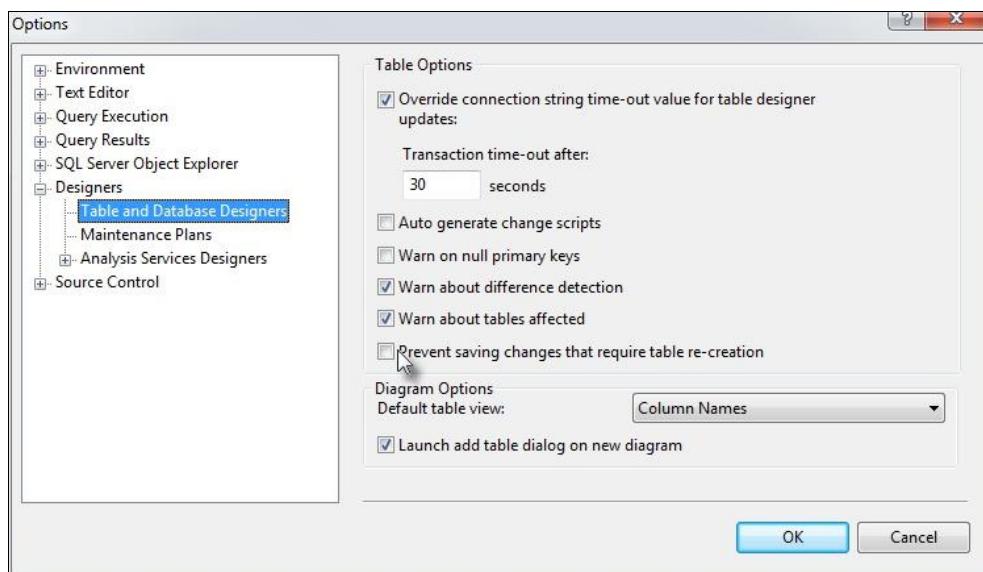


Рис. 5.48. Задание свойств программы на вкладке **Options**

В левой части окна раскройте папку **Designers**. В правой части окна найдите флажок **Prevent saving changes that require table re-creation** (препятствовать сохранению изменений, которые требуют пересоздания таблицы) и снимите его. Затем щелкните мышью по кнопке **OK**.

Теперь для реального сохранения изменений щелкните правой кнопкой мыши по заголовку вкладки окна списка столбцов таблицы (**Design**) и в контекстном меню выберите строку **Save REFCTR**. Появится окно (рис. 5.49), в котором сообщается о том, что будут сохранены (изменены) три таблицы в базе данных. Почему нужно вносить изменения в таблицы REFCTRCURR и REFREG я так и не понял.

В этом окне щелкните мышью по кнопке **Yes**. Внесенное изменение размера столбца будет помещено в базу данных. Закройте окно проектирования таблицы, щелкнув правой кнопкой мыши по заголовку вкладки окна списка столбцов таблицы и выбрав в контекстном меню элемент **Close**.

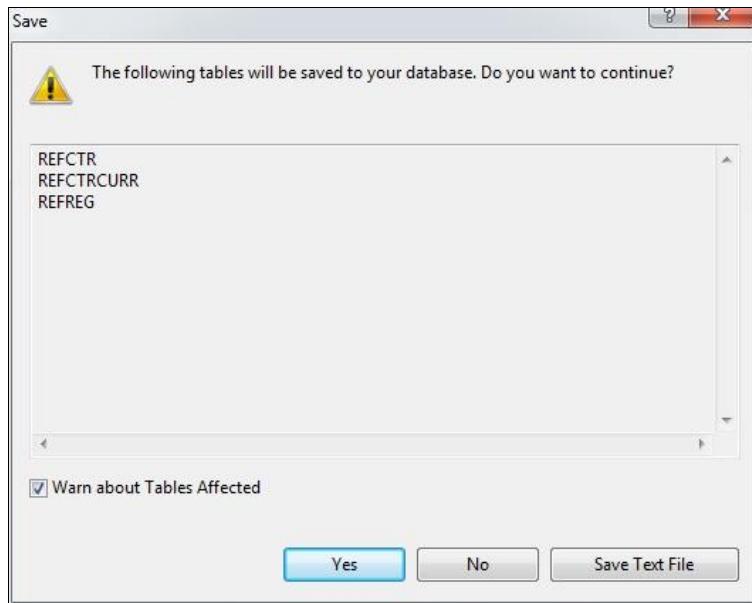


Рис. 5.49. Сообщение об изменяемых таблицах

Если в диалоговых средствах вы попытаетесь слишком уменьшить размер строкового данного, что может привести к усечению существующих в таблице значений, то при попытке сохранить изменения получите следующее предупреждающее диалоговое окно (рис. 5.50):



Рис. 5.50. Диагностическое сообщение о невозможности выполнить изменение размера

Здесь сообщается, что данные могут быть усечены при изменении размера столбца таблицы. Если вы щелкнете по кнопке **Yes**, то изменения все равно будут помещены в базу данных (в отличие от случая использования средств Transact-SQL). В этом случае возможно усечение существующих в таблице данных, т. е. "лишние" правые символы в строках будут потеряны. При щелчке мышью по кнопке **No** изменения не будут сохранены в базе данных.

5.8.2.4. Изменение порядка сортировки

Для изменения порядка сортировки строкового столбца таблицы вызовите окно **Design** для таблицы `REFCTR`. Щелкните по столбцу `FULLNAME`. В нижней части окна (вкладка **Column Properties**) щелкните мышью по строке **Collation**. Значением справа является `<database default>`, т. е. для этого столбца установлен порядок сортировки, используемый для строковых данных базы данных по умолчанию. В нашем случае это `Cyrillic_General_CI_AS`. Правее этого текста появляется кнопка с многоточием. Щелкните по этой кнопке. Появится окно выбора набора сортировки для данного столбца таблицы (рис. 5.51).

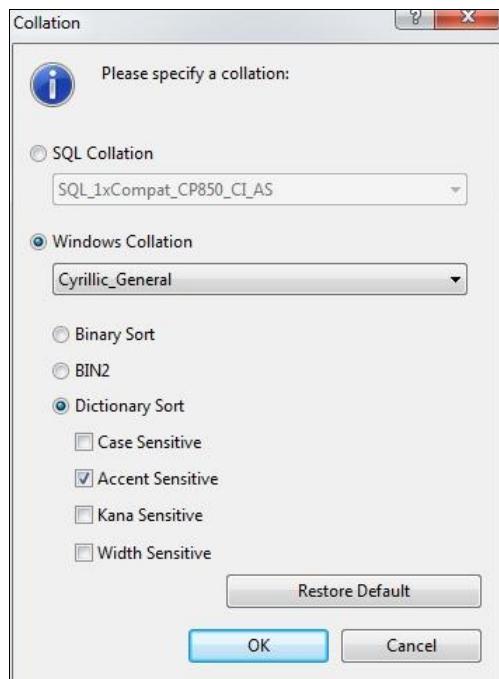


Рис. 5.51. Выбор порядка сортировки для столбца таблицы

Здесь из раскрывающегося списка можно выбрать нужный порядок сортировки, например тот же `French`, который мы ранее использовали в одном из примеров. Для того чтобы получить порядок сортировки `French_CI_AS`, здесь нужно снять флажок **Accent Sensitive** (и другие флажки в группе **Dictionary Sort**, если они были установлены по умолчанию) и щелкнуть по кнопке **OK**.

5.8.2.5. Изменение формулы для вычисляемого столбца

Формулу для получения значения вычисляемого столбца можно очень просто изменить. Для этого в окне **Design** нужно щелкнуть мышью по вычисляемому столбцу. В нижней части главного окна на вкладке **Column Properties** нужно раскрыть группу **Computed Column Specification**. В строке **(Formula)** содержится задание формулы получения значения вычисляемого столбца. Формулу можно изменить в этом поле.

Например, таблица персонала **STAFF** содержит столбец **SALARY**, в котором хранится оклад сотрудника. Столбец **NET_SALARY** является вычисляемым столбцом. В нем содержится сумма зарплаты за вычетом налога 13%. Формула в строке имеет следующий вид:

```
([SALARY]* (0.87))
```

Предположим, что для всех сотрудников организации устанавливается льготное налогообложение, скажем, не 13%, а 5%. Тогда в этом поле нужно заменить существующую формулу на следующую:

```
([SALARY]* (0.95))
```

После этого нужно нажать клавишу **<Enter>** и сохранить изменения таблицы.

5.8.2.6. Добавление нового столбца

На вкладке **Design** (см. ранее рис. 5.42) в конце списка столбцов таблицы есть пустая строка. Туда можно записать характеристики нового добавляемого в таблицу столбца, задав его имя, тип данных и допустимость значения **NULL**. В нижней части окна во вкладке **Column Properties** (см. рис. 5.43) можно установить и другие свойства нового столбца, например, **IDENTITY**.

5.8.2.7. Добавление и изменение ограничений

Каждое из четырех ограничений в диалоговых средствах добавляется через различные диалоговые окна. Похожим образом можно вносить и изменения в существующие ограничения.

Добавление и изменение ограничения первичного ключа

Первичный ключ можно добавить или изменить на вкладке **Design**. Нужно мышью выделить столбец, который вы хотите сделать первичным ключом, и в главном меню выбрать **Table Designer | Set Primary Key**. Можно также щелкнуть по имени столбца правой кнопкой мыши и в контекстном меню выбрать **Set Primary Key**.

Если таблица не имела первичного ключа, то выделенный столбец станет первичным ключом без лишних проверок и выдачи каких-либо сообщений о нарушениях в базе данных.

Если в таблице был уже первичный ключ, и в базе данных не существует таблиц, ссылающихся на этот первичный ключ, то произойдет простая смена первичного ключа без каких-либо сообщений.



Рис. 5.52. Диагностическое сообщение о наличии связей других таблиц с первичным ключом таблицы

Если же в таблице был первичный ключ, на который посредством внешних ключей ссылаются другие таблицы, то будет выдано сообщение, как показано на рис. 5.52.

Сейчас я собираюсь изменить первичный ключ таблицы REFACTIV, которая была создана следующим оператором:

```
CREATE TABLE REFACTIV
( COD      D_CHAR4 NOT NULL,      /* Код вида деятельности */
  NAME     D_CHAR110,           /* Наименование вида деятельности */
  CONSTRAINT PK_REFACTIV
    PRIMARY KEY (COD)
);
```

Первичный ключ теперь устанавливается для столбца NAME.

В диалоговом окне задается вопрос, хотите ли вы удалить существующие связи.

Если вы щелкнете мышью по кнопке **Нет**, то никаких изменений сделано не будет.

Если же вы щелкнете по кнопке **Да**, то появится следующее диалоговое окно, в котором запрашивается, нужно ли вносить изменения в две таблицы базы данных (рис. 5.53). Здесь речь в первую очередь идет об удалении внешнего ключа в таблице ORGACTIV, который ссылается на первичный ключ изменяемой таблицы.

Если щелкнуть мышью по кнопке **Yes**, то внешний ключ у подчиненной таблицы ORGACTIV будет удален, а для изменяемой таблицы REFACTIV будет установлен новый первичный ключ.

Однако с целью проверки поведения системы я поместил в таблицу REFACTIV строки, у которых дублируются значения столбца NAME. Поскольку первичный ключ не допускает дублирующих значений, то после щелчка по кнопке **Yes** будет отображено следующее диалоговое окно, сообщающее об ошибке (рис. 5.54).

В этом случае нужно только щелкнуть мышью по кнопке **OK**. Первичный ключ у таблицы изменен не будет.

Если же в подобном случае в таблице не будет дублирующих значений реквизита для нового назначаемого первичного ключа, изменение будет выполнено без ошибок.



Рис. 5.53. Диалоговое окно, запрашивающее изменение двух других таблиц

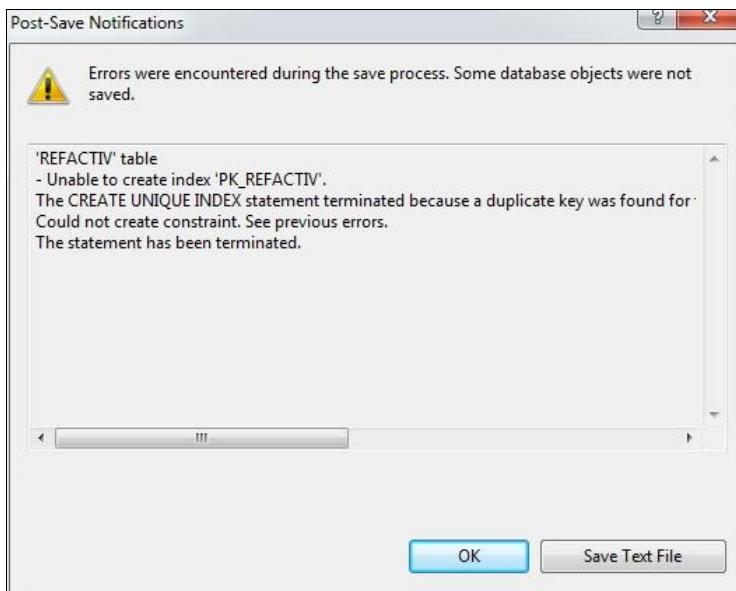


Рис. 5.54. Диалоговое окно, сообщающее о наличии дубликатов значений первичного ключа

Добавление составного первичного ключа

Если добавляемый первичный ключ должен содержать в своем составе более одного столбца или вы изменяете первичный ключ, добавляя к нему еще столбцы, то здесь нужно использовать окно **Indexes/Keys**. Давайте сейчас на примере той же таблицы REFACTIV добавим в состав первичного ключа и столбец NAME.

Щелкнув правой кнопкой мыши по имени этой таблицы, вызовите окно **Design**. Выберите в меню **Table Designer | Indexes/Keys** или щелкните правой кнопкой мыши по имени любого столбца таблицы и в контекстном меню выберите элемент **Indexes/Keys**. Появится окно **Indexes/Keys** (рис. 5.55).

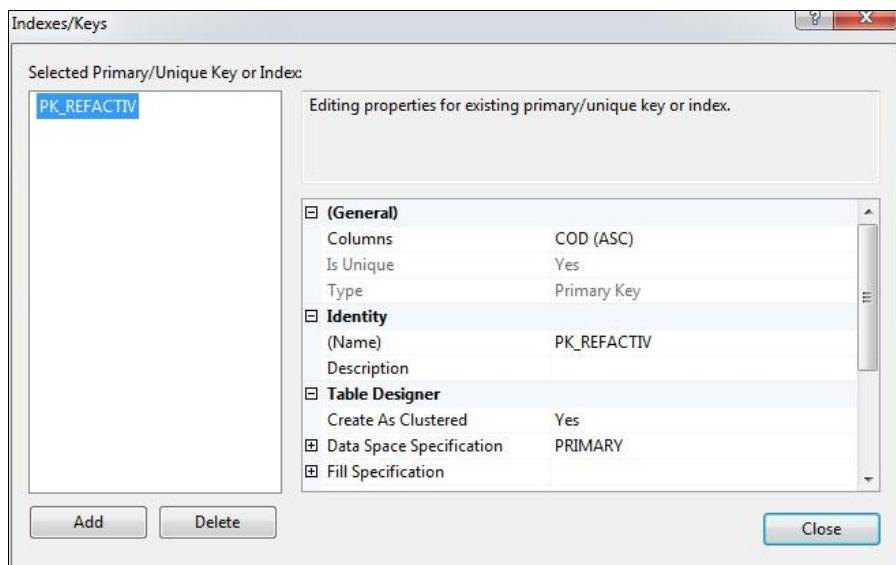


Рис. 5.55. Список ключей таблицы REFACTIV

Чтобы добавить в состав первичного ключа таблицы еще один столбец, выделите мышью строку **Columns**, щелкните в правой части этой строки по кнопке . Следующим будет окно, описывающее столбцы индекса первичного ключа данной таблицы (рис. 5.56).

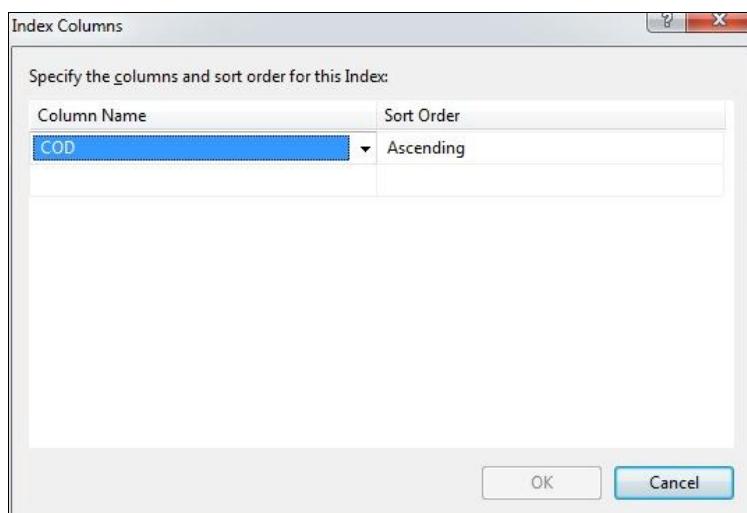


Рис. 5.56. Список столбцов первичного ключа таблицы REFACTIV

Добавьте в список столбец NAME, выбрав его из раскрывающегося списка, и установите для него упорядоченность по возрастанию значений (**Ascending**). Список станет следующим (рис. 5.57).

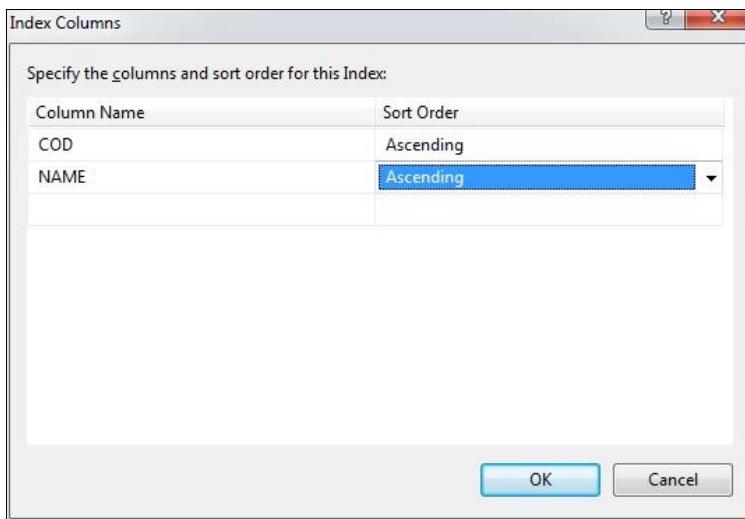


Рис. 5.57. Измененный список столбцов первичного ключа таблицы REFACTIV

Щелкните мышью по кнопке **OK**. Появится предупреждающее окно, в котором сообщается, что прежде чем изменять ограничение первичного или уникального ключа, нужно удалить существующее (рис. 5.58). Вас спрашивают, хотите ли вы удалить существующие отношения.



Рис. 5.58. Предупреждающее сообщение о необходимости удаления существующего первичного ключа таблицы

Щелкните мышью по кнопке **Да**. В окне **Indexes/Keys** (см. ранее рис. 5.55) щелкните по кнопке **Close**. В первичный ключ таблицы будет добавлен столбец NAME. Чтобы реально выполнить изменения, сохраните таблицу.

Если после этого просмотреть ключи подчиненной (бывшей подчиненной) таблицы ORGACTIV, то мы не обнаружим в ней внешнего ключа, ссылающегося на измененную нами таблицу REFACTIV.

Как вы понимаете, подобные изменения являются весьма надежным средством разрушить существующую базу данных.

Добавление и изменение ограничения уникального ключа

Таблица может содержать произвольное количество уникальных ключей (`UNIQUE`). На конкретном примере создадим уникальный ключ. Сделаем это для таблицы, описывающей людей, — `PEOPLE`. Для этой таблицы у нас существует искусственный автоинкрементный (`IDENTITY`) первичный ключ, `cod`. В принципе для этой таблицы есть возможность задания и довольно сложного первичного ключа, который будет состоять из фамилии, имени, отчества и даты рождения. Повторение записей с одинаковыми значениями такого первичного ключа в базе данных, даже содержащей сведения об очень большом количестве людей, под большим вопросом. В природе, конечно, существуют полные тезки, но чтобы они имели и одинаковую дату рождения — такое может встретиться ну очень уж редко.

Следует заметить, что создание больших по размеру первичных ключей не является хорошей практикой. Это увеличивает объем используемой внешней памяти, поскольку для первичного ключа создается индекс, что в случае большого размера ключа ухудшает производительность системы. Если же еще на такой первичный ключ должны ссылаться внешние ключи других таблиц, то про производительность можно будет забыть.

Добавим в таблицу `PEOPLE` уникальный ключ, включающий в себя все эти перечисленные столбцы: `NAME3` (фамилия), `NAME1` (имя), `NAME2` (отчество) и `BIRTHDAY` (дата рождения).

Щелкните правой кнопкой мыши по имени таблицы `PEOPLE` в **Object Explorer** и в контекстном меню выберите элемент **Design**. В главном окне появится вкладка, содержащая список столбцов таблицы.

Выберите в главном меню **Table Designer | Indexes/Keys** или щелкните правой кнопкой мыши по любому столбцу в списке столбцов таблицы и выберите в контекстном меню элемент **Indexes/Keys**.

Появится окно просмотра списка ключей и индексов таблицы (рис. 5.59).

В таблице присутствует только первичный ключ. Чтобы добавить новое ограничение уникального ключа или новый индекс, нужно в левой части окна внизу щелкнуть мышью по кнопке **Add**. Появится новое ограничение в этом списке. Имя нового ограничения отмечается символом "звездочка" справа. Для нашего ограничения это имя **IX_PEOPLE***. В правой части окна будут перечислены его характеристики по умолчанию, которые мы сейчас и будем изменять (рис. 5.60).

В правой части окна щелкните мышью по строке **Columns**. Справа в строке появится кнопка с многоточием . Щелкните мышью по этой кнопке. В результате откроется диалоговое окно, описывающее столбцы таблицы, входящие в состав индекса, используемого для первичного ключа таблицы. В списке будет только столбец `COD`.

Внесите в список изменения, выбирая из раскрывающегося списка **Column Name** поочередно имена столбцов (заменив вначале столбец `COD`): `NAME3`, `NAME1`, `NAME2` и

BIRTHDAY. Значения столбца **Sort Order** (порядок сортировки) для всех элементов ключа можно оставить **Ascending** (по возрастанию значений). Список примет следующий вид (рис. 5.61).

Щелкните по кнопке **OK**. В ограничение ключа будут помещены все перечисленные столбцы. Затем в первоначальном списке характеристик создаваемого индекса

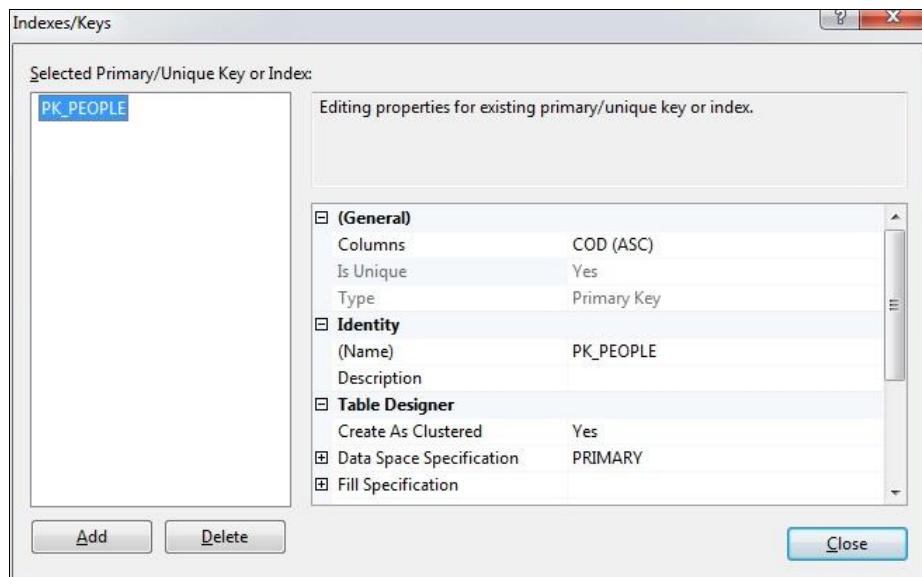


Рис. 5.59. Список ключей и индексов таблицы PEOPLE

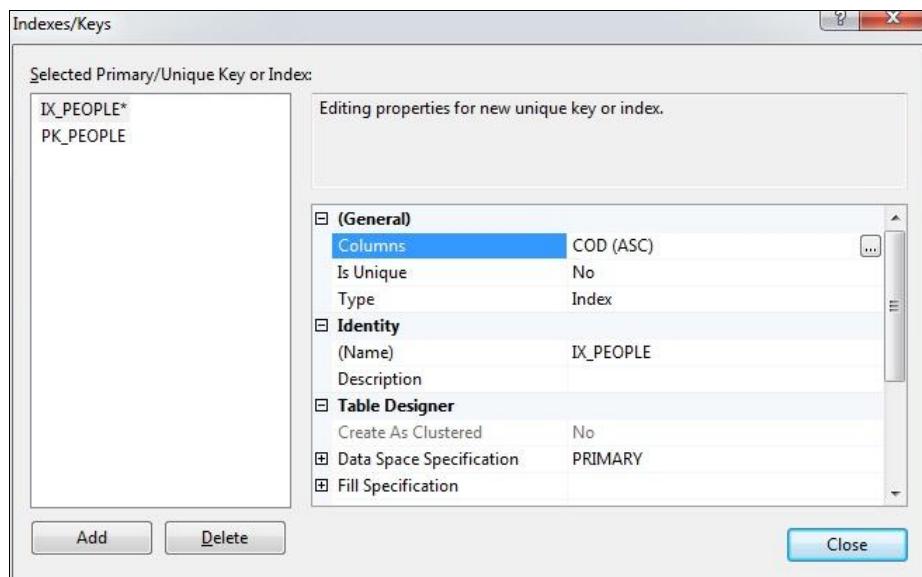


Рис. 5.60. Список характеристик вновь создаваемого ограничения

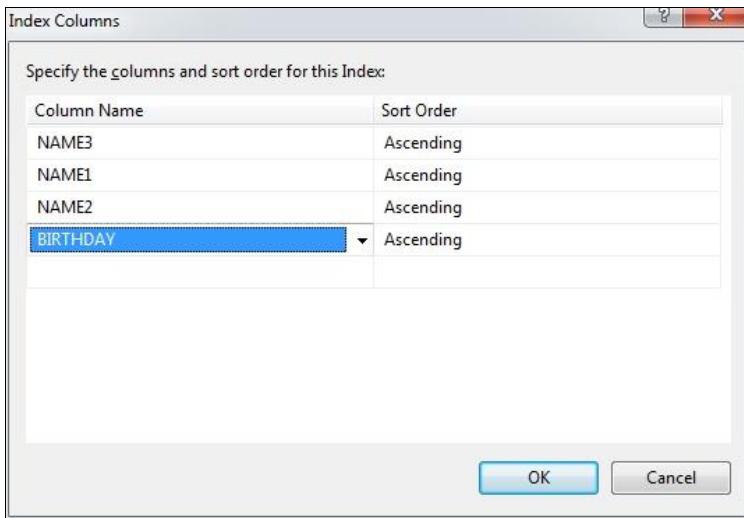


Рис. 5.61. Список элементов столбцов создаваемого ограничения уникального ключа

(пока система рассматривает всю нашу деятельность, как создание индекса) щелкните мышью по строке **Type** и справа из раскрывающегося списка выберите значение **Unique Key**. Теперь это ограничение уникального ключа, а не индекс.

Задайте новое имя ограничению, щелкнув по строке **(Name)** и изменив в правой части имя на **UK_PEOPLE**.

Для завершения добавления в таблицу уникального ключа щелкните мышью в окне по кнопке **Close**.

Чтобы поместить в таблицу выполненное добавление, сохраните таблицу. Чтобы увидеть сделанные изменения, обновите в **Object Explorer** список, щелкнув по имени таблицы правой кнопкой мыши и выбрав в контекстном меню **Refresh**. После этого раскройте базу данных **BestDatabase**, раскройте папку **Tables**, раскройте таблицу **PEOPLE** и ее папку **Keys**. В списке ключей таблицы можно будет увидеть и вновь созданное ограничение уникального ключа **UK_PEOPLE**.

Добавление и изменение ограничения внешнего ключа

Добавим опять же в таблицу людей **PEOPLE** ограничение внешнего ключа. Пусть это будет внешний ключ, ссылающийся на таблицу стран. Иными словами, мы собираемся для каждого человека указывать и страну проживания.

Щелкните правой кнопкой в **Object Explorer** по имени таблицы **PEOPLE** и в контекстном меню выберите элемент **Design**.

Вначале добавим в таблицу новый столбец **CODSTR**, который и станет внешним ключом, ссылающимся на таблицу стран. Для этого в последней пустой строке списка столбцов таблицы введем имя столбца, **CODSTR**, в поле **Data Type** выберем из раскрывающегося списка значение пользовательского типа данных — **D_CHAR3:char(3)**. Это пользовательский тип данных, который установлен для столбца первичного ключа таблицы стран. Напоминаю, типы данных первичного ключа родительской

таблицы и ссылающегося на него внешнего ключа дочерней таблицы должны полностью совпадать. Установим допустимость значения **NULL** (поле **Allow Nulls**). Впрочем, это значение и так устанавливается по умолчанию.

Теперь список столбцов таблицы выглядит так, как показано на рис. 5.62.

Column Name	Data Type	Allow Nulls
COD	D_INTEGER:int	<input checked="" type="checkbox"/>
NAME1	D_CHAR15:varchar(15)	<input checked="" type="checkbox"/>
NAME2	D_CHAR15:varchar(15)	<input checked="" type="checkbox"/>
NAME3	D_CHAR20:varchar(20)	<input checked="" type="checkbox"/>
BIRTHDAY	D_DATE:date	<input checked="" type="checkbox"/>
SEX	D_CHAR1:char(1)	<input checked="" type="checkbox"/>
FULLNAME		<input checked="" type="checkbox"/>
CODMOTHER	D_INTEGER:int	<input checked="" type="checkbox"/>
CODFATHER	D_INTEGER:int	<input checked="" type="checkbox"/>
CODOTHERHALF	D_INTEGER:int	<input checked="" type="checkbox"/>
► CODCTR	D_CHAR3:char(3)	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

Рис. 5.62. Новый список столбцов таблицы людей

Щелкнем правой кнопкой мыши по любому столбцу таблицы и в контекстном меню выберем элемент **Relationships**. Можно также в главном меню программы выбрать **Table Designer | Relationships**.

Появится окно, в котором содержится список внешних ключей, связанных с данной таблицей (рис. 5.63). Здесь представлены не только внешние ключи самой таблицы PEOPLE, но также внешние ключи других таблиц базы данных, которые ссылаются на первичный ключ этой таблицы.

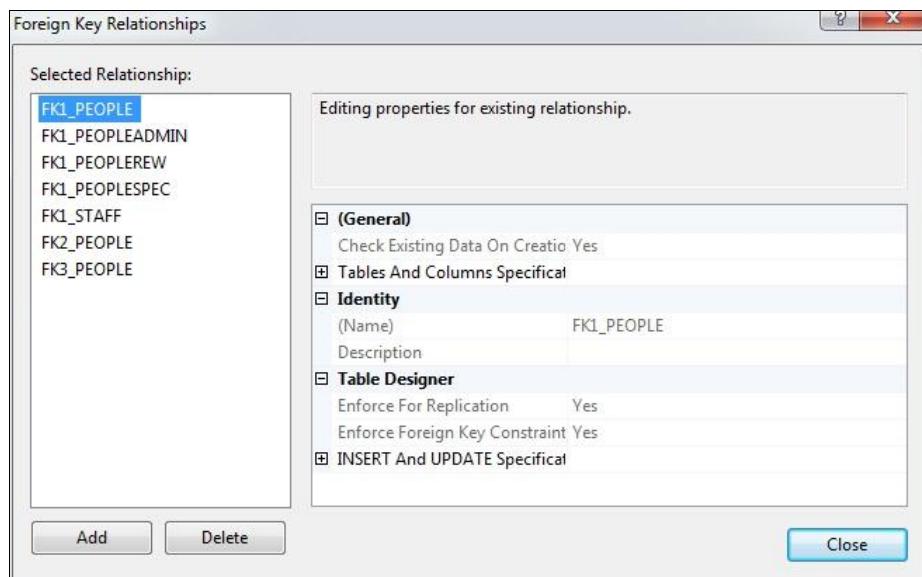


Рис. 5.63. Список внешних ключей, связанных с таблицей PEOPLE

Для создания нового внешнего ключа нужно в левой нижней части окна щелкнуть мышью по кнопке **Add**. В левой части окна в списке появится новое ограничение с именем `FK_PEOPLE_PEOPLE*`. К сожалению, в этом окне ничего изменить больше нельзя. Поэтому нужно щелкнуть мышью по кнопке **Close**.

Далее следует сохранить выполненные изменения для таблицы и закрыть окно **Design**. Теперь в окне **Object Explorer** нужно обновить список, чтобы можно было увидеть внесенные в таблицу изменения. Щелкните правой кнопкой мыши по имени таблицы `PEOPLE` и в контекстном меню выберите **Refresh**.

После этого опять раскройте таблицу `PEOPLE`, раскройте папку ключей (**Keys**), щелкните в появившемся списке ключей правой кнопкой мыши по имени вновь созданного ограничения `FK_PEOPLE_PEOPLE` и в появившемся контекстном меню выберите элемент **Modify**.

Будет выведено окно, содержащее список всех имеющих отношение к данной таблице ключей. Однако теперь мы можем вносить изменения в *любой* внешний ключ (вообще-то говоря, мы с самого начала могли вызвать это окно и более простым способом создавать новые внешние ключи).

Выделите в левой части окна созданный ключ `FK_PEOPLE_PEOPLE`. В правой части окна внесем необходимые изменения в характеристики этого ключа. Вначале изменим его имя. Выделите строку (**Name**) и в правой части строки введите новое имя: `FK4_PEOPLE`.

В этой же части окна выделите мышью строку **Tables And Columns Specification**. В правой части поля появится кнопка с многоточием . Щелкните по этой кнопке. Появится окно, описывающее таблицы и столбцы, принимающие участие в связях внешних и первичных ключей (рис. 5.64).

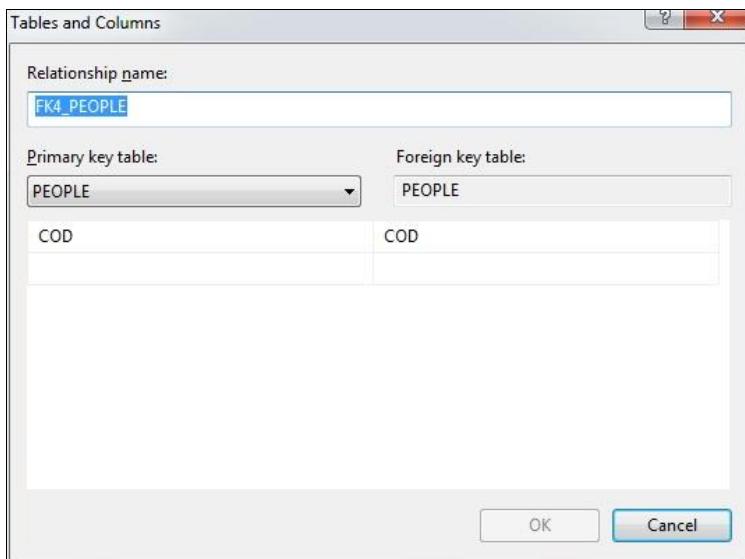


Рис. 5.64. Начальная заготовка таблиц и столбцов для создаваемого внешнего ключа

В левой части окна (**Primary key table**) выберем характеристики родительской таблицы, на первичный ключ которой будет ссылаться наш внешний ключ. Из первого раскрывающегося списка выберем имя таблицы REFCTR, ниже из раскрывающегося списка выберем имя столбца этой таблицы, CODCTR. В правой части окна (**Foreign key table**) выберем имя столбца, входящего в состав внешнего ключа, — CODCTR.

После внесенных изменений окно будет выглядеть так, как это показано на рис. 5.65.

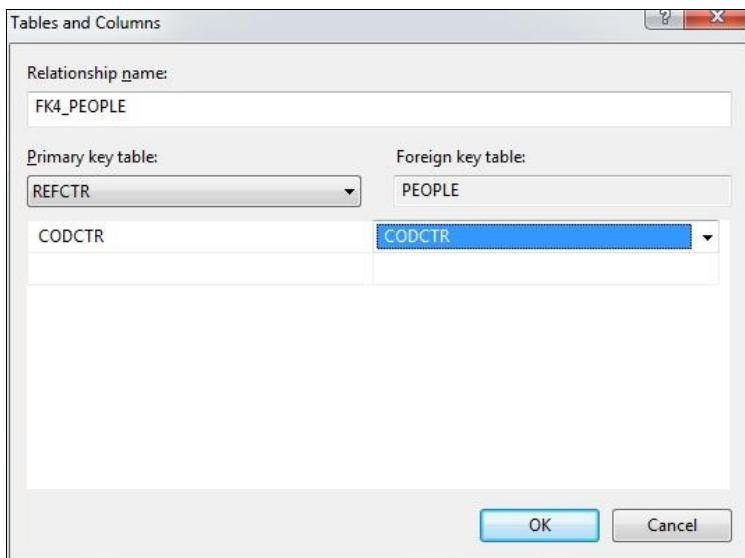


Рис. 5.65. Установленные характеристики внешнего ключа

Щелкните по кнопке **OK**. Теперь остается только для внешнего ключа в окне характеристик установить поведение системы при удалении соответствующей строки родительской таблицы и при изменении соответствующего значения первичного ключа в родительской таблице.

Раскройте узел **INSERT And UPDATE Specification** (неудачное название узла; конечно же, не **INSERT**, а **DELETE**). Для строки **Delete Rule** из раскрывающегося списка выберите **Set Null**, для строки **Update Rule** — **Cascade**. Эти установки соответствуют заданию в языке Transact-SQL следующих предложений:

```
ON DELETE SET NULL  
ON UPDATE CASCADE
```

Напомню, эти предложения означают, что при удалении соответствующей строки родительской таблицы значение внешнего ключа устанавливается в **NULL**, а при изменении значения первичного ключа родительской таблицы это изменение вносится и в значения внешнего ключа всех строк дочерней таблицы.

Полученный вид окна показан на рис. 5.66.

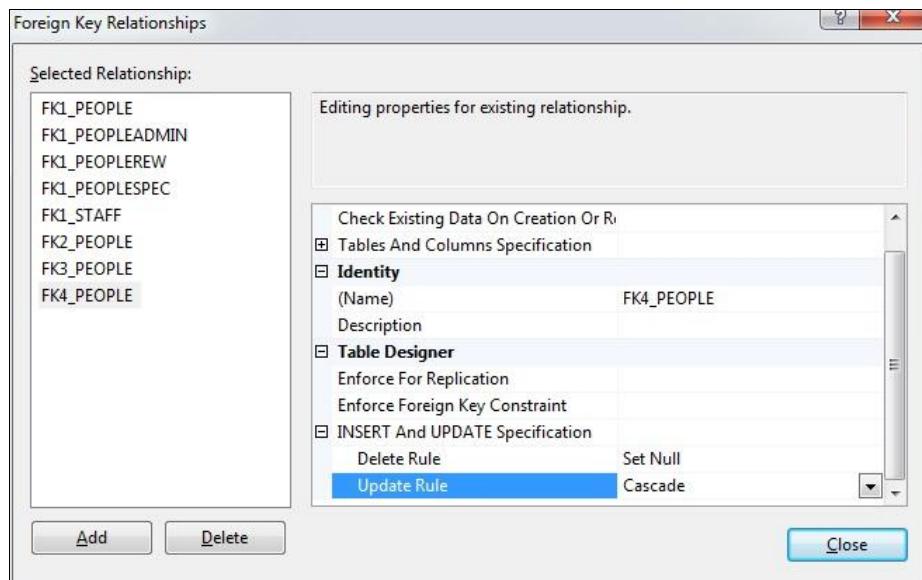


Рис. 5.66. Характеристики созданного нового внешнего ключа для таблицы PEOPLE

Щелкните в этом окне по кнопке **Close**. Новый внешний ключ со всеми необходимыми характеристиками будет создан.

Изменение ограничения CHECK

Сейчас внесем изменения в ограничение CHECK (довольно бессмысленные) в столбец SEX (напомню, что этот столбец содержит сведения про пол человека, а не то, о чём как-то подумали мои студенты) таблицы PEOPLE.

В окне **Object Explorer** раскройте таблицу PEOPLE, раскройте папку ограничений (**Constraints**), щелкните правой кнопкой мыши по имени ограничения CH_PEOPLE и в контекстном меню выберите элемент **Modify**. Появится окно **Check Constraints**, содержащее сведения о существующем единственном ограничении для столбца SEX этой таблицы (рис. 5.67).

Чтобы изменить ограничение, щелкните мышью по строке **Expression**. Здесь содержится формула ограничения:

```
([SEX]='1' OR [SEX]='0')
```

Для изменения ограничения щелкните по кнопке справа от формулы. Появится окно, в котором можно редактировать формулу (рис. 5.68).

Измените формулу на следующую, добавив в список допустимых значений для столбца и символ '2':

```
(([SEX]='1' OR [SEX]='0') OR [SEX]='2')
```

Опять же, следите за правильностью расстановки скобок в формуле. Щелкните по кнопке **OK**. В предыдущем окне щелкните по кнопке **Close**. Сохраните обычным образом изменения таблицы.

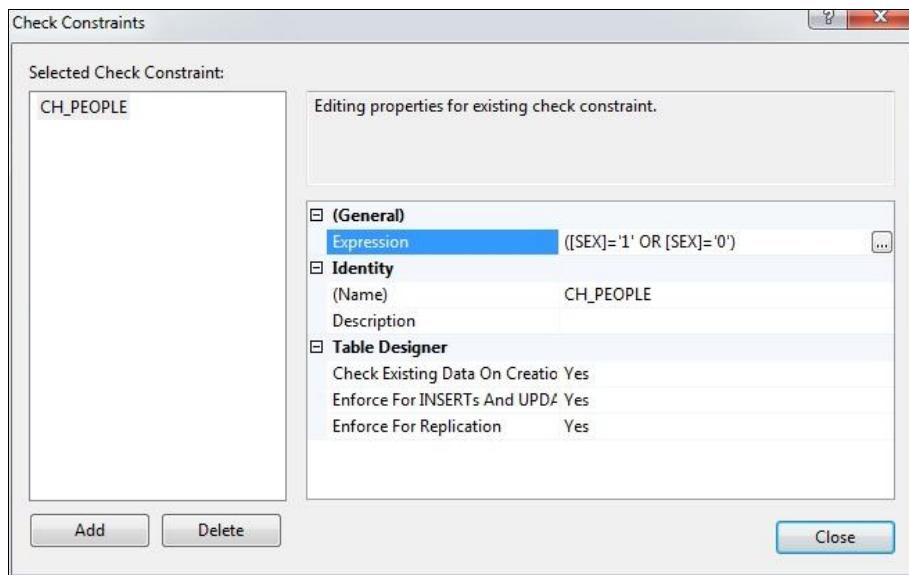


Рис. 5.67. Ограничение CHECK таблицы PEOPLE

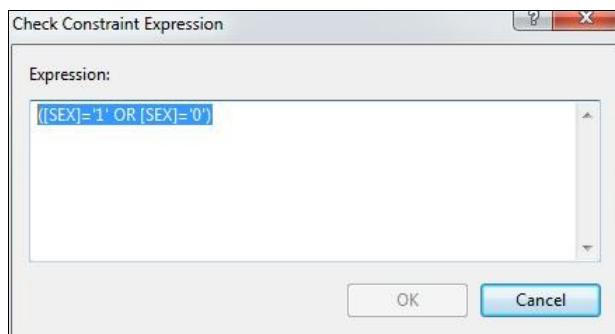


Рис. 5.68. Формула ограничения CHECK таблицы PEOPLE

Добавление ограничения CHECK

Теперь попытаемся добавить в эту же таблицу новое ограничение CHECK. Вначале будем добавлять конструкцию, содержащую правильный оператор SELECT, который, однако, недопустим в подобных ограничениях.

Щелкните правой кнопкой мыши по имени ограничения CH_PEOPLE и в контекстном меню выберите элемент **Modify**. Появится окно **Check Constraints**, как показано на рис. 5.67. Чтобы добавить новое ограничение, щелкните по кнопке **Add**. В списке ограничений появится ограничение с именем CK_PEOPLE*. В правой части окна замените это имя на CH2_PEOPLE. Щелкните мышью по строке **Expression**, щелкните по кнопке справа от формулы. В появившемся окне **Check Constraint Expression** введите выражение для проверки значения столбца CODCTR:

```
(SELECT COUNT(*) FROM REFCTR R WHERE R.CODCTR = CODCTR) > 0
```

Это условие (на самом деле лишнее) требует, чтобы в таблице REFCTR присутствовала строка, первичный ключ которой (CODCTR) равен вводимому или изменяемому значению столбца CODCTR таблицы PEOPLE. Лишним это условие является потому, что при помещении в базу данных строки дочерней таблицы или при изменении значения ее внешнего ключа система проверяет наличие соответствующей строки в родительской таблице, в таблице REFCTR. Таким образом, мы просто дублируем, а точнее, пытаемся дублировать аналогичные действия системы.

Такой же результат можно получить и при использовании функции exists():

```
exists (SELECT * FROM REFCTR R WHERE R.CODCTR = CODCTR)
```

Щелкните в окне **Check Constraint Expression** по кнопке **OK**. После этого закройте окно **Check Constraints**, щелкнув по кнопке **Close**. Попытайтесь сохранить изменения таблицы. Вы получите сообщение, что в ограничении CHECK недопустимо использование вложенного оператора SELECT.

Теперь создадим "правильное" ограничение CHECK для таблицы REFCTR. Сделаем так, чтобы пользователь в поле кода страны мог ввести не менее двух символов.

Вызовите окно **Design** для таблицы REFCTR, щелкните правой кнопкой мыши по имени любого столбца и в контекстном меню выберите элемент **Check Constraints**. Появится окно ограничений CHECK, в котором не будет ни одного ограничения. Щелкните мышью по кнопке **Add**. В списке появится ограничение с именем CK_REFCTR. Измените в поле **(Name)** имя ограничения на CH1_REFCTR. В поле **Expression** введите выражение для проверки количества символов столбца CODCTR:

```
(LEN(CODCTR) > 1)
```

В поле **Description** введите дополнительное текстовое описание, например:

Проверка на количество символов кода страны

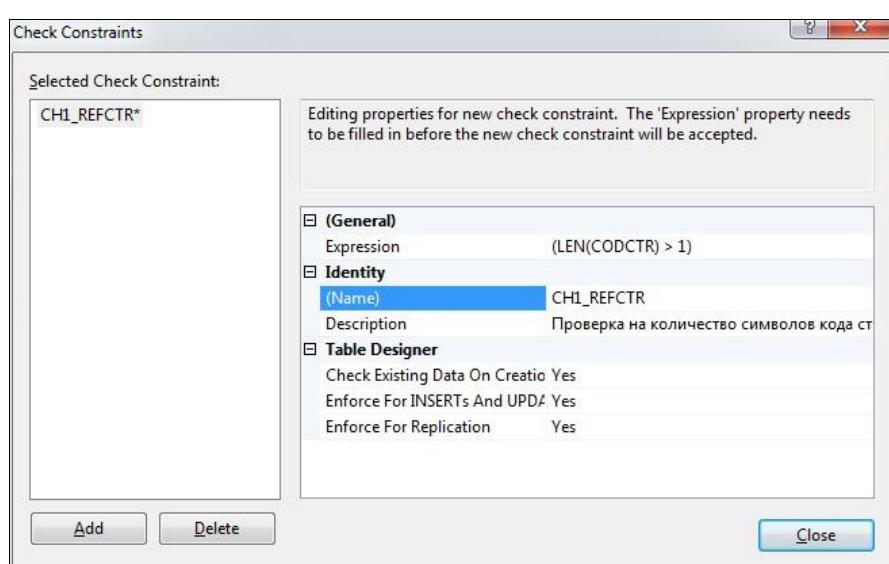


Рис. 5.69. Созданное ограничение CHECK таблицы REFCTR

В результате окно списка ограничений CHECK примет вид, показанный на рис. 5.69.

В этом окне щелкните по кнопке **Close**. Сохраните изменения таблицы REFCTR.

Можете проверить работоспособность нового ограничения. В окне **Object Explorer** щелкните правой кнопкой мыши по имени таблицы и в контекстном меню выберите элемент **Edit Top 200 Rows** (редактировать первые 200 строк). Попытайтесь в любой стране изменить код, оставив один символ. Вы получите сообщение об ошибке.

5.8.2.8. Удаление столбца

Столбец можно легко удалить, если он не входит в состав ограничения первичного, уникального или внешнего ключа, не используется для получения значения вычисляемого столбца и на него нет ссылок в ограничениях CHECK.

Если столбец входит в состав первичного ключа, и на эту таблицу ссылаются другие таблицы базы данных, то будет выдано сообщение, аналогичное тому, как было показано на рис. 5.52.

Когда столбец используется при получении значения вычисляемого столбца, то при его удалении никаких сообщений, как обычно, не выдается, но при попытке сохранить изменения таблицы будет появляться диалоговое окно с соответствующим предупреждающим сообщением.

Например, давайте в таблице PEOPLE удалим столбец NAME1, который используется в выражении для вычисляемого столбца FULLNAME (здесь мы не учитываем использование этого столбца в созданном только что ограничении уникального ключа). Столбец FULLNAME в таблице PEOPLE определен следующим образом:

```
FULLNAME AS          /* Вычисляемый столбец */  
    (NAME3 + ' ' + NAME1 + ' ' + NAME2),
```

На вкладке **Design** удалим столбец NAME1, щелкнув по его имени правой кнопкой мыши и выбрав в контекстном меню элемент **Delete Column**. При попытке сохранения таблицы появится диалоговое окно (рис. 5.70).

Здесь выдается сообщение о возникновении ошибки при проверке достоверности формулы для вычисляемого столбца FULLNAME.

При щелчке мышью по кнопке **Yes** (что означает требование продолжить попытку сохранения изменений) появляется следующее окно, сообщающее о невозможности изменения таблицы (рис. 5.71).

В окне сообщается, что столбец FULLNAME зависит от столбца NAME1 и говорится, что выполнение соответствующего оператора ALTER TABLE невозможно. Здесь остается только щелкнуть мышью по кнопке **OK**. Удаление столбца выполнено не будет.

Понятно, чтобы удалить такой столбец, нужно вначале исключить его из формул получения вычисляемых столбцов. Давайте проделаем такую процедуру.

Щелкните по столбцу FULLNAME. В нижней части окна на вкладке **Column Properties** раскройте группу **Computed Column Specification**. В строке **(Formula)** содержится

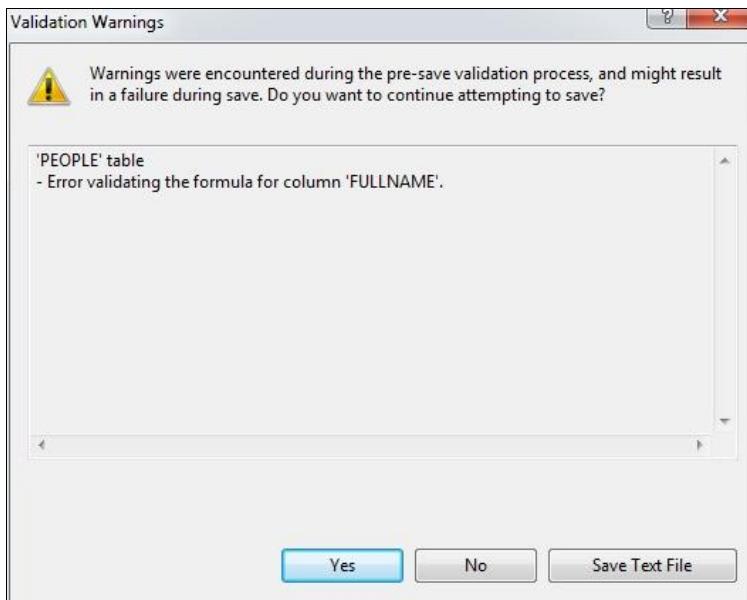


Рис. 5.70. Диалоговое окно, сообщающее о наличии вычисляемого столбца

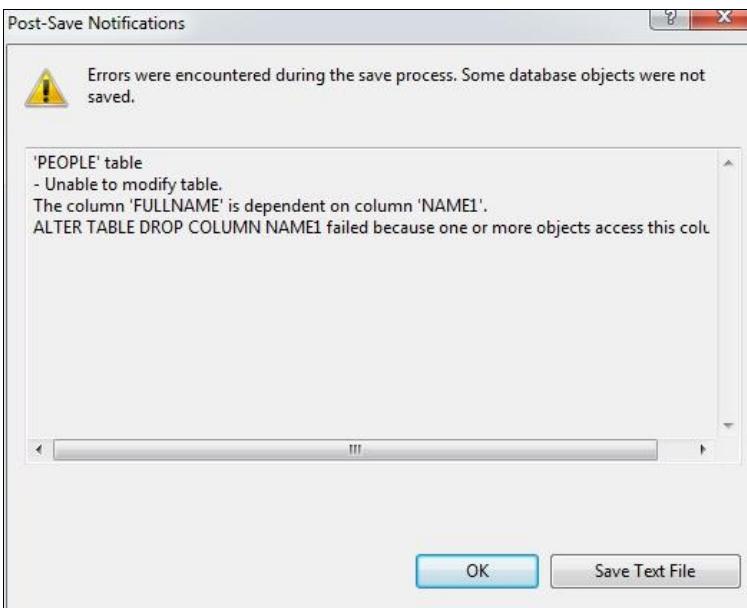


Рис. 5.71. Диалоговое окно, сообщающее о невозможности изменения таблицы

задание формулы получения значения этого вычисляемого столбца. В нашем случае будет записано:

```
((([NAME3] + ' ') + [NAME1]) + ' ') + [NAME2])
```

Нам нужно из этой формулы убрать упоминание столбца NAME1. Аккуратно удалим его из операции конкатенации. Следите за количеством левых и правых круглых скобок в выражении. Теперь формула должна выглядеть следующим образом:

```
(([NAME3] + ' ') + [NAME2])
```

Чтобы формула была изменена, нажмите клавишу <Enter>.

Сохраните изменения в таблице. Вначале вы получите диалоговое окно (рис. 5.72).

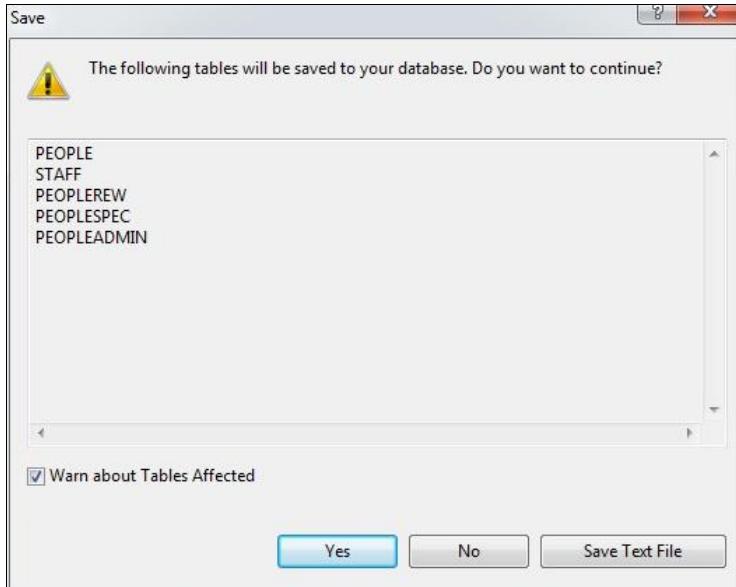


Рис. 5.72. Диалоговое окно, содержащее список изменяемых таблиц

Здесь перечисляются имена таблиц, в которые будут внесены изменения (я опять не понимаю, почему помимо нашей таблицы должны быть изменены другие таблицы). Если вы щелкнете по кнопке **Save Text File** (сохранить текстовый файл) и сохраните файл на диске, то ничего интересного там не увидите. Будут перечислены имена таблиц.

Щелкните по кнопке **Yes**. Изменения таблицы будут сохранены в базе данных.

Теперь пару слов о созданном только что ограничении уникального ключа для таблицы PEOPLE. Если мы удаляем столбец, входящий в состав уникального ключа, то система автоматически без каких-либо предупреждений удаляет уникальный ключ, куда входил удаляемый столбец таблицы, если на этот уникальный ключ не ссылаются внешние ключи других таблиц или той же самой таблицы. Учтите это поведение системы, когда надумаете удалять столбцы ваших таблиц.

5.8.2.9. Удаление ограничений

Удаление ограничения CHECK

Удалить можно любое ограничение CHECK без каких-либо ошибок и неприятностей. От такого ограничения не зависит никакой объект базы данных. Давайте, например, удалим ограничение для столбца SEX таблицы PEOPLE, которое некоторое время назад мы с вами немного покалечили.

Откройте окно **Design** для таблицы PEOPLE. Щелкните правой кнопкой мыши по любому столбцу таблицы и в контекстном меню выберите элемент **Check Constraints**. Появится уже хорошо нам знакомое окно, описывающее все ограничения CHECK этой таблицы (рис. 5.73).

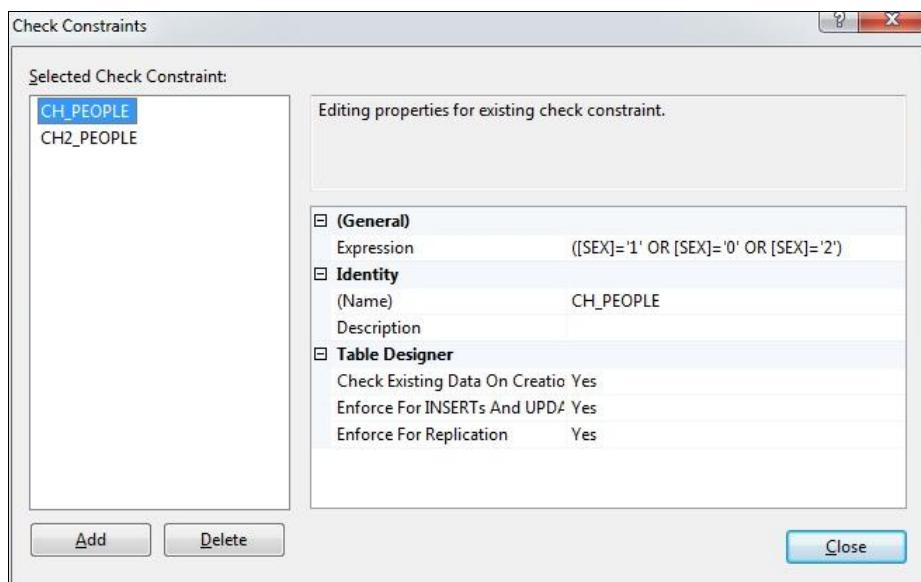


Рис. 5.73. Список ограничений таблицы PEOPLE

Чтобы удалить первое ограничение CH_PEOPLE, нужно, выделив это ограничение, щелкнуть мышью по кнопке **Delete** в левой нижней части окна. Ограничение пропадет из списка. Затем щелкните по кнопке **Close** для закрытия окна. Для завершения процесса нужно сохранить изменения таблицы.

Удаление ограничения первичного ключа

Если на первичный ключ не ссылается никакой внешний ключ другой или той же самой таблицы в базе данных, то его удаление не вызывает никаких проблем. В окне **Design** для соответствующей таблицы щелкните правой кнопкой мыши по столбцу, являющемуся первичным ключом (если первичный ключ составной, то щелкните по любому столбцу, входящему в состав первичного ключа), и в контекстном меню выберите элемент **Remove Primary Key** (удалить первичный ключ). После этого сохраните в базе данных описание таблицы.

Если же вы попытаетесь удалить первичный ключ, на который есть ссылки внешних ключей, то получите сообщение о наличии связей, которые системе будет нужно удалить.

Например, попробуйте удалить первичный ключ таблицы REFREG. Система выдаст сообщение, как показано на рис. 5.74. Здесь спрашивается, хотите ли вы удалить соответствующие отношения.

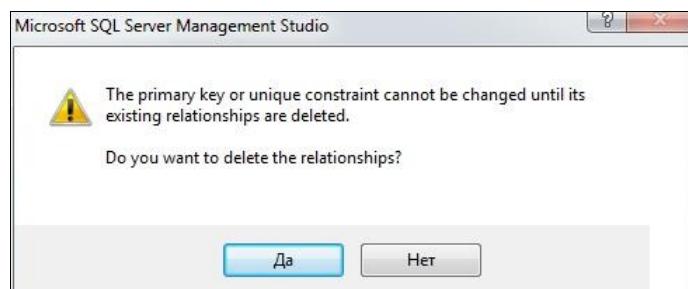


Рис. 5.74. Сообщение о необходимости удаления связей других таблиц

Если в этом окне вы щелкнете мышью по кнопке **Нет**, то удаление первичного ключа не будет выполнено. При щелчке по кнопке **Да** будет удален первичный ключ данной таблицы и во всех подчиненных таблицах будут удалены внешние ключи, ссылающиеся на этот первичный ключ. Однако реально такое удаление будет выполнено только при сохранении изменений таблицы.

Когда вы сохраняете эту измененную таблицу, система выдаст следующий запрос (рис. 5.75).

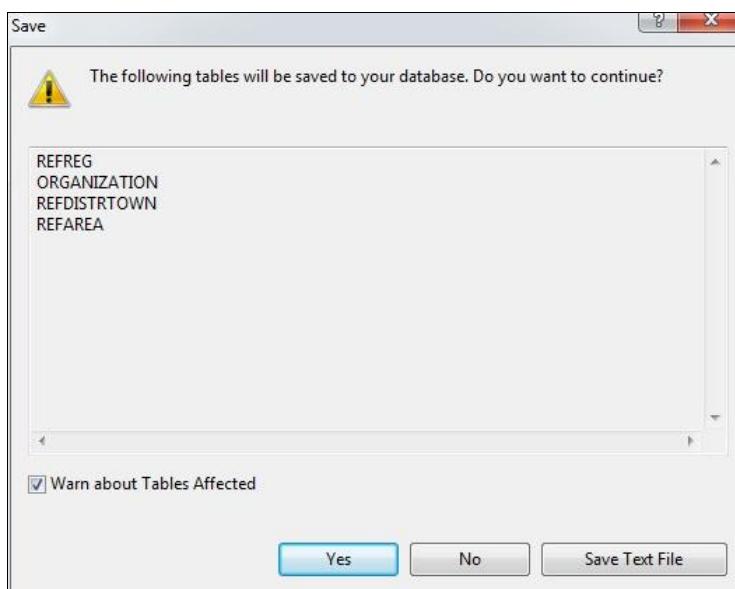


Рис. 5.75. Сообщение об изменении других таблиц

Здесь перечисляются таблицы, которые должны быть изменены (в тексте запроса — "сохранены"). Чтобы выполнить все изменения первичного ключа нашей таблицы и удаление внешних ключей в перечисленных таблицах, нужно щелкнуть мышью по кнопке **Yes**.

Вы можете просмотреть перечисленные в этом сообщении таблицы и убедиться, что система удалила в них все внешние ключи, ссылающиеся на удаленный первичный ключ таблицы REFREG.

Удаление ограничения уникального ключа

Все действия и все сообщения при удалении уникального ключа таблицы полностью соответствуют процессу удаления первичного ключа (с точностью до обозначений).

Удаление ограничения внешнего ключа

Внешний ключ также удаляется без лишних сообщений об ошибках. Давайте удалим внешний ключ таблицы PEOPLEREW (награды людей), ссылающийся на таблицу REFREW (справочник наград).

В окне **Design** для таблицы PEOPLEREW щелкните правой кнопкой мыши по любому столбцу и в контекстном меню выберите элемент **Relationships**. Появится окно, в котором перечисляются все внешние ключи данной таблицы (рис. 5.76).

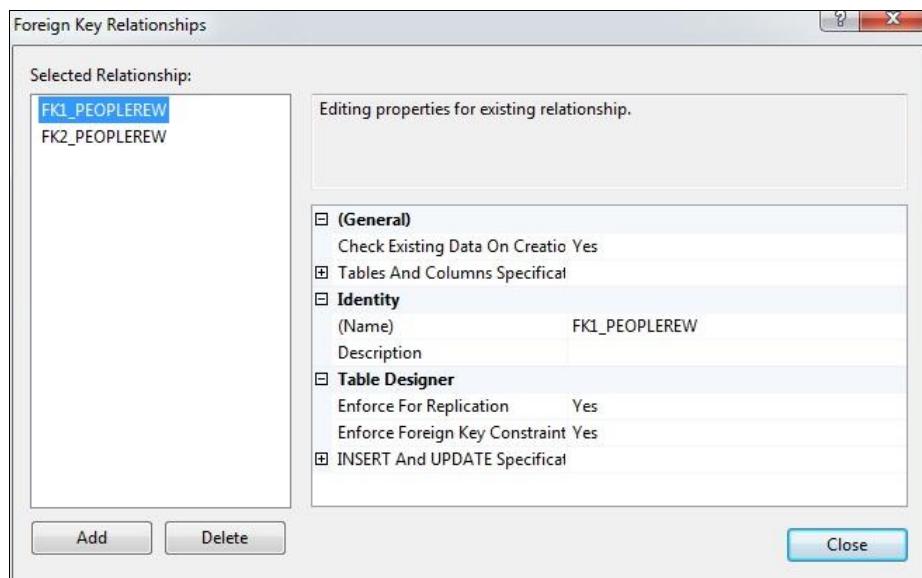


Рис. 5.76. Внешние ключи таблицы PEOPLEREW

Для удаления внешнего ключа в левой части таблицы выберите нужный ключ и щелкните по кнопке **Delete**. Закройте окно, щелкнув мышью по кнопке **Close**. Сохраните таблицу PEOPLEREW. При сохранении таблицы появится предупреждающее

окно (рис. 5.77). Несмотря на это сообщение, никаких изменений в таблице PEOPLE на самом деле выполнять не требуется.

При щелчке по кнопке Yes изменения таблицы будут сохранены в базе данных.



Рис. 5.77. Предупреждающее окно

5.9. Файловые таблицы

Файловые таблицы (FileTable) появились в SQL Server 2012. Они позволяют хранить файлы в виде особых таблиц SQL Server. При этом доступ к данным таких файлов возможен и из приложений, работающих с обычной файловой системой вне контекста какой-либо транзакции. Возможен также и доступ с использованием всех средств языка Transact-SQL.

Файловые таблицы используют файловые потоки, о которых мы уже говорили в этой главе в разд. 5.6.

Все файловые таблицы имеют одинаковую заранее установленную структуру (в документации используется термин *схема*, schema). По этой причине при создании такой таблицы не нужно описывать ее столбцы, все они создаются автоматически. Каждая строка описывает файл на внешнем носителе или каталог.

Таблица содержит столбцы: **Name**, задающий имя файла или каталога, **file_type**, указывающий тип файла, а так же ряд столбцов, описывающих такие характеристики файла, как дата создания, размер, является ли объект файлом или каталогом, можно ли его использовать только для чтения и др. Подробное описание см. в Books Online.

Сейчас мы создадим базу данных и файловые таблицы.

Для создания базы данных, содержащей файловые таблицы, выполните следующий скрипт Transact-SQL. База данных содержит файловую группу **FileTableGR**, предназначенную для хранения файловых потоков. В эту базу данных один мой знакомый помещает хорошую (действительно хорошую) музыку(пример 5.32).

Пример 5.32. Создание базы данных FileTableDB для размещения файловых таблиц

```
USE master;
GO
IF DB_ID('FileTableDB') IS NOT NULL
    DROP DATABASE FileTableDB;
GO
CREATE DATABASE FileTableDB
ON PRIMARY (NAME = FileTableDB_dat,
    FILENAME = 'D:\FileTable\FileTable1.mdf'),
    FILEGROUP FileTableGR CONTAINS FILESTREAM
        (NAME = FileTableGR, FILENAME ='D:\FileTableGR')
LOG ON (NAME = FileTableDB_log,
    FILENAME = 'D:\FileTable\FileTable1.ldf')
    WITH FILESTREAM
        (NON_TRANSACTED_ACCESS = FULL,
            DIRECTORY_NAME = 'FileTableDB');
GO
```

До выполнения этого скрипта на диске D: необходимо создать каталоги с именами **FileTable** и **FileTableDB**.

В предложении **WITH FILESTREAM** указывается имя каталога для хранения данных файлового потока и допустимость использования данных вне контекста транзакции обычными средствами операционной системы (**NON_TRANSACTED_ACCESS = FULL**).

Создадим в этой базе опять же средствами Transact-SQL первую файловую таблицу с именем **MUSIC**. Выполните скрипт из примера 5.33.

Пример 5.33. Создание файловой таблицы

```
USE FileTableDB;
GO
CREATE TABLE MUSIC AS FileTable
    WITH (FileTable_Directory = 'MusicFT');
GO
```

Не пытайтесь обычными средствами операционной системы найти созданные вами каталоги. В операционной системе создаются каталоги и имена файлов, имеющие довольно странные имена. Все каталоги для файловых потоков и файловой таблицы можно просмотреть, вызвав программу Компьютер и выбрав путь: **Сеть | Devrave | mssqlserver | FileTableDB | MusicFT**. Это на моем компьютере. У вас

имена в пути должны отличаться. Сюда же можно переписать некоторое количество нужных файлов. На рис. 5.78 показан список музыкальных произведений, помещенных при помощи копирования в программе Компьютер.

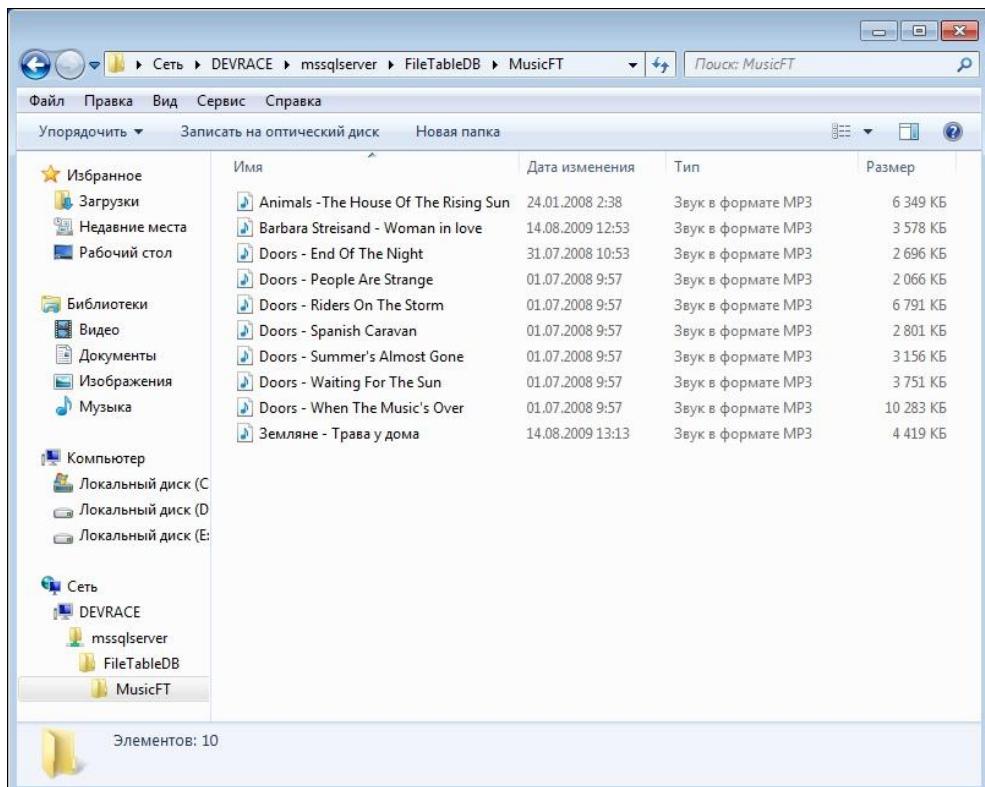


Рис. 5.78. Содержимое таблицы MUSIC

Этот список можно получить и при использовании оператора SELECT, который показан в примере 5.34.

Пример 5.34. Отображение содержимого файловой таблицы

```
USE FileTableDB;
GO
SELECT CAST(name AS CHAR(40)) AS 'Name',
       CAST(creation_time AS CHAR(35)) AS 'Created'
  FROM MUSIC
```

Результат будет следующим:

Name	Created
Animals – The House Of The Rising Sun.mp3	2012-01-28 23:35:31.1480398 +04:00
Barbara Streisand – Woman in love.mp3	2012-01-28 23:35:31.7480742 +04:00

Doors – End Of The Night.mp3	2012-01-28 23:35:32.1340962 +04:00
Doors – People Are Strange.mp3	2012-01-28 23:35:32.4211127 +04:00
Doors – Riders On The Storm.mp3	2012-01-28 23:35:32.6421253 +04:00
Doors – Spanish Caravan.mp3	2012-01-28 23:35:33.2341592 +04:00
Doors – Summer's Almost Gone.mp3	2012-01-28 23:35:33.7221871 +04:00
Doors – Waiting For The Sun.mp3	2012-01-28 23:35:34.0772074 +04:00
Doors – When The Music's Over.mp3	2012-01-28 23:35:34.5792361 +04:00
Земляне – Трава у дома.mp3	2012-01-28 23:35:35.5272903 +04:00

(10 row(s) affected)

А вот диалоговых средств создания файловых таблиц фактически не существует. Если вы в **Object Explorer** раскроете папку **Tables**, щелкните правой кнопкой мыши по папке **FileTables** и в контекстном меню выберете **New FileTable**, то получите лишь заготовку скрипта, который после задания необходимых значений можно использовать в Transact-SQL для создания таблицы.

* * *

В результате всех выполненных в этой главе действий по изменению характеристик существующих таблиц и неудачных попыток таких изменений я окончательно пришел к выводу, что лучше сразу правильно проектировать базу данных, чем потом пытаться вносить в нее изменения. Понятно, лучше быть богатым и здоровым, чем бедным и больным. Однако мой опыт показывает, что быть "богатым и здоровым" в данном случае не так уж и сложно. Если при проектировании базы данных хорошо продумать структуру данных, приводить *все* таблицы к третьей нормальной форме, то в дальнейшем могут потребоваться лишь небольшие изменения, связанные, в основном, лишь с увеличением размера строковых и числовых полей. А такие изменения действительно приходится выполнять довольно часто, поскольку заказчик на этапе проектирования системы клянется, что указанный им размер реквизита максимальный и никогда увеличен не будет. В процессе же эксплуатации системы выясняется несоответствие его заявлений реальной жизни.

Кроме того, при добавлении новой функциональности в созданную и находящуюся в промышленной эксплуатации систему может потребоваться создание новых таблиц. Обычно такие добавления проходят именно в "аддитивном" режиме, т. е. новые данные легко добавляются к уже созданным объектам базы данных. Они не требуют изменения существующих структур данных, если базу данных вы с самого начала проектировали правильно, т. е. так, как я вам показывал.

Что будет дальше?

В следующей главе мы поговорим об индексах в SQL Server. Во многих случаях индексы позволяют повысить производительность системы обработки данных. При неразумном использовании они также позволят резко ухудшить временные характеристики работы с базой данных.



ГЛАВА 6

Индексы

◆ Создание индексов средствами Transact-SQL

- **создание обычных индексов**
- **создание индексов columnstore**
- **создание индексов XML**
- **создание пространственных индексов**

◆ Удаление индексов средствами Transact-SQL

◆ Изменение индексов средствами Transact-SQL

◆ Работа с индексами в Management Studio

Создание индексов при правильном их проектировании может повысить производительность системы обработки данных.

Индексы в SQL Server создаются для таблиц и представлений в виде особых упорядоченных структур на отдельных страницах базы данных. *Индекс* является указателем на соответствующую строку таблицы или представления, в его состав может входить один или более столбцов индексируемой таблицы (представления).

В SQL Server 2012 существуют следующие виды индексов:

- ◆ *обычные индексы*; их еще называют *реляционными*;
- ◆ *индексы columnstore*;
- ◆ *индексы XML*;
- ◆ *пространственные индексы*.

Во многих случаях наличие индексов для таблиц базы данных и представлений может улучшить производительность системы, сильно сократить время, затрачиваемое на выборку и упорядочение данных. Если в системе часто используются запросы, в которых условие выборки данных в точности соответствует структуре одного из созданных индексов (или частично соответствуют старшей структуре индекса), то такие запросы выполняются с высокой скоростью. Индексы, соответ-

ствующие по структуре условию в запросе, называются *покрывающими индексами* (covering indexes).

При выборке данных вначале оптимизатор запросов определяет, можно ли в конкретном случае использовать какой-либо из существующих индексов. Как правило, оптимизатор использует действительно оптимальный алгоритм поиска данных, основываясь на наличии индексов, количестве записей, на статистических данных.

При отсутствии индексов выборка данных из таблицы выполняется сканированием. В этом случае просматриваются физические страницы базы данных, выбираются все строки таблицы и определяется соответствие каждой строки условию выборки.

Для первичного (PRIMARY KEY) и уникального (UNIQUE) ключей таблицы система автоматически создает индексы. Для таких индексов вы также можете задавать некоторые характеристики — параметры индекса.

6.1. Отображение индексов

Существует несколько системных представлений, позволяющих отобразить характеристики индексов: sys.indexes, sys.index_columns, sys.xml_indexes.

Некоторые наиболее интересные столбцы, получаемые из системного представления sys.indexes, которое отображает все индексы базы данных.

- ◆ object_id — идентификатор объекта базы данных, которому принадлежит индекс. Для того чтобы найти имя объекта базы данных по его идентификатору, используется функция OBJECT_NAME(), которой передается в качестве параметра идентификатор. Чтобы найти идентификатор объекта по его имени, используется функция OBJECT_ID(). Для просмотра объектов базы данных можно использовать системное представление sys.objects.
- ◆ name — имя индекса.
- ◆ type, type_desc — тип и название типа индекса: HEAP (куча), CLUSTERED (кластерный), NONCLUSTERED (некластерный), NONCLUSTERED COLUMNSTORE (некластерный columnstore), XML, SPATIAL (пространственный).

Системное представление sys.index_columns содержит сведения для каждого столбца, входящего в состав индексов базы данных. Некоторые его столбцы:

- ◆ object_id — идентификатор объекта базы данных, которому принадлежит индекс описываемого столбца;
- ◆ index_id — идентификатор индекса;
- ◆ index_column_id — идентификатор столбца индекса.

Представление sys.xml_indexes используется для отображения характеристик только индексов XML. Оно содержит следующие столбцы, наиболее интересные нам:

- ◆ name — имя индекса;
- ◆ type_desc — название типа индекса. В данном случае значением будет XML;

- ◆ secondary_type — тип вторичного индекса. Для первичного индекса значением будет NULL, для вторичных индексов XML значением являются V, P, R.
- ◆ secondary_type_desc — название типа вторичного индекса: VALUE, PATH, PROPERTY.

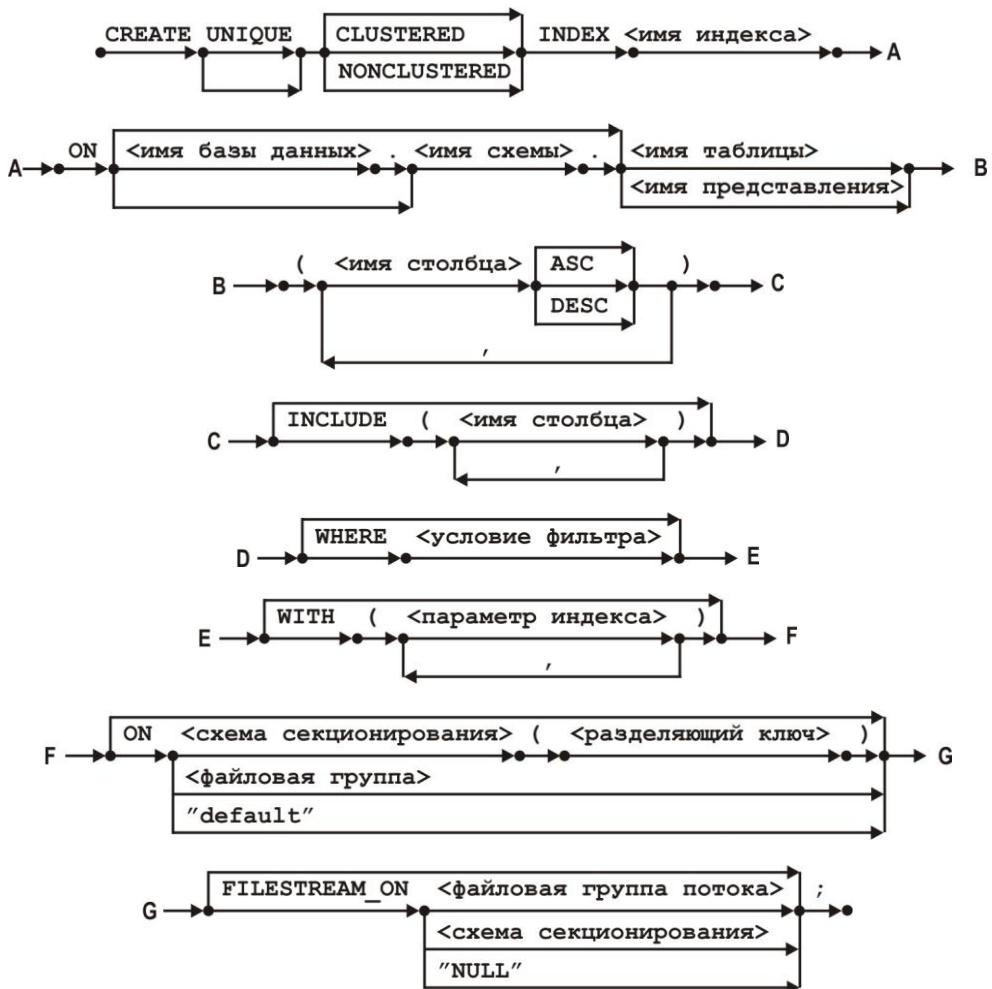
6.2. Работа с индексами средствами Transact-SQL

6.2.1. Создание обычного (реляционного) индекса

Для создания обычного индекса, т. е. индекса для столбцов, имеющих "классические" типы данных, средствами Transact-SQL используется оператор CREATE INDEX. Такие индексы в документации называются *реляционными*. Синтаксис оператора представлен в листинге 6.1 и соответствующих R-графах (графы 6.1—6.2).

Листинг 6.1. Синтаксис оператора CREATE INDEX для создания обычного индекса

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX <имя индекса>
    ON [ [<имя базы данных>.]<имя схемы>.]
        {<имя таблицы> | <имя представления>}
        (<имя столбца> [ ASC | DESC ] [, <имя столбца> [ ASC | DESC ]]...)
        [ INCLUDE (<имя столбца> [, <имя столбца>]...) ]
        [ WHERE <условие фильтра> ]
        [ WITH (<параметр индекса> [, <параметр индекса>]...) ]
        [ ON { <схема секционирования> (<разделяющий ключ>)
            | <файловая группа>
            | "default"
            } ]
        [ FILESTREAM_ON { <файловая группа потока>
            | <схема секционирования>
            | "NULL"
            } ]
    ];
<параметр индекса> ::==
{   PAD_INDEX = { ON | OFF }
| FILLFACTOR = <целое>
| SORT_IN_TEMPDB = { ON | OFF }
| IGNORE_DUP_KEY = { ON | OFF }
| STATISTICS_NORECOMPUTE = { ON | OFF }
| DROP_EXISTING = { ON | OFF }
| ONLINE = { ON | OFF }
| ALLOW_ROW_LOCKS = { ON | OFF }
| ALLOW_PAGE_LOCKS = { ON | OFF }
| MAXDOP = <максимальный уровень параллельности>
| DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS (<номер секции> [, <номер секции>] ...) ]
```

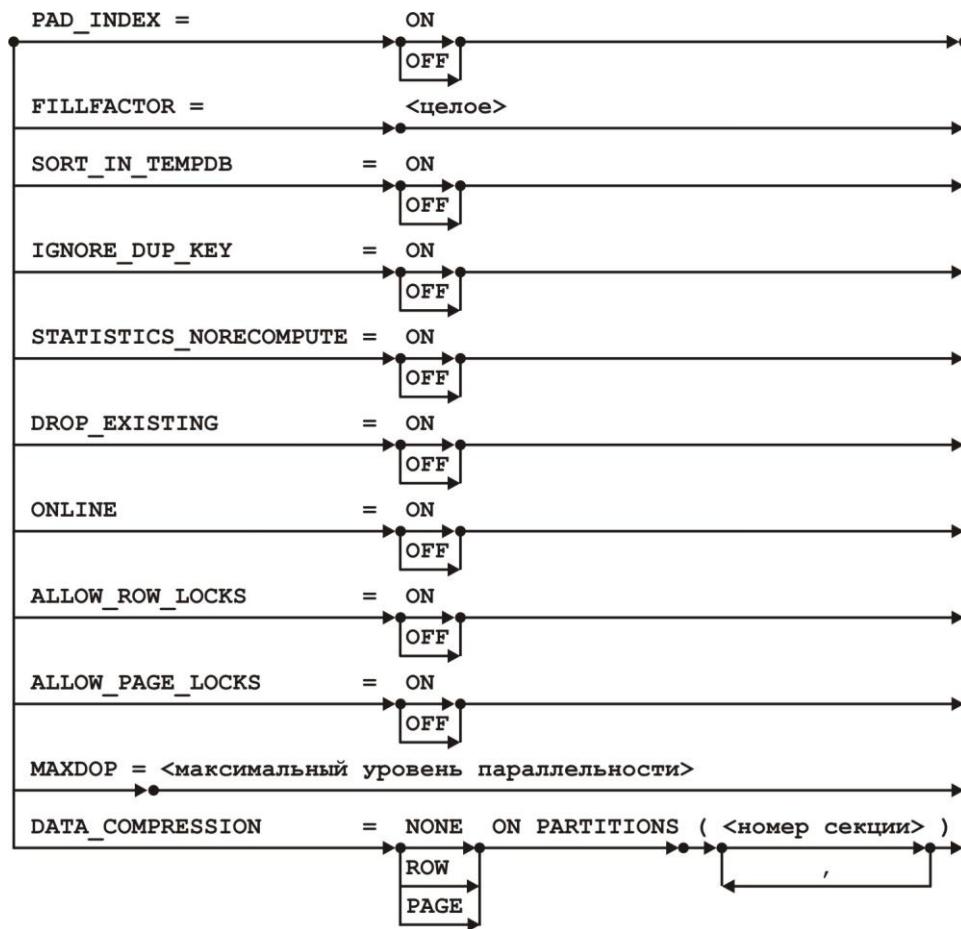


Граф 6.1. Синтаксис оператора CREATE INDEX

ЗАМЕЧАНИЕ ПО СИНТАКСИСУ

Обратите внимание, что в одном операторе присутствует два совершенно разных по смыслу предложения ON. Одно определяет имя объекта, для которого создается индекс, другое используется для задания характеристик секционированности индекса. Вообще говоря, спутать их невозможно, однако совпадение имен многим не нравится. Привередничают они.

В документации Books Online сказано, что индекс можно создавать только для пустой таблицы (представления). То есть, соответствующий объект не должен содержать ни одной строки. Однако это не так. Вы можете создавать индекс и для объекта, когда в нем находятся данные. При этом существует единственное ограничение. Если создается уникальный индекс, то в таблице, в представлении не должно быть строк, нарушающих эту уникальность.



Граф 6.2. Синтаксис параметра индекса

Имя объекта, для которого создается индекс

Индекс создается для одной таблицы или одного представления (такие представления называются *индексированными представлениями*). Объект индексирования задается в предложении **ON**. Перед именем объекта можно указать имя базы данных и имя схемы, отделяя их символом точки. Если в операторе не указано имя базы данных, то индекс создается для объекта, присутствующего в текущей базе данных — в той базе, которая была установлена в последнем операторе **USE**. Если не указано имя схемы, то индекс создается для объекта в схеме по умолчанию — обычно это схема **dbo**.

Имя индекса

Имя индекса должно быть уникальным среди имен индексов только данной таблицы или представления, для которого он создается. Для других объектов в базе данных могут создаваться индексы с теми же именами, хотя такую практику присваивания имен индексам нельзя назвать хорошим решением.

Задание уникального индекса

Если в операторе указано необязательное ключевое слово `UNIQUE`, то создается *уникальный индекс*. Это означает, что в таблице (представлении) не может присутствовать двух строк, у которых будут одинаковые значения всех столбцов, входящих в состав индекса. Если ключевое слово `UNIQUE` не задано, то создается *неуникальный индекс*. В нем допустимо дублирование значений столбцов, составляющих индекс.

Если осуществляется попытка создания уникального индекса для таблицы или представления, где уже существуют строки, в которых присутствует дублирование указанных значений, то такой индекс не будет создан. Вы получите соответствующее диагностическое сообщение.

Кластерный и некластерный индекс

Если указано ключевое слово `CLUSTERED`, то создается кластерный индекс. Если указано `NONCLUSTERED` или не указано ни одно из этих ключевых слов, то создается некластерный индекс по умолчанию.

В случае кластерного индекса строки индексируемой таблицы (представления) располагаются в базе данных как конечные элементы (листья) в иерархическом размещении элементов индекса.

В одном конкретном объекте базы данных может существовать только один кластерный индекс. По умолчанию для первичного ключа таблицы создается именно кластерный индекс. Следовательно, все остальные создаваемые индексы не могут быть кластерными.

Кластерный индекс для представления должен быть объявлен как уникальный.

Если кластерный индекс создается для таблицы, в которой уже существует большое количество строк, то сам процесс создания такого индекса может потребовать немалого времени.

Рекомендуется кластерный индекс создавать прежде, чем будут создаваться другие индексы. В противном случае это потребует немалых затрат на пересоздание всех остальных индексов.

Структура индекса

После имени объекта базы данных, для которого создается индекс, в скобках перечисляются имена столбцов, которые включаются в состав индекса. Индекс может быть простым, т. е. содержать в своем составе один столбец, или составным. Элементы в списке составного индекса отделяются друг от друга запятыми. После имени столбца можно указать способ упорядоченности индекса по этому столбцу. Ключевое слово `ASC` означает упорядоченность по возрастанию значений столбца, `DESC` — по убыванию. Если никакое слово не указано, то предполагается упорядочение по возрастанию. Следите за тем, чтобы один и тот же столбец не был дважды включен в состав индекса.

Составной индекс может содержать не более 16 столбцов. Общий размер значений любого индекса не может превышать 900 байтов.

Столбцы, входящие в состав обычного индекса, не могут иметь тип данных TEXT, NTEXT, VARCHAR (MAX), NVARCHAR (MAX), VARBINARY (MAX), XML, IMAGE. Для столбцов таблицы (только таблицы, но не представлений) типа данных XML можно создавать отдельные индексы XML, о чём мы поговорим чуть позже в этой главе.

Индексы можно создавать и для вычисляемых столбцов, как для столбцов, значения которых хранятся в базе данных (столбцы, описанные с атрибутом PERSISTED), так и для столбцов, значения которых вычисляются при обращении к строке таблицы. Существуют некоторые ограничения на характеристики вычисляемых столбцов. Чтобы по вычисляемому столбцу с числовым типом данных можно было создавать индекс, тип данных не должен быть приближительным числом (числом с плавающей точкой). Кроме того, значение вычисляемого столбца должно быть детерминированным.

Предложение *INCLUDE*

В необязательном предложении INCLUDE можно перечислить имена неключевых столбцов (только столбцов, не входящих в состав никаких ключей или индексов), значения которых будут размещаться на уровне листьев (элементов самого нижнего уровня) создаваемого некластерного индекса. Здесь мы получаем как бы суррогат кластерного индекса. Только на нижнем уровне индексной иерархии размещаются не все данные одной строки таблицы, а только отдельные ее столбцы, указанные в этом предложении.

Предложение *WHERE*

В операторе создания индекса можно указать предложение WHERE, которое определяет создание "фильтрованного" индекса — индекса, в состав которого включаются не все строки таблицы, а только те, которые отвечают условиям, заданным в этом предложении. Такой фильтрованный индекс не может быть кластерным.

Условие в предложении WHERE может быть довольно сложным. Допустимо использование операций сравнения, оператора IN, определяющего вхождение значения в указанный список, логических операций конъюнкции, дизъюнкции и отрицания. В условии нельзя использовать в качестве любого литерала неизвестное значение NULL. Для проверки на такое значение следует использовать конструкции IS NULL (является неизвестным значением) или IS NOT NULL (не является таким значением).

Задание условия фильтрации для нашей таблицы PEOPLE может быть выполнено, например, в таком виде:

```
WHERE BIRTHDAY > '01.01.1900'
```

Здесь будут индексироваться только те строки людей, дата рождения которых больше 1 января 1900 года. Это позволит сэкономить внешнюю память и несколько повысить скорость выборки данных по людям с соответствующими датами рождения.

Предложение **ON**

В необязательном предложении **ON** (во втором предложении **ON**) (граф 6.3) можно указать, где должны располагаться данные индекса.

```
[ ON { <схема секционирования> (<разделяющий ключ>)
      | <файловая группа>
      | "default"
      }
]
```



Граф 6.3. Синтаксис предложения **ON**

Существует три варианта размещения в базе данных индекса, как и в случае создания таблиц.

При указании **"default"** система разместит данные индекса в файловой группе по умолчанию.

Если задано имя файловой группы, то индекс будет размещаться в файлах этой файловой группы.

Можно создать секционированный индекс, задав имя схемы секционирования и в скобках разделяющий ключ. Все действия по созданию секционированного индекса очень похожи на те действия, которые нужно выполнить для создания секционированной таблицы. Интересный момент здесь в том, что разделяющий ключ не обязательно должен входить в состав данного индекса. Это может быть любой столбец таблицы. Работа с секционированными таблицами подробно описана в *разд. 5.3*.

Предложение **FILESTREAM_ON**

Это предложение используется при создании кластерного индекса. Позволяет задать размещение данных для столбцов таблицы **FILESTREAM**.

Здесь можно указать имя файловой группы, где будут размещаться данные файлового потока, имя схемы секционирования, в которой определяется размещение данных **FILESTREAM**, или **"NULL"**, если таблица не содержит столбцов **FILESTREAM**.

Задание параметров индекса

В предложении **WITH** можно перечислить параметры (опции) создаваемого индекса. Эти параметры также используются и при описании характеристик индекса, автоматически создаваемого системой для первичного или уникального ключа таблицы (см. *главу 5*).

```
PAD_INDEX = { ON | OFF }
```

Задает возможность использования разреженного индекса. **ON** — допустим разреженный индекс, **OFF** (значение по умолчанию) — не допустим. В случае разрежен-

ного индекса обязательно должен присутствовать и параметр **FILLFACTOR**, задающий процент разрежения.

FILLFACTOR = <целое>

Здесь можно указать целое число в диапазоне от 1 до 100. Это число определяет процент заполнения страницы индекса самого нижнего уровня. Применяется при первоначальном создании или при пересоздании индекса.

Если не предполагается, что в таблицу будет добавляться большое количество новых записей, для которых понадобится формировать элементы индекса, то можно смело указывать число 100 (или 0, это одно и то же). Здесь каждая страница нижнего уровня будет заполняться данными под завязку, что позволит сэкономить место на внешнем носителе и повысить производительность системы. В случае же, когда могут происходить добавления данных, влияющие на индекс, имеет смысл уменьшить процент заполнения страницы.

Следует уточнить, что значение этого параметра влияет только на процесс *первоначального создания* или *пересоздания* индекса. В процессе работы с таблицей, процент заполнения отдельных страниц индекса может измениться. В этом случае для восстановления первоначального значения процента заполнения страницы индекса имеет смысл внести изменение в индекс при помощи оператора **ALTER INDEX**.

SORT_IN_TEMPDB = { ON | OFF }

Задает необходимость сохранения промежуточных результатов сортировки данных во временной базе данных **tempdb**. **ON** — данные сохраняются в базе данных **tempdb**. Иногда это может сократить время работы системы, если временная база данных **tempdb** находится на физическом носителе, отличном от того, на котором располагается база данных (файлы базы данных), с которой выполняется работа. Значение **OFF** (значение по умолчанию) — промежуточные данные хранятся в той же самой базе данных, что и индекс.

IGNORE_DUP_KEY = { ON | OFF }

Имеет смысл только применительно к уникальному (**UNIQUE**) индексу. Определяет поведение системы при попытке добавления строки в таблицу, содержащую дублирующее значение, которое должно помещаться в уникальный индекс.

Если указано **ON**, то в случае дублирования значения выдается предупреждающее сообщение, отменяются лишь те операторы **INSERT**, которые пытаются записать дубликат значений столбцов, входящих в состав уникального индекса. Остальные действия в рамках данной транзакции будут выполнены.

При задании **OFF** (значение по умолчанию) если в добавляемых данных содержится дублирующее значение, то отменяется выполнение всего добавления данных, осуществляемого в контексте текущей транзакции.

STATISTICS_NORECOMPUTE = { ON | OFF }

Задает автоматическое вычисление статистики.

Значение `ON` означает, что статистические данные по индексу не будут вычисляться автоматически. Отмена такого вычисления может ухудшить временные характеристики выполнения поиска данных, т. к. оптимизатор запросов не будет располагать достоверными сведениями, позволяющими ему сформировать наиболее эффективный запрос к базе данных.

Если задано `OFF` (значение по умолчанию), то будет автоматически выполняться пересоздание статистических данных по индексу.

`DROP_EXISTING = { ON | OFF }`

Влияет на поведение системы, если в базе данных уже существует для указанного объекта индекс с тем же именем. Задает, следует ли системе удалить и перестроить существующий индекс. `ON` — существующий индекс удаляется и перстраивается, `OFF` (значение по умолчанию) — если индекс с тем же именем существует, то выдается ошибка.

`ONLINE = { ON | OFF }`

Определяет возможность использования таблицы или представления, для которого создается индекс, другими процессами во время создания индекса. Если задано `ON`, то блокировка соответствующего объекта не осуществляется, другие процессы могут выполнять любые действия с этим объектом. `OFF` (значение по умолчанию) вызывает блокировку объекта. Другие процессы не могут выполнять с ним никаких действий.

Надо сказать, что в любом случае создание нового индекса лучше выполнять в то время, когда с соответствующим объектом не проводятся никакие другие действия иными процессами.

`ALLOW_ROW_LOCKS = { ON | OFF }`

Задает возможность блокировки строк при обращении к индексу. Значение `ON` (по умолчанию) разрешает блокировку строк. `OFF` — блокировки не используются.

`ALLOW_PAGE_LOCKS = { ON | OFF }`

Похожим образом, как и в предыдущем параметре, задает возможность блокировки страниц при обращении к индексу. Значение `ON` (значение по умолчанию) разрешает блокировку страниц. `OFF` — блокировки не используются.

`MAXDOP = <максимальный уровень параллельности>`

Позволяет установить максимальное количество используемых процессоров при выполнении параллельных операций с индексами. Параметр может иметь значение от 0 до 64. Применяется только для версий SQL Server Enterprise и Developer.

Если задано 1, то допустимо использование только одного процессора. Значение, превышающее 1, задает максимальное количество процессоров, используемых в параллельных операциях. В реальности может быть использовано меньше процессоров, чем указано в параметре. Значение 0 (значение по умолчанию) задает использование всех существующих процессоров или, в зависимости от текущей нагрузки, меньшее их количество.

```
DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS <номер секции> [, <номер секции>] ... ]
```

Позволяет задать режим сжатия данных для индекса.

Если указано NONE, то никакого сжатия данных выполнятся не будет.

Задание ROW означает сжатие на уровне строк.

При задании PAGE выполняется сжатие страниц.

Предложение ON PARTITIONS может задаваться только для секционированного индекса. В скобках перечисляются, отделяясь друг от друга запятыми, номера секций, к которым относится указанный режим сжатия (NONE, ROW, PAGE). Для таких индексов параметр DATA_COMPRESSION может повторяться произвольное количество раз. Хотя понятно, что реально может потребоваться не более трех повторений — по одному на каждый режим сжатия.

В следующем примере (пример 6.1) создается некластерный индекс для столбца краткого названия страны NAME в справочнике стран REFCTR. Индекс упорядочивается по возрастанию значений столбца. Перед созданием индекса проверяется, существует ли такой индекс для указанной таблицы. Для этого используется представление просмотра каталогов sys.indexes. Если индекс существует, то он удаляется оператором DROP INDEX, который мы рассмотрим чуть позже.

Пример 6.1. Создание простого индекса

```
USE BestDatabase;
IF EXISTS(SELECT * FROM sys.indexes
          WHERE NAME = 'CTRNAME')
    DROP INDEX CTRNAME
        ON REFCTR;
GO
CREATE INDEX CTRNAME ON REFCTR
    (NAME ASC);
GO
```

Отобразите список всех индексов базы данных BestDatabase, выполнив скрипт примера 6.2.

Пример 6.2. Отображение списка индексов базы данных BestDatabase

```
USE BestDatabase;
SELECT CAST(OBJECT_NAME(object_id) AS CHAR(12)) AS "Object",
       CAST(name AS CHAR(36)) AS "Name",
       CAST(type_desc AS CHAR(14)) AS "Type",
       CAST(is_unique AS CHAR(1)) AS "Unique"
FROM sys.indexes;
GO
```

Список будет содержать более 100 строк. Вот маленький фрагмент этого списка:

Object	Name	Type	Unique
...			
REFCTR	PK_REFCTR	CLUSTERED	1
REFCTR	CTRNAME	NONCLUSTERED	0
REFREG	PK_REFREG	CLUSTERED	1
...			

Чтобы получить имя объекта, для которого создан индекс, а не его идентификатор, здесь использована функция `OBJECT_NAME()`.

6.2.2. Создание индекса для представлений

Представления являются виртуальными таблицами. Представление содержит оператор `SELECT`, который обращается к одной или к нескольким таблицам. Результатом выборки данных является набор данных. Однако результаты представлений не хранятся в базе данных. Они каждый раз получаются вновь при обращении к представлению. К представлению можно обращаться как к обычной таблице.

Результат выполнения представления можно сделать хранимым в базе данных, создав для представления уникальный кластерный индекс. В этом случае набор данных, получаемый при вызове представления, хранится в базе данных, как и в случае обычного кластерного индекса. При любых изменениях в базовых таблицах представления (в таблицах, к которым обращается представление) эти изменения отражаются в данных, создаваемых при помощи этого представления.

Создание для представления такого индекса во многих случаях позволяет повысить производительность системы.

Представления мы будем более подробно рассматривать в главе 9.

6.2.3. Создание индекса columnstore

Индексы columnstore появились в версии SQL Server 2012. Основное его отличие от обычного, "реляционного", классического (rowstore) заключается в способе формирования и форме хранения.

При создании обычного индекса группируются и сохраняются данные для строк. Для индекса columnstore выполняется группировка и сохранение данных для столбцов. При этом выполняется сжатие данных. Для некоторых типов запросов к базе такая структура индекса может сильно повысить производительность. Подобные запросы часто используются при работе с так называемыми хранилищами данных.

В индекс могут быть включены типы данных: строковые, числовые, даты и времени. Нельзя включать разреженные столбцы. Индекс не может содержать более 1024 столбцов. Кроме того, индекс может создаваться только для таблиц, но не для представлений.

Данные таблицы, для которой создан индекс columnstore, не могут быть изменены. Синтаксис оператора CREATE COLUMNSTORE INDEX представлен в листинге 6.2 и в соответствующем R-графе (граф 6.4).

Листинг 6.2. Синтаксис оператора CREATE COLUMNSTORE INDEX

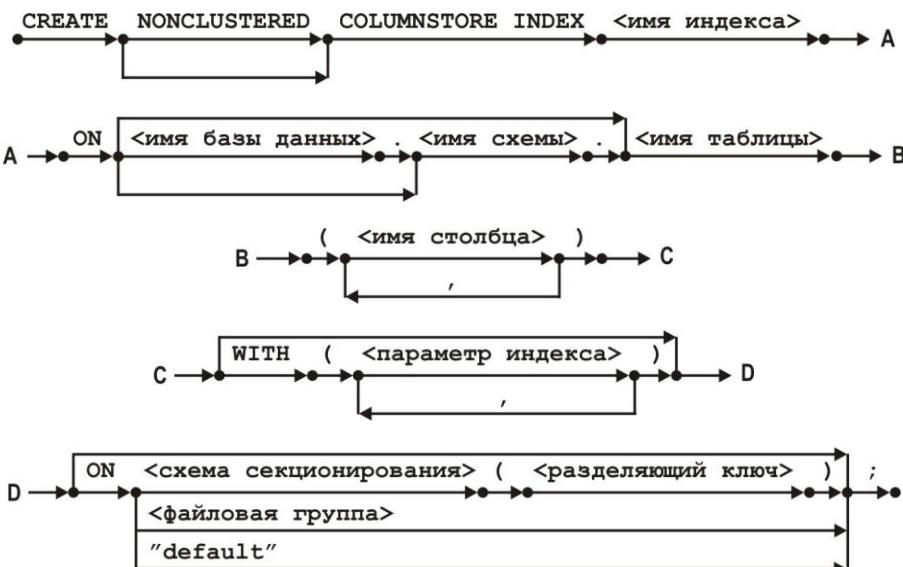
```

CREATE [ NONCLUSTERED ] COLUMNSTORE INDEX <имя индекса>
    ON [ [<имя базы данных>.]<имя схемы>.]<имя таблицы>
        (<имя столбца> [, <имя столбца>]...)
    [ WITH (<параметр индекса COLUMNSTORE>
            [, <параметр индекса COLUMNSTORE>]...) ]
    [ ON { <схема секционирования> (<разделяющий ключ>)
          | <файловая группа>
          | "default"
      } ];
    
```

<параметр индекса COLUMNSTORE> ::=

```

{   DROP_EXISTING = { ON | OFF }
    | MAXDOP = <максимальный уровень параллельности>
}
    
```



Граф 6.4. Синтаксис оператора CREATE COLUMNSTORE INDEX



Граф 6.5. Синтаксис параметра индекса columnstore

Надо полагать, нам в этом операторе все понятно. Мы уже рассматривали такие значения и параметры в реляционном индексе.

Объект индексирования задается в предложении `ON`. Имя индекса должно быть уникальным среди имен всех индексов данной таблицы. Список имен столбцов, входящих в состав индекса, перечисляется в скобках. Здесь допустимо использовать два параметра индекса. Параметр `DROP_EXISTING` задает удаление существующего индекса без выдачи сообщения об ошибке. `MAXDOP` задает максимальное количество процессоров при выполнении операций с индексом. В предложении `ON` задаются характеристики секционированности индекса.

6.2.4. Создание индекса для столбца XML

Столбцы с типом данных `XML` могут присутствовать в составе и обычного индекса. Для столбцов с этим типом данных можно создать специальные индексы XML. В составе такого индекса может присутствовать только один столбец. Для одной таблицы можно создать до 249 индексов XML. Для одного столбца XML можно создать один первичный (`PRIMARY`) и до трех вторичных индексов. Столбец, для которого создается индекс, не должен быть вычисляемым.

Существование индексов XML может сильно повысить производительность системы, если в запросах на выборку данных часто задаются условия поиска, связанные со столбцом XML. Данные в таком столбце могут занимать до 2 Гбайт памяти. При наличии индекса в процессе поиска нет необходимости выполнять синтаксический анализ большого объема данных каждого столбца XML. Такой анализ выполняется один раз при создании индекса и каждый раз при модификации данных XML.

Наличие индексов XML может снизить производительность системы, если выполняется частая модификация соответствующих данных.

При индексировании столбца XML выполняется индексирование всех тегов, путей и значений, хранимых в этом столбце.

Синтаксис оператора `CREATE XML INDEX` для создания индекса XML представлен в листинге 6.3 и в соответствующем ему R-графе (графы 6.6 и 6.7).

Листинг 6.3. Синтаксис оператора `CREATE XML INDEX` для создания индекса XML

```

CREATE [ PRIMARY ] XML INDEX <имя индекса>
  ON [[<имя базы данных>.]<имя схемы>.]<имя таблицы>
    (<имя столбца XML>
      [ USING XML INDEX <имя индекса XML>
        FOR { VALUE | PATH | PROPERTY } ]
      [ WITH (<параметр индекса XML> [, <параметр индекса XML>]...) ];
```

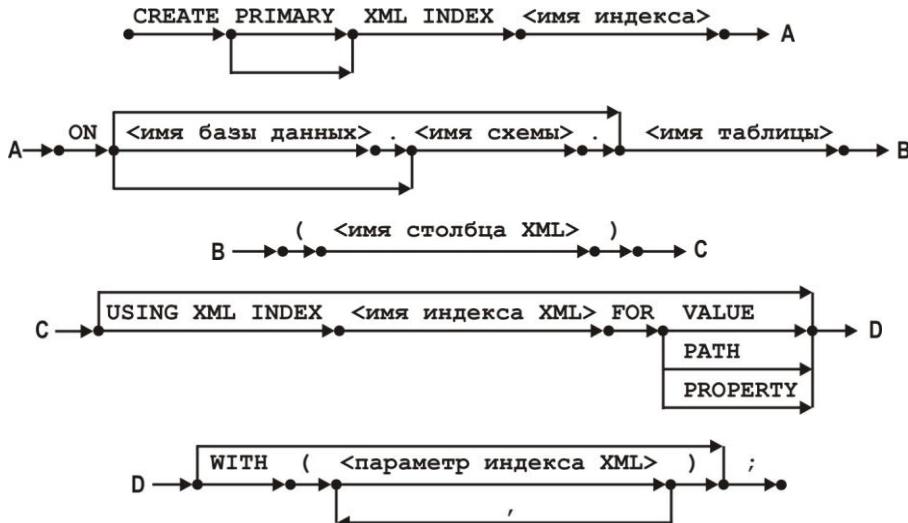
<параметр индекса XML> ::=

{ PAD_INDEX = { ON | OFF }
 | FILLFACTOR = <целое>
 | SORT_IN_TEMPDB = { ON | OFF }
 | IGNORE_DUP_KEY = OFF

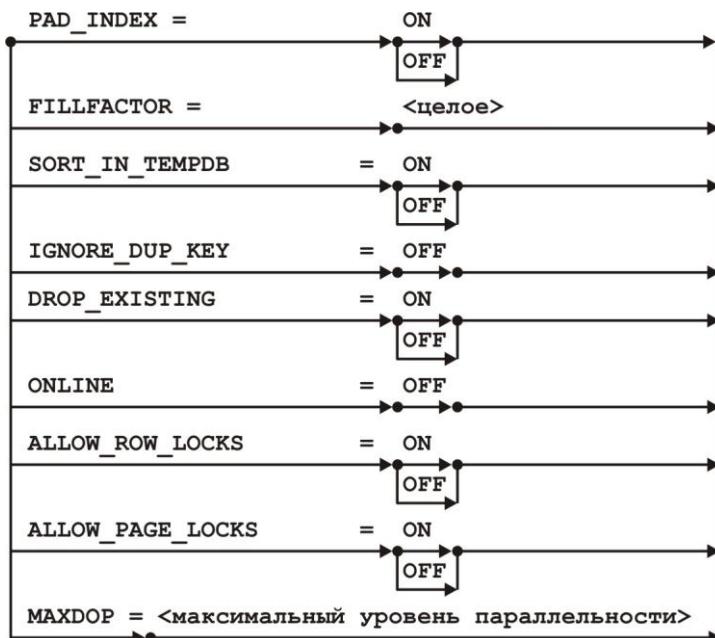
```

| DROP_EXISTING = { ON | OFF }
| ONLINE = OFF
| ALLOW_ROW_LOCKS = { ON | OFF }
| ALLOW_PAGE_LOCKS = { ON | OFF }
| MAXDOP = <максимальный уровень параллельности>
}

```



Граф 6.6. Синтаксис оператора CREATE XML INDEX



Граф 6.7. Синтаксис параметра индекса XML

Имя объекта, для которого создается индекс

Индекс XML создается только для одного столбца, имеющего тип данных XML. Индекс не может создаваться для группы столбцов XML.

Имя индекса

Имя индекса должно быть уникальным среди имен всех индексов данной таблицы.

Первичный индекс

Если указано ключевое слово PRIMARY, то создается первичный или так называемый *кластерный индекс XML*. В этом случае таблица должна иметь кластерный первичный ключ. Количество столбцов, входящих в состав такого первичного ключа, не должно превышать 15. Кластерный индекс состоит из первичного ключа таблицы и идентификатора узла XML. Кластерный индекс XML в столбце таблицы с типом данных XML может быть только один. Кроме этого для того же столбца может быть несколько вторичных индексов XML.

В первичном индексе индексируются все теги, пути и значения в столбце. При его создании выполняется синтаксический анализ текста XML (или, как сказано в документации, разборка, разбивка — shredding).

Создание вторичных индексов XML возможно только после создания первичного индекса XML.

После имени таблицы, для которой создается индекс XML (предложение ON, в скобках задается имя столбца XML).

Вторичные индексы

Предложение USING XML INDEX задает создание вторичного индекса указанного типа. В этом предложении указывается имя первичного индекса, на основании которого создается вторичный индекс.

Можно создавать три типа вторичных индексов для столбца XML. Тип вторичного индекса задается после *обязательного* (в Books Online синтаксис задан несколько некорректно) ключевого слова FOR:

- ◆ PATH;
- ◆ VALUE;
- ◆ PROPERTY.

Вторичный индекс PATH создается для выражений пути.

Вторичный индекс VALUE создается для значений. Значение может находиться на любом уровне иерархии в тегах документа XML.

Вторичный индекс PROPERTY содержит значения атрибутов.

Вторичные индексы могут повысить производительность системы при выполнении различных типов запросов.

Задание параметров индекса

В предложении `WITH` можно перечислить параметры создаваемого индекса. Эти параметры похожи на параметры обычного индекса. Перечислим их.

◆ `PAD_INDEX = { ON | OFF }`

Задает разреженность индекса. `ON` — допустим разреженный индекс, `OFF` (значение по умолчанию) — не допустим.

В случае разреженного индекса должен присутствовать и параметр `FILLFACTOR`.

◆ `FILLFACTOR = <целое>`

Здесь можно указать целое число в диапазоне от 1 до 100, которое определяет процент заполнения страницы индекса самого нижнего уровня. Применяется при первоначальном создании или при пересоздании индекса.

◆ `SORT_IN_TEMPDB = { ON | OFF }`

Задает сохранение промежуточных результатов сортировки данных во временной базе данных `tempdb`. `ON` — данные сохраняются в базе данных `tempdb`. `OFF` (по умолчанию) — промежуточные данные хранятся в той же базе данных, что и индекс.

◆ `IGNORE_DUP_KEY = OFF`

При задании `OFF` если в добавляемых данных содержится дублирующее значение, то отменяется выполнение всего добавления данных, осуществляемого в контексте текущей транзакции. Другое значение у этого параметра отсутствует. Вообще-то для XML-индекса этот параметр не нужен, поскольку такие индексы не являются уникальными.

◆ `STATISTICS_NORECOMPUTE = { ON | OFF }`

Задает автоматическое вычисление статистики.

Значение `ON` означает, что статистические данные по индексу не будут вычисляться автоматически. Если задано `OFF` (по умолчанию), будет автоматически выполняться пересоздание статистических данных по индексу.

◆ `DROP_EXISTING = { ON | OFF }`

Влияет на поведение системы, если в базе данных уже существует для объекта индекс с тем же именем. `ON` — существующий индекс удаляется и перестраивается, `OFF` (по умолчанию) — если индекс с тем же именем существует, то выдается ошибка.

◆ `ONLINE = OFF`

Определяет возможность использования таблицы, для которой создается индекс, другими процессами во время создания индекса. `OFF` (значение по умолчанию, единственное значение) вызывает блокировку объекта. Другие процессы не могут выполнять с ним никаких действий.

◆ `ALLOW_ROW_LOCKS = { ON | OFF }`

Задает возможность блокировки строк при обращении к индексу. Значение `ON` (по умолчанию) разрешает блокировку. `OFF` — блокировки не используются.

◆ ALLOW_PAGE_LOCKS = { ON | OFF }

Задает возможность блокировки страниц при обращении к индексу. ON (значение по умолчанию) разрешает блокировку. OFF — блокировки не используются.

◆ MAXDOP = <максимальный уровень параллельности>

Устанавливает максимальное количество используемых процессоров при выполнении параллельных операций с индексами. Может иметь значение от 0 до 64.

Если задано 1, то допустимо использование только одного процессора. Значение, превышающее 1, задает максимальное количество процессоров, используемых в параллельных операциях.

Создание индексов XML показано в примере 6.3. В главе 4 при рассмотрении типа данных XML мы с вами создали коллекцию схем XML, необходимую для включения в таблицу стран столбца XML. В следующем примере мы повторим создание коллекции схем XML и таблицы REFCTRXML, а также создадим четыре индекса XML для столбца таблицы REGIONDESCR — первичный и три вторичных.

Пример 6.3. Создание индексов XML

```
USE BestDatabase;
GO
IF EXISTS(SELECT * FROM sys.tables
           WHERE name = 'REFCTRXML')
    DROP TABLE REFCTRXML;
GO
IF EXISTS(SELECT * FROM sys.xml_schema_collections
           WHERE name = 'TestSchemaCtr')
    DROP XML SCHEMA COLLECTION TestSchemaCtr;
CREATE XML SCHEMA COLLECTION TestSchemaCtr AS
N'<?xml version="1.0" encoding="UTF-16"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="Country">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="IDCTR" type="xsd:string" />
        <xsd:element name="NameCTR" type="xsd:string" />
        <xsd:element name="Regions" type="Region" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
<xsd:complexType name="Region">
  <xsd:choice minOccurs="0" maxOccurs="unbounded" >
    <xsd:sequence>
      <xsd:element name="ID" type="xsd:string" />
```

```
<xsd:element name="Name" type="xsd:string" />
<xsd:element name="Center" type="xsd:string" />
</xsd:sequence>
</xsd:choice>
</xsd:complexType>
</xsd:schema>' ;
GO
CREATE TABLE REFCTRXML
( CODCTR      CHAR(3) NOT NULL,    /* Код страны */
  REGIONDESCR XML(TestSchemaCtr), /* Описание регионов */
  CONSTRAINT PK_REFCTRXML PRIMARY KEY (CODCTR)
);
GO
CREATE PRIMARY XML INDEX DESCRPRIMARY
  ON REFCTRXML(REGIONDESCR);
GO
CREATE XML INDEX DESCRIPTORSECONDARY1
  ON REFCTRXML(REGIONDESCR)
  USING XML INDEX DESCRPRIMARY
  FOR VALUE;
CREATE XML INDEX DESCRIPTORSECONDARY2
  ON REFCTRXML(REGIONDESCR)
  USING XML INDEX DESCRPRIMARY
  FOR PATH;
CREATE XML INDEX DESCRIPTORSECONDARY3
  ON REFCTRXML(REGIONDESCR)
  USING XML INDEX DESCRPRIMARY
  FOR PROPERTY;
GO
```

Вначале проверяется существование таблицы `REFCTRXML`, и при ее наличии она удаляется из базы данных. При этом автоматически удаляются и все ее индексы. Если бы было нужно удалить только индекс таблицы, то следовало бы выполнить оператор:

```
IF EXISTS(SELECT * FROM sys.indexes
          WHERE NAME = 'DESCRPRIMARY')
  DROP INDEX DESCRPRIMARY
    ON REFCTRXML;
GO
```

Если в операторе удаляется первичный индекс XML, то автоматически удаляются и все вторичные.

Далее для таблицы `REFCTRXML` для ее столбца `REGIONDESCR`, имеющего тип данных `XML`, создаются четыре индекса XML: один первичный и три вторичных.

Для отображения индексов XML базы данных используется системное представление sys.xml_indexes. Отобразите список индексов, как показано в примере 6.4.

Пример 6.4. Отображение индексов XML

```
USE BestDatabase;
GO
SELECT CAST(name AS CHAR(16)) AS "Name",
       CAST(type_desc AS CHAR(4)) AS "Type",
       CAST(secondary_type AS CHAR(14)) AS "Secondary Type",
       CAST(secondary_type_desc AS CHAR(11)) AS "Description"
FROM sys.xml_indexes;
GO
```

Вы получите следующий список:

Name	Type	Secondary Type	Description
DESCRPRIMARY	XML	NULL	NULL
DESCRSECONDARY1	XML	V	VALUE
DESCRSECONDARY2	XML	P	PATH
DESCRSECONDARY3	XML	R	PROPERTY

(4 row(s) affected)

Для отображения характеристик индексов XML существуют и другие системные средства. Если они понадобятся вам в вашей деятельности, обратитесь к Books Online.

Тип данных XML и все средства работы с такими данными являются довольно сложными. Им посвящаются отдельные книги. Если в вашей работе нужны такие данные и средства, рекомендую обратиться к специальной литературе, которой, правда, не так уж и много.

6.2.5. Создание пространственного индекса

Для столбцов таблиц типа данных GEOMETRY и GEOGRAPHY можно создавать пространственные индексы. Напомню, что тип данных GEOMETRY описывает элементы на плоской поверхности, в евклидовой геометрии, а тип данных GEOGRAPHY позволяет представлять объекты на поверхности Земли с учетом ее формы и размеров. Такие индексы мы рассмотрим довольно поверхностно, потому что это тема опять же для отдельной книги.

В состав пространственного индекса может включаться только один столбец.

Синтаксис оператора CREATE SPATIAL INDEX для создания пространственного индекса представлен в листинге 6.4 и в соответствующих R-графах (графы 6.8—6.17).

Листинг 6.4. Синтаксис оператора CREATE SPATIAL INDEX

```
CREATE SPATIAL INDEX <имя индекса>
    ON [ [<имя базы данных>.]<имя схемы>.]<имя таблицы>
        (<имя пространственного столбца>)
        { <геометрическая тесселяция> | <географическая тесселяция> }
    [ ON { <файловая группа> | "default" } ] ;

<геометрическая тесселяция> ::==
{ [ USING GEOMETRY_GRID ]
    WITH (<границы> [, <параметр тесселяции> ]... [, <опции> ]...)
| USING GEOMETRY_AUTO_GRID
    WITH (<границы> [, <параметр тесселяции> ]... [, <опции> ]...)
}

<географическая тесселяция> ::==
{ [ USING GEOGRAPHY_GRID
    WITH ([ <параметр тесселяции> ] [, <параметр тесселяции> ]...
          [, <опции> ]...)
| USING GEOGRAPHY_AUTO_GRID
    WITH ([ <параметр тесселяции> ] [, <параметр тесселяции> ]...
          [, <опции> ]...)
}

<границы> ::==
    BOUNDING_BOX = ( { <Xmin>, <Ymin>, <Xmax>, <Ymax> |
                      <ключевое слово> = <координата>
                      [, <ключевое слово> = <координата>] ... }
                     )
    XMIN | YMIN | XMAX | YMAX

<опции> ::==
[ <параметр тесселяции> [, <параметр тесселяции>] ]
[ <параметр индекса> [, <параметр индекса>] ]

<параметр тесселяции> ::==
    GRIDS = { ( { <плотность>, <плотность>, <плотность>, <плотность> |
                  <уровень> = <плотность>
                  [, <уровень> = <плотность>] ... }
                 )
               | CELLS_PER_OBJECT = <целое>
               }

<плотность> ::= { LOW | MEDIUM | HIGH }
```

<уровень> ::= { LEVEL_1 | LEVEL_2 | LEVEL_3 | LEVEL_4 }

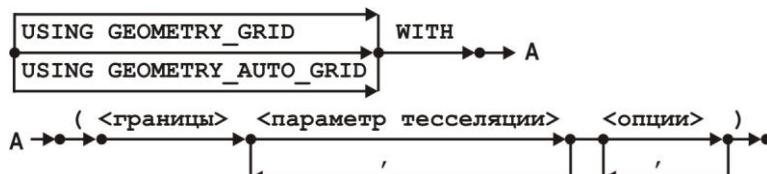
<параметр индекса> ::=

```
{ PAD_INDEX = { ON | OFF }
| FILLFACTOR = <целое>
| SORT_IN_TEMPDB = { ON | OFF }
| IGNORE_DUP_KEY = OFF
| DROP_EXISTING = { ON | OFF }
| ONLINE = OFF
| ALLOW_ROW_LOCKS = { ON | OFF }
| ALLOW_PAGE_LOCKS = { ON | OFF }
| MAXDOP = <максимальный уровень параллельности>
```

}



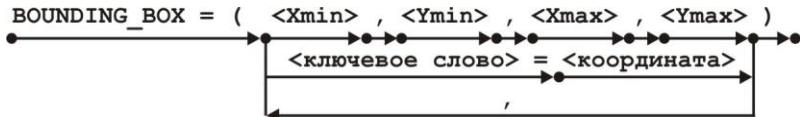
Граф 6.8. Синтаксис оператора CREATE SPATIAL INDEX



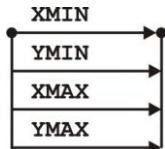
Граф 6.9. Синтаксис геометрической тесселяции



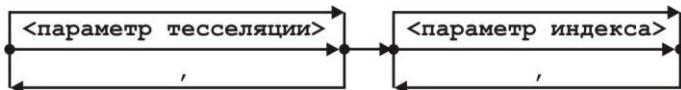
Граф 6.10. Синтаксис географической тесселяции



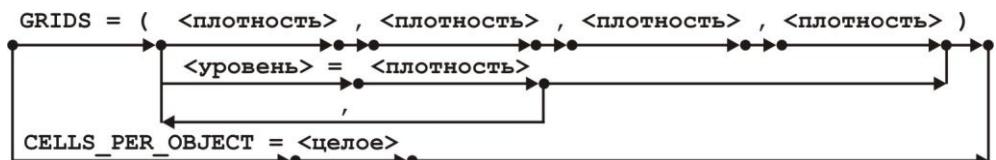
Граф 6.11. Синтаксис конструкции "границы"



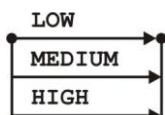
Граф 6.12. Синтаксис ключевого слова



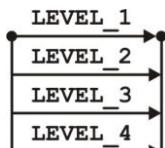
Граф 6.13. Синтаксис опции



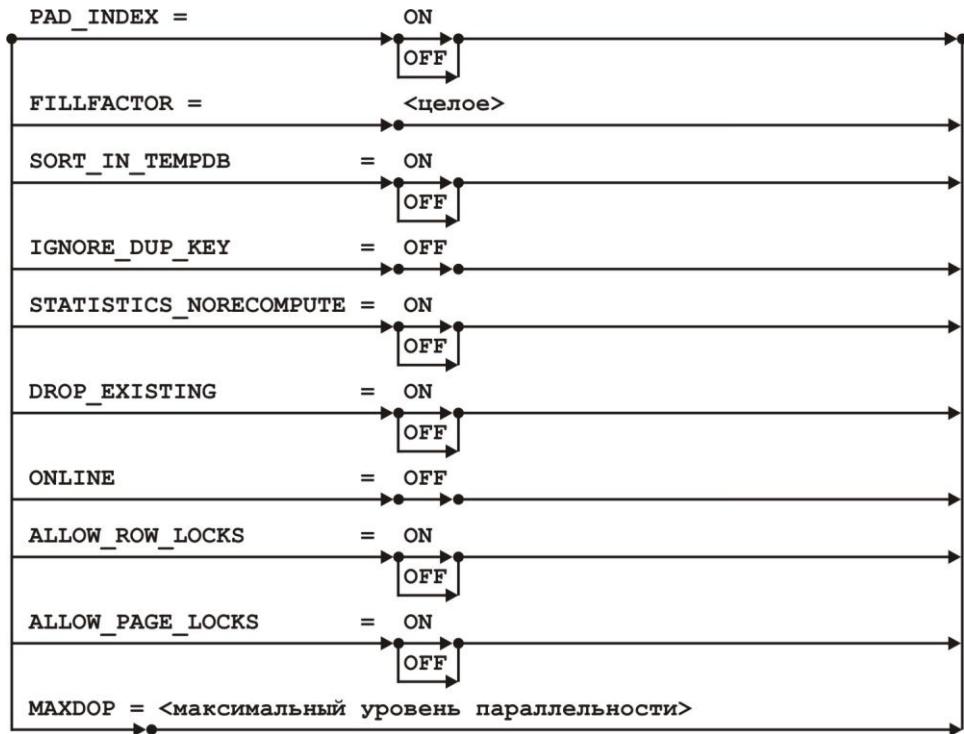
Граф 6.14. Синтаксис параметра тесселяции



Граф 6.15. Синтаксис конструкции "плотность"



Граф 6.16. Синтаксис конструкции "уровень"



Граф 6.17. Синтаксис параметра индекса

В процессе индексирования пространственного столбца выполняется декомпозиция пространства (содержимого столбца) в сеточную иерархию и так называемая **тесселяция**.

При декомпозиции производится преобразование пространства в четырехуровневую сеточную иерархию. Декомпозиция выполняется одинаково для геометрического и географического столбца и не зависит от используемых единиц изменения.

Для геометрического типа данных в предложении `BOUNDING_BOX` указываются четыре координаты ограничивающего прямоугольника.

После ключевого слова `GRIDS` в позиционном или ключевом формате задаются плотность (размер) сетки на каждом из четырех уровней. Ключевое слово `LOW` указывает размер сетки 4×4 элемента, `MEDIUM` — 8×8 , `HIGH` — 16×16 .

После декомпозиции выполняется тесселяция, в результате которой окончательно формируется пространственный индекс, позволяющий ускорить процесс выборки данных на основании запроса по пространственному столбцу.

Если в процессе вашей деятельности нужно использовать пространственные типы данных и соответствующие индексы, рекомендую обратиться к фирменной документации, а лучше — найти хорошую книгу по этим вопросам.

6.2.6. Удаление индекса

Удаление индекса, созданного пользователем, осуществляется оператором `DROP INDEX`. Оператор позволяет удалить любой индекс — обычный, XML или пространственный. Нельзя удалить индекс, автоматически созданный системой для поддержания ограничений первичного или уникального ключа. Такие индексы можно удалить, лишь удалив ограничение `PRIMARY KEY` или `UNIQUE` (оператор `ALTER TABLE`, предложение `DROP CONSTRAINT`), если от этих ограничений не зависят другие объекты базы данных — внешние ключи других или тех же самых таблиц, ссылающиеся на соответствующий первичный или уникальный ключ. Синтаксис оператора `DROP INDEX` представлен в листинге 6.5 и в соответствующем R-графе (графы 6.18—6.21).

Листинг 6.5. Синтаксис оператора `DROP INDEX`

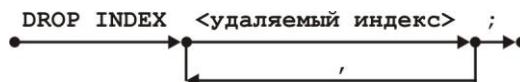
```

DROP INDEX <удаляемый индекс> [, <удаляемый индекс>] ... ;

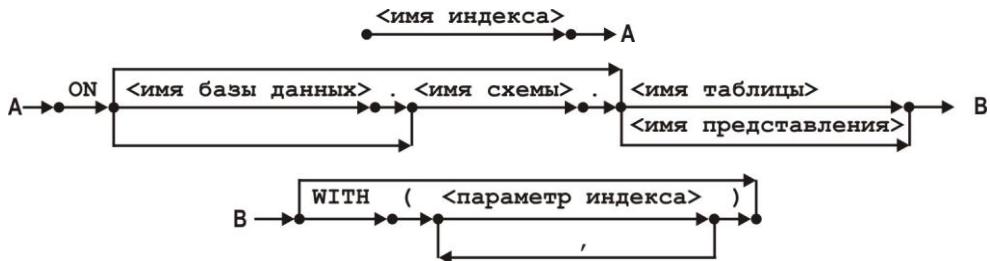
<удаляемый индекс> ::= <имя индекса>
  ON [ [<имя базы данных>.]<имя схемы>.]
    { <имя таблицы> | <имя представления> }
    [ WITH (<параметр индекса> [, <параметр индекса>]...) ]

<параметр индекса> ::=
{   MAXDOP = <максимальный уровень параллельности>
  | ONLINE = { ON | OFF }
  | <перемещение строк>
}

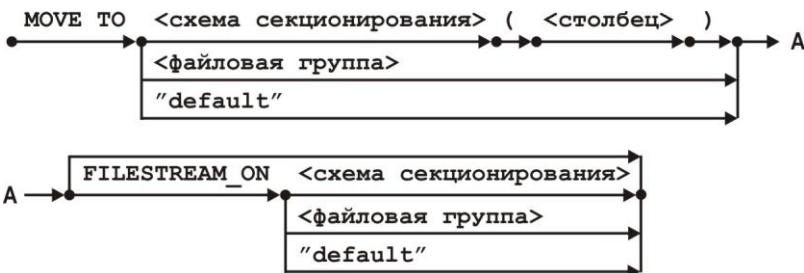
<перемещение строк> ::=
MOVE TO
{   <схема секционирования> (<столбец>)
  | <файловая группа>
  | "default"
}
[ FILESTREAM_ON
{   <схема секционирования>
  | <файловая группа>
  | "default"
}
]
```



Граф 6.18. Синтаксис оператора `DROP INDEX`



Граф 6.20. Синтаксис параметра индекса



Граф 6.21. Синтаксис перемещения строк

В одном операторе можно задать удаление нескольких индексов. Пространство базы данных, использовавшееся для удаленного индекса, может быть заполнено любыми другими данными.

В необязательном предложении **WITH** можно указать параметры удаляемого индекса.

Параметр **MAXDOP**, как и при создании индекса, позволяет установить максимальное количество используемых процессоров при выполнении параллельных операций с индексами. Параметр может иметь значение от 0 до 64.

Параметр **MOVE TO** может указываться только при удалении кластерных индексов. Он указывает, куда должны перемещаться строки таблицы при удалении индекса.

Параметр **ONLINE** определяет возможность использования таблицы или представления, для которого удаляется индекс, другими процессами во время удаления индекса. Если задано **ON**, то блокировка соответствующего объекта не осуществляется. **OFF** (значение по умолчанию) вызывает блокировку объекта. В случае блокировки операция удаления индекса, в особенности при большом количестве строк таблицы, выполняется более быстро.

Напомню, что в случае кластерного индекса строки таблицы располагаются в конечных узлах индексного дерева.

Таблицу можно сделать секционированной, строки можно разместить в указанной файловой группе или предоставить системе решить, куда будут помещаться строки таблицы.

При указании схемы секционирования и разделяющего столбца таблица станет секционированной. Сама схема секционирования и соответствующая функция секционирования уже должны существовать в базе данных.

При задании имени файловой группы строки таблицы будут помещены в файлы указанной файловой группы.

Если задано "default", то размещение строк определит система.

Параметр FILESTREAM_ON может задаваться, если таблица, у которой удаляется кластерный индекс, содержит один или более столбцов файловых потоков. Здесь можно указать схему секционирования, конкретную файловую группу или задать значение по умолчанию "default".

6.2.7. Изменение индекса

Для изменения характеристик созданного пользователем индекса для таблицы или представления используется оператор ALTER INDEX. Синтаксис оператора ALTER INDEX представлен в листинге 6.6 и в соответствующих R-графах (графы 6.22—6.25).

Листинг 6.6. Синтаксис оператора ALTER INDEX

```
ALTER INDEX { <имя индекса> | ALL }
  ON [ [<имя базы данных>.]<имя схемы>.]
    { <имя таблицы> | <имя представления> }
  { REBUILD
    [ [ PARTITION = ALL ]
      [ WITH (<параметр перестроения> [, <параметр перестроения>]...)
    | [ PARTITION = <номер секции>
      [ WITH (<параметр секции> [, <параметр секции>]...)
    ]
  ]
  | DISABLE
  | REORGANIZE
    [ PARTITION = <номер секции> ]
    [ WITH ( LOB_COMPACTION = { ON | OFF } )
  | SET ( <параметр индекса> [, <параметр индекса>]... )
};

<параметр перестроения> ::==
{ PAD_INDEX = { ON | OFF }
| FILLFACTOR = <целое>
| SORT_IN_TEMPDB = { ON | OFF }
```

```

| IGNORE_DUP_KEY = { ON | OFF }
| STATISTICS_NORECOMPUTE = { ON | OFF }
| ONLINE = { ON | OFF }
| ALLOW_ROW_LOCKS = { ON | OFF }
| ALLOW_PAGE_LOCKS = { ON | OFF }
| MAXDOP = <максимальный уровень параллельности>
| DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS ({ <номер секции> | <номер> TO <номер> }
                    [, {<номер секции> | <номер> TO <номер>}] ...) ]
}

```

<параметр секции> ::=

```

{   SORT_IN_TEMPDB = { ON | OFF }
| MAXDOP = <максимальный уровень параллельности>
| DATA_COMPRESSION = { NONE | ROW | PAGE }
}

```

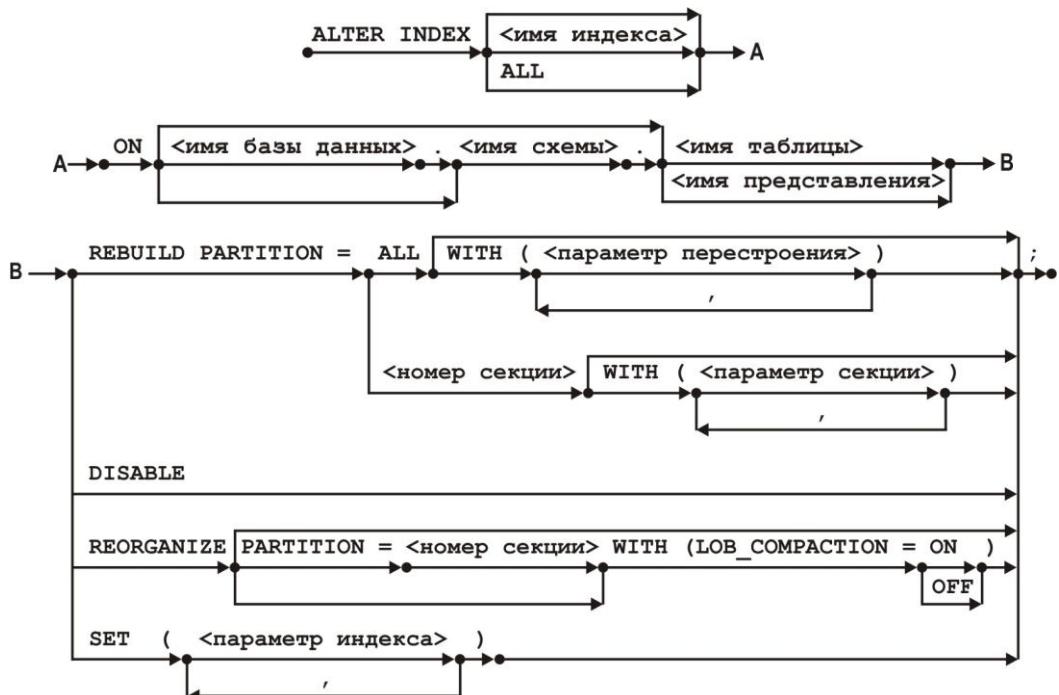
<параметр индекса> ::=

```

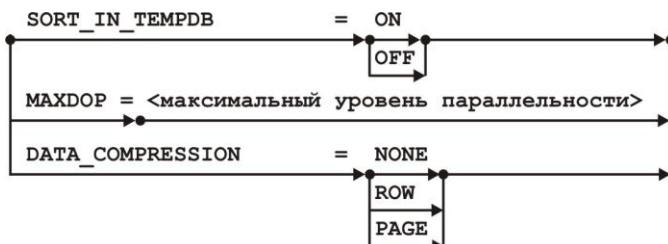
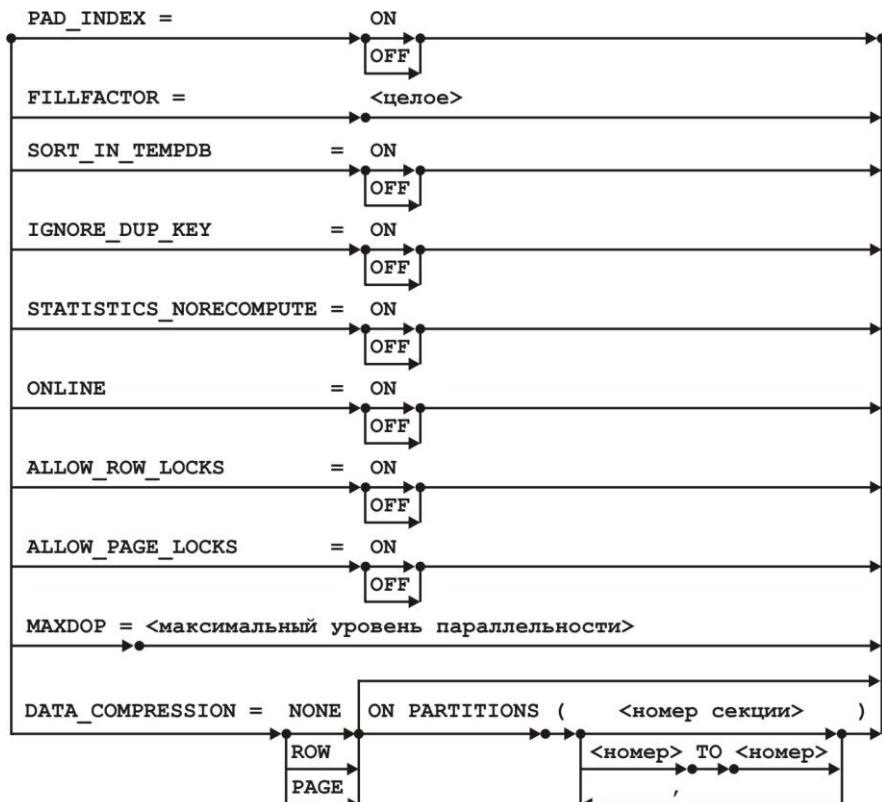
{   ALLOW_ROW_LOCKS = { ON | OFF }
| ALLOW_PAGE_LOCKS = { ON | OFF }
| IGNORE_DUP_KEY = { ON | OFF }
| STATISTICS_NORECOMPUTE = { ON | OFF }
}

```

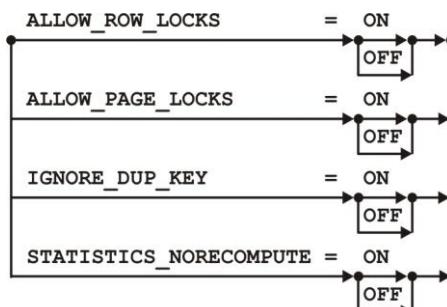
}



Граф 6.22. Синтаксис оператора ALTER INDEX



Граф 6.24. Синтаксис параметра секции



В операторе изменения индекса можно указать имя изменяемого индекса или задать ключевое слово `ALL`, которое указывает, что изменения должны относиться ко всем индексам данного объекта — таблицы или представления.

Если задано ключевое слово `REBUILD`, то выполняется перестроение существующего индекса (индексов при задании `ALL`). Если при этом указано ключевое слово `PARTITION`, то перестраиваются данные только из одной заданной секции или из всех секций (указано ключевое слово `ALL`). Параметры перестройки индексов задаются после ключевого слова `WITH`.

Ключевое слово `DISABLE` означает, что индекс отключается, т. е. становится недоступным для любых операций при обращении к соответствующей таблице (представлению).

Ключевое слово `REORGANIZE` означает запрос на реорганизацию конечного уровня индекса. Если задано ключевое слово `PARTITION`, то реорганизуются только данные указанной секции в случае секционированного индекса.

Если при реорганизации индекса указано и предложение `WITH` с параметром `LOB_COMPACTION = ON`, то выполняется и сжатие данных больших двоичных объектов LOB.

Остальные параметры оператора соответствуют параметрам оператора создания индекса.

6.3. Работа с индексами с помощью диалоговых средств Management Studio

Программа Management Studio предоставляет достаточно простые и удобные средства для выполнения действий с индексами таблиц и представлений. Некоторые варианты использования этой программы в отношении индексов мы с вами уже рассматривали при создании и изменении таблиц.

6.3.1. Создание индекса в Management Studio

Для создания индекса таблицы в Management Studio необходимо щелкнуть правой кнопкой мыши по папке **Indexes** соответствующей таблицы и в контекстном меню выбрать элемент **New Index**. Появится подменю, в котором будут перечислены возможные для текущей таблицы виды индексов:

- ◆ **Clustered Index** — кластерный индекс;
- ◆ **Non-Clustered Index** — некластерный индекс;
- ◆ **Primary XML Index** — первичный индекс XML;
- ◆ **Secondary XML Index** — вторичный индекс XML;
- ◆ **Spatial Index** — пространственный индекс;
- ◆ **Non-Clustered Columnstore Index** — индекс columnstore.

В зависимости от типов данных столбцов, входящих в состав таблицы, недопустимые для данной таблицы виды индексов будут представлены строками серого цвета.

Давайте создадим обычный, реляционный индекс для таблицы PEOPLE из нашей базы данных BestDatabase. Запустите на выполнение Management Studio, в панели **Object Explorer** раскройте папки **Databases**, **BestDatabase**, **Tables**. Раскройте папку таблицы `dbo.PEOPLE`. Щелкните правой кнопкой по папке **Indexes**. В появившемся контекстном меню выберите **New Index**, а затем **Non-Clustered Index**. Появится окно **New Index** (рис. 6.1).

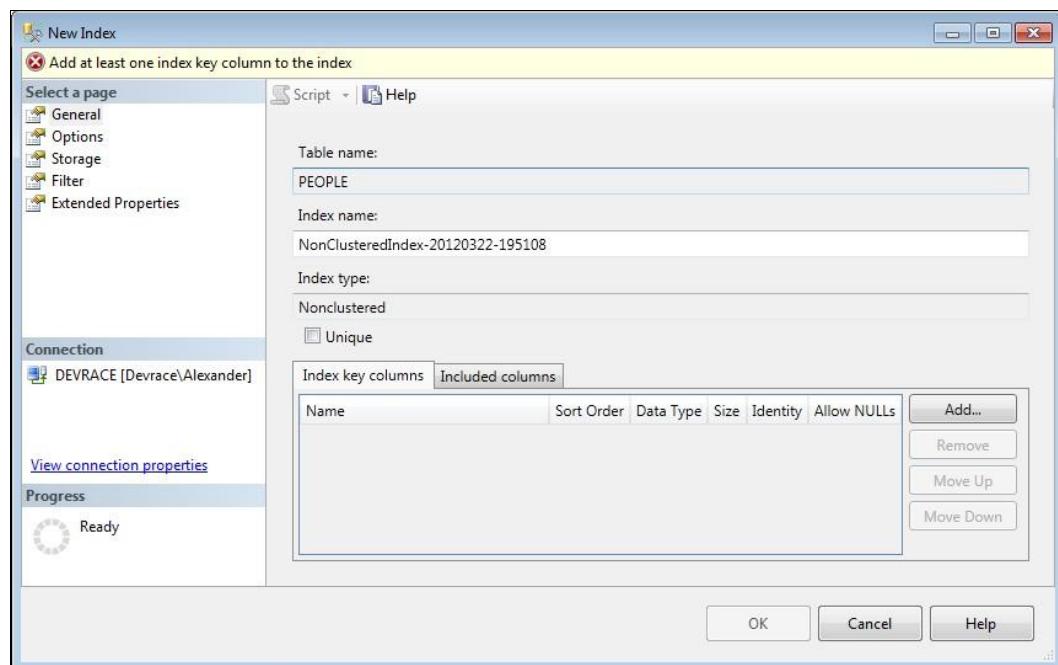


Рис. 6.1. Окно создания нового индекса. Вкладка **General**

В поле **Index name** введите имя индекса: `IND2_PEOPLE`. Чтобы указать, какие столбцы войдут в состав индекса, щелкните по кнопке **Add**. Появится окно, в котором будут перечислены имена столбцов таблицы. Отметьте столбцы `NAME1`, `NAME2`, `NAME3` (рис. 6.2) и щелкните по кнопке **OK**.

Опять появится окно создания индекса с выбранными столбцами (рис. 6.3). В окне можно корректировать список. Выделив столбец, можно переместить его выше по списку (кнопка **Move Up**), ниже (кнопка **Move Down**), можно удалить выделенный элемент (кнопка **Remove**), можно добавить новые элементы (кнопка **Add**).

Здесь же можно изменить упорядоченность индекса по любому столбцу. Для этого нужно в выделенном столбце щелкнуть мышью по полю с заголовком **Sort Order**. После этого в поле из выпадающего списка можно выбрать упорядоченность по

возрастанию значение (**Ascending**) или по убыванию (**Descending**). По умолчанию устанавливается возрастающий порядок.

Чтобы указать, что индекс уникальный, нужно отметить флажок в поле **Unique**.

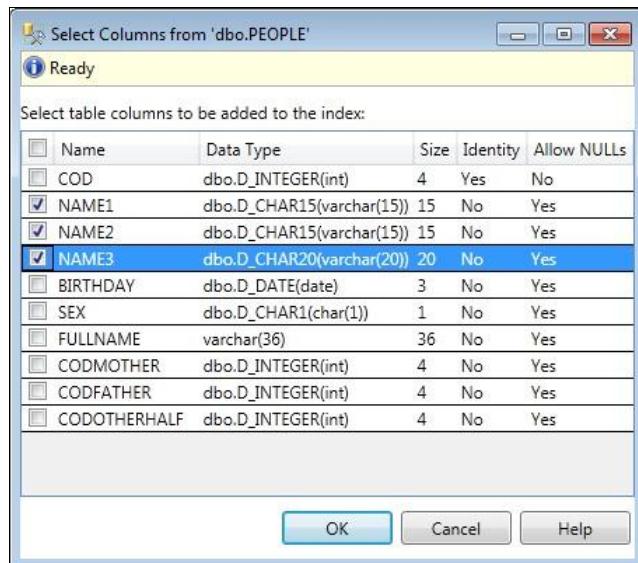


Рис. 6.2. Выбор столбцов для индекса

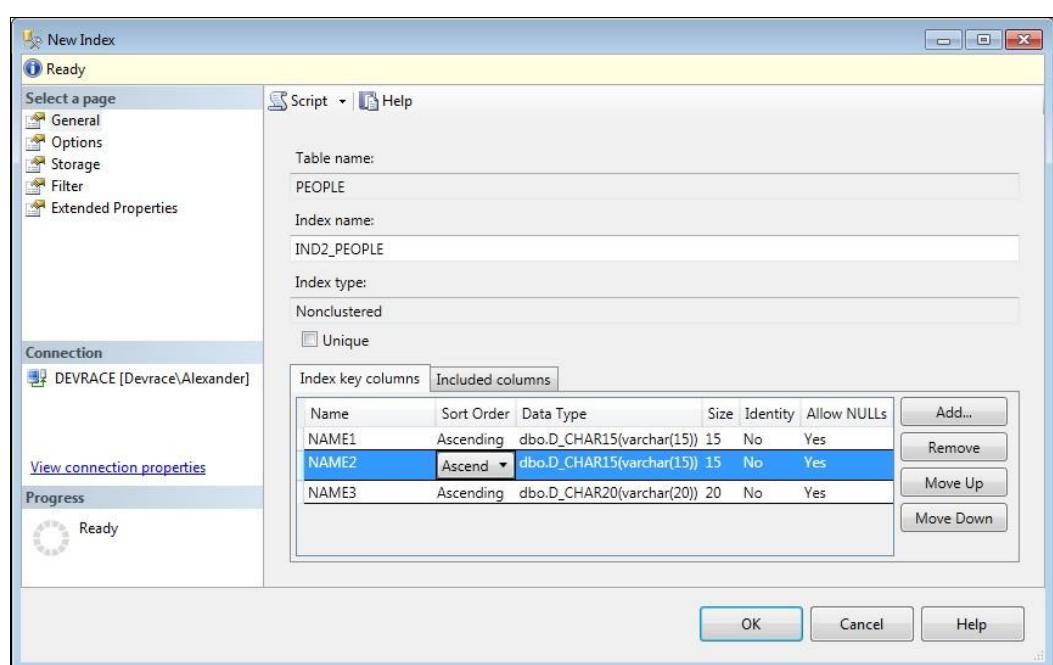


Рис. 6.3. Корректировка списка столбцов индекса

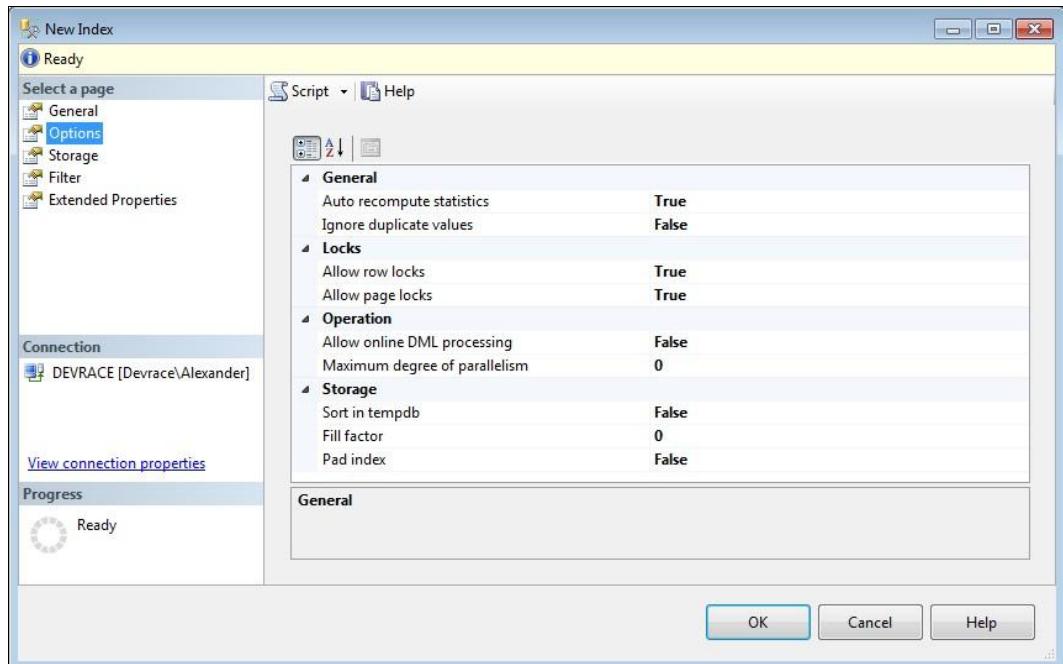


Рис. 6.4. Окно создания индекса. Вкладка Options

Если перейти на вкладку **Options**, то появится список параметров индекса (рис. 6.4). Все параметры мы уже рассмотрели, когда говорили об операторе создания индекса.

- ◆ **Auto recompute statistics** — задает автоматическое вычисление статистики.
- ◆ **Ignore duplicate values** — для уникального индекса значение **True** указывает, что в случае дублирования значения выдается предупреждающее сообщение, отменяются лишь те операторы, которые пытаются записать дубликат значений столбцов. Остальные действия будут выполнены. При значении **False** отменяется выполнение всего добавления данных.
- ◆ **Allow row locks** — задает возможность блокировки строк при обращении к индексу.
- ◆ **Allow page locks** — задает возможность блокировки страниц при обращении к индексу.
- ◆ **Allow online DML processing** — определяет возможность использования таблицы, для которой создается индекс, другими процессами во время создания индекса.
- ◆ **Maximum degree of parallelism** — определяет максимальное количество используемых процессоров при выполнении параллельных операций с индексами.
- ◆ **Sort in tempdb** — задает необходимость сохранения промежуточных результатов сортировки данных во временной базе данных **tempdb**.

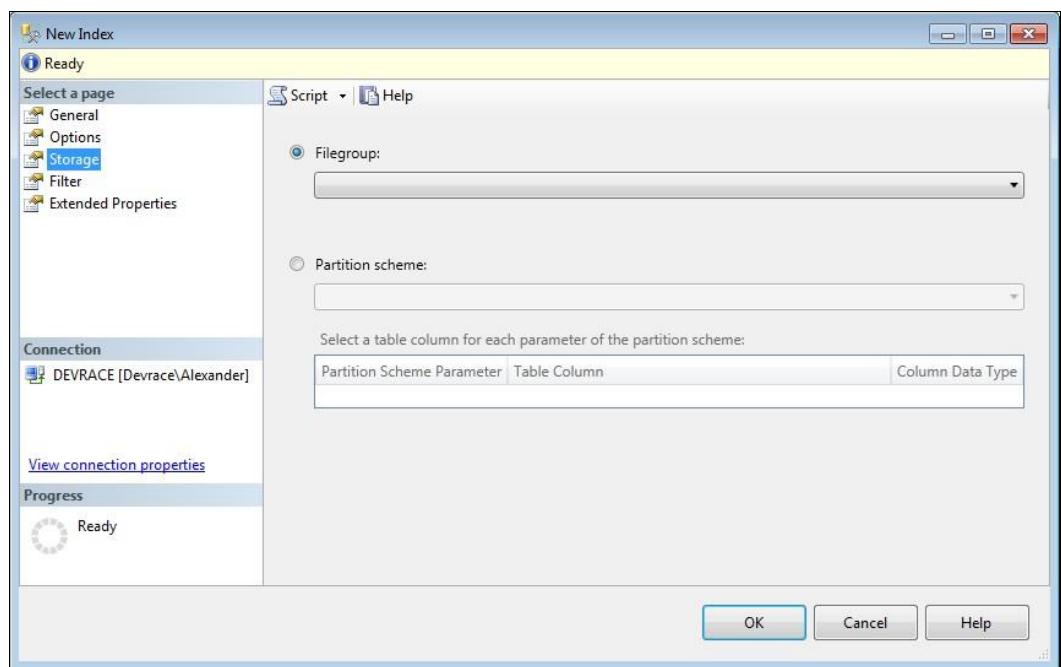


Рис. 6.5. Окно создания индекса. Вкладка **Storage**

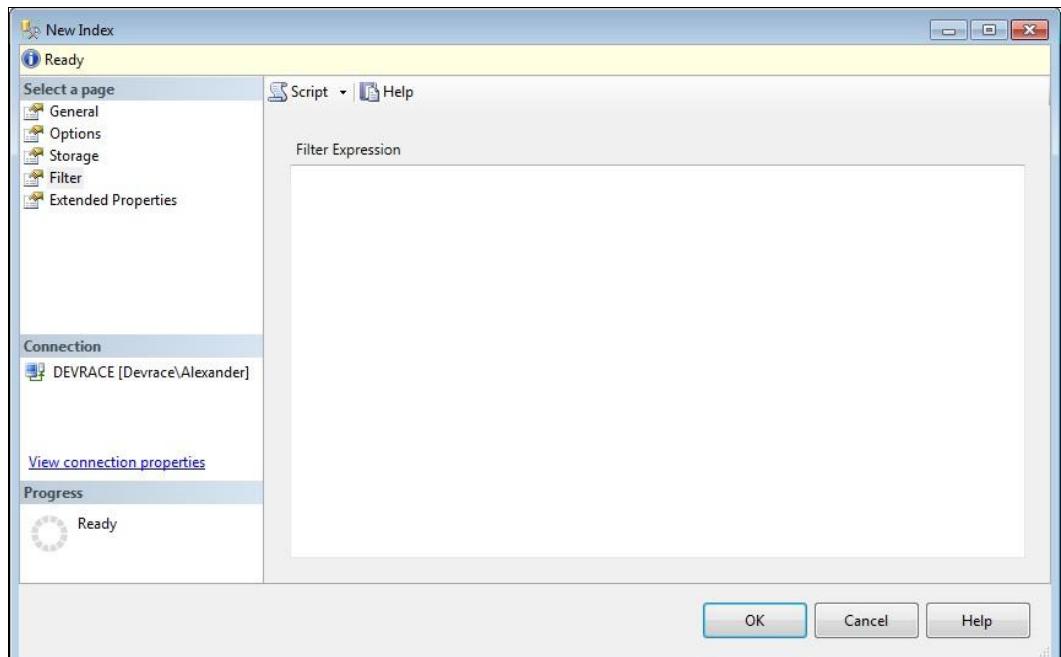


Рис. 6.6. Окно создания индекса. Вкладка **Filter**

- ◆ **Fill factor** — определяет процент заполнения страницы индекса самого нижнего уровня.
- ◆ **Pad index** — задает возможность использования разреженного индекса.

На вкладке **Storage** (рис. 6.5) можно выбрать файловую группу или схему секционирования для секционированного индекса.

Вкладка **Filter** (рис. 6.6) позволяет задать условие фильтрации индекса. Здесь задается выражение, которое определяет те строки исходного объекта, которые должны быть проиндексированы.

Для завершения создания индекса нужно щелкнуть мышью по кнопке **OK**. Новый индекс появится в списке индексов таблицы.

6.3.2. Удаление индекса в Management Studio

Для удаления индекса нужно щелкнуть правой кнопкой мыши по имени индекса в панели **Object Explorer** и в контекстном меню выбрать **Delete**. Можно также нажать клавишу <Delete>. После подтверждения удаления индекс будет удален из базы данных.

6.3.3. Изменение индекса в Management Studio

Для изменения состава индекса нужно щелкнуть правой кнопкой мыши по имени индекса в панели **Object Explorer** и в контекстном меню выбрать **Properties**. Появится окно в точности копирующее окно создания индекса (см. рис. 6.3). Здесь можно изменять состав, размещение и упорядоченность столбцов в индексе. На других вкладках можно изменять множество других характеристик индекса.

Для изменения имени индекса нужно щелкнуть правой кнопкой мыши по индексу и в появившемся контекстном меню выбрать **Rename**. Поле имени индекса станет доступным для изменения.

Для перестроения индекса в этом меню нужно выбрать **Rebuild**.

Чтобы сделать индекс недоступным, нужно выбрать **Disable**, а для реорганизации индекса выбирается **Reorganize**.

Что будет дальше?

В следующей главе 7 мы рассмотрим операторы, позволяющие добавлять, изменять и удалять данные таблиц базы данных.



ГЛАВА 7

Добавление, изменение и удаление данных

- ◆ Добавление данных. Оператор `INSERT`
- ◆ Изменение данных. Оператор `UPDATE`
- ◆ Удаление данных. Оператор `DELETE`
- ◆ Удаление всех строк таблицы. Оператор `TRUNCATE TABLE`
- ◆ Добавление, изменение, удаление данных. Оператор `MERGE`

7.1. Обобщенное табличное выражение

В операторах, осуществляющих работу с данными в базе данных, в предложении `WITH` может присутствовать одно или более обобщенных табличных выражений (Common Table Expression, *CTE*). В литературе также можно встретить термин "общее выражение таблицы".

СТЕ является временным именованным набором данных, т. е. его можно рассматривать как обычную таблицу базы данных. Этот набор данных получается при использовании оператора `SELECT`, обращающегося к одной или более реальных таблиц, присутствующих в базе данных. К этому набору данных можно обращаться по его имени в операторах, выполняющих действия с данными в базе данных.

Обобщенное табличное выражение может быть рекурсивным, т. е. оно может включать и ссылки на себя.

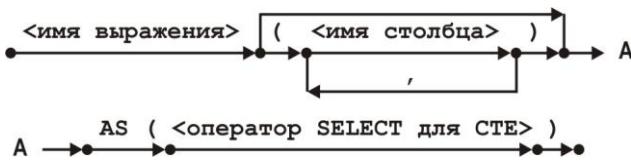
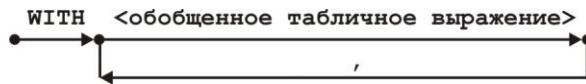
Синтаксис предложения `WITH` для обобщенного табличного выражения представлен в листинге 7.1 и в графическом виде (графы 7.1—7.2).

Листинг 7.1. Синтаксис предложения обобщенного табличного выражения

```
<предложение обобщенного табличного выражения> ::=  
WITH <обобщенное табличное выражение>  
[, <обобщенное табличное выражение>] ...
```

<обобщенное табличное выражение> ::=

<имя выражения> [(<имя столбца> [, <имя столбца>] ...)]
AS (<оператор SELECT для CTE>)



В одном предложении `WITH` может содержаться задание нескольких обобщенных табличных выражений. Они отделяются друг от друга запятыми.

Имя обобщенного табличного выражения должно быть уникальным среди имен объявляемых в предложении обобщенных табличных выражений. Интересно, что это имя может совпадать и с именем реальной таблицы или представления базы данных, если эта таблица или представление не используются в соответствующем операторе.

Количество указанных в скобках столбцов должно совпадать с количеством столбцов, возвращаемых оператором `SELECT` в предложении `AS`. По именам столбцов можно обращаться к соответствующим значениям. Имена столбцов, заданные в обобщенном табличном выражении, должны быть уникальны. Список имен столбцов не является обязательным, если все столбцы в запросе имеют уникальные имена.

Оператор `SELECT` в предложении `AS` формирует тот набор данных, который будет использован в качестве обобщенного табличного выражения. К этому набору данных можно обратиться при помощи оператора `SELECT`.

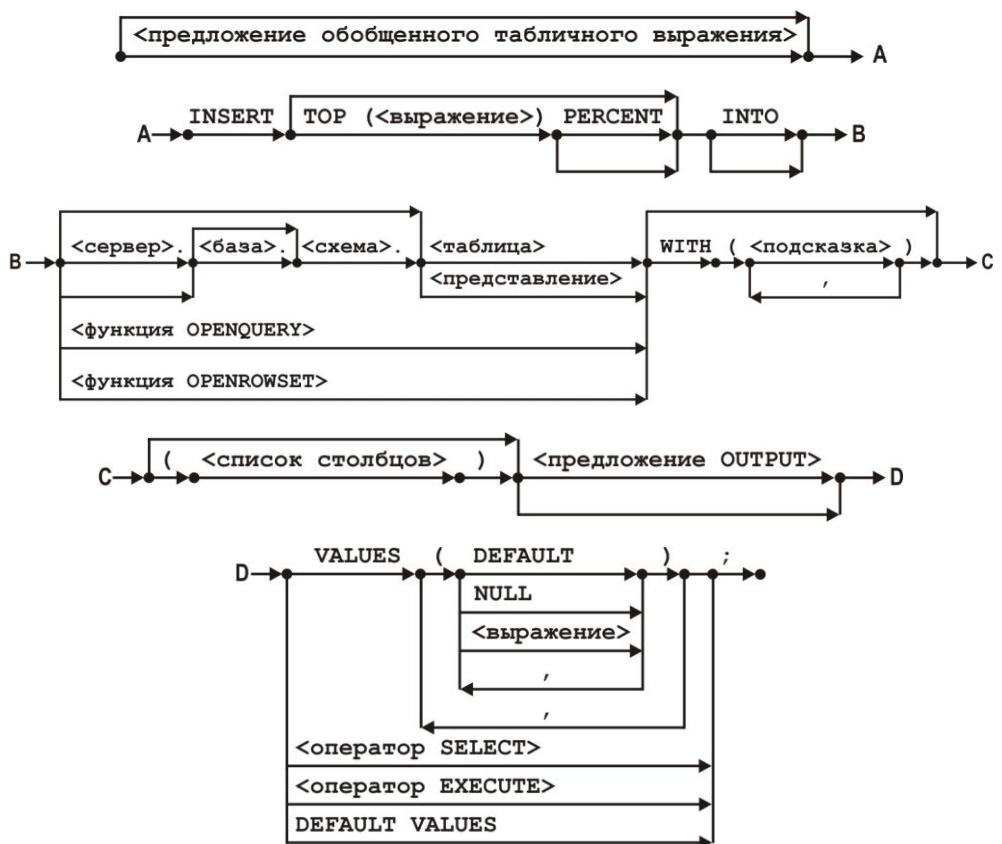
Если в предложении `WITH` присутствует несколько обобщенных табличных выражений, то они должны быть связаны одним из операторов `UNION ALL`, `UNION`, `EXCEPT`, `INTERSECT`.

7.2. Добавление данных (оператор `INSERT`)

Для добавления в таблицу (или в представление) одной или нескольких строк используется оператор `INSERT`. Его синтаксис представлен в листинге 7.2 и графике 7.3.

Листинг 7.2. Синтаксис оператора `INSERT`

```
[ <предложение обобщенного табличного выражения> ]
INSERT [ TOP (<выражение>) [ PERCENT ] ]
[ INTO ] { [<сервер>.] [[<база данных>.]<схема>.]
{ <таблица> | <представление> }
| { <функция OPENQUERY> | <функция OPENROWSET> }
}
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
[ (<список столбцов>) ] [ <предложение OUTPUT> ]
{ VALUES ( { DEFAULT | NULL | <выражение> }
[, { DEFAULT | NULL | <выражение> }]...)
[, ( { DEFAULT | NULL | <выражение> }
[, { DEFAULT | NULL | <выражение> }]...)...
| <оператор SELECT>
| <оператор EXECUTE>
| DEFAULT VALUES
} ;
```



Граф 7.3. Синтаксис оператора INSERT

В начале оператора может быть задано обобщенное табличное выражение, которое задает временный именованный набор данных. По этому имени к набору данных можно обращаться в операторе `INSERT`. Набор данных является временным, он существует только в процессе выполнения оператора добавления.

В предложении `TOP` указывается количество либо процент (ключевое слово `PERCENT`) добавляемых строк в этом операторе. Оператор может задавать добавление нескольких строк. При наличии предложения `TOP` количество добавляемых строк может быть уменьшено.

После необязательного ключевого слова `INTO` указывается, куда именно добавляются строки. Это может быть таблица, находящаяся в базе данных, принадлежащей текущему экземпляру сервера. Таблица может быть указана явно или таблица определяется при задании представления, которое должно быть изменяемым. О представлениях мы поговорим далее в главе 9.

Оператор позволяет также добавлять данные в таблицы баз данных, находящихся в других экземплярах сервера и даже в других системах управления базами данных при задании имени связанного сервера или при использовании функций `OPENQUERY` и `OPENROWSET`.

Связанный сервер (*linked server*) — это сервер базы данных, который создается хранимой процедурой `sp_addlinkedserver`. Его также можно создать в Management Studio. В результате становится возможным выполнять так называемые разнородные гетерогенные запросы к источникам данных OLE DB. *OLE* (*Object Linking and Embedding*) — набор протоколов Microsoft, позволяющих связывать и встраивать различные объекты, в данном случае базы данных.

В частности, связанными серверами могут быть другие экземпляры и другие версии MS SQL Server, базы данных Access, таблицы Excel, базы данных Oracle и IBM DB2.

Вместо имени связанного сервера можно задать обращение к функции `OPENDATASOURCE`, которая осуществит подключение к источнику OLE DB.

К источнику данных OLE DB также можно обратиться при помощи функций `OPENQUERY` и `OPENROWSET`. В этом случае нужно будет задать обращение к таблице связанного сервера, в которую будут добавляться данные.

Подробнее о связанных серверах см. документацию Books Online.

Если в операторе задается добавление данных в таблицу, то после ключевого слова `WITH` можно в скобках указать так называемые "табличные подсказки" или "табличные указания" (в оригинале "table hint").

Таких подсказок для оператора добавления существует около двух десятков. Они позволяют задавать некоторые режимы выполнения добавления. Это и поведение системы по отношению к триггерам, ограничениям, автоинкрементным первичным ключам; задание характеристик транзакции и ряд других. Посмотрите Books Online.

В операторе можно в круглых скобках указать список столбцов, которым явно будут присвоены значения. Если список не указан, то столбцам будут присваиваться

заданные далее значения с учетом того порядка, в котором столбцы существуют в таблице. Вообще говоря, есть хорошая рекомендация всегда указывать список столбцов. Если какой-то столбец отсутствует в списке, то ему будет присвоено значение по умолчанию. Если для столбца не задано такое значение, то ему будет присвоено `NULL`. Если же для столбца указана недопустимость неопределенного значения, то выполнение оператора вызовет ошибку.

Необязательное предложение `OUTPUT` позволяет поместить все добавленные оператором строки либо в локальную переменную с типом данных `TABLE`, либо в указанную таблицу базы данных. Это может быть обычная или временная таблица.

Сами значения, помещаемые в столбцы новой строки таблицы, могут быть заданы одним из четырех способов:

- ◆ в предложении `VALUES` с перечислением значений;
- ◆ при помощи оператора `SELECT`, который возвращает одну или более строк таблицы;
- ◆ при вызове хранимой процедуры оператором `EXECUTE`;
- ◆ присваиванием значений по умолчанию всем столбцам (`DEFAULT VALUES`).

Количество значений, возвращаемых операторами или указанных явно, должно соответствовать количеству заданных столбцов таблицы.

В случае задания предложения `VALUES` можно в одном операторе добавлять множество строк, повторяя через запятую список значений в скобках. Этот вариант синтаксиса хорошо виден на R-графе. В предложении `VALUES` параметр `<выражение>` означает и собственно выражение, которое возвращает конкретное значение, и константу.

Рассмотрим несколько примеров помещения данных в таблицы нашей базы данных `BestDatabase`.

Мы создали базу данных `BestDatabase` и множество таблиц, в том числе таблицы, описывающие страны, их регионы, районы. В первом примере даются фрагменты простых операторов, помещающих данные в эти таблицы.

Пример 7.1. Добавление данных в таблицы стран, регионов, районов

```
USE BestDatabase;
GO
SET NOCOUNT ON;
-- Страны
BEGIN TRANSACTION;
INSERT INTO REFCTR (CODCTR, FULLNAME, NAME, CAPITAL, TELCODE)
    VALUES ('RUS', 'Российская Федерация', 'Россия', 'Москва', '+7');
INSERT INTO REFCTR (CODCTR, FULLNAME, NAME, CAPITAL, TELCODE)
    VALUES ('USA', 'Соединенные Штаты Америки', 'США', 'Вашингтон', '+1');
GO
```

```
-- Регионы
/* Россия */
INSERT INTO REFREG (CODCTR, CODREG, NAME, CENTER)
    VALUES ('RUS', '01', 'Алтайский край', 'Барнаул');
INSERT INTO REFREG (CODCTR, CODREG, NAME, CENTER)
    VALUES ('RUS', '03', 'Краснодарский край', 'Краснодар');
/* Соединенные Штаты Америки */
INSERT INTO REFREG (CODCTR, CODREG, CENTER, NAME)
    VALUES ('USA', 'AL', 'MONTGOMERY', 'Alabama');
INSERT INTO REFREG (CODCTR, CODREG, CENTER, NAME)
    VALUES ('USA', 'AK', 'JUNEAU', 'Alaska');
-- Районы
-- Россия
INSERT INTO REFAREA (CODCTR, CODREG, CODAREA, NAME, CENTER)
    VALUES ('RUS', '01', '201', 'Алейский район', 'Алейск'),
    ('RUS', '01', '202', 'Алтайский район', 'Алтайский'),
    ('RUS', '01', '203', 'Баевский район', 'Баево');
-- USA
INSERT INTO REFAREA (CODCTR, CODREG, CODAREA, CENTER, NAME)
    VALUES ('USA', 'AL', '001', '', 'Autauga County'),
    ('USA', 'AL', '003', '', 'Barbour County');
GO
COMMIT;
SET NOCOUNT OFF;
```

Здесь при добавлении стран и регионов мы использовали один оператор `INSERT` для одной строки таблиц. При добавлении районов был использован один оператор `INSERT` для добавления всех строк районов.

Оператор `SET NOCOUNT ON;` используется для того, чтобы после каждой операции добавления строки мы не получали сообщение, что одна строка добавлена.

В следующем примере добавим несколько записей в таблицу людей.

Таблица людей, `PEOPLE`, была создана в следующем виде:

```
/** Список людей ***/
CREATE TABLE PEOPLE
( COD      INTEGER IDENTITY(1, 1)
            NOT NULL,      /* Код человека */
  NAME1    VARCHAR(15),      /* Имя */
  NAME2    VARCHAR(15),      /* Отчество */
  NAME3    VARCHAR(20),      /* Фамилия */
  BIRTHDAY DATE,           /* Дата рождения */
  SEX      CHAR(1) DEFAULT '0', /* Пол:
                                /* 0 — мужской,
                                /* 1 — женский.
  FULLNAME AS          /* Вычисляемый столбец
    (NAME3 + ' ' + NAME1 + ' ' + NAME2),
```

```

CODMOTHER      INTEGER
               DEFAULT NULL,      /* Ссылка на мать */
CODFATHER      INTEGER
               DEFAULT NULL,      /* Ссылка на отца */
CODOITHERHALF INTEGER
               DEFAULT NULL,      /* Ссылка на супруга */
CONSTRAINT PK_PEOPLE PRIMARY KEY (COD),
CONSTRAINT CH_PEOPLE CHECK (SEX IN ('0', '1')),
CONSTRAINT FK1_PEOPLE
FOREIGN KEY (CODMOTHER) REFERENCES PEOPLE (COD)
ON DELETE NO ACTION,
CONSTRAINT FK2_PEOPLE
FOREIGN KEY (CODFATHER) REFERENCES PEOPLE (COD)
ON DELETE NO ACTION,
CONSTRAINT FK3_PEOPLE
FOREIGN KEY (CODOITHERHALF) REFERENCES PEOPLE (COD)
ON DELETE NO ACTION
);

```

Первичным ключом в таблице является автоинкрементный столбец COD, который описывается с атрибутом IDENTITY. При добавлении новой строки в эту таблицу система автоматически формирует уникальное значение для такого столбца. По этой причине не нужно явно задавать для него никакого значения.

Пример 7.2. Добавление данных в таблицу людей

```

USE BestDatabase;
GO
SET NOCOUNT ON;
SET DATEFORMAT dmy;
/* Пример 1 */
INSERT INTO PEOPLE (NAME1, NAME2, NAME3, BIRTHDAY, SEX)
VALUES ('НАТАЛЬЯ', 'АЛЕКСАНДРОВНА', 'ИВАНОВА', '01.12.1961', '1'),
       ('АЛЕКСАНДР', 'АЛЕКСАНДРОВИЧ', 'ИВАНОВ', '07.01.1960', '0');
...
/* Пример 2 */
INSERT INTO PEOPLE (NAME1, NAME2, NAME3, BIRTHDAY, SEX)
VALUES ('НИНА', 'СЕРГЕЕВНА', 'ШАФРАН', '01.01.1941', '1');
INSERT INTO PEOPLE (NAME1, NAME2, NAME3, BIRTHDAY, SEX, CODOITHERHALF)
VALUES ('ИВАН', 'АНТОНОВИЧ', 'ШАФРАН', '01.01.1942', '0',
       (SELECT COD FROM PEOPLE
        WHERE NAME1 = 'НИНА' AND NAME2 = 'СЕРГЕЕВНА' AND NAME3 = 'ШАФРАН'));
...
/* Пример 3 */
INSERT INTO PEOPLE (NAME1, NAME2, NAME3, BIRTHDAY, SEX, CODOITHERHALF)
VALUES ('ПРАСКОВЬЯ', 'СТЕПАНОВНА', 'ПИНТЕРА', '01.01.1946', '1',
       (SELECT COD FROM PEOPLE
        WHERE NAME1 = 'ПРАСКОВЬЯ' AND NAME2 = 'СТЕПАНОВНА' AND NAME3 = 'ПИНТЕРА'));

```

```

WHERE NAME1 = 'НИКОЛАЙ' AND NAME2 = 'НИКОЛАЕВИЧ'
AND NAME3 = 'ПИНТЕРА'
AND BIRTHDAY = '01.01.1940')) ;
...

```

Первые два оператора (пример 1) добавляют в таблицу сведения о людях, для которых нет данных об их родственных связях, хотя можно предположить, что эти люди являются супругами.

Следующие два оператора (пример 2) добавляют данные по людям, на которые будут ссылки в дальнейших строках таблицы. Здесь во второй записи для того, чтобы указать, что конкретный человек является супругой добавляемого человека, для получения кода супруги (столбец CODOTHERHALF) используется оператор SELECT, выбирающий из базы данных значение кода этой супруги. Для однозначной идентификации записи используются значения фамилии, имени и отчества. В данном случае оператор вернет ровно одно значение, поскольку в базе данных существует только одна такая строка таблицы, имеющая такие значения столбцов.

В последнем операторе (пример 3) при добавлении строки таблицы для определения кода супруга в операторе SELECT помимо фамилии, имени и отчества используется и дата рождения супруга, поскольку в таблице существует более одной строки с такими значениями фамилии, имени и отчества.

В примере 7.3 для помещения данных в таблицу используется оператор SELECT. Здесь вначале создается новая таблица REFREGRUS, которая должна содержать список регионов России. Затем оператор INSERT добавляет в нее все регионы России. Для этого в операторе SELECT используется предложение WHERE, которое задает выборку только тех регионов, которые относятся к Российской Федерации.

Пример 7.3. Добавление данных в таблицу регионов России

```

USE BestDatabase;
GO
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'REFREGRUS')
    DROP TABLE REFREGRUS;
GO
CREATE TABLE REFREGRUS
( CODREG    CHAR(2) NOT NULL,    /* Код региона */
  NAME      VARCHAR(110),          /* Название региона */
  CENTER    VARCHAR(25),          /* Название центра региона */
  CONSTRAINT PK_REFREGRUS PRIMARY KEY (CODREG)
);
GO
INSERT INTO REFREGRUS (CODREG, NAME, CENTER)
SELECT CODREG, NAME, CENTER FROM REFREG
    WHERE CODCTR = 'RUS';
GO

```

Вначале проверяется, существует ли такая таблица в базе данных. Если да, то она удаляется и создается заново. Затем в таблицу добавляются соответствующие регионы.

В примере 7.4 выполняется создание такой же таблицы и в нее добавляются данные по регионам России, только здесь в операторе `INSERT` используется обобщенное табличное выражение.

Пример 7.4. Добавление данных в таблицу регионов России с использованием СТЕ

```
USE BestDatabase;
GO
IF EXISTS (SELECT * FROM sys.tables
            WHERE NAME = 'REFREGRUS')
    DROP TABLE REFREGRUS;
GO
CREATE TABLE REFREGRUS
( CODREG    CHAR(2) NOT NULL,    /* Код региона */
  NAME      VARCHAR(110),        /* Название региона */
  CENTER    VARCHAR(25),        /* Название центра региона */
  CONSTRAINT PK_REFREGRUS PRIMARY KEY (CODREG)
);
GO
WITH REG (CODCTR, CODREG, NAME, CENTER) AS
( SELECT CODCTR, CODREG, NAME, CENTER FROM REFREG
  WHERE CODCTR = 'RUS' )
INSERT INTO REFREGRUS (CODREG, NAME, CENTER)
  SELECT CODREG, NAME, CENTER FROM REG;
GO
```

В начале оператора `INSERT` объявляется обобщенное табличное выражение `REG`. Данные для добавления выбираются из этого выражения.

7.3. Изменение данных (оператор `UPDATE`)

Оператор `UPDATE` позволяет изменить существующее значение одного или большего количества столбцов в одной строке или во множестве строк одной таблицы.

Синтаксис оператора представлен в листинге 7.3 и в графах 7.4—7.5.

Листинг 7.3. Синтаксис оператора `UPDATE`

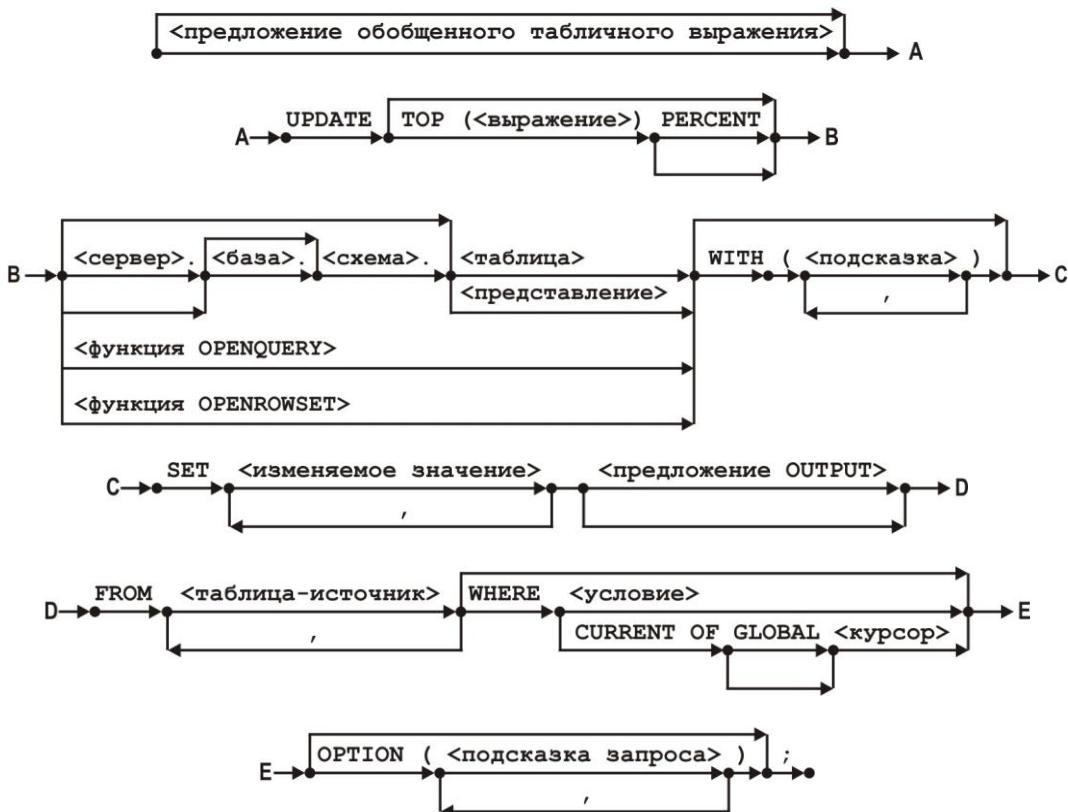
```
[ <предложение обобщенного табличного выражения> ]
UPDATE [ TOP (<выражение>) [ PERCENT ] ]
{ [<сервер>.][<база данных>.]<схема>.] 
  { <таблица> | <представление> }
| { <функция OPENQUERY> | <функция OPENROWSET> }
}
```

```
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
SET <изменяемое значение> [, <изменяемое значение>...]
[ <предложение OUTPUT> ]
[ FROM (<таблица-источник> [, <таблица-источник>]...)
[ WHERE { <условие>
          | CURRENT OF [ GLOBAL ] <имя курсора>
        }
  ]
[ OPTION (<подсказка запроса> [, <подсказка запроса>]...) ] ;
```

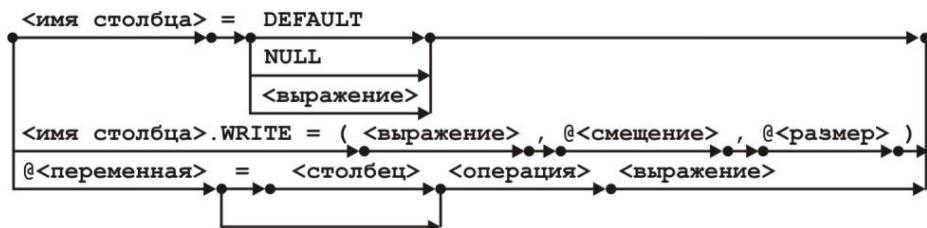
<изменяемое значение> ::=

```
{ <имя столбца> = { DEFAULT | NULL | <выражение> }
| <имя столбца>.WRITE(<выражение>, @<смещение>, @<размер>)
| @<переменная> [= <столбец>] <операция> <выражение>
  [, @<переменная> [= <столбец>] <операция> <выражение>]...
}
```

<операция> ::= += | -= | *= | /= | %= | &= | ^= | |= | =



Граф 7.4. Синтаксис оператора UPDATE



В начале оператора может быть задано обобщенное табличное выражение.

В необязательном предложении **TOP** указывается количество либо процент (ключевое слово **PERCENT**) строк, изменяемых в этом операторе.

Далее указывается таблица, для которой выполняются изменения. Это может быть таблица или представление (ссылающееся на таблицу) текущего экземпляра сервера, это также может быть таблица на связанном сервере. Немного об этом мы поговорили при рассмотрении оператора **INSERT**.

Здесь могут использоваться и подсказки таблицы, заданные в предложении **WITH**.

Сами изменения задаются в предложении **SET**. Задаваемые элементы отделяются друг от друга запятыми. Существует несколько вариантов указания столбцов и новых их значений.

В обычном варианте после имени столбца и знака равенства записывается константа, выражение, ключевое слово **DEFAULT** (присвоить значение по умолчанию) или **NULL** (присвоить неизвестное значение). Выражение может быть достаточно сложным. Это может быть и оператор **SELECT**, возвращающий ровно одно значение. Такой оператор должен быть заключен в круглые скобки.

Вариант, когда после имени столбца задается обращение к функции **.WRITE**, может быть использован для столбцов с типом данных **VARCHAR(MAX)**, **NVARCHAR(MAX)** и **VARBINARY(MAX)**. Функция позволяет заменить данные в столбце, начиная с позиции, заданной локальной переменной **@<смещение>**, размером, заданным локальной переменной **@<размер>**, на значение выражения, указанного первым параметром функции.

В третьем варианте используются локальные переменные. Им присваиваются конкретные значения. Эти же значения могут быть присвоены и столбцам изменяемой строки таблицы, когда конструкция представлена в виде:

@<переменная> [= <столбец>] <операция> <выражение>

Вначале столбцу таблицы присваивается значение, полученное в результате применения указанной операции к выражению, затем это же значение присваивается локальной переменной.

Здесь "операция" может иметь следующие значения:

- ◆ **=** — это обычное присваивание значения;
- ◆ **+=** — выполняется сложение указанного значения, полученного в результате вычисления выражения, с существующим значением столбца и результат присваивается переменной (и столбцу);

- ◆ == — из значения вычитается полученное значение и выполняется присваивание;
- ◆ *= — выполняется умножение и присваивание;
- ◆ /= — выполняется деление и присваивание;
- ◆ %= — получение остатка от деления и присваивание;
- ◆ &= — побитовая операция И и присваивание;
- ◆ ^= — побитовая операция исключающее ИЛИ и присваивание;
- ◆ |= — побитовая операция ИЛИ и присваивание.

Если в конструкции отсутствует имя столбца, то значение присваивается только локальной переменной.

Предложение OUTPUT возвращает измененные строки таблицы.

В предложении FROM задается список таблиц, использование которых позволяет ограничить количество строк изменяемой таблицы, к которым применяется операция обновления данных.

В предложении WHERE задаются условия выбора обновляемых строк таблицы. Здесь можно задать и очень сложное условие, и указать, что обновлению подвергается лишь одна строка, на которую ссылается конкретный курсор, текущая позиция курсора (вариант CURRENT OF). Курсор может быть локальным или глобальным (ключевое слово GLOBAL).

Необязательное предложение OPTION содержит подсказки оптимизатору запроса. Эти подсказки оказывают влияние на порядок выборки строк таблиц. Вообще говоря, оптимизатор запросов, который автоматически определяет стратегию выборки данных из таблицы (таблиц), формирует не худший алгоритм поиска данных. Если у вас найдутся идеи получше, попробуйте использовать подсказки запроса. Вначале посмотрите документацию Books Online.

В следующем примере 7.5 выполняется обновление данных в таблице персонала STAFF. Таблица была создана следующим оператором:

```
CREATE TABLE STAFF
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL, /* Код сотрудника – первичный ключ */
  CODEPEOPLE INTEGER,          /* Код человека из списка людей */
  CODORG    INTEGER,          /* Код организации */
  DUTIES    VARCHAR(40),       /* Должность */
  SALARY    DECIMAL(8, 2),     /* Оклад */
  NET_SALARY AS (SALARY * .87),
  CONSTRAINT PK_STAFF PRIMARY KEY (COD),
  CONSTRAINT FK1_STAFF
    FOREIGN KEY (CODEPEOPLE) REFERENCES PEOPLE (COD)
    ON DELETE CASCADE,
  CONSTRAINT FK2_STAFF
    FOREIGN KEY (CODORG) REFERENCES ORGANIZATION (COD)
    ON DELETE CASCADE
);
```

В таблице у нас записано несколько строк. Пусть теперь руководитель организации с кодом 11 решил увеличить оклады своим сотрудникам (хорошо они работали последний год). При этом он решил мужчинам увеличить оклады на 15%, а женщинам лишь на 9%. Вот такой несправедливый человек. Для внесения изменений в таблицу следует выполнить операторы, как показано в примере 7.5.

Пример 7.5. Изменение окладов сотрудников в таблице персонала STAFF

```
USE BestDatabase;
GO
UPDATE STAFF SET SALARY = SALARY * 1.15
    WHERE ((SELECT SEX FROM PEOPLE
        WHERE PEOPLE.COD = STAFF.CODPEOPLE) = '0') AND
        CODORG = 11;
UPDATE STAFF SET SALARY = SALARY * 1.09
    WHERE ((SELECT SEX FROM PEOPLE
        WHERE PEOPLE.COD = STAFF.CODPEOPLE) = '1') AND
        CODORG = 11;
GO
```

Первый оператор UPDATE вносит изменения в строки, относящиеся к мужчинам. Второй — к женщинам. Для выборки множества изменяемых строк используется предложение WHERE, задающее условие отбора необходимых строк таблицы. В этом условии используется оператор SELECT, отбирающий записи с нужным значением поля человека (сотрудника) и с кодом требуемой организации.

7.4. Удаление данных (оператор *DELETE*)

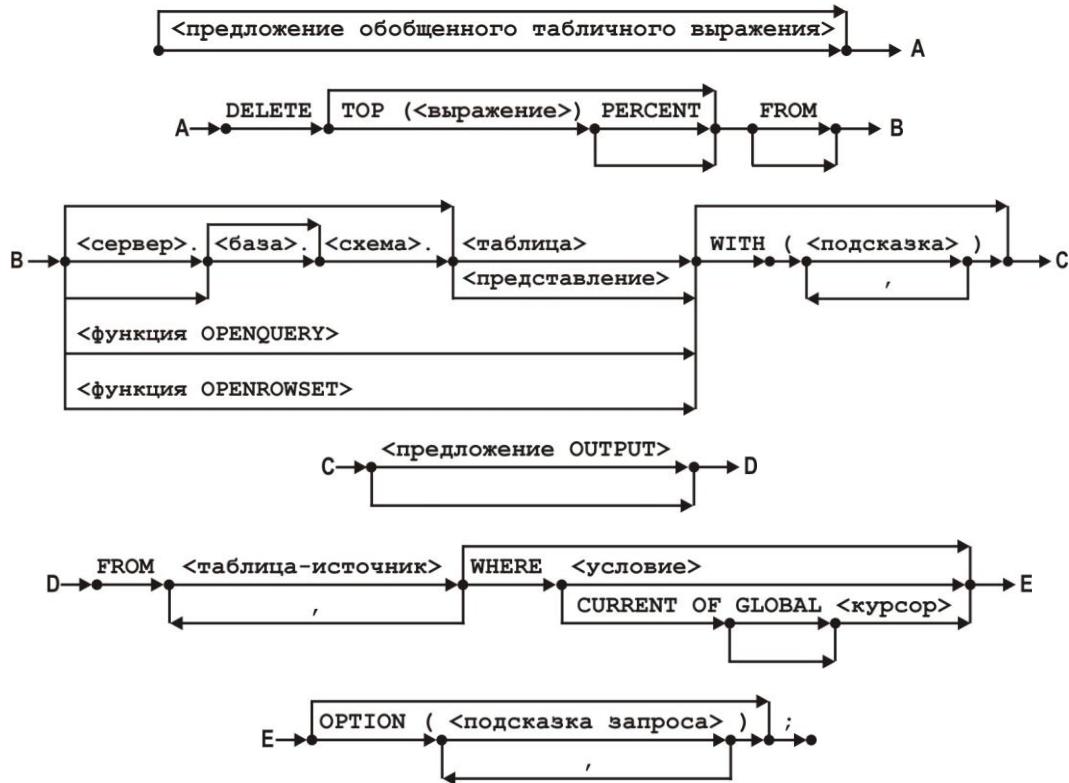
Оператор *DELETE* позволяет удалить одну или более строк одной таблицы или представления.

Синтаксис оператора представлен в листинге 7.4 и графе 7.6.

Листинг 7.4. Синтаксис оператора *DELETE*

```
[ <предложение обобщенного табличного выражения> ]
DELETE [ TOP (<выражение>) [ PERCENT ] ] [ FROM ]
{ [ <сервер>. ] [ [ <база данных>. ] <схема>. ]
  { <таблица> | <представление> }
  | { <функция OPENQUERY> | <функция OPENROWSET> }
}
[ WITH (<подсказка таблицы> [, <подсказка таблицы>...]) ]
[ <предложение OUTPUT> ]
[ FROM (<таблица-источник> [, <таблица-источник>]...) ]
```

```
[ WHERE { <условие>
    | CURRENT OF [ GLOBAL ] <имя курсора>
    }
]
[ OPTION (<подсказка запроса> [, <подсказка запроса>]...) ] ;
```



Граф 7.6. Синтаксис оператора DELETE

Надо полагать, никаких вопросов этот синтаксис у нас с вами не вызывает. Все предложения нам уже хорошо известны по предыдущим операторам.

Первым может быть задано обобщенное табличное выражение.

В предложении **TOP** указывается количество либо процент строк, удаляемых этим оператором.

Затем задается таблица, для которой выполняется удаление. Это таблица или представление текущего экземпляра сервера или таблица на связанном сервере.

Здесь же могут использоваться подсказки таблицы, заданные в предложении **WITH**.

Предложение **OUTPUT** возвращает измененные строки таблицы.

В предложении **FROM** задается список таблиц, использование которых позволяет ограничить количество удаляемых строк таблицы.

В предложении WHERE задаются условия выбора удаляемых строк таблицы. Здесь можно задать условие или указать, что удаляется одна строка, на которую ссылается курсор (вариант CURRENT OF). Курсор может быть локальным или глобальным (ключевое слово GLOBAL).

Условие выборки данных может быть очень сложным. Подробно условия мы рассмотрим в следующей главе 8, где будем заниматься оператором SELECT.

Необязательное предложение OPTION содержит подсказки запроса.

Теперь продолжим рассматривать ситуацию, сложившуюся в известной нам с вами организации, где руководитель решил дифференцированно повысить зарплаты мужчинам и женщинам.

Через некоторое время после повышения зарплат сотрудникам руководитель решил распустить весь персонал. Вначале он уволил женщин (мы с вами помним, что он не политкорректен), затем мужчин. Удаление соответствующих строк из таблицы персонала показано в примере 7.6.

Пример 7.6. Удаление сотрудников из таблицы персонала STAFF

```
USE BestDatabase;
GO
DELETE FROM STAFF
WHERE ((SELECT SEX FROM PEOPLE
        WHERE PEOPLE.COD = STAFF.CODPEOPLE) = '0') AND
      CODORG = 11;
DELETE FROM STAFF
WHERE ((SELECT SEX FROM PEOPLE
        WHERE PEOPLE.COD = STAFF.CODPEOPLE) = '1') AND
      CODORG = 11;
GO
```

Удаляются все сотрудники организации. Все параметры и условия в операторах нам с вами понятны. Что было дальше с этой организацией, я не знаю.

7.5. Удаление строк таблицы (оператор TRUNCATE TABLE)

Если в операторе DELETE не задано условие в предложении WHERE, то этот оператор удаляет все строки таблицы. Удалить все строки таблицы можно при использовании оператора TRUNCATE TABLE. При этом удаление происходит много быстрее.

Синтаксис оператора представлен в листинге 7.5 и графе 7.7.

Листинг 7.5. Синтаксис оператора TRUNCATE TABLE

```
TRUNCATE TABLE
[ [<имя базы данных>.]<имя схемы>.]<имя таблицы> ;
```



Граф 7.7. Синтаксис оператора TRUNCATE TABLE

Здесь есть только одна неприятность. Если на таблицу, которую вы хотите очистить быстрым способом, есть ссылки внешних ключей других таблиц, то попытка удаления будет завершена с ошибкой. Удаление не пройдет, даже если подчиненная таблица не имеет вообще никаких строк.

7.6. Добавление, изменение или удаление строк таблицы (оператор *MERGE*)

Этот довольно сложный и интересный оператор позволяет выполнить несколько вариантов действий: добавление, изменение и удаление данных таблицы в зависимости от условий, получаемых от взаимодействия с другой, исходной, таблицей.

Вначале синтаксис оператора, который представлен в листинге 7.6 и в графическом виде (графы 7.8—7.12). Синтаксис дается в несколько упрощенном варианте.

Листинг 7.6. Синтаксис оператора *MERGE*

```

[ <предложение обобщенного табличного выражения> ]
MERGE [ TOP (<выражение>) [ PERCENT ] ] [ INTO ]
[ [<имя базы>.]<имя схемы>.]<имя таблицы>
[ WITH (<подсказка слияния>) ] [ [ AS ] <псевдоним таблицы> ]
USING <исходная таблица> ON <условие слияния>
[ WHEN MATCHED [ AND <условие> ]
  THEN <слияние соответствия> [ <слияние соответствия> ] ... ]
[ WHEN NOT MATCHED [ BY TARGET ] [ AND <условие> ]
  THEN <слияние несоответствия> [ <слияние несоответствия> ] ... ]
[ WHEN NOT MATCHED BY SOURCE [ AND <условие> ]
  THEN <слияние соответствия> [ <слияние соответствия> ] ... ]
[ <предложение OUTPUT> ]
[ OPTION (<подсказка запроса> [, <подсказка запроса>]...) ] ;
<исходная таблица> :=
{ <имя таблицы> [ [ AS ] <псевдоним таблицы> ]
  [ WITH (<подсказка таблицы> [, <подсказка таблицы>]...) ] }
| <функция, определенная пользователем>
}

<слияние соответствия> :=
{ UPDATE SET <изменяемое значение> [, <изменяемое значение>] ... }
  
```

```
| DELETE
}
```

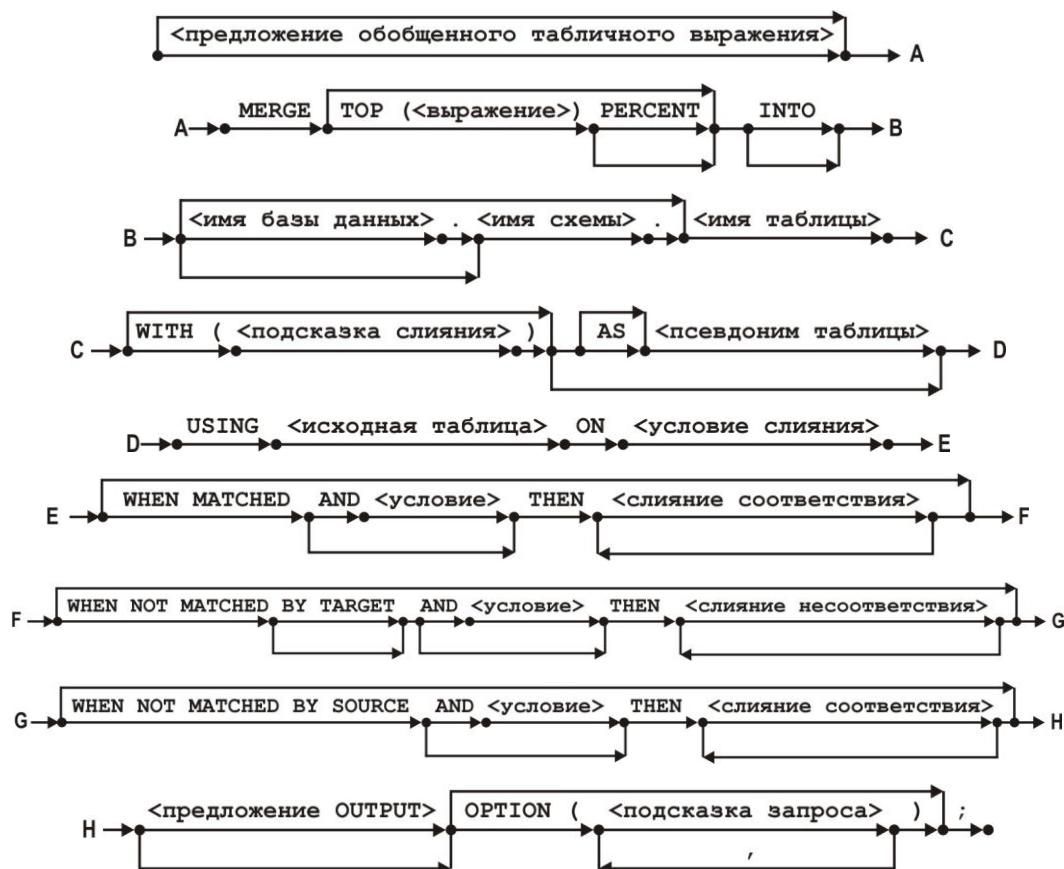
<изменяемое значение> ::=
{ <имя столбца> = { DEFAULT | NULL | <выражение> }
| <имя столбца>.WRITE(<выражение>, @<смещение>, @<размер>)
| @<переменная> [= <столбец>] <операция> <выражение>
}

<операция> ::= += | -= | *= | /= | %= | &= | ^= | |= | =

<слияние несоответствия> ::=

```
INSERT [ ( <список столбцов> ) ]
{ VALUES ( { DEFAULT | NULL | <выражение> }
[ , { DEFAULT | NULL | <выражение> } ]...
[ , ( { DEFAULT | NULL | <выражение> }
[ , { DEFAULT | NULL | <выражение> } ]...) ]...
| DEFAULT VALUES
}
```

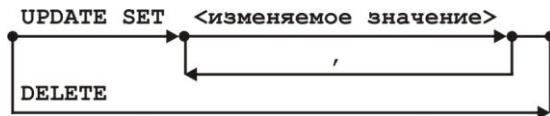
}



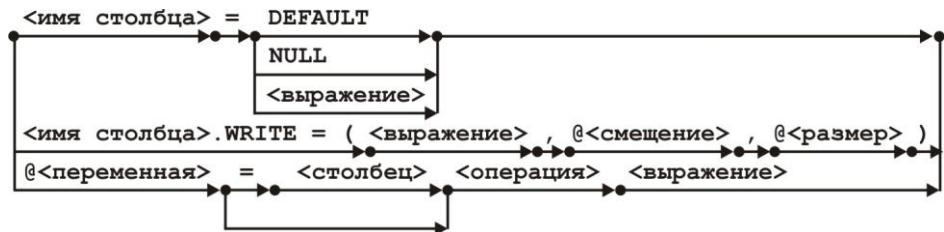
Граф 7.8. Синтаксис оператора MERGE



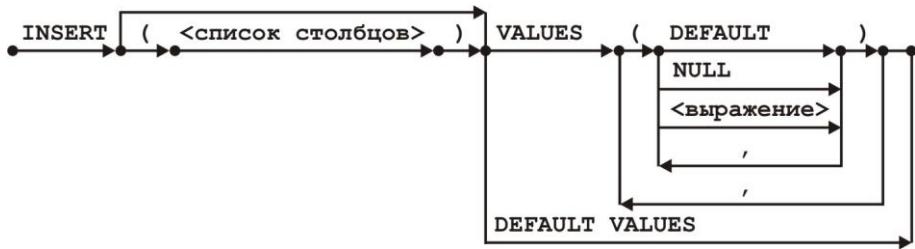
Граф 7.9. Синтаксис исходной таблицы



Граф 7.10. Синтаксис слияния соответствия



Граф 7.11. Синтаксис изменяемого значения



Граф 7.12. Синтаксис слияния несоответствия

Как и в других операторах, в начале оператора может следовать описание обобщенного табличного выражения.

После ключевого слова `TOP` можно указать, что изменениям подвергается только указанное количество строк исходной таблицы.

После необязательного ключевого слова `INTO` задается целевая таблица, к строкам которой будут применяться операции добавления, изменения или удаления.

После ключевого слова `USING` указывается исходная таблица, таблица, на основании данных которой будут выполняться изменения в целевой таблице. Этот процесс еще называется *соединением таблиц*. В предложении `ON` задается условие, при котором будут выполняться действия оператора.

Сами действия по изменению данных в целевой таблице задаются в предложениях, определяющих основное условие выполнения изменений: WHEN MATCHED, WHEN NOT

MATCHED BY TARGET и WHEN NOT MATCHED BY SOURCE. Дополнительно к основному условию можно при помощи операции конъюнкции (логическое И) присоединить еще одно условие. Условие после ключевого слова AND называется *дополнительным условием*. Могут быть заданы любые из этих предложений, но не меньше одного.

Предложение WHEN MATCHED задает действия, которые должны выполняться для строк целевой таблицы, которые соответствуют условиям, заданным в предложении ON в конъюнкции с дополнительным условием, если оно задано. Выполняемым действием может быть изменение данных в (единственной!) строке целевой таблицы или удаление группы строк целевой таблицы, соответствующих условию.

В операторе может присутствовать два предложения WHEN MATCHED. Второе будет выполняться только в том случае, если не было выполнено первое. В случае двух предложений одно должно содержать вариант UPDATE, а другое DELETE.

Предложение WHEN NOT MATCHED BY TARGET задает добавление строк в целевую таблицу из исходной таблицы, если эти строки не удовлетворяют условию поиска строк целевой таблицы, но удовлетворяют дополнительному условию.

Предложение WHEN NOT MATCHED BY SOURCE указывает, что все строки, не соответствующие возвращенным строкам исходной таблицы, но удовлетворяющие дополнительному условию, будут изменяться или удаляться.

Предложение OUTPUT позволяет помещать столбцы добавляемых, изменяемых или удаляемых строк целевой таблицы в любую таблицу или в локальные переменные.

В предложении OPTION можно задать подсказки запроса, изменяющие процедуру выборки данных, чего, как помните, я вам не советую делать.

* * *

Теперь давайте рассмотрим пример, где проиллюстрируем использование рассмотренных операторов. Заодно и вспомним сведения из предыдущих глав.

Задача в том, чтобы создать базу данных DoubleDatabase, которая будет копировать некоторые сведения из основной базы данных — BestDatabase. Эта вторая база данных будет путешествовать по всему миру на ноутбуке одного активного коммивояжера. В базу им будут вноситься изменения в соответствии с полученной в поездках информацией. После его приезда на родину нужно выполнить синхронизацию обеих баз данных.

Вначале создадим дублирующую базу данных и в ней три таблицы (пример 7.7).

Пример 7.7. Создание дублирующей базы данных

```
USE master;
GO
IF DB_ID('DoubleDatabase') IS NOT NULL
    DROP DATABASE DoubleDatabase;
GO
```

```

CREATE DATABASE DoubleDatabase
ON PRIMARY (NAME = DoubleDatabase_dat,
    FILENAME = 'D:\DoubleDatabase\Double.mdf')
LOG ON (NAME = DoubleDatabase_log,
    FILENAME = 'D:\DoubleDatabase\Double.ldf');
GO

USE DoubleDatabase;
GO

    /*** Справочник видов деятельности REFACTIV ***/
CREATE TABLE REFACTIV
( COD      CHAR(4) NOT NULL,      /* Код вида деятельности */
  NAME     VARCHAR(110),           /* Наименование вида деятельности */
  CONSTRAINT PK_REFACTIV
    PRIMARY KEY (COD)
);

    /*** Список организаций ***/
CREATE TABLE ORGANIZATION
( COD      INTEGER NOT NULL,      /* Код организации */
  NAME     VARCHAR(60),           /* Название организации */
  CONSTRAINT PK_ORGANIZATION PRIMARY KEY (COD)
);

    /*** Виды деятельности организации ***/
CREATE TABLE ORGACTIV
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL,      /* Код — первичный ключ */
  CODACT      CHAR(4),           /* Код вида деятельности */
  CODORG      INTEGER,           /* Код организации */
  TEXT        VARCHAR(110),       /* Описание вида деятельности */
  CONSTRAINT PK_ORGACTIV
    PRIMARY KEY (COD),          /* Первичный ключ */
  CONSTRAINT FK1_ORGACTIV
    FOREIGN KEY (CODACT) REFERENCES REFACTIV (COD)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  CONSTRAINT FK2_ORGACTIV
    FOREIGN KEY (CODORG) REFERENCES ORGANIZATION (COD)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
GO

```

Добавим в основную базу данных несколько видов деятельности для одной организации и скопируем необходимые данные в дублирующую базу (пример 7.8).

Пример 7.8. Копирование данных

```
USE BestDatabase;
GO
SET NOCOUNT ON;

DELETE FROM ORGACTIV;
GO

INSERT INTO ORGACTIV (CODACT, CODORG)
VALUES ('2010', 8),
       ('2020', 8),
       ('2021', 8),
       ('2022', 8),
       ('2023', 8),
       ('2029', 8);
GO

USE DoubleDatabase;
GO
DELETE FROM REFACTIV;
GO
INSERT INTO REFACTIV (COD, NAME)
    SELECT COD, NAME FROM BestDatabase.dbo.REFACTIV;

DELETE FROM ORGANIZATION;
GO
INSERT INTO ORGANIZATION (COD, NAME)
    SELECT COD, NAME FROM BestDatabase.dbo.ORGANIZATION;
INSERT INTO ORGACTIV (CODACT, CODORG)
    SELECT CODACT, CODORG FROM BestDatabase.dbo.ORGACTIV;
GO
SET NOCOUNT OFF;
```

Обратите внимание, как мы добавляем все строки видов деятельности организации в дублирующую базу данных:

```
INSERT INTO ORGACTIV (CODACT, CODORG)
    SELECT CODACT, CODORG FROM BestDatabase.dbo.ORGACTIV;
```

Можно отобразить этот список. Результатом будет:

CODACT	CODORG	TEXT
2010	8	NULL
2020	8	NULL
2021	8	NULL
2022	8	NULL

```
2023     8      NULL
2029     8      NULL
```

(6 row(s) affected)

Точно такие же значения на текущий момент будут и в основной базе данных.

Теперь наш коммивояжер отправляется по странам и континентам со своим ноутбуком. Он вносит следующие изменения в список видов деятельности организаций.

```
USE DoubleDatabase;
GO
INSERT INTO ORGACTIV (CODACT, CODORG)
VALUES ('2010', 9), ('2029', 9), ('1812', 9);
DELETE FROM ORGACTIV WHERE CODACT = '2010' AND CODORG = 8;
DELETE FROM ORGACTIV WHERE CODACT = '2029' AND CODORG = 8;
GO
```

Теперь виды деятельности организаций в дублирующей базе выглядят следующим образом:

CODACT	CODORG	TEXT
2020	8	NULL
2021	8	NULL
2022	8	NULL
2023	8	NULL
2010	9	NULL
2029	9	NULL
1812	9	NULL

(7 row(s) affected)

По приезде "в родные пенаты" коммивояжер выполняет синхронизацию основной и дублирующей баз данных, используя оператор `MERGE` (пример 7.9).

Пример 7.9. Синхронизация баз данных

```
USE BestDatabase;
GO
MERGE ORGACTIV USING [DoubleDatabase].[dbo].[ORGACTIV] AS FIRSTTAB
  ON FIRSTTAB.[CODACT] = ORGACTIV.[CODACT]
    AND FIRSTTAB.[CODORG] = ORGACTIV.[CODORG]
WHEN NOT MATCHED BY TARGET THEN
  INSERT (CODACT, CODORG)
    VALUES (FIRSTTAB.[CODACT], FIRSTTAB.[CODORG])
WHEN NOT MATCHED BY SOURCE THEN
  DELETE;
GO
```

Первым делом в операторе указывается, что работа будет выполняться с таблицей ORGACTIV базы данных BestDatabase. Исходная таблица находится в другой базе данных. Эта таблица задается после ключевого слова USING. Для таблицы указывается имя базы данных и имя схемы по умолчанию. Здесь же таблице присваивается псевдоним FIRSTTAB, чтобы далее не возникли двусмыслинности при обращении к столбцам двух разных таблиц.

Основное условие выполнения оператора задается после ключевого слова ON. Условие содержит требование к равенству двух ключевых столбцов в исходной и целевой таблице.

Если в целевой таблице отсутствует соответствующая строка, то выполняются действия, заданные в предложении WHEN NOT MATCHED BY TARGET, добавляется недостающая строка. Если в целевой таблице присутствуют строки, удаленные в исходной таблице, то выполняется удаление таких строк, как указано в предложении WHEN NOT MATCHED BY SOURCE.

Отобразите строки таблицы ORGACTIV в основной базе данных. Мы видим, что данные в обеих базах данных совпадают.

Что будет дальше?

В следующей главе мы, наконец, начнем рассматривать важнейший и наиболее сложный оператор выборки данных SELECT.



ГЛАВА 8

Выборка данных

- ◆ Оператор `SELECT`
- ◆ Оператор `UNION`
- ◆ Операторы `EXCEPT` и `INTERSECT`
- ◆ Примеры выборки данных
- ◆ Операторы `UNION`, `EXCEPT`, `INTERSECT`

В этой главе подробно рассмотрим выборку данных, оператор `SELECT` и, так сказать, вспомогательные операторы `UNION`, `EXCEPT`, `INTERSECT`.

Мы уже многократно использовали оператор `SELECT` для получения различных данных из баз данных. Настала пора подробно рассмотреть его синтаксис и примеры использования.

8.1. Оператор `SELECT`

В операторе `SELECT` так же, как и во многих других операторах DML, можно использовать обобщенное табличное выражение СТЕ. Начальный синтаксис оператора представлен в листинге 8.1 и в графической форме (граф 8.1).

Листинг 8.1. Синтаксис оператора `SELECT`

```
[ <предложение обобщенного табличного выражения> ]
SELECT [ ALL | DISTINCT ] [ TOP (<выражение>) [ PERCENT ] [ WITH TIES ] ] 
<элемент выбора> [, <элемент выбора>]... [ INTO <новая таблица> ]
[ FROM <исходная таблица> [, <исходная таблица>]... ]
[ WHERE <условие выборки> ]
[ GROUP BY <условие группировки> ]
[ HAVING <условие поиска> ]
[ ORDER BY <список упорядочения> ] [ <предложение FOR> ]
[ OPTION ( <подсказка запроса> [, <подсказка запроса>]... ) ] ;
```



Граф 8.1. Синтаксис оператора SELECT

Про обобщенное табличное выражение мы говорили в предыдущей главе 7. Там же описан его синтаксис. Выражение задает временный именованный набор данных. К этому набору данных можно обращаться в любом предложении оператора SELECT. После ключевого слова SELECT можно указать ALL или DISTINCT. Ключевое слово ALL (значение по умолчанию) указывает, что в результирующем наборе данных будут включены все выбранные строки, в том числе и те, которые содержат одинаковые данные. Если задано ключевое слово DISTINCT, то в набор данных будут помещаться только уникальные строки. Иными словами, в наборе данных не будет двух строк, все столбцы которых имеют одинаковые значения. При этом (внимание!) значения NULL считаются равными.

В предложении TOP указывается количество либо процент (ключевое слово PERCENT) выбираемых строк.

Далее идет список выбора, который содержит список элементов, разделенных запятыми. Синтаксис конструкции <элемент выбора> представлен в листинге 8.2 и графике 8.2.

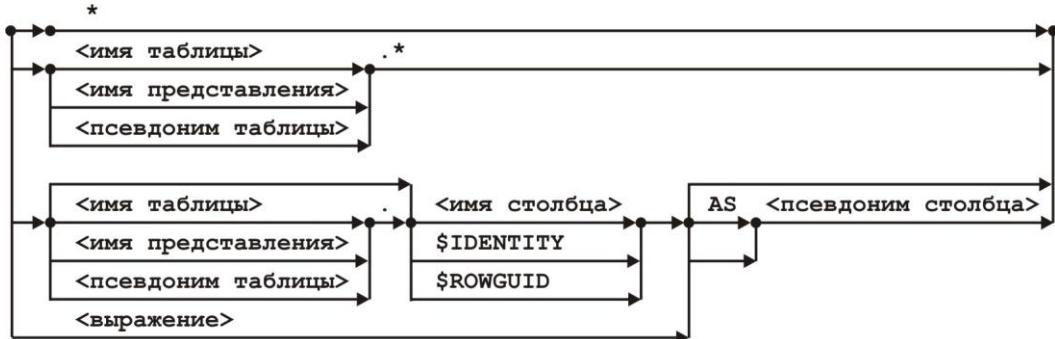
Листинг 8.2. Синтаксис элемента выбора

```
<элемент выбора> ::==
{ *
| { <имя таблицы> | <имя представления> | <псевдоним таблицы> }.*
```

```

| { [ { <имя таблицы> | <имя представления> | <псевдоним таблицы> } . ]
  { <имя столбца> | $IDENTITY | $ROWGUID }
  | <выражение> } [ [ AS ] <псевдоним столбца> ]
}

```



Граф 8.2. Синтаксис элемента выбора

Если список содержит символ *, это означает, что должны выбираться все столбцы из таблицы или представления, заданного в предложении FROM.

Если перед символом * стоит отделенное точкой имя таблицы или представления, то будут выбраны все столбцы указанной таблицы, представления. Таблица или представление могут быть заданы своим псевдонимом, который указывается в предложении FROM. Рекомендуется не использовать символ *, а явно задавать имена выбираемых столбцов.

В списке выбора можно задавать список столбцов. Если столбец с указанным именем присутствует в более чем одной таблице, представлении, принимающих участие в операторе выборки данных, то во избежание двусмысленности (выполнение операции будет завершено с ошибкой) перед именем столбца нужно через точку указать имя таблицы, представления или псевдоним.

Если задано ключевое слово \$IDENTITY, то в список выбора помещается столбец таблицы, представления со свойством IDENTITY. Ключевое слово \$ROWGUID означает, что в список выбора включается столбец, заданный со свойством ROWGUIDCOL.

После необязательного ключевого слова AS можно указать псевдоним столбца. Этот псевдоним будет использован в качестве заголовка отображаемого результирующего набора данных при выполнении запроса в командной строке или в Management Studio.

В списке выбора помимо столбцов могут присутствовать выражения. Это константы, функции, вложенные запросы, т. е. любое допустимое в языке Transact-SQL выражение, возвращающее одно значение.

Если после списка выбора присутствует предложение INTO, то в файловой группе по умолчанию создается новая таблица и в нее помещаются выбранные данные. Структура создаваемой таблицы, состав и типы данных элементов определяются

в соответствии с составом элементов в списке выбора. Таблица не может быть секционированной. В новую таблицу также не перемещаются существующие для исходной таблицы триггеры, ограничения и индексы.

В необязательном предложении `FROM` указываются объекты, из которых выбираются данные. Если предложение `FROM` не задано, то список выбора может содержать лишь константы, выражения. Такой оператор вернет только одну строку.

Несколько упрощенный синтаксис конструкции `<исходная таблица>` в предложении `FROM` показан в листинге 8.3 и графах 8.3—8.5.

Листинг 8.3. Синтаксис исходной таблицы

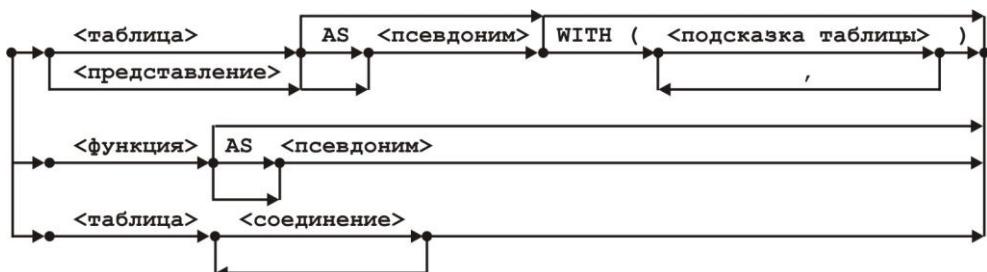
```

<исходная таблица> ::==
{ <таблица> | <представление> } [ [ AS ] <псевдоним> ]
    [ WITH (<подсказка таблицы> [, <подсказка таблицы>]...) ]
| <функция> [ [ AS ] <псевдоним> ]
| <таблица> <соединение> [ <соединение> ]...
}

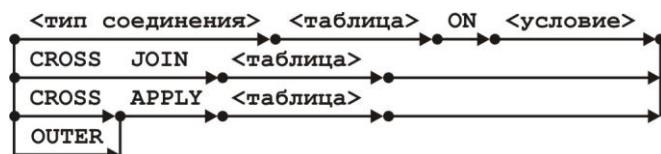
<соединение> ::==
{ <тип соединения> <таблица> ON <условие>
| CROSS JOIN <таблица>
| { CROSS | OUTER } APPLY <таблица>
}

<тип соединения> ::==
{ [ INNER ] JOIN
| { LEFT | RIGHT | FULL } [ OUTER ] } [ <подсказка соединения> ] JOIN
}

```



Граф 8.3. Синтаксис исходной таблицы



Граф 8.4. Синтаксис соединения



Граф 8.5. Синтаксис типа соединения

В предложении `FROM` исходная таблица может быть задана таблицей или представлением. Для задания этого объекта можно использовать все варианты, рассмотренные нами в предыдущей главе 7. Объект может находиться в том же или в другом экземпляре сервера, и вне экземпляра сервера SQL. После необязательного ключевого слова `AS` может быть указан псевдоним таблицы (представления), который возможно применять при обращении к данному объекту. После ключевого слова `WITH` в скобках указываются используемые в выборке данных подсказки. Подсказки включают варианты оптимизации запроса и виды применяемых блокировок. Эти указания будут использованы оптимизатором запросов.

Вместо таблицы или представления может быть указана функция с ее псевдонимом. Это может быть функция наборов строк или определенная пользователем функция. Функции должны возвращать объект, на который можно ссылаться как на таблицу. Функции набора строк — это функции `OPENDATASOURCE`, `OPENQUERY`, `OPENROWSET`, `OPENXML`.

Замечательной возможностью оператора `SELECT` является *соединение* (*join*). Когда мы выполняем нормализацию таблиц, то часто таблица разделяется на две или более. Данные из нескольких таблиц могут быть вновь объединены в процессе выборки при помощи средств соединения. В предложении соединения после ключевого слова `ON` задается условие соединения.

Существует несколько видов соединения.

- ◆ *Внутреннее соединение* (`INNER JOIN`) — оператор возвращает только все совпадающие пары строк.
- ◆ *Внешнее соединение* (`OUTER JOIN`) — бывает левым (`LEFT`), правым (`RIGHT`) и полным (`FULL`):
 - *левое внешнее соединение* (`LEFT OUTER JOIN`) — возвращает все строки левой (основной в операторе) таблицы, куда добавляются соответствующие условия соединения значения столбцов правой таблицы;
 - *правое внешнее соединение* (`RIGHT OUTER JOIN`) — является зеркальным отражением левого внешнего. В результирующий набор помещаются все строки правой таблицы с добавлением значений указанных столбцов левой таблицы;
 - *полное внешнее соединение* (`FULL OUTER JOIN`) — возвращает все строки левой и правой таблицы.
- ◆ *Перекрестное соединение* (`CROSS JOIN`) — в варианте `CROSS JOIN` возвращается перекрестное соединение двух таблиц, т. е. декартово произведение всех строк обеих таблиц. В этом варианте не используется условие соединения (ключевое слово `ON`).

Вариант APPLY с ключевыми словами CROSS и OUTER выполняет соединение левой таблицы с данными из правой таблицы. Для правой таблицы, как правило, используется функция с табличным значением, которой в качестве параметра передается значение столбца из левой таблицы. Функция использует полученный аргумент для выделения строки из правой таблицы. При задании ключевого слова CROSS будут возвращены строки, для которых найдено соответствие в правой таблице, если задано ключевое слово OUTER, то возвращаются строки, для которых соответствие не было найдено.

В необязательном предложении WHERE указывается условие, на основании которого в результирующий набор выбираются данные. Если предложение не задано, то возвращаются все строки. Синтаксис конструкции <условие выборки>, которую также называют и условием поиска, представлен в листинге 8.4 и графах 8.6—8.7.

Листинг 8.4. Синтаксис условия выборки

```

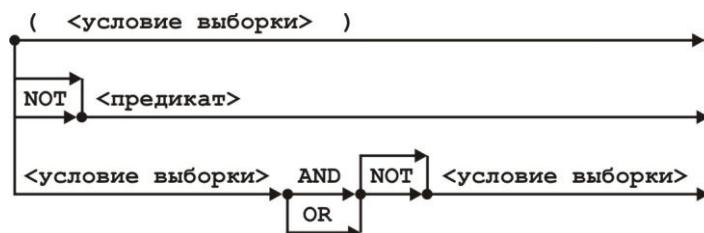
<условие выборки> ::=

{ (<условие выборки>)
| [ NOT ] <предикат>
| <условие выборки> { AND | OR } [ NOT ] <условие выборки>
}
}

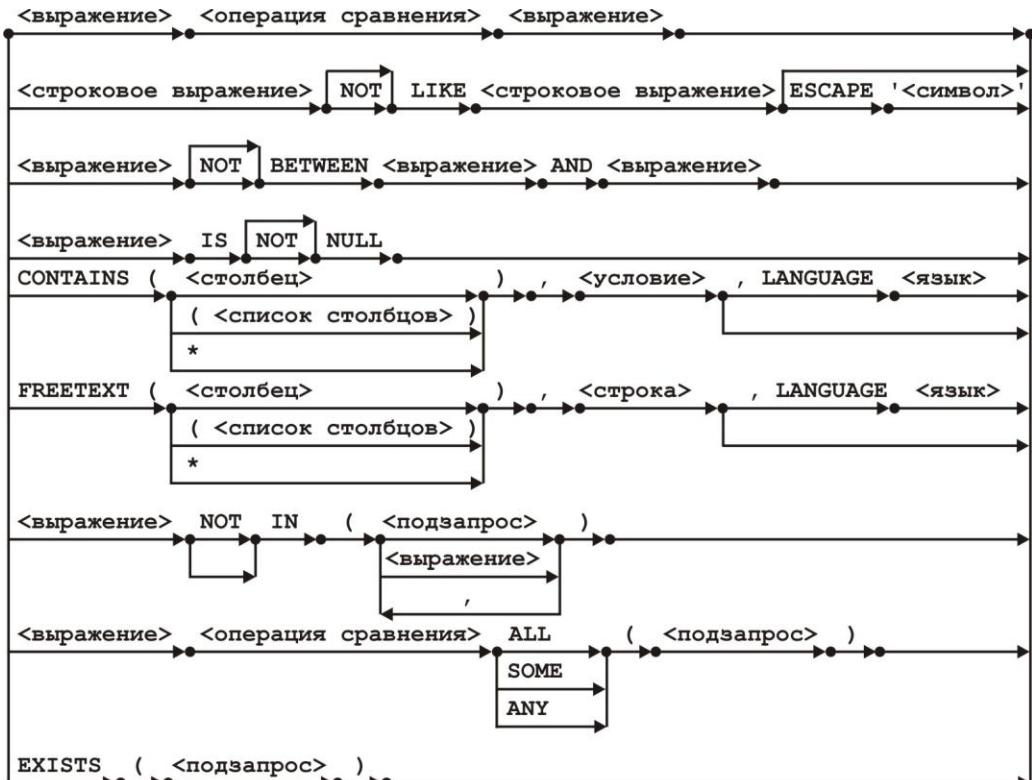
<предикат> ::=

{ <выражение> <операция сравнения> <выражение>
| <строковое выражение> [ NOT ] LIKE <строковое выражение>
  [ ESCAPE '<символ>' ]
| <выражение> [ NOT ] BETWEEN <выражение> AND <выражение>
| <выражение> IS [ NOT ] NULL
| CONTAINS ( { <столбец> | <список столбцов> | * } )
  , <условие CONTAINS> [ , LANGUAGE <язык> ]
| FREETEXT ( { <столбец> | <список столбцов> | * } )
  , <строка FREETEXT> [ , LANGUAGE <язык> ]
| <выражение> [ NOT ] IN ( { <подзапрос> | <выражение> } )
  [ , { <подзапрос> | <выражение> } ] ...
| <выражение> <операция сравнения>
  { ALL | SOME | ANY } ( <подзапрос> )
| EXISTS ( <подзапрос> )
}

```



Граф 8.6. Синтаксис условия выборки



Граф 8.7. Синтаксис предиката

Оператор `SELECT` выбирает строки из таблицы (таблиц) или из представления. Строки также могут быть получены при вызове функции, обращающейся к таблице (таблицам) или к представлению базы данных. Условие выборки — это конструкция, выражение, возвращающее логическое значение "истина", "ложь" или `NULL` для конкретного столбца или группы столбцов основной таблицы запроса для каждой исходной строки. На основании этого предложения в результирующий набор попадают, как правило, не все строки, а только та часть, которая соответствует условию выборки, т. е. для которых было получено значение "истина".

Предикат — это логическое выражение, довольно сложное в SQL, которое для каждой исходной строки возвращает значение "истина", "ложь" или неизвестное значение `NULL`. Здесь используются такие конструкции, как сравнение, конструкции `LIKE`, `BETWEEN` и некоторые другие.

В условии выборки естественным образом присутствуют классические логические операции отрицания (`NOT`), конъюнкции (`AND`) и дизъюнкции (`OR`). Конъюнкция дает значение "истина", если оба операнда истинны. Дизъюнкция дает ложь, если оба операнда ложны.

В табл. 8.1—8.3 приводятся таблицы истинности для логических операций отрицания, дизъюнкции и конъюнкции при использовании в качестве operandов как значений "истина" (`True`) и "ложь" (`False`), так и значения `NULL`.

Таблица 8.1. Таблица истинности для отрицания

Операнд	NOT Операнд
False	True
True	0
NULL	NULL

Таблица 8.2. Таблица истинности для дизъюнкции

Операнд 1	Операнд 2	Операнд 1 OR Операнд 2
False	False	False
False	True	True
True	False	True
True	True	True
True	NULL	True
False	NULL	NULL
NULL	True	True
NULL	False	NULL
NULL	NULL	NULL

Таблица 8.3. Таблица истинности для конъюнкции

Операнд 1	Операнд 2	Операнд 1 AND Операнд 2
False	False	False
False	True	False
True	False	False
True	True	True
True	NULL	NULL
False	NULL	False
NULL	True	NULL
NULL	False	False
NULL	NULL	NULL

Здесь поведение системы полностью соответствуют правилам логики высказываний.

Первой конструкцией в синтаксисе предиката является <выражение> <операция сравнения> <выражение>.

Выражением может быть любое допустимое выражение языка SQL. Чаще всего это имя столбца таблицы. В правой части предложения может использоваться оператор SELECT, который возвращает ровно одно значение. Примеры мы вскоре рассмотрим.

Операциями сравнения являются операции, которые уже были описаны в главе 4:

- ◆ = — равно;
- ◆ !=, <> — не равно;
- ◆ < — меньше;
- ◆ > — больше;
- ◆ <=, !> — меньше или равно, не больше;
- ◆ >=, !< — больше или равно, не меньше.

Конструкция LIKE позволяет проверить совпадение строки с указанным шаблоном или отсутствие такого совпадения при указании ключевого слова NOT. В самом шаблоне могут присутствовать шаблонные символы. Используемые шаблонные символы перечислены в табл. 8.4.

Таблица 8.4. Шаблонные символы

Символ	Значение
%	Произвольная строка, содержащая ноль или любое количество символов
_	Любой один символ
[]	Задание диапазона допустимых значений. Подходит любой символ из указанного диапазона. Символы диапазона указываются через дефис, например, [a-z]. Это означает, что допустимыми являются все буквы латинского алфавита
[^]	Задание диапазона недопустимых значений. Исключается любой символ из указанного диапазона. Указание [^a-z] означает, что все буквы латинского алфавита являются недопустимыми. В варианте [^abc] недопустимыми будут только перечисленные символы — a, b и c.

Ключевое слово ESCAPE позволяет задать символ, который дает возможность использовать шаблонные символы как обычные символы шаблона, а не как шаблонные символы. Если шаблонный символ нужно в строке использовать как обычный символ, который принимает участие в операции сравнения, то ему должен предшествовать символ, указанный после ключевого слова ESCAPE.

Конструкция BETWEEN возвращает истину, если значение находится в указанном диапазоне (значение меньше или равно второму значению диапазона и больше или равно первому значению) или, наоборот, не находится в этом диапазоне, если задано ключевое слово NOT.

Конструкция IS NULL (IS NOT NULL) выполняет проверку, является ли выражение неизвестным значением.

Конструкция `CONTAINS` позволяет выполнять полнотекстовый поиск в строковых типах данных `CHAR`, `VARCHAR`, `NCHAR`, `NVARCHAR`, `TEXT`, `NTEXT`, `XML`, `VARBINARY`, `VARBINARY(MAX)` полнотекстового индекса.

Полнотекстовый поиск позволяет выполнить очень сложную выборку данных не только на основании точного совпадения символов, но и с довольно сложными вариантами поиска. Здесь используются различные формы одного и того же слова. Язык, на основании которого определяется словообразование, задается после ключевого слова `LANGUAGE`.

Конструкция `FREETEXT` также позволяет выполнять полнотекстовый поиск, но не на основании формы текста (синтаксиса), а на основании содержания (семантики). Подробные описания полнотекстового поиска см. в Books Online.

Конструкция `IN (NOT IN)` задает выбор тех строк, значения столбцов которых находятся в указанном списке или не находятся в списке, при задании ключевого слова `NOT`. Элементами в списке могут быть константы, выражения, возвращающие одно значение, или подзапросы, также возвращающие ровно одно значение.

Следующий вариант в определении предиката содержит выражение, операцию сравнения, одно из ключевых слов `ALL`, `SOME`, `ANY` (их часто называют функциями, и с этим можно согласиться) и подзапрос, возвращающий одно или несколько значений. Этот вариант определяет список строк на основании операции сравнения, полученного списка из подзапроса и заданного ключевого слова.

Функция `ALL` вернет значение "истина", если выражение будет истинным для всех значений, полученных из подзапроса. Подзапрос должен возвращать любое количество значений *одного* столбца. Такие подзапросы называют в литературе *скалярными*.

Функции `SOME` и `ANY` — это синонимы. Подзапрос также является скалярным, т. е. возвращает произвольное количество значений *одного* столбца. Функции вернут значение "истина", если выражение будет истинным хотя бы для одного значения, полученного из подзапроса.

Функция `EXISTS` вернет значение "истина", если результирующий набор подзапроса будет содержать хотя бы одну строку. В этом случае подзапрос не обязательно должен быть скалярным.

ЗАМЕЧАНИЕ

Здесь отсутствует функция `SINGULAR`, существующая в других системах. Эта функция дает значение "истина", если подзапрос возвращает ровно одно значение. Такую функцию можно реализовать при помощи выражения `WHERE COUNT(...) = 1`.

Предложение `GROUP BY` позволяет выполнить группирование найденных строк для получения результата. Часто при этом используется и предложение `HAVING`.

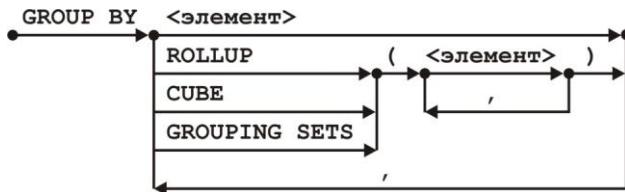
Синтаксис предложения показан в листинге 8.5 и графе 8.8.

Листинг 8.5. Синтаксис предложения `GROUP BY`

<предложение GROUP BY> ::=

`GROUP BY` <элемент GROUP BY> [, <элемент GROUP BY>] ...

```
<элемент GROUP BY> ::==
{ <элемент>
| { ROLLUP | CUBE | GROUPING SETS }
( <элемент> [, <элемент>]... )
```



Граф 8.8. Синтаксис предложения GROUP BY

Часто предложение GROUP BY используется при появлении в списке выбора оператора SELECT агрегатных функций AVG (среднее значение), COUNT (подсчет количества строк), MIN (минимальное значение столбца в группе строк), MAX (максимальное значение столбца в группе строк), SUM (сумма числовых значений). В этом случае предложение GROUP BY должно содержать имя каждого столбца в списке выбора, который не присутствует в агрегатной функции. Имена разделяются запятыми.

В предложении GROUP BY также могут присутствовать функции ROLLUP(), CUBE(), GROUPING SETS(). Они создают довольно сложные строки, содержащие некоторые итоговые данные, и обычно используются для формирования отчетов.

Функция ROLLUP() позволяет создавать иерархические итоговые данные по уровням элементов, указанных в параметрах функции. Функция CUBE() позволяет создавать итоги с использованием перекрестных вычислений, а функция GROUPING SETS() задает несколько вариантов группировки данных в одном запросе.

Примеры мы рассмотрим позже в этой главе.

Предложение HAVING позволяет ограничить количество выбранных строк с учетом предложения GROUP BY. Если же предложение GROUP BY не присутствует в операторе, то HAVING применяется как предложение WHERE для ограничения количества возвращаемых оператором строк.

Предложение ORDER BY задает упорядочение строк результата запроса. Синтаксис предложения представлен в листинге 8.6 и графике 8.9.

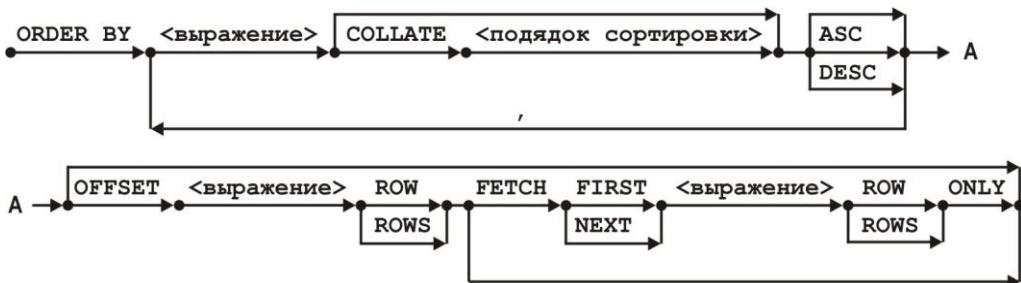
Листинг 8.6. Синтаксис предложения ORDER BY

```
<предложение ORDER BY> ::= ORDER BY <элемент упорядочения>
[ , <элемент упорядочения> ] ... [ <предложение OFFSET> ]
```

<элемент упорядочения> ::=

<выражение> [COLLATE <порядок сортировки>] [ASC | DESC]

<предложение OFFSET> ::=
 OFFSET <выражение> ROW[S]
 [FETCH { FIRST | NEXT } <выражение> ROW[S] ONLY]



Граф 8.9. Синтаксис предложения ORDER BY

Элемент **упорядочения** — это имя или псевдоним столбца либо номер столбца в списке выбора, по которому выполняется упорядочение. Если столбец имеет строковый тип данных, то после ключевого слова COLLATE можно указать используемый для упорядочения порядок сортировки. Упорядоченность может быть по возрастанию значений (ключевое слово ASC, значение по умолчанию) или по убыванию (DESC). Элементы в списке отделяются запятыми.

В предложении OFFSET можно указать, какое число начальных строк не будет помещено в результирующий набор данных. Предложение FETCH указывает количество возвращаемых строк. Ключевые слова FIRST и NEXT являются синонимами и используются лишь для совместимости со стандартом. В этих предложениях выражением является целочисленный литерал или любое выражение, возвращающее целочисленное значение.

Необязательное предложение FOR позволяет задать допустимость обновления данных во время их просмотра, а также указывает, что результат выборки должен возвращаться в виде XML-документа. Подробно рассматривать это предложение мы не станем.

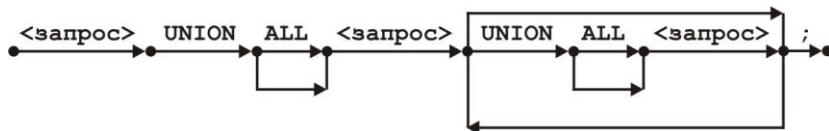
Предложение OPTION задает подсказки запроса, которые в этой книге мы также не рассматриваем.

8.2. Оператор UNION

Оператор UNION позволяет объединить в один выходной набор данных результаты выборки несколькими операторами SELECT. Синтаксис оператора представлен в листинге 8.7 и графике 8.10.

Листинг 8.7. Синтаксис оператора UNION

<оператор UNION> ::= <запрос> UNION [ALL] <запрос>
 [UNION [ALL] <запрос>] ... ;



Граф 8.10. Синтаксис оператора UNION

Результаты запросов в операторе должны полностью соответствовать по структуре. Допускаются некоторые отличия в типах данных столбцов, возвращаемых разными запросами. При этом типы данных должны быть совместимыми, т. е. должна быть возможность приведения типов данных из разных результатов к одному.

Присутствие ключевого слова ALL означает, что будут возвращены все строки запросов. При его отсутствии дублирующие строки будут удалены из результата.

Важный момент: предложение ORDER BY может присутствовать только в последнем операторе SELECT.

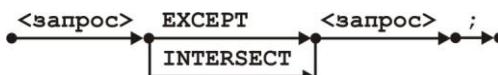
8.3. Операторы EXCEPT, INTERSECT

Эти операторы работают с двумя запросами. Оператор EXCEPT возвращает недублированные строки первого запроса, из которых удаляются строки, присутствующие во втором запросе. Иными словами, это теоретико-множественная операция вычитания из первого множества строк второго множества. Оператор INTERSECT возвращает недублированные строки первого запроса, которые соответствуют строкам, полученным из второго запроса. Это операция пересечения двух множеств.

Синтаксис обеих операций представлен в листинге 8.8 и графике 8.11.

Листинг 8.8. Синтаксис операторов EXCEPT и INTERSECT

```
<оператор EXCEPT, INTERSECT> ::=  
<запрос> { EXCEPT | INTERSECT } <запрос>;
```



Граф 8.11. Синтаксис операторов EXCEPT и INTERSECT

8.4. Примеры выборки данных

Теперь рассмотрим основные возможности оператора SELECT, используя данные нашей базы данных BestDatabase.

8.4.1. Список выбора

В командной строке или в Management Studio выполните следующие операторы для отображения списка людей из таблицы PEOPLE (пример 8.1).

Пример 8.1. Отображение списка людей

```
USE BestDatabase;
GO
SELECT * FROM PEOPLE;
GO
```

Вы получите 112 строк. В списке выбора мы указали символ *. Это означает, что выбираются все столбцы таблицы. При этом список выглядит не очень красиво. Не всем понятные заголовки, и присутствует вычисляемый столбец FULLNAME.

Измените оператор SELECT (пример 8.2).

Пример 8.2. Другой вариант отображения списка людей

```
USE BestDatabase;
GO
SELECT COD AS "Код",
       NAME1 AS "Имя",
       NAME2 AS "Отчество",
       NAME3 AS "Фамилия",
       BIRTHDAY AS "Дата рождения"
  FROM PEOPLE;
GO
```

Вот полученные первые несколько строк:

Код	Имя	Отчество	Фамилия	Дата рождения
1	НАТАЛЬЯ	АЛЕКСАНДРОВНА	ИВАНОВА	1961-12-01
2	АЛЕКСАНДР	АЛЕКСАНДРОВИЧ	ИВАНОВ	1960-01-07
3	ИРИНА	АЛЕКСАНДРОВНА	ИВАНОВА	1991-08-30
4	НИКОЛАЙ	АЛЕКСАНДРОВИЧ	ИВАНОВ	1990-02-21
5	ОЛЕГ	ЮРЬЕВИЧ	ГРАЧЕВ	1959-08-02
6	РОМАН	ОЛЕГОВИЧ	ГРАЧЕВ	1990-08-08

...

Дата рождения выглядит не очень привычно для нас. Чтобы отобразить дату в более приемлемом варианте, нужно использовать функцию CONVERT() (см. главу 4). В описании человека также присутствует код его пола. Значение 0 указывает на мужской пол, значение 1 — на женский. Чтобы отображать соответствующий текст, а не цифры, можно использовать функцию IIF(). Кроме того, код человека является искусственным первичным ключом и мало для чего в этом списке нам будет нужен.

Измените скрипт следующим образом (пример 8.3).

Пример 8.3. Еще один вариант отображения списка людей

```
USE BestDatabase;
GO
SELECT NAME1 AS "Имя",
       NAME2 AS "Отчество",
       NAME3 AS "Фамилия",
       IIF(SEX = '0', 'Мужской', 'Женский') AS "Пол",
       CONVERT(CHAR(12), BIRTHDAY, 104) AS "Дата рождения"
  FROM PEOPLE;
GO
```

Результат будет таким:

Имя	Отчество	Фамилия	Пол	Дата рождения
НАТАЛЬЯ	АЛЕКСАНДРОВНА	ИВАНОВА	Женский	01.12.1961
АЛЕКСАНДР	АЛЕКСАНДРОВИЧ	ИВАНОВ	Мужской	07.01.1960
ИРИНА	АЛЕКСАНДРОВНА	ИВАНОВА	Женский	30.08.1991
НИКОЛАЙ	АЛЕКСАНДРОВИЧ	ИВАНОВ	Мужской	21.02.1990
ОЛЕГ	ЮРЬЕВИЧ	ГРАЧЕВ	Мужской	02.08.1959
РОМАН	ОЛЕГОВИЧ	ГРАЧЕВ	Мужской	08.08.1990
...				

Теперь дата отображается в виде *дд.мм.гггг*. Пол указывается в нормальном текстовом виде. Кстати, для отображения пола человека можно использовать и оператор *CASE*. В нашем случае нужно записать отображение столбца *SEX* следующим образом:

```
CASE SEX WHEN '0' THEN 'Мужской'
          WHEN '1' THEN 'Женский'
END AS "Пол",
```

Это также можно записать и в несколько другом виде:

```
CASE WHEN SEX = '0' THEN 'Мужской'
      WHEN SEX = '1' THEN 'Женский'
END AS "Пол",
```

Чтобы не отображая весь список получить количество строк в таблице, нужно использовать функцию *COUNT()*:

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

Оператор вернет число 112.

Функция *COUNT()* является агрегатной функцией. В качестве примера использования других агрегатных функций можно выполнить оператор, отображающий некоторые сведения об окладах сотрудников таблицы *STAFF*.

```
SELECT AVG(SALARY), MIN(SALARY), MAX(SALARY), SUM(SALARY) from STAFF;
```

Будут получены числа: 21681.250000, 1500.00, 56000.00, 346900.00. Если это сведения о зарплатах в US долларах или в евро, то результат неплохой.

8.4.2. Упорядочение результата (*ORDER BY*)

Данные по людям из примера 8.3 отображаются в произвольном порядке. Во многих случаях нам необходима некоторая упорядоченность. Для этого нужно использовать предложение *ORDER BY*. Выполните операторы примера 8.4.

Пример 8.4. Упорядоченный список людей

```
USE BestDatabase;
GO
SELECT NAME1 AS "Имя",
       NAME2 AS "Отчество",
       NAME3 AS "Фамилия",
       IIF(SEX = '0', 'Мужской', 'Женский') AS "Пол",
       CONVERT(CHAR(12), BIRTHDAY, 104) AS "Дата рождения"
  FROM PEOPLE ORDER BY NAME3, NAME1;
GO
```

Здесь список упорядочивается по фамилии и имени. Тот же результат можно получить при использовании в предложении *ORDER BY* не имен столбцов, а их номеров в списке выбора. Например:

```
ORDER BY 3, 1
```

Однако такой вариант использовать не рекомендуется. Дело в том, что в процессе работы с таблицей вы можете изменять список выбора. В этом случае столбцы могут поменять свои номера.

Столбцы в списке упорядочения также можно указать и при помощи их псевдонимов, заданных в списке выбора. Такой же результат мы получим, записав:

```
ORDER BY "Фамилия", "Имя"
```

В нашем примере сортировка производится в возрастающем порядке по умолчанию. Чтобы изменить порядок на убывающий, нужно после имени столбца добавить ключевое слово *DESC*. Например, для упорядочения списка людей по возрастанию фамилий и по убыванию имен (звучит коряво, но понятно о чём речь) нужно использовать следующее предложение *ORDER BY*:

```
ORDER BY NAME3, NAME1 DESC;
```

Чтобы устраниТЬ повторяющиеся данные в списке, в операторе *SELECT* используется ключевое слово *DISTINCT*. Чтобы отобразить список людей с неповторяющимися фамилиями, нужно выполнить оператор:

```
SELECT DISTINCT NAME3 AS "Фамилия"
  FROM PEOPLE ORDER BY NAME3;
```

Оператор вернет 57 строк. Здесь устраняются однофамильцы.

Ключевое слово DISTINCT рассматривает все пустые значения NULL как равные, одинаковые.

Чтобы посмотреть, как система располагает значения NULL в упорядоченном списке, выполните оператор примера 8.5.

Пример 8.5. Упорядоченный список со значениями NULL

```
USE BestDatabase;
GO
SELECT CODPEOPLE AS "Код сотрудника",
       DUTIES AS "Должность",
       SALARY AS "Оклад"
  FROM STAFF
 WHERE CODORG = 18
 ORDER BY CODPEOPLE;
```

GO

Результат:

Код сотрудника	Должность	Оклад
NULL	Неизвестная должность1	NULL
NULL	Неизвестная должность2	NULL
NULL	Неизвестная должность3	NULL
14	Исполнительный директор	38500.00

Система рассматривает значение NULL, как наименьшее среди всех значений.

8.4.3. Условие выборки данных (*WHERE*)

Для задания условий выборки данных, т. е., какие именно данные нужно выбирать, используется предложение WHERE.

8.4.3.1. Использование операторов сравнения

Выполните скрипт примера 8.6. Здесь выбираются все регионы (республики, края, области) России.

Пример 8.6. Выбор регионов России

```
USE BestDatabase;
GO
SELECT CODREG AS "Код региона",
       NAMEREG AS "Регион",
       CENTER AS "Центр региона"
  FROM REFREG
```

```
WHERE CODCTR = 'RUS'
ORDER BY CENTER;
GO
```

Получим следующий список:

Код региона	Регион	Центр региона
19	Республика Хакасия	АБАКАН
80	Чукотский автономный округ	АНАДЫРЬ
29	Архангельская область	АРХАНГЕЛЬСК
30	Астраханская область	АСТРАХАНЬ
22	Алтайский край	БАРНАУЛ
31	Белгородская область	БЕЛГОРОД
...		

В предложении WHERE код страны можно задать и при помощи оператора SELECT следующим образом:

```
WHERE CODCTR = (SELECT CODCTR
                  FROM REFCTR
                  WHERE NAME = 'РОССИЯ')
```

Внутренний оператор SELECT должен возвращать ровно одно значение или никакого значения. В последнем случае не возникает никакого исключения, просто результат будет содержать нулевое количество строк.

Рассмотрим список сотрудников STAFF организации ЗАО "ИнфоТел". У этой организации в базе данных код 11, что можно отыскать при помощи соответствующего оператора SELECT. Мы будем в операторе выбора сотрудников использовать такой оператор, не указывая явно код организации. Выполните скрипт примера 8.7.

Пример 8.7. Выбор сотрудников организации ЗАО "ИнфоТел"

```
USE BestDatabase;
GO
SELECT CODPEOPLE AS "Код сотрудника",
       DUTIES AS "Должность",
      SALARY AS "Оклад"
  FROM STAFF
 WHERE CODORG = (SELECT COD
                  FROM ORGANIZATION
                  WHERE NAME = 'ЗАО "ИнфоТел"');
GO
```

Результатом будет список из тринадцати сотрудников.

Теперь выберем из списка сотрудников тех, чей оклад превышает средний оклад по организации. Выполните операторы примера 8.8.

Пример 8.8. Выбор сотрудников организации ЗАО "ИнфоТел", чей оклад превышает среднее значение по организации

```
USE BestDatabase;
GO
SELECT CODPEOPLE AS "Код сотрудника",
       DUTIES AS "Должность",
      SALARY AS "Оклад"
  FROM STAFF
 WHERE CODORG = (SELECT COD
                   FROM ORGANIZATION
                  WHERE NAME = 'ЗАО "ИнфоТел"')
AND
      SALARY > (SELECT AVG(SALARY)
                  FROM STAFF
                 WHERE CODORG = (SELECT COD
                                   FROM ORGANIZATION
                                  WHERE NAME = 'ЗАО "ИнфоТел"'));
GO
```

Результат — семь строк.

Давайте на этом примере проиллюстрируем использование обобщенного табличного выражения (CTE). Для получения кода организации в этом примере используются два одинаковых оператора `SELECT`. Зададим обобщенное табличное выражение в следующем виде и используем выборку из полученной таблицы в указанных двух случаях. Выполните операторы примера 8.9.

Пример 8.9. Использование обобщенного табличного выражения при выборе сотрудников организации

```
USE BestDatabase;
GO
WITH CTE (COD) AS (SELECT COD
                      FROM ORGANIZATION
                     WHERE NAME = 'ЗАО "ИнфоТел"')
SELECT CODPEOPLE AS "Код сотрудника",
       DUTIES AS "Должность",
      SALARY AS "Оклад"
  FROM STAFF
 WHERE CODORG = (SELECT COD FROM CTE)
AND
      SALARY > (SELECT AVG(SALARY)
                  FROM STAFF
                 WHERE CODORG = (SELECT COD FROM CTE));
GO
```

Разумеется, это только иллюстрация использования обобщенного табличного выражения. В сложных запросах использование СТЕ может сильно повысить скорость выполнения запроса.

8.4.3.2. Использование варианта *LIKE*

В варианте *LIKE* строковое значение должно содержать указанные символы. В этом варианте можно использовать шаблонные символы: % означает любое, в том числе и нулевое количество любых символов, знак подчеркивания _ означает ровно один любой символ, [] задает группу символов, [^] задает группу недопустимых символов. *LIKE* нечувствителен к регистру.

При отображении списка людей из таблицы *PEOPLE* мы получили 112 строк. Введем условие, по которому фамилия должна заканчиваться на "ОВ". Для этого перед буквами "ОВ" используем шаблонный символ % (пример 8.10).

Пример 8.10. Выбор людей, чья фамилия оканчивается на "ОВ"

```
USE BestDatabase;
GO
SELECT COD AS "Код",
       NAME3 AS "Фамилия",
       NAME1 AS "Имя",
       NAME2 AS "Отчество",
       CONVERT(CHAR(12), BIRTHDAY, 104) AS "Дата рождения"
  FROM PEOPLE
 WHERE NAME3 LIKE '%ОВ'
 ORDER BY NAME3;
GO
```

Вот начальные строки полученного результата:

Код	Фамилия	Имя	Отчество	Дата рождения
66	БРЮКОВ	АЛЕКСАНДР	АЛЕКСЕЕВИЧ	15.02.1958
70	БРЮКОВ	ИГОРЬ	АЛЕКСАНДРОВИЧ	12.05.1969
72	БРЮКОВ	ИГОРЬ	ИГОРЕВИЧ	28.03.1989
59	ВАЛЕНТИНОВ	ГЕННАДИЙ	ПАВЛОВИЧ	03.02.1928
44	ВОЛОСОВ	АНДРЕЙ	СЕРГЕЕВИЧ	16.08.1968

...

Если изменить параметр следующим образом:

```
WHERE NAME3 LIKE '%ОВ%'
```

то система вернет уже 38 строк. Здесь помимо фамилии БРЮКОВ будет уже присутствовать и БРЮКОВА. А вот если изменить параметр так:

```
WHERE NAME3 LIKE '%ОВ_'
```

то мы получим 17 строк, и БРЮКОВ не попадает в этот список, потому что в данном варианте после букв "OB" должна следовать еще одна любая буква (точнее, любой символ).

Чтобы указать, что фамилия должна начинаться на букву "A" или "B", нужно ввести:

```
WHERE NAME3 LIKE '[аб]%'
```

Результатом будет шесть строк.

ЗАМЕЧАНИЕ

Надо сказать, что вариант `LIKE` в SQL Server реализован очень хорошо. Он полностью заменяет существующие в других системах средства `STARTING AT` и `CONTAINING`.

8.4.3.3. Использование варианта `BETWEEN`

Вариант `BETWEEN` позволяет выбрать те строки таблицы, в которых значение указанного столбца находится в заданном диапазоне или не находится в этом диапазоне при наличии ключевого слова `NOT`.

Выберем из списка сотрудников ЗАО "ИнфоТел" только тех, чей оклад находится в диапазоне от 12000 до 23700. Выполните скрипт примера 8.11.

Пример 8.11. Выбор сотрудников с окладом в заданном диапазоне

```
USE BestDatabase;
GO
SELECT CODPEOPLE AS "Код сотрудника",
       DUTIES AS "Должность",
      SALARY AS "Оклад"
  FROM STAFF
 WHERE CODORG = 11 AND
       SALARY BETWEEN 12000 AND 23700;
GO
```

Оператор вернет шесть строк. Оклады этих сотрудников находятся в указанном диапазоне, включая и граничные значения.

Такой же результат мы получим, если используем логическое выражение и операторы сравнения:

```
SELECT CODPEOPLE AS "Код сотрудника",
       DUTIES AS "Должность",
      SALARY AS "Оклад"
  FROM STAFF
 WHERE (CODORG = 11) AND (SALARY >= 12000) AND (SALARY <= 23700);
```

8.4.3.4. Использование варианта `IN`

В варианте `IN` вы можете задать список, среди элементов которого должно (или не должно при присутствии `NOT`) находиться значение указанного столбца.

В этом варианте можно задать как явно представленный список литералов или выражений, так и указать оператор SELECT, который возвращает произвольное количество значений одного столбца.

Мы можем получить список нескольких штатов США их кодам. Выполните оператор, приведенный в примере 8.12.

Пример 8.12. Выбор нескольких штатов США

```
USE BestDatabase;
GO
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
  FROM REFREG
 WHERE CODCTR = 'USA' AND
       CODREG IN ('NY', 'MD', 'TX')
 ORDER BY CENTER;
GO
```

Результат:

Код штата	Штат	Столица штата
NY	New York	ALBANY
MD	Maryland	ANNAPOLIS
TX	Texas	AUSTIN

Такой же результат мы получим, если несколько усложним оператор выборки и зададим внутренний SELECT, который также использует вариант IN (пример 8.13).

Пример 8.13. Выбор нескольких штатов США в другом варианте

```
USE BestDatabase;
GO
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
  FROM REFREG
 WHERE CODCTR = 'USA' AND
       CODREG IN (SELECT CODREG
                  FROM REFREG
                 WHERE CODCTR = 'USA' AND
                       CENTER IN ('ALBANY', 'ANNAPOLIS', 'AUSTIN'))
 ORDER BY CENTER;
GO
```

Нужные данные можно также получать, используя оператор SELECT, который возвращает произвольное (возможно, пустое) количество значений одного столбца.

Например, для получения списка всех организаций, для которых в базе данных присутствует описание сотрудников, нужно выполнить оператор из примера 8.14.

Пример 8.14. Выбор организаций, для которых представлен список сотрудников

```
USE BestDatabase;
GO
SELECT COD AS "Код",
       NAME AS "Организация"
  FROM ORGANIZATION
 WHERE COD IN (SELECT DISTINCT CODORG FROM STAFF)
 ORDER BY NAME;
GO
```

Список будет содержать три строки. Во внутреннем операторе `SELECT`, возвращающем список кодов организаций, используется ключевое слово `DISTINCT` для того, чтобы исключить дублирование кодов в списке. Правда, на результат это никак не повлияет.

8.4.3.5. Использование функций `ALL`, `SOME`, `ANY`, `EXISTS`

Для исследования возможностей этих функций выведем список сотрудников организации с кодом 16:

```
SELECT DUTIES AS "Должность",
       SALARY AS "Оклад"
  FROM STAFF
 WHERE CODORG = 16;
```

В этой организации указано двое сотрудников.

Также выведем список всех сотрудников организации с кодом 11:

```
SELECT DUTIES AS "Должность",
       SALARY AS "Оклад"
  FROM STAFF
 WHERE CODORG = 11;
```

Получим список из 13 человек.

Функция `ALL`

Рассмотрим пример 8.15.

Пример 8.15. Использование функции `ALL`

```
USE BestDatabase;
GO
SELECT DUTIES AS "Должность",
       SALARY AS "Оклад"
```

```

FROM STAFF
WHERE CODORG = 11 AND
      SALARY >= ALL (SELECT SALARY
                      FROM STAFF
                     WHERE CODORG = 16);
GO

```

Из таблицы сотрудников организации с кодом 11 выбираются все сотрудники, чей оклад не меньше оклада любого сотрудника организации с кодом 16. Результатом будет список, состоящий из одиннадцати записей. Здесь более естественно было бы использовать функцию MAX:

```

SELECT DUTIES AS "Должность",
      SALARY AS "Оклад"
     FROM STAFF
    WHERE CODORG = 11 AND
          SALARY >= (SELECT MAX(SALARY)
                      FROM STAFF
                     WHERE CODORG = 16);

```

Функции ANY и SOME

Это два названия одной и той же функции, т. е. синонимы.

Функция возвращает значение "истина", если операция сравнения истинна хотя бы для одного значения, возвращаемого подзапросом.

Выполните операторы примера 8.16.

Пример 8.16. Использование функции SOME (ANY)

```

USE BestDatabase;
GO
SELECT DUTIES AS "Должность",
      SALARY AS "Оклад"
     FROM STAFF
    WHERE CODORG = 11 AND
          SALARY <= SOME (SELECT SALARY
                          FROM STAFF
                         WHERE CODORG = 16);
GO

```

В список должны попасть те записи, для которых найдется такая запись, возвращаемая подзапросом, в которой оклад будет больше, чем оклад отыскиваемой записи. Запрос вернет четыре строки.

Вместо функции SOME можно применить в данном случае функцию MAX, и запрос будет более понятным:

```

SELECT DUTIES AS "Должность",
      SALARY AS "Оклад"

```

```
FROM STAFF
WHERE CODORG = 11 AND
      SALARY <= (SELECT MAX(SALARY)
                  FROM STAFF
                  WHERE CODORG = 16);
```

Этот запрос, разумеется, вернет те же четыре записи.

Функция *EXISTS*

Эта функция возвращает значение "истина", если в списке выбора существует, как минимум, одно возвращаемое значение.

Следующий небольшой пример 8.17 иллюстрирует ее использование.

Пример 8.17. Использование функции *exists*

```
USE BestDatabase;
GO
SELECT CODCTR AS 'Код',
       NAME AS 'Название'
FROM REFCTR
WHERE EXISTS (SELECT * FROM REFREG
               WHERE REFCTR.CODCTR = REFREG.CODCTR);
GO
```

Здесь будут получены сведения о тех странах (таблица REFCTR), для которых в базе данных присутствуют сведения об их регионах (таблица REFREG).

В текущей версии базы данных это пять стран.

Здесь также существует альтернативный вариант выборки данных. Можно использовать агрегатную функцию COUNT():

```
WHERE (SELECT COUNT(*) FROM REFREG
           WHERE REFCTR.CODCTR = REFREG.CODCTR) <> 0;
```

8.4.4. Соединение таблиц

Соединение таблиц в операторе SELECT является одним из наиболее мощных и элегантных средств реляционных баз данных.

Левое внешнее соединение

Левое внешнее соединение чаще всего используется в наших операторах по причине его естественности.

Вначале отбираются строки первой, "главной", таблицы на основании условий, заданных в предложении WHERE. Затем к выбранным строкам добавляются данные из второй, присоединяемой, таблицы в соответствии с условиями соединения, заданными в предложении ON.

Следующий оператор (пример 8.18) выбирает всех сотрудников из организации с кодом 11, у которых заработка плата не превышает величины средней заработной платы в этой организации. При этом вместо ничего не значащего кода сотрудника мы путем соединения таблиц добавляем в результат выборки фамилию, имя и отчество каждого человека, выполнив конкатенацию строк.

Пример 8.18. Левое внешнее соединение таблиц сотрудников организации 11 и людей

```
USE BestDatabase;
GO
SELECT PEOPLE.NAME3 + ' ' +
       PEOPLE.NAME1 + ' ' +
       PEOPLE.NAME2 AS "Сотрудник",
       DUTIES AS "Должность",
       SALARY AS "Оклад"
FROM STAFF
LEFT OUTER JOIN PEOPLE
    ON STAFF.CODPEOPLE = PEOPLE.COD
WHERE CODORG = 11 AND
      SALARY <= (SELECT AVG(SALARY)
                  FROM STAFF
                  WHERE CODORG = 11)
ORDER BY 1;
GO
```

Результат выборки:

Сотрудник	Должность	Оклад
ПАНКОВ ИГОРЬ АЛЕКСАНДРОВИЧ	Начальник отдела	12000.00
ПАНКОВА ГАЛИНА АНАТОЛЬЕВНА	Заместитель начальника отдела	11000.00
ПАРШИНА АНТОНИНА ПЕТРОВНА	Техник	7300.00
ПИНТЕРА АНДРЕЙ НИКОЛАЕВИЧ	Главный бухгалтер	15000.00
ПИНТЕРА ЮЛИЯ НИКОЛАЕВНА	Финансовый директор	12000.00
ШАФРАН НИНА СЕРГЕЕВНА	Заместитель директора по кадрам	13200.00

Разберем выполненный оператор. Соединение таблиц задается в предложении FROM:

```
FROM STAFF
LEFT OUTER JOIN PEOPLE
    ON STAFF.CODPEOPLE = PEOPLE.COD
```

Ключевое слово JOIN задает вид соединения и присоединяемую таблицу. Здесь используется левое внешнее соединение (LEFT OUTER JOIN) таблицы STAFF с таблицей PEOPLE.

Условие соединения задается после ключевого слова ON. "Главной" таблицей здесь является таблица STAFF. К каждой выбранной строке этой таблицы (выбираемые строки главной таблицы определяются как обычно, в предложении WHERE) присо-

единяются данные из таблицы PEOPLE, которые удовлетворяют условию в предложении ON. В нашем случае требуется равенство значения столбца CODPEOPLE из таблицы STAFF, который является внешним ключом, значению столбца COD из таблицы PEOPLE, который является первичным ключом; на него и ссылается внешний ключ таблицы STAFF. На самом деле не существует требования, чтобы в условиях соединения задавались только внешние и первичные (или уникальные) ключи.

Поскольку в нашем операторе присутствует более одной таблицы, мы используем для столбцов уточнения, задавая перед именем столбца имя таблицы, псевдоним или аlias (от англ. *alias*) (см. дальше). Конкретно в этом операторе, чтобы избежать двусмыслинности, мы обязаны использовать уточняющее имя для столбца с именем COD, т. к. это имя встречается в обеих таблицах.

Коль скоро в таблице людей существует подходящий вычисляемый столбец, то мы можем записать предыдущий оператор в следующем виде:

```
SELECT PEOPLE.FULLNAME AS "Сотрудник",
       DUTIES AS "Должность",
       SALARY AS "Оклад"
  FROM STAFF
 LEFT OUTER JOIN PEOPLE
    ON STAFF.CODPEOPLE = PEOPLE.COD
 WHERE CODORG = 11 AND
       SALARY <= (SELECT AVG(SALARY)
                  FROM STAFF
                 WHERE CODORG = 11)
 ORDER BY 1;
```

При создании достаточно сложных операторов SELECT бывает утомительным набирать перед именами столбцов полные имена таблиц, особенно если эти имена достаточно длинные. По этой причине для нас, ленивых, существует возможность давать псевдонимы (или алиасы, alias) для имен таблиц. Сравните следующий оператор:

```
SELECT P.NAME3 + ' ' +
       P.NAME1 + ' ' +
       P.NAME2 AS "Сотрудник",
       DUTIES AS "Должность",
       SALARY AS "Оклад"
  FROM STAFF S
 LEFT OUTER JOIN PEOPLE P
    ON S.CODPEOPLE = P.COD
 WHERE CODORG = 11 AND
       SALARY <= (SELECT AVG(SALARY)
                  FROM STAFF
                 WHERE CODORG = 11)
 ORDER BY 1;
```

В предложении FROM мы для таблицы STAFF задали псевдоним S, а для таблицы PEOPLE — псевдоним P. Эти псевдонимы мы используем в списке выбора и в усло-

вии соединения. Однако во внутреннем операторе SELECT мы не можем использовать указанные во внешнем операторе SELECT псевдонимы, а всегда задаем полное имя таблицы.

Правое внешнее соединение

Правое внешнее соединение является зеркальным отражением левого внешнего соединения. При нем вначале отбираются все строки правой соединяемой таблицы (здесь она становится главной) на основании условий предложения WHERE, затем к ним добавляются строки второй, левой, таблицы, указанной сразу после ключевого слова FROM с учетом условий, заданных в предложении ON.

Для иллюстрации этой зеркальности выполните оператор примера 8.19.

Пример 8.19. Правое внешнее соединение таблицы сотрудников организации 11 и таблицы людей

```
USE BestDatabase;
GO
SELECT P.NAME3 + ' ' +
       P.NAME1 + ' ' +
       P.NAME2 AS "Сотрудник",
       DUTIES AS "Должность",
       SALARY AS "Оклад"
FROM PEOPLE P
RIGHT OUTER JOIN STAFF S
ON S.CODPEOPLE = P.COD
WHERE CODORG = 11 AND
      SALARY <= (SELECT AVG(SALARY)
                  FROM STAFF
                  WHERE CODORG = 11)
ORDER BY 1;
GO
```

Полное внешнее соединение

В случае полного внешнего соединения выбираются все соответствующие условия в предложении WHERE строки как левой, так и правой таблиц. Затем между ними устанавливается соответствие, заданное в предложении ON. При этом дублирующие строки левой и правой таблицы удаляются из результирующего списка.

Выполните скрипт примера 8.20.

Пример 8.20. Полное соединение таблиц сотрудников организации и людей

```
USE BestDatabase;
GO
SELECT P.FULLNAME AS "Сотрудник",
       DUTIES AS "Должность",
       SALARY AS "Оклад"
```

```

FROM STAFF S
FULL OUTER JOIN PEOPLE P
ON S.CODPEOPLE = P.COD
ORDER BY 1;
GO

```

Результатом будет 118 строк. Это строки таблицы STAFF, плюс строки таблицы PEOPLE, минус дублирующие строки.

Двойное соединение

Рассмотрим пример двойного внешнего соединения, т. е. тот случай, когда к первой таблице присоединяется не одна, а уже две таблицы.

Чтобы при отображении сотрудников организаций видеть и их фамилии с именами и отчествами, и организации, в которых они работают, нужно выполнить два левых внешних соединения таблицы STAFF с таблицей PEOPLE и с таблицей ORGANIZATION.

Выполните скрипт примера 8.21.

Пример 8.21. Двойное соединение таблиц

```

USE BestDatabase;
GO
SELECT P.FULLNAME AS "Сотрудник",
       S.DUTIES AS "Должность",
       O.NAME AS "Организация"
  FROM STAFF S
LEFT OUTER JOIN PEOPLE P
    ON S.CODPEOPLE = P.COD
LEFT OUTER JOIN ORGANIZATION O
    ON O.COD = S.CODORG
 WHERE S.CODPEOPLE IS NOT NULL
 ORDER BY 1;
GO

```

Здесь из таблицы PEOPLE мы получаем полное имя, а из таблицы ORGANIZATION — название организации. В предложении WHERE указываем, что нужно получить данные по сотрудникам, которые описаны в таблице PEOPLE, т. е. по тем сотрудникам, у которых внешний ключ, ссылающийся на PEOPLE, не имеет значения NULL.

Результат двойного соединения:

Сотрудник	Должность	Организация
ЕРМИШИН АНТОН ИВАНОВИЧ	Системный администратор	ЗАО "ИнфоТел"
ЕРМИШИН АРТЕМ ИВАНОВИЧ	Программист	ЗАО "ИнфоТел"
ЛЕВИН МИХАИЛ МИХАЙЛОВИЧ	Ночной сторож	РИАН
ЛЕВИН МИХАИЛ МИХАЙЛОВИЧ	Исполнительный директор	ЗАО "ИнфоТел"

ЛЕВИН МИХАИЛ МИХАЙЛОВИЧ	Генеральный директор	ЗАО "Фирма АТТО"
ЛОПАТНИКОВ ВАСИЛИЙ ГЕННАДИЕВИЧ	Программист	ЗАО "ИнфоТел"
ПАНКОВ ИГОРЬ АЛЕКСАНДРОВИЧ	Начальник отдела	ЗАО "ИнфоТел"
ПАНКОВ РОМАН ИГОРЕВИЧ	Программист	ЗАО "ИнфоТел"
ПАНКОВА ГАЛИНА АНАТОЛЬЕВНА	Заместитель начальника отдела	ЗАО "ИнфоТел"
ПАРШИНА АНТОНИНА ПЕТРОВНА	Техник	ЗАО "ИнфоТел"
ПАРШИНА АНТОНИНА ПЕТРОВНА	Служба безопасности	ЗАО "ИнфоТел"
ПИНТЕРА АНДРЕЙ НИКОЛАЕВИЧ	Главный бухгалтер	ЗАО "ИнфоТел"
ПИНТЕРА НИКОЛАЙ НИКОЛАЕВИЧ	Заместитель директора	ЗАО "ИнфоТел"
ПИНТЕРА ЮЛИЯ НИКОЛАЕВНА	Финансовый директор	ЗАО "ИнфоТел"
ЦВЕТКОВА ЕКАТЕРИНА АЛЕКСЕЕВНА	Программист	ЗАО "ИнфоТел"
ШАФРАН НИНА СЕРГЕЕВНА	Заместитель директора по кадрам	ЗАО "ИнфоТел"

Еще один похожий пример соединения. Давайте в таблице PEOPLEADMIN найдем все нарушения закона всеми людьми. При этом выполним двойное левое внешнее соединение с таблицами REFADMIN и PEOPLE (пример 8.22).

Пример 8.22. Двойное соединение таблиц

```
USE BestDatabase;
GO
SELECT P.FULLNAME AS "Человек",
       R.NAME AS "Преступление",
       A.DATEADMIN "Дата"
  FROM PEOPLEADMIN A
 LEFT OUTER JOIN REFADMIN R
    ON A.CODADMIN = R.COD
 LEFT OUTER JOIN PEOPLE P
    ON A.CODPEOPLE = P.COD
 ORDER BY 1;
GO
```

Жутковатое впечатление производит эта группа людей.

Рефлексивное соединение, или самосоединение

Соединяемые таблицы не обязательно должны отличаться от главной таблицы в операторе SELECT. Если таблица соединяется сама с собой, то такое соединение часто называется *рефлексивным* или *самосоединением*. Есть еще один термин для такого соединения — *реентерабельное*.

Рассмотрим следующий пример. Пусть мы собираемся получить список людей, куда добавляются и фамилии их супругов. Выполните операторы примера 8.23.

Пример 8.23. Выборка людей и их супругов

```
USE BestDatabase;
GO
```

```
SELECT PM.FULLNAME AS "Фамилия, имя, отчество",
       PH.NAME3 AS "Супруг / супруга"
  FROM PEOPLE PM
    LEFT OUTER JOIN PEOPLE PH
      ON PM.CODOTHERHALF = PH.COD
 WHERE PM.CODOTHERHALF IS NOT NULL
 ORDER BY PM.FULLNAME;
GO
```

Здесь выбирается полное имя человека и фамилия супруга (супруги).

ЗАМЕЧАНИЕ

Обратите внимание на то, что если в других случаях применения оператора `SELECT`, когда в выборке присутствуют *разные* таблицы, мы иногда можем и не использовать псевдонимы таблиц, то в данном случае мы *обязаны* для таблиц задавать псевдонимы, чтобы точно указывать в операторе, к какой именно таблице относится тот или иной столбец.

Результатом выполнения оператора будет список, содержащий 33 строки таблицы `PEOPLE`.

Теперь сформируем оператор, в котором с таблицей `PEOPLE` трижды будет соединяться эта же самая таблица. Выберем из нашей базы данных тех людей, для которых указаны супруги, мать и отец. Выполним необходимое тройное соединение. Устраним в нем строки с пустыми кодами (пример 8.24).

Пример 8.24. Выборка людей, их супругов и родителей

```
USE BestDatabase;
GO
SELECT PG.FULLNAME AS "Фамилия, имя, отчество",
       PH.NAME3 AS "Супруг / супруга",
       PM.NAME3 AS "Мать",
       PF.NAME3 AS "Отец"
  FROM PEOPLE PG          /* Главная таблица */
    LEFT OUTER JOIN PEOPLE PH        /* Супруг */
      ON PG.CODOTHERHALF = PH.COD
    LEFT OUTER JOIN PEOPLE PM        /* Мать */
      ON PG.CODMOTHER = PM.COD
    LEFT OUTER JOIN PEOPLE PF        /* Отец */
      ON PG.CODFATHER = PF.COD
 WHERE PG.CODOTHERHALF IS NOT NULL AND
       PG.CODMOTHER IS NOT NULL AND
       PG.CODFATHER IS NOT NULL
 ORDER BY PG.FULLNAME;
GO
```

Оператор вернет 12 записей.

Фамилия, имя, отчество	Супруг / супруга	Мать	Отец
ВОЛОСОВА ВАСИЛИСА АНДРЕЕВНА	ЛОПАТНИКОВ	МИТРОФАНОВА	ВОЛОСОВ
ГАЛКИНА ЕКАТЕРИНА ВАЛЕРЬЕВНА	ШИГАЕВ	ГАЛКИНА	ГАЛКИН
ЗАБРОДИНА ИРИНА НИКОЛАЕВНА	БРЮКОВ	ФРОЛОВА	ЗАБРОДИН
ЗАЙЦЕВА ДАРЬЯ ИГОРЬЕВНА	ЦВЕТКОВ	ЗАЙЦЕВА	ЗАЙЦЕВ
ЛЕВИНА ВИКТОРИЯ МИХАЙЛОВНА	ЛЕВИН	ПУШКИНА	ПУШКИН
...			

Внутреннее соединение

Внутреннее соединение похоже на внешнее с той разницей, что если во внешнем соединении в список попадают все строки главной таблицы, соответствующие условию поиска в предложении `WHERE`, то во внутреннем соединении список содержит обычно чуть меньше строк, а именно те строки, для которых в точности выполняется условие соединения (условие в предложении `ON`).

Если выполнить левое внешнее соединение таблицы `STAFF` и таблицы `PEOPLE`, то мы получим 19 строк. Если же выполнить внутреннее соединение этих двух таблиц, то оператор вернет лишь 16 строк (пример 8.25).

Пример 8.25. Внутреннее соединение

```
USE BestDatabase;
GO
SELECT P.FULLNAME AS "Сотрудник",
       DUTIES AS "Должность"
  FROM STAFF S
 INNER JOIN PEOPLE P
    ON S.CODPEOPLE = P.COD
 ORDER BY 1;
GO
```

Здесь три строки в таблице `STAFF` оказались лишними. Им не нашлось соответствия в таблице `PEOPLE`.

Для внутренних соединений не существует ни левых, ни правых вариантов, поскольку в результирующий список попадают только те строки, которые в точности соответствуют условию соединения.

8.4.5. Группировка результатов выборки (*GROUP BY, HAVING*)

Приведем несколько простых примеров группировки результатов запроса.

Мы хотим на законных основаниях получить сведения конфиденциального характера об обобщенных характеристиках окладов сотрудников всех организаций, представленных в нашей базе данных. Выполните скрипт из примера 8.26.

Пример 8.26. Обобщенные сведения об окладах сотрудников организаций

```
USE BestDatabase;
GO
SELECT AVG(SALARY) AS "Среднее",
       MAX(SALARY) AS "Максимум",
       MIN(SALARY) AS "Минимум",
       COUNT(COD) AS "Количество",
       CODORG AS "Организация"
  FROM STAFF
 GROUP BY CODORG
 ORDER BY 4;
GO
```

В операторе мы используем четыре агрегатные функции. В функции COUNT() в этом случае можно указать любой столбец таблицы STAFF или задать символ *. Это ни на что не влияет.

Группировка выполняется по столбцу CODORG, что и требуется по правилам группировки, поскольку только этот столбец является неагрегатным в списке выбора нашего оператора.

В результате выполнения оператора возвращается три строки, поскольку в нашей базе данных присутствуют сведения о трех организациях, имеющих сотрудников.

В полученном списке есть один столбец, который не имеет особого смысла. Это код организации. Выполним в рамках данного оператора еще и операцию соединения, чтобы вместо кода организации получить осмысленное название этой организации (пример 8.27).

Пример 8.27. Улучшенный результат получения конфиденциальных данных

```
USE BestDatabase;
GO
SELECT AVG(SALARY) AS "Среднее",
       MAX(SALARY) AS "Максимум",
       MIN(SALARY) AS "Минимум",
       COUNT(*) AS "Количество",
       O.NAME AS "Организация"
  FROM STAFF S
 LEFT OUTER JOIN ORGANIZATION O
    ON O.COD = S.CODORG
 GROUP BY O.NAME
 ORDER BY 4;
GO
```

Результат:

Среднее	Максимум	Минимум	Количество	Организация
6750.00	12000.00	1500.00	2	ЗАО "Фирма ATTO"
38500.00	38500.00	38500.00	4	РИАН
22684.61	56000.00	7300.00	13	ЗАО "ИнфоТел"

Теперь проиллюстрируем использование предложения HAVING. Для нашего примера зададим условие, что в результирующий список должны помещаться только те строки, где минимальный оклад выше 1500. Для этого изменим наш последний оператор выборки данных (пример 8.28).

Пример 8.28. Использование предложения HAVING

```
USE BestDatabase;
GO
SELECT AVG(SALARY) AS "Среднее",
       MAX(SALARY) AS "Максимум",
       MIN(SALARY) AS "Минимум",
       COUNT(*) AS "Количество",
       O.NAME AS "Организация"
FROM STAFF S
LEFT OUTER JOIN ORGANIZATION O
  ON O.COD = S.CODORG
GROUP BY O.NAME
HAVING MIN(SALARY) > 1500
ORDER BY 4;
GO
```

После предложения GROUP BY мы добавили предложение

```
HAVING MIN(SALARY) > 1500
```

Результат, естественно, сократится на одну строку:

Среднее	Максимум	Минимум	Количество	Организация
38500.00	38500.00	38500.00	4	РИАН
22684.61	56000.00	7300.00	13	ЗАО "ИнфоТел"

Основным требованием к составу предложения HAVING является то, что имена столбцов в этом предложении обязательно должны присутствовать в списке GROUP BY или быть параметрами агрегатной функции.

Давайте рассмотрим еще один пример, где предложение GROUP BY выполняет те же функции, что и ключевое слово DISTINCT.

Количество строк в таблице PEOPLE, как мы с вами выяснили, 112.

Для получения списка неповторяющихся фамилий мы использовали следующий оператор:

```
SELECT DISTINCT NAME3 AS "Фамилия"  
    FROM PEOPLE ORDER BY NAME3;
```

Он возвращал нам 57 строк. Теперь для получения того же списка используем предложение GROUP BY (пример 8.29).

Пример 8.29. Использование предложения GROUP BY вместо DISTINCT

```
USE BestDatabase;  
GO  
SELECT NAME3 AS "Фамилия"  
    FROM PEOPLE  
    GROUP BY NAME3  
    ORDER BY NAME3;  
GO
```

Этот оператор также вернет 57 строк.

Теперь продемонстрируем использование функций CUBE() и ROLLUP(). Вначале в нашей базе данных создадим новую таблицу, в которой будут храниться сведения о движении материалов на складах, и поместим туда несколько строк (пример 8.30).

Пример 8.30. Создание таблицы STOREHOUSE

```
USE BestDatabase;  
GO  
CREATE TABLE STOREHOUSE  
( COD          INT IDENTITY,      /* Ключ */  
  CODSTORE     CHAR(2),           /* Номер цеха */  
  OPERATION    CHAR(1),           /* Код операции: 1 – приход, 2 – расход */  
  MONTHNUM    CHAR(2),           /* Номер месяца операции */  
  GOODS        VARCHAR(100),      /* Товар */  
  NUM          DECIMAL(8,2),       /* Количество товара */  
  PRIMARY KEY (COD)  
);  
GO  
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)  
VALUES ('01', '1', '02', 'Goods1', 12.56);  
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)  
VALUES ('01', '2', '02', 'Goods1', 12.56);  
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)  
VALUES ('01', '1', '02', 'Goods1', 220.04);  
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)  
VALUES ('01', '1', '02', 'Goods2', 100);  
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)
```

```

VALUES ('01', '2', '03', 'Goods1', 12);
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)
VALUES ('01', '1', '02', 'Goods2', 156);
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)
VALUES ('01', '1', '03', 'Goods1', 10);
INSERT INTO STOREHOUSE (CODSTORE, OPERATION, MONTHNUM, GOODS, NUM)
VALUES ('01', '1', '02', 'Goods2', 18.34);
GO

```

Структура таблицы нам понятна. Несколько слов об этой структуре я скажу немноголиже. Выберем данные из этой таблицы, задав при помощи функции ROLLUP() группировку (подведение итогов в нашем случае) по столбцам CODSTORE, OPERATION, MONTHNUM, GOODS (пример 8.31).

Пример 8.31. Использование функции ROLLUP()

```

USE BestDatabase;
GO
SELECT CODSTORE AS "Цех",
       CASE WHEN OPERATION = '1' THEN 'Приход'
             WHEN OPERATION = '2' THEN 'Расход'
             WHEN OPERATION IS NULL THEN '*** Итог ***'
       END AS "Операция",
       MONTHNUM AS "Месяц",
       GOODS AS "Товар",
       SUM(NUM) AS "Количество"
FROM STOREHOUSE
GROUP BY ROLLUP(CODSTORE, OPERATION, MONTHNUM, GOODS)
ORDER BY CODSTORE, MONTHNUM, GOODS;
GO

```

В результате мы получим 13 строк. Страна, где столбцы имеют значение NULL, является итоговой. Страна, в которой все перечисленные в функции ROLLUP() столбцы являются NULL, это итог по всем строкам. Для столбца OPERATION я указал вариант получения итоговых данных. Аналогичные действия можно было бы выполнить и для других столбцов, включенных в функцию.

Теперь выполним выборку с использованием функции CUBE() (пример 8.32).

Пример 8.32. Использование функции CUBE()

```

USE BestDatabase;
GO
SELECT CODSTORE AS "Цех",
       CASE WHEN OPERATION = '1' THEN 'Приход'
             WHEN OPERATION = '2' THEN 'Расход'
             WHEN OPERATION IS NULL THEN '*** Итог ***'
       END AS "Операция",
       MONTHNUM AS "Месяц",
       GOODS AS "Товар",
       SUM(NUM) AS "Количество"
FROM STOREHOUSE
GROUP BY CUBE(CODSTORE, OPERATION, MONTHNUM, GOODS)
ORDER BY CODSTORE, MONTHNUM, GOODS;
GO

```

```
END AS "Операция",
MONTHNUM AS "Месяц",
GOODS AS "Товар",
SUM(NUM) AS "Количество"
FROM STOREHOUSE
GROUP BY CUBE(CODSTORE, OPERATION, MONTHNUM, GOODS)
ORDER BY CODSTORE, MONTHNUM, GOODS;
GO
```

Результатом будет уже 44 строки. Здесь мы получаем итоги в различных мыслимых разрезах.

Относительно наглядности подобных отчетов можно и не говорить. Такие данные требуют дополнительной обработки в других программах, чтобы их можно было показывать нормальным людям.

ЗАМЕЧАНИЕ

Надеюсь вам никогда не придет в голову для реально работающих систем создавать таблицу, аналогичную нашей STOREHOUSE. Ее мы использовали лишь для простенькой демонстрации возможностей выборки данных. По меньшей мере, в ней не должно храниться наименование товара. Справочник товаров должен размещаться в отдельной таблице.

8.5. Использование операторов *UNION, EXCEPT, INTERSECT*

В примере 8.12 мы выполнили выборку из таблицы регионов трех штатов США. Проиллюстрируем использование операторов UNION, EXCEPT, INTERSECT на различных вариантах выборки этих штатов.

Выполните скрипт примера 8.33.

Пример 8.33. Использование оператора UNION

```
USE BestDatabase;
GO
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
  FROM REFREG
 WHERE CODCTR = 'USA' AND
       CODREG IN ('NY', 'MD', 'TX')
UNION
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
  FROM REFREG
```

```

WHERE CODCTR = 'USA' AND
      CODREG IN ('NY', 'MD', 'TX')
ORDER BY CENTER;
GO

```

Однако оператор выбирает те же три штата. Дело в том, что мы не указали ключевое слово ALL. По этой причине дублированные строки удаляются из результата. Если в оператор UNION добавить это слово, то результат будет содержать шесть строк, где каждый штат отображается дважды.

В скрипте примера 8.34 иллюстрируется использование оператора EXCEPT.

Пример 8.34. Использование оператора EXCEPT

```

USE BestDatabase;
GO
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
FROM REFREG
WHERE CODCTR = 'USA' AND
      CODREG IN ('NY', 'MD', 'TX')
EXCEPT
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
FROM REFREG
WHERE CODCTR = 'USA' AND
      CODREG IN ('NY', 'MD')
ORDER BY CENTER;
GO

```

Здесь из списка трех штатов удаляются два штата, полученные во втором запросе. В результате мы получаем только одну строку, описывающую штат Техас.

В скрипте примера 8.35 выполняется пересечение двух множеств.

Пример 8.35. Использование оператора INTERSECT

```

USE BestDatabase;
GO
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
FROM REFREG
WHERE CODCTR = 'USA' AND
      CODREG IN ('NY', 'MD', 'TX')

```

```
INTERSECT
```

```
SELECT CODREG AS "Код штата",
       NAMEREG AS "Штат",
       CENTER AS "Столица штата"
  FROM REFREG
 WHERE CODCTR = 'USA' AND
       CODREG IN ('NY', 'MD')
 ORDER BY CENTER;
```

```
GO
```

Результатом будет две строки, которые возвращаются и первым, и вторым запросом. Это штаты Нью-Йорк и Мериленд.

Что будет дальше?

В следующей главе 9 мы рассмотрим представления, для получения которых используется оператор SELECT.



ГЛАВА 9

Представления

- ◆ Работа с представлениями в Transact-SQL
- ◆ Диалоговые средства создания представлений в Management Studio
- ◆ Примеры представлений

Представление (view) — это объект реляционной базы данных, который описывает *виртуальную* таблицу. Есть еще хороший термин — хранимый запрос SELECT. Определение представления содержит оператор SELECT любой сложности, при выполнении которого получается та самая виртуальная таблица, которая физически в базе данных не хранится.

Фактически представление — это заранее подготовленный и сохраненный в системе оператор SELECT, выбирающий нужные нам столбцы и строки из любого количества таблиц базы данных.

ЗАМЕЧАНИЕ

Результаты выполнения представления в базе данных не хранятся. Там присутствует лишь определение представления. Результаты будут получены при обращении к представлению при помощи опять же оператора SELECT.

Представления создаются для того, чтобы облегчить жизнь людям, регулярно осуществляющим сложные выборки из базы данных. Вместо того чтобы каждый раз писать сложный оператор поиска данных, достаточно обратиться к существующему представлению. Кроме этого, представления могут использоваться для скрытия от пользователя значений некоторых столбцов таблиц, к которым у этого пользователя нет полномочий. При помощи представлений такой пользователь может получать доступ к значениям тех столбцов таблицы (таблиц), которые ему необходимы для работы.

При использовании представления можно изменять данные в базовых таблицах представления. Изменениям подлежат только значения "обычных" столбцов. Нельзя изменить значение вычисляемого столбца. Разумеется, нет возможности внести

изменения в данные, полученные при помощи агрегатных функций. Нельзя при помощи представления добавить новую строку в базовую таблицу, если в списке выбора не присутствуют все столбцы, входящие в состав первичного ключа. При этом если первичный ключ описан с характеристикой `IDENTITY` и он не присутствует в списке выбора представления, то добавление новых строк возможно.

Представления могут быть индексированными. Для представления допустимо создание уникального кластерного индекса. В этом случае результирующий набор данных представления будет физически храниться в базе данных. Изменения в базовой таблице, связанные со столбцами, входящими в состав такого индекса, приведут к выполнению соответствующих изменений в индексе.

Существует понятие секционированных представлений. Это представления, строки которых получены из разных таблиц, может быть находящихся в разных базах данных. Результат получается при объединении данных операторов `SELECT` операторами `UNION ALL`. Здесь в каждой таблице должны выбираться все столбцы. Структура всех таблиц должна быть одинаковой.

9.1. Синтаксис операторов для представлений

9.1.1. Создание представления

Для создания представления в языке Transact-SQL используется оператор `CREATE VIEW`. Синтаксис оператора показан в листинге 9.1 и графе 9.1.

Листинг 9.1. Синтаксис оператора `CREATE VIEW`

```
<оператор CREATE VIEW> ::=  
CREATE VIEW [ <схема>. ] <имя представления>  
[ ( <столбец> [, <столбец>] ... ) ]  
[ WITH <атрибут> [, <атрибут>] ... ]  
AS <запрос> [ WITH CHECK OPTION ] ;  
  
<атрибут> ::= { ENCRYPTION | SCHEMABINDING | VIEW_METADATA }
```



Граф 9.1. Синтаксис оператора `CREATE VIEW`

Представление создается в указанной схеме или в схеме по умолчанию *текущей* базы данных. Имя представления должно быть уникальным среди имен представлений и таблиц данной схемы.

Список столбцов, если он указан, задает имена столбцов, возвращаемых представлением. Если список не указан, то столбцам присваиваются имена, возвращаемые оператором `SELECT`. Количество столбцов в списке должно совпадать с количеством столбцов оператора `SELECT`. Представление может возвращать до 1024 столбцов.

Необязательное предложение `WITH` позволяет задать до трех атрибутов (свойств) создаваемого представления.

Атрибут `ENCRYPTION` указывает, что текст представления будет зашифрован при помещении в систему.

Если указано `SCHEMABINDING`, то нельзя будет изменить описания базовых таблиц таким образом, что это повлияет на представление. При указании этого атрибута базовые таблицы в представлении должны быть записаны в виде `<схема>. <таблица>`. Все указанные в `SCHEMABINDING` объекты должны находиться в одной базе данных.

Атрибут `VIEW_METADATA` указывает, что в ODBC, в OLE DB и в API-интерфейсы DB-Library система по соответствующему запросу передаст сведения о метаданных представления, а не о базовых таблицах этого представления.

После ключевого слова `AS` располагается сколь угодно сложный оператор `SELECT`, на основании которого выбираются данные из базовой таблицы или из нескольких базовых таблиц.

Здесь предложение `ORDER BY` может присутствовать только в случае, если в операторе есть ключевое слово `TOP` или `OFFSET`. То есть, представление не гарантирует конкретный порядок возвращаемых строк. Такой порядок можно задать при обращении к самому представлению в предложении `ORDER BY`.

В операторе `SELECT` также недопустимо использование предложений `INTO`, `OPTION` и ссылок на временные таблицы и табличные переменные.

Предложение `WITH CHECK OPTION` означает, что при изменении данных в базовых таблицах представления при обращении к представлению будут разрешены только такие изменения, при которых измененные строки будут видны при использовании этого представления. Иными словами, изменения в этом случае также должны соответствовать условиям в предложении `WHERE` данного представления.

9.1.2. Изменение представления

Для изменения представления используется оператор `ALTER VIEW`. Синтаксис оператора представлен в листинге 9.2 и графе 9.2.

Листинг 9.2. Синтаксис оператора `ALTER VIEW`

```
<оператор ALTER VIEW> ::=  
    ALTER VIEW [ <схема>. ] <имя представления>  
        [ ( <столбец> [, <столбец>] ... ) ]
```

```
[ WITH <атрибут> [, <атрибут>] ... ]
AS <запрос> [ WITH CHECK OPTION ] ;
```

<атрибут> ::= { ENCRYPTION | SCHEMABINDING | VIEW_METADATA }



Граф 9.2. Синтаксис оператора ALTER VIEW

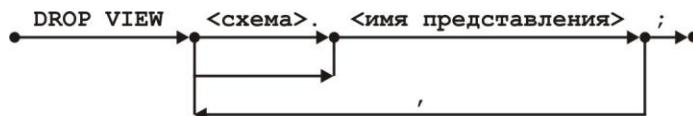
Оператор в точности повторяет все конструкции оператора создания представления.

9.1.3. Удаление представления

Для удаления группы представлений используется оператор DROP VIEW. Синтаксис представлен в листинге 9.3 и графе 9.3.

Листинг 9.3. Синтаксис оператора DROP VIEW

```
<оператор DROP VIEW> ::=
DROP VIEW [ <схема>. ] <имя представления>
[ , [ <схема>. ] <имя представления> ] ... ;
```



Граф 9.3. Синтаксис оператора DROP VIEW

Оператор позволяет удалить одно или более представлений.

9.2. Создание представлений в Transact-SQL

Создадим очень простое представление, возвращающее все регионы России из таблицы регионов (пример 9.1).

Пример 9.1. Создание простого представления

```
USE BestDatabase;
GO
IF OBJECT_ID('VIEW_RUSSIA', 'V') IS NOT NULL
    DROP VIEW VIEW_RUSSIA;
CREATE VIEW VIEW_RUSSIA (CODREG, NAMEREG, CENTER)
AS
SELECT CODREG,
       NAMEREG,
       CENTER
  FROM REFREG
 WHERE CODCTR = (SELECT CODCTR
                   FROM REFCTR
                  WHERE NAME = 'РОССИЯ');
GO
```

Вначале проверяется, существует ли представление с таким именем. Если существует, то оно удаляется.

Оператор `SELECT` представления мы уже использовали в предыдущей главе 8.

Обращение к такому представлению может быть следующим:

```
SELECT CODREG AS "Код региона",
       NAMEREG AS "Регион",
       CENTER AS "Центр региона"
  FROM VIEW_RUSSIA
 ORDER BY CENTER;
```

GO

При обращении к представлению можно, как и в случае обычного оператора `SELECT`, использовать дополнительные условия выборки данных. Например, можно выполнить такую выборку данных:

```
SELECT * FROM VIEW_RUSSIA
 WHERE CODREG IN ('01', '02', '03');
```

В результате будут возвращены три строки базовой таблицы.

Столбцы базовой таблицы этого представления можно изменить, используя обычный оператор `UPDATE`, например, следующим образом:

```
UPDATE VIEW_RUSSIA SET CODREG = 'zz' WHERE CODREG = '01';
```

Из базовой таблицы можно удалять строки, например:

```
DELETE FROM VIEW_RUSSIA WHERE CODREG = 'zz';
```

А вот добавлять данные в базовую таблицу при использовании этого представления нельзя, поскольку представление не возвращает значение столбца `CODCTR`, который входит в состав первичного ключа. При попытке добавления система вернет сообщение, что первичный ключ не может иметь значения `NULL`.

Создадим представление, которое будет возвращать список всех районов всех стран. Оператор создания представления показан в примере 9.2.

Пример 9.2. Создание представления, включающее внешнее соединение

```
USE BestDatabase;
GO
IF OBJECT_ID('VIEW_AREAS', 'V') IS NOT NULL
    DROP VIEW VIEW_AREAS;
CREATE VIEW VIEW_AREAS
    (CODCTR, NAMECTR, CODREG, CODAREA, CENTER)
AS
SELECT A.CODCTR, C.NAME, A.CODREG, A.CODAREA, A.CENTER
    FROM REFAREA A
    LEFT OUTER JOIN REFCTR C
        ON A.CODCTR = C.CODCTR;
GO
```

Для такого представления нельзя создать индекс, потому что оператор `SELECT` содержит внешнее соединение. Допустимым является внутреннее (`INNER`) соединение. Изменим наше представление, убрав внешнее соединение и добавив указание на схему при задании таблицы. Также для индексированного представления нужно указать атрибут `SCHEMABINDING` (пример 9.3).

Пример 9.3. Создание индексированного представления

```
USE BestDatabase;
GO
IF OBJECT_ID('VIEW_AREAS', 'V') IS NOT NULL
    DROP VIEW VIEW_AREAS;
CREATE VIEW dbo.VIEW_AREAS
    (CODCTR, CODREG, CODAREA, CENTER)
WITH SCHEMABINDING
AS
SELECT CODCTR, CODREG, CODAREA, CENTER
    FROM dbo.REFAREA;
GO
CREATE UNIQUE CLUSTERED INDEX IND_AREAS
    ON dbo.VIEW_AREAS (CODCTR, CODREG, CODAREA);
GO
```

Чтобы удалить созданные представления, выполните следующий скрипт:

```
USE BestDatabase;
GO
DROP VIEW VIEW_RUSSIA, VIEW_AREAS;
GO
```

9.3. Создание представлений диалоговыми средствами Management Studio

Создадим такое же представление, что и в примере 9.1.

Для создания в нашей базе данных представления средствами Management Studio нужно в **Object Explorer** раскрыть папку **Databases**, папку **BestDatabase**, щелкнуть правой кнопкой мыши по папке **Views** и в появившемся контекстном меню выбрать элемент **New View**. Появится окно выбора таблицы **Add Table** (рис. 9.1), которое содержит вкладки выбора таблицы (**Tables**), представления (**Views**), функций (**Functions**) и синонимов (**Synonyms**). Текущей будет вкладка **Tables**.

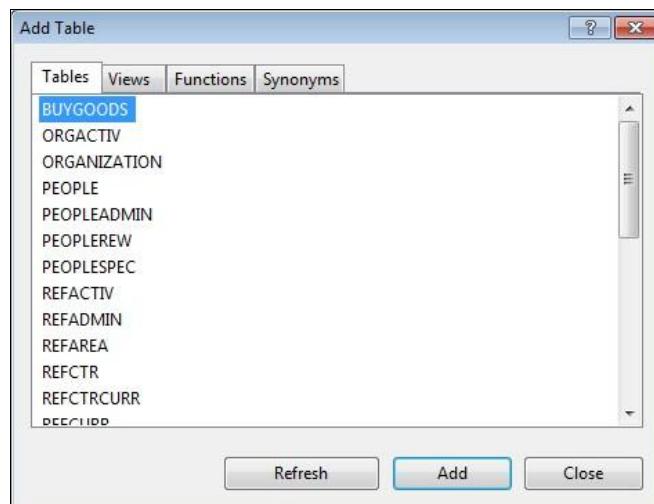


Рис. 9.1. Окно Add Table, вкладка Tables

Выберите таблицу **REFREG** и щелкните по кнопке **Add**, после чего закройте окно. Появится окно выбора столбцов таблицы **REFREG** (рис. 9.2).

Отметьте нужные три столбца (рис. 9.3).



Рис. 9.2. Окно выбора столбцов таблицы

Рис. 9.3. Выбор требуемых столбцов таблицы

Column	Alias	Table	Output	Sort Type	Sort Order	Filter
► CODREG		REFREG	<input checked="" type="checkbox"/>			
NAMEREG		REFREG	<input checked="" type="checkbox"/>			
CENTER		REFREG	<input checked="" type="checkbox"/>			

```
SELECT CODREG, NAMEREG, CENTER
FROM dbo.REFREG WHERE CODCTR = (SELECT CODCTR FROM REFCTR WHERE NAME = 'РОССИЯ')
```

Рис. 9.4. Созданный оператор SELECT

В нижней части окна будет сформирован оператор SELECT. Туда нужно будет только добавить предложение WHERE (рис. 9.4).

Выберите в меню **File | Save DEVRAVE.BestDatabase – dbo.View_1**. Появится окно задания имени создаваемого представления (рис. 9.5). Введите имя **VIEW_RUSSIA2** и щелкните по кнопке **OK**.

Представление будет сохранено в схеме `dbo` базы данных `BesDatabase`.



Рис. 9.5. Ввод имени представления

Что будет дальше?

В реляционных базах данных существует такой механизм, как транзакции. При кажущейся сложности транзакции достаточно просты и удобны в работе. От правильных вариантов задания свойств используемых в программах транзакциям зависит комфортность работы группы пользователей с одной базой данных в архитектуре "клиент/сервер".

Всю следующую главу 10 мы посвятим транзакциям, их характеристикам и влиянию этих характеристик на взаимодействие параллельных процессов. Мы будем рассматривать каждое свойство, каждую возможность и тут же проверять на практике, как это работает.



ГЛАВА 10

Транзакции

- ◆ Понятие транзакции
- ◆ Характеристики транзакций
- ◆ Синтаксис операторов работы с транзакциями

10.1. Понятие и характеристики транзакций

Все действия, выполняемые с базой данных — любые изменения, как данных, так и метаданных, а также любая выборка данных, — осуществляются в контексте (под управлением) какой-либо транзакции. Изменения, выполненные в контексте одной транзакции, можно либо все подтвердить (при отсутствии ошибок базы данных), тогда эти изменения будут записаны в базу данных, либо все отменить. Если в любой операции, выполняемой в контексте транзакции, произошла ошибка, то подтвердить такую транзакцию нельзя. Можно только отменить все действия.

Транзакция — это механизм, переводящий базу данных из одного непротиворечивого состояния в другое непротиворечивое состояние.

Транзакция может быть объявлена явно при помощи оператора `BEGIN TRANSACTION`. Транзакция может быть *неявной*, когда начинает выполняться оператор работы с базой данных без оператора старта транзакции. Кроме того, транзакции могут быть *вложенными*.

Важнейшая характеристика транзакции — *уровень изоляции*, который определяет, какие изменения данных, выполненные другими параллельными процессами, видны в текущей транзакции, и могут ли другие процессы одновременно использовать данные текущей транзакции.

При работе с транзакциями часто используется переменная `@@TRANCOUNT`, которая содержит количество вложенных запущенных на выполнение и не подтвержденных или отмененных в данном соединении с сервером транзакций. Число, большее 1, означает, что выполняется транзакция, являющаяся внутренней для основной, главной, транзакции. Для анализа успешности выполнения операторов обращения

к базе данных полезной является переменная @@ERROR. При успешном выполнении операции переменная имеет значение 0. @@ERROR очищается при начале выполнения любого другого оператора, поэтому ее надо проверять сразу после операции работы с базой данных.

Во всех операторах, связанных с транзакциями, имена транзакций и точек сохранения можно задавать и при помощи ссылки на строковые локальные переменные. Напомню, имена таких переменных должны начинаться с символа @. В операторах вместо ключевого слова TRANSACTION можно использовать сокращение TRAN.

10.2. Операторы работы с транзакциями

Для явного запуска транзакции используется оператор BEGIN TRANSACTION, синтаксис которого представлен в листинге 10.1 и граfe 10.1.

Листинг 10.1. Синтаксис оператора BEGIN TRANSACTION

```
<оператор BEGIN TRANSACTION> ::=  
    BEGIN TRANSACTION [ <имя> ] [ WITH MARK [ '<описание>' ] ] ;
```



Граф 10.1. Синтаксис оператора BEGIN TRANSACTION

Количество символов в имени транзакции не может превышать 32. Имя может даваться только для самой внешней транзакции в группе вложенных транзакций.

Необязательное предложение WITH MARK указывает, что транзакция отмечается в журнале транзакций. При этом должно быть задано имя транзакции. Наличие этого предложения позволяет выполнять восстановление базы данных до или после именованной отметки. В предложении WITH MARK можно задать описание, длина которого не должна превышать 128 символов.

Транзакция подтверждается оператором COMMIT TRANSACTION или отменяется оператором ROLLBACK TRANSACTION.

Синтаксис оператора COMMIT TRANSACTION представлен в листинге 10.2 и граfe 10.2.

Листинг 10.2. Синтаксис оператора COMMIT TRANSACTION

```
<оператор COMMIT TRANSACTION> ::=  
    COMMIT TRANSACTION [ <имя> ] ;
```



Граф 10.2. Синтаксис оператора COMMIT TRANSACTION

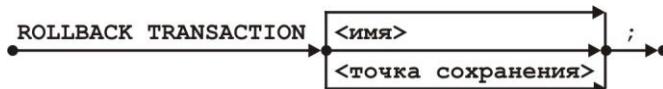
Оператор подтверждает транзакцию. Если значение переменной @@TRANCOUNT больше единицы (подтверждается вложенная транзакция), то оператор лишь уменьшает значение этой переменной на единицу. Данные, измененные операторами всех вложенных транзакций, станут постоянными в базе данных лишь после подтверждения основной транзакции самого верхнего уровня вложенности.

Аналогичный этому оператору применяется оператор COMMIT WORK. При этом ключевое слово WORK может быть опущено. Отличие этих операторов только в отсутствии имени транзакции.

Для отмены транзакции используется оператор ROLLBACK TRANSACTION. Его синтаксис показан в листинге 10.3 и графе 10.3.

Листинг 10.3. Синтаксис оператора ROLLBACK TRANSACTION

```
<оператор ROLLBACK TRANSACTION> ::=  
    ROLLBACK TRANSACTION [ <имя> | <точка сохранения> ] ;
```



Граф 10.3. Синтаксис оператора ROLLBACK TRANSACTION

Отменяет все выполненные изменения в рамках транзакции, т. е. выполняет откат на начало транзакции или осуществляет откат на указанную точку сохранения, созданную ранее оператором SAVE TRANSACTION (см. далее). В этом случае отменяются лишь изменения, выполненные после создания точки сохранения.

Важный момент. Независимо от того, на каком уровне вложенности транзакции выполняется откат транзакции (не на точку сохранения), будут отменены все действия, осуществляемые в контексте транзакции самого высокого уровня. При этом значение переменной @@TRANCOUNT устанавливается в 0.

Те же действия по откату транзакции можно выполнить при использовании оператора ROLLBACK WORK. Ключевое слово WORK в операторе может быть опущено.

Для создания точки сохранения используется оператор SAVE TRANSACTION. Его синтаксис показан в листинге 10.4 и графе 10.4.

Листинг 10.4. Синтаксис оператора SAVE TRANSACTION

```
<оператор SAVE TRANSACTION> ::=  
    SAVE TRANSACTION <точка сохранения>;
```



Граф 10.4. Синтаксис оператора SAVE TRANSACTION

Имя точки сохранения, как и имя транзакции, не должно превышать 32 символов. Оператор создает точку сохранения, к которой в дальнейшем можно вернуться (т. е. отменить все последующие операции изменения данных) при выполнении оператора `ROLLBACK TRANSACTION`.

В транзакции могут создаваться точки сохранения и с одинаковыми именами. При возврате на точку сохранения происходит переход к самой последней по времени точке с этим именем.

Для запуска распределенной транзакции используется оператор `BEGIN DISTRIBUTED TRANSACTION`, синтаксис которого представлен в листинге 10.5 и графе 10.5.

Листинг 10.5. Синтаксис оператора `BEGIN DISTRIBUTED TRANSACTION`

```
<оператор BEGIN DISTRIBUTED TRANSACTION> ::=  
    BEGIN DISTRIBUTED TRANSACTION [ <имя> ] ;
```



Граф 10.5. Синтаксис оператора `BEGIN DISTRIBUTED TRANSACTION`

Оператор запускает распределенную транзакцию, т. е. транзакцию, в контексте которой будут выполняться операторы обращения к разным базам данных в одном или в нескольких экземплярах сервера базы данных. Такие транзакции управляются координатором распределенных транзакций (*Microsoft Distributed Transaction Coordinator*, MS DTC). При использовании распределенных транзакций на компьютере должен быть установлен MS DTC.

Подтверждение и отмена распределенных транзакций выполняются теми же операторами, что и для обычных транзакций.

10.3. Уровни изоляции транзакции

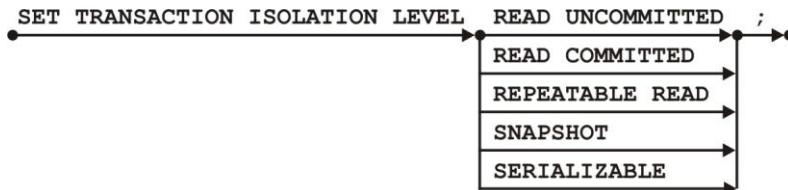
Уровни изоляции позволяют влиять на появление конфликтов блокировки записей, когда разные процессы пытаются изменить одну и ту же строку таблицы.

Для задания уровня изоляции, с которым будут выполняться транзакции, используется оператор `SET TRANSACTION ISOLATION LEVEL`. Его синтаксис показан в листинге 10.6 и графе 10.6.

Листинг 10.6. Синтаксис оператора `SET TRANSACTION ISOLATION LEVEL`

```
<оператор SET TRANSACTION ISOLATION LEVEL> ::=  
    SET TRANSACTION ISOLATION LEVEL  
    { READ UNCOMMITTED  
    | READ COMMITTED
```

```
| REPEATABLE READ
| SNAPSHOT
| SERIALIZABLE
};
```



Граф 10.6. Синтаксис оператора SET TRANSACTION ISOLATION LEVEL

Уровень изоляции — важнейшая характеристика транзакции. Он определяет, какие изменения других процессов будут видны в контексте данной транзакции, и будут ли другие процессы иметь доступ к используемым в рамках транзакции данным.

В SQL Server 2012 поддерживается пять уровней изоляции.

- ◆ **READ UNCOMMITTED.** Неподтвержденное или "грязное" чтение. Этот уровень вообще не изолирует транзакцию от других транзакций. Операторы в рамках такой транзакции могут видеть изменения, выполненные, но не подтвержденные другими параллельными процессами. Такие изменения могут быть еще раз изменены или отменены в процессе выполнения текущей транзакции. База данных в такой ситуации не является непротиворечивой. Я как-то порасспрашивал многих программистов, для чего такой уровень нужен. Получил большое количество серьезных объяснений. Однако на вопрос, используют ли они этот уровень изоляции, все ответили отрицательно.
- ◆ **READ COMMITTED.** Это значение установлено в системе по умолчанию. При этом уровне изоляции операторы транзакции видят все подтвержденные изменения в базе данных. Неподтвержденные, временные, грязные изменения здесь не видны. База данных представляется как непротиворечивая. Однако данные в базе данных могут быть изменены или удалены другими процессами. В результате появляются так называемые *фантомные данные*. При этом такой уровень изоляции, как правило, уменьшает количество блокировок и конфликтов. В большинстве своих разработок я использую именно этот уровень. Поведение системы при этом уровне изоляции зависит от значения параметра базы данных (см. *приложение 4*) READ_COMMITTED_SNAPSHOT.
 - Если значение параметра **OFF** (по умолчанию), то транзакция не сможет считать строки, которые обновляются другим процессом до момента подтверждения или отмены транзакции другого процесса. В этом случае возникают блокировки.
 - Если же параметр **READ_COMMITTED_SNAPSHOT** имеет значение **ON**, то текущая транзакция получает мгновенный снимок базы данных на момент старта этой транзакции. Никакие изменения считываемых записей не приводят к блоки-

ровкам текущей транзакции. Транзакция видит только то состояние данных, которое было на момент старта транзакции. Это достигается за счет версионности изменяемых данных — транзакция видит ту версию строк, которая существовала при запуске транзакции.

- ◆ **REPEATABLE READ.** Также исключает грязное чтение. Другие процессы не могут изменять или удалять данные, прочитанные в контексте такой транзакции, но могут добавлять новые данные. Этот уровень тоже может дать появление фантомных записей. Использование такого уровня изоляции не очень приветствуется.
- ◆ **SNAPSHOT.** База данных представляется на этом уровне как неизменный набор данных, как мгновенный снимок базы данных на момент запуска транзакции. То состояние, которое имели данные на начало старта транзакции, не изменяется для операторов в контексте этой транзакции, хотя другие процессы могут добавлять, изменять и удалять данные базы данных. Здесь исключено появление фантомных данных. Чтобы использовать этот уровень изоляции, опция этой базы данных `ALLOW_SNAPSHOT_ISOLATION` (см. *приложение 4*) должна быть установлена в значение `ON`.
- ◆ **SERIALIZABLE.** Доступны только подтвержденные изменения базы данных. Другие процессы могут читать любые данные, но не могут изменять данные, прочитанные в контексте этой транзакции, и не могут добавлять строки, которые могли бы стать доступными в данной транзакции. Здесь невозможно появление фантомных записей, сводится к нулю возможность появления блокировок при выполнении операций в контексте данной транзакции. При этом ограничиваются возможности работы с базой данных других процессов.

Основное правило использования транзакций такое: *независимо от используемого уровня изоляции рекомендуется транзакции, в которых производится изменение данных, делать как можно короче*. Транзакции чтения данных, если они не ухудшают возможности параллельных процессов, можно делать достаточно длинными.

Что будет дальше?

В следующей главе 11 мы рассмотрим хранимые процедуры и триггеры, которые позволяют предоставить сложные алгоритмы обработки данных на стороне сервера базы данных и во многих случаях дают возможность уменьшить сетевой трафик.



ГЛАВА 11

Хранимые процедуры, функции, определенные пользователем, триггеры

- ◆ Язык хранимых процедур и триггеров
- ◆ Хранимые процедуры
- ◆ Функции, определенные пользователем
- ◆ Триггеры

Хранимые процедуры, пользовательские функции и триггеры являются программами. Они хранятся в области метаданных базы данных и выполняются на стороне сервера, что во многих случаях может сильно сократить сетевой трафик, нагрузку сети.

Как правило, они выполняют какие-то действия с базой данных, в которой определены, однако это не является обязательным. Они могут выполнять и любые другие действия, никак не связанные с базой данных.

К хранимым процедурам и пользовательским функциям могут обращаться любые программы, работающие с базой данных — хранимые же процедуры, функции, триггеры, клиентские приложения.

К триггерам напрямую обращение невозможно. Они автоматически вызываются при наступлении некоторого события базы данных — добавление новой строки в таблицу, удаление строки, изменение существующей строки, а также при соединении с базой данных и при изменении метаданных.

Все эти программные компоненты могут быть созданы с использованием языковых средств Transact-SQL или при помощи сборок (assembly) в среде CLR (Common Language Runtime) платформы Microsoft .NET Framework. Здесь мы будем рассматривать только Transact-SQL.

11.1. Язык хранимых процедур и триггеров

Язык, используемый для написания хранимых процедур, пользовательских функций и триггеров, PSQL, является полноценным языком программирования, который позволяет выполнить любые действия по обработке данных — как локальных, так и хранящихся в базе данных.

В языке допустимы практически любые действия с базой данных, за исключением создания и изменения метаданных. Можно использовать операторы работы с данными базы данных: `INSERT`, `UPDATE`, `DELETE` и `SELECT`. Только в триггерах и хранимых процедурах можно отправлять сообщения (events), вызывать пользовательские исключения (exception). Нельзя устанавливать и разрывать связь с базой данных, манипулировать транзакциями.

При создании как процедур, так и триггеров в синтаксисе выделяются заголовок и тело (хранимой процедуры, триггера). Тело является *блоком операторов*.

Блок операторов `BEGIN/END`

Все действия описываются в блоке операторов между необязательными ключевыми словами `BEGIN` и `END`. Программы могут содержать произвольное количество блоков, как независимых, так и вложенных друг в друга.

Комментарии

В любое место программного кода мы можем поместить комментарий. Текст комментария располагается между символами `/*` и `*/`, как в языке С. Комментарий может занимать произвольное количество строк.

Другой способ задания комментария — два символа минус: `--`. В этом случае комментарий продолжается только до конца строки.

Локальные переменные

Программа может содержать описания локальных переменных. Их имена должны начинаться с символа `@`.

В теле процедур и триггеров могут присутствовать операторы присваивания, оператор ветвления и операторы циклов.

Вообще надо сказать, что все эти возможности можно использовать и в обычных пакетах в командной строке и в Management Studio.

Напомню, для объявления локальной переменной используется оператор `DECLARE`:

```
DECLARE <имя переменной> [ AS ] <тип данных> [ = <значение> ];
```

В одном операторе можно объявить произвольное количество локальных переменных, разделяя объявления запятыми. При объявлении переменной ей можно также задать и начальное значение.

Для присваивания значения локальной переменной используется оператор `SET`:

```
SET <имя переменной> = <выражение>;
```

Ветвление в программе. Операторы **IF** и **CASE**

Для осуществления ветвления в программе здесь используется, как и в большинстве языков программирования, оператор **IF**. Его синтаксис:

```
IF <условие>
  { <оператор> | <блок операторов> }
[ ELSE
  { <оператор> | <блок операторов> } ];
```

Условие может быть сколь угодно сложным выражением, возвращающим логическое значение. Для этого и других операторов: если условие содержит оператор **SELECT**, возвращающий значение, то весь этот оператор должен быть заключен в круглые скобки. При выполнении условия (когда условие возвращает значение **TRUE**) выполняется оператор или блок операторов, следующий за условием. Если присутствует ключевое слово **ELSE**, то при неистинности условия выполняется оператор (блок операторов), следующий за этим ключевым словом.

Ветвление в программах может быть выполнено и при помощи оператора **CASE**. Этот оператор имеет две формы. Синтаксис первой формы:

```
CASE <выражение>
  WHEN <значение> THEN { <оператор> | <блок операторов> }
  [ WHEN <значение> THEN { <оператор> | <блок операторов> } ]...
END;
```

Здесь выражение возвращает некоторое значение. Если в последующем предложении **WHEN** указано именно это значение, то выполняется соответствующий оператор (группа операторов), заданных после ключевого слова **THEN**. Остальные предложения **WHEN** не проверяются.

Синтаксис второй формы:

```
CASE
  WHEN <условие> THEN { <оператор> | <блок операторов> }
  [ WHEN <условие> THEN { <оператор> | <блок операторов> } ]...
  [ ELSE { <оператор> | <блок операторов> } ]
END;
```

В этой форме предложения **WHEN** в принципе никак не связаны друг с другом по используемым в них выражениям. Если условие имеет значение **TRUE**, то выполняется соответствующий оператор (группа операторов) после ключевого слова **THEN** и работа оператора **CASE** завершается. Если ни одно условие **WHEN** не возвращает истинного значения, то выполняется оператор (группа операторов) в предложении **ELSE**, если оно присутствует.

Организация циклов. Оператор **WHILE**

Для организации циклов используется оператор **WHILE**:

```
WHILE <условие>
  { <оператор> | <блок операторов> };
```

Оператор (блок операторов) выполняется, пока будет истинным заданное условие. При входе в цикл вначале проверяется условие. Если оно истинно, выполняются операторы цикла. Затем выполняется переход на начало цикла, на проверку условия. Чтобы цикл не был бесконечным, в теле цикла должны находиться операторы, которые изменяют данные, используемые в условии цикла.

Внутри цикла могут присутствовать операторы `BREAK` и `CONTINUE`. Оператор `BREAK` означает завершение цикла, т. е. прекращение выполнения операторов цикла и переход на оператор, следующий за конечным ключевым словом `END`. Этот оператор не означает завершения выполнения хранимой процедуры, триггера. Когда встречается оператор `CONTINUE`, происходит переход на начало цикла, операторы после ключевого слова `CONTINUE` игнорируются.

ЗАМЕЧАНИЕ

Обратите внимание, что в Transact-SQL отсутствует оператор цикла `UNTIL`, при котором всегда выполняется хотя бы один раз тело цикла, а в конце выполняется соответствующая проверка. Меня это радует.

Оператор `GOTO`

Операторы внутри тела процедуры, триггера могут быть снабжены метками. *Метка* — это идентификатор (имя), после которого следует двоеточие:

`<метка>: <оператор>;`

Метки используются для того, чтобы именовать операторы, на которые может выполняться переход при использовании оператора `GOTO`:

`GOTO <метка>;`

После этого оператора управление передается оператору, которому присвоена эта метка. Программисты часто называют такой оператор "смерть структурному программированию". При грамотном использовании операторов ветвления и циклов потребность в операторе перехода не существует.

Оператор `RETURN`

Оператор `RETURN` осуществляет выход из хранимой процедуры, пользовательской функции, триггера (завершение выполнения).

`RETURN [<целое>];`

Хранимая процедура может возвращать целое число — код возврата. По умолчанию, если целое в операторе не указано, возвращается число 0. Код возврата может использоваться в вызывающей программе для анализа результатов выполнения хранимой процедуры. Пользовательская функция может возвращать значения и других типов данных, объявленных при описании функции.

Конструкция `TRY/CATCH`

Можно сказать, что это классическая конструкция, применяемая во многих современных языках программирования для перехвата и обработки ошибок. Эта конст-

рукция не может быть использована в функциях, определенных пользователем. Синтаксис конструкции:

```
BEGIN TRY  
    { <оператор> | <блок операторов> }  
END TRY  
BEGIN CATCH  
    { <оператор> | <блок операторов> }  
END CATCH;
```

Если в операторах блока TRY возникает какая-либо ошибка работы с базой данных с кодом серьезности более 10 (см. далее), то управление передается операторам обработки ошибок в блоке CATCH. Если ошибок нет, то блок CATCH не выполняется. Обработанные в блоке ошибки не передаются вызывающей программе.

В процессе обработки ошибок можно использовать следующие системные функции для получения подробных сведений об ошибке:

- ◆ ERROR_NUMBER() — возвращает номер ошибки (целое число), если вызвана из блока CATCH. Иначе возвращает NULL;
- ◆ ERROR_SEVERITY() — возвращает степень серьезности ошибки (целое число), если вызвана из блока CATCH. Иначе возвращает NULL;
- ◆ ERROR_STATE() — код состояния ошибки (целое число), если функция вызвана из блока CATCH. Иначе возвращает NULL;
- ◆ ERROR_PROCEDURE() — имя хранимой процедуры или триггера, где произошла ошибка. Если ошибка произошла вне процедуры или триггера, то возвращает NULL;
- ◆ ERROR_LINE() — номер строки в триггере или процедуре, где была обнаружена ошибка;
- ◆ ERROR_MESSAGE() — полный текст сообщения об ошибке (до 4000 символов), если функция вызвана из блока CATCH. Иначе возвращает NULL.

В примере 11.1 демонстрируется использование этих функций и глобальной переменной @@ERROR. Пример с небольшими изменениями взят из Books Online.

Пример 11.1. Использование функций сообщения об ошибке

```
USE BestDatabase;  
GO  
BEGIN TRY  
    SELECT 1 / 0;  
END TRY  
BEGIN CATCH  
    SELECT ERROR_NUMBER() AS 'NUMBER',  
        ERROR_SEVERITY() AS 'SEVERITY',  
        ERROR_STATE() AS 'STATE',  
        ERROR_PROCEDURE() AS 'PROC',  
        ERROR_LINE() AS 'LINE',
```

```

    ERROR_MESSAGE() AS 'MESSAGE',
    @@ERROR AS '@@ERROR';
END CATCH;
GO

```

Результатом выполнения этого пакета будет:

NUMBER	SEVERITY	STATE	PROC	LINE	MESSAGE	@@ERROR
8134	16	1	NULL	2	Devide by zero error encountered.	8134

Здесь мы видим, что функция `ERROR_NUMBER()` и переменная `@@ERROR` возвращают одно и то же значение. Степень серьезности ошибки 16, имя процедуры `NULL`, потому что ошибка возникает в пакете, а не в триггере или хранимой процедуре. Текст сообщения: "встретилась ошибка деления на ноль".

Оператор `THROW`

Оператор выдает исключение и передает управление блоку `CATCH` в конструкции `TRY/CATCH`. Синтаксис:

```
THROW [ <номер ошибки>, <сообщение>, <состояние> ];
```

`<номер ошибки>` — это номер ошибки, определенной пользователем. Значение может находиться в диапазоне от 50000 до 2147483647. Ошибки с номером менее 50000 являются системными ошибками.

`<сообщение>` — является строкой, длиной до 2048 символов.

`<состояние>` — число в диапазоне от 0 до 255.

11.2. Хранимые процедуры

Хранимая процедура создается в текущей базе данных в указанной схеме. К ней могут обращаться другие процедуры, триггеры, клиентские программы. Процедура может получать входные параметры и может возвращать выходные параметры. Хранимая процедура выполняется на сервере, что может существенно уменьшить сетевой трафик.

Также можно создавать временную хранимую процедуру, которая будет находиться в базе данных `tempdb`.

11.2.1. Создание хранимой процедуры

Для создания хранимой процедуры используется оператор `CREATE PROCEDURE`. Его синтаксис представлен в листинге 11.1 и графах 11.1—11.2.

Листинг 11.1. Синтаксис оператора `CREATE PROCEDURE`

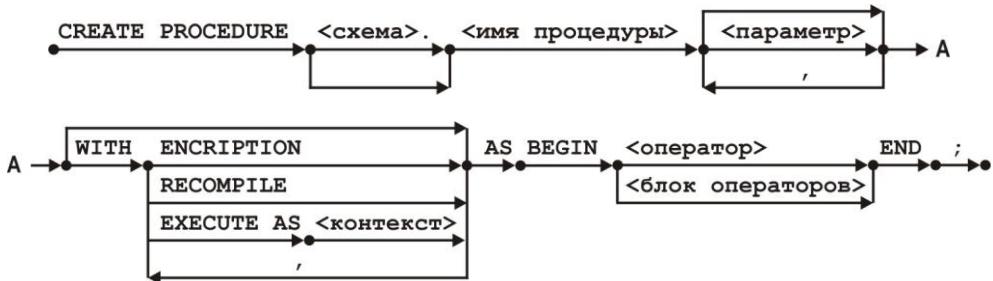
```

CREATE PROCEDURE [<имя схемы>.]<имя процедуры>
[ <параметр> [, <параметр> ]... ]

```

```
[ WITH <опция процедуры> [, <опция процедуры> ]... ]
AS BEGIN { <оператор> | <блок операторов> } END ;
<параметр> ::= <имя> [<имя схемы>.]<тип данных>
[ VARYING ] [ = <значение по умолчанию> ] [ OUTPUT ] [ READONLY ]

<опция процедуры> ::= { ENCRYPTION | RECOMPILE | EXECUTE AS <контекст> }
```



Граф 11.1. Синтаксис оператора CREATE PROCEDURE



Граф 11.2. Синтаксис задания параметра

В операторе наряду с ключевым словом PROCEDURE допустимо сокращение PROC.

Имя процедуры должно быть уникальным в данной схеме. Можно создать временную процедуру — локальную и глобальную. *Локальная процедура* видна только в текущем соединении с сервером базы данных. Имя локальной процедуры должно начинаться с символа #. *Глобальная временная процедура* видна всеми соединениями. Имя такой процедуры должно начинаться с двух символов ##.

Если имя схемы не указано, то хранимая процедура создается в схеме базы данных по умолчанию.

Для процедуры можно указать параметры. Имя параметра должно начинаться с символа @. Можно объявить до 2100 параметров. Их описания нужно отделять друг от друга запятыми. Должен быть указан тип данных параметра. Это может быть системный тип данных или пользовательский тип данных, определенный в указанной схеме.

Необязательное ключевое слово VARYING в описании параметра указывает, что в качестве выходного параметра будет использоваться результирующий набор. Применяется к типу данных CURSOR.

Для параметра можно указать значение по умолчанию, которое принимается, если при вызове процедуры значение для параметра не было задано. Значение по умолчанию помещается после знака равенства (=).

Ключевое слово `OUTPUT` (допустимо сокращение `OUT`) означает, что параметр является выходным. Его значение может быть использовано вызвавшей программой.

Ключевое слово `READONLY` означает, что значение параметра не может быть изменено внутри хранимой процедуры.

После ключевого слова `WITH` можно указать несколько параметров процедуры.

`ENCRYPTION` означает, что система преобразует, кодирует (в русском переводе Books Online, "затемняет", "запутывает") исходный текст процедуры. Другие пользователи не смогут просмотреть текст этой процедуры.

Ключевое слово `RECOMPILE` указывает, что при каждом вызове процедуры система будет перекомпилировать текст. В обычной ситуации система выполняет оптимизацию запроса на основании операторов процедуры. Скомпилированная процедура вместе с алгоритмом выборки данных сохраняется в кэше. Этим пользуются и другие вызовы данной хранимой процедуры. Если же произошли серьезные изменения в составе выбираемых данных, изменилась структура таблицы, появились новые индексы, которые могут повлиять на порядок выборки данных, то имеет смысл указать требование перекомпиляции, чтобы оптимизатор запросов сформировал более эффективный способ выборки данных.

Ключевые слова `EXECUTE AS` определяют контекст безопасности вызова процедуры, т. е., кто может вызывать данную хранимую процедуру.

Между ключевыми словами `BEGIN` и `END` (которые считаются необязательными, с чем не следует соглашаться) помещаются операторы, осуществляющие все действия хранимой процедуры.

Просмотреть список и характеристики хранимых процедур в Management Studio можно, раскрыв в Object Explorer базу данных, раскрыв папку **Programmability** и папку **Stored Procedures**. Для процедуры можно просмотреть ее параметры (папка **Parameters**), можно вывести ее текст, щелкнув правой кнопкой мыши по имени процедуры и выбрав в меню **Script Stored Procedure as | CREATE To | New Query Editor Window**. В новом окне появится текст создания процедуры.

11.2.2. Изменение хранимой процедуры

Изменение хранимой процедуры выполняется оператором `ALTER PROCEDURE`. Изменение означает полное пересоздание существующей процедуры. Синтаксис оператора практически полностью повторяет синтаксис оператора создания хранимой процедуры (листинг 11.2, графы 11.3—11.4).

Листинг 11.2. Синтаксис оператора `ALTER PROCEDURE`

```
ALTER PROCEDURE [<имя схемы>.]<имя процедуры>
[ <параметр> [ <параметр> ]... ]
[ WITH <опция процедуры> [, <опция процедуры> ]... ]
```

```

AS BEGIN { <оператор> | <блок операторов> } END ;
<параметр> ::= <имя> [<имя схемы>.]<тип данных>
[ VARYING ] [ = <значение по умолчанию> ] [ OUTPUT ] [ READONLY ]
<опция процедуры> ::= { ENCRYPTION | RECOMPILE | EXECUTE AS <контекст> }

```



Граф 11.3. Синтаксис оператора ALTER PROCEDURE



Граф 11.4. Синтаксис задания параметра

11.2.3. Удаление хранимой процедуры

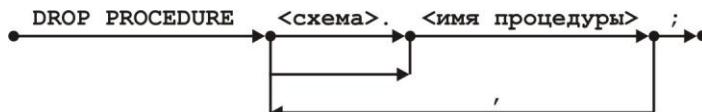
Для удаления процедуры используется оператор DROP PROCEDURE. Его синтаксис — в листинге 11.3 и графе 11.5.

Листинг 11.3. Синтаксис оператора DROP PROCEDURE

```

DROP PROCEDURE [<имя схемы>.]<имя процедуры>
[ , [<имя схемы>.]<имя процедуры>] ... ;

```



Граф 11.5. Синтаксис оператора DROP PROCEDURE

В одном операторе можно удалить несколько хранимых процедур, разделяя их имена запятыми.

ЗАМЕЧАНИЕ

Помимо использования операторов Transact-SQL с хранимыми процедурами можно выполнять действия и при помощи средств, предоставляемых Management Studio. Однако назвать эти средства действительно диалоговыми язык не поворачивается. При создании или изменении хранимой процедуры вы получаете обычное окно, в котором средствами Transact-SQL создаете или изменяете хранимую процедуру.

11.2.4. Использование хранимых процедур

Для вызова хранимой процедуры используется оператор EXECUTE (допустимо сокращение EXEC). Здесь задается имя вызываемой процедуры и список значений передаваемых процедуре параметров. Если параметр является выходным, то он задается в виде имени локальной переменной, за которым должно идти ключевое слово OUTPUT (OUT). Если процедура возвращает целочисленное значение при помощи оператора RETURN, то в операторе EXECUTE перед именем процедуры нужно записать имя локальной переменной, которой будет присвоено возвращаемое значение, и знак равенства.

Рассмотрим пример простой хранимой процедуры, которую можно было бы использовать для получения уникального значения, присваиваемого первичному или уникальному ключу в таблице.

Вначале создадим хранимую процедуру PROC_PEOPLE (пример 11.2).

Пример 11.2. Создание хранимой процедуры PROC_PEOPLE

```
USE BestDatabase;
GO
IF OBJECT_ID('PROC_PEOPLE', 'P') IS NOT NULL
    DROP PROCEDURE PROC_PEOPLE;
GO
CREATE PROCEDURE PROC_PEOPLE
    @COD INT OUTPUT
AS
BEGIN
    SET @COD = (SELECT MAX(COD) FROM PEOPLE);
END;
GO
```

Здесь вначале проверяется, существует ли процедура PROC_PEOPLE и при необходимости она удаляется. Затем создается сама процедура. В ней описывается один выходной целочисленный параметр.

В процедуре выполняется единственное действие. Выходному параметру присваивается максимальное существующее в базе данных значение первичного ключа в таблице PEOPLE.

Следует помнить, что создание хранимой процедуры должно быть единственным оператором в пакете. Предыдущая проверка существования в базе данных процедуры должна завершаться оператором GO. После него начинается новый пакет.

Обращение к этой процедуре можно выполнить следующим образом (пример 11.3).

Пример 11.3. Обращение к процедуре PROC_PEOPLE

```
USE BestDatabase;
GO
DECLARE @NEWCOD AS INT;
EXEC PROC_PEOPLE @NEWCOD OUTPUT;
PRINT @NEWCOD;
GO
```

В результате будет отображено число 112. Это действительно максимальное значение первичного ключа в таблице PEOPLE.

В этом примере мы использовали оператор PRINT, который позволяет отобразить одиночный текст.

Теперь несколько слов о разумности использования такой процедуры. Если мы собираемся ее применять для того, чтобы вручную формировать уникальное значение для столбца первичного ключа, то нужно быть готовыми к неприятным сюрпризам. Если у нас ровно один пользователь, работающий с этой базой данных, то нет никаких проблем. Каждый раз мы будем получать максимальное значение ключа, увеличивать его на единицу и присваивать полученное значение вновь создаваемой строке людей. Однако если к базе данных одновременно подключено несколько пользователей и некоторые из них в одно и то же время решили создать новую запись, используя данную процедуру для получения значения первичного ключа, то уникальности нам никто не гарантирует. Несколько пользователей могут получить одно и то же значение, что приведет к потере уникальности при попытке поместить в базу данных новую запись.

В подобном случае лучше использовать стандартные средства системы, т. е. автономный инкрементный ключ. Есть, правда, еще вариант получения уникального целочисленного значения при использовании так называемых последовательностей (SEQUENCE).

Сначала в базе данных создается последовательность. Для получения очередного значения используется конструкция NEXT VALUE FOR. Для иллюстрации использования последовательностей рассмотрим пример 11.4.

Пример 11.4. Создание и использование последовательности

```
USE BestDatabase;
GO
CREATE SEQUENCE UNIQUE_KEY;
GO
SELECT NEXT VALUE FOR UNIQUE_KEY;
SELECT NEXT VALUE FOR UNIQUE_KEY;
```

Создается последовательность со всеми значениями по умолчанию. При каждом выборе из нее значений мы получаем уникальное число.

Обращение к последовательностям осуществляется вне контекста какой-либо транзакции. По этой причине разные процессы, выполняемые одновременно, получат все равно различные значения.

Теперь рассмотрим более полезную процедуру, которая позволяет получить факториал целого числа — $n!$ (пример 11.5).

Пример 11.5. Создание хранимой процедуры FACTORIAL

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORYAL', 'P') IS NOT NULL
    DROP PROCEDURE FACTORYAL;
GO
CREATE PROCEDURE FACTORYAL
    @N INT,
    @NFACTORYAL INT OUTPUT
AS
BEGIN
    DECLARE @I INT = 1;
    SET @NFACTORYAL = 1;
    WHILE @I <= @N
        BEGIN
            SET @NFACTORYAL = @NFACTORYAL * @I;
            SET @I = @I + 1;
        END;
    END;
GO
```

Для получения значения факториала можно использовать следующие операторы (пример 11.6).

Пример 11.6. Вычисление факториала числа

```
USE BestDatabase;
GO
DECLARE @FACTORYAL AS INT;
EXEC FACTORYAL 5, @FACTORYAL OUTPUT;
PRINT @FACTORYAL;
GO
```

Первым параметром при вызове процедуры мы задаем исходное значение, целое число, для которого хотим получить факториал. Однако при задании для выходного

параметра типа данных INT мы можем получать факториал только для чисел, не превышающих 12. Чтобы иметь возможность работать и с большими значениями, мы можем использовать типы данных BIGINT, DECIMAL и даже FLOAT.

Давайте в процедуре зададим проверку на наличие ошибок (превышение допустимого значения исходного числа) с использованием конструкции TRY/CATCH.

Создайте несколько измененный вариант процедуры (пример 11.7).

Пример 11.7. Создание другой версии хранимой процедуры FACTORIAL2

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORIZATION', 'P') IS NOT NULL
    DROP PROCEDURE FACTORIZATION;
GO
CREATE PROCEDURE FACTORIZATION
    @N INT,
    @NFACTORIZATION INT OUTPUT
AS
BEGIN
    DECLARE @I INT = 1;
    SET @NFACTORIZATION = 1;
    BEGIN TRY
        WHILE @I <= @N
        BEGIN
            SET @NFACTORIZATION = @NFACTORIZATION * @I;
            SET @I = @I + 1;
        END;
    END TRY
    BEGIN CATCH
        PRINT 'Исходное значение превышает 12. Код ошибки: ' +
              CAST(ERROR_NUMBER() AS VARCHAR(10)) +
              '. Сообщение: ' + ERROR_MESSAGE();
        SET @NFACTORIZATION = -1;
    END CATCH;
END;
GO
```

При обращении к этой процедуре с исходным параметром, значение которого превышает допустимое число 12, мы получим такое сообщение:

Исходное значение превышает 12. Код ошибки 8115. Сообщение: Arithmetic overflow error converting expression to data type int.

Теперь создадим похожую процедуру, которая будет возвращать то же решение, но другим способом (пример 11.8).

Пример 11.8. Создание хранимой процедуры FACTORIAL3

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORYAL3', 'P') IS NOT NULL
    DROP PROCEDURE FACTORYAL3;
GO
CREATE PROCEDURE FACTORYAL3
    @N INT
AS
BEGIN
    DECLARE @I INT = 1;
    DECLARE @NFACTORYAL INT;
    SET @NFACTORYAL = 1;
    BEGIN TRY
        WHILE @I <= @N
        BEGIN
            SET @NFACTORYAL = @NFACTORYAL * @I;
            SET @I = @I + 1;
        END;
    END TRY
    BEGIN CATCH
        PRINT 'Исходное значение превышает допустимое. Код ошибки: ' +
              CAST(ERROR_NUMBER() AS VARCHAR(10)) +
              '. Сообщение: ' + ERROR_MESSAGE();
        SET @NFACTORYAL = -1;
    END CATCH;
    RETURN @NFACTORYAL;
END;
GO
```

Процедура возвращает значение при помощи оператора `RETURN`. Обращение к такой процедуре показано в примере 11.9.

Пример 11.9. Обращение к процедуре FACTORYAL3

```
USE BestDatabase;
GO
DECLARE @FACTORYAL AS INT;
EXEC @FACTORYAL = FACTORYAL 5;
PRINT 'Значение = ' + CAST(@FACTORYAL AS VARCHAR(10));
GO
```

Перед именем процедуры в операторе `EXECUTE` указывается имя переменной, куда будет помещаться возвращаемое функцией значение.

Мы можем использовать рекурсивные процедуры, т. е. процедуры, которые могут обращаться к самим себе. Однако уровень вложенности таких процедур не может превышать 32. Для примера — в InterBase уровень вложенности 1000.

Создание рекурсивной процедуры показано в примере 11.10. Здесь я только убрал обработку ошибок. Пусть вас не смущает номер процедуры 6. Я для собственного удовольствия создавал еще несколько процедур, которые может быть и не нужно здесь показывать.

Пример 11.10. Создание рекурсивной хранимой процедуры FACTORIAL6

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORIAL6', 'P') IS NOT NULL
    DROP PROCEDURE FACTORIAL6;
GO
CREATE PROCEDURE FACTORIAL6
    @N INT,
    @NFACTORIAL FLOAT(8) OUT
AS
BEGIN
    DECLARE @I INT = 1;
    SET @NFACTORIAL = 1;
    IF @I = 1 /****** Простейший случай: 1! = 1 *****/
        RETURN;
    /**** Рекурсия: @N! = (@N * (@N-1))! ****/
    SET @I = @N - 1;
    EXEC FACTORIAL @I, @NFACTORIAL OUT;
    SET @NFACTORIAL = @NFACTORIAL * @N;
END;
GO
```

Кстати, этот пример процедуры я взял с некоторыми изменениями из документации по InterBase.

Другие примеры использования хранимых процедур вы можете найти в Books Online.

Многие действия, совершаемые в хранимых процедурах, можно осуществить и при использовании функций, определенных пользователем.

11.3. Функции, определенные пользователем

Функция, определенная пользователем (User Defined Function, UDF), также является программой, которая может получать параметры и возвращать значение. Функ-

цию можно применять в качестве обычной переменной, чье значение используется для вычислений или для отображения результата.

11.3.1. Создание функции

Для создания определенной пользователем функции используется оператор `CREATE FUNCTION`. Его несколько упрощенный синтаксис представлен в листинге 11.4 и графах 11.6—11.7.

Листинг 11.4. Синтаксис оператора `CREATE FUNCTION`

```

CREATE FUNCTION [<имя схемы>.]<имя функции>
  ([ <параметр> [, <параметр> ]... ])
RETURNS <возвращаемый тип данных>
  [ WITH <опция функции> [, <опция функции> ]... ]
AS BEGIN { <оператор> | <блок операторов> } RETURN <значение> END ;
  
```

<параметр> ::= <имя> [AS] [<имя схемы>.]<тип данных>
 [= <значение по умолчанию>] [READONLY]



Граф 11.6. Синтаксис оператора `CREATE FUNCTION`



Граф 11.7. Синтаксис описания параметра

После имени функции (имя должно быть уникальным среди имен функций заданной схемы базы данных) задается список параметров, заключенных в круглые скобки и разделенные запятыми. Если функции не передаются параметры, то скоб-

ки все равно должны быть указаны. Хочу сказать, что с эстетической точки зрения обязательное наличие скобок и при отсутствии параметров является хорошим решением и общепринятой нормой в программировании.

При описании передаваемых функции параметров указывается тип данных параметра. Можно указать значение по умолчанию, если при обращении к функции параметр не передается. Необязательное ключевое слово `READONLY` означает, что внутри функции значение параметра не может изменяться. Мне эта возможность не совсем понятна, потому что любые изменения значения параметра внутри функции никак неказываются на начальном значении.

После необязательного ключевого слова `WITH` указывается список опций функции, которые я не хочу рассматривать в данном документе.

Блок операторов должен завершаться оператором `RETURN`, который возвращает то самое значение, для получения которого вызывалась функция.

11.3.2. Изменение функций

Для изменения функции, определенной пользователем, используется оператор `ALTER FUNCTION`, синтаксис которого почти полностью копирует синтаксис оператора создания пользовательской функции. Синтаксис оператора `ALTER FUNCTION` показан в листинге 11.5 и графах 11.8—11.9.

Листинг 11.5. Синтаксис оператора `ALTER FUNCTION`

```

ALTER FUNCTION [<имя схемы>.]<имя функции>
  ([ <параметр> [, <параметр> ]... ])
RETURNS <возвращаемый тип данных>
  [ WITH <опция функции> [, <опция функции> ]... ]
AS BEGIN { <оператор> | <блок операторов> } RETURN <значение> END ;
  
```

<параметр> ::= <имя> [AS] [<имя схемы>.]<тип данных>
 [= <значение по умолчанию>] [READONLY]



Граф 11.8. Синтаксис оператора `ALTER FUNCTION`



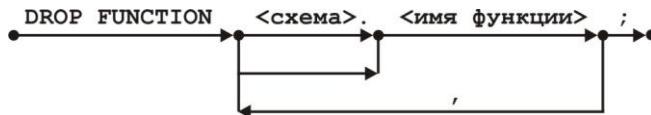
Граф 11.9. Синтаксис описания параметра

11.3.3. Удаление функций

Для удаления одной или более функций используется оператор `DROP FUNCTION`, в котором через запятую перечисляются имена удаляемых функций (листинг 11.6, граф 11.10).

Листинг 11.6. Синтаксис оператора `DROP FUNCTION`

```
DROP FUNCTION [<имя схемы>.]<имя функции>
[ , [<имя схемы>.]<имя функции>] ... ;
```

Граф 11.10. Синтаксис оператора `DROP FUNCTION`

11.3.4. Использование функций

Рассмотрим простой пример функции. Это опять тот же факториал целого числа. Создание функции `FACTORIAL7` показано в примере 11.11.

Пример 11.11. Создание хранимой процедуры `FACTORYAL7`

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORYAL7', 'FN') IS NOT NULL
    DROP FUNCTION FACTORYAL7;
GO
CREATE FUNCTION FACTORYAL7
    (@N AS INT)
RETURNS FLOAT(8)
AS
BEGIN
    DECLARE @I INT = 1;
    DECLARE @NFACTORYAL FLOAT(8);
```

```
WHILE @I <= @N
BEGIN
    SET @NFACTORIAL = @NFACTORIAL * @I;
    SET @I = @I + 1;
END;
RETURN @NFACTORIAL;
END;
GO
```

Обращение к функции показано в примере 11.12.

Пример 11.12. Обращение к функции FACTORIAL7

```
USE BestDatabase;
GO
DECLARE @FACTORIAL AS FLOAT(8);
SET @FACTORIAL = dbo.FACTORIAL7(12);
PRINT 'Значение = ' + CAST(@FACTORIAL AS VARCHAR(50));
GO
```

При обращении к функции обязательно нужно указывать имя схемы.

11.4. Триггеры

Триггер — это программный объект базы данных, который выполняется на стороне сервера. Во многих случаях это позволяет повысить производительность системы.

Напрямую обратиться к триггеру невозможно. Он вызывается автоматически при наступлении соответствующего события базы данных — добавление новой строки в таблицу, изменение или удаление строки. Триггер может срабатывать, когда соответствующее действие с базой данных выполняет клиентское приложение, хранящая процедура или триггер (другой или тот же самый).

Триггер выполняется в контексте той транзакции, под управлением которой выполняется и программа, инициировавшая вызов триггера.

Существуют три вида триггеров, которые отличаются по функциям и по синтаксису создания и изменения — триггеры языка манипулирования данными DML, триггеры языка описания данных DDL и триггеры входа в систему.

Триггеры DML вызываются при выполнении операторов `INSERT`, `UPDATE` или `DELETE`. Можно указать время вызова триггера:

- ◆ `AFTER` — триггер вызывается после всех действий оператора, если оператор был выполнен успешно. Синонимом в синтаксисе оператора создания триггера является `FOR`;
- ◆ `INSTEAD OF` — триггер вызывается вместо действий, заданных оператором.

11.4.1. Создание триггеров

Синтаксис оператора создания триггера DML CREATE TRIGGER показан в листинге 11.7 и графе 11.11.

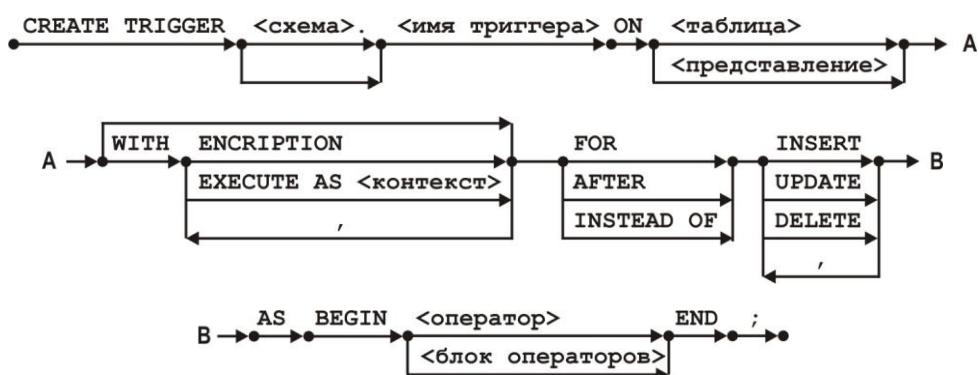
Листинг 11.7. Синтаксис оператора создания триггера DML

```

CREATE TRIGGER [<имя схемы>.]<имя триггера>
  ON { <таблица> | <представление> }
    [ WITH <опция триггера> [, <опция триггера>]... ]
    { FOR | AFTER | INSTEAD OF }
      <операция> [ , <операция> ] ...
AS BEGIN { <оператор> | <блок операторов> } END;

<опция триггера> ::= { ENCRYPTION | EXECUTE AS <контекст> }

<операция> ::= INSERT | UPDATE | DELETE
  
```



Граф 11.11. Синтаксис оператора создания триггера DML

Имя триггера не может превышать 128 символов и должно быть уникальным в данной схеме базы данных.

После ключевого слова ON указывается имя таблицы или представления, для которого создается триггер.

После необязательного ключевого слова WITH может быть задана одна или две опции. ENCRYPTION указывает, что текст триггера должен кодироваться при помещении в базу данных. EXECUTE AS определяет контекст безопасности обращения к триггеру.

Ключевые слова FOR и AFTER (напоминаю, это синонимы) означают, что к триггеру происходит обращение *после выполнения изменений базы данных и после проверки соответствия изменяемых, удаляемых или добавляемых данных декларативной целостности данных в базе*. Такой триггер сам может рекурсивно вызвать себя,

если в нем применяется операция, вызвавшая этот триггер. Однако глубина вложенности не может превышать магического числа 32, как и в случае с хранимыми процедурами. Если задано `INSTEAD OF`, то указанные действия в самом операторе, вызвавшем неявно триггер, не выполняются, а будут выполнены только действия триггера. Важен здесь такой момент. Если в триггере `INSTEAD OF` выполняется та же самая операция, по которой был запущен на выполнение этот триггер, то дополнительный вызов триггера не произойдет. Чуть позже мы рассмотрим соответствующий пример.

Для одной операции в одной таблице может быть задано несколько триггеров `INSTEAD OF` и несколько триггеров `AFTER`. Вначале запускаются триггеры `INSTEAD OF`, а затем триггеры `AFTER`.

Триггеры могут быть созданы для реагирования на любую операцию или группу операций: добавление данных (`INSERT`), изменение (`UPDATE`) и удаление (`DELETE`). А вот при выполнении операции усечения таблицы (`TRUNCATE TABLE`) никакой триггер вызываться не будет.

Если в операции добавления, изменения или удаления задействовано более одной строки, то триггеры будут вызываться не для каждой строки, а для всех включенных в операцию строк.

В триггерах DML возможно обращение к так называемым логическим или концептуальным таблицам. К таблице `deleted` может обращаться триггер, запускаемый при удалении строк таблицы. Эта логическая таблица содержит все строки, удаляемые вызвавшим триггер оператором. Похожим образом таблица `inserted` содержит все добавляемые строки в основную таблицу. Все они являются временными и располагаются в оперативной памяти. Эти логические таблицы в зависимости от вида соответствующего оператора могут содержать более одной строки.

Может несколько удивить отсутствие логической таблицы с именем, скажем, `updated`, которая содержала бы данные, измененные оператором `UPDATE`. Однако все необходимые данные находятся в таблицах `deleted` и `inserted`. Таблица `deleted` содержит данные до их изменения оператором, а таблица `inserted` — данные после их изменения. Соответствующий пример мы сегодня рассмотрим.

Синтаксис оператора создания триггера DDL показан в листинге 11.8 и графе 11.12.

Листинг 11.8. Синтаксис оператора создания триггера DDL

```
CREATE TRIGGER <имя триггера>
    ON { ALL SERVER | DATABASE }
    [ WITH <опция триггера> [, <опция триггера>]... ]
    { FOR | AFTER } { <событие> | <группа событий> }
        [ { <событие> | <группа событий> } ] ...
    AS BEGIN { <оператор> | <блок операторов> } END;

<опция триггера> ::= { ENCRYPTION | EXECUTE AS <контекст> }
```



Граф 11.12. Синтаксис оператора создания триггера DDL

После ключевого слова **ON** указывается, какова область действия триггера. Если задано **DATABASE**, то триггер будет срабатывать при наступлении одного из перечисленных далее событий текущей базы данных. При указании **ALL SERVER** триггер срабатывает при наступлении события для любой базы данных текущего экземпляра сервера.

После ключевого слова **FOR** или **AFTER** указываются одиночные события или группы похожих событий, при которых вызывается триггер. Все задаваемые события и их группы связаны с операторами DDL.

Одиночные события — это выполнение операций создания, изменения или удаления отдельных объектов базы данных: таблиц, индексов, пользовательских типов данных и многоного другого.

Группы событий, как правило, связаны с различными операциями по отношению к одному типу объектов базы данных. Например, есть группа событий, которая относится к операциям создания, изменения и удаления таблиц.

Полное описание событий и групп событий см. в Books Online.

Последний тип триггера — триггер входа. Синтаксис создания такого триггера представлен в листинге 11.9 и графике 11.13.

Листинг 11.9. Синтаксис оператора создания триггера входа

```

CREATE TRIGGER <имя триггера>
  ON ALL SERVER
  [ WITH <опция триггера> [, <опция триггера>]... ]
  { FOR | AFTER } LOGON
  AS BEGIN { <оператор> | <блок операторов> } END;

<опция триггера> ::= { ENCRYPTION | EXECUTE AS <контекст> }
  
```

Триггер входа запускается при соединении пользователя с экземпляром сервера. Его можно использовать для дополнительной проверки полномочий пользователей.



11.4.2. Изменение триггеров

Для изменения всех видов триггеров используется оператор ALTER TRIGGER. Синтаксис оператора для каждого вида триггера показан в листингах 11.10—11.12 и графах 11.14—11.16.

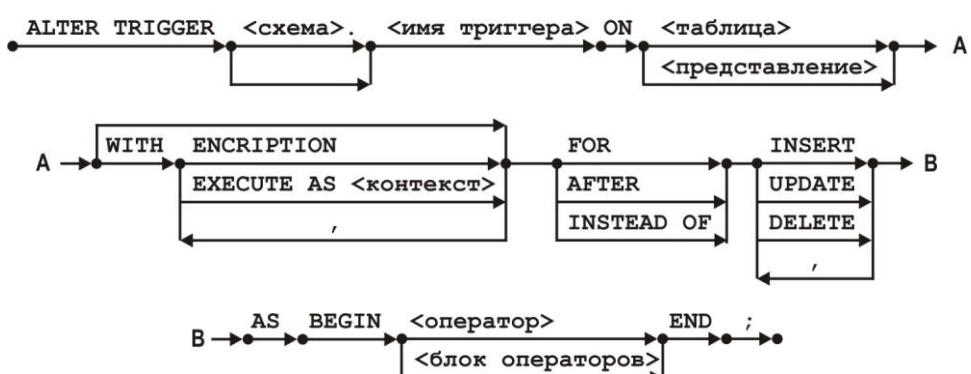
Листинг 11.10. Синтаксис оператора изменения триггера DML

```

ALTER TRIGGER [<имя схемы>.]<имя триггера>
    ON { <таблица> | <представление> }
    [ WITH <опция триггера> [, <опция триггера>]... ]
    { FOR | AFTER | INSTEAD OF }
    <операция> [ , <операция> ] ...
AS BEGIN { <оператор> | <блок операторов> } END;

<опция триггера> ::= { ENCRYPTION | EXECUTE AS <контекст> }

<операция> ::= INSERT | UPDATE | DELETE
  
```



Граф 11.14. Синтаксис оператора изменения триггера DML

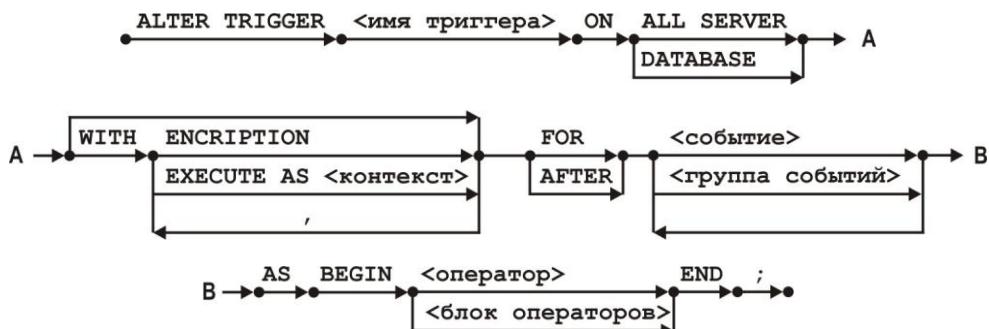
Листинг 11.11. Синтаксис оператора изменения триггера DDL

```

ALTER TRIGGER <имя триггера>
ON { ALL SERVER | DATABASE }
[ WITH <опция триггера> [, <опция триггера>]... ]
{ FOR | AFTER } { <событие> | <группа событий> }
[ { <событие> | <группа событий> } ] ...
AS BEGIN { <оператор> | <блок операторов> } END;

<опция триггера> ::= { ENCRYPTION | EXECUTE AS <контекст> }

```



Граф 11.15. Синтаксис оператора изменения триггера DDL

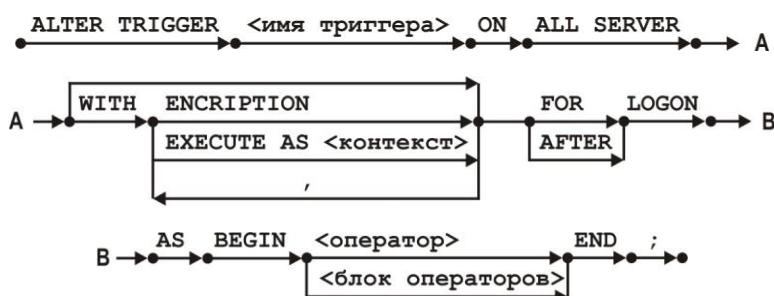
Листинг 11.12. Синтаксис оператора изменения триггера входа

```

ALTER TRIGGER <имя триггера>
ON ALL SERVER
[ WITH <опция триггера> [, <опция триггера>]... ]
{ FOR | AFTER } LOGON
AS BEGIN { <оператор> | <блок операторов> } END;

<опция триггера> ::= { ENCRYPTION | EXECUTE AS <контекст> }

```



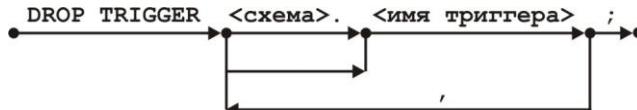
Граф 11.16. Синтаксис оператора изменения триггера входа

11.4.3. Удаление триггеров

Для удаления триггеров используется оператор `DROP TRIGGER`. Синтаксис оператора для каждого вида триггера показан в листингах 11.13—11.15 и графах 11.17—11.19.

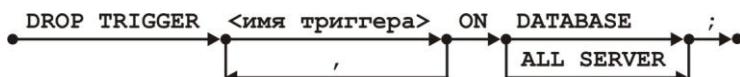
Листинг 11.13. Синтаксис оператора удаления триггера DML

```
DROP TRIGGER [<имя схемы>.]<имя триггера>
[, [<имя схемы>.]<имя триггера>]...;
```



Листинг 11.14. Синтаксис оператора удаления триггера DDL

```
DROP TRIGGER <имя триггера> [, <имя триггера>]...
ON { DATABASE | ALL SERVER } ;
```



Граф 11.18. Синтаксис оператора удаления триггера DDL

Листинг 11.15. Синтаксис оператора удаления триггера входа

```
DROP TRIGGER <имя триггера> [, <имя триггера>]...
ON ALL SERVER ;
```



Граф 11.19. Синтаксис оператора удаления триггера входа

Завершим на этом рассмотрение синтаксических конструкций работы с триггерами. Давайте разберем примеры из реальной жизни использования триггеров.

11.4.4. Использование триггеров

Рассмотрим пару ситуаций, когда триггеры будут действительно полезны.

Первое, огорчительное для меня, это когда в таблице людей при удалении одной из строк нельзя было установить в значение `NULL` внешние ключи для матери, отца или

супруга. Эта нехорошая ситуация (точнее, ошибка системы) может быть разрешена при использовании триггера INSTEAD OF.

Таблица людей создается следующим оператором:

```
CREATE TABLE PEOPLE
( COD      INTEGER IDENTITY(1, 1)
  NOT NULL,      /* Код человека */
  NAME1     VARCHAR(15),      /* Имя */
  NAME2     VARCHAR(15),      /* Отчество */
  NAME3     VARCHAR(20),      /* Фамилия */
  BIRTHDAY  DATE,            /* Дата рождения */
  SEX       CHAR(1) DEFAULT '0', /* Пол: */
                    /* 0 — мужской, */
                    /* 1 — женский. */
  FULLNAME AS              /* Вычисляемый столбец */
    (NAME3 + ' ' + NAME1 + ' ' + NAME2),
  CODMOTHER  INTEGER
    DEFAULT NULL,           /* Ссылка на мать */
  CODFATHER  INTEGER
    DEFAULT NULL,           /* Ссылка на отца */
  CODOTHERHALF INTEGER
    DEFAULT NULL,           /* Ссылка на супруга */
  CONSTRAINT PK_PEOPLE PRIMARY KEY (COD),
  CONSTRAINT CH_PEOPLE CHECK (SEX IN ('0', '1')),
  CONSTRAINT FK1_PEOPLE
    FOREIGN KEY (CODMOTHER) REFERENCES PEOPLE (COD)
    ON DELETE NO ACTION,
  CONSTRAINT FK2_PEOPLE
    FOREIGN KEY (CODFATHER) REFERENCES PEOPLE (COD)
    ON DELETE NO ACTION,
  CONSTRAINT FK3_PEOPLE
    FOREIGN KEY (CODOTHERHALF) REFERENCES PEOPLE (COD)
    ON DELETE NO ACTION
);
GO
```

Поскольку при удалении любого человека из таблицы невозможно декларативно указать, что три внешних ключа в других строках, ссылающихся на первичный ключ удаляемой строки, должны получить значение NULL, приходится создавать триггер INSTEAD OF.

Создание триггера показано в примере 11.13.

Пример 11.13. Создание триггера установки в значение NULL внешних ключей

```
USE BestDatabase;
GO
IF OBJECT_ID('PEOPLE_DELETE', 'TR') IS NOT NULL
  DROP TRIGGER PEOPLE_DELETE;
```

```

GO
CREATE TRIGGER dbo.PEOPLE_DELETE
    ON PEOPLE INSTEAD OF DELETE
AS
BEGIN
    SET NOCOUNT ON;
    UPDATE P SET P.CODMOTHER = NULL
        FROM PEOPLE AS P
        INNER JOIN deleted AS D ON P.COD = D.CODMOTHER;
    UPDATE P SET P.CODFATHER = NULL
        FROM PEOPLE AS P
        INNER JOIN deleted AS D ON P.COD = D.CODFATHER;
    UPDATE P SET P.CODOITHERHALF = NULL
        FROM PEOPLE AS P
        INNER JOIN deleted AS D ON P.COD = D.CODOITHERHALF;
    DELETE FROM PEOPLE WHERE COD IN (SELECT COD FROM deleted);
END;
GO

```

Здесь я использую вариант, структуру которого как-то предложил Аарон Берtrand (Aaron Bertrand). Это работает, однако немного удивляет использование внутренне-го соединения для определения изменяемых строк. Есть вариант более простой:

```

UPDATE PEOPLE SET CODMOTHER = NULL
    WHERE CODMOTHER IN (SELECT COD FROM deleted);
UPDATE PEOPLE SET CODFATHER = NULL
    WHERE CODFATHER IN (SELECT COD FROM deleted);
UPDATE PEOPLE SET CODOITHERHALF = NULL
    WHERE CODOITHERHALF IN (SELECT COD FROM deleted);

```

Рассмотрим еще один пример триггера, который будет создавать таблицу истории окладов сотрудников организаций. Таблица была создана следующим оператором:

```

CREATE TABLE STAFFHISTORY
( COD          D_INTEGER IDENTITY(1, 1)
            NOT NULL,      /* Код истории — первичный ключ */
  CODSTAFF     D_INTEGER,           /* Код сотрудника */
  SALARY       D_DECIMAL,          /* Оклад */
  DATESALARY   D_DATE,            /* Дата изменения оклада */
  CONSTRAINT PK_STAFFHISTORY PRIMARY KEY (COD),
  CONSTRAINT FK_STAFFHISTORY
    FOREIGN KEY (CODSTAFF) REFERENCES STAFF (COD)
    ON DELETE CASCADE
);

```

Каждый раз после изменения оклада сотрудника мы должны внести новую строку в таблицу истории окладов, указав старое значение оклада и дату, до которой существовал этот оклад.

Создание триггера формирования истории показано в примере 11.14.

Пример 11.14. Триггер создания истории

```
USE BestDatabase;
GO
IF OBJECT_ID('UPDATE_STAFF', 'TR') IS NOT NULL
DROP TRIGGER UPDATE_STAFF;
GO
CREATE TRIGGER dbo.UPDATE_STAFF
ON STAFF AFTER UPDATE
AS
IF UPDATE (SALARY)
BEGIN
SET NOCOUNT ON;
INSERT INTO STAFFHISTORY (CODSTAFF, SALARY, DATESALARY)
(SELECT COD, SALARY, GETDATE() FROM deleted);
END;
GO
```

Это триггер AFTER UPDATE. После выполнения изменений в таблице сотрудников запускается триггер UPDATE_STAFF. Он будет выполнять действия, только если был изменен оклад сотрудника. Для этого в начале триггера стоит условие IF UPDATE (SALARY) .

ПРИЛОЖЕНИЕ 1

12 правил Кодда

Здесь я повторю те описания, которые в любой литературе по базам данных даются по поводу 12 правил Кодда. Доктор И. Ф. Кодд предложил 13 правил определения реляционных систем. Их обычно называют "12 правилами доктора Кодда".

Правило 0 — фундаментальное правило

Любая реляционная СУБД должна быть способна управлять данными исключительно с помощью реляционных функций.

Правило означает, что в СУБД не должны применяться какие-либо нереляционные операции для определения данных и манипулирования ими.

Правило 1 — представление данных

Все данные в реляционной базе данных представляются в явном виде на логическом уровне и только одним способом — в виде значений в таблицах.

Все данные, включая метаданные, должны храниться в виде отношений и управляться с помощью тех же функций, которые используются для работы с данными.

Правило 2 — гарантированный доступ

Для каждого элемента данных должен быть гарантирован логический доступ на основе использования комбинации имени таблицы, значения первичного ключа и имени столбца.

Правило 3 — обработка неопределенных значений (NULL)

Неопределенные значения являются способом представления отсутствующих или неприемлемых данных независимо от типа данных.

Неопределенные значения — значения, отличные от пустой строки, строки с пробельными символами, а также от нуля или любого другого числа.

Правило 4 — динамический каталог, основанный на реляционной модели

Описание данных должно быть представлено на логическом уровне таким же образом, что и обычные данные.

Правило дает возможность пользователям для обращения к этому описанию применять тот же реляционный язык, что и при обращении к обычным данным.

Правило 5 — исчерпывающий подъязык данных

Реляционная система может поддерживать несколько языков. Однако должен существовать по крайней мере один язык, который позволял бы выражать следующие конструкции:

- 1) определение данных;
- 2) определение представлений;
- 3) операторы манипулирования данными;
- 4) ограничения целостности;
- 5) авторизация пользователей;
- 6) поэтапная организация транзакций.

Правило 6 — обновление представлений

Все представления, которые являются теоретически обновляемыми, должны быть обновляемыми в данной системе.

Правило 7 — высокоуровневые операции добавления, обновления и удаления

Способность обрабатывать базовые и производные (т. е. представления) таблицы в виде отдельного оператора на языке высокого уровня.

Правило 8 — физическая независимость от данных

Прикладные программы и средства работы с терминалами должны оставаться логически неизменными при внесении любых изменений в способы хранения данных или методы доступа к ним.

Правило 9 — логическая независимость от данных

Прикладные программы и средства работы с терминалами должны оставаться логически неизменными при внесении в базовые таблицы любых не меняющих данные изменений, которые теоретически не должны затрагивать прикладное программное обеспечение.

Правило 10 — независимость ограничений целостности

Ограничения целостности данных должны определяться на языке реляционных данных и храниться в базе данных, а не в прикладных программах.

Правило 11 — независимость от распределения данных

Язык манипулирования данными должен позволять прикладным программам и запросам оставаться логически неизменными, независимо от того, как хранятся данные — физически централизованно или в распределенном виде.

Правило 12 — запрет обходных путей

Если система имеет язык низкого уровня, он не может быть использован для отмены или обхода правил и ограничений целостности, составленных на языке более высокого уровня.

На языке низкого уровня можно обрабатывать одновременно одну запись. Язык высокого уровня одновременно обрабатывает сразу несколько записей.

ПРИЛОЖЕНИЕ 2

Зарезервированные слова Transact-SQL

В табл. П2.1 представлен список зарезервированных слов. Здесь присутствуют слова, являющиеся зарезервированными в текущей версии SQL Server, и слова, которые могут стать зарезервированными в ближайших будущих версиях системы.

Не используйте эти слова для именования объектов вашей базы данных.

Таблица П2.1. Зарезервированные слова Transact-SQL

ABSOLUTE	AUTHORIZATION	CAST
ACTION	BACKUP	CATALOG
ADD	BEFORE	CHAR
ADMIN	BEGIN	CHARACTER
AFTER	BETWEEN	CHECK
AGGREGATE	BINARY	CHECKPOINT
ALIAS	BIT	CLASS
ALL	BLOB	CLOB
ALLOCATE	BOOLEAN	CLOSE
ALTER	BOTH	CLUSTERED
AND	BREADTH	COALESCE
ANY	BREAK	COLLATE
ARE	BROWSE	COLLATION
ARRAY	BULK	COLLECT
AS	BY	COLUMN
ASC	CALL	COMMIT
ASENSITIVE	CALLED	COMPLETION
ASSERTION	CARDINALITY	COMPUTE
ASYMMETRIC	CASCADE	CONDITION
AT	CASCADED	CONNECT
ATOMIC	CASE	CONNECTION

Таблица П2.1 (продолжение)

CONSTRAINT	DEC	ESCAPE
CONSTRAINTS	DECIMAL	EVERY
CONSTRUCTOR	DECLARE	EXCEPT
CONTAINS	DEFAULT	EXCEPTION
CONTAINSTABLE	DEFERRABLE	EXEC
CONTINUE	DEFERRED	EXECUTE
CONVERT	DELETE	EXISTS
CORR	DENY	EXIT
CORRESPONDING	DEPTH	EXTERNAL
COVAR_POP	DEREF	FALSE
COVAR_SAMP	DESC	FETCH
CREATE	DESCRIBE	FILE
CROSS	DESCRIPTOR	FILLFACTOR
CUBE	DESTROY	FILTER
CUME_DIST	DESTRUCTOR	FIRST
CURRENT	DETERMINISTIC	FLOAT
CURRENT_CATALOG	DIAGNOSTICS	FOR
CURRENT_DATE	DICTIONARY	FOREIGN
CURRENT_DEFAULT_TRANSFORM_GROUP	DISCONNECT	FOUND
CURRENT_PATH	DISK	FREE
CURRENT_ROLE	DISTINCT	FREETEXT
CURRENT_SCHEMA	DISTRIBUTED	FREETEXTTABLE
CURRENT_TIME	DOMAIN	FROM
CURRENT_TIMESTAMP	DOUBLE	FULL
CURRENT_TRANSFORM_GROUP_FOR_TYPE	DROP	FULLTEXTTABLE
CURRENT_USER	DUMP	FUNCTION
CURSOR	DYNAMIC	FUSION
CYCLE	EACH	GENERAL
DATA	ELEMENT	GET
DATABASE	ELSE	GLOBAL
DATE	END	GO
DAY	END-EXEC	GOTO
DBCC	EQUALS	GRANT
DEALLOCATE	ERRLVL	GROUP

Таблица П2.1 (продолжение)

GROUPING	LAST	NO
HAVING	LATERAL	NOCHECK
HOLD	LEADING	NONCLUSTERED
HOLDLOCK	LEFT	NONE
HOST	LESS	NORMALIZE
HOUR	LEVEL	NOT
IDENTITY	LIKE	NULL
IDENTITY_INSERT	LIKE_REGEX	NULLIF
IDENTITYCOL	LIMIT	NUMERIC
IF	LINENO	OBJECT
IGNORE	LN	OCCURRENCES_REGEX
IMMEDIATE	LOAD	OF
IN	LOCAL	OFF
INDEX	LOCALTIME	OFFSETS
INDICATOR	LOCALTIMESTAMP	OLD
INITIALIZE	LOCATOR	ON
INITIALLY	MAP	ONLY
INNER	MATCH	OPEN
INOUT	MEMBER	OPENDATASOURCE
INPUT	MERGE	OPENQUERY
INSERT	METHOD	OPENROWSET
INT	MINUTE	OPENXML
INTEGER	MOD	OPERATION
INTERSECT	MODIFIES	OPTION
INTERSECTION	MODIFY	OR
INTERVAL	MODULE	ORDER
INTO	MONTH	ORDINALITY
IS	MULTISET	OUT
ISOLATION	NAMES	OUTER
ITERATE	NATIONAL	OUTPUT
JOIN	NATURAL	OVER
KEY	NCHAR	OVERLAY
KILL	NCLOB	PAD
LANGUAGE	NEW	PARAMETER
LARGE	NEXT	PARAMETERS

Таблица П2.1 (продолжение)

PARTIAL	REGR_AVGY	SECTION
PARTITION	REGR_COUNT	SECURITYAUDIT
PATH	REGR_INTERCEPT	SELECT
PERCENT	REGR_R2	SENSITIVE
PERCENT_RANK	REGR_SLOPE	SEQUENCE
PERCENTILE_CONT	REGR_SXX	SESSION
PERCENTILE_DISC	REGR_SXY	SESSION_USER
PIVOT	REGR_SYY	SET
PLAN	RELATIVE	SETS
POSITION_REGEX	RELEASE	SETUSER
POSTFIX	REPLICATION	SHUTDOWN
PRECISION	RESTORE	SIMILAR
PREFIX	RESTRICT	SIZE
PREORDER	RESULT	SMALLINT
PREPARE	RETURN	SOME
PRESERVE	RETURNS	SPACE
PRIMARY	REVERT	SPECIFIC
PRINT	REVOKE	SPECIFICTYPE
PRIOR	RIGHT	SQL
PRIVILEGES	ROLE	SQLEXCEPTION
PROC	ROLLBACK	SQLSTATE
PROCEDURE	ROLLUP	SQLWARNING
PUBLIC	ROUTINE	START
RAISERROR	ROW	STATE
RANGE	ROWCOUNT	STATEMENT
READ	ROWGUIDCOL	STATIC
READS	ROWS	STATISTICS
READTEXT	RULE	STDDEV_POP
REAL	SAVE	STDDEV_SAMP
RECONFIGURE	SAVEPOINT	STRUCTURE
RECURSIVE	SCHEMA	SUBMULTISET
REF	SCOPE	SUBSTRING_REGEX
REFERENCES	SCROLL	SYMMETRIC
REFERENCING	SEARCH	SYSTEM
REGR_AVGX	SECOND	SYSTEM_USER

Таблица П2.1 (окончание)

TABLE	UNIQUE	WITHIN
TABLESAMPLE	UNKNOWN	WITHOUT
TEMPORARY	UNNEST	WORK
TERMINATE	UNPIVOT	WRITE
TEXTSIZE	UPDATE	WRITETEXT
THAN	UPDATETEXT	XMLEGG
THEN	USAGE	XMLATTRIBUTES
TIME	USE	XMLBINARY
TIMESTAMP	USER	XMLCAST
TIMEZONE_HOUR	USING	XMLCOMMENT
TIMEZONE_MINUTE	VALUE	XMLCONCAT
TO	VALUES	XMLDOCUMENT
TOP	VAR_POP	XMLEMENT
TRAILING	VAR_SAMP	XML EXISTS
TRAN	VARCHAR	XMLFOREST
TRANSACTION	VARIABLE	XMLITERATE
TRANSLATE_REGEX	VARYING	XMLNAMESPACES
TRANSLATION	VIEW	XMLPARSE
TREAT	WAITFOR	XMLPI
TRIGGER	WHEN	XMLQUERY
TRUE	WHENEVER	XMLSERIALIZE
TRUNCATE	WHERE	XMLTABLE
TSEQUAL	WHILE	XMLTEXT
UESCAPE	WIDTH_BUCKET	XMLVALIDATE
UNDER	WINDOW	YEAR
UNION	WITH	ZONE

ПРИЛОЖЕНИЕ 3

Утилита командной строки *sqlcmd*

Утилита *sqlcmd* позволяет из командной строки выполнять операторы Transact-SQL. При запуске на выполнение этой утилиты ей можно передавать параметры. Можно также вызывать утилиту, не задавая никаких параметров.

Все параметры начинаются со знака минус (-). Значение параметра задается через пробел после названия параметра. Если значение содержит пробелы, то оно заключается в кавычки ("").

Сильно сокращенный синтаксис команды в нотациях Бэкуса — Наура выглядит следующим образом:

```
sqlcmd [[-U <имя пользователя> [-P <пароль>] | -E]
[-z <новый пароль>] [-Z <новый пароль>]
[-S <имя сервера>[</имя экземпляра>]]
[-d <имя базы данных>]
[-i <исходный файл>[, <исходный файл>]...]
[-o <выходной файл>]
[-q "запрос к базе данных"]
[-Q "запрос к базе данных"]
[-v <переменная>=<значение>" [<переменная>=<значение>"]...]...
[-?]
```

Все параметры утилиты являются необязательными. Основными, наиболее часто используемыми параметрами, являются следующие.

◆ **-U <имя пользователя>**

Задает имя пользователя. Вместо задания имени пользователя можно использовать переменную окружения SQLCMDUSER.

◆ **-P <пароль>**

Задает пароль пользователя. Для пароля можно использовать переменную окружения SQLCMDPASSWORD.

◆ **-E**

Указывает, что для соединения с экземпляром сервера используется аутентификация Windows. При этом значения переменных окружения (если они заданы)

SQLCMDUSER и SQLCMDPASSWORD игнорируются. В этом случае нельзя также использовать параметры -U и -P.

- ◆ -z <новый пароль>
- ◆ -Z <новый пароль>

Задает новый пароль текущего пользователя. Если используется параметр -z, то выполнение утилиты продолжается. В командной строке можно вводить операторы. Если же указан параметр -E, то выполнение утилиты завершается.

- ◆ -S <имя сервера> [</имя экземпляра>]

Параметр задает имя сервера в сети и имя экземпляра сервера базы данных, например, -S DEVRACE\MSSQLSERVER02. Если этот параметр не указан, то утилита выбирает значения из переменной окружения SQLCMDSERVER. Если значение этой переменной не задано, то утилита использует экземпляр сервера базы данных по умолчанию на локальном компьютере.

- ◆ -d <имя базы данных>

Задает имя начальной базы данных. При указании этого параметра утилита выдает команду USE <имя базы данных>. Если параметр не задан, то используется база данных по умолчанию. Если такая база данных отсутствует, то утилита завершается с ошибкой.

- ◆ -i <исходный файл> [, <исходный файл>] ...

Задает полный путь и имя исходного файла (имена исходных файлов). Эти файлы будут использоваться в указанном порядке в качестве ввода данных для утилиты. Исходными файлами являются обычные скрипты, содержащие операторы Transact-SQL, которые должны быть выполнены утилитой. Если в пути к файлу или в имени файла присутствуют пробелы, то все значение параметра должно заключаться в кавычки. Если параметр не указан, операторы должны вводиться с клавиатуры компьютера.

- ◆ -o <выходной файл>

Задает файл, в который будут помещаться сообщения и все выходные данные утилиты. Если файл с таким именем уже существует по указанному пути, то он будет перезаписан. Если в пути к файлу или в имени файла присутствуют пробелы, то все значение параметра должно заключаться в кавычки. Если параметр не указан, выходные данные будут выводиться на монитор.

- ◆ -q "запрос к базе данных"
- ◆ -Q "запрос к базе данных"

Позволяет при запуске утилиты выполнить указанный запрос или даже несколько запросов к базе данных. Если в параметре задается более одного запроса, то для указания завершения каждого оператора должен быть использован символ точка с запятой. В операторах можно выполнять любые действия с базой данных — задавать операторы INSERT, DELETE, UPDATE, SELECT.

При задании параметра -Q утилита выполняет запрос (запросы) и завершает свою работу. Если же задан параметр -q, то после выполнения запросов утилита

продолжает оставаться активной. В командной строке утилиты можно вводить и выполнять операторы.

- ◆ [-v <переменная>="<значение>" [<переменная>="<значение>"] . . .] . . .

Позволяет задать значения переменным, которые будут использованы в последующих операторах, вводимых в командной строке утилиты, или в параметрах, используемых в файле скрипта, если исходные операторы для утилиты содержатся во внешнем файле. В одном параметре можно задать произвольное количество значений переменных. Сам параметр -v может при вызове утилиты повторяться произвольное количество раз.

Переменная в операторах задается в виде \$(<имя переменной>).

- ◆ -?

Выводит краткое описание синтаксиса всех параметров, используемых при обращении к утилите sqlcmd.

ПРИЛОЖЕНИЕ 4

Характеристики базы данных

В этом приложении описаны все характеристики пользовательских баз данных, которые вы можете использовать в вашей работе и которые можете изменять для ваших существующих баз данных. Большинство характеристик присваиваются базе данных по умолчанию при создании базы данных. Эти значения вы можете впоследствии изменить при использовании оператора изменения базы данных `ALTER DATABASE` или с помощью Management Studio. Некоторые характеристики мы с вами рассмотрим достаточно подробно, относительно других — только сообщим об их существовании. Если они вам понадобятся в вашей нелегкой программистской жизни, вы найдете необходимые сведения в Books Online или в другой подходящей литературе.

Характеристик у базы данных достаточно большое количество. Традиционно их группируют по нескольким категориям.

Не существует нормальной терминологии для названия соответствующих характеристик, свойств, опций. По этой причине я в списке параметров базы данных привожу ключевые слова, используемые в операторе `ALTER DATABASE` для изменения значения соответствующего параметра, и в скобках название, отображаемые в окне **Database Properties** на вкладке **Options** для свойств базы данных в диалоговых средствах компонента MS SQL Server Management Studio. Напомню, что диалоговое окно **Database Properties** появляется, когда в окне **Object Explorer** вы щелкаете правой кнопкой мыши по имени интересующей вас базы данных и в появившемся контекстном меню выбираете элемент **Properties**. В этом окне в левой верхней части нужно выбрать вкладку **Options**.

Пример окна **Database Properties** с текущей вкладкой **Options** для нашей базы данных `BestDatabase` сразу после ее создания со всеми характеристиками, присваиваемыми базе данных по умолчанию при ее создании, показан на рис. П4.1.

"Главные" (General), наиболее часто используемые параметры сгруппированы в следующие основные категории.

- ◆ Параметры `Auto` (или `Automatic`).
- ◆ Параметры доступности базы данных (`Availability`).

- ◆ Параметры автономной базы данных (**Containment**).
- ◆ Параметры курсора (**Cursor**).
- ◆ Параметры восстановления (**Recovery, Recovery model**).
- ◆ Общие параметры SQL.

Знание этих параметров входит в состав требований сертификационных экзаменов Microsoft.

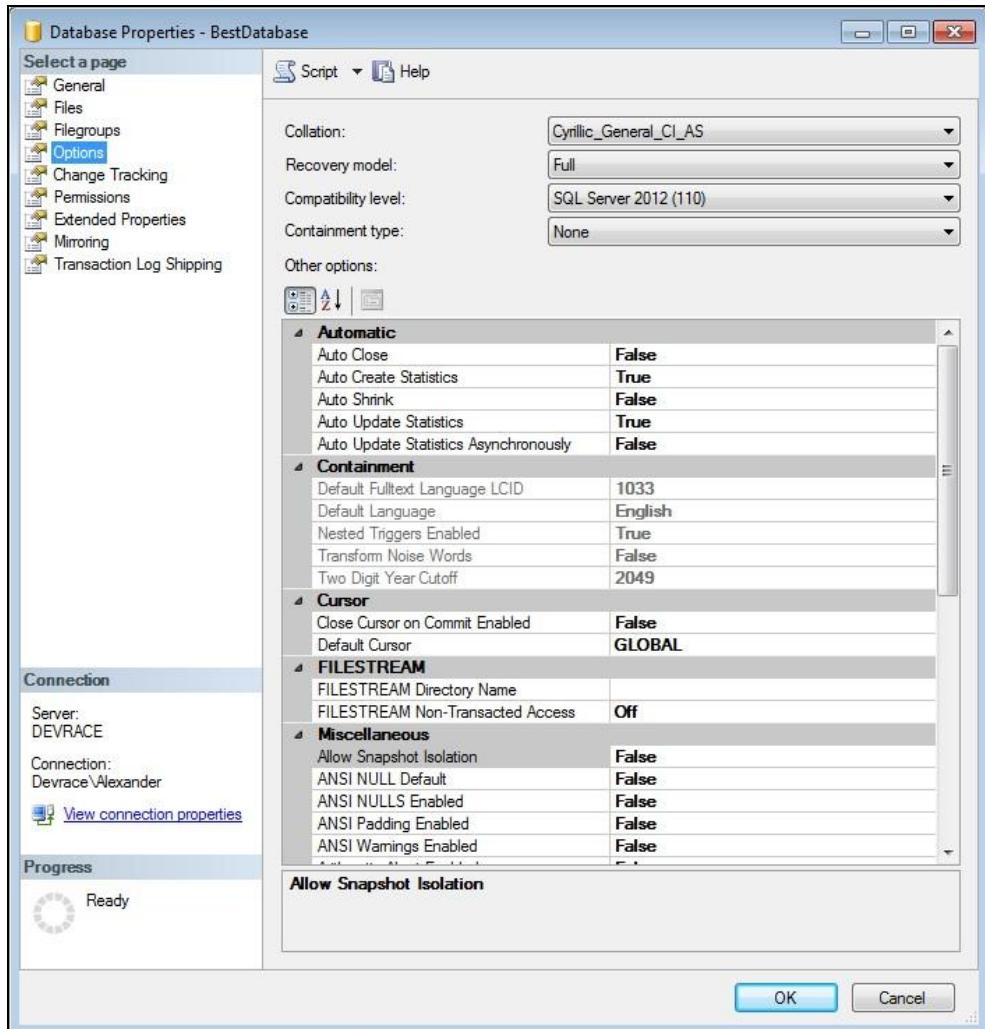


Рис. П4.1. Вкладка Options диалогового окна Database Properties в Management Studio

Существует еще несколько гораздо реже используемых параметров, знание которых не требуется для сертификационных экзаменов Microsoft, но которые когда-нибудь вам пригодятся в вашей деятельности. Их я только назову, не вдаваясь в подробности использования.

Они распределяются по следующим категориям:

- ◆ Параметры внешнего доступа (**External Access**).
- ◆ Параметры компонента Service Broker.
- ◆ Параметры изоляции мгновенных снимков (**Snapshot Isolation**).

При создании базы данных в языке Transact-SQL с помощью оператора `CREATE DATABASE` вы можете задать только значения для двух параметров — имя базы данных и порядок сортировки (`COLLATE`). Значения всех остальных параметров устанавливаются по умолчанию — выбираются из системной базы данных `model`. Имя базы данных является единственным обязательным параметром при создании новой пользовательской базы данных.

Значения по умолчанию для всех остальных параметров заданы в системной базе данных `model` и некоторые из них могут быть изменены пользователем в процессе работы с системой. Там же указан и порядок сортировки по умолчанию. Он задается во время инсталляции SQL Server. Если вы вносите изменения для значения любого из параметров в базу данных `model`, то все вновь создаваемые пользовательские базы данных будут получать новые значения по умолчанию этих измененных параметров. При описании параметров здесь указана и возможность изменения их значения по умолчанию в базе данных `model`.

Для изменения значения допустимого параметра в базе данных `model` используется оператор `ALTER DATABASE`. Например, чтобы изменить значение уровня совместимости по умолчанию с предыдущими версиями SQL Server для вновь создаваемых баз данных (параметр `COMPATIBILITY_LEVEL`), нужно выполнить следующий оператор изменения базы данных `model`:

```
ALTER DATABASE model  
SET COMPATIBILITY_LEVEL = 110;
```

Любой параметр для существующей пользовательской базы данных в дальнейшем может быть изменен в операторе Transact-SQL `ALTER DATABASE`, а некоторые параметры, но не все, можно изменить и с использованием диалоговых средств или с помощью Management Studio.

ЗАМЕЧАНИЕ

Вы, наверное, уже заметили некоторое несоответствие только что приведенной классификации параметров, которая взята мною из Books Online и некоторых других документов Microsoft и которая полностью соответствует синтаксису оператора `ALTER DATABASE`, группировке этих же параметров в Management Studio. Подобные вещи всегда существовали в программировании. К ним нужно просто спокойно относиться.

Рассмотрим параметры по их категориям.

П4.1. Параметры *Auto* (в Management Studio — группа *Automatic*)

В категории присутствуют параметры, определяющие автоматический порядок выполнения некоторых действий с базой данных.

◆ **AUTO_CLOSE** (в Management Studio — **Auto Close**). Автоматическое закрытие базы данных. Можно изменять в базе данных *model*.

- Если установлено значение **ON** (**True**), то база данных автоматически закрывается после отключения от нее последнего использующего ее пользователя (переводится в состояние **OFFLINE**). В этом состоянии базу данных можно копировать, перемещать в другое место, удалять. При последующем подключении любого пользователя к такой базе данных она автоматически открывается для нормальной с ней работы.
- Если установлено значение **OFF** (**False**), то и после отключения последнего пользователя база данных остается открытой. Это значение по умолчанию для большинства изданий SQL Server.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET AUTO_CLOSE { ON | OFF }` или при помощи выбора в Management Studio в поле **Auto Close** из раскрывающегося списка значения **False** (соответствует **OFF**) или **True** (соответствует **ON**).

◆ **AUTO_CREATE_STATISTICS** (в Management Studio — **Auto Create Statistics**). Задает или отключает автоматическое создание статистики по базе данных, требуемой для оптимизации запроса. Можно изменять в базе данных *model*.

- При значении **ON** (**True**) статистика создается автоматически. Это значение по умолчанию.
- Если задано **OFF** (**False**), то статистические данные автоматически не создаются. В этом случае создание статистики можно при необходимости выполнить вручную.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET AUTO_CREATE_STATISTICS { ON | OFF }` или при помощи выбора в Management Studio в поле **Auto Create Statistics** из раскрывающегося списка значения **False** или **True**.

◆ **AUTO_UPDATE_STATISTICS** (в Management Studio — **Auto Update Statistics**). Задает или отключает автоматическое обновление статистики по базе данных, требуемой для оптимизации запроса. Можно изменять в базе данных *model*.

- Если указано **ON** (**True**), то при необходимости выполняется автоматическое обновление статистических данных. Это значение по умолчанию.
- Если задано **OFF** (**False**), то статистические данные должны обновляться вручную.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET AUTO_UPDATE_STATISTICS { ON | OFF }` или при помощи выбора в Management

Studio в поле **Auto Update Statistics** из раскрывающегося списка значения **False** или **True**.

◆ **AUTO_UPDATE_STATISTICS_ASYNC** (в Management Studio — **Auto Update Statistics Asynchronously**). Влияет на выполнение запросов, которые требуют обновления статистики по базе данных. Можно изменять в базе данных *model*.

- Если указано **ON (True)**, то запрос, инициирующий обновление статистических данных, не будет ожидать завершения этого обновления, а будет использовать для оптимизации устаревшую статистику. Последующие же запросы будут использовать уже обновленные статистические данные.
- Если задано **OFF (False)**, то запрос, инициирующий обновление статистических данных, будет ожидать завершение этого обновления. Обновленная статистика будет использована при оптимизации этого запроса. Это значение по умолчанию.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }` или при помощи выбора в Management Studio в поле **Auto Update Statistics Asynchronously** из раскрывающегося списка значения **False** или **True**. Значение этого параметра **ON** не будет иметь эффекта, если параметр `AUTO_UPDATE_STATISTICS` установлен в **OFF**.

◆ **AUTO_SHRINK** (в Management Studio — **Auto Shrink**). Задает или отключает режим автоматического сжатия файлов базы данных. Периодически во время проверок файлы базы данных могут автоматически сжиматься системой, если в них существует неиспользуемое пространство на внешнем носителе. Можно изменять в базе данных *model*.

- **ON (True)** — файлы сжимаются автоматически. Файлы будут сжиматься, если неиспользуемое пространство превышает 25% размера файла. В результате автоматического сжатия файлов для них выделяется 25% свободного места на внешнем носителе. Файлы протокола транзакций сжимаются только в том случае, если для восстановления базы данных выбрана простая модель или создавалась резервная копия протокола транзакций. Нельзя также сжать базу данных, находящуюся в режиме только для чтения (`READ_ONLY`).
- **OFF (False)** — автоматическое сжатие файлов не происходит. Это значение по умолчанию.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET AUTO_SHRINK { ON | OFF }` или при помощи выбора в Management Studio в поле **Auto Shrink** из раскрывающегося списка значения **False** или **True**.

П4.2. Параметры доступности базы данных (*Availability*)

В категории присутствуют параметры, определяющие возможность выполнения различных действий с базой данных и ограничения на количество и характеристики одновременно подключенных к базе данных пользователей.

◆ Состояние базы данных (в Management Studio — **Database State**). Некоторые значения можно установить при использовании оператора `ALTER DATABASE`. В Management Studio изменять состояние базы данных нельзя. Нельзя также изменять и в базе данных `model`. Состояние может принимать следующие значения:

- **ONLINE** (в Management Studio это состояние отображается как `NORMAL`). База данных в доступном, оперативном, состоянии, с ней можно выполнять любые действия. Это значение по умолчанию;
- **OFFLINE**. База данных закрыта и находится в недоступном состоянии. Операции с такой базой данных выполнять невозможно, однако базу данных можно скопировать или переместить в другое место;
- **EMERGENCY**. База данных переводится в режим только для чтения. Доступ к базе данных разрешен только для членов роли сервера `sysadmin`. Такое состояние обычно используется для диагностики базы данных;
- **RESTORING**. База данных недоступна. В это состояние база данных переводится, когда происходит восстановление файлов данных базы данных;
- **RECOVERING**. База данных недоступна, она находится в процессе восстановления. После завершения восстановления база данных автоматически будет переведена в оперативное состояние (`ONLINE`);
- **RECOVERY_PENDING**. База данных недоступна. В процессе восстановления базы данных произошла ошибка, которая требует вмешательства пользователя. После исправления ошибки пользователь сам должен перевести базу данных в оперативное состояние;
- **SUSPECT**. База данных недоступна. Она помечена как подозрительная и может быть повреждена. Со стороны пользователя требуются действия по устранению ошибок.

В операторе `ALTER DATABASE` можно задать только одно из трех состояний базы данных — `ONLINE`, `OFFLINE` и `EMERGENCY`. Другие состояния устанавливаются системой при появлении конкретных условий или при выполнении некоторых функций с базой данных.

◆ Возможность изменения данных в базе данных (в Management Studio — **Database Read-Only**). Значение по умолчанию можно изменять в базе данных `model`.

- **READ_WRITE (False)**. База данных доступна для чтения и для изменения. Значение по умолчанию.
- **READ_ONLY (True)**. Из базы данных можно получать данные, но никакие изменения данных невозможны.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET { READ_WRITE | READ_ONLY }` или при помощи выбора в Management Studio в поле **Database Read-Only** из раскрывающегося списка **False** (соответствует `READ_WRITE`) или **True** (соответствует `READ_ONLY`).

◆ Допустимое количество подключаемых к базе данных пользователей (в Management Studio — **Restrict Access**). Можно изменять в базе данных model.

- MULTI_USER. Значение по умолчанию. Допускается подключение любых пользователей, имеющих соответствующие полномочия.
- SINGLE_USER. В одно и то же время к базе данных может быть подключен только один пользователь.
- RESTRICTED_USER. Количество пользователей не ограничивается, но к базе данных могут подключаться только пользователи, относящиеся к роли базы данных db_owner или к ролям сервера dbcreator и sysadmin.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET { MULTI_USER | SINGLE_USER | RESTRICTED_USER } или при выборе в Management Studio в поле **Restrict Access** из раскрывающегося списка соответствующего значения.

П4.3. Параметры автономной базы данных (*Containment*)

Определяют характеристики автономной базы данных.

◆ DEFAULT_FULLTEXT_LANGUAGE (в Management Studio — **Default Language**). Задает язык базы данных по умолчанию для полнотекстового поиска в индексированных столбцах.

◆ DEFAULT_LANGUAGE (в Management Studio — **Default Fulltext Language LCID**). Задает язык по умолчанию для вновь создаваемых регистрационных имен пользователей в виде кода языка.

◆ NESTED_TRIGGERS (в Management Studio — **Nested Triggers Enabled**). Задает возможность использования вложенных триггеров AFTER. Если указано OFF (**False**), вложенные триггеры недопустимы. При задании ON (**True**) может существовать до 32 уровней триггеров.

◆ TRANSFORM_NOISE_WORDS (в Management Studio — **Transform Noise Words**). Задает поведение сервера базы данных в ситуациях полнотекстового поиска. Это слова, которые часто присутствуют в текстах и не имеют особого смысла при выполнении поисковых действий.

- Значение OFF (по умолчанию, в Management Studio **False**) приводит к тому, что если в запросе встречаются такие слова и запрос возвращает нулевое количество строк, то просто выдается предупреждающее сообщение.
- Если указано ON (**True**), то система выполнит преобразование запроса, удалив из него соответствующие слова.

◆ TWO_DIGIT_YEAR_CUTOFF (в Management Studio — **Two Digit Year Cutoff**). Задает значение года в диапазоне между 1753 и 9999. По умолчанию 2049. Это число используется для интерпретации года, заданного двумя символами. Если двух-

символьный год меньше или равен последним двум цифрам указанного четырехсимвольного значения, то этот год будет интерпретироваться как год того же столетия. Если больше — то будет использовано столетие, следующее за указанным в операторе столетием.

П4.4. Параметры курсора (*Cursor*)

Определяют область видимости и поведение курсора.

◆ CURSOR_CLOSE_ON_COMMIT (в Management Studio — **Close Cursor on Commit Enabled**). Автоматическое закрытие курсора. Можно изменять в базе данных *model*.

- OFF (**False**). Значение по умолчанию. Когда транзакция подтверждается (*COMMIT*), курсоры остаются открытыми. При откате транзакции (*ROLLBACK*) курсоры закрываются.

- ON (**True**). При подтверждении или откате транзакции курсоры закрываются.

Значение устанавливается в операторе *ALTER DATABASE* заданием варианта *SET CURSOR_CLOSE_ON_COMMIT { ON | OFF }* или при помощи выбора в Management Studio в поле **Close Cursor on Commit Enabled** из раскрывающегося списка значения **False** или **True**.

◆ CURSOR_DEFAULT (в Management Studio — **Default Cursor**). Задает область действия курсора. Можно изменять в базе данных *model*.

- GLOBAL. Значение по умолчанию. Курсор может быть использован в любом пакете текущего соединения с базой данных.

- LOCAL. Курсор доступен и может быть использован только локально в том пакете (хранимой процедуре, триггере), где он был создан.

Значение устанавливается в операторе *ALTER DATABASE* заданием варианта *SET CURSOR_DEFAULT { GLOBAL | LOCAL }* или при помощи выбора в Management Studio в поле **Default Cursor** из раскрывающегося списка соответствующего значения.

П4.5. Параметры восстановления (*Recovery*, *Recovery model*)

Здесь присутствуют два параметра. Один (**Recovery** или **Recovery model**) действительно напрямую связан с резервным копированием и восстановлением базы данных с резервной копии и файла протокола транзакций. Другой же (**Page Verify**) определяет способ выявления поврежденных страниц в существующей базе данных

◆ RECOVERY (в Management Studio — **Recovery model**). Определяет стратегии создания резервных копий и восстановления поврежденной базы данных. Можно изменять в базе данных *model*.

- **FULL (Full)**. Значение по умолчанию. Полное восстановление базы данных с использованием резервных копий протоколов транзакций.
- **BULK_LOGGED (Bulk-logged)**. При восстановлении эффективно используются данные протокола транзакций, если для базы данных выполнялись крупномасштабные изменения.
- **SIMPLE (Simple)**. Используется простая стратегия создания резервных копий, которая позволяет минимизировать внешний объем памяти, отводимой под протокол транзакций.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET RECOVERY { FULL | BULK_LOGGED | SIMPLE }` или при помощи выбора в Management Studio в поле **Recovery model** (в верхней части вкладки **Options**) из раскрывающегося списка соответствующего значения.

- ◆ **PAGE_VERIFY (Page Verify** в Management Studio). Задает способ обнаружения поврежденных страниц базы данных. Можно изменять в базе данных `model`.
- **CHECKSUM**. Значение по умолчанию. При помещении измененной страницы базы данных на внешний носитель Database Engine рассчитывает контрольную сумму по всей странице (8 Кбайт) и помещает число в заголовок страницы. При чтении страницы контрольная сумма вычисляется заново и сравнивается с хранимой на странице. Отсутствие равенства рассчитанной и хранимой контрольной суммы свидетельствует о наличии ошибки.
 - **TORN_PAGE_DETECTION**. Для каждого сектора страницы размером 512 байт вычисляется значение одного бита и помещается в заголовок страницы. При чтении страницы с внешнего носителя это также позволяет определить отсутствие ошибок в данных.
 - **NONE**. В этом случае никаких проверок при считывании страницы с внешнего носителя не производится, правильность данных не контролируется.

ЗАМЕЧАНИЕ

Существовавший ранее параметр `TORN_PAGE_DETECTION` в настоящей версии уже не поддерживается.

Значение параметра может изменяться в операторе `ALTER DATABASE` заданием варианта `SET PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE }` или при помощи выбора в Management Studio в поле **Page Verify** из раскрывающегося списка соответствующего значения.

П4.6. Общие параметры SQL (Miscellaneous)

В этой категории присутствует больше всего параметров. Это в первую очередь те параметры, которые влияют на поведение системы при выполнении различных операций с метаданными и данными в пользовательской базе данных: значения по

умолчанию относительно пустых значений (`NULL`), результат использования пустых значений в операциях, ошибки округления и др.

◆ **ANSI_NULL_DEFAULT** (в Management Studio — **ANSI NULL Default**). Применяется для столбцов таблиц, которые основаны на типах данных alias, или для столбцов пользовательского типа CLR. Определяет значение по умолчанию: `NULL` или `NOT NULL`. Можно изменять в базе данных `model`.

- **ON (True)**. Значение по умолчанию `NULL`.
- **OFF (False)**. Установлено по умолчанию. Значением соответствующего столбца по умолчанию не может быть `NULL`.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET ANSI_NULL_DEFAULT { ON | OFF }` или при помощи выбора в Management Studio в поле **ANSI NULL Default** из раскрывающегося списка значения **False** или **True**.

◆ **ANSI_NULLS** (в Management Studio — **ANSI NULLS Enabled**). Задает соответствие стандарту ANSI для реляционных баз данных относительно сравнения любых значений с пустым значением `NULL`. Можно изменять в базе данных `model`.

- **ON (True)**. Соответствует стандарту ANSI: любое сравнение со значением `NULL` дает результат `UNKNOWN`.
- **OFF (False)**. Значение по умолчанию. При сравнении строковых данных не в Юникоде, если обе сравниваемые величины имеют значение `NULL`, то результатом будет `TRUE`.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET ANSI_NULLS { ON | OFF }` или при помощи выбора в Management Studio в поле **ANSI NULLS Enabled** из раскрывающегося списка значения **False** или **True**.

◆ **ANSI_PADDING** (в Management Studio — **ANSI Padding Enabled**). Влияет на добавление конечных пробелов в столбцы типов данных `VARCHAR` и `NVARCHAR`, а также на конечные нули в двоичных значениях `VARBINARY`. Применяется при добавлении новых строк в таблицы базы данных. Измененное значение этого параметра влияет только на определение столбцов, создаваемых после изменения данного параметра. Можно изменять в базе данных `model`.

- **OFF (False)**. Значение по умолчанию. Конечные пробелы (типы данных `VARCHAR` и `NVARCHAR`) и конечные нули (тип данных `VARBINARY`) отбрасываются при добавлении новых данных в таблицу.
- **ON (True)**. Столбцы `CHAR` и `BINARY`, допускающие значение `NULL`, подгоняются под длину столбца путем добавления конечных пробелов или нулей.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET ANSI_PADDING { ON | OFF }` или при помощи выбора в Management Studio в поле **ANSI Padding Enabled** из раскрывающегося списка значения **False** или **True**.

◆ **ANSI_WARNINGS (ANSI Warnings Enabled)**. Влияет на появление предупреждающих сообщений или сообщений об ошибке при возникновении таких ситуаций,

как появление значения `NULL` в агрегатных (статистических) функциях или при делении на ноль. Можно изменять в базе данных `model`.

- **ON (True)**. Предупреждающее сообщение или сообщение об ошибке в таких ситуациях выдается.
- **OFF (False)**. Значение по умолчанию. Предупреждающие сообщения не выдаются. При делении на ноль возвращается `NULL`.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET ANSI_WARNINGS { ON | OFF }` или при помощи выбора в Management Studio в поле **ANSI Warnings Enabled** из раскрывающегося списка значения **False** или **True**.

◆ **ARITHABORT** (в Management Studio — **Arithmetic Abort Enabled**). Определяет реакцию системы на арифметическое переполнение или при делении на ноль. Возможен вариант прекращения выполнения запроса или выдача сообщения и продолжение выполнения запроса. Можно изменять в базе данных `model`.

- **ON (True)**. При арифметическом переполнении или делении числа на ноль выполнение запроса прекращается.
- **OFF (False)**. Значение по умолчанию. Сообщение не создается. При делении на ноль возвращается значение `NULL`.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET ARITHABORT { ON | OFF }` или при помощи выбора в Management Studio в поле **Arithmetic Abort Enabled** из раскрывающегося списка значения **False** или **True**.

◆ **COMPATIBILITY_LEVEL** (в Management Studio — **Compatibility Level**). Определяет уровень совместимости с предыдущими версиями SQL Server. Можно изменять в базе данных `model`.

- **90 (SQL Server 2005 (90))**. Совместимость с версией SQL Server 2005.
- **100 (SQL Server 2008 (100))**. Совместимость с версией SQL Server 2008.
- **110 (SQL Server 2012 (110))**. Совместимость с версией SQL Server 2012.

Значение устанавливается в операторе `ALTER DATABASE` заданием варианта `SET COMPATIBILITY_LEVEL = { 90 | 100 | 110 }` или при помощи выбора в Management Studio в поле **Compatibility Level** (в верхней части вкладки **Options**) из раскрывающегося списка соответствующего значения.

◆ **CONCAT_NULL_YIELDS_NULL (Concatenate Null Yields Null)**. Определяет результат конкатенации (соединения) двух строковых данных, когда одно из строковых значений имеет пустое значение `NULL`. Можно изменять в базе данных `model`.

- **ON (True)**. Операция конкатенации вернет пустое значение `NULL`, если любой из операторов имеет значение `NULL`.
- **OFF (False)**. Значение по умолчанию. Пустое значение `NULL` рассматривается как нулевая строка (строка с нулевым количеством символов).

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET CONCAT_NULL_YIELDS_NULL { ON | OFF } или при помощи выбора в Management Studio в поле **Concatenate Null Yields Null** из раскрывающегося списка значения **False** или **True**.

◆ DATE_CORRELATION_OPTIMIZATION (**Date Correlation Optimization Enabled**). Определяет поддержание статистики между таблицами, связанными ограничением FOREIGN KEY и содержащими столбцы типа данных datetime. Можно изменять в базе данных model.

- ON (**True**). Поддерживается статистика корреляции.
- OFF (**False**). Значение по умолчанию. Статистика корреляции не поддерживается.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET DATE_CORRELATION_OPTIMIZATION { ON | OFF } или при помощи выбора в Management Studio в поле **Date Correlation Optimization Enabled** из раскрывающегося списка значения **False** или **True**.

◆ NUMERIC_ROUNDABORT (**Numeric Round-Abort**). Задает возможность появления ошибки при потере точности в процессе вычисления числового значения. Можно изменять в базе данных model.

- ON (**True**). Если при вычислении значения происходит потеря точности, то выдается сообщение об ошибке.
- OFF (**False**). Значение по умолчанию. Сообщение об ошибке не выдается, а значение округляется с точностью того элемента, для которого сохраняется результат.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET NUMERIC_ROUNDABORT { ON | OFF } или при помощи выбора в Management Studio в поле **Numeric Round-Abort** из раскрывающегося списка значения **False** или **True**.

◆ PARAMETERIZATION (**Parameterization**). Условие выполнения параметризации запросов. Можно изменять в базе данных model.

- SIMPLE (**Simple**). Параметризация основывается на поведении базы данных по умолчанию.
- FORCED (**Forced**). Выполняется параметризация всех запросов в базе данных.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET PARAMETERIZATION { SIMPLE | FORCED } или при помощи выбора в Management Studio в поле **Parameterization** из раскрывающегося списка соответствующего значения.

◆ QUOTED_IDENTIFIER (**Quoted Identifiers Enabled**). Определяет допустимость использования кавычек ("двойных кавычек") при задании идентификаторов с разделителями. Можно изменять в базе данных model.

- ON (**True**). Для идентификаторов с разделителями наряду с квадратными скобками можно использовать и кавычки.

- OFF (**False**). Значение по умолчанию. Идентификаторы с разделителями могут заключаться только в квадратные скобки в соответствии со всеми правилами языка Transact-SQL.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET QUOTED_IDENTIFIER { ON | OFF } или при помощи выбора в Management Studio в поле **Quoted Identifiers Enabled** из раскрывающегося списка значения **False** или **True**.

- ◆ RECURSIVE_TRIGGERS (**Recursive Triggers Enabled**). Определяет допустимость срабатывания рекурсивных триггеров AFTER. Можно изменять в базе данных model.

- ON (**True**). Допустимо срабатывание рекурсивных триггеров AFTER.
- OFF (**False**). Значение по умолчанию. Триггеры AFTER рекурсивно не выполняются.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET RECURSIVE_TRIGGERS { ON | OFF } или при помощи выбора в Management Studio в поле **Recursive Triggers Enabled** из раскрывающегося списка значения **False** или **True**.

П4.7. Параметры внешнего доступа (*External Access*)

Параметры управляют доступом к базе данных со стороны других, внешних, ресурсов, например, из другой базы данных.

- ◆ DB_CHAINING (**Cross-database Ownership Chaining Enabled**). Задает, может ли база данных находиться в межбазовой цепочке владения или быть источником в такой цепочке. В базе данных model изменить это значение нельзя.

- ON (**True**). База данных может присутствовать в цепочке владения.
- OFF (**False**). Значение по умолчанию. База данных в цепочке владения присутствовать не может.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET DB_CHAINING { ON | OFF }. В Management Studio изменять это значение нельзя.

- ◆ TRUSTWORTHY (**Trustworthy**). Определяет, могут ли модули базы данных получать доступ к ресурсам вне базы данных. В базе данных model изменить это значение нельзя. Параметр устанавливается в значение OFF при каждом соединении с базой данных.

- ON (**True**). Модули могут обращаться к другим ресурсам.
- OFF (**False**). Значение по умолчанию. Обращение к другим ресурсам невозможно.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET TRUSTWORTHY { ON | OFF }. В Management Studio изменять это значение нельзя.

П4.8. Параметры компонента Service Broker

Управляют поведением компонента Service Broker. В базе данных model изменить это значение нельзя.

- ◆ ENABLE_BROKER (**Broker Enabled**, значение **True**). Включает компонент Service Broker, запускает доставку сообщений. В базе данных сохраняется значение идентификатора компонента Service Broker. Значение по умолчанию.
- ◆ DISABLE_BROKER (**Broker Enabled**, значение **False**). Отключает компонент Service Broker. Отключается доставка сообщений. В базе данных сохраняется значение идентификатора компонента Service Broker.
- ◆ NEW_BROKER (**Service Broker Identifier**, в Management Studio изменить нельзя). База данных получает новый идентификатор брокера.
- ◆ ERROR_BROKER_CONVERSATIONS. Включается доставка сообщений. В базе данных сохраняется значение идентификатора компонента Service Broker. Диалоги в базе данных завершаются с ошибками.
- ◆ HONOR_BROKER_PRIORITY { ON | OFF } (**Honor Broker Priority**, в Management Studio изменить нельзя). При задании ON операции Send выполняются с учетом приоритетов, присвоенных диалогам. В случае OFF (значение по умолчанию) операции Send выполняются в ситуации, когда все диалоги имеют значение по умолчанию.

В Management Studio могут быть установлены только значения для параметров ENABLE_BROKER (группа Broker Enabled, значение **True**) и DISABLE_BROKER (Broker Enabled, значение **False**). Значения остальных параметров этой группы могут устанавливаться только в операторе ALTER DATABASE.

П4.9. Параметры изоляции транзакций для мгновенных снимков (SNAPSHOT)

Эти два параметра определяют допустимость и порядок использования уровня изоляции транзакций SNAPSHOT. Просмотреть существующие установки можно только при использовании системного представления отображения каталогов sys.databases, изменить значения можно при использовании этого системного представления и в операторе ALTER DATABASE. В Management Studio просмотр и изменение значений этих параметров невозможны.

- ◆ ALLOW_SNAPSHOT_ISOLATION. Определяет допустимость использования уровня изоляции транзакции SNAPSHOT. Изменить значение этого параметра нельзя, если база данных находится в состоянии OFFLINE.
- ON. Для базы данных допустимо использование уровня изоляции транзакции SNAPSHOT.

- OFF. Значение по умолчанию. Использование уровня изоляции транзакций SNAPSHOT невозможно.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET ALLOW_SNAPSHOT_ISOLATION { ON | OFF }.

- ◆ READ_COMMITTED_SNAPSHOT. Определяет поведение транзакции с уровнем изоляции READ COMMITTED. Изменить значение этого параметра нельзя, если база данных находится в состоянии OFFLINE.

- ON. Для транзакции с уровнем изоляции транзакции READ COMMITTED используется вместо блокировки управление версиями строк.
- OFF. Значение по умолчанию. Для транзакции с уровнем изоляции READ COMMITTED будут использоваться блокировки для конкурентных процессов.

ПРИЛОЖЕНИЕ 5

Языки, представленные в SQL Server

Система поддерживает множество языков. Чтобы просмотреть этот список, нужно обратиться к системному представлению `sys.syslanguages`.

Представление содержит следующие столбцы (в списке пропущены лишь два столбца, которые нам с вами не потребуются):

- ◆ `langid` — внутренний идентификатор языка;
- ◆ `dateformat` — формат представления даты:
 - `dmy` день-месяц-год;
 - `mdy` месяц-день-год;
 - `ymd` ГОД-месяц-день;
- ◆ `datefirst` — день недели, отображаемый первым: 1 — понедельник, 7 — воскресенье;
- ◆ `name` — официальное название языка в кодировке Юникод;
- ◆ `alias` — альтернативное название языка (алиас, псевдоним). Например, если официальное название русского языка представлено словом "русский", то альтернативное — словом "Russian". Названия также представлены в кодировке Юникод;
- ◆ `months` — перечисляются на соответствующем языке полные названия месяцев с января по декабрь. Элементы списка разделяются запятыми;
- ◆ `shortmonths` — краткие названия месяцев также разделенные запятыми;
- ◆ `days` — на соответствующем языке перечисляются дни недели с понедельника по воскресенье;
- ◆ `lcid` — код языка в Microsoft Windows.

Для вызова системного представления нужно выполнить:

```
USE master;
SELECT langid,          -- Внутренний идентификатор языка
       dateformat,      -- Формат представления даты
       datefirst,       -- День недели, отображаемый первым
```

```

name,          -- Официальное название языка (алиас, псевдоним)
alias,         -- Альтернативное название языка
months,        -- Полные названия месяцев
shortmonths,   -- Краткие названия месяцев
days,          -- Дни недели
lcid           -- Код языка
FROM sys.syslanguages;

```

Список присутствующих в системе языков представлен в табл. П5.1.

Таблица П5.1. Список языков, поддерживаемых SQL Server

ID	Формат даты	День недели	Офиц. название	Алиас	Месяцы (полно)	Месяцы (кратко)	Дни недели	lcid
0	mdy	7	us_english	English	January, February, March, April, May, June, July, August, September, October, November, December	Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec	Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday	1033
1	dmy	1	Deutsch	German	Januar, Februar, März, April, Mai, Juni, Juli, August, September, Oktober, November, Dezember	Jan, Feb, Mär, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez	Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag	1031
2	dmy	1	Français	French	janvier, février, mars, avril, mai, juin, juillet, août, septembre, octobre, novembre, décembre	janv, févr, mars, avr, mai, juin, juil, août, sept, oct, nov, déc	lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche	1036
3	ymd	7	日本語	Japanese	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12	月曜日, 火曜日, 水曜日, 木曜日, 金曜日, 土曜日, 日曜日	1041
4	dmy	1	Dansk	Danish	januar, februar, marts, april, maj, juni, juli, august, september, oktober, november, december	jan, feb, mar, apr, maj, jun, jul, aug, sep, okt, nov, dec	mandag, tirsdag, onsdag, torsdag, fredag, lørdag, søndag	1030

Таблица П5.1 (продолжение)

ID	Формат даты	День недели	Офиц. название	Алиас	Месяцы (полно)	Месяцы (кратко)	Дни недели	Icid
5	dmy	1	Español	Spanish	Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio, Agosto, Septiembre, Octubre, Noviembre, Diciembre	Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic	Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo	3082
6	dmy	1	Italiano	Italian	gennaio, febbraio, marzo, aprile, maggio, giugno, luglio, agosto, settembre, ottobre, novembre, dicembre	gen, feb, mar, apr, mag, giu, lug, ago, set, ott, nov, dic	lunedì, martedì, mercoledì, giovedì, venerdì, sabato, domenica	1040
7	dmy	1	Nederlands	Dutch	januari, februari, maart, april, mei, juni, juli, augustus, september, oktober, november, december	jan, feb, mrt, apr, mei, jun, jul, aug, sep, okt, nov, dec	maandag, dinsdag, woensdag, donderdag, vrijdag, zaterdag, zondag	1043
8	dmy	1	Norsk	Norwegian	januar, februar, mars, april, mai, juni, juli, august, september, oktober, november, desember	jan, feb, mar, apr, mai, jun, jul, aug, sep, okt, nov, des	mandag, tirsdag, onsdag, torsdag, fredag, lørdag, søndag	2068
9	dmy	7	Português	Portuguese	janeiro, fevereiro, março, abril, maio, junho, julho, agosto, setembro, outubro, novembro, dezembro	jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez	segunda-feira, terça-feira, quarta-feira, quinta-feira, sexta-feira, sábado, domingo	2070

Таблица П5.1 (продолжение)

ID	Формат даты	День недели	Офиц. название	Алиас	Месяцы (полно)	Месяцы (кратко)	Дни недели	Icid
10	dmy	1	Suomi	Finnish	tammikuuta, helmikuuta, maaliskuuta, huhtikuuta, toukokuuta, kesäkuuta, heinäkuuta, elokuuta, syyskuuta, lokakuuta, marraskuuta, joulukuuta	tammi, helmi, maalis, huhti, touko, kesä, heinä, elo, syys, loka, marras, joulu	maanantai, tiistai, keskiviikko, torstai, perjantai, lauantai, sunnuntai	1035
11	ymd	1	Svenska	Swedish	januari, februari, mars, april, maj, juni, juli, augusti, september, oktober, november, december	jan, feb, mar, apr, maj, jun, jul, aug, sep, okt, nov, dec	måndag, tisdag, onsdag, torsdag, fredag, lördag, söndag	1053
12	dmy	1	čeština	Czech	leden, únor, březen, duben, květen, červen, červenec, srpen, září, říjen, listopad, prosinec	I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII	pondělí, úterý, středa, čtvrtok, pátek, sobota, neděle	1029
13	ymd	1	magyar	Hungarian	január, február, március, április, május, június, július, augusztus, szeptember, október, november, december	jan, febr, márc, ápr, máj, jún, jól, aug, szep, okt, nov, dec	hétfő, kedd, szerda, csütörtök, péntek, szombat, vasárnap	1038
14	dmy	1	polski	Polish	styczeń, luty, marzec, kwiecień, maj, czerwiec, lipiec, sierpień, wrzesień, październik, listopad, grudzień	I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII	poniedziałek, wtorek, środa, czwartek, piątek, sobota, niedziela	1045

Таблица П5.1 (продолжение)

ID	Формат даты	День недели	Офиц. название	Алиас	Месяцы (полно)	Месяцы (кратко)	Дни недели	Icid
15	dmy	1	română	Romanian	ianuarie, februarie, martie, aprilie, mai, iunie, iulie, august, septembrie, octombrie, noiembrie, decembrie	Ian, Feb, Mar, Apr, Mai, Iun, Iul, Aug, Sep, Oct, Nov, Dec	luni, marți, miercuri, joi, vineri, sâmbătă, duminică	1048
16	ymd	1	hrvatski	Croatian	siječanj, veljača, ožujak, travanj, svibanj, lipanj, srpanj, kolovoz, rujan, listopad, studeni, prosinac	sij, vel, ožu, tra, svi, lip, srp, kol, ruj, lis, stu, pro	ponedeljak, utorak, srijeda, četvrtak, petak, subota, nedjelja	1050
17	dmy	1	slovenčina	Slovak	január, február, marec, apríl, máj, jún, júl, august, september, október, november, december	I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII	pondelok, utorok, streda, štvrtok, piatok, sobota, nedel'a	1051
18	dmy	1	slovenski	Slovenian	januar, februar, marec, april, maj, junij, julij, avgust, september, oktober, november, december	jan, feb, mar, apr, maj, jun, jul, avg, sept, okt, nov, dec	ponedeljek, torek, sreda, četrtek, petek, sobota, nedelja	1060
19	dmy	1	ελληνικά	Greek	Ιανουαρίου, Φεβρουαρίου, Μαρτίου, Απριλίου, Μαΐου, Ιουνίου, Ιουλίου, Αυγούστου, Σεπτεμβρίου, Οκτωβρίου, Νοεμβρίου, Δεκεμβρίου	Ιαν, Φεβ, Μαρ, Απρ, Μαΐ, Ιουν, Ιουλ, Αυγ, Σεπτ, Οκτ, Νοε, Δεκ	Δευτέρα, Τρίτη, Τετάρτη, Πέμπτη, Παρασκευή, Σάββατο, Κυριακή	1032

Таблица П5.1 (продолжение)

ID	Формат даты	День недели	Офиц. название	Алиас	Месяцы (полно)	Месяцы (кратко)	Дни недели	Icid
20	dmy	1	български	Bulgarian	януари, февруари, март, апвил, май, юни, юли, август, септември, октомври, ноември, декември	януари, февруари, март, април, май, юни, юли, август, септември, октомври, ноември, декември	понеделник, вторник, сряда, четвъвътък, петък, събота, неделя	1026
21	dmy	1	русский	Russian	Январь, Февраль, Март, Апрель, Май, Июнь, Июль, Август, Сентябрь, Октябрь, Ноябрь, Декабрь	янв, фев, мар, апр, май, июн, июл, авг, сен, окт, ноя, дек	понедельник, вторник, среда, четверг, пятница, суббота, воскресенье	1049
22	dmy	1	Türkçe	Turkish	Ocak, Şubat, Mart, Nisan, Mayıs, Haziran, Temmuz, Ağustos, Eylül, Ekim, Kasım, Aralık	Oca, Şubat, Mar, Nis, May, Haz, Tem, Ağu, Eyl, Eki, Kas, Ara	Pazartesi, Salı, Çarşamba, Perşembe, Cuma, Cumartesi, Pazar	1055
23	dmy	1	British	British English	January, February, March, April, May, June, July, August, September, October, November, December	Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec	Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday	2057
24	dmy	1	eesti	Estonian	jaanuar, veebruar, märts, aprill, mai, juuni, juuli, august, september, oktoober, november, detsember	jaan, veebr, märts, apr, mai, juuni, juuli, aug, sept, okt, nov, dets	esmaspäev, teisipäev, kolmapäev, neljapäev, reede, laupäev, pühapäev	1061

Таблица П5.1 (продолжение)

ID	Формат даты	День недели	Офиц. название	Алиас	Месяцы (полно)	Месяцы (кратко)	Дни недели	Icid
25	ymd	1	latviešu	Latvian	janvāris, februāris, marts, aprīlis, maijs, jūnijs, jūlijjs, augusts, septembris, oktobris, novembris, decembris	jan, feb, mar, apr, mai, jūn, jūl, aug, sep, okt, nov, dec	pirmadiena, otrdiena, trešdiena, ceturtdiena, piektdiena, sestdiena, svētdiena	1062
26	ymd	1	lietuvių	Lithuanian	sausis, vasaris, kovas, balandis, gegužė, birželis, liepa, rugpjūtis, rugsėjis, spalis, lapkritis, gruodis	sau, vas, kov, bal, geg, bir, lie, rgp, rgs, spl, lap, grd	pirmadienis, antradienis, trečiadienis, ketvirtadienis, penktadienis, šeštadienis, sekmadienis	1063
27	dmy	7	Português (Brasil)	Brazilian	Janeiro, Fevereiro, Março, Abril, Maio, Junho, Julho, Agosto, Setembro, Outubro, Novembro, Dezembro	Jan, Fev, Mar, Abr, Mai, Jun, Jul, Ago, Set, Out, Nov, Dez	Segunda-Feira, Terça-Feira, Quarta-Feira, Quinta-Feira, Sexta-Feira, Sábado, Domingo	1046
28	ymd	7	繁體中文	Traditional Chinese	一月, 二月, 三月, 四月, 五月, 六月, 七月, 八月, 九月, 十月, 十一月, 十二月	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12	星期一, 星期二, 星期三, 星期四, 星期五, 星期六, 星期日	1028
29	ymd	7	한국어	Korean	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12	월요일, 화요일, 수요일, 목요일, 금요일, 토요일, 일요일	1042
30	ymd	7	简体中文	Simplified Chinese	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12	01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12	星期一, 星期二, 星期三, 星期四, 星期五, 星期六, 星期日	2052

Таблица П5.1 (окончание)

ID	Формат даты	День недели	Офиц. название	Алиас	Месяцы (полно)	Месяцы (кратко)	Дни недели	Icid
31	dmy	1	Arabic	Arabic	Muharram, Safar, Rabie I, Rabie II, Jumada I, Jumada II, Rajab, Shaaban, Ramadan, Shawwal, Thou Alqadah, Thou Alhajja	Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec	Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday	1025
32	dmy	7	ไทย	Thai	มกราคม, กุมภาพันธ์, มีนาคม, เมษายน, พฤษภาคม, มิถุนายน, กรกฎาคม, สิงหาคม, กันยายน, ตุลาคม, พฤษจิกายน, ธันวาคม	ม.ค., ก.พ., มี.ค., เม.ย., พ.ค., มี.ย., ก.ค., ส.ค., ก.ย., ต.ค., พ.ย., ธ.ค.	จันทร์, อังคาร, พุธ, พฤหัสบดี, ศุกร์, เสาร์, อาทิตย์	1054
33	dmy	1	norsk (bokmål)	Bokmål	januar, februar, mars, april, mai, juni, juli, august, september, oktober, november, desember	jan, feb, mar, apr, mai, jun, jul, aug, sep, okt, nov, des	mandag, tirsdag, onsdag, torsdag, fredag, lørdag, søndag	1044

ПРИЛОЖЕНИЕ 6

Описание электронного архива

Электронный архив к книге расположен на FTP-сервере издательства по адресу <ftp://ftp.bhv.ru/9785977505017.zip>. Эта ссылка доступна и со страницы книги на сайте www.bhv.ru.

Архив содержит скрипты создания базы данных и ее объектов, помещения данных в таблицы. Структура архива показана в табл. П6.1.

Таблица П6.1. Структура электронного архива

Файл	Описание
01-CreateDatabaseAndTables.sql	Создание базы данных, пользовательских типов данных и таблиц
02-InsertCountriesAndRegions.sql	Добавление в базу данных стран и регионов России, штатов США, графств Великобритании
03-InsertAreaRus.sql	Добавление районов России
04-InsertAreaUSA.sql	Добавление районов США
05-InsertActive.sql	Добавление видов деятельности
06-InsertGoods.sql	Добавление товаров
07-InsertOrgTypeProp.sql	Добавление организационно-правовых форм, видов организаций и форм собственности
08-InsertSpec.sql	Добавление специальностей
09-InsertAdmLaw.sql	Добавление административных и уголовных правонарушений
10-InsertPeople.sql	Добавление людей, включая сведения о родственных связях
11-InsertOrgStaff.sql	Добавление организаций, сотрудников организаций и правонарушений людей

Предметный указатель

B

BLOB 28

C

Common Table Expression (CTE) 445

D

DCL 56

DDL 56

DML 56

G

GUID 252

M

Management Studio 113

Microsoft Distributed Transaction Coordinator
(MS DTC) 520

O

OLAP 45

OLE 448

OLTP 45

S

SQL Server 2012

◊ инсталляция 9

SQL Server Enterprise 60

SQL Server Express 60

T

TCL 56

U

User Defined Function (UDF) 34, 537

А

- Автоинкрементные столбцы 297
 Автономные базы данных (contained) 150
 Агрегатные функции 193
 Алиас (alias) 218, 495

Б

- База данных:
- ◊ владелец 70
 - ◊ добавление нового файла 132
 - ◊ изменение имени 128
 - ◊ изменение характеристик файла 133
 - ◊ концептуальная (содержательная) модель 46
 - ◊ копирование и восстановление 170
 - ◊ логическая модель 46
 - ◊ отсоединение 161
 - ◊ переименование 141
 - ◊ переименование файла 133
 - ◊ пользовательская 66
 - ◊ присоединение 156, 159
 - ◊ проектирование 45
 - концептуальный уровень 46
 - логический уровень 46
 - физический уровень 46
 - ◊ системная 66
 - ◊ удаление существующего файла 133
 - ◊ характеристики 70
- Блок операторов 524

В

- Ветвление в программе 525
 Владелец базы данных 70
 Внешний ключ 32, 303
 Вторичные файлы 69
 Выборка данных 469
- ◊ использование операторов сравнения 485, 488, 489, 491
 - ◊ функция:
 - ALL 491
 - ANY 492
 - EXISTS 493
 - SINGULAR 493
- Вычисляемые столбцы 307

Г

- Группировка результатов выборки 500
 Группы событий 544

Д

- Двоичное число 51
 Двоичные типы данных 224
 Декларативная целостность 33
 Денормализация таблиц 45
 Диалоговое окно:
 - ◊ Database Properties 565
 - ◊ свойств сервера 62
 Дизьюнкция 29
 Добавление данных:
 - ◊ оператор INSERT 446
 Добавление, изменение или удаление строк таблицы:
 - ◊ оператор MERGE 460
 Дополнительное условие 463

З

- Задание даты и времени 213
 Задание уровня изоляции транзакции:
 - ◊ оператор SET TRANSACTION ISOLATION LEVEL 520
 Задачи оперативной обработки транзакций 45
 Запуск и останов экземпляра сервера 60
 Запуск распределенной транзакции:
 - ◊ оператор BEGIN DISTRIBUTED TRANSACTION 520
 Запуск сервера:
 - ◊ из командной строки 61
 - ◊ из программы Configuration Manager 61
 - ◊ с помощью программы Management Studio 62
 Запуск транзакции:
 - ◊ оператор BEGIN TRANSACTION 518
 Значение NULL 30

И

- Идентификатор 52
 ◊ обычный 52
 ◊ с разделителями 53
- Изменение:
 - ◊ базы данных:
 - диалоговые средства Management Studio 141
 - оператор ALTER DATABASE 128
 - ◊ данных:
 - оператор UPDATE 453
 - ◊ индекса:
 - оператор ALTER INDEX 435
 - с помощью Management Studio 443

Изменение (*прод.*):

- ◊ представления:
 - оператор ALTER VIEW 511

- ◊ схемы базы данных:
 - ALTER SCHEMA 166

◊ таблицы:

- оператор ALTER TABLE 371
- с помощью Management Studio 378

◊ триггера:

- оператор ALTER TRIGGER 545

◊ функции, определенной пользователем:

- оператор ALTER FUNCTION 539

◊ хранимой процедуры:

- оператор ALTER PROCEDURE 530

Индексы 31, 78, 302, 409

◊ columnstore 409

◊ XML 409

◊ кластерные 31, 302, 414

- XML 424

◊ некластеризованные 302, 419

◊ обычные 409, 411

◊ покрывающие 410

◊ пространственные 409

◊ реляционные 409, 411

◊ уникальные 414

◊ фильтрованные 415

Искусственный первичный ключ 36

K

Кластерный индекс 414

Ключи в таблицах 31

◊ внешний ключ 32

◊ задание первичных ключей 35

◊ первичный ключ 31

◊ уникальный ключ 32

Коллекция схем XML 269

Команда:

◊ net start 61

◊ net stop 65

◊ SET DATEFORMAT 213

Комментарии 51, 524

Конкатенация 29

Конструкция TRY/CATCH 526

Конъюнкция 29

Координатор распределенных транзакций 520

Копирование и восстановление баз данных 170

◊ диалоговые средства Management Studio 172

◊ оператор BACKUP 171

◊ оператор RESTORE 171

Курсор 254

Куча 31

Л

Логарифмические функции 195

Логическая функция:

◊ CHOOSE() 183

◊ IIF() 183

Локальные переменные 180, 524

◊ объявление 180

Ломаная линия 231

М

Мгновенные снимки 162

Метаданные 27

Метка 526

Н

Набор:

◊ данных 28

◊ символов 30

Неопределенные значения 551

Нетипизированный элемент 266

◊ XML 266

Нормализация таблиц 41

◊ вторая нормальная форма 43

◊ нормальная форма Бойса — Кодда 44

◊ первая нормальная форма 41

◊ пятая нормальная форма 44

◊ третья нормальная форма 43

◊ четвертая нормальная форма 44

Нормальные формы 41

О

Обобщенное табличное выражение 445

Ограничение:

◊ CHECK 33, 306

◊ DEFAULT 33

◊ внешнего ключа 303

◊ первичного ключа 299

◊ таблицы 33, 298

◊ уникального ключа 299

Одиночные события 544

Окно регистрации нового сервера 64

Оперативный анализ данных 45

- Оператор:
- ◊ ALTER 57
 - ◊ ALTER DATABASE 90, 127, 128, 565
 - ◊ ALTER FUNCTION 539
 - ◊ ALTER INDEX 435
 - ◊ ALTER PARTITION FUNCTION 327
 - ◊ ALTER PARTITION SCHEME 325
 - ◊ ALTER PROCEDURE 530
 - ◊ ALTER SCHEMA 166
 - ◊ ALTER TABLE 371
 - ◊ ALTER TRIGGER 545
 - ◊ ALTER VIEW 511
 - ◊ ALTER XML SCHEMA COLLECTION 270
 - ◊ BACKUP 171
 - ◊ BEGIN DISTRIBUTED TRANSACTION 57, 520
 - ◊ BEGIN TRANSACTION 57, 517, 518
 - ◊ COMMIT TRANSACTION 57, 518
 - ◊ COMMIT WORK 57
 - ◊ CREATE 56
 - ◊ CREATE COLUMNSTORE INDEX 421
 - ◊ CREATE DATABASE 80
 - предложение COLLATE 84
 - предложение CONTAINMENT 83
 - предложение LOG ON 84
 - предложение ON 83
 - предложение WITH 84
 - ◊ CREATE FUNCTION 538
 - ◊ CREATE INDEX 411
 - ◊ CREATE PARTITION FUNCTION 325
 - ◊ CREATE PARTITION SCHEME 323
 - ◊ CREATE PROCEDURE 528
 - ◊ CREATE SCHEMA 165
 - ◊ CREATE SPATIAL INDEX 428
 - ◊ CREATE TABLE 290
 - ◊ CREATE TRIGGER 542
 - ◊ CREATE TYPE 276
 - ◊ CREATE VIEW 510
 - ◊ CREATE XML INDEX 422
 - ◊ CREATE XML SCHEMA COLLECTION 269
 - ◊ DECLARE 180, 524
 - ◊ DECLARE CURSOR 254
 - ◊ DELETE 57, 457
 - ◊ DENY 57
 - ◊ DROP 57
 - ◊ DROP DATABASE 90
 - ◊ DROP INDEX 433
 - ◊ DROP FUNCTION 540
 - ◊ DROP PARTITION FUNCTION 326
 - ◊ DROP PARTITION SCHEME 325
 - ◊ DROP PROCEDURE 531
 - ◊ DROP SCHEMA 166
 - ◊ DROP TABLE 368
 - ◊ DROP TRIGGER 547
 - ◊ DROP TYPE 286
 - ◊ DROP VIEW 512
 - ◊ DROP XML SCHEMA COLLECTION 270
 - ◊ EXCEPT 481, 506
 - ◊ EXECUTE 79, 532
 - ◊ FETCH 255
 - ◊ GO 95
 - ◊ GOTO 526
 - ◊ GRANT 57
 - ◊ IF 525
 - ◊ INSERT 57, 446
 - ◊ INTERSECT 481, 506
 - ◊ IS NULL 30
 - ◊ MERGE 460
 - ◊ RESTORE 171
 - ◊ RETURN 526
 - ◊ REVOKE 57
 - ◊ ROLLBACK TRANSACTION 57, 519
 - ◊ ROLLBACK WORK 57
 - ◊ SAVE TRANSACTION 57, 519
 - ◊ SELECT 28, 57, 469
 - ◊ SET 181, 524
 - ◊ SET IDENTITY_INSERT 297
 - ◊ SET LANGUAGE 218
 - ◊ SET TRANSACTION ISOLATION LEVEL 520
 - ◊ THROW 528
 - ◊ TRUNCATE TABLE 459
 - ◊ UNION 480, 505
 - ◊ UPDATE 57, 453
 - ◊ WHILE 525
- Операции сравнения 182, 477
- Определение зависимостей таблицы 364
- Организация циклов:
- ◊ оператор WHILE 525
- Останов сервера 65
- ◊ из командной строки 65
 - ◊ из программы Configuration Manager 65
 - ◊ из программы Management Studio 65
- Отмена (откат) транзакции:
- ◊ оператор ROLLBACK TRANSACTION 519
- Отношения между таблицами 33, 38—40

Отрицание 29

Отсоединение базы данных 161

П

Первичный файл 69

Подтверждение транзакции:

◊ оператор COMMIT TRANSACTION 518

Полигон 225, 235

Пользовательские базы данных 66

Пользовательские типы данных 276

Порядок сортировки 29, 30, 70

Правила Кодда 37, 551

Предикат 475

Представление 33, 509

◊ изменение 511

◊ изменяемое 34

◊ индексированное 413

◊ неизменяемое 34

◊ создание 510

◊ удаление 512

Привилегии 35

Принципалы 164

Присоединение базы данных 156

Программа SQL Server Configuration Manager 61

Пространственные типы данных 225

Процедура хранимая 528

Р

Разреженные столбцы 298

Рекурсия 34

Реляционная система управления базами данных (РСУБД) 27

Реляционная база данных 27

Роль 35

С

Самосоединение таблиц 498

Связанный сервер 448

Секционированные таблицы 322

Семантика 47

Символьные типы данных 197

Символьный репертуар 29

Синтаксис 47

◊ описания файловой группы 90

◊ спецификации файла 86

▫ предложение FILEGROWTH 89

▫ предложение FILENAME 87

▫ предложение MAXSIZE 89

▫ предложение NAME 87

▫ предложение SIZE 88

Системная процедура:

◊ sp_changedbowner 70

◊ хранимая 79

Системная функция:

◊ DB_ID() 79, 100

◊ DB_NAME() 79

◊ FILE_ID() 79

◊ FILE_NAME() 79, 101

◊ FILEGROUP_ID() 79

◊ FILEGROUP_NAME() 80

Системная хранимая процедура 79

◊ sp_databases 79

Системное представление:

◊ sys.columns 79

◊ sys.database_files 75

◊ sys.database_permissions 78

◊ sys.database_principals 78

◊ sys.database_role_members 78

◊ sys.databases 73

◊ sys.events 78

◊ sys.filegroups 77

◊ sys.indexes 78

◊ sys.master_files 74

◊ sys.schemas 78

◊ sys.tables 78

◊ sys.types 79

◊ sys.views 78

Системные базы данных 66

◊ база данных master 67

◊ база данных model 67

◊ база данных msdb 67

◊ база данных resource 68

◊ база данных tempdb 68

Системные функции 79

Скалярные подзапросы 478

Событие 78

События базы данных 35

Соединение таблиц 462, 493

◊ внутреннее 500

◊ двойное 497

◊ левое внешнее 493

◊ полное внешнее 496

◊ правое внешнее 496

◊ реинтерабельное 498

◊ рефлексивное 498

Создание:

◊ базы данных 80

▫ диалоговые средства Management Studio 123

▫ оператор CREATE DATABASE 80

Создание (прод.):

- ◊ индекса:
 - XML 422
 - оператор CREATE INDEX 411
 - с помощью диалоговых средств Management Studio 438
- ◊ индекса columnstore:
 - оператор CREATE COLUMNSTORE INDEX 421
- ◊ мгновенных снимков базы данных 162
- ◊ определенной пользователем функции:
 - оператор CREATE FUNCTION 538
- ◊ представления:
 - оператор CREATE VIEW 510
 - с помощью Management Studio 515
- ◊ пространственного индекса:
 - оператор CREATE SPATIAL INDEX 428
- ◊ псевдонима 280
- ◊ схемы базы данных:
 - оператор CREATE SCHEMA 165
- ◊ таблицы:
 - оператор CREATE TABLE 290
 - с помощью Management Studio 342
- ◊ точки сохранения транзакции:
 - оператор SAVE TRANSACTION 519
- ◊ хранимой процедуры:
 - оператор CREATE PROCEDURE 528

Состояние:

- ◊ базы данных 71
 - EMERGENCY 71
 - OFFLINE 71
 - ONLINE 71
 - RECOVERING 71
 - RECOVERY_PENDING 71
 - RESTORING 71
 - SUSPECT 71
- ◊ файлов базы данных 72
 - DEFUNCT 72
 - OFFLINE 72
 - ONLINE 72
 - RECOVERY_PENDING 72
 - RESTORING 72
 - SUSPECT 72

Столбец таблицы:

- ◊ автоинкрементный 36
- ◊ вычисляемый 283
- ◊ имя 281
- ◊ ограничения 282, 298
- Строковые:
 - ◊ константы 51
 - ◊ функции 200

Сущность 41

- Схемы базы данных 164
- ◊ изменение 166
- ◊ создание 165
- ◊ удаление 166

Т

Таблица 28, 289

- ◊ главная 32
- ◊ дочерняя 32
- ◊ записи 28
- ◊ изменение характеристик 371
- ◊ ограничения 289, 298
- ◊ параметры 293
- ◊ подчиненная 32
- ◊ поля 28
- ◊ родительская 32
- ◊ секционированная 322
- ◊ столбцы 28
- ◊ строка 28
- ◊ файловая 292

Таблицы:

- ◊ соединение 493
- ◊ истинности 475

Текущий экземпляр сервера 60

Тесселяция 432

Тип данных:

- ◊ BIT 183
- ◊ CURSOR 254
- ◊ GEOGRAPHY 238
- ◊ GEOMETRY 226
- ◊ HIERARCHYID 247
- ◊ SQL_VARIANT 243
- ◊ TABLE 261
- ◊ UNIQUEIDENTIFIER 252
- ◊ XML 262

Типизированный элемент 266

- ◊ XML 269

Типы данных 28, 178

- ◊ дата и время 178, 212
- ◊ двоичные 224

◊ двоичные строки 179

◊ пользовательские 276

◊ пространственные 179, 225

◊ символьные 178

◊ числовые 178, 181

Точка 226

Транзакция 37, 517

- ◊ уровни изоляции 520

- Транслитерация 29
- Триггер 34, 541
 - ◊ изменение 545
 - ◊ использование 547
 - ◊ создание 542
 - ◊ удаление 547
- Тригонометрические функции 193

У

- Удаление:
 - ◊ базы данных:
 - диалоговые средства Management Studio 150
 - оператор DROP DATABASE 90
 - ◊ данных:
 - оператор DELETE 457
 - ◊ индекса:
 - диалоговые средства Management Studio 443
 - оператор DROP INDEX 433
 - ◊ представления:
 - оператор DROP VIEW 512
 - ◊ строк таблицы:
 - оператор TRUNCATE TABLE 459
 - ◊ схемы базы данных:
 - оператор DROP SCHEMA 166
 - ◊ таблицы 364
 - оператор DROP TABLE 368
 - с помощью Management Studio 368
 - ◊ триггера:
 - оператор DROP TRIGGER 547
 - ◊ функций:
 - оператор DROP FUNCTION 540
 - ◊ хранимой процедуры:
 - оператор DROP PROCEDURE 531

Улучшенное инвертированное отношение 239

Уникальный глобальный идентификатор (GUID) 252

Уникальный ключ 32

Упорядочение результата:

- ◊ предложение ORDER BY 484

Уровни изоляции транзакции 520

Условие выборки 474

Утилита sqlcmd 91, 102, 561

Ф

Файловые:

- ◊ группы 90, 108

- ◊ потоки 358, 360
- ◊ таблицы 405
- Функция:
 - ◊ CAST() 94
 - ◊ COLLATIONPROPERTY() 131
 - ◊ CURRENT_TIMESTAMP 217
 - ◊ DATEADD() 224
 - ◊ DATEFROMPARTS() 214
 - ◊ DATENAME() 219
 - ◊ DATEPART() 219
 - ◊ DATETIME2FROMPARTS() 215
 - ◊ DATETIMEFROMPARTS() 215
 - ◊ DATETIMEOFFSETFROMPARTS() 223
 - ◊ EOMONTH() 224
 - ◊ ERROR_LINE() 527
 - ◊ ERROR_MESSAGE() 527
 - ◊ ERROR_NUMBER() 527
 - ◊ ERROR_PROCEDURE() 527
 - ◊ ERROR_SEVERITY() 527
 - ◊ ERROR_STATE() 527
 - ◊ GETDATE() 217
 - ◊ GETUTCDATE() 217
 - ◊ ISNULL() 30
 - ◊ NEWSEQUENTIALID() 253
 - ◊ ROLLUP() 504
 - ◊ SMALLDATETIMEFROMPARTS() 224
 - ◊ SQL_VARIANT_PROPERTY() 244
 - ◊ SYSDATETIME() 217
 - ◊ SYSDATETIMEOFFSET() 217
 - ◊ SYSUTCDATETIME() 217
 - ◊ TIMEFROMPARTS() 217
 - ◊ агрегатная:
 - AVG() 193
 - MAX() 193
 - MIN() 193
 - STDEV() 193
 - STDEVP() 193
 - SUM() 193
 - VAR() 193
 - VARP() 193
 - ◊ даты и времени 217
 - ◊ логарифмическая:
 - EXP() 195
 - LOG() 195
 - LOG10() 195
 - POWER() 195
 - SQRT() 195
 - SQUARE() 195
 - ◊ математическая:
 - ABS() 196
 - CEILING() 196

- ◊ математическая (*прод.*):
 - FLOOR() 196
 - RAND() 196
 - ROUND() 196
 - SIGN() 196
- ◊ определенная пользователем 34, 537
- ◊ строковая 200
 - ASCII() 204
 - CHAR() 204
 - CHARINDEX() 208
 - DATALENGTH() 200
 - LEFT() 202
 - LEN() 201
 - LOWER() 207
 - LTRIM() 203
 - NCHAR() 204
 - PATINDEX() 210
 - QUOTENAME() 207
 - REPLACE() 208
 - REPLICATE() 208
 - REVERSE() 208
 - RIGHT() 202
 - RTRIM() 203
 - STR() 204
 - SUBSTRING() 202
 - UNICODE() 204
 - UPPER() 207
- ◊ тригонометрическая:
 - ACOS() 193
 - ASIN() 193
 - ATAN() 193
 - ATN2() 193
 - COS() 193
 - COT() 193
 - DEGREES() 194
 - RADIANS() 194

- SIN() 193
- TAN() 193

X

Характеристики:

- ◊ базы данных 70
- ◊ файлов базы данных 72

Хранимые процедуры 34, 528

Ц

Целое:

- ◊ со знаком 50
- ◊ число 49

Ч

Число с плавающей точкой 50

Числовые типы данных 181

Ш

Шаблонные символы 477

Я

Язык:

- ◊ Transact-SQL 27, 46
 - синтаксис 47
- ◊ манипулирования данными (DML) 56
- ◊ определения данных (DDL) 56
- ◊ управления доступом к данным (DCL) 56
- ◊ управления транзакциями (TCL) 56
- ◊ хранимых процедур и триггеров 56, 524